A Multifactor Empirical Framework for Explaining Power and Energy Variation in Managed Runtime Systems: The Java Runtime Environment Case

VOLUME I OF II

Foundations, Methodology and Main Effects



Jonathan Lambert

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare, Ireland

Supervisors: Professor Rosemary Monahan and Dr Kevin Casey
Heads of Department:
Dr Joseph Timoney (until 2024)
Dr Aidan Mooney (from 2024)
Internal Examiner: Dr Stephen Brown
External Examiner: Professor David Gregg, Trinity College Dublin

This thesis has been prepared in accordance with the PhD regulations of Maynooth University and is subject to copyright. For more information see PhD Regulations (December 2022).

TABLE OF CONTENTS

VOLUME I: Foundations, Methodology and Main Effects

List of Figures x

List of Tables xix

Preface — Volume I xxxi

Abstract xxxii

Declaration xxxiv

Acknowledgements xxxv

1 Introduction 1

1.1 Motivation . . . 3
1.2 Terminology and Nomenclature . . . 7
1.3 Aims and Objectives . . . 14
1.4 Research Questions . . . 16
1.5 Thesis Contributions . . . 22
1.5.1 Publications and Research Outputs . . . 27
1.5.2 Overall Contribution . . . 29
1.6 Thesis Organisation . . . 30

2 Background 34

2.1 The Java Language . . . 35
2.2 The Java Runtime Environment . . . 37
2.2.1 The Class Loader Subsystem . . . 39
Class Loading . . . 40
Class Linking . . . 41
Class Initialisation . . . 46
2.2.2 The Runtime Data Areas . . . 47
The Method Area . . . 47
The Heap . . . 48
The Stacks . . . 48

The Native Method Stacks 49
2.2.3 The JVM Execution Engine 50
The JVM Interpretive Environment 51
The JVM C1 and C2 Compilers 52
2.2.4 Garbage Collection 55
2.2.5 Standardised Specification, Implementation Flexibility . . 60
2.2.6 OpenJDK Enhancements Across Versions 9–14 61
2.3 JVM Benchmark Suites 65
2.4 Power Use and Energy Consumption Profiling 69

3 Related Work **73**
3.1 JVM Power and Energy Modelling 74
3.1.1 Limitations of Existing Literature 82
3.2 Build Toolchain Effects 83
3.2.1 Limitations of Existing Literature 87
3.3 Execution Mode 88
3.3.1 Interpretive Execution 88
3.3.2 Just-in-time Execution 91
3.3.3 Limitations of Existing Literature 93
3.4 JVM Garbage Collection 94
3.4.1 Limitations of Existing Literature 98
3.5 JVM Workload Modelling 99
3.5.1 Limitations of Existing Literature 110
3.6 JVM Microarchitectural Modelling 111
3.6.1 Branch miss-rates 113
3.6.2 Data Cache Miss-Rates 118
3.6.3 Limitations of Existing Literature 119
3.7 ARM-Based Microarchitectures 121
3.8 Related Work Gaps and Contributions 123

4 Methodology **127**
4.1 Introduction 127
4.2 Theoretical Framework 128
4.3 Conceptual Framework 130
4.3.1 Study Variables 132
4.4 Experimental Framework 134
4.5 Experimental Platform 136
4.5.1 Hardware 140
Raspberry Pi4 Model B Cluster 140
Storage Platform Standardisation 144

DC Energy Analyser . . . 144
Network Infrastructure . . . 145
4.5.2 Software . . . 146
Operating System Environment . . . 146
OpenJDK Runtime Environment Platform . . . 147
Joulescope Driver and Data Acquisition Interface . . . 148
Experimental Software Artefacts . . . 149
4.5.3 Measurement Apparatus . . . 150
Measurement Synchronisation and Data Acquisition . . . 152
4.6 Construction of JRE JVM Instances . . . 153
4.6.1 Construction of GNU GCC Versions . . . 154
4.6.2 Construction of JRE JVM Versions . . . 156
4.6.3 Runtime Configuration Factors . . . 160
4.7 Benchmark Workloads . . . 162
4.7.1 Renaissance Benchmark Suite . . . 163
4.7.2 Workload Category Factor . . . 166
4.8 Data Collection Procedure . . . 168
4.8.1 Benchmark Execution Procedure . . . 169
4.8.2 Microarchitectural Performance Data Collection . . . 173
Linux perf Configuration . . . 174
4.8.3 Power Use and Energy Consumption Measurement Procedure 178
Verification of Tiered Compilation Activity . . . 181
4.8.4 Experimental Reliability Assessment . . . 182
Measurement Reliability: JRE JVM Build Reproducibility Assessment . . . 182
Measurement Reliability: Temporal Stability Assessment . 183
Measurement Reliability: Single-Platform Execution Stability Assessment . . . 184
Measurement Reliability: Cross-Platform Execution Stability Assessment . . . 185
4.9 Dataset Construction . . . 186
4.9.1 Power Use and Energy Consumption Data Capture . . . 186
4.9.2 Microarchitectural Performance Event Counter Data Capture 188
4.9.3 Dataset Overview . . . 192
4.9.4 Dataset Summary . . . 195
4.10 Statistical Analysis Framework . . . 196
4.10.1 Descriptive Statistical Analysis . . . 198
4.10.2 Inferential Statistical Analysis . . . 199
Distribution Assessment . . . 199

Kruskal–Wallis H Tests . . . 199
Mann–Whitney U Tests . . . 200
Null-Hypothesis Significance Testing Framework . . . 201
Effect Magnitude Estimation . . . 203
4.10.3 Analysis Supporting Research Question 1 . . . 204
4.10.4 Analysis Supporting Research Question 2 . . . 205
Interaction Effects Analysis . . . 206
4.11 Empirical Modelling Framework . . . 207
4.11.1 The General Framework . . . 209
4.11.2 Dummy Coding of Regression Predictors . . . 210
4.11.3 Model Specification . . . 211
4.11.4 Hierarchical Regression Procedure . . . 213
4.11.5 Model Diagnostics . . . 214
4.11.6 Bootstrap Estimation of Regression Coefficients and Standard Errors . . . 217
4.11.7 Analysis Supporting Research Question 3 . . . 220
4.12 Threats to Validity . . . 222
4.12.1 Internal Validity . . . 222
4.12.2 External Validity . . . 224
4.12.3 Construct Validity . . . 225
4.12.4 Conclusion Validity . . . 227
4.13 Chapter Summary . . . 228

5 Research Question 1: Main Effects 230
5.1 How this chapter addresses RQ1 . . . 231
5.1.1 Statistical Assumption Assessment . . . 232
5.2 OpenJDK JRE Version Effects . . . 232
5.2.1 Power Use . . . 234
Descriptive Analysis . . . 235
Inferential Statistical Analysis . . . 237
Summary . . . 239
5.2.2 Energy Consumption . . . 240
Descriptive Analysis . . . 241
Inferential Statistical Analysis . . . 244
Summary . . . 245
5.2.3 Interpretive Execution Relative to Tiered Execution . . . 246
Power Use Ratios . . . 247
Energy Consumption Ratios . . . 248
Relationship Between Power and Energy Ratios . . . 250

Summary . . . 250
5.2.4 Summary . . . 251
5.3 Build Toolchain Compiler Effects . . . 253
5.3.1 Power Use . . . 254
Descriptive Analysis . . . 255
Inferential Statistical Analysis . . . 258
Summary . . . 259
5.3.2 Energy Consumption . . . 261
Descriptive Analysis . . . 261
Inferential Statistical Analysis . . . 264
Summary . . . 266
5.3.3 Interpretive execution relative to tiered execution . . . 267
Power Use Ratios . . . 268
Energy Consumption Ratios . . . 269
Summary . . . 271
5.3.4 Summary . . . 272
5.4 Garbage Collector Variant Effects . . . 273
5.4.1 Power Use . . . 275
Descriptive Analysis . . . 276
Inferential Statistical Analysis . . . 277
Summary . . . 279
5.4.2 Energy Consumption . . . 281
Descriptive Analysis . . . 281
Inferential Statistical Analysis . . . 283
Summary . . . 284
5.4.3 Interpretive Execution Relative to Tiered Execution . . . 286
Power Use Ratios . . . 286
Energy Consumption Ratios . . . 288
Summary . . . 289
5.4.4 Summary . . . 290
5.5 Workload Effects . . . 292
5.5.1 Power Use . . . 294
Descriptive Analysis . . . 295
Inferential Statistical Analysis . . . 297
Summary . . . 298
5.5.2 Energy Consumption . . . 299
Descriptive Analysis . . . 300
Inferential Statistical Analysis . . . 303
Summary . . . 305

5.5.3 Interpretive Execution Relative to Tiered Execution 306
Energy Consumption Ratios 309
Summary 311
5.5.4 Summary 312
5.6 Microarchitectural Performance Event Relationships 314
5.6.1 Branch Misprediction Relationships 316
JRE Effects 316
Branch Misprediction and Power Use Relationships 321
Branch Misprediction and Energy Consumption Relationships322
Summary 324
5.6.2 L1 Data Cache Load Miss Rate Relationships 325
JRE Effects 325
L1 Data Cache Load Miss Rate and Power Use Relationships330
L1 Data Cache Load Miss Rate and Energy Consumption Relationships 332
5.6.3 L2 Data Cache Load Miss Rate Relationships 336
JRE Effects 336
L2 Data Cache Load Miss Rate and Power Use Relationships339
L2 Data Cache Load Miss Rate and Energy Consumption Relationships 341
5.6.4 Page Fault Relationships 345
JRE Effects 345
Page Fault Rate and Power Use Relationships 349
Page Fault Rate and Energy Consumption Relationships . 351
5.6.5 Summary 355
5.7 How this Chapter Answers RQ1 360

References — Volume I **364**

VOLUME II: Interaction Effects, Model Development and Synthesis

List of Figures **v**

List of Tables **xiv**

Preface — Volume II **xxvi**

6 Research Question 2: Interaction Effects **1**
6.1 How this Chapter Addresses RQ2 2
6.2 OpenJDK JRE Version and Workload Interaction Effects 3

6.2.1 Power Use . . . 4
Descriptive Analysis . . . 5
Inferential Statistical Analysis . . . 9
Summary . . . 10
6.2.2 Energy Consumption . . . 12
Descriptive Analysis . . . 13
Inferential Statistical Analysis . . . 17
Summary . . . 19
6.2.3 Summary . . . 22
6.3 GNU GCC version and workload interaction effects . . . 25
6.3.1 Power Use . . . 26
Descriptive Analysis . . . 26
Inferential Statistical Analysis . . . 31
Summary . . . 34
6.3.2 Energy Consumption . . . 36
Descriptive Analysis . . . 37
Inferential Statistical Analysis . . . 42
Summary . . . 44
6.3.3 Summary . . . 46
6.4 Garbage collector variant and workload interaction effects . . . 48
6.4.1 Power Use . . . 50
Descriptive Analysis . . . 50
Inferential Statistical Analysis . . . 56
Summary . . . 59
6.4.2 Energy Consumption . . . 61
Descriptive Analysis . . . 62
Inferential Statistical Analysis . . . 67
Summary . . . 71
6.4.3 Summary . . . 73
6.5 How this Chapter Answers RQ2 . . . 75

7 Research Question 3: Model Development 79
7.1 How this Chapter Addresses RQ3 . . . 80
7.2 Power Use Model . . . 81
7.2.1 Theory-Driven Model Specification . . . 82
7.2.2 Hierarchical Regression Results . . . 83
7.2.3 Regression Diagnostics . . . 87
Assessment of Linearity . . . 87
Assessment of Residual Distribution . . . 89

Assessment of Influential Observations 91
Assessment of Homoskedasticity 92
Assessment of Multicollinearity 93
Sensitivity Analysis 94
Bootstrap Resampling Analysis 97
7.2.4 Summary 99
7.3 Power Use Model: Interpretive Execution 102
7.3.1 Hierarchical Regression Results 102
7.3.2 Regression Diagnostics 107
Sensitivity Analysis 109
Bootstrap Validation 110
7.3.3 Summary 113
7.4 Power Use Model: Tiered Execution 115
7.4.1 Hierarchical Regression Results 116
7.4.2 Regression Diagnostics 122
Sensitivity Analysis 124
Bootstrap Validation 126
7.4.3 Summary 129
7.5 Power Use Cross-Model Comparative Analysis 133
7.5.1 Comparative Variance Decomposition Analysis 134
7.5.2 Observed-versus-Fitted Model Correspondence 136
7.5.3 Comparative Coefficient Restructuring Analysis 138
7.6 Energy Consumption Model 140
7.6.1 Theory-Driven Model Specification 142
7.6.2 Hierarchical Regression Results 143
7.6.3 Regression Diagnostics 149
Assessment of Linearity 149
Assessment of Residual Distribution 151
Assessment of Influential Observations 153
Assessment of Homoskedasticity 155
Assessment of Multicollinearity 157
7.6.4 Sensitivity Analysis 158
7.6.5 Bootstrap Validation 161
7.6.6 Summary 164
7.7 Energy Consumption Model: Interpretive Execution 167
7.7.1 Hierarchical Regression Results 168
7.7.2 Regression Diagnostics 173
Sensitivity Analysis 174
Bootstrap Validation 177

7.7.3 Summary . . . 180
7.8 Energy Consumption Model: Tiered Execution . . . 182
7.8.1 Hierarchical Regression Results . . . 183
7.8.2 Regression Diagnostics . . . 189
Sensitivity Analysis . . . 190
Bootstrap Validation . . . 194
7.8.3 Summary . . . 197
7.9 Energy Consumption Cross-Model Comparative Analysis . . . 199
7.9.1 Comparative Variance Decomposition Analysis . . . 200
7.9.2 Observed-versus-Fitted Model Correspondence . . . 202
7.9.3 Comparative Coefficient Restructuring Analysis . . . 204
7.10 How this Chapter Answers RQ3 . . . 207

8 Discussion **211**
8.1 Revisiting the Research Questions . . . 212
8.2 Runtime Configuration Effects Are Real but Highly Uneven . . . 214
8.3 Workload Category Condition Runtime Relationships . . . 221
8.4 Execution Mode Restructures Software–Hardware Relationships . . . 227
8.5 Microarchitectural Behaviour Reveals Hidden Runtime Characteristics 233
8.6 Interpreting the OpenJDK 13 Behaviour . . . 239
8.7 Hierarchical Modelling Reveals Context-Dependent Relationships . . . 247
8.8 Distinct Explanatory Structures Governing Power Use and Energy Consumption . . . 252
8.9 Implications for the Analysis of Managed Runtime Systems . . . 257
8.10 Practical Implications . . . 261
8.11 Limitations . . . 266
8.12 Future Research . . . 272

9 Conclusion **281**
9.1 Conclusions in Relation to the Research Questions . . . 282
9.2 Overall Contribution of the Research . . . 283
9.3 Concluding Perspective . . . 285

A Statistical Analysis Code **287**

A Raspberry Pi 4 Model B Configuration File **289**

References — Volume II **292**

List of Figures

1.1 Structural overview of the dissertation showing the sequential progression from foundational theory and experimental methodology through empirical analysis addressing research questions, culminating in discussion and overall research conclusions. 32

2.1 Schematic representation of the JRE/JVM runtime architecture, distinguishing components internal to the runtime environment from external application artefacts and native libraries. The figure illustrates the principal JVM subsystems, including class loading, runtime data areas, the execution engine, and the Java Native Interface (JNI), together with their interactions with external class files and native method libraries. 38

2.2 Thread Structures of the Garbage Collector Variants 57

4.1 Theoretical Framework showing the high-level relationship between primary parameter groups and their collective influence on modelled outcomes. 129

4.2 Conceptual framework illustrating the key architectural, virtual machine, compiler, memory management, and workload parameters considered in the modelling of system power use and energy consumption, along with associated analytical methods. 131

4.3 Experimental workflow illustrating the construction of JRE JVM instances, benchmark execution, collection of power use, energy consumption, and microarchitectural measurements, and the subsequent statistical analyses used to address RQ1, RQ2, and RQ3. . 135

4.4 Raspberry Pi4 Model B board peripheral device layout. Detailing position of Broadcom BCM2711 processor, RAM, USB Type-C power supply, and network ports. Source: [1] 143
4.5 Jetperch LLC JS110 Joulescope DC Analyzer. Source: [2] 145
4.6 Jetperch LLC JS110 Joulescope depicting the rear-panel binding posts. Source: [2] 145
4.7 TP-Link TL-SG116 16-Port Gigabit Ethernet Switch Used to Interconnect the Experimental Cluster Nodes. Source: [3] 146
4.8 Schematic of the experimental measurement apparatus showing the Joulescope positioned in-line between the mains power supply and the Raspberry Pi 4 Model B under test. The master Raspberry Pi records measurement data from the Joulescope, while the TP-Link TL-SG116 switch provides network connectivity between the experimental cluster nodes. 151
4.9 Representative snapshot of the final analytical dataset comprising runtime configuration factors, workload descriptors, power use and energy consumption observations, and microarchitectural performance measures. Each observation represents a unique JRE JVM instance and benchmark workload used in the subsequent analyses. The full dataset is available through the accompanying research data repository [4]. 191
4.10 Overview of the statistical analysis framework employed to address RQ1 and RQ2, and to support the empirical modelling activities associated with RQ3. 197

5.1 Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE power use distributions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b). 236

5.2 Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across all measurements pooled (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b). 242
5.3 Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting power use ratios for interpretive execution relative to tiered execution across all JRE JVM version (a), energy consumption ratios for interpretive execution relative to tiered execution (b), and a scatterplot illustrating the association between the corresponding power use and energy consumption ratios (c). 247
5.4 Power Use Distributions: Box-and-whisker plots depicting JRE JVM power use distributions across GNU GCC versions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b). 256
5.5 Energy Consumption Distributions: Box-and-whisker plots depicting JRE JVM energy consumption distributions across GNU GCC versions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b). 262
5.6 Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution across all GCC versions (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b). 268
5.7 Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE average power use performance across garbage collector implementations, all measurements pooled (a), and clustered based on tiered (HS) and interpretive (INT) modes of execution (b). . . 276

5.8 Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across garbage collector implementations, all measurements pooled (a), and clustered based on tiered (HS) and interpretive (INT) modes of execution (b). . . 282
5.9 Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution across all garbage collector variants (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b). 287
5.10 Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE average power use across workload categories, all measurements pooled (a), and grouped based on tiered (HS) and interpretive (INT) modes of execution (b). 295
5.11 Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across workload categories, all measurements pooled (a), and grouped based on tiered (HS) and interpretive (INT) modes of execution (b). 301
5.12 Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b). 307
5.13 Branch misprediction rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b). . 316
5.14 Branch Misprediction and Power Use Relationship: Scatterplot depicting the relationship between observed branch misprediction rates and measured JRE JVM power use across all pooled observations. 321

5.15 Branch Misprediction and Energy Consumption Relationship: Scatterplot depicting the relationship between observed branch misprediction rates and measured JRE JVM energy consumption across all pooled observations. 322
5.16 L1 Data Cache Load Miss Rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b). . 326
5.17 L1 Data Cache Load Miss Rate and Power Use Relationship: Scatterplot depicting the relationship between observed L1 data cache load miss rates and measured JRE JVM power use across all pooled observations. 330
5.18 L1 Data Cache Load Miss Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed L1 data cache miss rates and measured JRE JVM energy consumption across all pooled observations. 332
5.19 L2 Data Cache Load Miss Rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b). . 336
5.20 L2 Data Cache Load Miss Rate and Power Use Relationship: Scatterplot depicting the relationship between observed L2 data cache load miss rates and measured JRE JVM power use across all pooled observations. 340
5.21 L2 Data Cache Load Miss Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed L2 data cache load miss rates and measured JRE JVM energy consumption across all pooled observations. 342
5.22 Page Fault Rate and Power Use Relationship: Scatterplot depicting the relationship between observed page fault rates and measured JRE JVM power use across all pooled observations. 349

5.23 Page Fault Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed page fault rates and measured JRE JVM energy consumption across all pooled observations. 352

6.1 Power Use Distributions: Box-and-whisker plots detailing power use distributions, panelled by workload category, and clustered within each workload category by OpenJDK JRE version. 5

6.2 Power Use Distributions: Box-and-whisker plots depicting power use distributions for each workload category, clustered within by the OpenJDK JRE version. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. 6

6.3 Energy Consumption Distributions: Box-and-whisker plots detailing energy consumption distributions, panelled by workload category, and clustered within each workload category by OpenJDK JRE version. 13

6.4 Energy Consumption Distributions: Box-and-whisker plots depicting power use distributions for each workload category, clustered within by the OpenJDK JRE version. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. 14

6.5 Power Use Distributions: Box-and-whisker plots detailing power use distributions, panelled by workload category, and clustered within each workload category by GNU GCC version. 27

6.6 Power Use Distributions: Box-and-whisker plots depicting power use distributions for each workload category, clustered within by the GNU GCC version. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. 28

6.7 Energy Consumption Distributions: Box-and-whisker plots detailing energy consumption distributions, panelled by workload category, and clustered within each workload category by GNU GCC version. 37
6.8 Energy Consumption Distributions: Box-and-whisker plots depicting energy consumption distributions for each workload category, clustered by GNU GCC version. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. . . . 38
6.9 Power Use Distributions: Box-and-whisker plots detailing power use distributions, panelled by workload category, and clustered within each workload category by Garbage Collector variant. . . . 50
6.10 Power Use Distributions: Box-and-whisker plots depicting power use distributions for each workload category, clustered within by the Garbage Collector variant. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. . . . 51
6.11 Energy Consumption Distributions: Box-and-whisker plots detailing energy consumption distributions, panelled by workload category, and clustered within each workload category by garbage collection variant. 62
6.12 Energy Consumption Distributions: Box-and-whisker plots depicting energy consumption distributions for each workload category, clustered by the garbage collector used for heap management. Results presented for both tiered execution, panel HS, and interpretive execution, panel INT. 63

7.1 Linearity Diagnostics (Pooled Power Use Model): Residuals versus fitted values plot used to assess the adequacy of the linear regression specification. The majority of observations remained broadly centred around zero, although modest systematic residual structure was observed together with two observations exhibiting substantially elevated positive residuals. These observations indicate the presence of heterogeneous execution contexts not fully captured within the aggregated pooled model specification. 88

7.2 Residual Distribution Diagnostics (Pooled Power Use Model): Histogram depicting the distribution of regression residuals associated with the pooled power use model (a), and normal Q–Q plot assessing conformity of residuals to the theoretical normal distribution (b). 90

7.3 Residual Distribution Diagnostics (Interpretive Power Use Model): Histogram depicting the distribution of regression residuals associated with the interpretive power use model. 108

7.4 Residual Distribution Diagnostics (Tiered Power Use Model): Histogram depicting the distribution of regression residuals associated with the tiered execution power use model. 123

7.5 Observed-versus-fitted correspondence for the final power use regression models: (a) pooled observations, (b) interpretive execution, and (c) tiered execution. The dashed identity line represents perfect correspondence between observed and fitted mean power use. . . . 136

7.6 Residual-versus-fitted plots comparing linearity diagnostics for the preliminary untransformed energy consumption model and the final log-transformed energy consumption model. 149

7.7 Residual distribution diagnostics for the final pooled JRE JVM energy consumption regression model using the log-transformed dependent variable $\ln(energyWh)$. 152

7.8 Residual-versus-fitted plot for the final pooled JRE JVM energy consumption regression model using the log-transformed dependent variable ln(*energyWh*). 156
7.9 Residual Distribution Diagnostics (Interpretive Energy Consumption Model): Histogram of regression residuals (a) and corresponding normal Q-Q plot (b) for the interpretive execution energy consumption model. 173
7.10 Residual Distribution Diagnostics (Tiered Energy Consumption Model): Histogram of regression residuals (a) and corresponding normal Q-Q plot (b) for the tiered execution energy consumption model. 189
7.11 Observed-versus-fitted correspondence for the final log-transformed energy consumption regression models: (a) pooled observations, (b) interpretive execution, and (c) tiered execution. The dashed identity line represents perfect correspondence between observed and fitted ln(energyWh) values. 202

List of Tables

1.1 Dissertation Contributions and Supporting Empirical Evidence . . 28

2.1 Selected JDK Enhancement Proposals by OpenJDK Version . . . 65

4.1 Summary of Experimental Variables 133

4.2 Summary of the common experimental configuration employed throughout the study 137

4.3 Key GNU GCC configuration flags used during toolchain construction and their associated purposes. 156

4.4 Key OpenJDK configuration flags used during JRE JVM construction and their associated purposes. 158

4.5 Garbage collector availability across the OpenJDK source versions examined in this study. The CMS+ParNew garbage collector (CPGC) was deprecated and excluded from OpenJDK versions subsequent to OpenJDK 9. 161

4.6 Distribution of valid JRE/JVM instances by OpenJDK source version and execution mode. The reduced number of JDK 9 interpretive instances reflects the absence of one garbage collector variant, while the reduction in configurations for OpenJDK versions 10–14 reflects the exclusion of the CMS+ParNew garbage collector (CPGC). 162

4.7 The 24 application workloads composing the Renaissance Benchmark Suite, together with a brief description of each workload. Source: [5]. 166

4.8 Workload Category (WC) factor showing the grouping of Renaissance benchmark workloads into six workload categories. 167
4.9 JRE JVM runtime configuration flags used within the experimental framework. 170
4.10 Key JRE JVM runtime flags used during benchmark execution and their associated purposes. 172
4.11 Microarchitectural and operating system performance measures collected using the Linux `perf stat -d` framework. 177
4.12 Key elements of the benchmark execution procedure and their associated purposes. 180
4.13 Representative HotSpot tiered-compilation transitions observed during verification execution of the Renaissance `db-shootout` workload. 181
4.14 Distribution of JRE/JVM instances and benchmark-condition observations included in the final analytical dataset. 187
4.15 A description of all categorical dataset variables, detailing each factor's name, dataset identifier, underlying type, a short description of the variable, and the values associated with each factor level. . 193
4.16 A description of all numerical dataset variables, detailing each factor's name, dataset identifier, underlying type, and a short description of the variable. 194
4.17 Summary of the analytical dataset. 195

5.1 Power Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use. 236

5.2 Energy Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with energy consumption. 243

5.3 Power and Energy Ratios: Descriptive statistics associated with the ratio of interpretive execution relative to tiered execution. . . 247

5.4 Power Distributions: Descriptive statistics associated with each of the four GNU GCC versions, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use. 257

5.5 Energy Distributions: Descriptive statistics associated with each of the four GNU GCC versions, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with energy consumption. 264

5.6 Power Energy Ratios: Descriptive statistics associated with each of the four GNU GCC versions, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution. 268

5.7 Power Distributions: Descriptive statistics associated with each of the five garbage collector variants, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use. 277

5.8 Energy Distributions: Descriptive statistics associated with each of the five garbage collector variants, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use. 283

5.9 Power Energy Ratios: Descriptive statistics associated with each garbage collector variant, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution. 287

5.10 Power Distributions: Descriptive statistics associated with each of the six workload categories, reported for all pooled observations and grouped by execution mode, detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and sample size (**n**) associated with power use. 296

5.11 Energy Distributions: Descriptive statistics associated with each of the six workload categories, reported for all pooled observations and grouped by execution mode, detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and sample size (**n**) associated with power use. 302

5.12 Power Energy Ratios: Descriptive statistics associated with each of the six workload categories, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution. 307

5.13 Branch Misprediction Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with branch misprediction rates. 317

5.14 L1 Data Cache Load Miss Rate Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with L1 data cache load miss rates. 327

5.15 L2 Data Load Miss Rate Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with L2 data cache load miss rates. 337

5.16 Page Fault Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with page fault rates. 346

6.1 Power Use Distributions (Pooled Observations): Descriptive statistics summarising power use observations across workload categories and OpenJDK JRE versions when observations are pooled across both interpretive and tiered execution modes. 7

6.2 Power Use Distributions (Tiered Execution): Descriptive statistics summarising power use observations across workload categories and OpenJDK JRE versions under tiered execution. 8
6.3 Power Use Distributions (Interpretive Execution): Descriptive statistics summarising power use observations across workload categories and OpenJDK JRE versions under interpretive execution. 9
6.4 Energy Consumption Distributions (Pooled Observations): Descriptive statistics summarising energy consumption observations across workload categories and OpenJDK JRE versions when observations are pooled across both interpretive and tiered execution modes. . 15
6.5 Energy Consumption Distributions (Tiered Execution): Descriptive statistics summarising energy consumption observations across workload categories and OpenJDK JRE versions under tiered execution. 17
6.6 Energy Consumption Distributions (Interpretive Execution): Descriptive statistics summarising energy consumption observations across workload categories and OpenJDK JRE versions under interpretive execution. 17
6.7 Power Use Distributions (Pooled Execution (W)): Descriptive statistics summarising power use observations across workload categories and GNU GCC versions under pooled execution. 29
6.8 Power Use Distributions (Tiered Execution (W)): Descriptive statistics summarising power use observations across workload categories and GNU GCC versions under tiered execution. 30
6.9 Power Use Distributions (Interpretive Execution (W)): Descriptive statistics summarising power use observations across workload categories and GNU GCC versions under interpretive execution. . 31

6.10 Energy Consumption Distributions (Pooled Execution (Wh)): Descriptive statistics summarising energy consumption observations across workload categories and GNU GCC versions under pooled execution. 39
6.11 Energy Consumption Distributions (Tiered Execution (Wh)): Descriptive statistics summarising energy consumption observations across workload categories and GNU GCC versions under tiered execution. 40
6.12 Energy Consumption Distributions (Interpretive Execution (Wh)): Descriptive statistics summarising energy consumption observations across workload categories and GNU GCC versions under interpretive execution. 41
6.13 Power Use Distributions (Pooled Execution (W)): Descriptive statistics summarising power use observations across workload categories and Garbage Collector variants under pooled execution (w). . . . 52
6.14 Power Use Distributions (Tiered Execution (W)): Descriptive statistics summarising power use observations across workload categories and Garbage Collector variants under tiered execution (w). 53
6.15 Power Use Distributions (Interpretive Execution (W)): Descriptive statistics summarising power use observations across workload categories and Garbage Collector variants under interpretive execution (w). 54
6.16 Energy Consumption Distributions (Pooled Execution (Wh)): Descriptive statistics summarising energy consumption observations across workload categories and Garbage Collector variants under pooled execution. 64

6.17 Energy Consumption Distributions (Tiered Execution (Wh)): Descriptive statistics summarising energy consumption observations across workload categories and Garbage Collector variant under tiered execution. 65
6.18 Energy Consumption Distributions (Interpretive Execution (W)): Descriptive statistics summarising energy consumption observations across workload categories and Garbage Collector variants under interpretive execution. 66

7.1 Hierarchical regression model development for JRE JVM pooled power use. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. 84
7.2 Hierarchical regression coefficients for pooled JRE JVM power use. Standard errors correspond to HC3 heteroskedasticity-consistent robust variance estimation. 86
7.3 Cross-classification of influential observations identified through Cook's distance analysis. Percentages are calculated relative to all influential observations ($n = 96$). 92
7.4 Adjusted Generalised Variance Inflation Factor statistics for predictors included within the pooled power use regression model. . . 94
7.5 Final pooled power use regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. Coefficients are reported in watts. The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. 96
7.6 Stratified bootstrap coefficient stability analysis for the pooled power use regression model. 98

7.7 Hierarchical regression model development for JRE JVM interpretive execution power use. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. 103
7.8 Hierarchical regression coefficients for interpretive execution JRE JVM power use. Standard errors correspond to HC3 heteroskedasticity-consistent robust variance estimation. 105
7.9 Final interpretive power use regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. 110
7.10 Stratified bootstrap coefficient stability analysis for the interpretive execution power use regression model. 111
7.11 Hierarchical regression model development for JRE JVM tiered execution power use. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. 118
7.12 Hierarchical regression coefficients for tiered execution JRE JVM power use. Standard errors correspond to HC3 heteroskedasticity-consistent robust variance estimation. 119
7.13 Final tiered execution power use regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. Dependent variable is $\ln(energyWh)$. 125
7.14 Stratified bootstrap coefficient stability analysis for the tiered execution power use regression model. 127

7.15 Comparative variance contributions across hierarchical power use regression models 134
7.16 Comparative coefficient restructuring across final power use regression models 138
7.17 Hierarchical regression model development for JRE JVM pooled energy consumption. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. 144
7.18 Hierarchical regression coefficients for pooled JRE JVM energy consumption ($\ln(energyWh)$). Standard errors correspond to HC3 heteroskedasticity-consistent robust variance estimation. 145
7.19 Cross-classification of influential observations identified through Cook's distance analysis, detailing the number of influential observations belonging to each workload category. Percentages are calculated relative to all influential observations ($n = 130$). 154
7.20 Multicollinearity diagnostics for the final pooled JRE JVM energy consumption regression model. Adjusted GVIF values are reported to permit comparison across predictors with differing degrees of freedom. 158
7.21 Final pooled energy consumption regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. Dependent variable is $\ln(energyWh)$. 160
7.22 Stratified bootstrap coefficient stability analysis for the pooled energy consumption regression model. 162

7.23 Hierarchical regression model development for JRE JVM interpretive execution energy consumption. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. . . . 169
7.24 Hierarchical regression coefficient development for interpretive execution JRE JVM energy consumption (HC3 robust standard errors) 171
7.25 Final interpretive execution energy consumption regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. Coefficients are reported on the log-transformed energy consumption scale ($\ln(energyWh)$). The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. 175
7.26 Stratified bootstrap coefficient stability analysis for the interpretive execution energy consumption regression model. 178
7.27 Hierarchical regression model development for JRE JVM tiered execution energy consumption. The table reports explained variance for each model block, adjusted explained variance, and change in explained variance relative to the preceding block. 184
7.28 Hierarchical regression coefficient development for tiered execution JRE JVM energy consumption ($\ln(energyWh)$). HC3 heteroskedasticity-consistent robust standard errors are reported for all coefficient estimates. 186

7.29 Final tiered execution energy consumption regression model with HC3 robust standard errors and sensitivity analysis excluding influential observations. Coefficients are reported on the log-transformed energy consumption scale ($\ln(energyWh)$). The full model reports HC3 heteroskedasticity-consistent robust standard errors. The sensitivity model excludes observations exceeding the Cook's distance threshold of $4/n$. 191
7.30 Distribution of influential observations identified within the tiered execution energy consumption sensitivity analysis. Influential observations were identified using Cook's distance threshold $4/n$. Percentages in the final row indicate the proportion of all influential observations attributable to each workload category. 193
7.31 Stratified bootstrap coefficient stability analysis for the tiered execution energy consumption regression model. 195
7.32 Comparative variance contributions across hierarchical energy consumption regression models 200
7.33 Comparative coefficient restructuring across final energy consumption regression models 205

Volume I Preface

This dissertation is presented in two volumes as a single, continuous body of research. The division is intended solely to facilitate presentation and navigation and does not separate the research into independent studies. Chapter, section, figure, table, equation, and reference numbering therefore remain continuous across both volumes, while each volume retains independent page numbering.

Volume I: Foundations, Methodology and Main Effects establishes the conceptual, technical, and methodological foundations of the research and presents the first stage of the empirical investigation. Chapter 1 introduces the motivation, terminology, aims, research questions, and contributions. Chapters 2 and 3 provide the technical background and related literature, while Chapter 4 presents the theoretical, conceptual, experimental, measurement, dataset, and statistical frameworks. Chapter 5 addresses Research Question RQ1 through analysis of the main effects of JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures on JRE/JVM power use and energy consumption.

Volume II: Interaction Effects, Model Development and Synthesis continues the investigation through the workload-conditioned interaction analyses for Research Question RQ2 and the hierarchical modelling undertaken for Research Question RQ3. It then synthesises the findings across all three research questions and presents the discussion, conclusions, limitations, implications, future research directions, and supporting appendices. For navigation, the Table of Contents identifies the contents of both volumes, and a complete reference list is provided at the end of each volume so that either may be consulted independently.

ABSTRACT

The Java Runtime Environment (JRE) and its associated Java Virtual Machine (JVM) underpin a wide range of contemporary software systems, from embedded platforms to enterprise and cloud computing. While decades of research have focused on improving execution-time performance through advances in just-in-time compilation, considerably less attention has been directed towards understanding the factors that influence power use and energy consumption within managed runtime systems. Furthermore, existing studies have typically examined individual runtime components in isolation, providing limited insight into how multiple layers of runtime configuration collectively contribute to observed variation in power use and energy consumption.

This thesis develops and evaluates a multifactor empirical framework for investigating power use and energy consumption across the OpenJDK runtime environment. The study examines the influence of OpenJDK source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures on observed system-level power use and energy consumption. Experimental measurements were obtained using OpenJDK versions 9–14 executing the Renaissance benchmark suite on a Raspberry Pi 4 Model B platform, with electrical measurements acquired using high-resolution instrumentation and microarchitectural performance event measures collected through hardware performance monitoring.

The empirical investigation proceeds through three complementary stages. First, descriptive and non-parametric inferential analyses examine the independent effects of individual runtime configuration factors on power use and energy consumption. Second, interaction analyses investigate the extent to which these relationships vary across differing workload category levels. Finally, hierarchical regression models are developed to examine the extent to which variation in power use and energy consumption can be accounted for by progressively incorporating

runtime configuration factors, workload category, and emergent microarchitectural performance event measures. Model adequacy is assessed through regression diagnostics, sensitivity analyses, and bootstrap resampling procedures.

The results demonstrate that the factors influencing power use and energy consumption differ substantially in both magnitude and structure. Workload category consistently emerges as one of the strongest determinants of both outcomes, while execution mode exerts a particularly important influence on energy consumption. The analyses further demonstrate that microarchitectural performance event measures provide additional explanatory value beyond higher-level runtime configuration factors, although their relationships differ between interpretive and tiered execution. Comparative modelling also reveals that predictor effects are not invariant across execution modes, with several coefficients exhibiting substantial changes in magnitude and, in some cases, changes in direction when analysis is restricted to execution-mode-specific observations. In addition, the results identify an anomalous behavioural pattern associated with one OpenJDK release that is consistently observed across multiple analyses and is substantially moderated by execution mode.

Collectively, these findings provide a comprehensive empirical characterisation of the relationships between runtime configuration, workload categories, execution strategy, and microarchitectural performance event measures in explaining variation in JRE JVM power use and energy consumption. Beyond the individual empirical findings, the thesis contributes a reproducible analytical framework for investigating managed runtime systems, integrating descriptive statistical analysis, interaction modelling, and hierarchical regression within a unified methodology for examining software-hardware relationships in modern virtual machine environments.

DECLARATION

I have read and understood the Departmental policy on plagiarism. I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

Signature: 

Date: 3^{rd} September 2026

ACKNOWLEDGEMENTS

I would like to take this opportunity to acknowledge and recognise a number of important people who have influenced, guided, and supported the journey that I have taken.

From a professional and supervisory perspective, I must first acknowledge the late Dr James Power, who played possibly one of the most significant roles in inspiring me to focus on the inner workings of the Java Virtual Machine. James was such a kind and warm person who, at all times during my early PhD journey, seemed to know what it was that I was meant to find before I even understood my own positioning along that journey. Thanks, James.

Professor Rosemary Monahan and Dr Kevin Casey were exceptionally brave in taking me on as a returning PhD student after a long absence. To you both I will be forever thankful. I especially thank you for helping me frame my ideas into a clear and structured set of milestones. I am also so thankful for those questions during our many, many bi-weekly meetings, the answers to which have formed numerous additional sections within this work. Most importantly, thank you for your support and reassurance in reminding me that I was moving forward when, at times, I felt that I was regressing. Thank you both.

I would also like to express my sincere gratitude to my examiners, Dr Stephen Brown of Maynooth University and Professor David Gregg of Trinity College Dublin. The time, care, and considerable effort that you both invested in examining this dissertation, and particularly in providing such detailed, rigorous, and constructive feedback, have had a significant influence on its final form. The months of revision that followed the examination required me to revisit, clarify, restructure, and strengthen many aspects of the research, from its methodological presentation and empirical analysis through to the interpretation and articulation

of its contributions. While this represented a substantial body of work, the dissertation is considerably stronger as a consequence. I am deeply grateful to you both for the seriousness with which you engaged with the research and for the insight and attention you brought to the examination process.

My friend from our undergraduate years through to today, Dr Keith Maycock, has been especially supportive, constantly encouraging me to get back on the PhD track after many years away and also for taking the many phone calls. Thank you. Thanks also to Michael Bradford, Noel Cosgrave, Dr Corina Sheerin, Dr Deirdre Bane, and Dr Horacio Gonzalez-Velez for listening to my many ramblings. Oh, and let me not forget Murray's of Lusk; thank you for access to the "free" WiFi.

My parents, Mary and Joe, have been a continual source of encouragement throughout my life. From primary school, through secondary school and university, and long beyond those formal stages of education, they were always there: supporting, sacrificing, encouraging, and reminding us of the value of learning. One of the messages that stayed with me most strongly was that education is never a burden to carry. That belief has remained with me throughout this journey and, in many ways, made it possible for me to return to and ultimately complete it. I will always be grateful for the opportunities you created for us, often through sacrifices that I probably did not fully appreciate at the time.

I also want to acknowledge my brothers and sister, who have shared that journey with me in different ways over the years. Growing up within the same family, with the same encouragement and emphasis on education, has shaped all of us, and I am grateful for the support, humour, perspective, and sense of family that has remained constant throughout.

Finally, and without seeming to leave them until last, Aideen, my partner and friend, has made so many sacrifices for me so that I could get to this stage. I will forever be thankful to you and to our beautiful children, Ellie and Sean. Love you all. This is for you.

Chapter 1

Introduction

The evolution of programming languages has been driven by continual advances in computer hardware. As processor architectures have become increasingly complex, moving from simple single-core designs to highly parallel multi-core systems incorporating instruction pipelining, speculation, hardware multithreading, and other performance-enhancing techniques, programming languages and runtime environments have assumed a greater role in abstracting hardware complexity from software developers [6]. Among the languages that have successfully achieved this abstraction, Java has emerged as one of the most widely adopted programming platforms.

Since its initial release in 1995, Java has established itself as a mainstream software development language across a diverse range of application domains, including enterprise systems, web applications, mobile computing, embedded devices, scientific computing, and cloud-based services [7, 8, 9, 10, 11]. A key factor underpinning this success has been the Java Virtual Machine (JVM) and Java Runtime Environment (JRE), which provide platform independence through the execution of an intermediate bytecode representation rather than processor-specific machine code [12, 13]. The JVM has also become a target platform for numerous other programming languages, further extending the reach and importance of Java technology [14, 15, 16].

Over the last two decades, the JVM has undergone substantial architectural evolution. Early Java systems relied heavily upon bytecode interpretation, which

resulted in significant performance limitations relative to native compiled languages [17, 18, 12]. To address these limitations, successive JVM releases introduced increasingly sophisticated just-in-time (JIT) compilation techniques. Contemporary JVM implementations employ tiered execution, combining interpretation with multiple levels of dynamic compilation to balance start-up performance and long-term execution efficiency [19]. As a result, modern JVM performance can approach, and in some cases exceed, that of traditionally compiled languages for certain workloads [20, 21, 22, 23, 9].

The performance of Java applications is influenced not only by JVM design, but also by the interaction between the runtime environment and the underlying microarchitecture.

Processor-level mechanisms involving branch prediction, cache hierarchies, speculative execution, and parallel execution capabilities play an important role in determining application computational performance [18, 24, 25, 26]. Consequently, understanding Java performance requires consideration of both software-level runtime decisions and hardware-level execution characteristics.

While execution-time performance has traditionally dominated JVM research, contemporary computing environments increasingly require consideration of power use and energy consumption alongside performance. Economic pressures, environmental concerns, and the growing deployment of Java across resource-constrained and large-scale computing platforms have elevated energy consumption to a first-class design consideration. Understanding how JRE and JVM design decisions influence power use and energy consumption is therefore an important research challenge.

Against this backdrop, this thesis investigates the power use and energy consumption characteristics of the Java Runtime Environment (JRE) and Java Virtual Machine (JVM). The following section presents the motivation for this research and outlines the factors that make energy-aware analysis of contemporary JVM implementations both timely and necessary.

1.1 Motivation

The Java Runtime Environment (JRE) and its associated Java Virtual Machine (JVM) provide the execution infrastructure for a wide range of contemporary software systems. Decades of JVM research have resulted in increasingly sophisticated mechanisms for improving application execution, including adaptive interpretation, just-in-time compilation, garbage collection, memory management, and runtime optimisation. Much of this research has traditionally evaluated such developments principally through computational performance measures, particularly execution time and throughput. As power use and energy consumption have become increasingly important considerations across embedded, edge, mobile, cloud, and large-scale computing environments, there is a corresponding need to understand the electrical consequences associated with the software and runtime mechanisms through which applications execute.

This requirement is particularly important for managed runtime systems because application execution is mediated by multiple interacting software layers. The electrical characteristics observed while executing a JVM workload can potentially reflect decisions associated with the JRE implementation itself, the compiler toolchain used to construct that implementation, the JVM execution strategy, the garbage collector employed for memory management, and the characteristics of the workload being executed. These higher-level software factors ultimately determine the processor activity generated during execution, producing observable differences in microarchitectural performance events such as branch mispredictions, cache misses, and page faults. Understanding JRE/JVM power use and energy consumption therefore requires consideration of multiple levels of the software–hardware execution stack rather than the isolated analysis of a single runtime component.

Power use and energy consumption also represent related but distinct electrical outcomes. Power use describes the instantaneous rate at which electrical energy is consumed during execution, whereas energy consumption represents

the cumulative electrical demand over the duration of that execution. A runtime configuration can therefore alter these outcomes through different mechanisms. Changes in processor activity can affect instantaneous power demand, while changes in execution duration can substantially influence accumulated energy consumption. Consequently, conclusions drawn from one electrical outcome should not automatically be assumed to apply to the other, motivating their separate empirical investigation throughout this dissertation.

JVM execution mode represents one particularly important dimension within this wider problem. The contemporary HotSpot JVM combines interpretive execution with adaptive just-in-time compilation. Interpretation remains an important component of JVM execution, providing an initial execution pathway and runtime profiling information subsequently used to inform compilation decisions [27]. Interpretive execution also remains relevant to lightweight runtime implementations and constrained computing environments where the computational and memory requirements associated with dynamic compilation may be undesirable [7, 28]. Conversely, tiered execution progressively transforms frequently executed bytecode into optimised native machine code and provides substantial execution-time advantages over continued interpretation [29]. These fundamentally different execution strategies provide a strong motivation for examining whether their relationships with power use, energy consumption, and lower-level processor activity are similarly distinct.

The importance of this question extends to embedded and resource-constrained computing platforms, where both instantaneous electrical demand and cumulative energy consumption can impose practical constraints [30]. Single-board computers such as the Raspberry Pi increasingly provide sufficient computational capability to execute sophisticated managed runtime environments while retaining the resource and electrical constraints characteristic of embedded computing. Such platforms therefore provide a useful experimental context in which alternative JRE/JVM configurations can be examined under controlled hardware conditions.

The objective is not to assume that interpretive or tiered execution is universally preferable, but rather to determine how alternative execution strategies and runtime configurations are associated with differing power use and energy consumption characteristics.

The motivation for adopting a multifactor perspective is also informed by the work of Blackburn et al. [31]. Their work challenged performance-evaluation methodologies that were insufficiently representative of modern managed runtime environments and emphasised the need to consider multiple dimensions of JVM execution, including architecture, compilation, virtual machine implementation, memory management, and workload characteristics. The present research extends this general perspective to the empirical investigation of power use and energy consumption. Rather than examining individual JVM components in isolation, it considers multiple runtime, workload, and processor-level dimensions within a common experimental and analytical framework.

The related literature reviewed in Chapter 3 provides further motivation for this approach. Existing research has established that JVM implementation, execution strategy, garbage collection, workload characteristics, and lower-level processor activity can each be associated with differences in computational or electrical outcomes. However, this literature remains fragmented across individual runtime components and experimental contexts. In particular, comparatively little attention has been directed towards whether the GNU GCC build toolchain compiler used to construct an OpenJDK runtime subsequently contributes to observed variation in its power use or energy consumption. Similarly, although workload characteristics are recognised as important, less is known about the extent to which workload category conditions relationships associated with alternative runtime configurations. The relationships between emergent microarchitectural performance event measures and system-level electrical outcomes also remain incompletely characterised, particularly when considered jointly with higher-level runtime configuration factors.

These gaps motivate an integrated investigation capable of moving beyond isolated factor comparisons. Accordingly, this dissertation examines JRE/JVM power use and energy consumption through JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and selected microarchitectural performance event measures. The runtime configuration factors and workload category define experimentally controlled execution conditions, while the microarchitectural performance event measures provide observations of processor-level activity emerging during execution. Their joint consideration provides a means of examining both higher-level software configuration and the lower-level execution characteristics accompanying the measured electrical outcomes.

The empirical investigation progresses across three complementary analytical levels. First, the independent relationships associated with the principal runtime configuration factors, workload category, JVM execution mode, and microarchitectural performance event measures are examined for both power use and energy consumption. Second, the analysis determines whether relationships associated with runtime configuration remain consistent across workload categories or instead depend upon the workload context under which execution occurs. Finally, these dimensions are integrated within hierarchical regression models to determine the extent to which they collectively account for observed variation in power use and energy consumption, and whether their explanatory relationships differ between interpretive and tiered execution. Through this progression, the dissertation seeks to provide a multi-layer empirical characterisation of JRE/JVM power use and energy consumption and a structured framework for investigating electrical variation within managed runtime systems.

1.2 Terminology and Nomenclature

Java Runtime Environment (JRE) and Java Virtual Machine (JVM)

In this thesis, the terms Java Runtime Environment (JRE) and Java Virtual Machine (JVM) are closely linked and often discussed together. The JRE refers to the complete environment required to run Java applications, which comprises the JVM itself, the core libraries, and the supporting files. The JVM, as a central component of the JRE, is responsible for executing Java bytecode, managing memory, and providing system-level services. The JVM supports multiple execution strategies, including interpretive execution, Just-In-Time (JIT) compilation, and tiered execution. In this thesis interpretive execution denotes the mode in which Java bytecode is executed directly, instruction by instruction, without prior compilation to machine code. However, JIT compilation refers to the dynamic translation of bytecode to native machine code at runtime, while tiered execution describes the use of multiple compilation and interpretation strategies (e.g., interpreter, client C1, server C2) within the JVM.

Throughout this thesis, the term **execution mode** is used consistently to refer specifically to either interpretive execution (INT), in which bytecode is executed directly without prior compilation, or tiered execution (HS), in which multiple compilation stages are used during runtime execution.

Runtime Components and Experimental Factors

The JVM manages its own memory through automated memory management. In this thesis, garbage collector refers to the component responsible for this task within the JVM, with different strategies (e.g., Parallel, G1, CMS, Serial) used as experimental factors. Workload refers to a specific benchmark, application, or task executed on the JVM during measurement or analysis, often classified by category (e.g., parallel, object-oriented). Microarchitectural performance events are low-

level hardware events, including cache misses and branch mispredictions, that are recorded using hardware performance counters. Measures derived from these events are subsequently incorporated as predictor variables within the statistical modelling.

In addition, in this thesis, the term **workload category** refers specifically to the benchmark classification categories defined by the Renaissance benchmark suite.

In this thesis, the term **instance** refers to a fully specified experimental runtime configuration defined by a unique combination of JRE version, build toolchain compiler version, JVM execution mode, and garbage collector variant under which a benchmark workload is executed.

Power Use, Energy Consumption, and Experimental Measurement

The measurements of power use and energy consumption investigated in this thesis relate to the entire JRE, including both the JVM and its associated libraries and infrastructure, since these components operate in tandem and are inseparable during execution. Thus, references to power and energy throughout this work encompass the combined impact of the JVM and the broader JRE, reflecting the realities of how Java applications are deployed and run in practice.

In addition, although this thesis investigates the effects of JRE configuration on power use and energy consumption, the reported measurements do not represent power use or energy consumption attributable solely to the JRE in isolation. Electrical power measurements were obtained at the system boundary of the experimental platform, such that recorded power use and derived energy consumption reflect the combined activity of the underlying hardware platform, operating system environment, and the executing JRE operating as an integrated software stack. However, throughout all experimental executions, the underlying hardware platform, operating system environment, and all lower-level system

configurations were intentionally held constant. Consequently, observed variation in power use and energy consumption can be interpreted as arising primarily from differences in Java runtime execution conditions, while references throughout this thesis to JRE JVM power use and energy consumption should be understood as system-level measurements observed under differing Java runtime configurations rather than direct measurements attributable exclusively to the JRE itself.

Although the term *power consumption* is widely used within the literature, this thesis adopts the term *power use* throughout for terminological consistency when discussing measured electrical power, and refers to *electrical power*, defined as the instantaneous rate at which electrical energy is transferred or consumed within an electrical system. In accordance with standard electrical engineering terminology, electrical power is expressed as the product of voltage and current, measured in watts (W):

$$P = VI$$

where (P) denotes electrical power, (V) denotes voltage, and (I) denotes electric current [32].

Similarly, the term *energy consumption* refers to *electrical energy*, defined as the total quantity of electrical energy transferred or consumed by a system over a period of time. Electrical energy is equivalent to the time integral of power and may be expressed as:

$$E = \int P(t), dt$$

or, under conditions of approximately constant power, as:

$$E = Pt$$

where (E) denotes electrical energy, (P) denotes electrical power, and (t) denotes elapsed time. Under the International System of Units (SI), energy is

measured in joules (J), although for practical purposes within this thesis energy consumption is also reported in watt-hours (Wh) [32].

These definitions of power use and energy consumption establish the physical basis upon which all measurements and subsequent analyses conducted throughout this thesis are founded. In examining how various system configurations and workloads influence these quantities, standard terminology from statistical modelling and experimental design is adopted. Specifically, the attributes examined in relation to power use and energy consumption are referred to throughout this thesis as predictor variables when incorporated within statistical analyses. These comprise experimentally specified runtime and workload variables and emergent microarchitectural performance event measures, with the latter observed during execution rather than directly manipulated. Predictor variables are the controllable or measurable attributes whose influence on power use and energy consumption is being assessed. Examples in this research include JRE version, compiler version, execution mode, garbage collector type, microarchitectural performance event measures, and workload category. For predictor variables that are categorical in nature, the terms factor and factor levels are used.

Predictor, Outcome, and Emergent Variables

The measured quantities, **power use** and **energy consumption**, are referred to as outcome variables (or dependent variables). Outcome variables are the principal quantities of interest in this thesis and represent the measured responses associated with differing experimental conditions. Predictor variables (also referred to in the literature as explanatory variables or independent variables) represent the measurable or controllable attributes whose influence on these outcome variables is being assessed.

Within this thesis, the term **model** refers to a statistical representation of the relationship between predictor variables and measured outcome variables. Models developed in this thesis are intended to estimate and quantify statistical

relationships observed within the experimental data rather than establish direct causal mechanisms or provide direct explanatory accounts of the underlying physical processes generating those observations.

Throughout this thesis, the term **difference** refers to a comparison between measurements obtained under two distinct experimental conditions or categories. In contrast, the term **variation** refers to the degree of dispersion or spread observed within repeated measurements or within a statistical distribution. The term **effect** refers to the estimated influence of a predictor variable on an outcome variable as determined through inferential statistical analysis or regression modelling.

Among the predictor variables examined in this thesis, branch misprediction rate, L1 data cache load miss rate, L2 data cache load miss rate, and page fault rate are derived from performance monitoring events collected during execution. These **microarchitectural performance event measures** are distinguished as **emergent variables** because they are observed during execution rather than directly manipulated. Unlike directly controllable factors (e.g., JRE version or garbage collector), these emergent variables are not set prior to execution. Instead, their values arise from the interaction between the software workload, JVM configuration, and underlying hardware during runtime. These microarchitectural performance event measures are measured as part of the system's natural response to specific experimental conditions and are included in the model as predictors to capture their relationship with observed power use and energy consumption, although they are not directly manipulated.

In regard to the scale of measurement associated with the predictor variables considered within this thesis, with the exception of the microarchitectural performance event measures, all remaining predictor variables are categorical in nature and are measured at either the nominal or ordinal scale. To facilitate their inclusion within the regression models, these variables are represented using dummy-coded indicator variables. This approach enables categorical predictors to be incorporated within a linear regression framework without imposing assump-

tions of arithmetic distance between categories, particularly in the case of ordinal variables. Accordingly, the regression models estimate the differential effect associated with category membership relative to a designated reference category. A detailed discussion of the use of dummy-coded variables within regression models can be found in [33].

Consequently, the modelling approach adopted in this thesis estimates variation in JRE JVM power use and energy consumption as a function of categorical execution characteristics and emergent continuous microarchitectural performance event measures within a general linear modelling framework.

Bootstrap Terminology and Usage

Throughout this thesis, the terms **bootstrap** and **bootstrapping** are used in four distinct technical contexts. To avoid ambiguity, each use is explicitly qualified according to its meaning.

First, **statistical bootstrap resampling** refers to the repeated resampling procedure used to assess regression coefficient stability and to estimate coefficient standard errors and confidence intervals. Where the terms **bootstrap** or **bootstrapping** are used without an additional systems-related qualifier, they refer exclusively to this statistical procedure.

Second, the term **JVM bootstrap class loader** refers to the Java Virtual Machine class-loading mechanism responsible for loading the core Java runtime classes required during JVM initialisation.

Third, **JRE bootstrap compilation** refers to the requirement for an existing valid JRE and Java compiler to build a newer OpenJDK JRE and its associated JVM from source.

Fourth, **GCC bootstrap compilation** refers to the corresponding requirement for an existing valid GNU GCC compiler to compile a newer GCC compiler version from source.

These qualifiers are used consistently throughout the thesis to distinguish

statistical resampling from the separate bootstrap processes associated with JVM initialisation, OpenJDK construction, and GCC compiler construction.

Statistical Distributions and Distributional Characteristics

Throughout this thesis, the term **distribution** refers to the statistical representation of how the observed values of a particular variable are spread or dispersed across a set of measurements. For example, the distribution of power use or energy consumption describes the range, frequency, and pattern of observed values obtained under differing experimental conditions, including Java runtime configurations, build toolchain compiler versions, garbage collector variants, JVM execution modes, and workload category. Understanding these distributions is essential for identifying patterns within the data, assessing variability, detecting outliers, and determining the appropriateness of subsequent statistical analyses. Unless otherwise specified, the term distribution is used exclusively in the statistical sense (for example, probability distribution, empirical distribution, or observed data distribution), rather than in a software deployment or code packaging context.

Throughout this thesis, the term **characteristics**, when used in reference to power use, energy consumption, execution time, or microarchitectural performance event measures, refers specifically to the **statistical properties that define and differentiate one observed distribution from another**. These descriptive statistical measures characterise the distribution and comprise the minimum value, maximum value, arithmetic mean, first quartile, median, third quartile, standard deviation, interquartile range, skewness, and kurtosis. Collectively, these measures describe important aspects of a distribution, including its central tendency, variability, dispersion, asymmetry, and overall shape.

By analogy, a tailored suit can be described by observable attributes including neck width, shoulder width, chest width, waist width, sleeve length, leg length, fabric type, and colour. Similarly, a statistical distribution can be characterised by measurable properties that distinguish it from other distributions. Consequently,

throughout this thesis, references to power use, energy consumption, execution time, or microarchitectural event characteristics should be interpreted as referring to the statistical properties of the observed distributions associated with those variables, rather than as informal or qualitative descriptions of a system.

In addition, throughout this thesis, the term **pooled observations** refers to the aggregation of observations across multiple experimental categories into a single combined dataset for the purpose of examining overall statistical distributions or estimating overall effects. For example, pooled observations may refer to measurements aggregated across all workload categories, execution modes, or runtime configurations when analysing overall trends independent of subgroup membership.

The terminology and definitions established in this section are applied consistently throughout the experimental design, statistical modelling, and results interpretation chapters. Where multiple alternative terms exist within the literature, the nomenclature adopted in this thesis has been selected deliberately to minimise ambiguity and ensure terminological consistency throughout the work.

1.3 Aims and Objectives

Given the central role of the Java Runtime Environment (JRE) and Java Virtual Machine (JVM) in executing enterprise, mobile, web, financial, embedded, and cloud-based applications, understanding the factors associated with their power use and energy consumption is of increasing importance within contemporary computing.

The overarching aim of this research is to develop a comprehensive empirical understanding of the factors associated with variation in JRE/JVM power use and energy consumption. In pursuing this aim, the research examines multiple dimensions of runtime configuration, workload context, execution strategy, and emergent microarchitectural activity. These dimensions include JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage

collector variant, workload category, and selected microarchitectural performance event measures.

Specifically, this research seeks to:

(i) characterise and quantify the differences in JRE/JVM power use and energy consumption associated with JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category;

(ii) examine the relationships between selected microarchitectural performance event measures and observed JRE/JVM power use and energy consumption;

(iii) determine the extent to which relationships associated with runtime configuration vary according to workload category and JVM execution mode;

(iv) evaluate whether the explanatory relationships associated with runtime configuration, workload category, and microarchitectural performance event measures differ between interpretive and tiered execution; and

(v) develop and evaluate statistical models capable of accounting for variation in JRE/JVM power use and energy consumption through the combined consideration of runtime configuration factors, workload category, and microarchitectural performance event measures.

Through addressing these objectives, the research seeks to provide a multi-layer empirical characterisation of JRE/JVM power use and energy consumption and to establish a structured analytical framework through which runtime configuration, workload context, execution strategy, and processor-level activity can be examined collectively.

To operationalise these aims and objectives, the following section introduces the research questions that structure the investigation and guide the empirical analysis of JRE/JVM power use and energy consumption.

1.4 Research Questions

As power use and energy consumption becomes a paramount concern in both mobile and large-scale computing environments, understanding the variables that influence software power use and energy consumption is increasingly vital. The Java Runtime Environment (JRE) and its associated Java Virtual Machine (JVM) play a central role in modern software systems, yet their power use and energy consumption characteristics remain underexplored.

The JVM is a complex execution platform whose resource usage is shaped by numerous internal and external factors, ranging from the JRE source version and build toolchain compiler, to execution mode (interpretive vs. tiered), garbage collection strategies, and microarchitectural performance event measures. Workload category further adds to this complexity by affecting how the JVM utilises system resources during execution. In this context, it is necessary to ask: How do these diverse system factors influence JRE/JVM power use and energy consumption? This question guides a detailed investigation of the foundational components that shape JRE JVM power use and energy consumption, helping to clarify how each contributes to the profile of the system. Our first research question is therefore:

RQ 1: How do variations in JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category affect JRE JVM power use and energy consumption, and what relationships exist between these outcomes and observed microarchitectural performance event measures?

Within this thesis, JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category are treated as controllable system-level factors, whereas microarchitectural performance event measures are treated as emergent runtime measurements observed during execution.

In addition, although JVM execution mode constitutes an important analytical

factor within this study, it is not treated as a standalone predictor category within a dedicated results section. This decision reflects the fact that JVM execution mode is inherently cross-cutting and context dependent. Rather than existing independently of other system-level factors, the effects associated with interpretive and tiered execution are conditional upon JRE source version, garbage collector variant, build toolchain compiler version, workload category, and observed microarchitectural performance event measures. Consequently, execution mode is systematically decomposed and evaluated within each results section to facilitate direct comparison of interpretive and tiered execution under varying runtime conditions.

This analytical structure enables the study to assess whether differences between interpretive and tiered execution are stable across contexts or whether their effects are moderated by workload and system-level characteristics.

In regard to RQ1, we derive six sub-research questions, which we assess individually across Chapter 5. They are:

SRQ1.1: What effect does JRE source version have on JRE JVM power use and energy consumption? [Chapter 5, §5.2]

SRQ1.2: What effect does GNU GCC build toolchain compiler source version have on JRE JVM power use and energy consumption? [Chapter 5, §5.3]

SRQ1.3: What effect does garbage collector variant have on JRE JVM power use and energy consumption? [Chapter 5, §5.4]

SRQ1.4: What effect does workload category have on JRE JVM power use and energy consumption? [Chapter 5, §5.5]

SRQ1.5: What relationship exists between observed microarchitectural performance event measures and JRE JVM power use and energy consumption? [Chapter 5, §5.6]

SRQ1.6: How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions? [Addressed throughout Chapter 5 via decomposition of all predictor analyses according to execution mode.]

In addition, and to save space, in regard to sub-research questions **SRQ1.1** through **SRQ1.4**, we assess power use and energy consumption separately through "**a**," and "**b**" versions. For example, **SRQ1.1a** focuses on the effect that JRE source version have on JRE JVM power use. Whereas, **SRQ1.1b** focuses on the effect that JRE source version have on JRE JVM energy consumption.

Identifying influential system-level factors and the differences associated with their respective levels is an important first step in understanding JRE JVM power use and energy consumption. However, an additional question naturally emerges: are these effects stable across differing workload conditions, or are they dependent upon the characteristics of the workload being executed? Although JRE source version, build toolchain compiler version, JVM execution mode, and garbage collector variant may exhibit statistically significant effects when examined independently, the magnitude and practical significance of these effects may vary considerably according to workload category. A runtime configuration that produces favourable power use or energy consumption behaviour for one class of workload may not necessarily yield the same outcome for another. Accordingly, following the identification of influential factors in RQ1, it becomes necessary to examine the extent to which these effects are moderated by workload category and whether observed differences remain consistent across differing computational contexts. Thus, RQ2 builds upon RQ1 by asking:

RQ 2: To what extent are the effects associated with JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, and garbage collector variant on JRE JVM power use and energy consumption dependent upon workload category?

Similar to RQ1, we derive four sub-research questions by asking:

SRQ2.1: To what extent do the effects of JRE source version on JRE JVM power use and energy consumption differ across workload categories? [Chapter 6, §6.2]

SRQ2.2: To what extent do the effects of GNU GCC build toolchain compiler source version on JRE JVM power use and energy consumption differ across workload categories? [Chapter 6, §6.3]

SRQ2.3: To what extent do the effects of garbage collector variant on JRE JVM power use and energy consumption differ across workload categories? [Chapter 6, §6.4]

SRQ2.4: To what extent do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing workload categories? [Addressed throughout Chapter 6 via decomposition of contextual workload analyses according to execution mode.]

While identifying influential factors and understanding the extent to which their effects depend upon workload category is critical, a deeper and more practical question follows: can these effects collectively be quantified and used to account for observed variation in JRE/JVM power use and energy consumption? Moving from exploratory and contextual analyses (RQ1 and RQ2) to a model-driven approach (RQ3) enables actionable insights for development and system optimisation. If system-level factors, workload category, and observed microarchitectural performance event measures significantly influence power use and energy consumption, then these same factors may serve as predictors within a multivariate regression modelling framework. Accordingly, the findings of RQ1 and RQ2 provide the empirical basis for the modelling strategy adopted in RQ3. Thus, RQ3 builds upon the preceding research questions by asking: to what extent can regression modelling that incorporate these factors account for observed variation in JRE/JVM power use and energy consumption?

RQ 3: To what extent can variation in JRE/JVM power use and energy consumption be accounted for through regression modelling incorporating JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures?

Similar to RQ1 and RQ2, we derive two sub-research questions by asking:

SRQ3.1: To what extent can variation in JRE/JVM power use be accounted for through regression modelling incorporating JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures? [Chapter 7, §7.2]

SRQ3.2: To what extent can variation in JRE/JVM energy consumption be accounted for through regression modelling incorporating JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures? [Chapter 7, §7.6]

In a similar manner to RQ1 and RQ2, and to conserve space throughout the subsequent analysis, sub-research questions **SRQ3.1** and **SRQ3.2** are additionally decomposed according to execution mode using the '**a**" and '**b**" designations. Specifically, the **a** designation refers to regression models estimated using observations generated under interpretive execution, whereas the **b** designation refers to regression models estimated using observations generated under tiered execution. For example, **SRQ3.1a** focuses on the extent to which variation in JRE/JVM interpretive execution power use can be accounted for through regression modelling, whereas **SRQ3.1b** examines the corresponding extent to which variation in tiered execution power use can be accounted for using the same modelling framework.

RQ3 is addressed through a regression modelling framework in which power use (P) and energy consumption (E) are modelled as functions of JRE source version

(JSV), GNU GCC build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), workload category (WC), and microarchitectural performance event measures (MPE). Because the two outcome variables represent distinct computational processes, separate model specifications are subsequently evaluated for each outcome measure. The general form of the modelling framework is presented here, with full specification and parameterisation discussed in Chapter 4 §4.11 and Chapter 7, titled Model Development. In particular, Equation 1.1 details the regression of power use, whereas Equation 1.2 details the regression of energy consumption [both models are operationalised in §4.11].

$$\begin{aligned} P_i = \beta_0 + \beta_1(JSV_i) + \beta_2(BTCV_i) + \beta_3(JEM_i) \\ + \beta_4(GCV_i) + \beta_5(WC_i) + \beta_6(MPE_i) + \varepsilon_i \end{aligned} \tag{1.1}$$

$$\begin{aligned} E_i = \beta_0 + \beta_1(JSV_i) + \beta_2(BTCV_i) + \beta_3(JEM_i) \\ + \beta_4(GCV_i) + \beta_5(WC_i) + \beta_6(MPE_i) + \varepsilon_i \end{aligned} \tag{1.2}$$

Together, these research questions establish a structured framework for examining JRE/JVM power use and energy consumption. RQ1 characterises the relationships associated with individual runtime, workload, and microarchitectural factors; RQ2 determines the extent to which runtime-configuration relationships are conditioned by workload category; and RQ3 evaluates the extent to which these variables jointly account for observed variation through hierarchical regression modelling. Collectively, the three research questions provide a progression from factor-level empirical characterisation, through contextual interaction analysis, to multivariable explanatory modelling of JRE/JVM power use and energy consumption.

RQ1 and RQ2 are addressed through the inferential statistical analyses pre-

sented in Chapter 5 and Chapter 6, where the effects of individual system-level factors, workload category, and runtime measurements are examined independently and in combination. RQ3 is addressed through the statistical modelling analyses presented in Chapter 7, where multivariate linear models are used to assess the extent to which these factors collectively account for variation in JRE JVM power and energy consumption.

1.5 Thesis Contributions

This dissertation contributes to understanding the factors influencing Java Runtime Environment (JRE) and Java Virtual Machine (JVM) power use and energy consumption through an integrated empirical investigation of runtime implementation, workload category, execution strategy, and emergent microarchitectural behaviour. The contributions extend beyond the identification of individual factor effects by demonstrating that the relationships governing JRE/JVM power use and energy consumption are conditional upon workload, execution context, lower-level processor behaviour, and the electrical outcome being considered. The principal contributions arising from this research are summarised below.

C1: Comprehensive Multi-Layer Characterisation of JRE JVM Power Use and Energy Consumption.

This research provides an integrated empirical characterisation of JRE/JVM power use and energy consumption across multiple OpenJDK JRE source versions, GNU GCC build toolchain compiler versions, JVM execution modes, garbage collector variants, workload category, and emergent microarchitectural performance event measures. By considering these factors within a common experimental framework, the dissertation establishes that their relationships with power use and energy consumption are markedly uneven rather than equally influential. OpenJDK implementation, execution strategy, workload category,

build toolchain compiler version, garbage collection strategy, and lower-level processor behaviour contribute at differing levels depending upon the analytical context and outcome measure under consideration. The research therefore provides a multi-layer empirical account of how software implementation and execution characteristics are associated with system-level electrical outcomes within a contemporary OpenJDK environment.

C2: Workload as a Conditioning Context for Runtime Configuration Effects.

This dissertation demonstrates that workload category does more than independently differentiate JRE/JVM power use and energy consumption. The influence associated with runtime configuration is itself conditioned by the computational characteristics of the executing workload. Explicit analysis of workload-dependent relationships shows that the magnitude and statistical detectability of effects associated with OpenJDK JRE source version, GNU GCC build toolchain compiler version, and garbage collector variant vary across workload category, with the strength of this conditioning additionally differing between power use and energy consumption and, in several cases, between interpretive and tiered execution. The research therefore demonstrates that workload should be understood as the computational context within which runtime configuration effects are expressed, rather than solely as an additional independent predictor. This finding has implications for the generalisability of conclusions derived from individual benchmarks or narrowly defined application domains.

C3: Execution-Mode-Dependent Statistical Observability of Software–Hardware Relationships.

This dissertation demonstrates that interpretive and tiered JVM execution represent materially different software–hardware analytical contexts rather than simply alternative execution strategies associated with differing execution times. Decomposition according to execution mode substantially alters variance contributions, coefficient magnitudes, statistical detectability, and, in some instances, coefficient direction. For power use in particular, several microarchitectural relationships become substantially more statistically observable under interpretive execution, whereas tiered execution shifts a greater proportion of explanatory importance towards higher-level runtime implementation characteristics. These findings demonstrate that execution strategy affects the level within the software–hardware stack at which variation becomes statistically visible. Consequently, relationships estimated from pooled observations should not automatically be assumed to characterise those evident within the constituent interpretive and tiered execution contexts.

C4: Microarchitectural Performance Events as an Analytical Bridge Between Runtime Configuration and Processor Behaviour.

This research demonstrates the value of microarchitectural performance event measures not merely as statistical predictors of power use and energy consumption, but as emergent indicators through which higher-level runtime and workload category can be related to lower-level processor execution. Branch misprediction rate, L1 and L2 data cache load miss rates, and page fault rate exhibited substantially different distributions and statistical relationships according to runtime and execution context. These measures therefore exposed execution characteristics that were not evident from system-level electrical

measurements alone. The findings further demonstrate that microarchitectural performance events should not necessarily be interpreted as independent causal determinants, since their observed values emerge through interactions between workload, runtime implementation, execution strategy, and processor architecture. Their incorporation within the analysis consequently provides an empirical bridge between configured software characteristics, processor-level execution behaviour, and observed electrical outcomes.

C5: A Layered Hierarchical Framework for Identifying Context-Dependent Explanatory Relationships.

This dissertation develops and applies a theory-driven hierarchical modelling framework that progresses from runtime implementation characteristics, through execution configuration and workload category, to emergent microarchitectural performance event measures. The framework operationalises the conceptual distinction between directly configured runtime characteristics and lower-level execution behaviour emerging through interactions between workload, runtime configuration, and processor architecture. Its contribution extends beyond the partitioning of incremental explained variance. Empirical application demonstrates that successive predictor blocks can restructure relationships attributed to previously entered runtime characteristics, with coefficient magnitudes, statistical significance, and, in some cases, coefficient direction changing as progressively deeper analytical layers are represented. Combined with pooled and execution-mode-specific modelling, diagnostic assessment, heteroscedasticity-consistent inference, sensitivity analysis, and stratified bootstrap validation, the framework provides a systematic means of distinguishing relationships that remain comparatively stable across analytical contexts from those whose

apparent explanatory importance depends upon workload, execution strategy, or lower-level processor behaviour.

C6: Demonstration of Distinct Explanatory and Statistical Structures Governing Power Use and Energy Consumption.

This research demonstrates that JRE/JVM power use and energy consumption, although physically related, exhibit distinct explanatory and statistical structures. workload category represented the dominant source of additional explained variation within the power use models, whereas JVM execution mode accounted for the dominant contribution within the pooled energy consumption model. The two outcomes also required different statistical treatment: power use could be represented satisfactorily on its original measurement scale, whereas energy consumption required logarithmic transformation to obtain an appropriately specified regression model. These differences are consistent with power use representing an instantaneous rate of electrical demand and energy consumption representing a cumulative, duration-dependent outcome. Consequently, statistical models, predictor relationships, and substantive conclusions derived for power use should not be assumed to transfer directly to energy consumption.

C7: Multi-Level Empirical Characterisation of a Distinctive OpenJDK 13 Execution Characteristic.

This dissertation identifies and systematically characterises a distinctive execution characteristic associated with OpenJDK 13 within the experimental environment. OpenJDK 13 exhibited consistently elevated power use relative to the remaining assessed JRE source versions, while also exhibiting systematic differences in selected microarchitectural performance characteristics, most notably branch misprediction

rate and L1 data cache load miss rate. The observation persisted across independent comparisons, workload-conditioned analyses, hierarchical power use models, execution-mode decomposition, sensitivity assessment, and bootstrap validation, providing convergent evidence that the observed power use difference represents a persistent implementation-specific characteristic rather than an isolated statistical observation. Its energy consumption relationship was considerably more dependent upon execution strategy, becoming strongly expressed under interpretive execution but largely disappearing within the tiered model specification. The OpenJDK 13 case therefore provides a substantive example of why runtime-version effects should be evaluated across multiple analytical levels and separately for power use and energy consumption. While the precise implementation mechanism responsible cannot be established from the present measurements, the findings provide a clearly defined empirical phenomenon for subsequent source-level investigation.

The contributions outlined above are supported by evidence developed across the methodological, empirical, and interpretive stages of this dissertation. To make this evidential basis explicit, Table 1.1 maps each contribution to the principal chapters and sections in which it is developed, empirically demonstrated, and synthesised within the discussion. This provides a direct link between the contribution claims and the analyses from which they arise, while also showing that several contributions are supported through convergent evidence drawn from more than one research question or analytical level.

1.5.1 Publications and Research Outputs

The following publications and research outputs arose from this research and contributed to the development and dissemination of the work presented in this dissertation:

Table 1.1: Mapping of dissertation contributions to supporting empirical evidence.

ID	Contribution	Supporting Evidence
C1	Comprehensive multi-layer characterisation of JRE JVM power use and energy consumption	Chapter 5, Sections 5.2–5.6, establishing the independent runtime, workload, execution-mode, and microarchitectural relationships; Chapters 6 and 7 provide complementary interaction and multivariable evidence; synthesised across Chapter 8, Sections 8.2–8.5.
C2	Workload as a conditioning context for runtime configuration effects	Chapter 6, Sections 6.2–6.4, together with the workload predictor blocks and execution-mode-specific comparisons presented in Chapter 7; synthesised in Chapter 8, Section 8.3.
C3	Execution-mode-dependent statistical observability of software–hardware relationships	Execution-mode decompositions throughout Chapters 5 and 6, together with the interpretive and tiered power and energy models and the comparative variance decomposition and coefficient restructuring analyses in Chapter 7; synthesised in Chapter 8, Section 8.4.
C4	Microarchitectural performance events as an analytical bridge between runtime configuration and processor behaviour	Chapter 5, Section 5.6, particularly the branch misprediction, L1 and L2 data-cache load miss, and page fault analyses; supported by the microarchitectural predictor blocks and cross-model comparisons in Chapter 7; synthesised in Chapter 8, Section 8.5.
C5	Layered hierarchical framework for identifying context-dependent explanatory relationships	Chapter 4, Section 4.2, and particularly Sections 4.3 and 4.4, where the layered conceptual and hierarchical modelling framework is developed and operationalised; Chapter 7 (RQ3), where the framework is empirically applied through pooled and execution-mode-specific power use and energy consumption models, and evaluated through comparative variance decomposition and coefficient restructuring; and Chapter 8, Section 8.7, where the resulting context-dependent explanatory relationships are synthesised and interpreted.
C6	Distinct explanatory and statistical structures governing power use and energy consumption	Chapter 7, particularly the pooled and execution-mode-specific power use and energy consumption models and Sections 7.5 and 7.9; synthesised in Chapter 8, Section 8.8.
C7	Multi-level empirical characterisation of a distinctive OpenJDK 13 execution characteristic	Chapter 5, Section 5.2 and the OpenJDK-specific analyses within Section 5.6; Chapter 6, Section 6.2; and the pooled, interpretive, tiered, sensitivity, bootstrap, and cross-model analyses in Chapter 7; synthesised in Chapter 8, Section 8.6.

- J. Lambert, R. Monahan, and K. Casey, "Power Consumption Profiling of a Lightweight Development Board: Sensing with the INA219 and Teensy 4.0 Microcontroller," *Electronics*, vol. 10, no. 7, Art. no. 775, 2021, doi: 10.3390/electronics10070775.
- J. Lambert, R. Monahan, and K. Casey, "Accidental Choices—How JVM Choice and Associated Build Tools Affect Interpreter Performance," *Computers*, vol. 11, no. 6, Art. no. 96, 2022, doi: 10.3390/computers11060096.
- J. Lambert, K. Casey, and R. Monahan, "Repositioning Tiered HotSpot Execution Performance Relative to the Interpreter," arXiv:2304.06460, 2023.

1.5.2 Overall Contribution

Collectively, these contributions extend current understanding of JRE/JVM power use and energy consumption beyond the effects associated with individual runtime configuration factors. The findings demonstrate that observed electrical characteristics emerge through relationships spanning runtime implementation, workload, execution strategy, and lower-level processor behaviour, and that the statistical visibility of these relationships depends upon the analytical context within which they are examined. The dissertation further contributes an integrated analytical framework for progressing from independent effects, through workload-conditioned relationships, to hierarchical and execution-mode-specific modelling of the resulting software–hardware system.

More broadly, the research demonstrates that instantaneous power use and cumulative energy consumption require distinct analytical interpretation, while the OpenJDK 13 findings illustrate the value of combining electrical, workload-conditioned, microarchitectural, and multivariable evidence when investigating implementation-specific runtime characteristics. Together, these contributions provide both new empirical evidence concerning contemporary OpenJDK execution and a methodological basis for future investigation of managed runtime power use, energy consumption, and software–hardware relationships across Java and related managed runtime environments.

1.6 Thesis Organisation

This dissertation is organised into nine chapters, progressing from theoretical background and methodological foundations through empirical analysis, predictive modelling, discussion, and conclusions.

Chapter 2 presents the background material underpinning this dissertation. We first introduce the Java programming language and then present an overview of the Java Runtime Environment (JRE) and the structural components of the Java Virtual Machine (JVM), including the class loader subsystem, runtime data areas, and execution engine. This chapter also introduces Java benchmark workloads and their characteristics, before concluding with an overview of approaches for the measurement and collection of electrical power and energy.

Chapter 3 presents a detailed review of the relevant literature. The review begins with studies examining JRE JVM power use and energy consumption modelling, followed by literature relating to microarchitectural performance event measures and their relationship with system-level behaviour. We then review prior research concerning workload category and workload modelling, before examining literature associated with GNU GCC build toolchain compiler effects, garbage collector strategies, and JVM execution modes. Each subsection concludes with an assessment of the associated gaps in the literature that motivate the present study.

Chapter 4 presents the methodological framework adopted throughout this dissertation (§4.2–§4.3). This chapter introduces the overarching experimental design (§4.4), hardware and operating system environment (§4.5), and the collection of OpenJDK JRE instances analysed in this study, including the OpenJDK source versions, GNU GCC build toolchain compiler versions used in their construction, and garbage collector variants relied upon for memory management (§4.6). We follow that section with a review of the benchmark workloads used to stress the JRE JVM instances (§4.7). We follow that with a detailed overview of the benchmark execution procedure, microarchitectural performance event data collec-

tion, power use and energy consumption measurement procedure, and then data associated with experimental reliability (§4.8). Following that section we present a description of the shared dataset underpinning this dissertation. Specifically, the chapter introduces the categorical and numerical variables contained within the dataset, detailing their characteristics, associated factor levels, measurement scales, and dataset identifiers. In addition, the chapter presents information relating to the availability of the associated data artefact to support transparency and reproducibility (§4.9). Then we present the statistical procedures relied upon for inferential analysis (§4.10). The penultimate section covers a presentation of the overarching empirical framework, including the dependent and independent variables used throughout the subsequent analyses and predictive modelling (§4.11). The final section details threats to validity (§4.12).

Chapter 5 presents the empirical results associated with Research Question **RQ1**, focusing on the independent effects associated with JRE JVM power use and energy consumption. The chapter examines, in sequence, the effects associated with OpenJDK JRE source version, GNU GCC build toolchain compiler version, garbage collector variant, workload category, microarchitectural performance event measures, and JVM execution mode. Across each analytical section, results are reported for both power use and energy consumption and are evaluated across interpretive and tiered execution modes.

Chapter 6 presents the empirical results associated with Research Question **RQ2**, examining the interaction effects associated with workload category. Specifically, the chapter evaluates the extent to which the effects associated with OpenJDK JRE source version, GNU GCC build toolchain compiler version, garbage collector variant, and JVM execution mode on JRE JVM power use and energy consumption differ according to workload category.

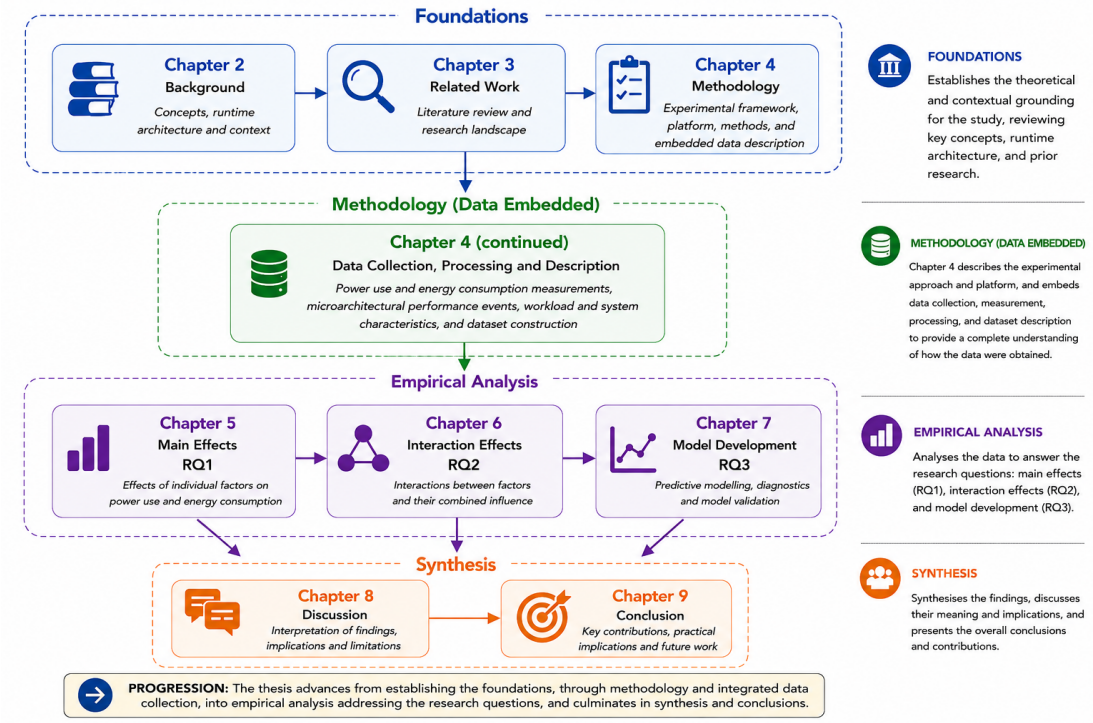


Figure 1.1: Structural overview of the dissertation showing the sequential progression from foundational theory and experimental methodology through empirical analysis addressing research questions, culminating in discussion and overall research conclusions.

Chapter 7 presents the results associated with Research Question **RQ3**, examining the extent to which variation in JRE JVM power use and energy consumption can be accounted for through regression modelling. Separate hierarchical regression models are developed for power use and energy consumption using pooled observations, followed by execution-mode-specific models for interpretive and tiered execution. Each model is evaluated through incremental variance decomposition, coefficient interpretation, regression diagnostics, sensitivity analysis, and stratified bootstrap estimation. The chapter concludes with cross-model comparative analyses examining differences in explanatory structure and coefficient relationships across the pooled, interpretive, and tiered model specifications.

Chapter 8 presents a discussion of the findings emerging from this dissertation. The chapter synthesises results relating to JRE source version effects, GNU GCC build toolchain compiler effects, JVM execution mode, garbage collector variant,

workload category, and microarchitectural performance event measures, situating the findings in relation to prior literature and practical implications.

Finally, Chapter 9 concludes the dissertation by summarising the principal findings, outlining the contributions of this research, discussing limitations, and identifying directions for future research.

From a graphical perspective, the dissertation is structured as illustrated in Figure 1.1.

Chapter 2

Background

This chapter establishes the technical background required to contextualise the empirical investigation presented in this dissertation. At the centre of the research is the analysis of the power use and energy consumption of the JRE and its associated JVM across multiple runtime configurations. To provide the necessary architectural context, the chapter first introduces the Java language and then examines the principal structural components of the JVM. This includes the class loader subsystem, runtime data areas, and execution engine, together with the interpretive and just-in-time compilation mechanisms through which Java applications are executed.

Because garbage collector configuration represents an important component of the empirical investigation, the chapter also reviews the principal garbage collection approaches used for heap memory management. This background provides the basis for interpreting differences observed between garbage collector variants and for relating those differences to their underlying design and execution characteristics.

The empirical analyses undertaken throughout this dissertation use benchmark workloads as controlled probes through which the effects of alternative JRE/JVM configurations can be examined. Accordingly, the chapter reviews the role of JVM benchmark suites and their development across successive generations, providing context for the workload suite adopted within this research. Finally, the chapter considers approaches for profiling power use and energy consumption, establishing

the measurement background required for the experimental methodology and subsequent empirical analyses.

2.1 The Java Language

The Java Programming Language is principally an object-oriented programming language, although it incorporates features associated with several programming paradigms. One way of conceptualising its structure is as a collection of nested cores, with each successive core extending the functionality of the preceding one. At its foundation is an imperative core that provides the constructs associated with imperative programming. This is encapsulated within a procedural core that supports the definition and execution of methods. The object-oriented core extends these capabilities through the constructs required for object-oriented programming, while an exception-handling core provides mechanisms for managing exceptional execution conditions. Finally, the concurrent core provides the language constructs required to support multithreaded execution. A comprehensive treatment of the static and dynamic semantics of the Java language is provided by Stark [34].

From an execution perspective, Java programs pass through distinct compile-time and runtime environments. During compilation, source code is translated into a representation suitable for subsequent execution. For programs targeting the Java Virtual Machine (JVM), this representation takes the form of Java bytecode contained within Java class files conforming to the Java class file specification [35]. Although Java source code provides the principal route to this representation, the source language need not itself be Java, as other programming languages can also compile to JVM-compatible bytecode.

For class files targeting the JVM, subsequent loading, linking, initialisation, and execution are undertaken within the Java Runtime Environment (JRE), which incorporates the JVM and its supporting runtime libraries [35]. This runtime abstraction separates JVM-targeted applications from many of the architecture-dependent details of the underlying hardware platform. The JVM itself is defined

as an abstract computing machine [35]. Like a physical processor, it operates according to a defined instruction set and executes sequences of instructions encoded within Java class files as JVM bytecode.

The Java class file format specified in [35] defines 201 bytecode instructions, together with three additional instructions reserved for internal JVM use. These instructions possess defined static and dynamic semantics that establish the operations permitted by the JVM instruction set. Several levels of abstraction can be applied when analysing these instructions. Dujmovic et al. [36], for example, presented a formal framework in which bytecode instructions are grouped at different abstraction levels to support analysis of different aspects of JVM execution. Such instruction groupings provide a means of characterising and comparing JVM workloads according to the operations they exercise. The JVM specification similarly provides an instruction-level categorisation comprising 11 groups: Constants, Loads, Stores, Stack, Math, Conversions, Comparisons, References, Control, Extended, and Reserved.

The bytecode instructions composing Java class methods can therefore be considered in terms of the JVM functionality exercised during program execution. Java bytecode workloads can be analysed from both static and dynamic perspectives, and their characteristics and execution effects have been investigated extensively within the literature [28, 37, 38, 39, 40, 16, 41, 42, 43, 44, 45, 46, 47, 48, 49]. Quantitative analysis of Java bytecode has contributed to both JVM optimisation and the modelling of Java applications. Previous work examining JVM execution characteristics has focused on specific aspects of the virtual machine, including heap operation, garbage collection, object creation, polymorphism, and type safety.

JVM implementations may execute Java bytecode through interpretation, just-in-time (JIT) compilation, native execution mechanisms, or combinations of interpretation and JIT compilation, including tiered execution. Within contemporary tiered JVM implementations, the interpreter provides the initial execution

path for Java bytecode and contributes runtime profiling information used to determine whether methods should subsequently be compiled. Lower compilation tiers may apply comparatively inexpensive optimisations through the C1 compiler, while more highly optimised code can be generated at later tiers through the C2 compiler. The interpreter nevertheless remains relevant beyond the initial stages of application execution. Through deoptimisation, execution can return from compiled code to an interpreted representation when assumptions supporting a compiled method are invalidated or when execution conditions otherwise require the compiled state to be reconstructed [50]. Deoptimisation therefore introduces additional runtime work because the execution state required by the interpreter must be reconstructed from the compiled representation.

The execution of Java bytecode occurs within the broader structure of the JRE. Its constituent components and their interactions determine how class loading, memory management, interpretation, compilation, and other runtime functions are performed. The following section therefore examines the principal components of the JRE and their respective roles. This background is important for the later empirical analyses because, although power use and energy consumption are measured at the system level and the JRE/JVM is treated principally as a black-box runtime system, the observed measurements arise while different runtime configurations exercise these underlying components in different ways.

2.2 The Java Runtime Environment

The Java Runtime Environment (JRE) is central to the research questions addressed within this dissertation, which examine variation in power use and energy consumption associated with alternative JRE/JVM configurations. Accordingly, this section provides an overview of the principal components and layers that constitute the JRE and establishes the architectural context required for the subsequent empirical analyses. More detailed treatments of the Java language, JVM specification, and runtime architecture are provided in [51, 52, 6].

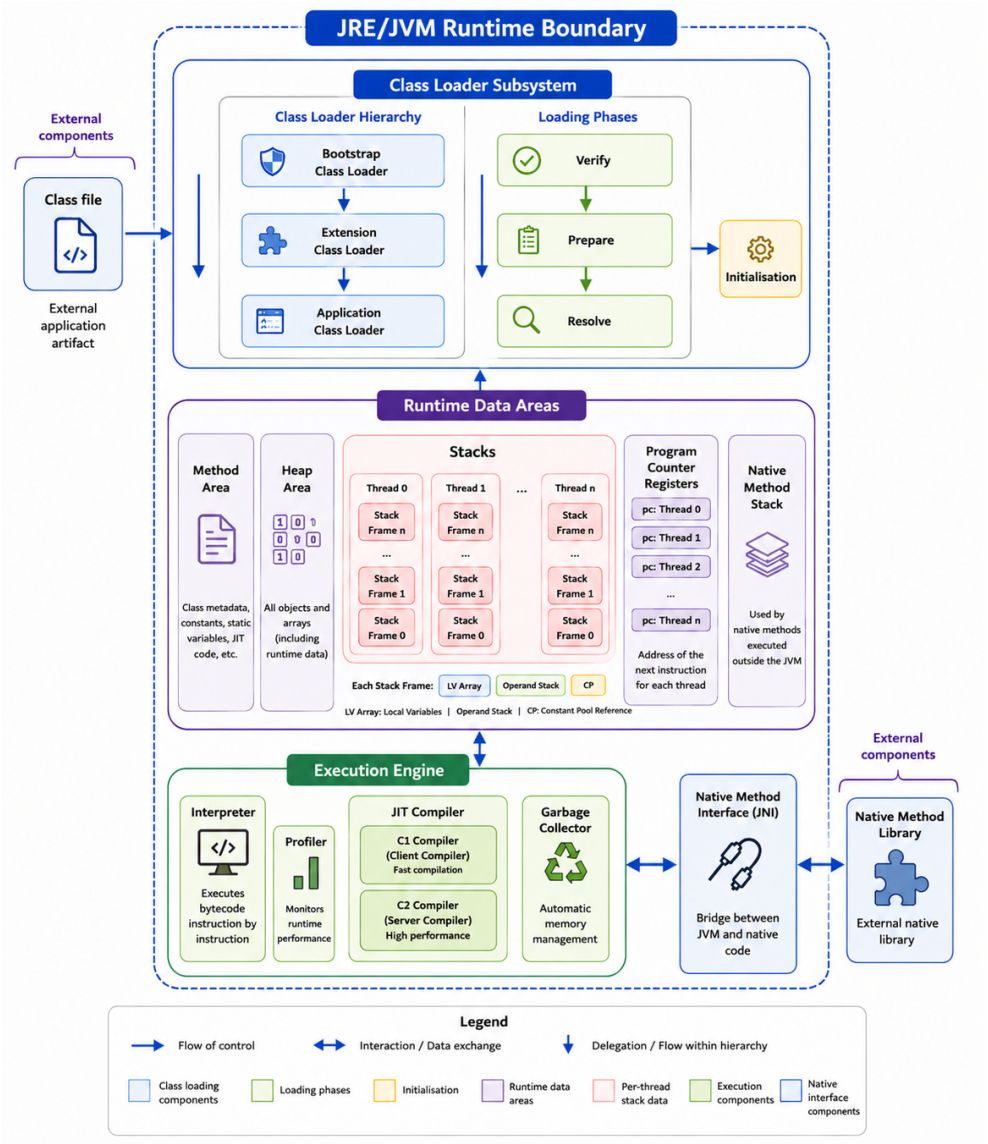


Figure 2.1: Schematic representation of the JRE/JVM runtime architecture, distinguishing components internal to the runtime environment from external application artefacts and native libraries. The figure illustrates the principal JVM subsystems, including class loading, runtime data areas, the execution engine, and the Java Native Interface (JNI), together with their interactions with external class files and native method libraries.

The JRE provides the software environment required for the execution of JVM applications and operates as a software layer above the underlying operating system and hardware architecture. It incorporates the Java Virtual Machine (JVM), runtime class libraries, and supporting components required for application execution. The JVM forms the execution core of this environment and provides the mechanisms through which JVM class files containing Java bytecode are

loaded, linked, initialised, and executed.

At a high level, the JVM comprises several principal functional components, including the Class Loader Subsystem, Runtime Data Areas, and Execution Engine. The wider runtime environment also provides mechanisms such as the Java Native Interface (JNI), through which JVM applications can interact with externally supplied native libraries. Application class files and native libraries can therefore be distinguished from the internal components responsible for their loading, execution, memory management, and interaction with the underlying platform.

Figure 2.1 provides a schematic overview of this architecture and distinguishes the internal JRE/JVM components from the external application artefacts and native libraries with which they interact. The following subsections examine these principal JVM components in greater detail, providing the technical foundation required for interpreting the runtime configuration factors investigated later in this dissertation.

2.2.1 The Class Loader Subsystem

The JVM Class Loader Subsystem prepares classes and interfaces for use during program execution and contributes to the construction of their runtime representations. The JVM specification distinguishes three principal stages in this process: loading, linking, and initialisation. Loading locates or generates the binary representation of a class or interface and creates its internal JVM representation. Linking subsequently verifies and prepares that representation and may resolve its symbolic references. Initialisation then executes the class or interface initialisation logic when required. The following subsections examine each of these stages in turn.

Class Loading

The JVM supports the dynamic loading of classes and interfaces as they are required during execution. For the OpenJDK versions considered within this dissertation, the runtime provides three principal built-in class loaders: the JVM bootstrap class loader, platform class loader, and system class loader [53]. These loaders participate in a delegation structure through which responsibility for locating and defining classes can be transferred between loaders.

The JVM bootstrap class loader is built into the virtual machine and is responsible for loading classes required during JVM startup and classes associated with critical runtime modules. Unlike the other built-in class loaders, it is not represented as an instance of the Java `ClassLoader` class and is typically represented by `null` through the `ClassLoader` API. The platform class loader makes Java platform classes available to the runtime, while the system class loader, also referred to as the application class loader, is typically responsible for loading application classes from the application class path or module path [53]. User-defined subclasses of `ClassLoader` may additionally be created where alternative class-loading mechanisms are required.

The terminology used for these loaders is particularly relevant to the OpenJDK versions examined in this dissertation. From JDK 9 onwards, the class loader previously referred to as the extension class loader was renamed the platform class loader. Accordingly, the term *platform class loader* is used here when referring to the JDK 9–14 runtime environments examined within this research.

During JVM startup, the bootstrap class loader participates in loading the core classes required before application execution can begin. For example, inspection using `java -verbose:class` shows that classes including `java.lang.Object`, `java.lang.String`, `java.lang.Class`, and `java.lang.ClassLoader`, together with their associated dependencies, are loaded before execution reaches the application `main` method. Preliminary instrumentation undertaken during development of the experimental environment similarly showed that approximately 600 classes

were loaded through the JVM bootstrap class loader before application-level execution in the configuration examined. This illustrates that substantial runtime infrastructure is established before the benchmark workload itself begins execution.

When a class loader locates the binary representation of a required class, the JVM derives an implementation-specific internal representation of that class within the method area. Before that representation can be used, the binary class-file structure is checked for conformance with the format defined by the JVM specification. This format checking is distinct from the semantic verification performed during linking. A malformed class-file representation can result in a `ClassFormatError`, while an unsupported class-file version can result in an `UnsupportedClassVersionError`. Once the class has been successfully created within the JVM, the linking process can proceed. The JVM specification permits aspects of linking, particularly resolution, to occur lazily when the relevant symbolic references are subsequently required.

Class Linking

Linking comprises three stages: verification, preparation, and resolution. Verification and preparation must occur before a class or interface is initialised, whereas resolution may be performed either during linking or lazily when a symbolic reference is first required during execution. Resolution therefore does not necessarily occur as a single complete stage immediately following preparation.

Class Linking: Verification

Verification determines whether the binary representation of a loaded class or interface satisfies the structural and semantic constraints defined by the JVM class-file specification. In particular, verification establishes type safety at both class and method levels and ensures that JVM bytecode satisfies the constraints required for safe execution.

The JVM specification formalises aspects of bytecode type checking using

a Prolog-style notation. An illustrative representation is provided in Program Segment 2.1. To determine whether a class is type-safe, verification considers the class identity, its defining loader, superclass hierarchy, implemented interfaces, field and method definitions, and the bytecode associated with individual methods. Verification also checks constraints arising from inheritance, including restrictions associated with `final` classes and methods.

Program Segment 2.1: Prolog-style notation for bytecode type checking.

```
classIsTypeSafe(Class) :-
    classClassName(Class, Name),
    classDefiningLoader(Class, L),
    superclassChain(Name, L, Chain),
    Chain \= [],
    classSuperClassName(Class, SuperclassName),
    loadedClass(SuperclassName, L, Superclass),
    classIsNotFinal(Superclass),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).
```

Successful verification provides several important assurances concerning the loaded class and its methods. It verifies, among other constraints, that inheritance rules associated with `final` classes and methods are respected; that class constant-pool entries satisfy the required static and structural constraints; that local-variable loads and stores are consistent with their expected types; that bytecode operands have appropriate types; and that operand-stack use remains consistent with the stack constraints established for the method. Verification therefore establishes that the bytecode satisfies the JVM's structural and type-safety requirements before the class proceeds towards execution.

Class Linking: Preparation

Following successful verification, the JVM undertakes preparation. Preparation creates the storage required for class-level `static` fields and assigns their initial default values, subject to values represented through applicable `ConstantValue`

attributes. Although preparation follows verification within the linking process, it remains distinct from class initialisation, during which explicitly defined static initialisers are subsequently executed.

Loading constraints may also be imposed during linking to preserve type consistency where classes with related symbolic references have been loaded through different class loaders. Violations of these constraints can result in a `LinkageError`. This distinction is important because a class or interface is identified at runtime not only by its binary name but also by the class loader that defines it. Consequently, classes with the same binary name may represent distinct runtime types when defined by different class loaders.

Class Linking: Resolution

Java class files contain symbolic references within their constant pools. During execution, instructions that refer to classes, interfaces, fields, methods, or other constants may require these symbolic references to be resolved before the referenced entity can be used. Resolution is the process through which a symbolic reference in the runtime constant pool is checked and converted into a runtime reference that can subsequently be used by the JVM.

A range of JVM instructions can require such resolution. These include instructions concerned with object and array creation or type checking, such as `new`, `anewarray`, `multianewarray`, `instanceof`, and `checkcast`; method-invocation instructions `invokeinterface`, `invokespecial`, `invokevirtual`, `invokestatic`, and `invokedynamic`; field-access instructions including `putfield`, `getfield`, `putstatic`, and `getstatic`; and constant-loading instructions such as `ldc` and `ldc_w`. Resolution can itself cause additional classes or interfaces to be loaded and linked where the symbolic reference identifies a type that has not yet been created within the JVM.

An `invokevirtual` instruction provides a useful example of this process. Program Segment 2.2 presents a Java `Harness` class containing an invocation of

the method `cwmtbr.getInt()`. The corresponding JVM bytecode is presented in Program Segment 2.3, while the associated class-file constant pool is presented in Program Segment 2.4.

Program Segment 2.2: Java method invocation code example.

```
public class Harness {
    public static int age;

    public static void main(String[] args) {
        ClassWithMethodToBeResolved cwmtbr
            = new ClassWithMethodToBeResolved();

        int value = cwmtbr.getInt();
    }
}
```

Program Segment 2.3: Java bytecode associated with Harness.java

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=1
      0: new           #2   // class ClassWithMethodToBeResolved
      3: dup
      4: invokespecial #3   // Method ClassWithMethodToBeResolved."
                                                        <init>":()V
      7: astore_1
      8: aload_1
      9: invokevirtual #4   // Method ClassWithMethodToBeResolved.
                                                        getInt:()I
     12: istore_2
     13: return
  LineNumberTable:
    line 5: 0
    line 6: 8
    line 7: 13
```

Program Segment 2.4: Constant pool associated with Harness.java

```
Constant pool:
   #1 = Methodref          #6.#17         // java/lang/Object."<init>":()V
   #2 = Class              #18            // ClassWithMethodToBeResolved
   #3 = Methodref          #2.#17         // ClassWithMethodToBeResolved.
                                                        "<init>":()V
   #4 = Methodref          #2.#19         // ClassWithMethodToBeResolved.
                                                          getInt:()I
   #5 = Class              #20            // Harness
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               age
   #8 = Utf8               I
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               main
  #14 = Utf8               ([Ljava/lang/String;)V
  #15 = Utf8               SourceFile
  #16 = Utf8               Harness.java
  #17 = NameAndType        #9:#10       // "<init>":()V
  #18 = Utf8               ClassWithMethodToBeResolved
  #19 = NameAndType        #22:#23      // getInt:()I
  #20 = Utf8               Harness
  #21 = Utf8               java/lang/Object
  #22 = Utf8               getInt
  #23 = Utf8               ()I
```

At bytecode offset 9 in Program Segment 2.2, the `invokevirtual` instruction references constant-pool entry `#4`. As shown in Program Segment 2.3, this entry is a `Methodref` containing references to entries `#2` and `#19`. Entry `#2` identifies the target class through the UTF-8 entry `#18`, resolving to `ClassWithMethodToBeResolved`. Entry `#19` is a `NameAndType` entry that ultimately identifies the method name `getInt` and descriptor `()I`. Together, these entries establish the symbolic method reference `ClassWithMethodToBeResolved.getInt:()I`.

If the referenced class has not already been loaded and linked, resolution can trigger the required loading and linking operations before the method reference can be used. Once the symbolic reference has been successfully resolved, the resulting runtime reference can be used during subsequent method selection and invocation.

JVM implementations can retain resolved information so that repeated execution of an instruction need not repeat the complete symbolic-resolution process. The first use of a previously unresolved reference can therefore require additional runtime work that is absent from subsequent uses.

Class Initialisation

Once a class or interface has been loaded and linked, it can be initialised when subsequently required by execution. Class initialisation is distinct from preparation. During preparation, storage for class-level fields is created and appropriate default values are assigned; during initialisation, the explicitly specified class or interface initialisation logic is applied.

A class or interface can contain at most one class initialisation method, identified within JVM bytecode by the special name `<clinit>`. Where present, this method incorporates the relevant static field initialisers and static initialisation blocks generated from the source program. The JVM invokes this method as part of the class-initialisation procedure.

At the bytecode level, important triggers for class initialisation include execution of the `new`, `getstatic`, `putstatic`, and `invokestatic` instructions where the referenced class or interface has not already been initialised. Class initialisation can also arise through other mechanisms defined by the JVM specification, including particular reflective operations, method-handle resolution and invocation, JVM startup, and the initialisation relationships between classes and their superclasses. Before a class is initialised, the JVM ensures that the required superclass initialisation has been completed in accordance with the JVM initialisation procedure.

Class initialisation therefore represents the final stage through which a loaded and linked class becomes ready for the execution contexts that require its initialised state.

2.2.2 The Runtime Data Areas

The loading, linking, and initialisation of Java class files populate and make use of several JVM runtime data areas. As depicted in Figure 2.1, these areas can be considered in terms of the `method area`, `heap`, per-thread `JVM stacks`, per-thread `program counter` registers, and `native method stacks`. These areas differ in both purpose and lifetime, with some shared across all JVM threads and others created separately for each thread of execution.

From a threading perspective, the `method area` and `heap` are shared across all executing JVM threads. In contrast, each JVM thread has its own `program counter` register and JVM `stack`. For a thread executing a non-native JVM method, the program counter identifies the JVM instruction currently being executed; its value is undefined while the thread is executing a native method. Each JVM stack contains a sequence of `frames`, with a new frame created for each method invocation and removed when that invocation completes.

The Method Area

As described previously, loading a class or interface causes the JVM to create an implementation-specific internal representation of that type. The method area stores per-class structures required during execution, including the runtime constant pool, field and method information, and the code associated with methods and constructors. These structures are shared across JVM threads, allowing methods associated with the same loaded class to be executed concurrently by different threads.

Each loaded class or interface also has an associated runtime constant pool. This contains the constants and symbolic references derived from the class-file constant pool, including references to classes, interfaces, fields, and methods. As outlined in the preceding section, symbolic references within this pool may subsequently be resolved into runtime references as they are required during execution.

The Heap

The JVM heap is the runtime data area from which memory for class instances and arrays is allocated. Object allocation can be initiated by several JVM bytecode instructions. The `new` instruction allocates memory for an instance of a class, while `newarray`, `anewarray`, and `multianewarray` allocate array objects containing primitive values, reference values, and multidimensional array structures, respectively.

For the `new` instruction, its operand identifies a symbolic class reference within the runtime constant pool. That reference is resolved to the required class type, with the associated class loaded and linked where necessary. If use of the class requires initialisation and the class has not already been initialised, the JVM performs the corresponding initialisation procedure before the newly created object is subsequently used.

The JVM specification deliberately leaves the physical representation of objects within the heap to the implementation. An implementation must provide the information required to determine an object's class and access its instance data, but the precise memory layout is not prescribed by the specification. In HotSpot-based JVM implementations, for example, object representations typically contain implementation-specific header information in addition to the storage required for instance fields. References stored within object fields or array elements can, in turn, identify other objects allocated within the heap.

The heap is managed by the JVM and supports automatic reclamation of memory occupied by objects that are no longer reachable. The garbage collection mechanisms through which this memory is reclaimed are examined in Section 2.2.4.

The Stacks

Each JVM thread has an associated JVM stack containing the `frames` created during method invocation. A new frame is pushed onto the thread's JVM stack when a method is invoked and is removed when that invocation completes, either

through a normal return or abrupt termination.

Each frame contains an array of `local variables`, an `operand stack`, and a reference to the runtime `constant pool` associated with the class containing the currently executing method. The local-variable array stores method parameters, locally defined values, and other values required during execution. The operand stack provides the working storage through which JVM bytecode instructions consume operands and produce intermediate results.

For an instance method, local-variable slot 0 contains a reference to the object on which the method was invoked, conventionally represented in Java source code by `this`. Method arguments occupy subsequent local-variable slots according to their respective types, followed by any additional local values required by the compiled method. The required size of the local-variable array is recorded within the method's `Code` attribute and remains fixed for that invocation.

The operand stack operates in a last-in-first-out manner. Bytecode instructions remove the operands they require from the top of the stack and, where applicable, push their results back onto it. For example, the `iadd` instruction removes two integer values from the operand stack and pushes their integer sum onto the stack. The maximum depth required by the operand stack is also recorded within the method's `Code` attribute. When a new frame is created, its operand stack is initially empty.

The Native Method Stacks

The JVM may also provide native method stacks to support methods implemented in languages other than Java and invoked through native interfaces. These stacks are distinct from the JVM stacks used for Java bytecode execution and are implementation-dependent in their organisation.

It is important to distinguish these specification-defined native method stacks from implementation-specific native memory regions used by a particular JVM implementation. In HotSpot, for example, native memory is also used for structures

associated with runtime-generated machine code. The Code Cache stores machine code produced by the C1 and C2 just-in-time compilers, together with other generated runtime code. Similarly, the template interpreter relies upon native machine-code routines generated as part of the HotSpot runtime implementation. These structures are relevant to the execution mechanisms examined later in this dissertation, but they are implementation-specific components rather than separately defined JVM runtime data areas.

2.2.3 The JVM Execution Engine

The JVM execution engine is responsible for executing the bytecode contained within loaded Java class files. In HotSpot-based JVM implementations, this execution capability is provided principally through an interpreter and one or more just-in-time (JIT) compilers. The wider JVM runtime also incorporates garbage collection mechanisms for reclaiming heap memory occupied by objects that are no longer reachable.

The JVM can execute bytecode using different execution modes. Under the default tiered execution strategy used by the OpenJDK versions examined in this dissertation, execution combines interpretation with the C1 and C2 just-in-time compilers. Tiered execution is organised across five conceptual execution levels, conventionally numbered from tier 0 to tier 4. Tier 0 corresponds to interpreted execution, tiers 1–3 are associated with different C1 compilation and profiling configurations, and tier 4 corresponds to highly optimised compilation by C2. Transitions between these levels are informed by runtime profiling information collected during execution.

Execution of a Java method initially occurs through interpretation unless the method has already been compiled or an alternative execution path applies. During interpreted and lower-tier compiled execution, the JVM collects profiling information that can subsequently inform compilation decisions. This information includes method invocation and loop activity, together with other runtime ob-

servations used by the tiered compilation policy to determine whether a method should be scheduled for compilation at a higher tier.

The JVM Interpretive Environment

Java interpreters can be implemented using several instruction-dispatch strategies. A principal distinction between these strategies concerns how control is transferred from one bytecode instruction implementation to the next. Common approaches include switch-based dispatch, direct threading, direct call threading, and inline threading [54, 55].

In a switch-based interpreter, a central dispatch loop examines the current opcode and transfers control to the corresponding instruction implementation. Once that instruction has completed, execution returns to the dispatch mechanism so that the next opcode can be selected. The simplicity of this approach makes it straightforward to implement, but the repeated return to a central dispatch point introduces additional control-transfer overhead.

Direct threading removes the central opcode-selection switch by associating bytecode instructions with the addresses of their corresponding instruction handlers. At the completion of one instruction handler, control is transferred directly to the handler associated with the next instruction. The instruction handlers therefore participate directly in maintaining the sequence of interpreted execution rather than repeatedly returning control to a central switch-based dispatcher [56].

Direct call threading differs from direct threading in that the translated instruction sequence is represented as a sequence of native calls to instruction handlers. Each call invokes the implementation associated with a particular bytecode operation before control returns to the surrounding threaded sequence. This retains reusable instruction handlers while reducing dependence on a central opcode-dispatch loop.

Inline threading reduces dispatch overhead further by replacing calls or transfers to shared instruction handlers with inlined native instruction sequences

corresponding to the bytecodes being executed. Where several bytecodes occur within the same basic block, their implementations can therefore be combined into a longer native instruction sequence, reducing the number of dispatch operations required between individual bytecodes. Casey et al. [54] and Gagnon et al. [55] provide more detailed treatments of these interpreter implementation strategies.

Instruction dispatch has historically represented an important source of interpreter overhead. HotSpot addresses this problem through its template interpreter, in which machine-code templates are generated for individual JVM bytecodes during JVM initialisation. These templates provide architecture-specific implementations of bytecode operations and are connected through the interpreter's dispatch mechanism. More complex runtime operations can transfer control from these generated templates to supporting runtime routines implemented elsewhere within HotSpot.

The JVM C1 and C2 Compilers

Tiered execution combines the interpreter with two principal HotSpot just-in-time compilers: C1 and C2. Historically, C1 originated as the client compiler and was designed to provide relatively fast compilation with modest compilation overhead, making it suitable for applications in which startup and responsiveness were important. C2 originated as the server compiler and applies more extensive optimisation, accepting greater compilation cost in exchange for more highly optimised generated machine code [57].

Within C1, bytecode is transformed through intermediate representations that support a range of compiler optimisations before native machine code is generated. Its high-level intermediate representation supports optimisations including constant folding, value numbering, method inlining, and null-check elimination, while its lower-level intermediate representation is subsequently used during register allocation and machine-code generation.

Under tiered execution, C1 participates across tiers 1, 2, and 3. These tiers

differ in the extent of compilation and profiling retained within the generated code. Tier 4 is associated with C2 compilation and represents the most highly optimised execution level within the conventional HotSpot tiered compilation structure. Methods do not necessarily progress through every tier sequentially. Instead, compilation decisions are governed by the tiered compilation policy, which considers runtime activity, profiling requirements, compiler availability, and the current state of the compilation queues.

Compilation is performed asynchronously by compiler threads. When a method satisfies the conditions established by the tiered compilation policy, a compilation task can be placed on an appropriate compiler queue. C1 or C2 subsequently compiles the method into native machine code for the underlying processor architecture. Once compilation is complete, execution can transfer from the interpreted or previously compiled representation to the newly generated machine code.

The native machine code generated by the JIT compilers is stored within HotSpot's Code Cache [58]. For the OpenJDK versions examined in this dissertation, the segmented Code Cache distinguishes between non-method code, profiled compiled code, and non-profiled compiled code. The non-method segment contains JVM-generated code that is not associated directly with Java methods, while the profiled and non-profiled segments principally accommodate compiled methods produced at different stages of tiered execution.

The size of the Code Cache can be controlled through several JVM runtime options. The principal options considered here include:

- `-XX:InitialCodeCacheSize`, which specifies the initial Code Cache size;
- `-XX:ReservedCodeCacheSize`, which specifies the maximum reserved Code Cache size;
- `-XX:NonNMethodCodeHeapSize`, which controls the non-method code heap;
- `-XX:ProfiledCodeHeapSize`, which controls the profiled code heap; and
- `-XX:NonProfiledCodeHeapSize`, which controls the non-profiled code heap.

These memory regions are implementation-specific HotSpot structures and should therefore be distinguished from the runtime data areas defined abstractly by the JVM specification.

Compiled methods stored within the Code Cache can subsequently become invalid, non-entrant, or otherwise eligible for reclamation. HotSpot maintains mechanisms for removing obsolete compiled code so that Code Cache space can be reused. This process is distinct from ordinary Java heap garbage collection and should not be interpreted as the JVM garbage collector directly managing the Code Cache. Similarly, deoptimisation refers to the process through which execution returns from compiled code to a less optimised representation when assumptions supporting the compiled code are no longer valid; it is not simply a mechanism for removing methods that have become less frequently executed.

The path followed by a method through tiered compilation is therefore not fixed. A method may be interpreted, compiled by C1 with differing levels of profiling, and subsequently compiled by C2 where its execution profile and the tiered compilation policy justify further optimisation. Conversely, queue state and profiling requirements can result in alternative paths through the available C1 tiers before C2 compilation occurs [58]. This adaptive structure allows HotSpot to balance the cost of compilation against the potential benefit of executing increasingly optimised native code.

The distinction between C1 and C2 reflects their different optimisation objectives. C1 prioritises relatively rapid compilation with lower compilation overhead, whereas C2 applies more extensive optimisation at greater compilation cost. Across the development of HotSpot, both compilers have incorporated numerous optimisation techniques intended to improve the generated machine code. These techniques can be grouped according to their optimisation strategy, as discussed by Ivanov et al. [59], with a more extensive enumeration provided by Rose et al. [60]. The principal optimisation techniques relevant to the present discussion are summarised below:

- **Delayed compilation:** tiered compilation and on-stack replacement;
- **Proof-based techniques:** exact type inference, memory value inference, constant folding, null-check elimination, algebraic simplification, and common sub-expression elimination;
- **Flow-sensitive rewrites:** conditional constant propagation, flow-carried type narrowing, and dead code elimination;
- **Language-specific techniques:** class hierarchy analysis, devirtualisation, symbolic constant propagation, autobox elimination, escape analysis, and lock elision and fusion;
- **Speculative profile-based techniques:** optimistic nullness and type assertions, optimistic type and array-length strengthening, untaken branch pruning, optimistic N-morphic inlining, branch-frequency prediction, and call-frequency prediction;
- **Memory and placement transformations:** expression hoisting, expression sinking, redundant store elimination, and adjacent store fusion;
- **Loop transformations:** loop unrolling, loop peeling, safepoint elimination, iteration-range splitting, range-check elimination, and loop vectorisation;
- **Global code shaping:** inlining (graph integration), global code motion, heat-based code layout, switch balancing, and throw inlining;
- **Control-flow and code-generation transformations:** local code scheduling, local code bundling, delay-slot filling, graph-colouring register allocation, linear-scan register allocation, live-range splitting, copy coalescing, constant splitting, copy removal, address-mode matching, instruction peephole optimisation, and DFA-based code generation.

2.2.4 Garbage Collection

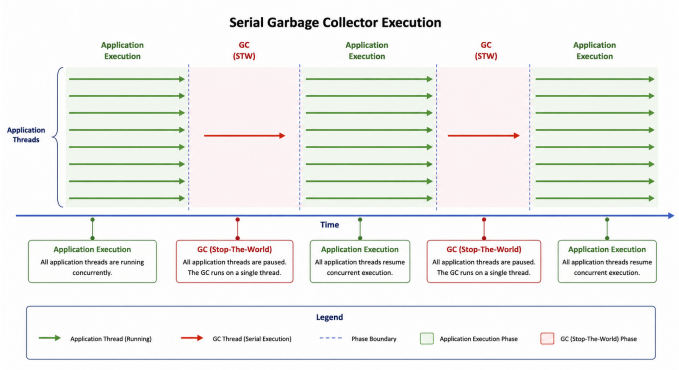
The JVM provides automatic management of memory allocated within the heap. Objects and arrays created during application execution are allocated heap memory through JVM bytecode instructions including `new`, `newarray`, `anewarray`, and

`multianewarray`. These instructions allocate memory for class instances, arrays of primitive types, arrays of reference types, and multidimensional arrays, respectively. Unlike object allocation, no JVM bytecode instruction explicitly releases the heap memory occupied by an object. Instead, reclamation is performed automatically by the garbage collector when objects are determined to be no longer reachable [6].

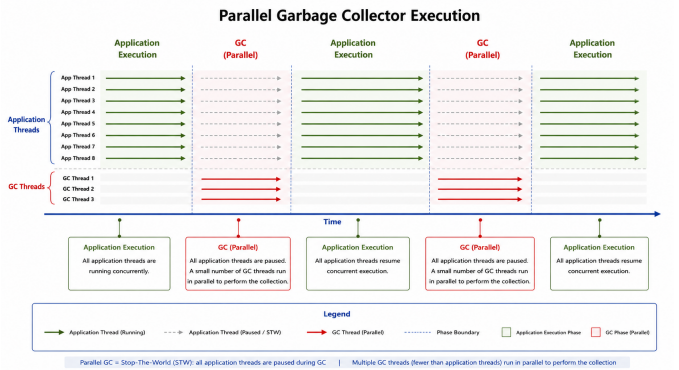
Figure 2.2 provides high-level representations of the execution characteristics associated with the principal garbage collector variants considered within this research. The diagrams distinguish periods of application execution from garbage collection activity and illustrate the differing use of serial, parallel, stop-the-world, and concurrent collection phases.

The empirical investigation examines five Garbage Collector Variants: the Serial Collector (SGC), Parallel Collector (PGC), Concurrent Mark Sweep Collector (CGC), the corresponding configuration incorporating parallel young-generation collection (CPGC), and the Garbage-First Collector (G1GC). These collectors differ substantially in how garbage collection work is scheduled relative to application execution and in the number of collector threads used to perform that work.

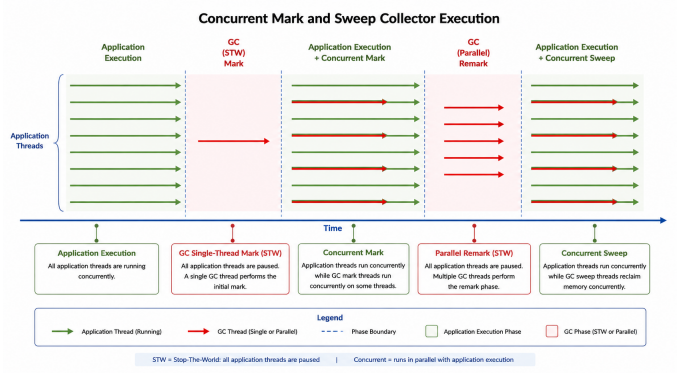
The SGC performs garbage collection using a single collector thread and suspends application threads while collection is undertaken. The PGC similarly performs collection during stop-the-world pauses, but distributes garbage collection work across multiple collector threads operating in parallel. The Concurrent Mark Sweep collector differs from these approaches by performing substantial portions of old-generation marking and sweeping concurrently with application execution, although stop-the-world pauses remain necessary for particular phases. G1GC similarly combines stop-the-world and concurrent activity but organises the heap into multiple independently managed regions and selects sets of those regions for reclamation.



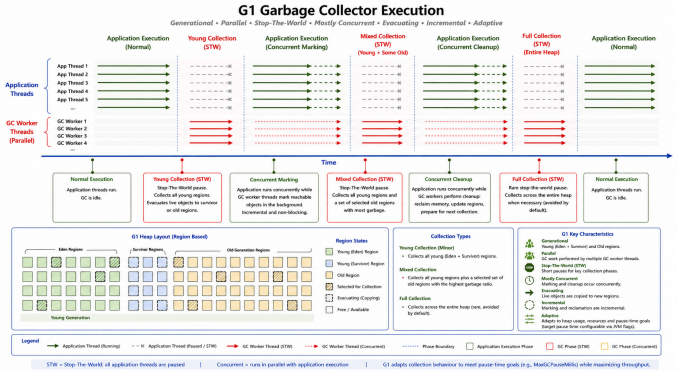
(a) The Serial Collector



(b) The Parallel Collector



(c) The Concurrent Collector



(d) The Garbage-First Collector

Figure 2.2: High-level depictions of the thread structure associated with serial, parallel, concurrent mark-and-sweep, and garbage-first garbage collectors.

Several characteristics can therefore be used to distinguish garbage collector designs:

1. **Generational**, where heap management distinguishes between young and old object generations;
2. **Parallel**, where multiple garbage collector threads perform collection work simultaneously;
3. **Concurrent**, where particular garbage collection operations execute concurrently with application threads;
4. **Stop-the-world**, where application threads are suspended while a garbage collection phase is performed;
5. **Evacuating or compacting**, where live objects are relocated as part of space reclamation or heap compaction; and
6. **Incremental**, where reclamation work is divided across multiple collection operations rather than requiring the entire heap to be reclaimed within a single collection pause.

These characteristics do not apply uniformly to every phase of every collector. A collector can, for example, perform some operations concurrently while requiring stop-the-world execution for others, or use evacuation within one generation while applying a different reclamation strategy within another.

The generational collectors considered within this research distinguish between young and old object generations. In conventional generational heap organisations, newly allocated objects enter an *Eden* space within the young generation. Objects surviving young-generation collections can subsequently be copied into survivor spaces and, after surviving sufficient collection cycles, promoted to the old generation. G1GC retains the conceptual distinction between young and old generations but implements it using independently managed heap regions that can be assigned dynamically to Eden, survivor, or old-generation roles.

Garbage collection relies upon reachability analysis to determine which objects remain accessible to the executing application. This analysis begins from a set of

garbage collection roots that can include references held within active JVM stack frames, static fields, JNI handles, and JVM-internal structures. The collector traverses references reachable from these roots to identify the set of live objects. Objects that cannot be reached through this traversal can subsequently become eligible for reclamation.

The SGC considered within this research is a generational, serial, stop-the-world collector. Collection work is performed by a single garbage collector thread while application threads remain suspended. Young-generation collection involves evacuation of surviving objects between heap spaces, while full collections can compact the heap. The Serial Collector is principally intended for comparatively small heaps and environments in which the additional coordination required by parallel collection would provide little advantage [61].

The PGC is also a generational, stop-the-world collector, but differs from the SGC through its use of multiple garbage collector threads. Collection work can therefore be performed in parallel while application threads remain suspended. Parallel collection is particularly suited to multiprocessor systems and applications for which overall throughput is prioritised over minimising individual garbage collection pauses [61].

The Concurrent Mark Sweep collector was designed to reduce the duration of application pauses by performing substantial portions of old-generation collection concurrently with application execution. A CMS collection cycle includes short stop-the-world initial-mark and remark phases, separated by concurrent marking activity, followed by concurrent sweeping of unreachable objects. Application threads can therefore execute while the concurrent phases are being performed. Unlike compacting collectors, the normal CMS old-generation collection cycle does not relocate live objects to compact the heap, and consequently fragmentation can accumulate over time [61].

The G1GC is a generational, parallel, mostly concurrent, stop-the-world, evacuating, and incremental collector. Rather than dividing the heap into large

contiguous young and old spaces, G1GC partitions the heap into multiple similarly sized regions that can assume Eden, survivor, or old-generation roles. Young collections evacuate live objects from the young-generation regions during stop-the-world pauses. Following concurrent marking of the old generation, mixed collections can additionally include selected old-generation regions containing comparatively large amounts of reclaimable space.

G1GC therefore performs reclamation incrementally across selected region sets rather than requiring routine whole-heap collection. Its region selection and collection-set sizing are influenced by current heap occupancy, previous collection measurements, available resources, and configured pause-time objectives. A full collection remains available as a fallback mechanism when normal G1GC reclamation cannot provide sufficient memory, but it does not represent the normal collection path. G1GC can therefore adapt its collection strategy to balance reclamation requirements against application pause-time objectives.

2.2.5 Standardised Specification, Implementation Flexibility

At the core of the JVM specification are the Java class file format and the instruction set recognised and executed by the virtual machine. The specification defines both the structure of the class file and the semantics associated with individual JVM instructions, thereby establishing a common execution model against which conforming JVM implementations can be developed. These publicly defined requirements allow the meaning and expected effects of JVM bytecode to be understood independently of the internal implementation used to execute it.

The JVM specification deliberately leaves many implementation decisions unspecified. A conforming JVM may, for example, execute bytecode through interpretation, just-in-time compilation, or a combination of both. Similarly, the physical organisation of the heap, the representation and storage of class metadata, the implementation of the method area, and the mechanisms used for

memory management can differ substantially between JVM implementations. The specification defines the required functionality and observable execution semantics without prescribing a single internal architecture through which those requirements must be realised.

This separation between specification and implementation is fundamental to the JVM design. Implementers retain considerable freedom in selecting internal data structures, execution mechanisms, memory-management strategies, and optimisation techniques, provided that the resulting JVM conforms to the requirements established by the specification. Consequently, two JVM implementations can differ substantially in their internal construction while presenting the same specified execution model to JVM-targeted applications.

2.2.6 OpenJDK Enhancements Across Versions 9–14

Since the first release of Java in 1995, the Java language and its associated JVM specification have been implemented by numerous vendors, including Sun Microsystems, Oracle, IBM, Microsoft, and Azul. Although these implementations conform to the common requirements established by the Java and JVM specifications, their underlying runtime implementations have evolved substantially over successive releases.

The Sun JDK codebase began its transition to open-source development in 2006, leading to the establishment of the Open Java Development Kit (OpenJDK) project. OpenJDK subsequently became the principal open-source implementation from which successive JDK releases have been developed [62]. An important mechanism through which significant changes to the JDK are proposed and documented is the JDK Enhancement Proposal (JEP) process. The OpenJDK JEP index records enhancement proposals spanning language, runtime, compiler, garbage collection, library, tooling, security, and other platform components [63]. JEPs therefore provide an important record of significant changes introduced across successive JDK releases, together with the motivation and design rationale

underlying those changes.

Although OpenJDK development has continued substantially beyond version 14, the following discussion concentrates principally on enhancements affecting OpenJDK versions 9 through 14. This range corresponds directly to the JRE Version factor examined empirically within this dissertation. OpenJDK version 9 represents the earliest JRE Version included within the experimental design and provides the reference point against which subsequent versions are compared. The restriction to versions 9 through 14 therefore reflects the defined experimental scope of the research rather than an assumption that later OpenJDK releases introduced no changes relevant to power use or energy consumption.

Subsequent OpenJDK releases have continued to modify HotSpot runtime, compilation, garbage collection, memory-management, and concurrency mechanisms. Such modifications may alter the relationships identified within the version range examined here. The extent to which the findings reported within this dissertation persist across later OpenJDK releases is therefore a question of temporal external validity rather than an assumption that JVM implementation characteristics remain invariant beyond OpenJDK version 14.

JEPs are associated with particular JDK components and, where appropriate, more specific subcomponents. Examples include changes associated with core libraries, language and JVM specifications, development tools, security, client libraries, and HotSpot. HotSpot-related proposals can additionally concern areas such as garbage collection, runtime implementation, JIT compilation, and JVM tooling. This classification is particularly relevant to the present research because changes within these components can alter the runtime mechanisms exercised during application execution.

Several enhancements introduced between OpenJDK versions 10 and 14 are particularly relevant to the runtime characteristics examined within this dissertation. In OpenJDK version 10, JEP 307 introduced parallel execution of G1 full garbage collections [64]. Prior to this enhancement, the G1 full-collection

fallback path did not exploit the same degree of parallelism available within other G1 collection phases. JEP 307 therefore modified the execution characteristics of full collections by allowing multiple garbage collector threads to participate in this work.

OpenJDK version 12 introduced two further G1 enhancements. JEP 344 introduced abortable mixed collections [65]. Under this approach, the mixed-collection set is divided into mandatory and optional components, allowing optional collection work to be curtailed where continuing that work would risk exceeding the targeted pause duration. JEP 346 introduced mechanisms through which G1 can more promptly return unused committed heap memory to the operating system, particularly during periods of reduced application activity [66]. This enhancement was motivated in part by runtime environments in which retaining committed but unused memory imposes an unnecessary resource cost.

G1 was further modified in OpenJDK version 14 through JEP 345, which introduced NUMA-aware memory allocation [67]. The enhancement improved G1 memory placement on systems implementing non-uniform memory access by increasing awareness of the relationship between heap regions and the processor nodes from which they are accessed. Although the experimental platform used within this dissertation does not implement a multi-socket NUMA architecture, the enhancement illustrates the continuing modification of garbage collection mechanisms across the JRE Version range examined.

A further OpenJDK version 14 change of direct relevance to garbage collection was JEP 363, which removed the Concurrent Mark Sweep (CMS) garbage collector from HotSpot. CMS had previously been deprecated in OpenJDK version 9 but remained available in a number of OpenJDK 14 development builds before its removal was integrated later in the OpenJDK 14 development cycle. Consequently, the availability of CMS was dependent upon the specific OpenJDK 14 build being used rather than simply the major version designation. By OpenJDK version 15, CMS was no longer available and the previously associated JVM selection option

had also been removed. This progression illustrates that differences between JRE Versions can include not only modifications to existing runtime components but also the staged deprecation and eventual removal of previously available implementation mechanisms.

Changes to class-data sharing also occurred across the investigated version range. JEP 310 extended Class Data Sharing (CDS) in OpenJDK version 10 through Application Class Data Sharing (AppCDS), allowing application classes to be incorporated into shared archives [68]. This extended the potential benefits of class-data sharing beyond the platform classes traditionally stored within CDS archives. The mechanisms associated with AppCDS subsequently evolved, and the `-XX:+UseAppCDS` option became obsolete from JDK 11 as support for application and platform classes became integrated into the standard CDS mechanisms.

OpenJDK version 12 introduced JEP 341, through which default CDS archives could be generated as part of the JDK build process [69]. OpenJDK version 13 subsequently introduced JEP 350, Dynamic CDS Archives, extending this mechanism by supporting the creation of additional class-data archives following application execution [70]. Collectively, these enhancements illustrate how successive OpenJDK versions can modify class-loading and runtime initialisation mechanisms even where the underlying JVM bytecode executed by an application remains unchanged.

Table 2.1 summarises selected JDK Enhancement Proposals delivered across OpenJDK versions 10 through 14. The table is not intended to provide an exhaustive catalogue of all modifications introduced within these releases; rather, it identifies representative language, runtime, memory-management, and HotSpot enhancements that illustrate the continuing evolution of the platform across the JRE Version range examined within this dissertation. A comprehensive index of JDK Enhancement Proposals is provided in [63].

OpenJDK development has continued beyond the version range examined within this research, with subsequent releases introducing further changes to JVM

Table 2.1: Selected JDK Enhancement Proposals delivered across OpenJDK versions 10 through 14.

OpenJDK 10		OpenJDK 11	
JEP286:	Local-Variable Type Inference	**JEP309**:	Dynamic Class-File Constants
JEP304:	Garbage-Collector Interface	**JEP323**:	Local-Variable Syntax for Lambda Parameters
JEP307:	Parallel Full GC for G1	**JEP331**:	Low-Overhead Heap Profiling
JEP310:	Application Class-Data Sharing		
JEP312:	Thread-Local Handshakes		
JEP316:	Heap Allocation on Alternative Memory Devices		
JEP319:	Root Certificates		
OpenJDK 12		**OpenJDK 13**	
JEP325:	Switch Expressions (Preview)	**JEP350**:	Dynamic CDS Archives
JEP341:	Default CDS Archives	**JEP354**:	Switch Expressions (Second Preview)
JEP344:	Abortable Mixed Collections for G1		
JEP346:	Promptly Return Unused Committed Memory from G1		
OpenJDK 14			
JEP305:	Pattern Matching for `instanceof` (Preview)		
JEP345:	NUMA-Aware Memory Allocation for G1		
JEP352:	Non-Volatile Mapped Byte Buffers		
JEP361:	Switch Expressions		
JEP363:	Remove the Concurrent Mark Sweep (CMS) Garbage Collector		

execution, compilation, garbage collection, memory management, and concurrency mechanisms. These later developments fall outside the experimentally assessed OpenJDK version 9 through 14 range. Accordingly, the power use and energy consumption relationships identified within this dissertation should be interpreted as applying to the investigated JRE Versions and experimental context rather than as evidence that equivalent relationships necessarily persist across later OpenJDK releases.

2.3 JVM Benchmark Suites

The JRE and its associated JVM constitute a mature runtime environment that has received considerable research attention, particularly in relation to computational performance. Empirical evaluation of JVM implementations has consequently relied extensively upon controlled benchmark suites. Several suites have featured prominently within the published literature, including the JOlden Benchmark Suite [71, 72, 73], Java Grande Benchmark Suite [74], DaCapo Benchmark Suite

[31], and SciMark Benchmark Suite [75]. Benchmark suites developed by the Standard Performance Evaluation Corporation (SPEC) have also been widely employed, including SPECjbb [76], SPECjEnterprise [77], and SPECjvm [78]. More recently, the Renaissance Benchmark Suite has provided workloads intended to exercise characteristics of contemporary JVM applications and frameworks [5]. The Computer Language Benchmarks Game (CLBG) has also been used for comparative language and runtime evaluation, although it has featured less prominently within peer-reviewed JVM research [79].

An important consideration in the development and selection of benchmark suites is the extent to which their constituent workloads represent the applications and execution characteristics encountered in practice. Benchmark suites such as those developed by SPEC sought to incorporate workloads representative of application execution on contemporary JVM implementations. This represented a progression away from exclusive reliance upon synthetic workloads, computational kernels, and micro-benchmarks, which can isolate particular runtime characteristics but may provide only limited representation of the complexity associated with complete applications.

Earlier benchmark collections nevertheless provided valuable workloads representing particular computational problems. The JOlden collection, for example, included the `health` workload, which models aspects of a healthcare system, and `Em3d`, which models the propagation of electromagnetic waves. Such workloads provide controlled and repeatable execution contexts, but their representativeness of the wider range of applications executed on contemporary JVM implementations is necessarily limited. Benchmark-suite development has consequently progressed towards collections containing increasingly diverse applications, libraries, frameworks, programming paradigms, and execution characteristics.

This progression is also evident within the Java Grande Benchmark Suite, whose developers sought to provide workloads representative of large-scale applications requiring substantial computational resources, particularly scientific and numerical

workloads. More recently, the Renaissance Benchmark Suite was developed to exercise aspects of contemporary JVM execution that were less extensively represented within earlier suites, including parallelism, concurrency, functional programming abstractions, and applications constructed using modern software frameworks [5]. The original Renaissance study demonstrated that its workloads occupied portions of the measured workload-characteristic space not covered to the same extent by DaCapo, SPECjvm2008, and ScalaBench [5].

The empirical investigation undertaken within this dissertation uses the workloads composing the Renaissance Benchmark Suite as controlled execution workloads through which alternative JRE/JVM configurations are evaluated. Several characteristics of the suite support this choice. First, Renaissance was designed specifically to increase the representation of parallel and concurrent JVM workloads and incorporates applications exercising multiple forms of parallel and concurrent programming [5]. This is particularly relevant to the multicore ARM platform employed within this research, as the workloads provide opportunities for the JVM, application threads, compiler threads, and garbage collector threads to exercise the underlying processor cores under differing execution conditions.

Second, the Renaissance suite exercises programming abstractions and frameworks associated with contemporary JVM application development. These include workloads involving futures, streams, actor-based execution, fork/join parallelism, transactional memory, data-processing frameworks, database systems, and machine-learning algorithms. The suite therefore provides a broader range of execution characteristics than collections focused principally upon individual computational kernels or narrowly defined application classes.

Third, and importantly for the present research, Renaissance is not restricted to applications written in the Java programming language. The suite contains workloads implemented using both Java and other JVM-targeted languages, particularly Scala, together with applications that make extensive use of libraries and frameworks implemented across these languages. Regardless of their source

language, these workloads ultimately execute through JVM-compatible representations and exercise the same underlying JRE/JVM infrastructure. This enables the experimental investigation to examine the JRE/JVM across a more heterogeneous set of application implementations than would be obtained from a suite restricted solely to Java source programs. The Renaissance project itself describes the suite as being written in both Java and Scala and as incorporating modern Java and Scala frameworks [5].

The inclusion of non-Java JVM workloads is particularly relevant because the JVM has developed into an execution platform for multiple programming languages rather than functioning exclusively as the runtime target of Java source code. Different source languages and frameworks can generate substantially different patterns of JVM bytecode, object allocation, method invocation, concurrency, and library interaction while ultimately executing through the same JVM implementation. Including both Java and non-Java workloads therefore broadens the range of runtime activity represented within the experimental dataset without changing the underlying execution platform being investigated.

Finally, the workload analysis undertaken by Prokopec et al. [5] demonstrated that Renaissance spans a diverse space of concurrency-related and application characteristics and extends the coverage provided by established JVM suites including DaCapo, SPECjvm2008, and ScalaBench. This diversity is important within the present research because the objective is not to characterise the power use and energy consumption of a single application, but to determine how alternative JRE/JVM configurations relate to power use and energy consumption across a heterogeneous collection of workloads. Renaissance therefore provides the controlled workload diversity required for the subsequent examination of JRE Version, Build Toolchain Compiler Version, JVM Execution Mode, Garbage Collector Variant, Workload Category, and Microarchitectural Performance Event Measures.

2.4 Power Use and Energy Consumption Profiling

Power use, current draw, and voltage can be measured empirically at several points within the electrical supply path of a computer system. Goel et al. [80] distinguish three principal measurement locations: at the wall outlet supplying the complete system, at individual power rails within the system, and close to the processor through measurements associated with its voltage-regulation circuitry. Each approach provides a different level of measurement granularity and introduces corresponding advantages and limitations.

Measurement at the wall outlet captures the electrical demand of the complete system under test, including the processor, memory, storage, peripheral devices, and losses associated with the power supply. This provides a useful system-level representation of power use but does not permit the measured demand to be attributed directly to individual hardware components. The measurement instrument is positioned between the external power source and the system under test, allowing voltage and current to be sampled and corresponding power measurements to be obtained. Goel et al. [81] note that this approach remains useful where trends in system power use are of greater interest than the absolute power attributable to individual processor or memory components.

Measurements taken from individual system power rails provide greater granularity at the hardware-level because the electrical supply to particular subsystems can be observed separately. This increased granularity is achieved at the cost of greater experimental complexity, as access to the relevant power rails and appropriate sensing hardware is required. Measurements made at different locations within the power-delivery path must also account for losses introduced by intervening power-conversion components. Consequently, measurements taken at the external power source and those taken on internal DC power rails do not necessarily represent equivalent quantities.

A still finer level of measurement can be obtained close to the processor voltage regulator, allowing processor-specific electrical demand to be isolated more directly. Such measurements can provide high measurement fidelity for processor power but are considerably more intrusive because they require physical access to the relevant circuitry and provide limited information about the electrical demand of the remainder of the system [82]. The appropriate measurement location therefore depends upon whether the objective is to characterise complete-system power use or the contribution of a particular hardware component.

The physical measurement approaches described by Goel et al. [81, 80] remain important because they provide direct observations of electrical quantities rather than estimates derived exclusively from software or hardware performance models. Notwithstanding differences in measurement location, instrumentation, sampling rate, and precision, direct physical measurement is generally regarded as an appropriate approach where experimentally observed power profiles are required [83]. Where voltage and current are sampled over time, instantaneous power can be determined from their product and energy consumption can subsequently be obtained by integrating power over the duration of the measured execution.

The Standard Performance Evaluation Corporation (SPEC) has developed methodological guidance for integrating power measurement with application benchmarking [84]. Its methodology emphasises the explicit definition of the system under test (SUT), the components included within the measured power boundary, the characteristics of the measurement instrumentation, environmental conditions, measurement metrics, automation, validation, and the consistent reporting of experimental results. These principles are intended to support repeatable and verifiable measurements across systems and experimental environments.

SPEC also provides the SPEC Power and Temperature Daemon (SPEC PTDaemon), which standardises communication between a benchmark harness and supported power analysers or temperature sensors. This abstraction allows measurement instrumentation to be controlled independently of the benchmark work-

load while maintaining synchronisation between workload execution and the corresponding environmental measurements. The SPECpower_ssj2008 benchmark, for example, applies this methodology to single-address-space server systems or homogeneous collections of such systems [85]. Within this methodology, system-level electrical measurements are obtained at the power input to the SUT, while the precision, resolution, and sampling characteristics of the resulting measurements remain dependent upon the specification of the power analyser employed.

Research involving embedded systems has additionally examined the construction of lower-cost measurement platforms capable of providing higher-frequency current and voltage sampling. Mao [86], for example, developed a low-cost measurement system for ARM-based embedded platforms using a Teensy 3.1 development board and an INA122 instrumentation amplifier. The system was evaluated using Raspberry Pi devices and was designed to capture current measurements at substantially higher sampling frequencies than many conventional low-cost power monitors. A principal limitation identified within this design concerned transfer of the captured measurements from the sampling device to a host system. Data transmission over the serial interface temporarily interrupted analogue-to-digital conversion, creating gaps within the captured sample stream. Mao therefore highlighted the importance of considering not only nominal sampling frequency but also the ability of the measurement architecture to sustain continuous data acquisition.

Related work by Tanner et al. [87] similarly investigated low-cost measurement of current, voltage, and power using an Arduino Uno incorporating an ATmega328P microcontroller. Their experimental configuration employed commercially available current-sensing components, including Hall-effect and current-shunt-based sensing approaches, to examine the feasibility of low-cost electrical measurement. Together, these studies illustrate the trade-off between the accessibility of inexpensive microcontroller-based measurement platforms and limitations arising from analogue-to-digital conversion, sensor characteristics, communication

bandwidth, and achievable sampling frequency.

Dezfouli, Amirtharaj, and Li [88] addressed similar challenges through the development of EMPIOT, a low-cost energy measurement platform designed for resource-constrained and Internet-of-Things devices. EMPIOT combines a Raspberry Pi base board with a shield incorporating an INA219 current and bus-voltage monitoring device. The Raspberry Pi controls the measurement hardware and records the resulting samples, while the INA219 measures bus voltage and the voltage drop across a shunt resistor from which current can be determined. The authors examined how factors including I^2C driver implementation, bus speed, buffering, and input voltage affect sampling rate, measurement accuracy, and processing overhead [88]. Their results demonstrate that the performance of a measurement platform depends not only upon the sensing device itself but also upon the communication and software mechanisms through which measurements are acquired. :contentReference[oaicite:1]index=1

EMPIOT was subsequently employed by Munoz et al. [89] in an investigation of the energy consumption associated with Advanced Encryption Standard execution on embedded devices. Collectively, these studies demonstrate the increasing use of dedicated external instrumentation for profiling embedded-system power use and energy consumption. They also illustrate several experimental considerations that are directly relevant to the methodology adopted within this dissertation, including measurement location, sampling frequency, measurement resolution, synchronisation with workload execution, communication overhead, and the separation of the measurement apparatus from the system being evaluated.

Chapter 3

Related Work

The runtime environment of the Java language (JRE) has undergone significant optimisations in the past two decades. Initially, the Java Virtual Machine (JVM) used separate execution models - the interpreter, client, and server virtual machines - but has since transitioned to a tiered system that incorporates all three. Before Java 7, an application's life-cycle was confined to an interpretive environment, the client or server VM. The tiered VM model utilises the benefits of all three execution models, enabling an application to begin in the interpreter, shift to the client C1 compiler, and eventually to the server C2 compiler for extremely hot code.

Recent JVM performance benchmarking has revealed that, for a limited number of application cases, Java's performance has surpassed that of ahead-of-time compiled code or is comparable [90, 20, 21, 91, 92, 22, 23, 93, 9]. Furthermore, Java has also been recognised as one of the top five energy-efficient languages [94, 95], but it still under-performs compared to native code [22], with C++ providing more significant performance advantages [23].

The JRE's core component is the JVM, which utilises a contemporary tiered execution model that provides several benefits, including faster startup times due to the interpreter and C1 compiler, as well as more powerful optimisations with the C2 compiler. However, the tiered architecture has its drawbacks, notably in terms of resource allocation, including CPU and memory consumption, and general runtime overhead [96, 97], consider the increased processing power needed

for advanced just-in-time optimisations. While interpretive execution of JVM bytecode has many advantages, it also has performance limitations when compared to JIT execution and execution time performance, which has been shown in early studies to significantly improve performance compared to interpretation [17, 98].

In this section we present the related work in regard to JRE JVM performance analysis. We first present research that directly considers the power and energy consumption behaviour of the JRE JVM. Subsequent to that, we consider those communications that focus on the effects associated with Java compilers and build toolchain compilers. We follow that with research relevant to JRE JVM modes of execution. We then present an overview of the work undertaken in relation to garbage collector performance. That discussion is then followed by a narrative relative to the modelling of JRE JVM workloads. We finally, present research directed at the microarchitectural behaviour of the JRE JVM.

3.1 JVM Power and Energy Modelling

Although execution time performance and memory usage are important factors, decisions to deploy on a JVM interpreter or just-in-time JVM also include considerations associated with power use and energy consumption [99, 100], which is still of considerable contemporary interest [94, 95]. Compiled languages do not always offer favourable energy consumption [90, 101]. Faster execution does not always equate to better energy consumption. Consider, for example, the properties associated with frequency scaling, with increased clock frequencies resulting in higher voltages [102], and thus greater power consumption. As the early work of Chen et al. in [99] highlighted, at the time, the JVM provided little support in regard to the optimisation of energy consumption of the JVM, and in addition provided no profiling support for its analysis. That work also highlighted that the energy consumption associated with memory accesses accounts for a large proportion of the energy consumed by the JVM. Even earlier work, for example, Vijaykrishnan et al. in [103] reported that performance, all though of importance,

focus had moved toward energy consumption. Their analysis compared and contrasted the JVM energy behaviour associated with instruction and data cache. The workloads used in the study were from the SPECjvm98 benchmark suite. Overall, their results showed that energy consumption efficiency is overwhelmingly more favourable for JIT execution compared to interpretive execution. However, their work is based on total energy consumed and no normalisation is undertaken to compensate for the execution time differences between interpretive and JIT execution modes.

Early analysis of JVM energy consumption focused primarily on handheld and resource-constrained computing platforms, particularly PDAs, mobile phones, and embedded devices. In particular, battery devices with significant constraints on their battery life. Modelling the battery life for such devices, Lafond et al. in [104] presented a power consumption model for interpretive JVM execution. Their analysis was associated with total energy consumption, and was based on the profiling of the Java Grande collection of workloads, specifically, the synthetic workloads, as well as a small number of synthetic micro workloads designed to capture JVM startup behaviour. The complexities in regard to measuring individual bytecode execution power consumption cost were highlighted, and as such they consider a subset of the bytecode specification. Their measurements, facilitated through the functionality of the KVM virtual machine, which allowed them to separate the power consumption behaviour associated with, for example, JVM startup, JVM interpreter execution, and JVM garbage collection. Their findings identified some interesting behaviour within bytecode instruction categories. For example, the loading of constant values, which can be undertaken through `bipush`, and dedicated typed `load` instructions, were found to have different power consumption costs. With the specialised `bipush` instruction associated with the better efficiency compared to instructions that rely upon local variable array access. Operand type was identified as a significant determinant in regards to bytecode energy consumption, both for arithmetic and conditional types. In regard to arithmetic

operations, their findings indicate that the operand type and the operation is a predominate factor in regards to energy usage efficiency. The differences between arithmetic bytecode operations whose operands are of type integer or long is in the order of 50%, this widens further for division operations to approximately 200%, for example. In contrast, their findings indicate that for the creation of array object types, `anewarray`, the arrays primitive type had no impact, but that the size of the memory allocation did impact on energy consumption. Method invocation energy cost, measured as an invocation/return bytecode pair was not differentiated by the method return type.

More recently, the trade-off between performance and power was considered by Cao et al. in [105] and specifically relative to the opportunities afforded by asymmetric multicore processor architectures. Their work focused on the performance of JVM services, those service categories representing the proportion of a metric spent doing garbage collection, just-in-time compilation, interpreter execution, or execution of optimised code. Their findings indicate that JVM services account for approximately 40% of execution time or energy consumption. Interestingly, they highlight that interpretive execution can account for approximately 15% and can be as high as 30% (imputed from inspection of Figure 1(a)). They also suggest that the interpreter remains a significant target in regard to power optimisation and energy optimisation. Their work considers asymmetric multicore resources comprising higher-performance big cores and lower-power small cores, together with their respective suitability for JVM services. They highlight that garbage collection and interpretive execution, running on small cores can be more beneficial. While JIT execution can benefit more from execution on the performance big cores. Their results also indicate that in-order pipeline architectures are associated with lower power use and lower energy consumption compared to out-of-order architectures.

Although previous research has examined the power use and energy consumption associated with JVM bytecode and individual JVM execution-engine

components, including the interpreter, just-in-time compilers, and memory management through garbage collection, other studies have investigated these electrical outcomes at a finer level of granularity. Noureddine et al. [106], as cited in [107], examined method-level execution, focusing specifically on method invocations associated with the Jetty Java Web Server. Their findings indicate that method tasks have varying levels of power consumption efficiency. In particular, their results indicate that HTTP requests have the largest impact on power consumption, and are specifically associated with least favourable behaviour.

Earlier work of Pinto et al., [108], focused on the management of concurrent programming constructs and specifically on the energy consumption behaviour associated with Java threads. They observed some interesting relationships between the number the number of application threads and the number of available CPU cores. In particular, as the number of threads increased, energy consumption also increased, however, as the number of live threads surpassed the number of available CPU cores energy consumption decreased. Their calculation of energy consumption is based on the energy consumed over a defined sampling interval. They attribute the observed butterfly effect partly to processor characteristics associated with the on-demand frequency governor. Under this mechanism, the operating frequency of idle CPU cores is reduced, although these cores continue to consume energy. The reduction in energy consumption observed when the thread count exceeds the number of available CPU cores is attributed to changes in execution time. Specifically, the proportional increase in power consumption associated with additional threads is smaller than the corresponding proportional reduction in execution time, resulting in lower overall energy consumption.

Although not primarily focused on JRE JVM power and energy consumption, the work of Pereira et al. in [94, 95] does provide a lens from which the JVM can be benchmarked, specifically relative to other languages and runtimes. Overall their results position the JRE in the top tier relative to other virtual machine languages, compiled languages, and interpretive languages. However, their results

indicate that virtual machine executed languages have a disadvantage compared to compiled languages in regard to energy consumption. In particular, their findings indicate that JVM executed workloads can be approximately 2.8 times less efficient, compared to the best performing language, C. However, depending on workload type the difference can be as low as 1.4 times. Their results do indicate that not all compiled languages outperform Java, for example, Pascal, Haskell, Go, and Swift, have less favourable energy profiles compared to Java. Relative to other virtual machine executed workloads Java offers better efficiency.

The focus of the research in [109] was in relation to understanding the power and energy consumption profiles of Java IO API classes. They modelled both `InputStream` and `OutputStream` as well as `java.io.Reader` and `java.io.Reader`, which provide IO interfaces for reading `bytes` and `chars`, respectively. The findings of Rochar et al. in [109], indicate differences in energy and power behaviour between input and output operations. In addition, within each IO class differences were also observed, for example, from an input perspective, `PushbackInputStream`, `FileInputStream`, and `RadomAccessFile` were associated with considerably less favourable energy and power consumption behaviour. Whereas the `Files.readAllLines`, `Files.newBufferedReader`, `Files.lines` and the `StringWriter` classes were associated with more favourable efficiency. In regard to IO output, the `FileWriter` and `PrintStream` classes were least favourable, whereas, the `ByteArrayOutputStream` class was more favourable.

The work of Longo et al. in [110] provides some insight into the costs associated with functions of the JVM other than garbage collection, interpretive execution, and just-in-time execution. They specifically addresses the effects associated with exception handling, the use of primitive data types over object types, object creation, and the construction of strings - fundamental processes within the JVM. They also considered a number of code refactoring techniques, for example, the removal of immutable classes, the elimination of recursion, the swapping of object data types with their primitive counterparts, for example. Their results

showed that for all those factors, alternatives offer better energy consumption efficiency. For example, removing exception handling results in an energy saving of approximately 99%; the use of primitive data types over object data types reduces energy consumption by approximately 23%; using StringBuilder over concatenation via the + operator reduces energy consumption by approximately 94%. In regard to refactoring, their refactoring was associated with energy savings ranging between approximately 4% and 99%.

Similar to the focus adopted by Rocha et al. [109], Oliveira et al. [111] demonstrate that decisions made at the application-workload level can also significantly influence power use and energy consumption. Their analysis compared thread-safe collection implementations, including `Vector`, `Hashtable`, `ConcurrentHashMap`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `ConcurrentSkipListMap`, and `ConcurrentSkipListSet`, with their non-thread-safe counterparts, including `ArrayList`, `HashMap`, `HashSet`, `LinkedList`, `LinkedHashMap`, `TreeMap`, `TreeSet`, and `LinkedHashSet`.

Their results indicate that implementations based on `Hashtable`, `HashSet`, and `HashMap` were consistently associated with lower energy consumption. However, the authors note that the energy consumption of a collection implementation remains dependent upon the characteristics of the underlying hardware platform.

More recent research, modelling the impact of JVM implementation and instances on energy consumption, was presented by Ournani et al. in [112], as cited in [113]. Their work focused on modelling differences across multiple JVM distributions and their associated versions. Their findings showed general increases in power consumption efficiency for each JVM distribution across all of their associated versions. Although there was an observation of increased efficiency across multiple distribution versions, transitioning between relatively close JVM versions was observed in many cases to result in a degradation in performance (imputed from an inspection of their associated results, Figure 1). In addition, some JVM distributions were associated with lower power consumption behaviour,

for example, GraalVM. In regards to OpenJDK distributions, power performance is seen to deteriorate for versions 11 and 12, relative to versions 9 and 10, with lower power consumption behaviour identified with versions 13, 14, and 15. In addition, their work considered the effect from redefining the JVM's JIT compiler parameters. The parameters examined comprise the minimum number of method calls required before JIT compilation, the number of JIT compilation threads, and the level of tiered compilation.Their findings show that the default JVM configuration predominately provides the lowest energy consumption behaviour. However, their findings did show a number of workloads for which, fixing the JIT compilation level was more efficient. In addition to the aforementioned JVM parameters, the work of Ournani et al. also considered the effects associated with changing the garbage collection algorithm used for heap management. In general, their findings indicated that the default garbage collector offered the best energy consumption performance. However, the type of workload was an important factor, and they suggest that garbage collection strategy should be considered relative to the workload that the JVM will be executing. In regard to their HotSpot JVM distributions, that included an analysis of two OpenJDK implementations, showed that the default garbage first collector did was not necessarily offer the best efficiency, for many workload cases the serial collector offered statistically significantly lower energy consumption.

The choice of design pattern followed in regard to application development and its subsequent impact on power and energy was the focus of Bree et al. in [114]. They specifically consider the impact that the decorator pattern has on energy consumption. Their rational is associated with the extra overhead that the implementation of the decorator pattern has on the runtime behaviour of an application. The overhead associated with the redirection of messages from the decorator to the decorated component was shown to have a substantial impact on energy consumption. Their findings show that refactoring can have a significant impact on energy efficient, and in regard to their analysis, removal of the decorator

pattern statistically significantly improves energy consumption.

In regards to energy consumption based on garbage collector algorithm implementations, the work of Shimchenko et al. in [113] provides another recent analysis of energy consumption differences. Their work is similar to that undertaken by Ournani et al. [112], as both studies consider JVM version and garbage collector type. However, their work extends this analysis through the inclusion of microarchitectural performance event measures and the development of a machine learning model of JVM energy consumption. Shimchenko's results were associated with JRE versions 13 and 16. They observed that the earlier version of the JRE, version 13, was more energy efficient in contrast to version 16. They attributed the differential to differences in default method inlining depths defined for each JVM version. The regression in energy performance across JVM versions had also been observed by Ournani et al., however, their results indicated that JVM version 13 was less efficient compared to JVM version 16. In regard to the effect that garbage collector variant has on JVM energy behaviour, their findings indicated an association between heap size and energy consumption, in particular, a negative association, with increased heap memory sizes associated with a reduction in overall energy consumption. An observation they attribute to the reduced need for garbage collections due to more available memory. A notable characteristic present within their results is that workload type is associated with significant differences in energy consumption. Specifically in regard to garbage collector variants, JVMs using the ZGC and Shenandoah collectors, fully concurrent collectors, are associated with the worst energy consumption performance. They attribute those observations to the extra overhead incurred due to thread synchronisation. Their analysis also tested whether optimisations targeting increased execution time performance are associated with a decrease in energy consumption. Their results show that workload category are a moderating variable in regard to the correlation between execution time performance and energy consumption.

3.1.1 Limitations of Existing Literature

Although significant progress has been made in understanding JVM power use and energy consumption, several persistent gaps remain. Foundational studies by Chen et al. [115] and Lafond et al. [104] established baseline models of JVM energy consumption, often focusing on embedded or battery-constrained environments. While these studies offered valuable insights into the contributions of interpreter, garbage collection, and JIT execution to overall energy use, they largely restricted their analysis to coarse-grained measurements, specific workloads, or limited JVM variants. Subsequent research, for example by Cao et al. [116] and Noureddine et al. [106], broadened the scope to multicore environments and method-level effects, yet challenges remain in isolating and attributing power use to specific runtime components and code patterns.

Recent literature, including the work of Ournani et al. [112] and Shimchenko et al. [113], has begun to investigate relationships between JVM distribution and version, garbage collector algorithm, heap size, and observed power use and energy consumption. However, inconsistencies across these findings, particularly concerning whether newer JVM versions are consistently associated with lower energy consumption, indicate that additional variables may contribute to the observed variation and highlight the complexity of the JVM execution environment. Moreover, while workload type is increasingly recognised as an important explanatory dimension of energy consumption, the field still lacks comprehensive, fine-grained models capable of accounting for variation across application domains and JVM configurations.

Additionally, recent studies by Longo et al. [110], Oliveira et al. [111], and Bree and O Cinnéide [114] have demonstrated significant relationships between development-level decisions involving exception handling, data structure selection, and design pattern usage and JVM energy consumption. Nevertheless, the relationships between these code-level decisions, JVM configuration parameters, and the underlying hardware remain insufficiently characterised within the current

literature. Consequently, the field still lacks integrated models or guidelines capable of systematically informing developers and JVM designers about the power use and energy consumption implications of these decisions.

3.2 Build Toolchain Effects

Most production-grade JVMs, including HotSpot/OpenJDK, Eclipse OpenJ9 (IBM J9), Azul Zing, and Oracle JRockit, are implemented primarily in C and C++. While there are also JVMs implemented in Java (e.g., Oracle GraalVM), the vast majority rely on C/C++ for production-level deployment. Building such JVMs necessitates an appropriate C/C++ compiler toolchain, with the choice of compiler closely linked to the target architecture and host operating system. For instance, the GNU Compiler Collection (GCC) is standard for Linux, Unix, and ARM-based embedded systems, whereas Microsoft Visual C++ (MSVC) is typically used for Windows targets. The selection of the build toolchain can directly influence the characteristics and performance of the resulting binaries. Different compilers—and even different versions or configurations of the same compiler—apply different code generation strategies, optimisations, and low-level implementations. These choices are associated with differences in instruction selection and scheduling, optimisation strategies, compatibility, vectorisation support, binary size, memory footprint, and cross-platform consistency. However, as Kumar [117] observes, traditional compilers and optimisers may not always generate truly optimal code. Thus, the selection of build toolchain compilers has the potential to directly influence the characteristics and performance of JRE and JVM binaries.

Recent studies have shown that the performance of large systems can vary significantly depending on the GCC release used for building, supporting the need to consider alternative compiler versions during system builds. Marcon [118] specifically examined the impact of GCC version in the build toolchain for the Geant4 simulation toolkit [119], which is widely used in high-energy physics

experiments at the Large Hadron Collider. Analysing three GCC versions (4.8.5, 6.2.0, and 8.2.0) and four standard optimisation flags (−`O1`, −`O2`, −`O3`, and −`Os`), along with static and dynamic compilation modes, Marcon et al. found that newer GCC versions could improve execution time performance by up to 30%. They also observed statistically significant performance differences between statically and dynamically compiled systems.

The work of Pallister et al. [30] addressed not only the standard GCC optimisation levels but also the broader challenge of finding optimal combinations of more than 150 available compiler flags, with an emphasis on improving the energy consumption of embedded applications. Notably, they highlight that energy consumption optimisation has received relatively little attention in the literature compared to execution time. Although energy consumption is often assumed to be directly proportional to execution time, Pallister et al. demonstrated that this relationship is not always linear. For instance, on the ARM Cortex A8, increased instruction-level parallelism improved execution speed, but energy reductions were not always proportional. Their findings show that enabling or disabling compiler optimisations associated with levels −`O1` and −`O2` can significantly affect energy use. Specifically, disabling the `-fguess-branch-probability` optimisation led to a reduction in energy consumption of about 4

Earlier, Tiwari et al. [120] identified a range of GCC optimisations relevant to energy reduction, including instruction reordering (with about 2% difference in energy footprint), code generation through pattern matching, and minimisation of memory operands—since memory reads and writes can increase energy consumption by approximately 43% and 77%, respectively, compared to register operands.

Branco et al. [121] further examined the impact of GCC optimisation levels on energy use, finding that while optimisation levels do affect energy consumption, there was no statistically significant difference between the median values at levels −`O1`, −`O2`, and −`O3` ($\chi^2 = 0.0117$, $df = 2$, $p = 0.994$).

In the context of large-scale software, several studies offer useful comparisons. For example, Yuan et al. in [122] and Ugur et al. in [123]. Both studies considered the large codebase associated with the Linux Kernel, However, Yuan et al. investigated performance differences in the Linux Kernel when built with four different GCC versions. Although not statistically analysed, their results showed modest but consistent performance improvements across a small suite of applications for kernels built with newer GCC versions. They also found that the use of alternative optimisation flags within the `-Ox` series could produce greater improvements. Even though the reported gains were relatively small (around 1%), Ugur et al.[123] argue that such single-digit gains are highly meaningful in large-scale settings, leading to substantial reductions in data centre costs and carbon footprint. Ugur et al. [123] extended this work by focusing on profile-guided optimisation. By aggregating control flow profiles from typical data center applications, they generated a universal profile used to optimize the Linux Kernel, achieving performance improvements of about 8%.

More recently, Palkowski et al. [124] leveraged compiler-level optimisations, specifically, loop tiling to improve cache spatial locality to significantly reduce both execution time and application energy consumption across a set of benchmark applications.

Despite the critical influence of compilers on the efficiency of generated binaries, there remains a notable gap in research specifically examining how changes to the GNU GCC compiler version within the JRE/JVM build toolchain affect power use and energy consumption of the resulting JRE and JVM binaries during execution. Venugopal et al. [125] suggest that this gap may be due in part to the pointer-intensive and call-intensive nature of JVM implementations, which limits the effectiveness of global compiler optimisations—especially given the JVM's late binding model, where symbol and branch targets are unknown at build time. Their work proposed the concept of a "domain specific Java Virtual Machine" tailored for static, embedded applications, where more effective compiler optimizations

may be possible.

On the continuum between general-purpose JRE/JVM builds and application-specific tailored builds, Wimmer et al. [126] investigated GraalVM Native Image for microservices and serverless functions. Their research applied points-to analysis for memory layout, ahead-of-time class initialisation, and full ahead-of-time compilation, finding that the resulting native images could achieve performance improvements of up to two orders of magnitude.

Within the specific context of the JRE and JVM, recent work by Lambert et al. [127] is one of the few detailed studies of how modern compiler selection and integration within the JRE build toolchain affects runtime performance. While their focus was on execution time (and not power use or energy consumption) for interpreter mode, they demonstrated that GCC compiler version can have a substantial impact on performance, with workload category further moderating these effects.

Building on this, Lambert et al. [29] comprehensively evaluated the performance gap between interpretive and tiered execution using workloads from the Renaissance Benchmark Suite. Their findings show that, for modern workloads on multicore architectures, tiered execution outperforms interpretation by factors of 4x to 37x. Importantly, they found that OpenJDK version and GCC compiler version did not substantially alter the performance gap between execution modes; instead, workload type (web-based versus functional/Scala-based) was the key differentiator.

Given that energy consumption is typically proportional to both power use and execution time, these findings raise important questions about the extent to which build toolchain compiler version influences the power use and energy consumption of JRE and JVM binaries in real-world scenarios. Where such differences exist, understanding their nature and drivers is crucial for software developers and system architects seeking to optimize Java-based systems for both performance and sustainability.

3.2.1 Limitations of Existing Literature

While the influence of compiler toolchain selection on execution time performance has been repeatedly demonstrated across both application-level and system-level software (e.g., Marcon et al. [118], Pallister et al. [30], and Yuan et al.), there remains a notable lack of research directly addressing its impact on power use and energy consumption in the context of JVM and JRE binaries. Although studies by Pallister et al. and Tiwari et al. have demonstrated significant relationships between specific compiler optimisations and energy consumption at the embedded application level, these findings have not yet been systematically extended to the JVM itself, which exhibits distinctive characteristics associated with late binding and pointer and call intensity, as discussed by Venugopal et al. [125].

Recent work by Lambert et al. [127, 29] has highlighted the performance sensitivity of JVM interpreter execution to GCC version within the build toolchain, yet their analyses were restricted to execution time, leaving questions about the compiler's impact on energy and power largely unexplored. This omission is particularly salient given that workload type and JVM execution mode have both been shown to moderate performance effects, suggesting that similar complexities may exist for energy use as well. Furthermore, while some efforts (e.g., Wimmer et al. [126]) have explored the efficiency of highly optimized native images, systematic comparisons of power and energy consumption across standard JRE/JVM binaries built with different compilers and optimization configurations are still lacking.

Overall, there is a clear research gap concerning the extent to which compiler toolchain selection within JVM builds can be leveraged to achieve meaningful improvements in power use and energy consumption for Java systems—especially in real-world, multicore, and data center environments where small gains may have large aggregate impact (as noted by Ugur et al. [123]). Addressing this gap will require empirical, workload-sensitive analysis of both power and energy behaviour under different build toolchain scenarios, going beyond existing performance-centric studies.

3.3 Execution Mode

The significance of the JVM interpreter in modern times [27] goes beyond its role in the JRE and providing guidance for just-in-time compilation. Its importance also stems from its widespread use for executing code on various platforms. Many languages target the contemporary JVM due to its ability to efficiently and competitively execute bytecode, for example; Scala, Groovy, Clojure, Jython, JRuby, Kotlin, Ceylon, X10, and Fantom [128]. The availability of JVM interpreters reduces the costs associated with language development and allows languages to target different architectures without constructing dedicated compilers. This significantly facilitates the development of modern languages, new features, and testing of alternative paradigms. The Truffle language implementation framework is an example of a contemporary framework that enables the implementation of languages on interpreters that use the GraalVM and its associated optimised JRE [129]. Additionally, the growth in paradigms and workloads targeting the JVM presents new optimisation possibilities for current compilers [130, 131]. Interpreters are more adaptable [132], simpler, and more portable than compilers [55], with a compact code footprint, reduced hardware complexity, and development costs [133]. Debugging and profiling can be performed more easily with an interpreter [132], while expensive tasks involving register allocation and call-stack management are not required [48].

3.3.1 Interpretive Execution

Interpreters do have limitations relative to compiled and just-in-time executed languages. The obvious and most notable being execution time [17, 98], with more contemporary work identifying a significant differential between interpretive and tiered execution, interpreted execution can be between 5 and 40 times slower, as per more recent studies [29].

Both interpretive execution and just-in-time execution have been extensively studied, although there has been less of a focus on (Java) interpretive execution

in recent years. Early research on interpretive execution examined the dynamic semantics of interpretation and found that the interpreter dispatch loop was a major contributor to the JVM's underperformance. Java interpreters rely on a number of instruction dispatching mechanisms, including switch-based, direct threading, direct call threading, and inline direct threading. These techniques have varying levels of efficiency, with switch-based being the simplest but the least efficient, requiring at least three control transfer instructions per instruction [54, 55].

Direct threaded dispatch, in contrast, updates the internal opcode structure with the implementation's address, removing the instruction switch and encapsulating the dispatch loop. Direct threaded code's instructions control the program counter while the switch expression updates it. Inline threaded implementations combine instruction sequences within bytecode basic blocks, eliminating dispatch overhead for all instructions except the last one. Both techniques improve the efficiency of instruction dispatching. Casey et al. [54] and Gagnon et al. [55] offer detailed overviews of all three techniques. Early research found that instruction dispatching was the most expensive bottleneck for bytecode execution.

The benefits of threaded implementations were observed in earlier work by Ertl and Gregg [134, 135], who examined the computational performance of the GForth, OCaml, Scheme48, YAP, and XLISP interpreters. They reported that threaded implementations provided up to a 200% increase in runtime performance over switch-based implementations. However, the direct-threading optimisation used by the JVM interpreter encounters underperformance issues due to branch misprediction resulting from the large number of indirect branches introduced into direct threaded code. This issue is discussed in Casey's work [54] and was addressed through the implementation of instruction replication and extending beyond basic block boundaries using super-instructions. However, aggressive superinstruction implementations can increase cache miss-rates, as discussed in [136]. Berndl et al. [136] proposed context-threaded interpretation of Java

bytecode. The optimisation eliminates indirect branches through the introduction of subroutine calls and a return from that subroutine call. These instructions are optimised on modern architectures, and as reported by Berndl et al. outperform direct-threaded code that heavily relies on indirect branches.

A more recent contribution to interpreter design, aiming at removing the inefficiency associated with instruction dispatch, has been the development of subroutine threaded code, communicated by Savrun-Yeniçeri et al. in [137]. The technique has wider applications than just removing the overhead associated with dispatch and in fact opens the JVM runtime environment up to languages that might wish to use it as an execution environment target. Similar to the context threading optimisation communicated by Berndl et al. [136], although differing in regards to relying upon native call instructions and native jump instructions, with no introduction of native subroutine calls. It could be argued that the novelty of the approach forms the backbone of the modern template interpreter built during JVM startup.

In that regard, the modern JVM interpreter, the "template interpreter," has taken a similar approach to instruction dispatching at a lower level. The template interpreter creates a dispatch table that maps Java bytecode to optimised assembly instruction sequences during the build process. The construction of the template interpreter allows the JVM to recognise the architecture it is running on and produce optimised code for each bytecode, called codelets. These codelets are designed for individual bytecode instructions, and frequently executed bytecode sequences [138]. The most recent analysis of the performance of the template interpreter relative to the traditional C++ interpreter identified that the template interpreter outperforms the C++ interpreter by a factor of two to three, as reported by Simonis et al. in [139], and as cited in [138]. Simonis in [139] also observed a performance differential when analysing tiered execution, with either the template interpreter or the C++ interpreter, albeit with a performance ranging from 1.1 times to twice as fast.

Although the template interpreter can be viewed as one of the more efficient interpreter implementations, it has achieved its efficiency through the implementation of many optimisations that have been proposed within the literature prior to its release, for example, its implementation can be considered to encapsulate the direct-threaded behaviour that was communicated by Casey et al. in [140]. Although, its bytecodes are implemented in assembly, the implementation in assembly is also not new, and in particular the benefits of assembly implementations has been documented in the literature prior to the integration of a template interpreter within the JVM, for example, within the work of Shi et al. in [141, 142].

3.3.2 Just-in-time Execution

Albeit there are a number of definitions of a compiler, in this discussion we will be restricting our narrative to those that convert their inputs directly to machine executable instructions. In that respect, there are two broad categories, the static compilers that translate directly to machine executable code, producing an executable artefact; and the dynamic compilers that translate to machine code while executing.

The broad methodology of just-in-time compilation was not introduced by the Java Language, as it had been utilised by several other programming languages prior to Java's release, although the Java Language developers are attributed with the introduction of the term [13]. Further, Aycock et al., in [13], provide a detailed chronological account in regard to the adoption of the just-in-time compilation paradigm.

Early research on just-in-time compilers for the JRE and its associated JVM can be divided into two groups depending on the intended execution target of the Java code, either on the client or server VMs. The client VM just-in-time compiler (C1) implemented fast compilation techniques and specifically those that required a small overall footprint, while the server just-in-time compiler (C2) offered highly optimised code, generated from more complex analysis and profiling,

with a significantly higher demand for resources. However, the modern JVM utilises a tiered execution methodology, in which code starts execution within the interpreter, with the C1 compiler providing optimising options, followed by the C2 compiler for extremely hot code segments. Regardless of the target, the JVM just-in-time compilers have incorporated numerous optimisations over the past thirty years, which can be classified based on compiler tactics, as outlined more recently by Ivanov et al. in [59] and extensively by Rose et al. in [143].

- Delayed compilation: tiered compilation, or on-stack replacement strategies;
- Proof based techniques: exact type inference, memory value inference, constant folding, null-check elimination, algebraic simplification, common sub-expression elimination;
- Flow sensitive rewrites: conditional constant propagation, flow-carried type narrowing, dead code elimination;
- Language specific techniques: class hierarchy analysis, devirtualisation, symbolic constant propagation, autobox elimination, escape analysis, lock elision and fusion;
- Speculative profile based techniques: optimistic nullness and type assertions, optimistic type and array length strengthening, untaken branch pruning, optimistic N-morphic inlining, branch frequency prediction, call frequency prediction; memory and placement transformation: expression hoisting, expression sinking, redundant store elimination, adjacent store fusion;
- Loop transformations: loop unrolling, loop peeling, safepoint elimination, iteration range splitting, range check elimination, loop vectorisation;
- Global code shaping: inlining (graph integration), global code motion, heat-based code layout, switch balancing, throw inlining;

- Control flow graph transformation: local code scheduling, local code bundling, delay slot filling, graph-coloring register allocation, linear scan register allocation, live range splitting, copy coalescing, constant splitting, copy removal, address mode matching, instruction peep-holing, and DFA-based code generation.

3.3.3 Limitations of Existing Literature

While there is a rich body of research characterizing both interpretive and just-in-time (JIT) execution in the JVM, several notable gaps persist. Much of the foundational work on interpreter performance has examined switch-based, direct-threaded, and context-threaded instruction dispatch mechanisms, with a substantial body of evidence demonstrating the performance benefits associated with these approaches (Casey et al.; Ertl & Gregg; Berndl et al.; Savrun-Yeniçeri et al.; Simonis et al.). However, despite these advances, recent work has largely shifted away from a detailed analysis of interpreter execution, particularly in the context of modern workloads, architectures, and the highly optimised template interpreter now used in production JVMs. There remains limited understanding of the relationship between optimised template interpreter execution and hardware mechanisms involving branch prediction and cache hierarchies within multicore architectures.

Similarly, while the progression of JIT compilation has been thoroughly documented, especially regarding optimisation strategies and the evolution from client/server to tiered execution models (Aycock; Ivanov et al.; Rose)—there is a lack of empirical studies that systematically compare the power use and energy consumption between modern interpretive or tiered execution across a diverse set of modern JVM workloads. The literature predominantly addresses performance (i.e., execution time) rather than power use and energy consumption characteristics, with only a handful of recent studies (e.g., Lambert et al.) extending comparisons to modern multicore platforms or considering the impact of workload

type (e.g., web-based vs. functional workloads) on the gap between execution modes.

Finally, although interpreter and JIT optimisations are well-catalogued at a technical level, there is little work exploring the practical implications of these choices for contemporary language implementers or for JVM deployments targeting domains with strict energy or thermal constraints. In particular, there is limited data-driven guidance for selecting JVM execution modes under varying system constraints, and comparatively little examination of the relationships between interpreter or JIT enhancements and system-level measures of power use and energy consumption.

3.4 JVM Garbage Collection

Traditionally the analysis of garbage collector efficiency had primarily concentrated on three performance dimensions, throughput, latency, and memory footprint, for example [144, 31, 50]. Garbage collector variants were typically associated with tradeoffs across those dimensions. For example, the traditional parallel collectors were tuned with throughput in mind, at the expense of latency, whereas, the concurrent collectors minimise latency at the expense of throughput. Early work in regard to garbage collector behaviour and their characteristics is primarily void concerning the impact of collectors on application energy consumption, and more recent communications still fail to deal with power and energy dimensions, for example [145].

However, some early work did consider the impact of garbage collection strategy on energy consumption, for example the work of Chen et al. in [115]. Their work addresses energy consumption efficiency through the gating of power supply to memory regions that are not occupied by "useful data", effectively reducing leakage energy. They report savings of approximately 28% in regard to heap energy for their adaptive garbage collector. Other early work by Vijaykrishnan [103], as cited in [115], considered the energy consumption behaviour of Java workloads.

Their results are based on a simulation of memory behaviour and an associated energy model. They considered three JRE/JVM activities associated with memory use: class loading, just-in-time compilation of methods to native code and the subsequent storage of compiled code in memory, and garbage collection. However, their analysis includes the performance of a single garbage collector variant, an exact collector. They identify significant energy consumption behaviour between interpreted and just-in-time executed code, however, their results do indicate that the relative differences are dependent on the JRE JVM workload.

Other work, for example that of Contreras et al. in [146], considered four garbage collector variants as part of their analysis of JVM energy consumption. Their workloads were composed of the SPECjvm98 benchmarks and the DaCapo benchmarks. They instrumented the Jikes RVM using entry and exit system calls, that were triggered when a JVM component was utilised. They instrumented the thread scheduler within the Kaffe JVM, as the Kaffe JVM runs certain system services on different threads. Their results indicate that garbage collection can account for an average of 37% of total energy consumption, specifically for the SPECjvm98 workloads, and on average 32% for the DaCapo workloads. Specifically in regard to effects associated with garbage collector variants, their findings indicate that generational collectors are associated with reduced energy consumption compared to non-generational collectors. However, their findings do indicate that the efficiency gap between generational and on-generational collectors narrows as the available heap size increases. In addition, their findings indicate that increasing heap size reduces the the number of garbage collector pauses. Which is advantageous toward execution time, reducing execution time, and as a result reducing energy consumption.

Later work dealing with power and energy consumption performance efficiency relative to code bloat was the focus of research undertaken by Bhattacharya et al. in [147].

More recent analysis of the effects associated with changing garbage collector

was undertaken by Pufek et al. in [148]. Their work concentrated on performance differences across OpenJDK JRE versions 8, 11, AND 12. Their performance comparisons were associated with the default serial, parallel, concurrent mark and sweep, and the garbage first collectors. Although not concentrating on the impact of garbage collector choice on power or energy consumption, their results do shed light in regard to execution time differences and garbage collection frequency performance. Their findings, based on an analysis of the DaCapo workloads, vary, and highlight that workload type is a dominant factor in regard to identifying an optimum garbage collector, both from an execution time and collection frequency perspective. The G1 collector was predominately the better collector across all OpenJDK versions compared to the serial, and the concurrent mark and sweep collector. However, the parallel collector outperformed the G1 collector in a large number of cases.

More recent work has included energy consumption as an additional dimension in relation to garbage collector performance analysis. In addition to the analysis of JVM parameter specifications, the work of Ournani et al. in [112] considered the effects associated with changing the garbage collection algorithm used for heap management. In general, their findings indicated that the default garbage collector offered the best energy consumption performance. However, the type of workload was an important factor, and they suggest that garbage collection strategy should be considered relative to the workload that the JVM will be executing. In regard to their HotSpot JVM distributions, that included an analysis of two OpenJDK implementations, they showed that the default garbage first collector did not necessarily offer the best energy consumption efficiency, for many workload cases the serial collector and the parallel collector offered statistically significantly lower energy consumption.

The work of Shimchenko et al. in [113] also focuses on an analysis of garbage collector energy consumption differences. Their work is similar to that of Ournani et al. [112], as both studies consider JRE source version and garbage collector

variant. However, their work differs in regard to the inclusion of microarchitectural parameters and the development of a machine learning model of JVM energy consumption. Shimchenko's results were associated with JRE versions 13 and 16. They observed that the earlier version of the JRE, version 13, was more energy efficient in contrast to version 16. They attributed the differential to differences in default method inlining depths defined for each JVM version. The regression in energy performance across JVM versions had also been observed by Ournani et al., however, their results indicated that JVM version 13 was less efficient compared to JVM version 16. In regard to the effect that garbage collector variant has on JVM energy behaviour, their findings indicated an association between heap size and energy consumption, in particular, a negative association, with increased heap memory sizes associated with a reduction in overall energy consumption. An observation they attribute to the reduced need for garbage collections due to more available memory. A notable characteristic present within their results is that workload type is associated with significant differences in energy consumption. Specifically in regard to garbage collector variants, JVMs using the ZGC and Shenandoah collectors, fully concurrent collectors, are associated with the worst energy consumption performance. They attribute those observations to the extra overhead incurred due to thread synchronisation. Their work also tested whether optimisations targeting increased execution time performance are associated with a decrease in energy consumption. Their results show that workload category are a moderating variable in regard to the correlation between execution time performance and energy consumption.

As cited in [145], the work of Cai et al. in [149] considered the energy consumption of a number of garbage collector variants distributed as part of OpenJDK version 17. Their methodology focuses on positioning the cost of garbage collection relative to execution without explicit garbage collection of heap memory.

Although not focusing on the power and energy consumption effects asso-

ciated with garbage collector variants, more recent work, for example that of Tavakolisomeh et al. in [145], has focused on the challenge of identifying the best performing garbage collection strategy in regard to JRE memory management. Their findings highlight that the default collector, G1GC, underformed relative to the parallel collector across all analysed workloads in regard to execution time. Switching to the parallel collector is associated with reduced execution time. In contrast, in regard to latency, the parallel collector underformed relative to the G1 collector.

3.4.1 Limitations of Existing Literature

While the efficiency of JVM garbage collectors has long been analysed in terms of throughput, latency, and memory footprint (see Dykstra et al.; Blackburn et al.; Lengauer et al.), there is a pronounced historical gap in systematic evaluation of power use and energy consumption across garbage collection strategies. Early studies by Chen et al. and Vijaykrishnan provided initial insights into the energy consumption associated with specific garbage collectors, but often relied on simulation or model-based methods, limited their scope to individual collectors, or did not examine the corresponding relationships with power use during real execution. Most of these foundational works also did not explore the moderating effects of workload type or JVM version, leaving unclear the broader relevance of their findings.

Subsequent research expanded the evaluation to more collector variants and benchmark suites (Contreras et al.), but the focus remained primarily on execution time and garbage collection frequency, with power use and energy consumption rarely measured directly or normalised across collector designs. Moreover, recent works like Pufek et al. and Tavakolisomeh et al. continue to concentrate on throughput and latency, providing only partial or indirect information about the implications for power use and energy consumption, despite contemporary relevance for green computing and sustainable systems.

Only in the last several years have studies begun to systematically incorporate energy and power metrics (e.g., Ournani et al.; Shimchenko et al.; Cai et al.), examining the influence of garbage collector algorithms, JVM version, heap size, and microarchitectural factors. These studies demonstrate that both collector type and workload category significantly moderate power use and energy consumption, and reveal that default collectors are not always the most efficient. However, the coverage of JVM distributions, hardware, and workload types remains limited, and inconsistencies between studies (e.g., on whether version 13 or 16 is more energy efficient) highlight the need for more robust, generalisable findings.

In summary, despite progress, there is a lack of comprehensive, comparative research directly quantifying the effects of garbage collector choice on both power use and energy consumption across JVM versions, collector variants, and representative workloads. This gap hampers the ability of developers and system architects to make data-driven choices optimising for both performance and sustainability in real-world Java deployments.

3.5 JVM Workload Modelling

Early work in regard to the modelling of the static and dynamic behaviour of the JVM concentrated on the characterisation of the workloads that the JVM is expected to execute.

Although not directly concentrating on the JVM instruction set, the work of Davidson et al. in [150] focused on modelling the relationship between static and dynamic machine instruction frequencies and their associated addressing modes. Their findings, showed a strong relationship between static and dynamic measurements. However, tests of their model on a Motorola CISC machine showed significant deviation from their observed behaviour. An indication that model definitions are not necessarily applicable to cases which were not included as part of overall model development.

Motivated partly by the need to produce more optimal instruction sequences,

Bowers et al. [28], as cited in [36], presented one of the earliest analyses of the dynamic characteristics of JVM bytecode. Consistent with the stack-machine semantics of the JVM, instructions associated with operand-stack operation, particularly loads, stores, and stack manipulation instructions, accounted for a substantial proportion of executed instructions, with object field access making an additional substantial contribution. Their analysis of the dynamic characteristics of the JVM operand stack showed that, in approximately 99% of cases, the stack depth did not exceed four items. Their characterisation of dynamic bytecode frequencies therefore has important implications for top-of-stack operand caching, indicating that four registers would be sufficient to implement a stack-caching mechanism within the JVM.

Around the same period, McGhan et al. in [37] communicated an architecture of a microprocessor core, PicoJava, that executes Java bytecode at the silicon level. The designed instruction set architecture specified a larger collection of instructions compared to the original Java instruction set, 341 in total. Those extra instructions accommodate register reads and writes and system software support, for example. One of the significant features implemented was an instruction operand stack cache, that eliminated the overhead associated with the management of the JVM software operand stack.

In regards to method invocations and the execution of their bytecode instructions, the Java operand stack frame plays an integral part, both in regards to storing bytecode operands but also in regards to storing the results of bytecode interpretation/execution. The Java stacks have been identified as one of the significant bottlenecks in regard to efficient bytecode execution. The early work of Waldron et al. in [38], like the work of Bowers et al. in [28], modelled the behaviour of the JVM stacks, as well as the local variable array, and parameter types passed to Java methods. Their work would support the findings communicated by Bowers, in that the depth of the Java operand stack is relatively shallow. Although, Bowers identified a predominate stack depth of approximately 4 items,

the work of Waldron moved that to approximately 5 items. A differentiating factor between both studies are the workloads analysed. In addition, some of the workloads analysed by Waldron exhibited stack depth behaviour in excess of 8 items. These findings have implications for stack caching mechanisms, with larger stack depth behaviour placing pressure on instruction scheduling through registers availability. Unlike the previous work, Waldron also considers the targets of method invocations, finding that method invocations predominately target API defined methods, with a large proportion of native method library code.

Expanding on their previous analysis, Waldron in [39], undertake a full bytecode level analysis. Communicating dynamic bytecode frequencies, and applying an instruction categorisation consisting of 22 instruction groupings. Their findings show that local variable loads onto the operand stack account for the largest frequency of instruction types, followed by field access and stores. Like other reported findings, and as communicated by Waldron, the operand stack is a significant impediment to efficiency, although, this is more emphasised by the inherent non-random access implementations of the JVM operand stacks. Finally, Waldron suggests that the small instruction set size of the JVM, could benefit from expansion with more instructions that execute more efficiently, a characteristic found within the PicoJava processor.

Examining the effects in regard to the compilation of Java source to bytecode, and the bytecode distributions emitted by different Java compiler versions, the communications of Waldron et al. in [40] present those effects. Their work is of importance in regard to questions related to the generation of API library code, in that their findings have shown that method invocation targets are predominately toward native libraries and other Java APIs. Their overall findings highlighted the constraints associated with the compilers generating bytecode, in that similar distributions were observed, an inference from which is that traditional compiler optimisations are not applied by Java bytecode compilers in general. Their findings show the dominance of API method invocations over actual program method

invocations, and they suggest compilation of such library code to native code in order to achieve a significant performance improvement. They found significant levels of association between bytecode category frequencies across workloads. Similar work to that mentioned was communicated by Daly et al. in [16] and Gregg et al. in [43]. In those studies they examined the effect from changing Java-to-bytecode compiler, and specifically examined the effects associated with Java Grande benchmark workloads. Their findings indicate that predominately a few methods, five, are responsible for the vast majority of method invocations. In addition, on average approximately 37% of method invocations are targeted at API code, in regards to native method invocations, approximately 14% are in that direction. Considerable differences are seen across benchmark workloads, in some cases API invocations account for up-to 99% of calls.

The work of Horgan et al in [16] expanded the domain of analysis and added to the findings in regard to dynamic bytecode modelling. Their work specifically focused on modelling the similarity between JVM workloads at the bytecode level. That work is important in regard to benchmark workload design and for the identification of optimal workload sets. Their findings indicate that in regard to the Java Grande Benchmark workload collection that there is a high degree of dissimilarity between workloads, a favourable characteristic in regard to benchmark collections. Their work also considered the effects associated with the use of different Java to bytecode compilers. Their findings, although numerous, show that the choice of Java to bytecode compiler can result in increases in the number of bytecode instructions executed, the gcj Java to bytecode compiler, for example, was associated with an average increase of 4.7% in bytecode execution, with an upper threshold of approximately 8.7% depending on the workload compiled. It would seem that the increase would have implications in regard to execution time efficiency, as well as total energy consumption. In regard to overall dynamic instruction mix characteristics, their findings show levels of dissimilarity across compiler versions.

The characteristics of method invocation target was the focus of the work of Byrne et al. in [41], in particular, the contrast between method invocation target within API and non-API invocations. They propose a metric, the ratio between the number of methods invoked at least once within API code and the percentage of executed bytecodes, as a measure to quantify workload behaviour. In regard to method invocation target characteristics, Byrne et al. indicate differences in target frequencies between benchmark suites. For example, their reported results indicate that approximately 39% of method invocations are targeted at methods of the SPEC JVM98 application codebase, whereas, approximately 15% are to program methods of the Java Grande applications. That difference highlights the importance of understanding the atomic characteristics of a workload, as in the case of the Grande applications, predominately API code is driving the JVM's execution. Their findings also support previous findings as communicated in [40] in regard to the dominance of API code execution.

The findings of Davidson et al., as noted earlier, did generate interest in exploring if the static-dynamic association, observed at the microarchitectural instruction set level, was transferable across to the JVM instruction set. The work of Dowling et al. in [42] examine the degree to which static bytecode frequencies correlate with their dynamic frequencies. Their findings are interesting in that they communicate different levels of association across the bytecode categorisations. A number of interpretations of correlation coefficient magnitudes exist, and are quite subjective. In their paper they set the strong correlation threshold at correlation magnitudes exceeding 0.8, and based on that threshold only four instruction categories are deemed to have a strong association between static and dynamic frequencies. An alternative categorisation was communicated by Cohen et al. in [49], in which correlation magnitudes greater than 0.5 can be deemed to be of a strong type. Applying that threshold, the data reported by Dowling et al. in [42] indicate that for the vast majority of bytecode instruction categories there exist a strong association between static and dynamic frequencies. With

that said, the 0.5 threshold indicates that no less than 25% of the variation in dynamic frequencies is accounted for by their associated static frequencies. In a larger model, static bytecode frequencies might help in explaining some of the unaccounted for variation. However, no statistical significance is reported in regard to the correlation findings.

Focusing specifically on JVM benchmarking, the contribution of API library code, and differences associated with Java source-to-bytecode compilers, Gregg et al. [151] extended the earlier analysis reported in [41]. An important distinction between the two analyses concerns the measurement of API and non-API code. The earlier analysis considered the proportion of unique methods associated with API and non-API code, whereas the subsequent study examined the dynamic frequency with which these methods were invoked. For the SPEC JVM98 workloads, although up to 60% of the identified methods targeted API code, API method invocations accounted for approximately 40% of dynamically executed method calls. Furthermore, methods targeting API code accounted for approximately 33% of all dynamically executed bytecodes. Among non-native API method invocations, the `java.lang` and `java.util` packages were particularly prominent.

Gregg et al. [151] also examined the distribution of executed bytecodes across individual API packages. Different packages accounted for different proportions of dynamically executed bytecodes; however, no substantial difference was observed between the proportion of methods executed within each package and the corresponding proportion of bytecodes executed within that package. The study additionally examined characteristics of the method stack frame, particularly the local variable array. Although the size of the local variable array is determined statically at compile time, its observed dynamic characteristics have implications for the use of specialised bytecode instructions. The results indicated that the number of parameters passed to methods was predominantly no greater than four. Approximately 34% of executed methods also contained no temporary variables, indicating that these methods operated primarily on variables supplied

as parameters and their subsequent manipulation.

Consistent with earlier findings concerning differences between Java source-to-bytecode compilers [152], Gregg et al. [151] found that compiler choice could contribute substantially to bytecode bloat. They attributed the observed differences to the more aggressive optimisation strategies employed by the earlier Java source-to-bytecode compiler used to produce the bytecode distributed with the SPEC JVM98 benchmark suite. Importantly, Gregg et al. argued that these differences would not have been apparent from static analysis of the source code alone, demonstrating the additional information provided by dynamic analysis. Their findings therefore indicate that different Java source-to-bytecode compilers employ different optimisation strategies, with corresponding differences in the characteristics of the generated bytecode.

As can be seen from the previous literature, the modelling of JVM behaviour had primarily focused on understanding the characteristics associated with the execution of individual Java bytecode instructions. To a certain degree, and although offering significant insights into JVM performance, a large part of the dynamic semantic behaviour of the JVM cannot be captured at the individual bytecode level. The work of O'Donoghue et al. in [44, 45, 47] extend the previous work and contribute to a deeper level of understanding in regards to the effects that workload category, encoded within their respective bytecode, have on the JVM. Their work applies the theories developed in the field of natural language processing to the language of Java bytecode. In their first paper [44] they communicate the results associated with a bi-gram analysis of bytecode sequences. Their analysis is based on the results captured from the execution of the Java Grande and SPEC JVM98 workloads. Their results emphasise an important point, in that unweighted bytecode frequency analysis can skew underlying characteristics. Their findings have implications in regard to super-instruction implementation within the JVM and the specialising of the dynamic specialisation of the JVM, in that not all workload's frequently executed bytecode bi-grams appear in the top-ten

ranking. With that said, a number of bi-grams occurring in the pooled bi-gram analysis do appear in the individual workload top-ten rankings. For example, the bytecode sequence `aload_0`, `getfield` is the most frequently executed bytecode, and excluding a single benchmark workload, is ranked first for all others. To emphasise magnitude of ranking differences, and in particular from an imputation from their reported results, the bi-gram `invokevirtual` and **aload_0** is the second most frequently executed pair across all pooled observations. However, when its ranking is averaged across the individual workloads, the `invokevirtual`, `aload_0` bi-gram is ranked substantially lower, demonstrating that its second-place position within the pooled analysis is disproportionately influenced by its execution frequency within particular workloads.

In their second paper [45] O'Donoghue et al. extend on their original methodology by characterising n-gram bytecode sequences. In addition, they encode those frequently occurring n-grams as super-instructions within a custom JVM. Their findings indicate better performance gains for a switch-based interpreter compared to a threaded interpreter, a result they expected, as the threaded interpreter is explicitly optimised to remove the penalty associated with instruction dispatch. Their findings agree with the communications of Proebsting et al. in [132], in that diminishing returns are observed when more superinstructions are implemented.

As noted by Davis et al. [153] the execution of virtual machine instructions has three dominant costs, instruction dispatch, the accessing of operands, and the cost of performing its computations. In addition, those costs are shared by both virtual register and stack machines. However, the cost associated with undertaking a computation can be substantially smaller for virtual register machines, due in part to the the availability of specialised instructions for the accessing of operands. For example, and as outlined by Davis et al., a local variable assignment like, `a = b + c` is encoded in bytecode as, `iload c`, `iload b`, `iadd`, `istore a`, whereas it can be encoded as a single instruction within a virtual register machine, `iadd a, b, c`. The work of Davis et al. in [153] assessed the performance difference

between virtual stack code and register machine code. Their work involved the development of a translator to translate stack code to register code. The first phase mapping each bytecode to appropriate register instructions. a side-effect of the technique was the introduction of many redundant register moves. Their implementation also includes copy propagation algorithms to eliminate redundant register move instructions. Their special handling of invoke instructions, and in particular, their passed parameters, results in the a large increase in the number of additional bytecode load instructions. However, the technique allows their copy propagation algorithm to eliminate a large proportion of register move instructions. The increase in bytecode loads is balanced by the elimination of indirect branch instructions, and although their reported results are based on an evaluation of the number of and type of instructions, loads and indirect branches, an actual evaluation of the execution time performance improvements were not presented.

Adding to previous findings on the characteristics of method invocations, Gregg et al. in [46] present their findings on their targets, both to API and non-API code, in addition to the extent of polymorphicity of virtual method invocations. Their work has implications in regard to method inlining, which more aggressive class hierarchy profiling required by the JIT compiler when virtual calls have multiple targets. Interestingly, their results show contrasting features between Java Grande workloads and those composing the SPEC JVM98 benchmark suite. In that invoke virtual invocations dominant within the SPEC applications, and significantly lower proportions within the Java Grande benchmark applications. Indeed the Java Grande applications have a high proportion of invoke static method invocations, an indication of a more procedural style of programming. Another interesting contrast between both collections of workloads, is associated with the targeting of API code, the Java Grande applications involving a high proportion of API native code, whereas the SPEC JVM98 applications primarily target non-native API code. Their findings showed that monomorphic invocation behaviour is a predominant feature of Grande applications, with the SPEC JVM98

workloads exhibiting a greater degree of polymorphism. The communications of Gregg et al. also consider method bytecode size distributions, an important metric also in regard to method inlining, as well as a characteristic of object oriented programming, with smaller methods being an obvious candidate for method inlining. Their findings follow a similar pattern to their other findings, in that their is a significant difference between the characteristics of the Grande workloads in contrast to the SPEC JVM98 workloads. The method size distributions show that SPEC JVM98 application code methods are considerably smaller than the Java Grande the method comprising its applications code.

As communicated by Dujmovic et al. in [36], early work on bytecode characterisation had been void of a formal framework from which the analysis could be positioned. In that regard, Dujmovic introduced a formal framework that offers multiple perspectives on the 201 bytecode instructions. The framework enables a deeper understanding of different aspects of JVM behaviour by abstracting bytecode instructions. By grouping bytecode instructions into well-defined categories, it becomes possible to compare JVM workloads, as each workload emphasises different functional units of the JVM. Consequently, these groupings serve as a "lens" through which the JVM can be analysed.

The JVM specification itself includes one such categorisation, defining 11 instruction categories: Constants, Loads, Stores, Stack operation, Math operations, Conversions, Comparisons, References, Control, Extended, and Reserved. The names of these categories reflect their implied functions.

Workloads vary considerably, and considering their effect on power consumption is essential. The impact of workload variation has been widely researched, although primarily from the perspective of the JVM interpreter. Early analysis of workload characteristics focused on the static and dynamic modelling of bytecode instruction frequencies, with the objective of identifying the degree of association between these two perspectives. This focus on workload characterisation is consistent with the broader literature, where workload behaviour has been shown to influence both

software energy consumption [154] and JVM memory-management behaviour [50].

In [155], Dujmovic et al. presents and develops a number of comprehensive whitebox and blackbox models of benchmark similarity and difference metrics. The motivation for the model development was toward the design of universal benchmark suites, and in particular, the development of tools that would aid in the identification of workloads that covered as much of program space and that also minimised the similarity between those workloads. Continuing that work, Dujmovic et al. in [156] applies those previously developed metric models in order to analyse the progression of SPEC benchmarks distributed as SPEC89, SPEC92 AND SPEC95 benchmarks. Their analysis highlights the utility associated with similarity and difference metrics in regard to assessing, not just a benchmark collections workloads, but how those benchmarks workloads have developed across suite generations. Their results are interesting in from a number of perspectives, first, the size of the respective benchmark suites, specifically the proportion of program space that they cover, showed that the early releases significantly under represented program space, however, the latest analysed generation SPEC95 exhibited an extremely high level of program space coverage. In regard to redundancy, a number of the suites showed a high level of redundant workloads. In conclusion, Dujmovic et al. emphasise that equipped with difference metrics that the inadequacies associated with benchmark suites can be measured, which will provide opportunities for their enhancement before their release to the systems performance analysis community. Although not specifically dealing with Java benchmark workloads, the work of Dujmovic et al. is without much effort generalisable to such.

In [157], Dujmovic et al. expands on the foundations set in their previous communications, specifically focusing on the development of a "Universal" benchmark suite through the definition of a suite topology. Their analysis includes an assessment of the SPECjvm98 benchmark suite. The results of the analysis, in that regard, suggest that SPECjvm 98 is a well-formed collection of benchmark

workloads, with very low similarity between workloads. However, from a size perspective, the benchmark still has it deficiencies. The power of Dujmovic's difference model is exemplified from the analysis of the SPEC CINT92. That analysis shows that semantically different workloads, for example, `gcc` and `li` (Lisp), are only slightly different from an instruction perspective, and that SPEC CINT92 is an example of a benchmark which includes redundant workloads.

In regard to modelling of JRE JVM workloads, early work focused on producing a framework upon which quantitative analysis of the JVM could be supported. In particular, Dujmovic et al. in [158] present one such framework upon which Java workload characterisation can be undertaken. Their results show that different characterisation granularities can undercover relationships that might be masked by an alternative instruction grouping strategy, in particular, Dujmovic et al. highlight that different characterisation granularities can uncover characteristics that are invariant at that level. The framework, has two broad dimensions, computation type and data type, both types having four associated grouping levels. Expanding on their workload characterisation models, Dujmovic et al. in [36] highlight that communications in regard to workload behaviour had in many cases been absent of any rational for the choices of instruction cluster grouping relied upon within their analysis.

3.5.1 Limitations of Existing Literature

Despite a rich history of JVM workload modelling, several important gaps remain. Early research (e.g., Davidson et al.; Bowers et al.; Waldron et al.) provided foundational insight into the static and dynamic characteristics of JVM bytecode and stack behaviour, laying the groundwork for modelling interpreter and method invocation efficiency. However, most studies have relied on a limited set of benchmarks, predominantly SPEC JVM98 and Java Grande, raising concerns about the generalisability of their findings to more diverse workloads representative of Java application domains involving cloud computing and machine learning.

Furthermore, while formal frameworks and instruction clustering models (as articulated by Dujmovic et al.) have advanced the understanding of workload similarity and benchmark suite quality, their application to power use and energy consumption remains underexplored. he focus has been primarily on execution time and computational performance, with limited empirical work examining relationships between bytecode frequency, method invocation patterns, and direct measurements of power use and energy consumption on physical hardware.

Additionally, there is a persistent challenge in transferring static and dynamic workload models from historical JVM versions or legacy hardware to modern multi-core, heterogeneous, or energy-constrained platforms. The degree to which findings about, for example, stack depth, API call frequency, or superinstruction effectiveness, translate to current JVM implementations and architectures has not been rigorously tested. This limits the actionable value of much of the earlier modelling work for current JVM deployment scenarios.

In summary, there remains a notable need for up-to-date, empirically validated workload models that integrate contemporary benchmarks, consider power use and energy consumption as first-class metrics, and are generalisable across modern JVM implementations and hardware architectures.

3.6 JVM Microarchitectural Modelling

Java computational performance is associated not only with the workloads executed and the JVM execution mode employed, including interpretive and tiered execution, but also with the underlying microarchitectural design of the processor. Frequency and voltage scaling represented an important source of processor performance improvement until approximately 2005, while subsequent processor designs have increasingly exploited parallelism through instruction pipelining, speculative execution, hardware multithreading, SIMD and MIMD architectures, and software-level parallelism [24]. These techniques are not necessarily implemented within every processor architecture, and their applicability depends upon the

requirements of the computational problem [25]. Pipelining represents a particularly important architectural technique for increasing processor performance [26]. Consequently, microarchitectural events associated with pipeline disruption and stalls warrant consideration when examining power use and energy consumption, particularly across different workload contexts.

In regards to the JRE and its associated JVM, and specifically communications concentrating on the microarchitectural performance of interpreters, early work focused on challenging the conventional opinion of the time, in that rather than relying on improving hardware support for interpretive execution that a more efficient implementation of an interpreter would be more beneficial [159] and as cited in [134]). Addressing the validity of that opinion, Ertl and Gregg in [134] communicated results associated with efficient interpretation, which showed that efficient interpreters would in fact benefit from changes in microarchitectural strategies, specifically in relation to branch misprediction. Their work set the scene for a position paper [160] that proposed consideration for alternative branch prediction strategies, as well as static and dynamic super-instruction implementation, the use of a register VM, the exploitation of software pipelining - through the encoding of the dispatch code associated with the next instruction within the current instruction, and finally consideration to the penalty associated with cache miss rates - through the movement of those common instruction instances into the same cache lines. The modern JVM certainly delivers in regards to the observation and recommendations communicated by Ertl and Gregg, in their many papers, implementing, threading, code caching, and specialised instructions, for example, in addition to the generation of instruction stubs that encode Java bytecodes in optimised assembly instructions.

Although earlier research focused extensively on improving interpretive execution performance and identified numerous optimisation opportunities, more recent work has emphasised the importance of examining JVM execution across multiple analytical levels. Papadakis et al. [161] characterised the runtime properties of

DaCapo and Renaissance benchmark workloads and provided further insight into their associated cache and microarchitectural activity. Their analysis considered two complementary perspectives. At the higher level, workload characteristics were examined through object allocation and object access measures and their corresponding rates. At the microarchitectural level, the workloads were characterised using cycles per instruction, L1 and L2 data cache measures, last-level cache (LLC) measures, and branch miss rates. Their findings highlighted the limitations associated with analysis at a single level and emphasised the importance of adopting a "multifaceted" perspective to reduce the risk of drawing incomplete or potentially misleading conclusions.

In addition, workloads within the Renaissance Benchmark Suite exhibit a broader range of characteristics than those represented within earlier benchmark suites, notably DaCapo and SPECjvm. Examining these workloads across multiple analytical dimensions, particularly through their associated microarchitectural performance event measures, can therefore provide additional insight into the relationships between workload characteristics and JVM execution.

In this section we present a review of the literature in regard to branch misprediction, L1 cache miss-rates, L2 cache miss-rates, and the paging behaviour of the JVM.

3.6.1 Branch miss-rates

Although not explicitly focused on JVM branch misprediction, the work of Ertl and Gregg [134], subsequently extended in [135], challenged the prevailing view that improvements in interpreter performance should primarily be pursued through software implementation rather than enhanced hardware support for interpretive execution, a position previously expressed by Romer et al. [159] and discussed in [134]. Ertl and Gregg instead examined the relationship between microarchitectural branch prediction strategies and the performance of interpreters employing optimised dispatch mechanisms. Their analysis considered Gforth, OCaml, Scheme48,

Yap, Perl, and XLisp under several alternative branch prediction strategies. The results demonstrated substantial differences in branch prediction accuracy according to predictor strategy, ranging from 71%–88% for profile-guided prediction, 57%–63% for a branch target buffer (BTB), 50%–61% for a BTB2 predictor, and 2%–18% for a two-level predictor [134, 135]. These findings provide an important reference point for the present study by demonstrating that interpreter execution characteristics can exhibit markedly different branch prediction outcomes depending upon the underlying predictor strategy.

The early work of Radhakrishnan et al. in [162] focusing on Java JVM performance communicated one of the first comparisons of the microarchitectural performance of the JVM, comparing interpretive execution to JIT execution. They present results concerning branch prediction efficiency. Their results showed a clear difference between interpretive execution and JIT execution, with interpretive execution performing considerably worse across all benchmark workloads and the four branch prediction strategies. In particular, a 2-bit predictor, a 1-level branch history table, the GShare predictor, and a 2-level predictor. Branch misprediction associated with interpretive execution ranged between 13% and 62%, depending on workload and prediction mechanism. JIT execution performed better, with branch misprediction ranging between 9% and 41%.

A survey of JVM optimisations by Davis et al. in [56] reported on, among other things, the state of performance at the time regarding switch-based and threaded interpreters. Their review highlighted the magnitude of the challenge in improving the performance of traditional switch-based interpreters, in that switch-based interpretation spends approximately 70% of execution time dealing with branch misprediction. In the case of switch interpretation, up to 90% of branches are mispredicted; in threaded interpreters, up to 60% of branches are mispredicted. Davis's survey also communicated that the branch prediction implementation impacts overall performance. Two-level branch target buffers provide the best branch prediction performance, with an accuracy ranging between 82% and 98%.

Also focusing on the branch misprediction penalty, Berndl et al. in [136] considered an alternative dispatching strategy to the optimisations mentioned above; in particular, Berndl et al. communicate their findings in regards to context-threading. Their results expressed relative to Java and OCaml. A dispatching optimisation that However, aggressive superinstruction implementations can increase cache miss-rates, as discussed in [136]. Berndl et al. [136] proposed context-threaded interpretation of Java bytecode. The optimisation eliminates indirect branches by introducing subroutine calls and a return from that subroutine call. Those instructions are optimised on modern architectures and, as reported by Berndl et al., outperform direct-threaded code that heavily relies on indirect branches.

The work of Li et al. in [163] differentiated indirect branch instructions based on the number of destinations they targeted. Their results show that polymorphic indirect branches account for over 50% of the branches executed by the JVM interpreter. In the case of JIT execution, 20% of indirect branches are polymorphic. Their findings show significant variations in the degree of dynamic polymorphism, depending on the benchmark workload. From a static perspective, the utilised workloads (SpecJVM98) indirect branch characteristics are predominately monomorphic, irrespective of JVM execution mode, with monomorphic indirect branches accounting for more than 95%. Their findings concerning branch prediction performance showed significant performance differences between interpretive and JIT execution. Interpretive execution branch miss-rates ranging between 20% and 96%, and between 7% and 27% for JIT execution (percentages imputed from bar charts). In addition, they identify that a small number of "critical" polymorphic indirect branches are responsible for most mispredictions observed. Their results showed that a typical branch target buffer implementation is associated with branch misprediction rates between 3.2% and 35% for interpretive execution and between 1.5% and 3.1% for JIT execution. Their re-hashable BTB showed considerable reductions in branch misprediction rates for interpretive

execution, with those magnitudes reported ranging between 1.7% and 11.7% and between 0.9% and 2.9% for JIT execution.

Later work, also concentrating on optimisations targeted at improving virtual machine indirect branch instructions performance, particularly the effects of those optimisations on branch target buffer performance, was communicated by Casey et al. in [54]. In particular, their work examined the performance of the branch target buffer after the implementation of superinstructions and instruction replication. Their work demonstrated significant performance improvements regarding the accurate prediction of the targets of indirect branch instructions. Although their results showed significant improvements in static replication and superinstruction transformations, their dynamic implementations outperformed their static.

The more recent work of Rohou et al. in [164] provides a baseline from which we can benchmark contemporary JRE JVM branch prediction performance. Their work considers the performance of three switch-based interpreters: Javascript, Python, and the Common Language Infrastructure. Their work was motivated by the prior effort undertaken regarding interpreter threading. Their investigations assessed if there was evidence to infer that modern branch prediction strategies have closed the gap between the prediction of the indirect branches associated with switch-based interpreters and their threaded counterparts. Their results show that architecture is a factor; they considered three Intel architectures, Nehalem, Sandy Bridge, and Haswell. The more recent architecture Haswell was associated with the best branch prediction performance. The Python interpreter was associated with misprediction rates of 1.28%, 0.35%, and 0.14% across the three architectures, respectively. Javascript had the second-best performance, 1.36%, 0.63%, and 0.17% across the architectures. CLI was The worst-performing interpreter with branch misprediction rates of 1.75%, 1.48%, and 0.3%. We imputed percentages from the misprediction rate per kilo instructions reported in their work. They conclude that modern microarchitectural branch prediction strategies achieve significantly better predictions than their predecessors.

In other recent work focusing on addressing the branch prediction underperformance associated with interpretive execution, Kim et al. in [165] assess the efficiency of a Python and Lua interpreter, implementing short-circuit dispatch and contrasting it against direct threaded implementations. The modifications are a microarchitectural optimisation that maps the bytecode jump table to a branch target buffer (BTB), effectively caching the bytecode. Fetched bytecodes are looked up in the BTB using their opcode. If they hit, execution directly transfers to the target address stored in the BTB. In the case of a BTB miss, execution continues following the "original path" [165], and the BTB is updated with the new target address. The findings Kim et al. reported on the results gathered from the simulation of the ARM Cortex-a5 architecture on the gem5 simulator. Their results are related to the performance of Lua and Javascript interpreter behaviour, with garbage collection disabled. Relative to threaded interpreter execution, short-circuit dispatch performs significantly better with more accurate branch prediction. Imputation of the results presented in [165] at a course-grained level shows that threaded interpretation has a branch misprediction rate between 0.3% and 2.4%, while short-circuit dispatch is associated with misprediction rates between 0,1% and 1.9%, for the Lua interpreter. Regarding the Javascript interpreter, threaded dispatch was associated with branch miss-rates between 1.1% and 3.3%, with short-circuit dispatch ranging between 0.6% and 2.5%.

The exploitation of microprocessor caching is not necessarily new. Prior work on the caching of top-of-stack variables was considered and communicated by Ertl et al. in [166]. The implementation removes the significant overhead associated with stack manipulation instructions. Their findings showed a decrease in the number of load and store operations as the number of available registers increased, although the gain observed by Ertl et al. was offset by introducing a more significant proportion of move instructions as the number of available registers increased.

3.6.2 Data Cache Miss-Rates

In addition to examining the branch prediction behaviour of the Java bytecode-to-native compiler *Caffeine* relative to interpretive execution, Hsieh et al. [18] also investigated JVM L1 data cache performance. Analysis of the data presented, including the derivation of miss-rate percentages from the reported read and write miss statistics, indicates that L1 data cache miss-rates ranged between approximately 0.05% and 1.42% under interpretive execution. These findings provide an early benchmark for evaluating JVM data cache behaviour and demonstrate that the interpreter can exhibit relatively low L1 data cache miss-rates under certain workload conditions.

Subsequent work by Radhakrishnan et al. [162] expanded this analysis by comparing interpretive and JIT-compiled execution using the SPECjvm98 benchmark suite. Their findings indicate that interpreter L1 data cache miss-rates ranged between 0.4% and 3.1%, whereas JIT execution exhibited higher miss-rates ranging between 3.2% and 8.3%. The authors attribute this deterioration in data cache performance to the behaviour of the JIT compiler. Specifically, following compilation, frequently executed bytecode sequences are transformed into native machine code and relocated from the data cache into the instruction cache. Consequently, the composition of the data cache changes, altering the balance between bytecode accesses and application data accesses. Interestingly, the same study also reported higher instruction cache miss-rates under JIT execution, suggesting that the relationship between compilation and cache behaviour is more complex than a simple migration of bytecode from one cache hierarchy to another.

Further insight into these observations is provided by Hu et al. [167], who examined cache performance across the constituent phases of JVM execution. Their analysis concluded that the execution engine largely determines instruction cache behaviour, whereas garbage collection is responsible for a substantial proportion of data cache misses. According to Hu et al., the elevated data cache miss-rates observed during garbage collection arise from the large working sets involved

and the extensive pointer-chasing behaviour characteristic of tracing garbage collection algorithms. Their study further reported that data cache performance under JIT execution was generally worse than under interpretive execution. The authors argue that this deterioration stems partly from a reduction in bytecode accesses following compilation and partly from compulsory write misses introduced during native code installation. These observations provide a microarchitectural explanation for the higher L1 data cache miss-rates reported by Radhakrishnan et al. [162].

The importance of cache behaviour within JVM execution has also been highlighted by Ertl et al. [134, 135], Gregg et al. [160], and Davis et al. [56], whose work focused on the microarchitectural performance measures influencing JVM interpreter performance. A recurring theme throughout this literature is that JVM bytecode is treated as data by the interpreter and therefore competes with application data for space within the data cache hierarchy. As noted by Hsieh et al. [18], this behaviour increases the likelihood of data cache contention and contributes to the distinctive cache behaviour exhibited by interpretive execution.

More recently, Papadakis et al. [161] examined the runtime characteristics of DaCapo and Renaissance benchmark workloads using the `MaxineVM` [168]. Their results indicate L1 data cache miss-rates ranging between approximately 0.8% and 6.8% across the Renaissance benchmark suite. These findings demonstrate that data cache performance remains an important characteristic of contemporary JVM workloads and suggest that cache behaviour continues to play a significant role in explaining differences in runtime performance, power use, and energy consumption.

3.6.3 Limitations of Existing Literature

Despite decades of microarchitectural research into JVM performance, significant gaps remain in our understanding of how processor-level features impact power use and energy consumption during JVM execution. Foundational work by Ertl

and Gregg [134, 135] examined interpreter performance and branch prediction strategies, focusing primarily on execution time and cache behaviour and providing only limited or indirect insight into power use and energy consumption. Later contributions, for example Papadakis et al. [161], demonstrate the complex and workload-dependent nature of modern cache and branch prediction effects, but even these studies rarely quantify how such microarchitectural behaviours translate into differences in power use and energy consumption across representative, contemporary JVM workloads.

Furthermore, although early work by Radhakrishnan et al.[162] and Hsieh et al.[18] established baseline measurements for cache miss-rates and highlighted key contrasts between interpretive and JIT execution, these findings remain largely confined to performance and do not systematically explore their implications for power use or energy consumption. Where more recent workload suites, particularly Renaissance, have been examined, as demonstrated by Papadakis et al. [161], the emphasis has again been on cache and pipeline performance, with limited direct measurement or modelling of power use and energy consumption. Furthermore, the increasing architectural diversity of JVM target platforms, encompassing ARM and multicore systems and environments in which power and energy constraints are operationally important, limits the extent to which findings obtained on earlier x86-based hardware can be generalised.

Moreover, while a number of works have proposed interpreter and dispatch optimisations to reduce branch mispredictions or exploit cache locality (as demonstrated by Berndl et al.[136] and Kim et al.[165]), the literature seldom follows through with end-to-end evaluations of how these strategies influence power use and energy consumption in real systems. Similarly, the relationships between higher-level software engineering choices involving superinstruction implementation and register-machine architectures and their associated microarchitectural activity and energy consumption remain largely unexplored beyond proof-of-concept and simulation-based studies.

Overall, the literature is characterised by a persistent gap in integrating microarchitectural, workload, and JVM-level studies to build a cohesive, empirical understanding of power use and energy consumption. There is a clear need for studies that couple microarchitectural performance metrics with direct energy and power measurements, spanning both traditional and emerging hardware architectures, and that account for the complex, multifaceted interactions present in modern Java workloads.

3.7 ARM-Based Microarchitectures

As a cost-effective single-board computer, the Raspberry Pi has transitioned away from its initial goal of providing affordable access to computers for teaching purposes to one in which it has become a common driving component within a significant proportion of IoT technology [169]. Raspberry Pi application use cases are wide-ranging, ranging from their integration in applications as diverse as home automation [170], agricultural monitoring and agrorobotics [171, 172, 173, 174], collision avoidance [175], point-of-care technologies [176], image recognition and construction [177, 178], and process automation [179] to mention a few. In most cases, the Raspberry Pi provides the computing infrastructure for the integration of a large collection of sensor devices and the subsequent uploading of the associated captured data or in situ processing of that data. A wide array of sensors are in common usage capturing changes in temperature, light, humidity, sound and ultrasound, pressure, motion, proximity, position, force and touch, magnetic forces, angular velocity, acceleration, and pH, for example. In addition, the range of Raspberry Pi deployment contexts has expanded to encompass off-grid and power-constrained environments associated with agriculture [180], weather sensing, security surveillance, and military and defence applications. More recently, Raspberry Pi platforms have also become increasingly relevant to the broader shift towards edge computing.

Investigations of power use in single-board computers, particularly the Rasp-

berry Pi family, are well represented within the literature. Recent research has examined the relationships between overclocking, underclocking, operating system selection, and observed power use, notably in the work of Astudillo-Salinas, Barrera-Salamea, Vázquez-Rodas, and Solano-Quinde [181]. Other works have focused on the regression of system power consumption on microarchitectural performance counters, for example, [182, 183]. In addition, Paniego et al. in [183] isolate the effects of parallelism, as measured by the number of threads, and show an increase in power consumption levels moving from single-threaded execution to dual-threaded and to quad-threaded execution. Those assessments have been relative to older, more persistent benchmark applications workloads, for example, those comprising the NAS [184], RODINIA [185], and LINPACK [186] benchmark suites. The work of Paniego et al. in both [182, 183] indicates that architectural level performance counters, and the degree of inherent parallelism, are parameters that account for a proportion of the variation in power consumption levels of applications running on Raspberry Pi generation boards.

Power consumption model building focused on linear regression and was identified as providing a degree of predictive power, [182, 183, 154, 187]. More recently, support vector regression, Gaussian regression, and kernel ridge regression have been applied to the development of power use models for Raspberry Pi single-board computers. The results of Paniego et al. in [188] indicate that support vector regression outperforms all other models regarding mean error reduction, although the time to train such a model is many orders of magnitude greater than multiple regression. Lee and Brooks, in [189], question the efficacy of linear models in predicting power consumption and have suggested that power consumption modelling should consider non-linearity within the specification. With that said, Kaup, Gottschling, and Hausheer in [190] had identified a linear relationship between power consumption and CPU utilisation, for example. Their work primarily focused on developing a power consumption model for the Raspberry Pi. Benchmarking the Raspberry Pi against desktop computers, laptop computers,

tablet devices, and smartphones was the focus of Bekaroo and Santokhee in [191]. Their work highlights the power consumption efficiency of the Raspberry Pi relative to other devices.

The lightweight and flexible nature of the Raspberry Pi development board has seen it form the basis of homogeneous clusters in [192, 193, 194, 187]. Collectively, their assessments of power consumption characteristics are associated with loads from the execution of High-Performance Computing (HPC) benchmarks, namely, High-Performance Linpak (HPL) [195] and STREAM [196], a memory performance benchmark, and from the MiniMD [197], a computation-intensive benchmark simulating the interaction of atomic structures, as well as a collection of data structure sorting algorithms.

The effect of including software within an operating system build has been considered by Shizukuishi and Matsubara in [198]. Their findings show that deactivating some kernel build configuration options can statistically significantly affect overall system memory usage. However, they observed no significant effect concerning power consumption. Amirtharaj, Groot, and Dezfouli in [199] undertook similar work but focused on duty-cycle performance during boot-up and the effect of thinning the modules started in the boot-up phase, their results suggesting that unit configuration can have a significant impact on overall energy consumption across boot-up.

3.8 Related Work Gaps and Contributions

Despite a substantial and varied body of research on the performance and efficiency of Java Virtual Machine (JVM) systems, significant limitations persist in the current literature, limitations that map directly onto the five key pillars of this study's theoretical framework, namley: Architecture Parameters, Virtual Machine Parameters, Compiler Parameters, Memory Management Parameters, and Workload Parameters.

Architecture Parameters Processor architecture, cache design, and branch

prediction strategies, together with the increasing prevalence of heterogeneous computing platforms, have been shown to be associated with differences in JVM computational performance. However, most foundational work (e.g., Ertl & Gregg; Papadakis et al.) has focused predominantly on execution time and cache behaviour, providing only limited or indirect insight into their effects on power use and energy consumption. The literature rarely couples microarchitectural performance event metrics with direct power use or energy consumption measurements, and often generalises from legacy x86-based systems, leaving open questions about the applicability of results to contemporary multicore, ARM, and other modern architectures. Furthermore, the growing diversity of hardware in JVM deployment environments further complicates this gap, underscoring the need for systematic, empirical studies that integrate architectural, JVM-level, and workload perspectives to yield actionable insights on energy consumption.

Virtual Machine Parameters While extensive research has catalogued interpreter and just-in-time (JIT) optimisation strategies (Casey et al.; Simonis et al.; Rose et al.), there remains limited empirical evidence examining relationships between virtual machine configuration dimensions, particularly JVM execution mode, and measurable differences in power use and energy consumption across different JVM implementations. Most existing studies concentrate on execution time or throughput, with only a handful addressing how modern interpreters, template-based dispatch, or tiered execution interact with underlying hardware to affect system-level power use and energy consumption. Data-driven guidance for selecting execution modes under varying constraints remains scarce, limiting practical decision-making for system architects targeting sustainable or energy-constrained environments.

Compiler Parameters Although it is well-established that compiler toolchain selection and optimisation strategies significantly affect application performance (Marcon et al.; Pallister et al.), the direct impact of compiler parameters on the power use and energy consumption of JVM and JRE binaries is underexplored.

Few studies extend analysis beyond execution time, and those that do (e.g., Pallister et al.) typically focus on embedded systems or application-level binaries rather than JVM-specific contexts. Recent evidence (Lambert et al.) demonstrates that JVM interpreter performance can be sensitive to build toolchain choice, yet systematic evaluation of compiler version, flag configuration, and cross-compiler comparisons in terms of energy outcomes remains lacking. This gap is particularly critical in real-world, large-scale, or data-centre deployments, where even marginal efficiency gains can have substantial aggregate effects.

Memory Management Parameters Research on JVM garbage collection has traditionally emphasised throughput, latency, and memory footprint (Blackburn et al.; Dykstra et al.), while systematic evaluation of the relationships between garbage collector variant, power use, and energy consumption remains comparatively underdeveloped. Early work (Chen et al.; Vijaykrishnan) provided only initial, often simulation-based insights, while more recent studies (Ournani et al.; Shimchenko et al.) have begun to directly incorporate power use and energy consumption metrics. However, coverage remains limited in terms of JVM distributions, collector variants, and workload diversity, and there is inconsistency in findings regarding which configurations are most efficient. This lack of comprehensive, comparative research constrains the ability to make informed, data-driven choices for energy-aware JVM deployments.

Workload Parameters Workload modelling has been foundational in understanding JVM behaviour, yet most studies (Davidson et al.; Waldron et al.; Dujmovic et al.) have relied on a narrow set of legacy benchmarks, raising concerns about the relevance and generalisability of their findings to modern, heterogeneous workloads encountered in cloud computing, data analytics, and machine learning domains. While advanced workload clustering and benchmark quality frameworks exist, their application to power use and energy consumption studies is minimal. The persistent challenge of transferring static and dynamic workload models from legacy JVMs or hardware to contemporary platforms further limits the practical

utility of much of this research.

Notably, as ARM-based microarchitectures become increasingly prevalent in both edge computing and large-scale data centre deployments, it is crucial to consider how workload category interact with the distinct profiles of power use and energy consumption on ARM systems. The power use and energy consumption behaviour of JVM workloads may differ significantly between traditional x86 and modern ARM platforms, particularly given ARM's architectural emphasis on energy efficiency. However, the literature offers little empirical insight into how representative Java workloads perform with respect to power use and energy consumption on ARM-based hardware, underscoring an urgent need for targeted studies that bridge this gap.

Taken together, the current literature is characterised by fragmented, performance-centric approaches and outdated or incomplete coverage of power use and energy consumption. No single piece of prior work has systematically integrated the architectural, virtual machine, compiler, memory management, and workload perspectives needed to fully understand and optimise the cumulative effects on power use and energy consumption in JVM-based systems. This work is the first to offer a homogeneous synthesis across all five pillars, to support the evaluation of Java power use and energy consumption.

Chapter 4

Methodology

4.1 Introduction

This chapter presents the methodological framework adopted to investigate the factors influencing JRE JVM power use and energy consumption. Building upon the background and related work presented in Chapters 2 and 3, the chapter details the theoretical foundations, experimental procedures, data collection processes, and analytical techniques used to address the research questions guiding this study. The methodological approach was designed to facilitate systematic evaluation of OpenJDK source versions, GNU GCC build toolchain compiler versions, JVM execution modes, garbage collector variants, workload category, and microarchitectural performance measures, together with their individual and combined effects on JRE JVM power use and energy consumption.

The chapter begins by establishing the theoretical and conceptual frameworks underpinning the study, identifying the principal factors shown within the literature to influence managed runtime environments and defining their hypothesised relationships with power use and energy consumption. An experimental framework is then introduced to illustrate how these factors were operationalised within the empirical investigation and how the resulting observations were transformed into datasets suitable for addressing the research questions.

Subsequently, the chapter describes the experimental platform used throughout the investigation, including the Raspberry Pi 4 Model B hardware environment,

operating system configuration, measurement instrumentation, construction of the GNU GCC build toolchains, OpenJDK JRE JVM builds, and the runtime configuration factors defining the experimental configurations under investigation.

The benchmark workloads employed throughout the study are then introduced together with the workload category framework, data collection procedures, and the process through which power use, energy consumption, and microarchitectural observations were integrated into the final experimental dataset.

Finally, the chapter presents the statistical analysis and empirical modelling frameworks underpinning the analyses conducted in Chapters 5, 6, and 7. Collectively, these methodological components provide the foundation for investigating main effects (RQ1), workload-conditioned interaction effects (RQ2), and explanatory models of JRE JVM power use and energy consumption (RQ3).

4.2 Theoretical Framework

In this section, we present the theoretical framework that underpins this research, detailing the rationale behind the selection of all independent variables and their association with the dependent variables: power use and energy consumption.

Early work on benchmarking JRE and JVM behaviour predominantly relied on collections of workloads and methodologies originally developed for modeling C and C++ systems. However, Blackburn et al. [31] argued that these approaches were inadequate for benchmarking managed runtime environments because they failed to capture important characteristics associated with JRE/JVM execution. In response, they proposed a multifaceted methodology that integrates factors associated with the underlying architecture, the compilers involved in code generation, virtual machine characteristics and implementations, memory management strategies, and the nature of the executed workloads.

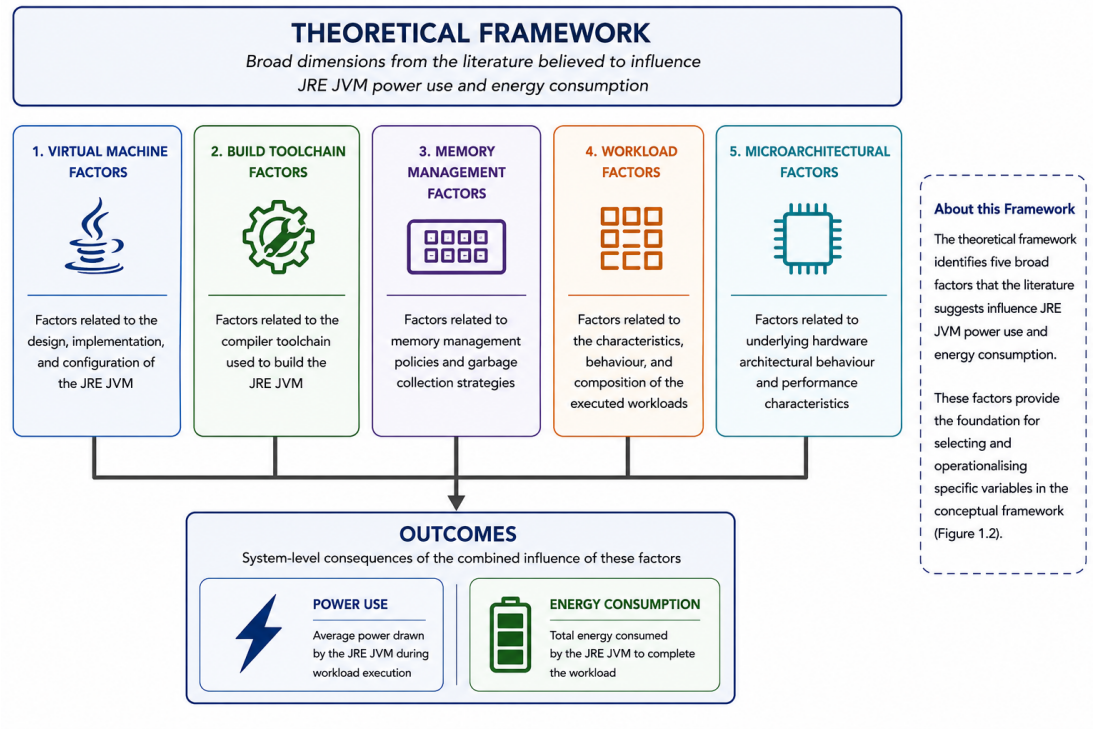


Figure 4.1: Theoretical Framework showing the high-level relationship between primary parameter groups and their collective influence on modelled outcomes.

Our analysis of the power use and energy consumption of the Java Runtime Environment and its associated Virtual Machine follows this established framework, as depicted in Figure 4.1. Moreover, we extend its application by including both variables identified as important in previous research and additional factors that we believe further strengthen the comprehensiveness of our analysis.

The theoretical framework underpinning this research consists of five primary dimensions that have been repeatedly identified within the JVM benchmarking and performance evaluation literature as influencing runtime behaviour. Those dimensions are labelled 1 through 5 in Figure 4.1. These dimensions comprise virtual machine configuration factors, build toolchain compiler factors, memory management factors, workload category, and microarchitectural performance measures. Collectively, these dimensions form the basis of the explanatory framework used throughout the subsequent analyses of JRE JVM power use and energy consumption.

The theoretical framework underpinning this research consists of five primary

dimensions that have been repeatedly identified within the JVM benchmarking and performance evaluation literature as relevant to runtime behaviour. These dimensions, labelled 1 through 5 in Figure 4.1, comprise virtual machine configuration factors, build toolchain compiler factors, memory management factors, workload category, and microarchitectural performance measures. Collectively, they form the basis of the explanatory framework used throughout the subsequent analyses of JRE/JVM power use and energy consumption.

4.3 Conceptual Framework

The conceptual framework underpinning this study distinguishes between manipulated runtime factors, emergent runtime behaviour, and the resulting system outcomes. Specifically, the framework comprises five categories of manipulated factors: JRE/JVM version factors, build toolchain compiler factors, JVM execution mode factors, memory management factors, and workload category. These factors represent the experimental conditions that are directly controlled and varied throughout the study.

The framework further recognises that the effects of these manipulated factors are expressed through runtime execution behaviour, which in turn gives rise to a collection of observable microarchitectural performance event measures. These emergent characteristics, including branch misprediction rate, cache miss rate, and the proportion of page faults, are not directly manipulated by the experimenter but arise from the interaction between the executed workloads, the JRE/JVM configuration, and the underlying hardware platform. Consequently, they are treated as measured emergent runtime variables that provide insight into the mechanisms through which the manipulated factors influence system behaviour.

The conceptual framework therefore proposes that JRE/JVM power use and energy consumption are shaped not only by the direct effects of runtime configuration and workload factors, but also by the emergent microarchitectural behaviour that those factors produce during execution, and which is captured through asso-

ciated microarchitectural performance event measures. The complete framework, illustrating the relationships between manipulated factors, emergent runtime behaviour, and the dependent variables of power use and energy consumption, is presented in Figure 4.2.

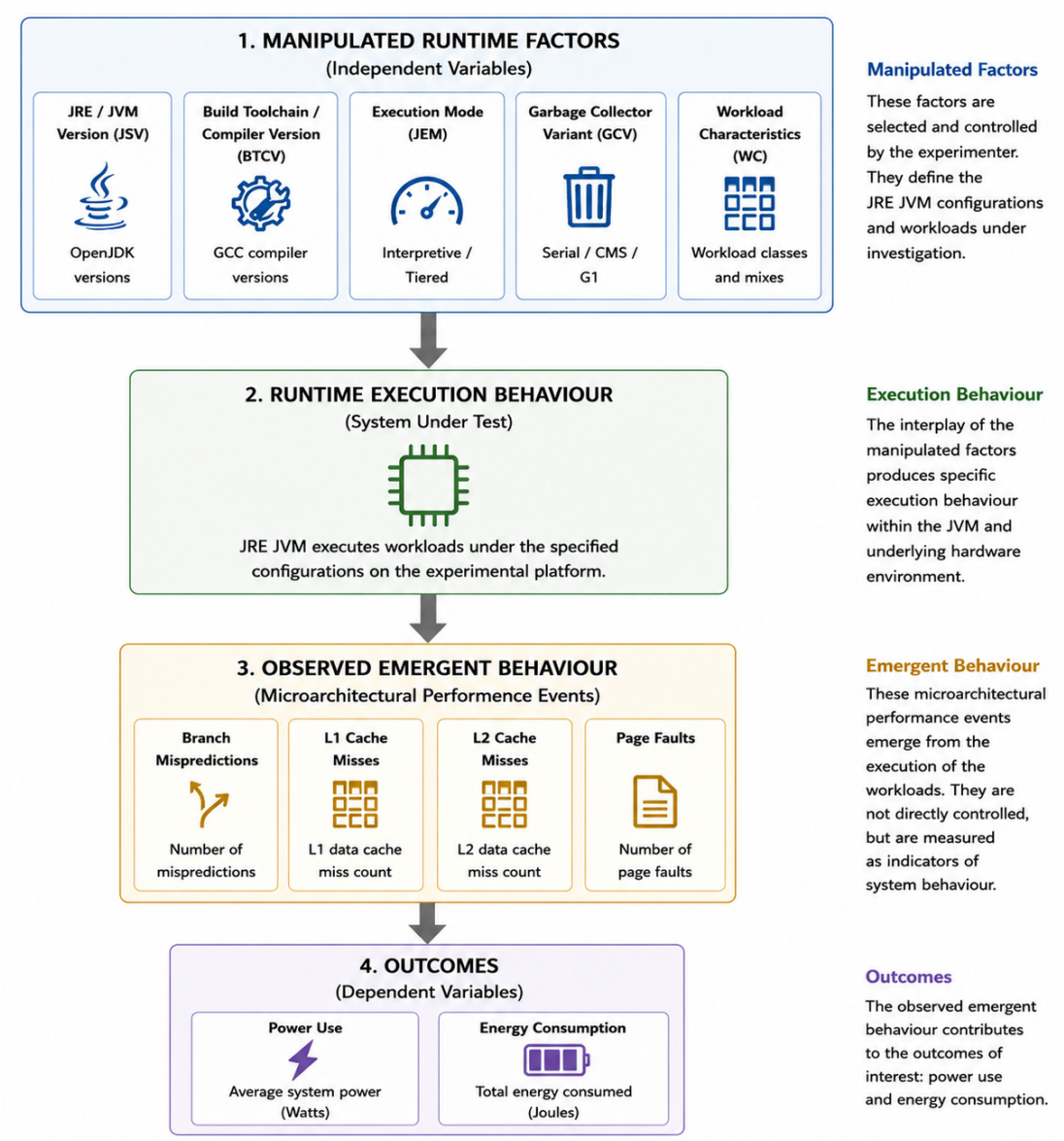


Figure 4.2: Conceptual framework illustrating the key architectural, virtual machine, compiler, memory management, and workload parameters considered in the modelling of system power use and energy consumption, along with associated analytical methods.

4.3.1 Study Variables

The conceptual framework presented in the previous section distinguishes between manipulated runtime factors, emergent runtime behaviour, and the resulting system outcomes. To operationalise this framework, a collection of variables was identified to represent each of these components within the experimental design and subsequent analyses.

The experimental factors comprise OpenJDK source version (JSV), GNU GCC build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and workload category (WC). The first four represent manipulated runtime configuration factors, while workload category provides a classification of the Renaissance workloads employed in the study. These variables represent the experimental conditions that were systematically varied throughout the study. In contrast, the microarchitectural performance event variables (MPE) were not directly manipulated but emerged during benchmark execution as a consequence of the interaction between the workload, the runtime environment, and the underlying hardware platform. Finally, mean power use (P) and energy consumption (E) represent the dependent variables that constitute the principal outcomes of interest.

Collectively, these variables support the investigation of factor-level effects (RQ1), workload-conditioned interaction effects (RQ2), and the development of explanatory models of JRE/JVM power use and energy consumption (RQ3). Table 4.1 summarises the principal variables employed throughout the study, together with their definitions and analytical roles.

Further details regarding the operationalisation of these variables, including their levels of measurement, coding structures, and representation within the analytical dataset, are presented in Section 4.9.

As shown in Table 4.1, the study incorporates distinct classes of independent variables. OpenJDK source version, GNU GCC build toolchain compiler version, JVM execution mode, and garbage collector variant represent experimentally

manipulated runtime configuration factors, while workload category provides a classification of the executed workloads. These variables were systematically represented across the benchmark executions.

Table 4.1: Summary of independent and dependent variables employed throughout the experimental study.

Variable	Description	Type	Role
JSV	OpenJDK source version used to construct the JRE/JVM instance (versions 9–14).	MV	IV
BTCV	GNU GCC compiler version used within the OpenJDK build toolchain (versions 5–8).	MV	IV
JEM	JVM execution mode, consisting of interpretive execution (INT) or tiered just-in-time compilation (TIER).	MV	IV
GCV	Garbage collector variant employed by the JVM.	MV	IV
WC	Renaissance benchmark workload category (Apache, Concurrency, Database, Functional, Scala, Web).	CLV	IV
MPE	Microarchitectural performance event measurements collected during benchmark execution, including cache miss rates, branch mispredictions, and page faults.	MEV	EV
P	Mean power use recorded during benchmark execution, measured in Watts (W).	MeV	DV
E	Total energy consumption recorded during benchmark execution, measured in Watt-hours (Wh).	MeV	DV

Note: MV = Manipulated Variable; CLV = Classification Variable; MEV = Measured Emergent Variable; MeV = Measured Variable; IV = Independent Variable; EV = Explanatory Variable; DV = Dependent Variable.

The second class consists of measured microarchitectural performance event variables, including branch misprediction rates, cache load miss rates, and page fault activity. Unlike the runtime configuration factors, these variables were not directly manipulated prior to execution. Instead, their observed values emerged dynamically during benchmark execution as a consequence of the interaction between the workload, the configured runtime environment, and the underlying hardware architecture.

This distinction is particularly important within the empirical modelling framework developed later in this thesis. Whereas the runtime configuration factors represent experimentally controlled conditions, the microarchitectural variables represent emergent execution characteristics observed during runtime.

Consequently, the statistical models developed in support of Research Question 3 incorporate both experimentally manipulated predictors and emergent runtime execution measures when evaluating the extent to which observed variation in JRE/JVM power use and energy consumption can be accounted for.

4.4 Experimental Framework

The purpose of the experimental framework is to provide an overview of the processes through which the conceptual framework was operationalised within the present study. Whereas the conceptual framework identifies the factors hypothesised to influence JRE JVM power use and energy consumption, the experimental framework describes how those factors were instantiated, measured, and analysed in order to address the research questions. The framework therefore serves as a bridge between the conceptual model presented in the preceding section and the detailed methodological procedures presented throughout the remainder of this chapter.

As illustrated in Figure 4.3, the experimental process commenced with the construction of multiple JRE JVM instances from different combinations of OpenJDK source versions and GNU GCC build toolchain compiler versions. These runtime environments were subsequently configured according to alternative execution modes and garbage collector strategies before being subjected to a common collection of benchmark workloads.

Execution of the benchmark workloads produced three principal forms of data: power use measurements, energy consumption measurements, and microarchitectural performance events. These observations were integrated into a unified experimental dataset used to address RQ1, RQ2, and RQ3 through analysis of main effects, workload-conditioned interactions, and empirical modelling of variation in JRE JVM power use and energy consumption.

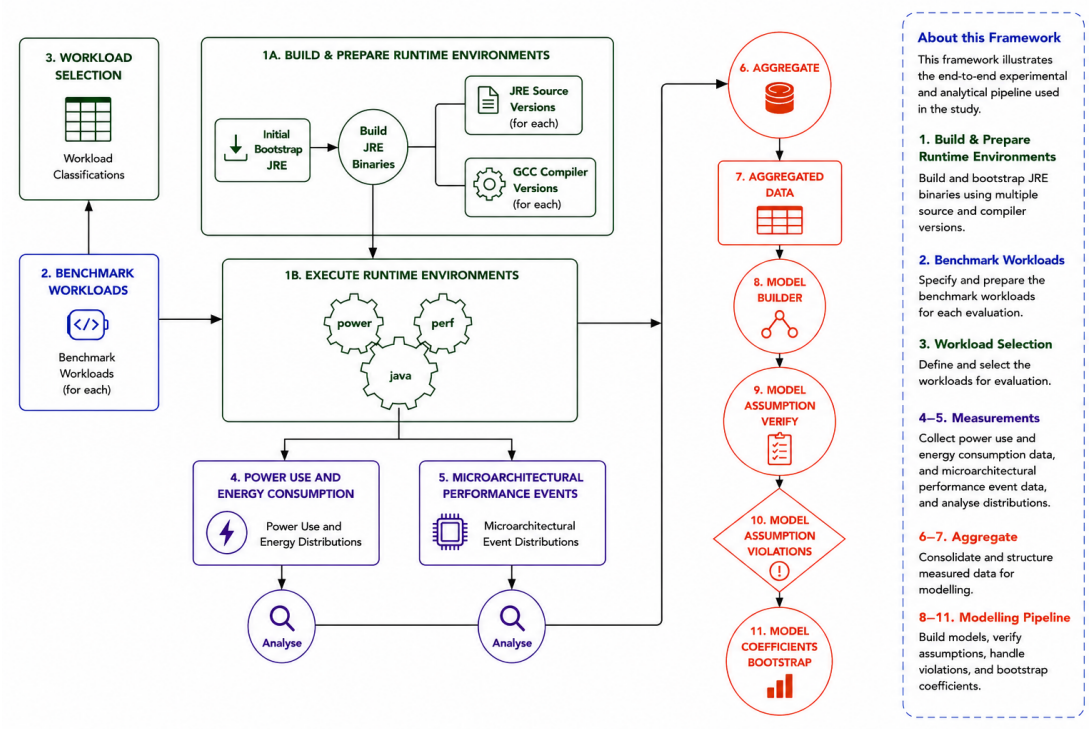


Figure 4.3: Experimental workflow illustrating the construction of JRE JVM instances, benchmark execution, collection of power use, energy consumption, and microarchitectural measurements, and the subsequent statistical analyses used to address RQ1, RQ2, and RQ3.

The experimental workflow can be broadly divided into four phases. The first phase concerns the preparation of the runtime environments, including the construction of OpenJDK JRE/JVM instances from alternative source and compiler version combinations. The second phase focuses on workload preparation and execution, including workload selection, benchmark execution, and the collection of power use, energy consumption, and microarchitectural performance event measurements. The third phase involves the aggregation and integration of the collected measurements into a unified analytical dataset. Finally, the fourth phase concerns the development and evaluation of statistical models, including model construction, assumption verification, treatment of assumption violations where necessary, and bootstrap estimation of model coefficients.

Together, these phases provide a structured pathway from runtime environment construction and benchmark execution through to empirical model development and evaluation.

The individual stages illustrated within Figure 4.3 are discussed in greater detail throughout the remainder of this chapter. Specifically, Sections 4.6–4.7 describe the construction of the runtime environments and workload preparation stages, Sections 4.8–4.9 describe the collection and integration of the experimental measurements, while Sections 4.10–4.11 describe the statistical analyses and modelling procedures used to address RQ1, RQ2, and RQ3.

4.5 Experimental Platform

The purpose of this section is to describe the experimental platform upon which the study was conducted. The platform encompasses the hardware infrastructure, software environment, and measurement apparatus used to construct OpenJDK JRE/JVM runtime environments, execute benchmark workloads, and collect the power use, energy consumption, and microarchitectural performance event measurements analysed throughout this thesis.

Because power use, energy consumption, and microarchitectural performance emerge from the interaction between hardware and software components, careful specification and control of the experimental environment is necessary to ensure the validity, reliability, and reproducibility of the experimental results. Consequently, detailed information is provided regarding the computational platform, operating system environment, runtime software, measurement instrumentation, and supporting software artefacts employed throughout the experimental campaign.

In accordance with the experimental framework presented across Section 4.4 and depicted in Figure 4.3, the platform was designed to provide a controlled environment in which OpenJDK source versions, compiler toolchain versions, execution modes, garbage collector variants, and benchmark workloads could be systematically manipulated while simultaneously capturing power use, energy consumption, and microarchitectural performance event measurements.

To provide a consolidated overview of the experimental environment, Table 4.2 summarises the principal hardware, software, runtime, network, measurement,

and execution settings employed throughout the study. The table is intended to establish the common experimental configuration in a single location before the individual components are described in greater detail within the subsequent subsections. It also distinguishes those conditions that were standardised across the experimental platform from those, such as processor temperature and clock frequency, that were not explicitly controlled or continuously recorded and are therefore treated as sources of residual experimental variation.

Table 4.2: Summary of the common experimental configuration employed throughout the study

Configuration Element	**Experimental Setting**
Hardware Platform	Ten Raspberry Pi 4 Model B development boards incorporating the Broadcom BCM2711 system-on-chip, four ARM Cortex-A72 processor cores, and 4 GB RAM. One node operated as the master node, while nine nodes were used as experimental workers. All nine worker nodes were used for microarchitectural performance event collection. Electrical measurements were performed sequentially on individual worker nodes using a single Joulescope JS110.
Operating System	Raspbian Buster Lite, based on Debian GNU/Linux, using Linux kernel version 4.19. A common cloned operating-system image was deployed across all experimental nodes.
Storage	Identical 32 GB SanDisk Ultra microSDHC A1 memory cards imaged from a common master operating-system image.

Continued on next page

Table 4.2 – continued from previous page

Configuration Element	Experimental Setting
Network Infrastructure	TP-Link TL-SG116 16-port Gigabit Ethernet switch. SSH was enabled for administration and workload deployment, with no active SSH sessions maintained during measured benchmark execution.
Runtime Platform	OpenJDK JRE/JVM source versions 9.0.4 through 14.0.0, constructed from source using GNU GCC build toolchain compiler versions 5.5.0 through 8.3.0.
Runtime Configuration	Interpretive JVM execution was specified using `-Xint`, while tiered JVM execution was specified using `-Xmixed`. Garbage collector variants were selected using `-XX:+UseSerialGC` for the Serial Garbage Collector (SGC), `-XX:+UseParallelGC` for the Parallel Garbage Collector (PGC), `-XX:+UseConcMarkSweepGC` for the Concurrent Mark-Sweep Garbage Collector (CGC), `-XX:+UseG1GC` for the Garbage First Garbage Collector (G1GC), and the combined `-XX:+UseConcMarkSweepGC -XX:+UseParNewGC` configuration for CMS+ParNew (CPGC), where supported by the corresponding OpenJDK version. A common 2 GB initial and maximum heap allocation was specified using `-Xms2G` and `-Xmx2G`.

Continued on next page

Table 4.2 – continued from previous page

Configuration Element	Experimental Setting
Benchmark Platform	Renaissance Benchmark Suite version 0.11.0, with benchmark execution performed under a common invocation procedure.
Electrical Measurement	Joulescope JS110 Precision DC Energy Analyser positioned in-line between the power supply and the Raspberry Pi under test.
Electrical Acquisition	Simultaneous voltage and current acquisition at up to 2 MSPS with a specified measurement bandwidth of 250 kHz.
Measurement Synchronisation	GPIO pin 17 used as an external hardware trigger, with logic-high and logic-low transitions identifying the beginning and end of the benchmark measurement interval.
Electrical Outcomes	Mean power use, expressed in watts (W), and total energy consumption, expressed in watt-hours (Wh).
Microarchitectural Measurement	Branch misprediction rate, L1 data cache load miss rate, L2 data cache load miss rate, and page fault rate collected separately using Linux `perf`.
Processor Frequency	Processor clock frequency was not fixed or continuously recorded during the retained principal experiments; normal Raspberry Pi processor-frequency management remained active.

Continued on next page

Table 4.2 – continued from previous page

Configuration Element	Experimental Setting
Processor Temperature	Processor temperature was not explicitly controlled or continuously recorded and is therefore treated as a source of residual experimental variation.

The following subsections describe the hardware infrastructure, software environment, and measurement apparatus that collectively supported the construction, execution, and evaluation of the JRE/JVM configurations investigated throughout this study.

4.5.1 Hardware

The hardware infrastructure forms the physical foundation of the experimental platform and was designed to support the construction, execution, and measurement of the JRE/JVM runtime environments evaluated throughout this study. The platform comprises a cluster of Raspberry Pi4 Model B development boards used for benchmark execution, a precision DC energy analyser used for power use and energy consumption measurement, and a dedicated network infrastructure used to coordinate workload deployment, remote administration, and data collection. The following subsections describe each hardware component and its role within the experimental workflow.

Raspberry Pi4 Model B Cluster

The Raspberry Pi 4 Model B was intentionally selected as a controlled ARMv8-A experimental platform for investigating how JRE/JVM runtime configuration decisions influence power use and energy consumption. Although the hardware characteristics of the Raspberry Pi 4 differ substantially from larger server-class ARM processors, particularly in relation to processor scale, cache hierarchy, and

memory subsystem design, the primary objective of this study is not to characterise absolute hardware performance. Rather, the platform provides a stable and reproducible environment in which software-controlled runtime factors can be systematically manipulated and their relative effects observed. As ARM-based architectures continue to expand across embedded, edge, and cloud computing environments, such controlled experimental analysis remains important for understanding software-induced variation in computational energy demand.

The experimental platform consists of a cluster of ten ARMv8-A based computational nodes built around the Broadcom BCM2711 system-on-chip, incorporating quad-core Cortex-A72 64-bit superscalar processors with a nominal maximum clock frequency of 1.5 GHz. This value represents the nominal maximum processor frequency specified for the Raspberry Pi 4 Model B and should not be interpreted as indicating that processor frequency was fixed at 1.5 GHz during benchmark execution. Processor clock frequency was not explicitly fixed during the retained experimental programme, and normal Raspberry Pi dynamic processor-frequency management remained active.

The Cortex-A72 represents a modern out-of-order ARM microarchitecture incorporating speculative execution, branch prediction, multi-level cache hierarchies, and hardware prefetching mechanisms, thereby providing a realistic contemporary ARM execution environment suitable for controlled investigation of software-induced variation in JRE/JVM power consumption and energy consumption. Although the experimental platform differs in scale from larger ARM deployments, the underlying ARMv8-A architectural mechanisms remain directly relevant to the software–hardware interactions examined in this study.

Raspberry Pi 4 Model B development boards were selected as the physical deployment platform because they provide a stable, reproducible, and experimentally controllable hardware environment, while simultaneously allowing direct low-voltage power instrumentation and automated hardware-level measurement synchronisation through the GPIO interface.

Each Raspberry Pi provides general-purpose input-output (GPIO) functionality through a 40 pin GPIO header. A Raspberry Pi can be powered over a 5V DC USB type C connector or 5v DC GPIO headers. We power our cluster nodes using two five port USB adaptors powered from mains. The maximum tolerable current is 3 Amperes [200]. A specification diagram is presented in Figure 4.4.

Although Raspbian Buster Lite was operated as a headless command-line environment without a graphical desktop or windowing system, the Raspberry Pi graphics subsystem was not disabled at the hardware level. The common `config.txt` configuration employed across all experimental nodes included the `dtoverlay=vc4-fkms-v3d` setting, enabling the VC4/V3D graphics driver stack, together with `max_framebuffers=2`. Consequently, the use of the Lite operating-system distribution should be interpreted as removing graphical desktop workload and associated user-space processes rather than disabling the integrated graphics hardware.

All Raspberry Pi 4 development boards were configured with Secure Shell (SSH) access enabled, allowing the cluster to be administered remotely and facilitating the automated deployment and management of benchmark workloads throughout the experimental measurement phase. However, no active SSH sessions were maintained while benchmark workloads were executing. This ensured that user-initiated network communication did not introduce additional processor, memory, or network-interface activity during the measured execution interval, thereby reducing the potential for SSH-related background activity to influence the observed power use, energy consumption, or microarchitectural performance event measurements.

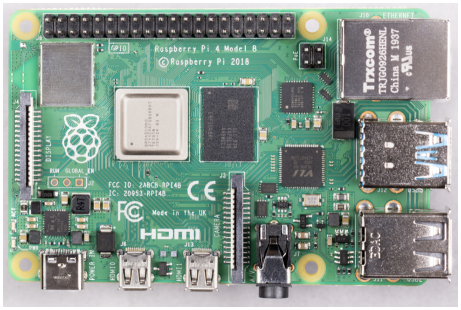


Figure 4.4: Raspberry Pi4 Model B board peripheral device layout. Detailing position of Broadcom BCM2711 processor, RAM, USB Type-C power supply, and network ports. Source: [1]

To ensure consistency across all experimental nodes, each Raspberry Pi was deployed using an identical operating system image and firmware configuration. The experimental platform utilised the Raspbian Buster Lite distribution, a lightweight headless operating system providing only a command-line interface and omitting graphical desktop components. The use of a headless operating system reduced unnecessary background activity and ensured that system resources were dedicated primarily to benchmark execution and measurement collection. The operating system image was cloned across all cluster nodes, thereby ensuring that each experimental platform instance executed under an identical software environment.

In addition, a common Raspberry Pi boot configuration file (`config.txt`) was applied to every node. The configuration remained intentionally close to the default Raspberry Pi firmware settings, with only those parameters necessary to support experimental operation explicitly enabled, including the Inter-Integrated Circuit (I^2C) interface used for hardware communication. By maintaining a common operating system image and firmware configuration across all cluster nodes, the possibility that differences in software, firmware, or peripheral settings could influence the observed power use, energy consumption, or microarchitectural performance event measurements was minimised. The complete Raspberry Pi configuration file used throughout the study is provided in Appendix A.

Storage Platform Standardisation

The operating system for each Raspberry Pi 4 Model B was deployed on identical 32GB SanDisk Ultra microSDHC A1 memory cards [201]. All microSD cards were imaged from a common master operating-system image to ensure consistency of software configuration across the experimental cluster. The use of identical storage media controlled for potential variation arising from differences in flash-memory controllers, storage performance characteristics, and power requirements. Although benchmark execution predominantly exercised processor, memory, and JVM runtime behaviour after application launch, storage devices can influence operating-system activity, application loading, class loading, and virtual-memory operations. Consequently, the storage subsystem was standardised across all experimental nodes to minimise a potential source of experimental variability and improve the reproducibility of the study.

DC Energy Analyser

JRE power use and energy consumption under load from each of the 24 Renaissance benchmark workloads were captured using a Joulescope DC Energy Analyser [2]. The Joulescope has a voltage range specification from $1V$ to $15V$ and a current range specification between $-1A$ and $3A$. Current resolution is in the order of $1.5nA$, equivalent to 32-bits of dynamic range. The Joulescope JS110 performs simultaneous voltage and current measurements using an internal analogue-to-digital acquisition system operating at up to 2 million samples per second (2 MSPS). Although the device samples at this rate, its specified measurement bandwidth is 250 kHz. The distinction is important: the sampling rate determines how frequently measurements are acquired, whereas the measurement bandwidth represents the highest frequency signal component that can be accurately captured. The higher sampling rate provides oversampling relative to the measurement bandwidth, improving measurement fidelity and reducing the effects of aliasing during power use and energy consumption acquisition.



Figure 4.5: Jetperch LLC JS110 Joulescope DC Analyzer. Source: [2]



Figure 4.6: Jetperch LLC JS110 Joulescope depicting the rear-panel binding posts. Source: [2]

Seamless dynamic current measurements are achieved through rapid $1\mu s$ switching between shunt resistors. All sensor ports are electrically isolated from USB. The Joulescope has an adaptable front panel for the integration of device-under-test, with a number of possible variants, for example, a USB 2.0 high-speed communication, binding posts which support banana plugs and direct wire connection, or for lower noise measurements a BNC front panel that supports coax cable connections [2].

Network Infrastructure

The experimental cluster nodes were interconnected using a TP-Link TL-SG116 16-port Gigabit Ethernet switch [3], Figure 4.7. The network infrastructure supported workload deployment, remote administration, node monitoring, and experimental data collection throughout the study. Although maintaining network connectivity introduces a small computational overhead associated with operating-system network services and protocol processing, this overhead remained constant across all experimental treatments and was therefore not expected to differentially influence comparisons between JRE configurations, execution modes, workload categories, or garbage collector variants.



Figure 4.7: TP-Link TL-SG116 16-Port Gigabit Ethernet Switch Used to Interconnect the Experimental Cluster Nodes. Source: [3]

4.5.2 Software

The software infrastructure forms the execution layer of the experimental platform and supports the construction, deployment, execution, and measurement of the JRE/JVM runtime environments evaluated throughout this study. The software platform comprises the operating system environment upon which the experimental workloads and measurement tools execute, together with a standardised storage platform used to host the operating system and runtime environments. The following subsections describe each software component and its role within the experimental workflow.

Operating System Environment

In this subsection, we describe the operating system environment deployed across the Raspberry Pi 4 cluster. The operating system provides the foundational software layer upon which the OpenJDK runtime environments, benchmark workloads, measurement tools, and performance monitoring infrastructure operate. Consequently, the selection and standardisation of the operating system formed an important component of the experimental platform.

To minimise variation arising from differences in system software, all Raspberry Pi 4 Model B development boards were configured with an identical installation of the Raspbian Buster Lite operating system, based on the Debian GNU/Linux distribution and utilising Linux Kernel version 4.19. The operating system was deployed using cloned Micro-SD card images to ensure consistency across all nodes

within the experimental cluster.

The use of a common operating system image ensured that all benchmark executions were conducted under equivalent software conditions, thereby reducing the likelihood that observed differences in power use, energy consumption, or microarchitectural behaviour could be attributed to variation in the underlying operating system environment. Such standardisation is particularly important in performance evaluation studies, where maintaining a consistent execution environment assists in ensuring that observed differences can be attributed to the factors under investigation rather than uncontrolled sources of variation [202].

Raspbian Buster Lite was selected because it provides a lightweight headless operating environment without a graphical desktop interface. The absence of a graphical user environment reduces the number of background services, scheduled tasks, and system processes competing for processor, memory, and I/O resources. Consequently, the operating system minimises non-essential workload activity and reduces the potential for interference with benchmark execution, power use and energy consumption, and microarchitectural performance event observations.

OpenJDK Runtime Environment Platform

The experimental platform utilised OpenJDK Java Runtime Environment (JRE) releases 9.0.4, 10.0.2, 11.0.12, 12.0.2, 13.0.5, and 14.0.0 [203]. Each JRE was constructed from source using alternative GNU GCC compiler versions and configured to support both interpretive and tiered execution modes. The resulting runtime environments formed the primary experimental treatments evaluated throughout this study. Details regarding the JRE construction process, compiler configurations, execution modes, and garbage collector variants are presented in Section 4.6.

Joulescope Driver and Data Acquisition Interface

To facilitate automated communication between the measurement infrastructure and the experimental platform, the master Raspberry Pi node was configured with the Joulescope software development kit (SDK) and associated device drivers provided by Jetperch LLC [2]. The SDK provides a programmatic interface through which measurement devices can be discovered, configured, controlled, and monitored from external software applications.

Within the experimental platform, the Joulescope driver layer was responsible for establishing communication between the master Raspberry Pi and the JS110 Precision DC Energy Analyser. Through this interface, the acquisition software was able to initialise the measurement device, configure sampling parameters, receive real-time voltage, current, and power use measurements, and coordinate measurement collection with benchmark execution events.

The experimental platform utilised the Joulescope device driver and software development kit (SDK) supplied by Jetperch LLC [2]. The SDK provided the software interface through which the master Raspberry Pi communicated with the JS110 Precision DC Energy Analyser. Measurement acquisition was performed using a Python-based application that interfaced with the SDK, enabling the collection of voltage, current, and power use measurements and the synchronisation of data acquisition with benchmark execution events, available at [4].

The Joulescope hardware measures voltage and current directly. Instantaneous power use was derived from the corresponding sampled voltage and current observations according to

$$P(t) = V(t)I(t), \tag{4.1}$$

where $P(t)$ represents instantaneous power in watts (W), $V(t)$ represents the measured voltage in volts (V), and $I(t)$ represents the measured current in amperes (A).

Mean power use over the benchmark execution interval was calculated from the

collection of instantaneous power observations acquired between the GPIO-defined beginning and end of benchmark execution:

$$\bar{P} = \frac{1}{N} \sum_{i=1}^{N} P_i, \tag{4.2}$$

where N denotes the number of power observations acquired during the benchmark execution interval.

Cumulative energy consumption was calculated by integrating instantaneous power over the same execution interval:

$$E = \int_{t_0}^{t_1} P(t)\, dt, \tag{4.3}$$

where t_0 and t_1 denote the GPIO-defined beginning and end of benchmark execution, respectively. Within the acquisition software, this quantity was obtained numerically from the discrete power observations and their associated sampling interval. The resulting energy quantity was converted from joules to watt-hours (Wh) for the analyses reported throughout this thesis.

The use of a software-controlled measurement interface enabled fully automated and repeatable data collection across the large number of benchmark executions performed during this study. By integrating the Joulescope driver directly into the experimental workflow, measurement acquisition could be initiated, monitored, and terminated without manual intervention, thereby reducing measurement overhead and improving experimental consistency.

Experimental Software Artefacts

To support the reproducibility of this research, the software artefacts developed for data acquisition, benchmark execution have been made publicly available through a version-controlled GitHub repository. The repository includes the Python-based Joulescope acquisition framework, and benchmark execution scripts required to reproduce the measurement process described throughout this thesis.

The Joulescope acquisition framework was responsible for interfacing with the JS110 precision DC energy analyser, capturing voltage, current, and instantaneous power measurements. A data acquisition script was deployed on the master Raspberry Pi which kept a running summation of instantaneous power measurements, which were used to calculate average power use. In addition, the acquisition script also calculating cumulative energy consumption from the acquired power samples. The implementation employed throughout this study is available at:

`https://github.com/jonathan-lambert/PhD-Data-JoulescopeAcquisitionScript`

In addition, all R language statistical scripts used for the analysis of the data, and the generation of all table data, figures, inferential statistical results, regression models, and regression model diagnostics is available at:

`https://github.com/jonathan-lambert/PhD-RScripts`

The specific version of the software used for the experiments reported in this thesis has been archived and tagged within the repository to ensure long-term reproducibility.

4.5.3 Measurement Apparatus

Power use and energy consumption measurements were obtained using a Joulescope precision DC energy analyser [2] positioned in-line between the power supply and the Raspberry Pi 4 Model B under test. The Joulescope supplied power to the experimental node while acquiring voltage and current measurements throughout benchmark execution. The Joulescope data stream provided sampled current, voltage, and instantaneous power values to a Python acquisition script executing on the master Raspberry Pi. Cumulative energy consumption was then calculated within the acquisition script by numerically integrating the sampled power signal over time, using the sampling period associated with the Joulescope output stream.

Consequently, the recorded dataset contained voltage, current, instantaneous power use, and derived energy consumption values for each benchmark execution. To support transparency and reproducibility, the Python acquisition script used throughout the experimental campaign has been made publicly available through the accompanying GitHub repository:

`https://github.com/jonathan-lambert/PhD-Data-JoulescopeAcquisitionScript`

The repository contains the implementation used to interface with the Joulescope, acquire the sampled electrical measurements, and derive the power use and cumulative energy consumption values reported throughout this thesis.

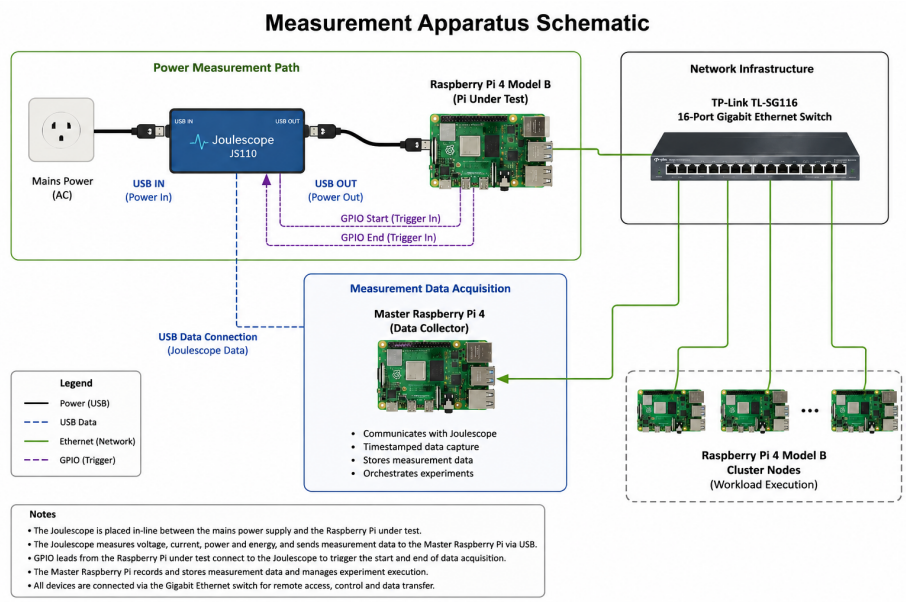


Figure 4.8: Schematic of the experimental measurement apparatus showing the Joulescope positioned in-line between the mains power supply and the Raspberry Pi 4 Model B under test. The master Raspberry Pi records measurement data from the Joulescope, while the TP-Link TL-SG116 switch provides network connectivity between the experimental cluster nodes.

The Raspberry Pi under test remained connected to the experimental network during benchmark execution; however, the primary measurement-related communication occurred between the Joulescope and the master Raspberry Pi responsible for recording and storing measurement data. Consequently, network activity

on the experimental node was limited to normal background operating-system services and remained consistent across all experimental treatments. As such, network-related activity was not considered a source of systematic variation in the comparative analysis of JRE/JVM power use and energy consumption.

The measurement boundary consisted solely of the Raspberry Pi under test and excluded the power use and energy consumption of the network switch, master Raspberry Pi, and other supporting infrastructure.

Measurement Synchronisation and Data Acquisition

To support automated and synchronised measurement collection, the JS110 Precision DC Energy Analyzer was integrated into the experimental platform through a custom software interface deployed on the master Raspberry Pi node. Benchmark execution on the Raspberry Pi under test was instrumented to generate trigger events through its GPIO interface. Specifically, GPIO state transitions were used to signal the commencement and completion of benchmark execution, with these trigger signals connected directly to the GPIO inputs of the JS110. This arrangement enabled the measurement apparatus to precisely identify the interval associated with benchmark execution while excluding periods of system idle activity.

In parallel, the master Raspberry Pi communicated with the JS110 through a network-based interface. Custom acquisition software coordinated measurement collection, retrieved the voltage, current, and power use observations recorded by the analyser, and stored the resulting data for subsequent processing and analysis. The overall architecture provided a repeatable and automated mechanism for synchronising workload execution with power use and energy consumption measurement collection.

A simplified representation of the triggering procedure is presented in Program Segment 4.1. Prior to benchmark execution, GPIO pin 17 on the Raspberry Pi under test was configured as an output and driven to a logic-high state, signalling

the start of the measurement interval. Following completion of the benchmark workload, the GPIO pin was returned to a logic-low state, signalling the end of the measurement interval. These hardware-level trigger events were captured by the GPIO interface of the JS110 and used by the acquisition framework to accurately delimit the benchmark execution window from surrounding system activity.

Program Segment 4.1: Simplified GPIO-based measurement triggering procedure.

```
# Configure GPIO 17 as an output
raspi-gpio set 17 op

# Signal start of benchmark execution
raspi-gpio set 17 dh

java [...] benchmark

# Signal completion of benchmark execution
raspi-gpio set 17 dl
```

A complete implementation of the measurement acquisition framework, including the GPIO-based synchronisation mechanism, Joulescope JS110 interface, and data collection procedures, is available through a publicly accessible source code repository [4].

4.6 Construction of JRE JVM Instances

The objective of this section is to describe the construction of the OpenJDK Java Runtime Environment (JRE) and Java Virtual Machine (JVM) instances examined throughout this study. Because the research seeks to evaluate the effects of OpenJDK source version, GNU GCC build toolchain compiler version, JVM execution mode, and garbage collector variant on power use and energy consumption, it was necessary to construct the runtime environments under controlled and reproducible conditions.

Rather than relying upon precompiled binary distributions, all OpenJDK

JRE JVM instances were built from source on the experimental Raspberry Pi 4 Model B platform. This approach provided explicit control over both the OpenJDK source version and the GNU GCC compiler toolchain used during the build process, thereby enabling the systematic investigation of build-time and runtime configuration effects. The construction process comprised two stages. First, multiple GNU GCC compiler toolchains were constructed and configured for the target platform. Second, these compiler toolchains were employed to construct multiple OpenJDK JRE JVM instances representing the runtime environments evaluated throughout the experimental programme.

The following sections describe the construction of the GNU GCC compiler toolchains, the subsequent construction of the OpenJDK JRE JVM distributions, and the runtime configuration factors that collectively defined the experimental JRE JVM instances used throughout this study.

4.6.1 Construction of GNU GCC Versions

Our experimental JRE JVM instances were built using four different GNU GCC versions, namely: GNU GCC compiler release versions 5.5.0, 6.5.0, 7.3.0, and 8.3.0. The GNU GCC toolchains were constructed natively on the Raspberry Pi 4 Model B platform using the configuration shown in Program Segment 4.2. The build was configured to support the C and C++ languages, target the ARM Cortex-A72 microarchitecture, utilise the ARMv8 NEON floating-point unit, and employ hardware floating-point operations. The `--build`, `--host`, and `--target` parameters were specified consistently to produce native ARM Linux toolchains, while the `--program-suffix` option enabled the installation of multiple GCC versions concurrently without naming conflicts. These compiler toolchains subsequently formed the basis for the construction of the experimental OpenJDK JRE JVM instances.

Program Segment 4.2 presents a representative GNU GCC configuration command used during the construction of the compiler toolchains employed in

this study. The placeholders `gcc-version`, `install-dir`, and `version` denote the specific GCC source version and installation paths associated with a particular compiler build.

Program Segment 4.2: Representative GNU GCC build configuration for the Raspberry Pi 4 Model B.

```
./configure \
  CC=/path/to/bootstrap/gcc \
  CXX=/path/to/bootstrap/g++ \
  CXXFLAGS="-O2 -Wall" \
  --enable-languages=c,c++ \
  --with-cpu=cortex-a72 \
  --with-fpu=neon-fp-armv8 \
  --with-float=hard \
  --build=arm-linux-gnueabihf \
  --host=arm-linux-gnueabihf \
  --target=arm-linux-gnueabihf \
  --prefix=/path/to/install-dir \
  --program-suffix=-version
```

Table 4.3 summarises the principal GNU GCC configuration flags employed during the construction of the compiler toolchains used within this study.

The resulting compiler toolchains provided native ARM Linux compilation environments targeting the Raspberry Pi 4 Model B platform. These toolchains subsequently formed the basis for the construction of the experimental OpenJDK JRE JVM instances.

Table 4.3: Key GNU GCC configuration flags used during toolchain construction and their associated purposes.

Configuration Flag	**Purpose**
`CC`	Specifies the path to the GNU bootstrap C compiler used during the GNU GCC build process.
`CXX`	Specifies the path to the GNU bootstrap C++ compiler used during the GNU GCC build process.
`CXXFLAGS="-O2 -Wall"`	Enables compiler optimisation level `O2` and activates a broad collection of compiler warning messages.
`--enable-languages=c,c++`	Restricts the GCC build to support the C and C++ programming languages.
`--with-cpu=cortex-a72`	Generates code optimised for the ARM Cortex-A72 microarchitecture used within the Raspberry Pi 4 Model B platform.
`--with-fpu=neon-fp-armv8`	Targets the ARMv8 NEON SIMD and floating-point processing unit.
`--with-float=hard`	Configures the compiler to utilise hardware floating-point instructions rather than software emulation.
`--build=arm-linux-gnueabihf`	Specifies the platform on which GCC is built.
`--host=arm-linux-gnueabihf`	Specifies the platform on which the resulting compiler will execute.
`--target=arm-linux-gnueabihf`	Specifies the target platform for which code is generated.
`--prefix`	Defines the installation directory for the completed GCC toolchain.
`--program-suffix`	Appends a version-specific suffix to GCC executables, allowing multiple GCC versions to coexist without naming conflicts.

4.6.2 Construction of JRE JVM Versions

We analysed six major OpenJDK releases—9.0.4, 10.0.2, 11.0.12, 12.0.2, 13.0.5, and 14.0.0—and evaluated the power use and energy consumption characteristics of their associated Java Runtime Environments (JREs). All OpenJDK JRE instances were built natively from source on the Raspberry Pi 4 Model B platform. For each OpenJDK release, four separate builds were produced using GNU GCC

compiler versions 5.5.0, 6.5.0, 7.3.0, and 8.3.0, enabling the effects of both JRE version and build toolchain compiler version to be examined.

Because OpenJDK is a self-hosting system, the construction of each JDK release required an existing bootstrap JDK. OpenJDK 9 was constructed using a pre-built OpenJDK 8 binary distribution, while each subsequent OpenJDK release was constructed using the immediately preceding version built within this study. For example, the construction of OpenJDK 10 relied upon the OpenJDK 9 build produced during the preceding stage of the build process.

We downloaded all source bundles from the OpenJDK project's repository, which the reader can find at `https://openjdk.org` (accessed May 2022). We downloaded all GNU Compiler Collection releases from `https://gcc.gnu.org` (accessed May 2022).

Program Segment 4.3: Representative OpenJDK build configuration for the Raspberry Pi 4 Model B.

```
./configure
  --enable-headless-only
  --with-boot-jdk=/path/to/bootstrap/jdk
  CC=/path/to/gcc
  CXX=/path/to/g++
  --includedir=/path/to/gcc-version/
                install-dir/include
  --libdir=/path/to/gcc-version/
            install-dir/lib
  --disable-warnings-as-errors
```

Program Segment 4.3 presents a representative OpenJDK build configuration used during the construction of the experimental JRE JVM instances. The `--enable-headless-only` option was used to restrict the build to headless components, thereby excluding graphical user interface libraries that were not required for benchmark execution. The `--with-boot-jdk` option specifies the location of the bootstrap JDK required by the OpenJDK build process, as OpenJDK is a self-hosting system that requires an existing JDK to compile subsequent versions.

Table 4.4 summarises the principal OpenJDK configuration flags employed during the construction of the experimental JRE JVM instances.

The configuration options detailed in Table 4.4 ensured that all OpenJDK JRE JVM instances were constructed in a consistent and reproducible manner while maintaining explicit control over the GNU GCC compiler version employed during the build process. This level of control was necessary to isolate the effects associated with variations in the build toolchain compiler version from those attributable to the OpenJDK source version itself.

Table 4.4: Key OpenJDK configuration flags used during JRE JVM construction and their associated purposes.

`Configuration Flag`	**Purpose**
`--enable-headless-only`	Restricts the OpenJDK build to headless components by excluding graphical user interface libraries and desktop-related modules that were not required for benchmark execution.
`--with-boot-jdk`	Specifies the location of the bootstrap JDK required by the OpenJDK build process. OpenJDK is a self-hosting system and therefore requires an existing JDK to compile subsequent releases.
`CC`	Specifies the GNU GCC C compiler used throughout the OpenJDK build process.
`CXX`	Specifies the GNU GCC C++ compiler used throughout the OpenJDK build process.
`--includedir`	Provides the location of the header files associated with the selected GNU GCC installation.
`--libdir`	Provides the location of the runtime libraries associated with the selected GNU GCC installation.
`--disable-warnings-as-errors`	Prevents compilation warnings from being treated as fatal errors, improving build portability across differing OpenJDK source versions and GNU GCC compiler versions.

The `CC` and `CXX` variables were used to explicitly specify the GNU GCC C and C++ compilers under investigation, ensuring that the intended compiler version was employed throughout the build process rather than the system-default compiler. The `--includedir` and `--libdir` options were used to provide the

locations of the header files and runtime libraries associated with the selected GNU GCC installation, thereby ensuring that all compiler dependencies were resolved consistently. Finally, the `--disable-warnings-as-errors` option prevented compilation warnings from being treated as fatal errors, improving build portability across differing OpenJDK source versions and GNU GCC compiler versions.

Collectively, these configuration options ensured that OpenJDK JRE instances could be constructed in a consistent and reproducible manner while maintaining explicit control over the GNU GCC compiler version used during the build process.

The generation of each JRE binary using the four GNU GCC versions could have several effects on the resulting JRE binaries. Firstly, variations in the application of compiler optimisations, the generation of machine code, and the linking order of binaries could lead to variations in the behaviour of the built runtime environments. Secondly, the collection of compiled native methods that compose the native method application programming interfaces could exhibit variations based on the specific compiler version used for their compilation. Thirdly, when building all JRE libraries that were compiled with the JDK generated `javac` compiler, differences should be apparent based on the properties of the compiler binaries.

Specifically, regarding the JRE JVM interpreter, two implementation choices are available at build time: the production Template interpreter and a C++ interpreter bundled primarily for ease of implementation and testing of proposed research optimisations. In this work, we focus our concentration on the behaviour of the JRE with an integrated Template Interpreter. Furthermore, the Template interpreter is the default interpreter used in the first tier of tiered execution.

Collectively, the construction process yielded twenty-four distinct OpenJDK JRE builds, comprising six OpenJDK source versions and four GNU GCC build toolchain compiler versions. These JRE builds subsequently formed the basis for the runtime configurations examined throughout the remainder of the study.

An important distinction should be made between the construction of the OpenJDK JRE JVM instances and the benchmark applications used to evaluate them. Each OpenJDK source version includes its own implementation of the Java Class Library (JCL), and the compilation of these libraries formed part of the OpenJDK build process. Consequently, the resulting JRE JVM instances incorporated Java Class Library classes compiled using the `javac` compiler associated with the corresponding OpenJDK release under construction. As a result, differences between JRE JVM instances may arise not only from variations in the native runtime components generated by the GNU GCC toolchains, but also from version-specific differences within the Java Class Library implementation itself.

In contrast, the Renaissance Benchmark Suite applications were not rebuilt as part of the experimental process. The benchmark class files remained constant across all experimental executions and were those distributed by the Renaissance development team. Consequently, all benchmark workloads were executed using an identical collection of application bytecode, ensuring that any observed differences in power use, energy consumption, or microarchitectural performance event measures originated from variations within the JRE JVM runtime environment rather than differences in the benchmark application bytecode themselves.

4.6.3 Runtime Configuration Factors

For the purposes of this study, a JRE JVM instance is considered to be fully specified by four factors: (i) the OpenJDK source version (JSV) from which the JRE was constructed, (ii) the GNU GCC build toolchain compiler version (BTCV) used during the JDK build process, (iii) the JVM execution mode (JEM), either interpretive or tiered just-in-time execution, and (iv) the garbage collector variant (GCV) employed for memory management.

The experimental design was based on the factorial combination of these factors. However, because certain garbage collector implementations were deprecated and subsequently removed during the evolution of the OpenJDK platform, not all

factor combinations could be realised across all source versions. Consequently, the final experimental design consisted of 196 distinct and executable JRE JVM instances, each representing a unique combination of runtime configuration factors.

Tables 4.5 and 4.6 summarise the availability of the garbage collector variants across the OpenJDK source versions examined in this study and the resulting distribution of valid JRE JVM instancs. Together, these tables illustrate how the deprecation and exclusion of the CMS+ParNew garbage collector (CPGC) influenced the final experimental design.

Table 4.5: Garbage collector availability across the OpenJDK source versions examined in this study. The CMS+ParNew garbage collector (CPGC) was deprecated and excluded from OpenJDK versions subsequent to OpenJDK 9.

OpenJDK	**SGC**	**PGC**	**CGC**	**CPGC**	**G1GC**
JDK 9	✓	✓	✓	✓	✓
JDK 10	✓	✓	✓	–	✓
JDK 11	✓	✓	✓	–	✓
JDK 12	✓	✓	✓	–	✓
JDK 13	✓	✓	✓	–	✓
JDK 14	✓	✓	✓	–	✓

The experimental design originally comprised 200 constructed JRE/JVM instances. However, data collected for one garbage collector variant under interpretive execution for OpenJDK 9 were inadvertently omitted during construction of the final master dataset. As this configuration was represented across four GNU GCC build toolchain compiler versions, four JRE/JVM instances were consequently absent from the final analytical dataset, resulting in 196 JRE/JVM instances represented within the subsequent analyses.

The underlying measurements for these configurations were successfully captured during data collection but were not merged into the master dataset used for the subsequent statistical analyses.

As shown in Table 4.6, OpenJDK 9 contributed 36 distinct JRE JVM instances, while each subsequent OpenJDK version contributed 32 instances owing to the exclusion of the CMS+ParNew garbage collector variant. Collectively, these

configurations formed the basis of the empirical analyses presented in Chapters 5, 6, and 7.

Table 4.6: Distribution of valid JRE/JVM instances by OpenJDK source version and execution mode. The reduced number of JDK 9 interpretive instances reflects the absence of one garbage collector variant, while the reduction in configurations for OpenJDK versions 10–14 reflects the exclusion of the CMS+ParNew garbage collector (CPGC).

OpenJDK Versions	**GC Versions**	**GCC Versions**	**Exec. Mode**	**#Inst.**
JDK 9	5	4	T	20
JDK 9	4	4	I	16
JDK 10	4	4	B	32
JDK 11	4	4	B	32
JDK 12	4	4	B	32
JDK 13	4	4	B	32
JDK 14	4	4	B	32
			Total	**196**

Note: T = Tiered Execution; I = Interpretive Execution; B = Both Execution Modes; Exec. Mode = Execution Mode; #Inst. = Number of instances;

4.7 Benchmark Workloads

The benchmark workloads employed within this study form the mechanism through which the experimental JRE/JVM instances were exercised and observed. As the primary objective of this dissertation is to investigate the effects of JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures on JRE/JVM power use and energy consumption, it is essential that the selected benchmark suite exhibits a diverse range of execution characteristics representative of contemporary managed runtime environments.

The SPECjvm2008 benchmark suite [78] was considered during the benchmark selection phase. However, SPECjvm2008 was originally designed for pre-modular Java platforms and relies upon JVM functionality that is no longer fully supported in modern Java releases. In particular, the suite makes use of the `-Xbootclasspath/p` option to prepend classes to the JVM bootstrap class path. Following the introduction of the Java Platform Module System (JPMS) in Java

9, the JVM bootstrap class path mechanism was superseded by the modular runtime architecture, and the `-Xbootclasspath/p` option was deprecated and subsequently removed. Consequently, SPECjvm2008 cannot be executed unmodified across the full range of OpenJDK versions examined in this study (OpenJDK 9–14), limiting its suitability as a benchmark platform for consistent comparison across JRE source versions.

Given these compatibility limitations, a benchmark suite capable of executing consistently across the complete range of OpenJDK versions under investigation was required. The selection process therefore favoured a contemporary JVM benchmark collection that could provide both cross-version applicability and sufficient workload diversity to exercise the runtime configurations examined within this study.

This section therefore introduces the benchmark workloads employed throughout the study and describes the workload abstraction subsequently incorporated within the statistical analyses. First, the Renaissance Benchmark Suite is introduced and justified as the principal workload collection used to exercise the experimental JRE/JVM configurations. Second, the workload category factor (WC) is presented, providing a higher-level classification of workloads according to their principal application domain. Collectively, the Renaissance workload collection and the workload category factor provide the foundation for the analyses of workload effects associated with RQ1, workload-conditioned interaction effects associated with RQ2, and the empirical modelling framework developed in response to RQ3.

4.7.1 Renaissance Benchmark Suite

The Renaissance Benchmark Suite was selected because it was specifically designed to evaluate modern managed runtime environments and contemporary JVM workloads [130, 131, 5]. Unlike earlier benchmark suites, which primarily emphasise sequential execution and computational throughput, Renaissance incorporates

workloads that exercise concurrency frameworks, memory-management subsystems, functional programming paradigms, large-scale data-processing frameworks, and contemporary JVM language ecosystems. Consequently, the benchmark suite provides a diverse collection of execution characteristics that are well suited to evaluating the effects of JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures on JRE/JVM power use and energy consumption.

An important characteristic of Renaissance for the present study is that it broadens the workload base beyond applications developed exclusively in Java while retaining Java-based applications within the benchmark collection. The Renaissance Benchmark Suite incorporates workloads drawn from contemporary Java and Scala frameworks [130], thereby providing a heterogeneous collection of applications that execute within a common JRE/JVM environment.

This mixed-language composition is particularly appropriate to the scope of the present research because the principal object of investigation is the JRE/JVM as a managed runtime environment rather than the behaviour of the Java source language itself. Whether originating from Java- or Scala-based application and framework contexts, the workloads ultimately execute within the JVM environment and exercise common runtime facilities associated with class loading, interpretation, tiered just-in-time compilation, memory allocation, garbage collection, and interaction with the underlying operating system and processor architecture. The mixed-language composition of Renaissance therefore enables JRE/JVM power use and energy consumption to be examined across a more heterogeneous collection of JVM workloads while retaining a common managed-runtime execution substrate.

Accordingly, the empirical findings reported throughout this thesis should be interpreted as characterising JRE/JVM power use and energy consumption across the assessed collection of JVM workloads rather than applications written exclusively in the Java programming language. Source language was not incorporated

as an experimental factor, and the analyses therefore do not seek to attribute observed differences in power use or energy consumption specifically to Java- or Scala-originating workloads.

Detailed analyses of the Renaissance Benchmark Suite presented by Prokopec et al. [130, 131, 5] demonstrate that the suite exhibits substantial behavioural coverage of the principal component space associated with several established JVM benchmark suites, including DaCapo, SPECjvm2008, and ScalaBench. Furthermore, Renaissance workloads were shown to occupy additional regions of the principal component space not represented within those benchmark suites. Renaissance therefore provides both continuity with prior JVM benchmarking studies and exposure to a broader range of contemporary runtime behaviours. This characteristic makes the suite particularly suitable for investigating relationships between JRE/JVM configuration, workload category, microarchitectural performance event measures, power use, and energy consumption.

The 196 JRE/JVM instances represented within the final analytical dataset were evaluated using 24 workloads drawn from version 0.11.0 of the Renaissance Benchmark Suite. For the purposes of the present analytical framework, these workloads were classified into six workload categories: Apache, concurrency, database, functional, Scala, and web. These categories distinguish broad application and workload contexts represented within the benchmark collection, including Apache Spark-based analytical workloads, concurrent programming workloads, database applications, workloads exhibiting functional programming characteristics, web-oriented applications, and workloads associated with the Scala ecosystem.

Importantly, these workload categories should not be interpreted as mutually exclusive classifications of source programming language. In particular, the designation of a Scala workload category does not imply that workloads assigned to the remaining categories originate exclusively from Java source code. Renaissance incorporates applications and frameworks drawn from both Java and Scala contexts

[130], while the workload category factor employed within this study provides an analytical classification of the principal workload contexts represented by the individual benchmark applications. Table 4.7 presents an overview of the 24 benchmark workloads and their associated workload category.

Table 4.7: The 24 application workloads composing the Renaissance Benchmark Suite, together with a brief description of each workload. Source: [5].

Workload	**Description**
`akka-uct`	Unbalanced Cobwebbed Tree actor workload in Akka.
`chi-square`	Chi-square test from Spark MLlib.
`db-shootout`	Shootout test using several in-memory databases.
`dec-tree`	Random Forest algorithm from the Spark ML library.
`dotty`	Dotty compiler on a set of source code files.
`finagle-chirper`	Microblogging service using Twitter Finagle.
`finagle-http`	Finagle HTTP requests to a Finagle HTTP server and awaits response.
`fj-kmeans`	K-means algorithm using the fork/join framework.
`future-genetic`	Genetic algorithm using the Jenetics library and futures.
`gauss-mix`	Gaussian mixture model using expectation-maximization.
`log-regression`	Logistic Regression algorithm from the Spark ML library.
`mnemonics`	Solves the phone mnemonics problem using JDK streams.
`movie-lens`	Recommends movies using the ALS algorithm.
`naive-bayes`	Multinomial Naive Bayes algorithm from the Spark ML library.
`neo4j-analytics`	Neo4J graph queries against a movie database.
`page-rank`	PageRank iterations using RDDs.
`par-mnemonics`	Solves the phone mnemonics problem using parallel JDK streams.
`philosophers`	Dining philosophers workload using ScalaSTM.
`reactors`	Reactor workloads inspired by Savina microbenchmarks.
`rx-scrabble`	Solves the Scrabble puzzle using Rx streams.
`scala-doku`	Sudoku puzzles using Scala collections.
`scala-kmeans`	K-Means algorithm using Scala collections.
`scala-stm-bench7`	STMbench7 benchmark using ScalaSTM.
`scrabble`	Solves the Scrabble puzzle using JDK Streams.

4.7.2 Workload Category Factor

The workload categorisation provided by Renaissance is of particular importance within the present study because it forms the basis of the workload category factor (WC). This factor is examined as an independent predictor within the main-effects analyses associated with RQ1 (Chapter 5), as a moderating factor within the

interaction-effects analyses associated with RQ2 (Chapter 6), and as a predictor within the development of the empirical models associated with RQ3 (Chapter 7). Consequently, the Renaissance workload taxonomy provides the foundation for evaluating both the direct effects of workload category on JRE JVM power use and energy consumption, and the extent to which workload context conditions the effects associated with alternative JRE JVM runtime configurations.

The Renaissance Benchmark Suite groups its constituent workloads according to the dominant application characteristics they are intended to represent. These categories encompass applications developed for the Apache Spark framework, concurrent applications, database applications, functional programming applications, Scala applications, and web applications. Collectively, these categories form the workload category factor (WC) used throughout the present study.

The resulting workload category provide a mechanism for examining whether JRE JVM power use and energy consumption characteristics vary according to the nature of the computational workload being executed. As shown in Table 4.8, the 24 Renaissance benchmark workloads are distributed across six workload categories, providing a diverse collection of application behaviours encompassing differing computational, memory management, concurrency, and execution characteristics.

Table 4.8: Workload Category (WC) factor showing the grouping of Renaissance benchmark workloads into six workload categories.

Category	**Workloads**	**n**
Apache Spark	`chi-square`, `dec-tree`, `gauss-mix`, `log-regression`, `movie-lens`, `naive-bayes`, `page-rank`	7
Concurrency	`akka-uct`, `fj-kmeans`, `reactors`	3
Database	`db-shootout`, `neo4j-analytics`	2
Functional	`future-genetic`, `mnemonics`, `par-mnemonics`, `rx-scrabble`, `scrabble`	5
Scala	`dotty`, `philosophers`, `scala-doku`, `scala-kmeans`, `scala-stm-bench7`	5
Web	`finagle-chirper`, `finagle-http`	2

Within the subsequent analyses, workload category is treated as a categorical

predictor variable. Consequently, the workload taxonomy provides the basis for evaluating both the direct effects of workload category on JRE JVM power use and energy consumption, and the extent to which workload context influences the effects associated with alternative JRE JVM runtime configurations.

4.8 Data Collection Procedure

Having established the experimental platform, the construction of the JRE JVM instances, and the benchmark workloads used throughout the study, this section details the procedures employed to collect the power use, energy consumption, and microarchitectural performance event data analysed in subsequent chapters. The objective of the data collection process was to ensure that all benchmark executions were performed under consistent and reproducible conditions while providing accurate synchronisation between benchmark execution and external measurement instrumentation.

Data collection consisted of the systematic execution of benchmark workloads across all valid JRE JVM configurations. Microarchitectural performance event measures, power use and energy consumption measurements were collected through separate benchmark executions. Microarchitectural data were obtained using the Linux `perf` framework, while power use and energy consumption measurements were acquired during independent benchmark executions without performance monitoring instrumentation enabled. This approach was adopted to minimise the potential influence of performance counter collection overhead on the observed power use and energy consumption measurements. Consequently, the power use and energy consumption observations reported throughout this thesis reflect the behaviour of the JRE JVM configurations under evaluation rather than the combined behaviour of the runtime environment and the measurement instrumentation.

Each execution generated observations relating to runtime power use, energy consumption, and a range of microarchitectural performance event measures.

Power use and energy consumption measurements were acquired using a Joulescope DC Energy Analyser, while microarchitectural performance measures were collected using the Linux `perf` performance monitoring framework. To minimise the influence of residual runtime state and to ensure comparability between execution modes, all benchmark workloads were executed from newly initialised JVM instances under cold-start conditions.

The following subsections describe the benchmark execution procedure, the acquisition of power use and energy consumption measurements, the collection of microarchitectural performance measures, and the mechanisms used to synchronise benchmark execution with external measurement instrumentation. Together, these procedures provided the foundation for the construction of the experimental dataset used throughout the statistical analyses and empirical modelling presented in Chapters 5, 6, and 7.

4.8.1 Benchmark Execution Procedure

Each JRE JVM instance was evaluated under cold-start conditions. No JVM warmup iterations were undertaken. Our motivation in that regard was partly influenced by the dynamic characteristics of the JVM executions run in interpreter mode and that "the interpreter does not need a warmup" [27]. In order to compare and contrast the behaviour between interpretive and tiered execution we also ran all tiered execution JRE JVM instances without warmup phases. Specifically, all benchmark workloads were executed on newly initialised JVMs, and the construction of all runtime JRE JVM sampling distributions were based on fresh executions of each benchmark workload.

IInterpretive execution was specified for each JVM using the `-Xint` runtime flag. Consistent with the SPECjvm2008 benchmark specification, no additional JVM tuning was applied [204], with the exception of the `-Xms` and `-Xmx` runtime options, which were used to set the initial and maximum heap sizes, respectively, to 2 GB across all JRE/JVM instances.

An evaluation of all JDK Enhancement Proposals (JEP) showed that OpenJDK had implemented no fundamental changes regarding the JVM Template Interpreter, and the interpreter can be considered constant across OpenJDK9 through to 14.

As a managed memory runtime, JRE JVMs are deployed with a specified garbage collector, used for heap memory management. We considered five garbage collector variants, namely the serial collector (SGC), the concurrent-mark-and-sweep collector (CGC), the parallel collector (PGC), the garbage first collector (G1GC), as well as dual memory management collector combining a parallel collector for the reclamation of the new object heap alongside the base concurrent-mark-and-sweep collector. The concurrent-mark-and-sweep paired with the parallel new collector was declared only in relation to OpenJDK version 9 deployments, as the parallel new object collector was deprecated and removed from subsequent OpenJDK JRE versions.

The memory management runtime configuration associated with each JRE JVM instance was specified at start-up through the JVM flags detailed in Table 4.9. The garbage collector variant was explicitly declared during JVM initialisation, ensuring that the memory management strategy remained fixed throughout benchmark execution. In addition, the initial heap allocation was standardised across all experimental configurations through the use of the −`Xms2G` and −`Xmx2G`flag, resulting in a consistent initial heap size of 2GB for all JRE JVM instances.

Table 4.9: JRE JVM runtime configuration flags used within the experimental framework.

Configuration	**Runtime Flag**
SGC	`-XX:+UseSerialGC`
PGC	`-XX:+UseParallelGC`
CGC	`-XX:+UseConcMarkSweepGC`
G1GC	`-XX:+UseG1GC`
CPGC	`-XX:+UseConcMarkSweepGC + -XX:+UseParNewGC`
Initial Heap Allocation	`-Xms2G`
Maximum Heap Allocation	`-Xmx2G`

Program Segment 4.4: Representative JRE JVM benchmark execution command, running in interpreter mode, with a minimum and maximum heap size of 2GB, executing the `db-shootout` renaisance workload.

```
/path/to/jre/bin/java \
    -Xint \
    -XX:+UseSerialGC \
    -Xms2G \
    -Xmx2G \
    -classpath /path/to/Renaissance/dir/ \
    -jar renaissance-gpl-0.11.0.jar \
    -r 1 \
    db-shootout \
    1>> "./stdOut.txt" \
    2>> "./stdErr.txt"
```

Program Segment 4.4 presents a representative command used to execute a benchmark workload. The command illustrates the specification of the JRE JVM binary, execution mode, garbage collector variant, heap allocation, benchmark workload, and output redirection. In particular, it encodes the invocation of a JVM, operating in interpreter mode, using the serial garbage collector for heap object reclamation, with a fixed heap space of $2GB$, and a single repetition of the `db-shootout`. Throughout the study, equivalent commands were generated automatically by the experimental framework to execute each valid combination of JRE source version, build toolchain compiler version, execution mode, garbage collector variant, and benchmark workload.

Table 4.10 summarises the principal runtime flags employed during benchmark execution and their associated purposes.

Table 4.10: Key JRE JVM runtime flags used during benchmark execution and their associated purposes.

Runtime Flag	**Purpose**
`-Xint`	Executes all Java bytecode using the JVM interpreter, disabling just-in-time compilation. This flag was used to evaluate interpretive execution mode.
`-Xmixed`	Enables tiered execution by combining interpretation and just-in-time compilation. This represents the default execution mode of modern HotSpot JVM implementations.
`-XX:+UseSerialGC`	Selects the Serial Garbage Collector (SGC), which performs garbage collection using a single execution thread.
`-XX:+UseParallelGC`	Selects the Parallel Garbage Collector (PGC), which employs multiple threads during garbage collection activities.
`-XX:+UseConcMarkSweepGC`	Selects the Concurrent Mark-Sweep Garbage Collector (CGC), designed to minimise application pause times through concurrent garbage collection phases.
`-XX:+UseParNewGC`	Enables the ParNew young-generation collector and was used in conjunction with `-XX:+UseConcMarkSweepGC` to form the CMS+ParNew configuration (CPGC).
`-XX:+UseG1GC`	Selects the Garbage First Garbage Collector (G1GC), a region-based collector designed to balance throughput and pause-time performance.
`-Xms2G`	Specifies an initial Java heap allocation of 2 GB, ensuring a consistent starting memory allocation across all experimental configurations.
`-classpath`	Specifies the location of the Renaissance Benchmark Suite class files required for benchmark execution.
`-jar`	Executes the Renaissance benchmark suite packaged as an executable Java archive (JAR) file.
`-r 1`	Configures the Renaissance framework to execute a single benchmark repetition within a given JVM invocation.

The runtime flags detailed in Table 4.10 were combined programmatically to generate the JRE JVM configurations examined in this study. These flags controlled the execution mode, heap allocation, memory management strategy,

and benchmark invocation procedures associated with each JRE JVM instance.

4.8.2 Microarchitectural Performance Data Collection

In addition to power use and energy consumption measurements, a collection of microarchitectural performance measures was acquired for each JRE JVM configuration. The purpose of collecting these measures was to provide insight into the underlying processor behaviours associated with benchmark execution and to facilitate the investigation of relationships between microarchitectural activity and observed power use and energy consumption. These measures subsequently formed an important component of the empirical modelling framework presented in Chapter 7.

Microarchitectural performance measures were collected using the Linux `perf` performance monitoring framework, which provides access to hardware performance monitoring units (PMUs) exposed by the ARM Cortex-A72 microarchitecture. The framework was used to capture measures relating to cache behaviour, branch prediction behaviour, instruction execution, memory activity, and virtual memory management. These observations were recorded for each benchmark workload and JRE JVM configuration examined in the study.

The collection of microarchitectural performance measures was undertaken independently of the power use and energy consumption measurement process. Specifically, benchmark executions used for performance counter collection were performed separately from those used for power use and energy consumption acquisition. A methodological benefit of this approach is that it limits the extent to which performance monitoring instrumentation could influence the observed power use and energy consumption characteristics. Consequently, the power use and energy consumption measurements reported throughout this thesis reflect the behaviour of the JRE JVM configurations under evaluation rather than the combined effects of benchmark execution and performance counter instrumentation.

The microarchitectural performance measures should be interpreted as observed workload characteristics rather than experimentally manipulated factors. Consequently, they were incorporated into the empirical modelling framework, presented in Chapter 7, as explanatory variables describing emergent runtime behaviour.

Linux perf Configuration

Within the present study, benchmark workloads were executed using the `perf stat -d` command, which reports an extended collection of hardware and operating system performance statistics derived from the underlying performance monitoring units and kernel instrumentation facilities. These statistics included measures relating to instruction execution, processor cycles, branch behaviour, cache activity, virtual memory activity, and operating system interactions.

The resulting observations were subsequently incorporated into the experimental dataset and used within the empirical modelling framework to investigate the extent to which variations in microarchitectural behaviour were associated with differences in power use and energy consumption.

In addition, the Linux `perf` framework supports the collection of performance counter data originating from user-space execution, kernel-space execution, or a combination of both. User-space measurements capture events generated directly by application execution, whereas kernel-space measurements capture events associated with operating system activities involving scheduling, memory management, interrupt handling, and system call processing.

In the present study, microarchitectural performance measures were collected using the default `perf stat -d` configuration, which records events arising from both user-space and kernel-space execution. This approach was adopted because JRE/JVM execution extends beyond the direct execution of Java bytecode and encompasses runtime activities involving garbage collection and memory allocation, together with interactions with the operating system involving thread scheduling

and virtual memory management. Restricting measurements to user-space execution would therefore omit a component of the processor activity associated with benchmark execution.

Consequently, performance counter measurements were collected across both user-space and kernel-space execution, providing a more complete representation of the runtime behaviour of the JRE JVM configurations examined in this study.

Program Segment 4.5 presents a representative command used to collect microarchitectural performance measures during benchmark execution. The Linux `perf stat -d` framework was employed to obtain an extended collection of hardware and operating system performance statistics, including measures relating to instruction execution, branch prediction behaviour, cache utilisation, and virtual memory activity. The command illustrates the invocation of a benchmark workload under a specified JRE JVM configuration while simultaneously collecting the associated performance counter data.

Program Segment 4.5: Representative microarchitectural performance data collection command.

```
perf stat -d \
    "$jvm" -Xmixed $gc -Xms2G \
    -classpath /path/to/Renaissance/dir \
    -jar renaissance-gpl-0.11.0.jar \
    -r 1 "$bench"
```

The Program Segment has been simplified for presentation purposes and is intended to illustrate the general microarchitectural data collection procedure rather than the complete implementation used throughout the study. Equivalent commands were generated programmatically across all valid combinations of JRE JVM binaries, execution modes, garbage collector variants, and benchmark workloads. As discussed previously, the collection of microarchitectural performance measures was undertaken independently of the power use and energy consumption measurement process to minimise the potential influence of performance

monitoring instrumentation on the observed energy characteristics.

Table 4.11 summarises the performance measures collected and their associated interpretations. The selected measures characterise several aspects of benchmark execution, including processor utilisation, instruction execution, branch prediction behaviour, cache utilisation, virtual memory activity, and operating system interactions. These observations were subsequently incorporated into the experimental dataset and used to investigate relationships between microarchitectural behaviour, power use, and energy consumption.

Although the Linux `perf stat -d` framework reported a broad collection of processor and operating system performance measures, only a subset was retained for incorporation within the Conceptual Framework and subsequent empirical analyses. Specifically, four derived microarchitectural performance event measures were retained: branch misprediction rate (BMR), L1 data cache load miss rate (L1DLMR), L2 data cache load miss rate (L2DLMR), and page fault rate (PFMR). These measures were derived from the underlying performance events collected during benchmark execution and were subsequently incorporated as emergent predictor variables within the statistical analyses.

This distinction is consistent with the conceptual framework presented in Figure 4.2, where microarchitectural behaviour is represented through measures of branch prediction activity, cache behaviour, and virtual memory activity. These characteristics provide observable manifestations of the interaction between workload execution and the underlying processor architecture and therefore offer a plausible explanatory basis for variations in observed power use and energy consumption.

Table 4.11: Microarchitectural and operating system performance measures collected using the Linux `perf stat -d` framework.

Performance Measure	Category	Description
`task-clock`	Processor Activity	Total processor execution time consumed by the benchmark workload.
`context-switches`	Operating System Activity	Number of operating system context switches occurring during benchmark execution.
`cpu-migrations`	Operating System Activity	Number of process migrations between processor cores during execution.
`page-faults`	Virtual Memory	Number of virtual memory page faults generated during benchmark execution.
`cycles`	Processor Activity	Total number of processor clock cycles consumed during execution.
`instructions`	Instruction Execution	Total number of instructions retired during benchmark execution.
`branches`	Branch Prediction	Total number of branch instructions executed.
`branch-misses`	Branch Prediction	Number of branch instructions incorrectly predicted by the processor.
`L1-dcache-loads`	L1 Data Cache	Total number of L1 data cache load operations.
`L1-dcache-load-misses`	L1 Data Cache	Total number of L1 data cache load operations resulting in a cache miss.
`LLC-loads`	Last-Level Cache	Total number of last-level cache load operations.
`LLC-load-misses`	Last-Level Cache	Total number of last-level cache load operations resulting in a cache miss.

4.8.3 Power Use and Energy Consumption Measurement Procedure

Program Segment 4.6 presents a representative benchmark execution procedure used during data collection. The procedure illustrates the automated iteration across JRE JVM binaries, garbage collector variants, and Renaissance benchmark workloads. Prior to the invocation of each benchmark workload, GPIO pin 17 was driven high, providing an external synchronisation signal to the Joulescope DC Energy Analyser and thereby marking the commencement of the measurement interval. Upon completion of the benchmark workload, the GPIO pin was driven low, signalling the termination of data acquisition.

The execution command specifies the JRE JVM binary under evaluation, the execution mode, garbage collector variant, heap allocation, benchmark workload, and output redirection. Through systematic iteration across all valid combinations of JRE source version, build toolchain compiler version, execution mode, garbage collector variant, and benchmark workload, the automation framework ensured a consistent and reproducible execution environment throughout the study. The Program Segment has been simplified for presentation purposes and is intended to illustrate the overall benchmark execution workflow rather than the complete implementation used during data acquisition.

Program Segment 4.6: Representative power use and energy consumption data collection command.

```
#!/bin/bash

rBenchmarkNames=("akka-uct" "chi-square" "db-shootout" ...
                 "scala-stm-bench7" "scrabble")

jvms=("/path/to/jre9/gcc5/jdk/bin/java"
      "/path/to/jre9/gcc6/jdk/bin/java"
      "/path/to/jre9/gcc7/jdk/bin/java"
      "/path/to/jre9/gcc8/jdk/bin/java")

gcs=("-XX:+UseSerialGC" "-XX:+UseParallelGC"
     "-XX:+UseConcMarkSweepGC" "-XX:+UseG1GC"
     "-XX:+UseConcMarkSweepGC -XX:+UseParNewGC")

raspi-gpio set 17 op

for jvm in "${jvms[@]}" do
  for gc in "${gcs[@]}" do
    for bench in "${rBenchmarkNames[@]}" do
      raspi-gpio set 17 dh

      "$jvm" -Xmixed $gc -Xms2G -Xmx2G\
        -classpath /path/to/Renaissance/dir \
        -jar renaissance-gpl-0.11.0.jar \
        -r 1 "$bench" \
        1>> "./stdOut.txt" \
        2>> "./stdErr.txt"

      raspi-gpio set 17 dl
    done
  done
done
```

Table 4.12 summarises the principal elements of the benchmark execution procedure and their associated purposes. Collectively, these elements controlled the traversal of the experimental design space, the invocation of JRE JVM instances, the execution of benchmark workloads, the synchronisation of power use and energy consumption measurements, and the capture of execution logs used for subsequent verification and analysis.

Table 4.12: Key elements of the benchmark execution procedure and their associated purposes.

Script Element	**Purpose**
`jvms`	Array containing the paths to the JRE JVM binaries under evaluation. Each entry corresponds to a distinct OpenJDK source version and GNU GCC build toolchain compiler version combination.
`gcs`	Array containing the garbage collector variants examined within the study.
`rBenchmarkNames`	Array containing the Renaissance benchmark workloads executed throughout the study.
`raspi-gpio set 17 op`	Configures GPIO pin 17 as an output pin, enabling its use as an external synchronisation signal for the Joulescope DC Energy Analyser.
`raspi-gpio set 17 dh`	Drives GPIO pin 17 high, signalling the commencement of the power use and energy consumption measurement interval.
`raspi-gpio set 17 dl`	Drives GPIO pin 17 low, signalling the termination of the power use and energy consumption measurement interval.
`"$jvm"`	Invokes the JRE JVM binary associated with the current experimental configuration.
`-Xmixed`	Enables tiered execution mode. Equivalent invocations using `-Xint` were employed when evaluating interpretive execution mode.
`$gc`	Expands to the garbage collector variant associated with the current experimental condition.
`-Xms2G`	Specifies a minimum heap allocation of 2 GB.
`-Xmx2G`	Specifies a maximum heap allocation of 2 GB.
`-classpath`	Specifies the location of the Renaissance Benchmark Suite classes and supporting libraries.
`-jar renaissance-gpl-0.11.0.jar`	Executes the Renaissance Benchmark Suite packaged as a Java Archive (JAR) file.
`-r 1`	Configures the benchmark framework to execute a single benchmark repetition within the current JVM invocation.
`"$bench"`	Specifies the Renaissance benchmark workload to be executed.
`1>> "./stdOut.txt"`	Appends standard output generated during benchmark execution to a log file.
`2>> "./stdErr.txt"`	Appends standard error output generated during benchmark execution to a log file.

Verification of Tiered Compilation Activity

To verify that the tiered execution configurations employed within the experimental investigation resulted in actual just-in-time compilation, separate verification executions were undertaken with HotSpot compilation logging enabled. These verification runs were conducted independently of the power use and energy consumption measurement executions because compilation logging introduces additional output and execution activity that could perturb the electrical measurements. Consequently, compilation logging was not enabled during any of the principal Joulescope measurement runs.

The resulting compilation traces confirmed progression through the HotSpot tiered compilation system. In particular, methods were initially observed at compilation level 3, corresponding to C1 compilation with profiling, before subsequently being promoted to compilation level 4, corresponding to C2 optimised compilation.

As a representative example, during execution of the Renaissance `db-shootout` workload, the benchmark method `org.lmdbjava.bench.MapDb$1::run` was compiled at level 3 before subsequently being compiled at level 4. The earlier level-3 compilation was then reported as `made not entrant`, indicating replacement by the more highly optimised compiled version. Similar level-3 to level-4 transitions were observed for database-library methods including `org.mapdb.BTreeMap::get` and `org.mapdb.StoreDirect::getBinaryLong`.

Table 4.13: Representative HotSpot tiered-compilation transitions observed during verification execution of the Renaissance `db-shootout` workload.

Method	Initial Level	Later Level	Transition
`MapDb$1::run`	3	4	C1 → C2
`BTreeMap::get`	3	4	C1 → C2
`StoreDirect::getBinaryLong`	3	4	C1 → C2

These verification traces therefore confirm that the tiered execution condition did not simply enable the HotSpot tiered-compilation mechanism; C1 and C2 compilation activity occurred during execution of the assessed workload. Importantly, the power use and energy consumption observations themselves were

obtained without compilation logging or performance-monitoring instrumentation enabled. The reported tiered measurements should consequently be interpreted as measurements of the complete cold-start tiered execution process, encompassing interpretation, runtime profiling, JIT compilation, and progressive optimisation, rather than as measurements restricted exclusively to steady-state C2-compiled execution.

4.8.4 Experimental Reliability Assessment

Prior to undertaking the main experimental study, a series of reliability assessments were conducted to evaluate the consistency and reproducibility of the experimental platform. Because the study relies upon comparative measurements collected across a large number of runtime configurations, it was necessary to establish that the experimental environment produced stable and repeatable execution behaviour under controlled conditions. Four complementary reliability assessments were therefore undertaken, examining JRE JVM build reproducibility, temporal stability across repeated benchmark executions, repeated execution stability on a single experimental node, and execution consistency across multiple Raspberry Pi cluster nodes.

Measurement Reliability: JRE JVM Build Reproducibility Assessment

To assess the reproducibility of the JRE JVM build process, four independent instances of an identical OpenJDK 9 runtime environment were constructed using an identical GNU GCC version 8 build toolchain. Following construction, the `akka-uct` benchmark was executed twenty times on each independently constructed JRE JVM instance, and benchmark execution time was recorded for each execution. The objective of this assessment was to determine whether independently constructed JRE JVM instances exhibited consistent execution behaviour, thereby establishing whether the build process itself introduced any measurable variability.

Prior to inferential analysis, distributional normality was assessed using the Shapiro–Wilk test. No statistically significant departures from normality were observed across any of the four build distributions (all $p > .05$), indicating that the benchmark execution time distributions associated with each independently constructed JRE JVM instance were approximately normally distributed.

A one-way analysis of variance was subsequently undertaken to evaluate differences in benchmark execution time across the four independently constructed JRE JVM instances. The analysis indicated no statistically significant differences between build instances ($F(3, 76) = 0.104$, $p = .957$), indicating that independently constructed JRE JVM instances exhibited highly consistent execution behaviour.

These findings provide evidence that the JRE JVM build process was highly reproducible under the experimental conditions employed within this study. The absence of statistically significant differences between independently constructed JRE JVM instances indicates that repeated construction of identical runtime environments did not introduce measurable variability in benchmark execution behaviour. Consequently, independently compiled JRE JVM instances may be considered functionally equivalent for the purposes of the experimental analyses undertaken throughout this experimental study.

Measurement Reliability: Temporal Stability Assessment

To assess the extent to which repeated benchmark execution introduced systematic time-dependent variation, a temporal stability assessment was conducted using four independently constructed instances of an identical OpenJDK 9 runtime environment compiled using GNU GCC version 8. For each build instance, the `akka-uct` benchmark was executed repeatedly across twenty consecutive runs, and benchmark execution time was recorded for each execution.

To evaluate whether execution times exhibited evidence of systematic temporal drift across successive benchmark executions, Spearman rank correlation tests were conducted between benchmark execution order and recorded execution time

for each of the four build instances.

The resulting analyses indicated no statistically significant monotonic relationship between execution order and benchmark execution time for any of the four build instances. Spearman correlation coefficients remained weak across all four builds, ranging from $\rho = -0.301$ to $\rho = 0.057$, with all statistical tests returning non-significant results (all $p > .05$).

The absence of systematic temporal drift further suggests that transient thermal accumulation, progressive resource contention, and runtime state persistence were unlikely to have materially affected the observed power use and energy consumption under the experimental conditions employed.

These findings indicate that repeated benchmark execution did not produce evidence of systematic temporal drift over successive benchmark runs. Although small fluctuations in execution time were observed between individual runs, no consistent increase or decrease in execution time was identified across repeated execution. Collectively, these results suggest that time-dependent variation associated with repeated benchmark execution did not constitute a substantive source of experimental bias within the measurement procedures adopted throughout this study.

Measurement Reliability: Single-Platform Execution Stability Assessment

To further assess the stability of benchmark execution under controlled experimental conditions, five benchmark applications (`chi-square`, `future-genetic`, `neo4j-analytics`, `scala-kmeans`, and `scrabble`) were randomly selected from the Renaissance Benchmark Suite and repeatedly executed twenty times on a single Raspberry Pi 4 development board. The objective of this analysis was to evaluate the degree of run-to-run variation in benchmark execution time when all hardware, software, and runtime conditions were held constant.

Execution stability was assessed using the coefficient of variation (CV), calcu-

lated as the ratio of the sample standard deviation to the sample mean for the observed execution times associated with each benchmark application. Across the five benchmark workloads, coefficients of variation ranged from 0.35% to 2.85%. The lowest variation was observed for `chi-square` ($CV = 0.35\%$), followed by `scrabble` ($CV = 0.82\%$), `scala-kmeans` ($CV = 1.14\%$), and `future-genetic` ($CV = 1.17\%$). The highest observed variation occurred for `neo4j-analytics`, although the coefficient of variation remained relatively low overall ($CV = 2.85\%$).

Collectively, these results indicate a high degree of execution consistency across repeated benchmark executions performed under identical experimental conditions. The low observed run-to-run variability demonstrates that benchmark execution behaviour remained highly stable on an individual Raspberry Pi 4 development board and suggests that repeated executions introduced minimal uncontrolled experimental variation during benchmark execution throughout this study.

Measurement Reliability: Cross-Platform Execution Stability Assessment

A single OpenJDK 9 runtime environment, compiled using GNU GCC version 8, was deployed across seven Raspberry Pi 4 development boards within the experimental cluster. Five benchmark applications were subsequently executed independently on each platform, and one-way analysis of variance was undertaken to evaluate the consistency of benchmark execution times across the hardware platform.

At the 5% significance level, no statistically significant differences in execution time were observed for four of the five benchmark applications ($p > .05$). The single exception was `scala-kmeans`, which exhibited a statistically significant difference in execution time across the seven Raspberry Pi 4 development boards ($p < .05$). However, the average execution time associated with `scala-kmeans` was approximately 155 s, while the largest observed difference attributable to the most divergent node was approximately 1.97 s, corresponding to only 1.28% of

the overall execution time.

Although a statistically significant difference was identified for the `scala-kmeans` benchmark, the magnitude of the observed difference was small in practical terms. Collectively, these findings indicate a high degree of execution consistency across the Raspberry Pi 4 cluster and suggest that platform-level variation between experimental nodes did not constitute a substantive source of uncontrolled experimental variability during benchmark execution throughout this study.

4.9 Dataset Construction

The experimental dataset used throughout this study was constructed from two principal data sources: power use and energy consumption measurements obtained using the Joulescope precision DC energy analyser, and microarchitectural performance event measurements obtained using the Linux `perf` subsystem.

4.9.1 Power Use and Energy Consumption Data Capture

For each unique benchmark-condition combination, consisting of a specific OpenJDK source version, GNU GCC compiler version, JVM execution mode, garbage collector variant, and benchmark workload, a single benchmark execution was performed for the collection of power use and energy consumption measurements. Each power use observation therefore corresponds to one benchmark execution under a fully specified JRE/JVM configuration.

The experimental design originally comprised 200 constructed JRE/JVM instances. However, data collected for one garbage collector variant under interpretive execution for OpenJDK 9 were inadvertently omitted during construction of the final master dataset. As this configuration was represented across four GNU GCC build toolchain compiler versions, four JRE/JVM instances were consequently absent from the final analytical dataset, resulting in 196 JRE/JVM instances represented within the subsequent analyses. The underlying measure-

ments for these four configurations had been successfully captured during data collection but were not merged into the master dataset used for statistical analysis.

As shown in Table 4.14, the 196 JRE/JVM instances were evaluated across the applicable Renaissance benchmark workloads. OpenJDK versions 9–11 were represented across all 24 workloads, whereas OpenJDK versions 12–14 were represented across 23 workloads because the *db-shootout* workload could not be successfully executed for these JRE versions. This resulted in a final analytical dataset containing 4,608 benchmark-condition observations. Each observation represents one workload executed under one fully specified JRE/JVM configuration and contains the corresponding average power use and total energy consumption measurements.

Table 4.14: Distribution of JRE/JVM instances and benchmark-condition observations included in the final analytical dataset.

JDK Version	**#GC**	**#GCC**	**Exec. Mode**	**#JRE**	**Workloads**	**#Inst.**
JDK 9	5	4	T	20	24	480
JDK 9	4	4	I	16	24	384
JDK 10	4	4	B	32	24	768
JDK 11	4	4	B	32	24	768
JDK 12	4	4	B	32	23	736
JDK 13	4	4	B	32	23	736
JDK 14	4	4	B	32	23	736
			Total	**196**		**4,608**

Note: #GC = Number of Garbage Collectors; #GCC = Number of GCC Compiler Versions; T = Tiered Execution; I = Interpretive Execution; B = Both Execution Modes; Exec. Mode = Execution Mode; #Inst. = Number of instances;

Although power use and energy consumption were measured for a single execution of each benchmark-condition combination, each reported observation represents a summary statistic derived from a large volume of high-frequency voltage, current, and power use measurements acquired throughout the execution interval. The Joulescope acquisition system sampled at up to two million samples per second, resulting in each reported power use and energy consumption observation being derived from a large number of underlying electrical measurements rather than a single instantaneous reading. Consequently, uncertainty associated

with the electrical measurement process itself is expected to be small relative to the uncertainty arising from potential execution-to-execution variation. The principal limitation of the design is therefore not the precision of the underlying electrical measurements, but the inability to directly estimate execution-to-execution variability in power use and energy consumption for individual benchmark-condition combinations.

The experimental reliability assessments presented in Section 4.8.4 demonstrated that homogeneous OpenJDK builds exhibited consistent runtime behaviour and that benchmark execution was stable across the Raspberry Pi 4 cluster nodes. These findings provide evidence that the experimental platform operated consistently throughout data collection and that uncontrolled variability associated with hardware deployment and software construction was limited. While these assessments do not quantify execution-to-execution variability in the electrical outcome measures themselves, they provide supporting evidence that the experimental environment within which the power use and energy consumption measurements were collected remained stable.

4.9.2 Microarchitectural Performance Event Counter Data Capture

Microarchitectural performance counters were collected over thirty repeated executions of each benchmark-condition combination. The repeated executions were undertaken to characterise the stability and typical behaviour of the microarchitectural performance measures, whereas the power use and energy consumption observations were derived from high-frequency electrical measurements acquired continuously throughout benchmark execution. Because the microarchitectural performance measures were intended to characterise the typical runtime behaviour associated with each benchmark-condition combination, the thirty repeated observations were reduced to their arithmetic mean values prior to dataset integration. The resulting averaged microarchitectural measures were therefore aligned with

the corresponding power use and energy consumption observations at the same configuration level.

Accordingly, the unit of analysis used throughout the inferential and modelling procedures was the benchmark-condition observation. Consequently, each row within the analytical dataset represents a unique benchmark-condition combination rather than an individual benchmark execution or an individual microarchitectural measurement. This design allowed the models to investigate how differences in JRE/JVM configuration, workload category, and typical microarchitectural behaviour were associated with differences in observed power use and energy consumption across the experimental design.

The aggregation of repeated microarchitectural measurements prior to dataset integration ensured that repeated executions of the same benchmark-condition combination were not treated as independent observations within the subsequent regression analyses.

Following aggregation, the microarchitectural dataset was joined with the power use and energy consumption dataset using the common configuration identifiers: OpenJDK source version (JSV), GNU GCC compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and benchmark workload. The benchmark workload identifier uniquely distinguished each of the 24 Renaissance benchmark applications and ensured that the aggregated microarchitectural observations were aligned with the corresponding power use and energy consumption measurements obtained from the same benchmark-condition combination.

The resulting analytical dataset therefore contained one observation for each unique benchmark-condition combination, comprising the observed power use and energy consumption measures together with the associated runtime configuration factors, benchmark workload, workload category, and aggregated microarchitectural performance measures used throughout the subsequent inferential and modelling analyses. Specifically, the final analytical dataset contained 4,608

benchmark-condition observations, reflecting the 196 JRE/JVM instances represented within the master dataset and differences in benchmark availability across OpenJDK versions.

To provide a concrete illustration of the structure of the resulting analytical dataset, Figure 4.9 presents a representative snapshot of the integrated dataset. The figure illustrates the combination of runtime configuration variables, workload descriptors, power use and energy consumption measurements, and aggregated microarchitectural performance event measures. Each row corresponds to a unique benchmark–condition combination, while the columns represent the categorical and quantitative variables subsequently used throughout the descriptive, inferential, and modelling analyses.

It is important to distinguish the sources of variability represented by the electrical and microarchitectural measures within each analytical observation. During the single benchmark execution used for electrical measurement, the Joulescope JS110 sampled continuously at a frequency of two million samples per second, from which mean power use, standard deviation, minimum power use, maximum power use, and total energy consumption were derived. The reported standard deviation, minimum, and maximum values therefore characterise variation within an individual benchmark execution trace rather than variation across repeated executions. In contrast, the microarchitectural performance event measures represent arithmetic means derived from 30 separate benchmark executions under the corresponding experimental condition. Consequently, execution-to-execution replication is represented within the construction of the microarchitectural measures but not within the power use and energy consumption outcome measures.

The dataset snapshot presented in Figure 4.9 is intended to illustrate the overall structure of the analytical dataset rather than to provide a complete description of all variables. Detailed descriptions of the categorical and quantitative variables, including their identifiers, scales of measurement, and interpretations, are presented in Tables 4.15, 4.16, and 4.17.

	jvm	mode	gcc	gc	benchmark	category	meanW	stdW	minW	maxW	energyWh	bmr	l1dlmr	l2dlmr	pfmr	catmp	catmb	catdp	catds	catml	catid
1	j09	HS	GCC5	CGC	akka-uct	concurrency	4.76160	0.643733	2.52975	8.45837	0.12537300	0.037495	0.023730	0.158460	2.7881482	mp	nmb	ndp	nds	nml	nid
2	j09	HS	GCC5	CGC	chi-square	apache	3.58701	0.623937	2.39109	7.75331	0.06002100	0.038110	0.025965	0.140295	5.2287460	nmp	nmb	dp	nds	ml	nid
3	j09	HS	GCC5	CGC	db-shootout	database	3.57932	0.785589	2.38321	8.51868	0.21680000	0.040960	0.024975	0.140310	5.0304065	nmp	nmb	ndp	ds	nml	id
4	j09	HS	GCC5	CGC	dec-tree	apache	3.73316	0.767620	2.39500	7.82453	0.07618050	0.037345	0.033785	0.158025	2.1751856	nmp	nmb	dp	nds	ml	nid
5	j09	HS	GCC5	CGC	dotty	scala	4.09620	0.492657	2.51347	7.34678	0.08004010	0.035930	0.051790	0.178835	0.8975196	nmp	nmb	ndp	ds	nml	id
6	j09	HS	GCC5	CGC	finagle-chirper	web	4.25972	0.585597	2.27069	7.52982	0.06153400	0.041380	0.047980	0.226745	0.6108781	nmp	nmb	ndp	nds	nml	id
7	j09	HS	GCC5	CGC	finagle-http	web	4.30659	0.538655	2.48430	7.70133	0.05444540	0.037455	0.055710	0.216120	0.6399286	mp	nmb	ndp	nds	nml	id
8	j09	HS	GCC5	CGC	fj-kmeans	concurrency	4.31310	0.795769	2.42423	8.99633	0.04313250	0.037760	0.026705	0.349225	0.7117630	nmp	nmb	ndp	ds	nml	nid
9	j09	HS	GCC5	CGC	future-genetic	functional	4.05625	0.504770	2.55217	7.40962	0.01472460	0.040495	0.042030	0.174920	2.3090346	nmp	nmb	ndp	nds	nml	id
10	j09	HS	GCC5	CGC	gauss-mix	apache	3.38874	0.650962	2.38948	7.67937	0.06784260	0.037905	0.027870	0.147235	2.8721037	nmp	nmb	ndp	nds	ml	nid
11	j09	HS	GCC5	CGC	log-regression	apache	3.54838	0.647094	2.38550	8.20649	0.07127000	0.035665	0.033070	0.169875	3.7148813	nmp	nmb	dp	nds	ml	nid
12	j09	HS	GCC5	CGC	mnemonics	functional	3.76371	0.417379	2.66359	6.76237	0.02738630	0.041705	0.023955	0.135140	6.6164226	nmp	mb	ndp	nds	nml	nid
13	j09	HS	GCC5	CGC	movie-lens	apache	3.90075	0.873113	2.26007	8.29519	0.13709300	0.037955	0.031220	0.166570	0.7933957	nmp	nmb	dp	nds	nml	id
14	j09	HS	GCC5	CGC	naive-bayes	apache	3.68119	0.915806	2.38436	8.53275	0.22077000	0.038475	0.016565	0.178150	5.1932845	nmp	nmb	dp	nds	ml	nid
15	j09	HS	GCC5	CGC	neo4j-analytics	database	3.93370	0.745056	2.40517	8.04845	0.20261500	0.040855	0.029090	0.156665	2.9589180	nmp	nmb	ndp	nds	nml	id

Figure 4.9: Representative snapshot of the final analytical dataset comprising runtime configuration factors, workload descriptors, power use and energy consumption observations, and microarchitectural performance measures. Each observation represents a unique JRE JVM instance and benchmark workload used in the subsequent analyses. The full dataset is available through the accompanying research data repository [4].

Consequently, the regression models were fitted using 4,608 benchmark-condition observations rather than the larger collection of repeated microarchitectural executions from which the explanatory measures were derived.

The resulting analytical dataset formed the basis of all subsequent analyses. Specifically, the dataset supported the examination of factor-level effects associated with power use and energy consumption (RQ1), workload-conditioned interaction effects (RQ2), and the development of linear regression models of power use and energy consumption (RQ3).

4.9.3 Dataset Overview

The analytical dataset developed for this study comprises measurements describing JRE JVM runtime configuration, benchmark workload category, microarchitectural performance behaviour, power use, and energy consumption. As described in Section 4.9, the dataset was formed through the integration of power use and energy consumption measurements obtained using the Joulescope precision DC energy analyser with aggregated microarchitectural performance event measurements collected using the Linux `perf` subsystem.

The final analytical dataset contains 4,608 benchmark-condition observations. Each observation represents a unique combination of OpenJDK source version, GNU GCC compiler version, JVM execution mode, garbage collector variant, and benchmark workload. Associated with each observation are measurements of power use, energy consumption, workload descriptors, and aggregated microarchitectural performance measures.

The dataset contains both categorical and quantitative variables. The categorical variables describe the runtime configuration and benchmark workload category associated with each observation, while the quantitative variables capture microarchitectural performance event measurements, power use, and energy consumption measurements. These variables form the basis of the analyses undertaken in response to RQ1, RQ2, and RQ3.

Tables 4.15 and 4.16 provide an overview of the variables included within the analytical dataset, their associated identifiers, scales of measurement, and descriptions.

Table 4.15: A description of all categorical dataset variables, detailing each factor's name, dataset identifier, underlying type, a short description of the variable, and the values associated with each factor level.

Factor	**Label**	**Type**	**Description**	**Values**
OpenJDK Version	jvm	C	An identifier specifying the version of the OpenJDK JRE and associated JVM.	OpenJDK version 9: j09 OpenJDK version 10: j10 OpenJDK version 11: j11 OpenJDK version 12: j12 OpenJDK version 13: j13 OpenJDK version 14: j14
GNU GCC Version	gcc	C	An identifier specifying the version of GNU GCC used in the building of the OpenJDK JRE.	GNU GCC version 5: GCC5 GNU GCC version 6: GCC6 GNU GCC version 7: GCC7 GNU GCC version 8: GCC8
Execution Mode	mode	C	An identifier specifying the JVM execution mode, either interpretive execution or tiered execution.	Interpreter: INT Tiered: HS
Garbage Collector	gc	C	An identifier specifying the variant of garbage collector used for heap memory management.	Serial: SGC Parallel: PGC Concurrent: CGC Garbage First: G1GC Concurrent + Parallel: CPGC
Workload Category	category	C	An identifier specifying a categorisation of the individual benchmark workloads.	apache concurrency database functional scala web
Benchmark	benchmark	C	A variable specifying the individual benchmark workload.	akka-uct, chi-square, db-shootout, dec-tree, dotty, finagle-chirper, finagle-http, fj-kmeans, future-genetic, gauss-mix, log-regression, mnemonics, movie-lens, naive-bayes, neo4j-analytics, page-rank, par-mnemonics, philosophers, reactors, rx-scrabble, scala-doku, scala-kmeans, scala-stm-bench7, scrabble

Note: C = Categorical Variable Type;

The analytical dataset therefore contains both experimentally specified configuration factors and observed or derived explanatory measures. OpenJDK source version, GNU GCC compiler version, JVM execution mode, garbage collector variant, and benchmark workload were specified within the experimental design,

while workload category provides a higher-level classification of the individual benchmark workloads. In contrast, the microarchitectural performance measures represent emergent runtime characteristics that arose during benchmark execution and were subsequently incorporated as explanatory predictors within the regression models developed in Chapter 7.

Table 4.16: A description of all numerical dataset variables, detailing each factor's name, dataset identifier, underlying type, and a short description of the variable.

Factor	**Label**	**Type**	**Description**
Branch miss-rate	bmr	N	The branch misprediction rate.
L1 data cache miss-rate	l1dlmr	N	The L1 data cache miss-rate.
L2 data cache miss-rate	l2dlmr	N	The L2 data cache miss-rate.
Page fault miss-rate	pfmr	N	Page fault miss-rate.
Average power use	meanW	N	Average power use measured.
St.Dev. power use	stdW	N	Standard deviation of average power use measurements.
Minimum power use	minW	N	Minimum power use measured.
Maximum power use	maxW	N	Maximum power use measured.
Power use samples taken	samplesW	N	Number of power use samples taken.
Energy consumed	energyJ	N	The total energy consumed in Joules.
Energy consumed	energyWh	N	The total energy consumed in Watt-hours.

Note: N = Numerical Variable Type;

The quantitative variables capture microarchitectural event measurements, benchmark execution characteristics, and electrical measurements obtained during benchmark execution. These include measures associated with branch prediction miss rates, cache utilisation, page fault activity, average power use, and energy consumption.

The analytical dataset therefore contains both experimentally manipulated factors and observed explanatory measures. OpenJDK source version, GNU GCC compiler version, JVM execution mode, garbage collector variant, workload category, and benchmark workload were defined by the experimental design. In

contrast, the microarchitectural performance measures represent emergent runtime characteristics that arose during benchmark execution and were subsequently incorporated as explanatory predictors within the regression models developed in Chapter 7.

4.9.4 Dataset Summary

Table 4.17 presents a summary of the principal characteristics of the analytical dataset used throughout this study. The dataset comprises benchmark-condition observations describing combinations of OpenJDK source versions, GNU GCC compiler versions, JVM execution modes, garbage collector variants, and benchmark workloads. Associated with each observation are measurements of power use, energy consumption, workload descriptors, and aggregated microarchitectural performance measures.

Table 4.17: Summary of the analytical dataset.

Dataset Component	**Count**
Benchmarkcondition observations	4,608
Categorical variables	6
Quantitative variables	47
Total variables	53
Benchmark workloads	24
Workload categories	6
OpenJDK source versions	6
GNU GCC compiler versions	4
JVM execution modes	2
Garbage collector variants	5

The summary highlights the overall scale of the dataset and provides context for the descriptive, inferential, and modelling procedures described in the subsequent sections. Collectively, these observations form the empirical foundation upon which the analyses associated with RQ1, RQ2, and RQ3 are based.

As illustrated in Table 4.17, the analytical dataset encompasses a broad range of runtime configuration factors, workload descriptors, microarchitectural

performance measures, and power use and energy consumption observations. The combination of categorical configuration variables and quantitative performance measures enables the investigation of both factor-level effects and explanatory relationships associated with JRE/JVM power use and energy consumption.

4.10 Statistical Analysis Framework

This section describes the statistical procedures employed to analyse the power use and energy consumption data collected throughout the experimental investigation. The analytical framework was designed to address Research Questions 1 and 2 through a combination of descriptive and inferential statistical techniques, while also providing the evidential foundation for the empirical modelling activities undertaken to address Research Question 3.

The analysis proceeded in three stages. First, descriptive statistical analyses were undertaken to characterise the distributions associated with power use and energy consumption across runtime configurations, workload category, and benchmark workloads. Second, inferential statistical procedures were applied to evaluate the presence and magnitude of differences between factor levels and to investigate workload-dependent variation in these relationships. Finally, the resulting observations informed the development of the empirical models presented in Chapter 7.

The statistical framework therefore provides the methodological foundation for identifying factor-level effects associated with JRE JVM power use and energy consumption (RQ1), examining the extent to which such effects vary across workload categories (RQ2), and informing the construction of predictive models of power use and energy consumption (RQ3).

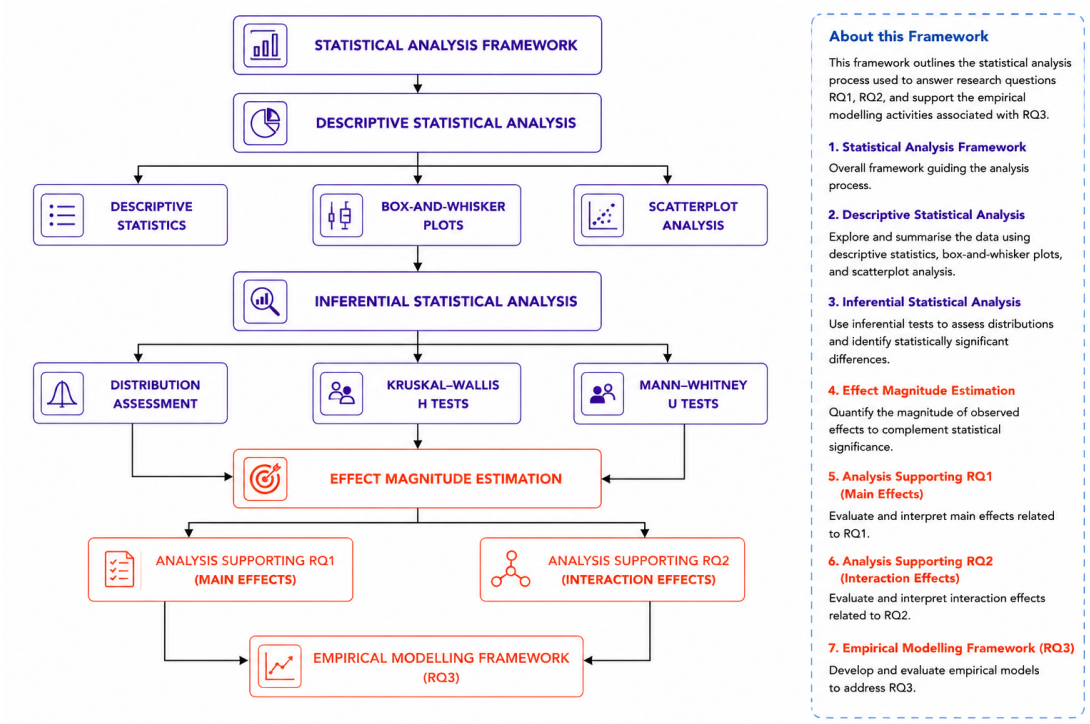


Figure 4.10: Overview of the statistical analysis framework employed to address RQ1 and RQ2, and to support the empirical modelling activities associated with RQ3.

Figure 4.10 presents an overview of the statistical analysis framework employed throughout this study. The framework illustrates the progression from descriptive statistical analyses, used to characterise the distributions associated with power use and energy consumption, through inferential statistical procedures used to evaluate factor-level differences and workload-dependent effects. The resulting observations subsequently informed the empirical modelling activities undertaken to address RQ3.

As illustrated, the analytical process commenced with descriptive statistical analyses involving both numerical summaries and graphical visualisations of the collected data. These analyses provided an initial understanding of the distributional characteristics associated with the runtime configurations under investigation. Inferential statistical procedures were then applied to evaluate the statistical significance and practical magnitude of observed differences. Collectively, these analyses provided the evidence used to address RQ1 through the evaluation of main effects and RQ2 through the examination of workload-dependent interaction

effects. Finally, the insights obtained from these analyses informed the specification and development of the empirical models presented later in this chapter.

4.10.1 Descriptive Statistical Analysis

Prior to inferential statistical testing, a comprehensive descriptive analysis was undertaken to characterise the distributions associated with power use and energy consumption. Descriptive analyses were performed at multiple levels of aggregation, including pooled observations, execution mode, runtime configuration factor, workload category, and individual workload level. The objective of these analyses was to identify central tendencies, dispersion characteristics, and distributional patterns that could provide insight into the behaviour of the JRE JVM configurations under investigation.

For each distribution, descriptive statistics including the minimum, maximum, mean, median, standard deviation, interquartile range, and sample size were calculated. In addition, graphical analyses were employed to facilitate the visual examination of distributional characteristics and relationships between variables. Box-and-whisker plots constituted the principal visualisation technique throughout the study and were used extensively to compare distributions across factor levels, execution modes, workload category, and benchmark workloads. These plots enabled the identification of differences in central tendency, dispersion, skewness, and potential outlying observations.

Where relationships between power use and energy consumption were examined, scatterplots were employed to visualise the association between the two measures and to identify potential trends, clusters, and deviations from expected patterns. Together, the descriptive statistics and graphical analyses provided a foundation for the subsequent inferential statistical procedures and empirical modelling activities. More specifically, they facilitated the exploration of factor-level differences associated with JRE JVM power use and energy consumption (RQ1.4), the examination of workload-dependent variation in these relationships (RQ1.4),

and the identification of patterns and associations relevant to the development of empirical models of power use and energy consumption (RQ1.4).

4.10.2 Inferential Statistical Analysis

Distribution Assessment

Prior to inferential statistical testing, the distributions associated with power use and energy consumption were examined to assess their distributional characteristics and to inform the selection of appropriate statistical procedures. Distributional assessments were undertaken across pooled observations, execution modes, runtime configuration factors, and workload category.

Normality was assessed through application of the Shapiro–Wilk statistical test. Across the majority of analyses, the resulting power use and energy consumption distributions exhibited statistically significant departures from normality ($p < .05$), together with varying degrees of skewness and heterogeneity in distributional shape. Consequently, non-parametric inferential procedures were selected throughout the analysis. Specifically, Mann–Whitney U tests were employed for two-group comparisons, while Kruskal–Wallis H tests were employed for factors containing more than two independent groups.

Kruskal–Wallis H Tests

For factors containing more than two independent groups, the Kruskal–Wallis H test was employed to evaluate differences in outcome variable distributions across multiple factor levels. As a non-parametric alternative to one-way analysis of variance (ANOVA), the procedure enables comparison of multiple independent groups through ranked observations without requiring normally distributed data.

A statistically significant Kruskal–Wallis H test indicates that at least one group differs from the others with respect to the distribution of the outcome variable. However, the test does not identify which specific groups differ. Consequently, where statistically significant results were observed, post-hoc pairwise compar-

isons were undertaken using Mann–Whitney U tests with Bonferroni-adjusted significance thresholds.

The Kruskal–Wallis H test is based upon the ranking of observations across all groups under comparison. Rather than comparing group means directly, the procedure evaluates whether the distributions of ranked observations differ between groups [205, 206]. The resulting test statistic, denoted by H, is calculated as:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{k} \frac{R_i^2}{n_i} - 3(N+1) \tag{4.4}$$

where:

N Total number of observations across all groups.

k Number of groups being compared.

n_i Number of observations within group i.

R_i Sum of ranks associated with group i.

Within the context of this study, statistically significant Kruskal–Wallis H test results were interpreted as evidence that at least one experimental factor level exhibited a distribution differing from the remaining groups. Subsequent pairwise Mann–Whitney U tests were then used to identify the specific groups responsible for the observed differences.

Mann–Whitney U Tests

The Mann–Whitney U test was employed to evaluate differences between factors containing two independent groups. Within the present study, the test was used primarily to compare interpretive and tiered execution modes and was also employed as a post-hoc procedure following statistically significant Kruskal–Wallis H tests. As a non-parametric alternative to the independent samples t-test, the procedure enables comparison of two independent groups through ranked observations.

A statistically significant Mann–Whitney U test indicates that the distributions associated with the two groups differ with respect to the outcome variable.

Consequently, the procedure was used both to identify direct differences between two-level factors and to determine which specific group pairs contributed to statistically significant Kruskal–Wallis H test results.

Similar to the Kruskal–Wallis procedure, the Mann–Whitney U test operates on ranked observations rather than raw values [206]. The resulting test statistic, denoted by U, is calculated as:

$$U = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} - \sum R_1 \tag{4.5}$$

where:

U	Mann–Whitney test statistic.
n_1	Number of observations within group 1.
n_2	Number of observations within group 2.
R_1	Sum of ranks associated with group 1.

Within the context of this study, statistically significant Mann–Whitney U test results were interpreted as evidence that the distributions associated with the two groups under comparison differed significantly. Where the procedure was employed as a post-hoc test following statistically significant Kruskal–Wallis H results, Bonferroni-adjusted significance thresholds were applied to control the family-wise error rate across multiple pairwise comparisons.

Null-Hypothesis Significance Testing Framework

The inferential analyses employed throughout this thesis are interpreted within a conventional null-hypothesis significance testing (NHST) framework. Rather than specifying a separate null and alternative hypothesis for every individual factor, outcome, workload-conditioned subset, execution-mode decomposition, and post-hoc comparison, a common hypothesis structure is applied systematically throughout the analyses addressing RQ1 and RQ2.

To illustrate this structure, consider SRQ1.1a, which examines whether JRE source version is associated with differences in JRE/JVM power use. For the

corresponding omnibus comparison, the statistical hypotheses can be represented as:

> H_0**:** JRE/JVM power-use distributions do not differ across the assessed JRE source versions.
>
> H_A**:** At least one assessed JRE source version exhibits a different JRE/JVM power-use distribution.

The corresponding inferential test evaluates the observed data against H_0. Where the resulting p-value falls below the predefined significance threshold of $\alpha = 0.05$, H_0 is rejected and the result is interpreted as providing statistical evidence of a difference between factor levels. Where $p \geq 0.05$, H_0 is not rejected; this is interpreted as insufficient statistical evidence to establish a difference rather than as proof that the factor levels are equivalent.

Where an omnibus test indicates a statistically significant difference, subsequent pairwise comparisons apply the same logic to individual factor-level contrasts. For example, a comparison between JRE source versions 9 and 13 evaluates:

> H_0**:** JRE/JVM power-use distributions do not differ between JRE source versions 9 and 13.
>
> H_A**:** JRE/JVM power-use distributions differ between JRE source versions 9 and 13.

Where multiple pairwise comparisons are undertaken, the corresponding adjustment procedure is applied before statistical significance is interpreted.

This hypothesis structure is applied consistently throughout the inferential analyses. For RQ1, the factor represented within H_0 and H_A changes according to the corresponding sub-research question, including JRE source version, GNU GCC build toolchain compiler version, garbage collector variant, workload category, JVM execution mode, and the relevant microarchitectural performance event measures. The dependent electrical outcome is correspondingly specified as power use or energy consumption.

For RQ2, the same framework is applied conditionally within workload category levels. The null hypothesis therefore represents the absence of a statistically detectable factor-level difference within the workload context under examination, while the alternative hypothesis represents the presence of at least one such difference. Where the analyses are additionally decomposed according to JVM execution mode, the hypotheses are evaluated separately within the interpretive and tiered execution subsets.

Accordingly, the inferential p-values reported throughout Chapters 5 and 6 provide the statistical evidence used to determine whether the corresponding null hypothesis is rejected or not rejected for each analysis. The specific factor, outcome, workload context, and execution-mode subset associated with each test are identified directly within the corresponding results section, avoiding the need to reproduce several hundred structurally equivalent null and alternative hypothesis statements.

For analyses concerning relationships between continuous microarchitectural performance event measures and the electrical outcomes, the corresponding null hypothesis specifies the absence of a statistical association between the variables, while the alternative hypothesis specifies the presence of a statistical association. The same decision rule based upon the predefined significance threshold is subsequently applied.

Effect Magnitude Estimation

While inferential statistical procedures provide evidence regarding the presence or absence of statistically significant differences, they do not necessarily convey the practical magnitude of those differences. Consequently, throughout this study, statistically significant findings were interpreted alongside measures of effect magnitude derived directly from the observed power use and energy consumption distributions.

Effect magnitudes were quantified through the examination of absolute differ-

ences between distributional point estimates, including differences in means and medians, together with proportional and percentage differences where appropriate. In addition, ratios comparing interpretive and tiered execution modes were calculated to provide a direct indication of the relative magnitude of execution mode effects across runtime configurations and workload classes.

This approach enabled statistically significant findings to be interpreted within the context of their practical importance. For example, factors exhibiting statistically significant differences in power use or energy consumption were further assessed in terms of the absolute magnitude of those differences, thereby distinguishing between effects that were statistically detectable and those that were materially meaningful from a power use and energy consumption perspective.

Accordingly, the interpretation of results throughout Chapters 5 and 6 considers both statistical significance and effect magnitude. This combined approach supports the evaluation of factor-level effects associated with JRE JVM power use and energy consumption (RQ1), workload-dependent variation in those effects (RQ2), and the practical relevance of relationships subsequently incorporated within the empirical models developed to address RQ3.

4.10.3 Analysis Supporting Research Question 1

The principal objective of Research Question 1 (RQ1) is to determine whether differences in JRE JVM power use and energy consumption are associated with runtime configuration factors, workload category, and observed microarchitectural performance measures. To address this objective, the descriptive and inferential statistical procedures described in Sections 4.10.1–4.10.2 were applied to each factor independently.

The analysis focused upon the evaluation of main effects associated with JRE source version (JSV), build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and workload category (WC) derived from the Renaissance Benchmark Suite. For each factor, power use and

energy consumption distributions were examined descriptively and subsequently evaluated using the appropriate non-parametric inferential procedures.

Factors containing more than two levels were analysed using Kruskal–Wallis H tests, while factors containing two levels were analysed using Mann–Whitney U tests. Where statistically significant differences were identified, post-hoc pairwise comparisons and effect magnitude assessments were undertaken to determine the practical significance of the observed differences.

The analyses undertaken in support of RQ1 evaluated the independent effects associated with JRE source version (JSV), build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and workload category (WC). For each factor, power use and energy consumption distributions were examined descriptively and subsequently evaluated using the inferential statistical procedures described previously.

JVM execution mode, comprising two levels, was analysed using the Mann–Whitney U test. JRE source version, build toolchain compiler version, garbage collector variant, and workload category, each comprising more than two levels, were analysed using the Kruskal–Wallis H test. Where statistically significant differences were identified, effect magnitude assessments were undertaken to evaluate the practical significance of the observed differences.

Collectively, these analyses provide the evidence used to determine whether individual runtime configuration factors, and workload category are associated with differences in JRE JVM power use and energy consumption addressing RQ1, presented in Chapter 5.

4.10.4 Analysis Supporting Research Question 2

The principal objective of Research Question 2 (RQ2) is to determine the extent to which the relationships between runtime configuration factors and JRE JVM power use and energy consumption vary across different workload categories. To address this objective, the inferential statistical procedures described previously

were applied separately within workload categories to examine whether the effects associated with JRE source version, build toolchain compiler version, and garbage collector variant remained consistent across differing workload contexts.

Rather than considering factors in isolation, the analyses undertaken for RQ2 examined the extent to which workload category conditioned the observed effects. This enabled the identification of workload-dependent patterns that may not be apparent when analysing pooled observations alone.

Accordingly, RQ2 extends the main-effects analyses undertaken for RQ1 by examining the contextual dependence of factor-level effects across workload categories, presented in Chapter 6.

Interaction Effects Analysis

Within the context of this study, an interaction effect refers to a situation in which the relationship between a runtime configuration factor and an outcome variable differs across workload category. Consequently, rather than assuming that the effects associated with JRE source version, build toolchain compiler version, or garbage collector variant remain constant across all workloads, the analyses undertaken for RQ2 examined whether these relationships were conditioned by workload category.

To evaluate such effects, the inferential statistical procedures described previously were applied separately within each workload category. Specifically, power use and energy consumption distributions associated with alternative JRE source versions, build toolchain compiler versions, and garbage collector variants were compared within the concurrency, database, web, functional, and Scala workload categories. This approach enabled the identification of workload-dependent differences that may not be apparent when analysing pooled observations alone.

The interaction effects analyses therefore extend the main-effects analyses undertaken for RQ1 by examining the extent to which workload context influences the observed relationships between runtime configuration factors and JRE JVM

power use and energy consumption. These analyses provide a more nuanced understanding of JRE JVM behaviour by identifying factors whose effects vary according to the computational characteristics of the executed workload.

The analyses undertaken in support of RQ2 extend the main-effects analyses conducted for RQ1 by examining whether the observed relationships remain consistent across differing workload contexts. Rather than considering all observations as a single pooled population, the analyses were repeated within workload categories to determine whether workload context conditioned the effects associated with runtime configuration factors.

This analytical strategy enables the identification of workload-dependent effects that may not be apparent when considering pooled observations alone and provides a basis for evaluating the extent to which workload category moderates the relationships between runtime configuration factors and observed power use and energy consumption.

4.11 Empirical Modelling Framework

The statistical analyses presented in the preceding sections were designed to address Research Question 1 (RQ1) and Research Question 2 (RQ2) through the identification of statistically significant differences, the estimation of effect magnitudes, and the evaluation of workload-conditioned interaction effects. Collectively, these analyses provide insight into whether OpenJDK JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures are associated with differences in JRE/JVM power use and energy consumption.

Research Question 3 (RQ3) extends beyond the identification of factor-level effects and interaction effects. Rather than asking whether differences exist, RQ3 seeks to determine the extent to which variation in JRE JVM power use and energy consumption can be accounted for through the combined influence of the manipulated runtime factors, workload category, and observed microarchitectural

performance measures. Consequently, the focus shifts from hypothesis testing and effect estimation towards explanatory modelling.

The objective of the empirical modelling framework is therefore to develop statistical models capable of quantifying the relationships between the study variables and the dependent measures of power use and energy consumption. In doing so, the framework provides an analytical mechanism through which the relative contribution of individual predictors can be evaluated, the combined explanatory power of multiple factors in accounting for variation can be assessed, and the extent to which observed variation in JRE JVM behaviour can be accounted for by the experimental and emergent factors considered throughout this study.

The modelling process was based upon hierarchical multiple linear regression. This approach was selected because it permits the sequential introduction of theoretically motivated predictor groups, enabling the incremental contribution of individual factor categories to be evaluated. Furthermore, hierarchical modelling facilitates the assessment of changes in explained variance as additional predictors are introduced, thereby providing insight into the relative explanatory importance of JRE source version, build toolchain compiler version, execution mode, garbage collection strategy, workload category, and microarchitectural performance measures.

The following subsections describe the procedures used to prepare the predictor variables, specify the regression models, evaluate the assumptions associated with linear regression, assess model adequacy, and estimate model coefficients. In addition, bootstrap resampling techniques were employed to evaluate the stability and robustness of the resulting regression coefficients and associated standard errors. Collectively, these procedures provide the methodological foundation for addressing RQ3 and for developing explanatory models of JRE JVM power use and energy consumption.

4.11.1 The General Framework

At a conceptual level, the modelling process assumes that observed power use and energy consumption arise from a combination of factors associated with the OpenJDK source version from which the JRE was constructed, the GNU GCC build toolchain compiler version used during JRE construction, the JVM execution mode, the garbage collector variant, the workload category, and a collection of microarchitectural performance measures describing emergent runtime behaviour. The general form of the empirical modelling framework is presented in Equations 4.6 and 4.7, which express power use and energy consumption, respectively, as functions of these explanatory variables.

These equations are intended to represent the general structure of the modelling framework rather than the final fitted models. The subsequent modelling procedure employed hierarchical multiple linear regression to evaluate the incremental contribution of each factor group, assess model assumptions, and determine the extent to which the specified variables account for variation in JRE/JVM power use and energy consumption.

$$\begin{aligned} P_i = \beta_0 + \beta_1(JSV_i) + \beta_2(BTCV_i) + \beta_3(JEM_i) \\ + \beta_4(GCV_i) + \beta_5(WC_i) + \beta_6(MPE_i) + \varepsilon_i \end{aligned} \tag{4.6}$$

$$\begin{aligned} E_i = \beta_0 + \beta_1(JSV_i) + \beta_2(BTCV_i) + \beta_3(JEM_i) \\ + \beta_4(GCV_i) + \beta_5(WC_i) + \beta_6(MPE_i) + \varepsilon_i \end{aligned} \tag{4.7}$$

The microarchitectural performance measures (MPE) differ conceptually from the remaining predictors included within the model. Whereas JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category represent experimentally controlled factors, the microarchitectural measures represent observed runtime characteristics that emerge during

benchmark execution. Consequently, the inclusion of MPE variables within the modelling framework should be interpreted as capturing relationships between observed processor behaviour and JRE JVM power use and energy consumption rather than the effects of experimentally manipulated factors.

An important feature of the empirical modelling framework is that the predictor variables incorporated within the regression models represent two distinct methodological classes. System-level runtime configuration variables were directly manipulated as part of the experimental design, whereas the microarchitectural performance event measures were not directly controlled but emerged dynamically during workload execution. Consequently, the empirical models combine experimentally controlled predictor variables with observed runtime execution measures, requiring careful interpretation of the relationships identified by the resulting statistical models.

4.11.2 Dummy Coding of Regression Predictors

The hierarchical regression models developed in this study incorporate both categorical and continuous predictor variables. The categorical predictors consist of OpenJDK JRE source version (JSV), GNU GCC build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and workload category (WC). In contrast, the microarchitectural performance event variables are continuous measures obtained during benchmark execution.

Ordinary least squares regression requires that all predictors be represented numerically within the model design matrix. While continuous variables can be entered directly, categorical variables must first be transformed into an appropriate numerical representation. Consequently, all categorical predictors were represented using dummy-coded indicator variables [33].

For a categorical predictor containing k levels, $k - 1$ dummy variables were constructed, with one level omitted and treated as the reference category. Each dummy variable was coded as either 0 or 1, indicating non-membership or mem-

bership of the corresponding category level. The estimated regression coefficients therefore represent the expected change in the dependent variable relative to the selected reference category while holding all other predictors constant.

The use of dummy-coded indicator variables is a standard and widely accepted approach within the general linear modelling framework and permits categorical predictors to be incorporated within ordinary least squares regression models without violating the underlying assumptions associated with model estimation [33].

The following reference categories were adopted throughout the regression analysis:

- OpenJDK 9 for JRE source version (JSV);
- GCC5 for build toolchain compiler version (BTCV);
- Tiered execution for JVM execution mode (JEM);
- CGC for garbage collector variant (GCV);
- Apache Spark for workload category (WC).

Accordingly, all estimated regression coefficients associated with these factors should be interpreted relative to their respective reference category.

4.11.3 Model Specification

The general modelling framework presented in Equations 4.6 and 4.7 provides a conceptual representation of the relationships examined within this study. Specifically, the framework identifies the principal categories of explanatory factors hypothesised to influence JRE/JVM power use and energy consumption, namely OpenJDK source version, build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance behaviour.

For empirical estimation, however, these high-level factor categories must be operationalised as measurable predictor variables that can be incorporated within a regression model. Consequently, the conceptual microarchitectural performance event (MPE) construct was decomposed into a collection of specific performance measures, including branch misprediction rate (BMR), L1 data cache miss-rate (L1DLMR), L2 data cache miss-rate (L2DLMR), and page fault rate (PFR). Similarly, the categorical runtime factors were represented using the dummy-coded variables described in the preceding subsection.

Equations 4.8 and 4.9 therefore present the full operationalised regression models used within the hierarchical modelling procedure. These equations represent the complete model specification considered within the analysis and incorporate all experimentally specified factors together with the retained microarchitectural performance event measures. Subsequent stages of the modelling process evaluate the incremental explanatory contribution associated with each variable group and examine the extent to which the resulting model specifications account for variation in power use and energy consumption.

$$\begin{aligned} P_i = \beta_0 &+ \sum_{j=1}^{5} \beta_{1j} \mathrm{JSV}_{ij} + \sum_{k=1}^{3} \beta_{2k} \mathrm{BTCV}_{ik} + \beta_3 \mathrm{JEM}_i \\ &+ \sum_{l=1}^{4} \beta_{4l} \mathrm{GCV}_{il} + \sum_{m=1}^{4} \beta_{5m} \mathrm{WC}_{im} \\ &+ \beta_7 \mathrm{BMR}_i + \beta_8 \mathrm{L1DLMR}_i + \beta_9 \mathrm{L2DLMR}_i + \beta_{10} \mathrm{PFMR}_i \\ &+ \varepsilon_i \end{aligned} \tag{4.8}$$

$$\begin{aligned} E_i = \beta_0 &+ \sum_{j=1}^{5} \beta_{1j} \mathrm{JSV}_{ij} + \sum_{k=1}^{3} \beta_{2k} \mathrm{BTCV}_{ik} + \beta_3 \mathrm{JEM}_i \\ &+ \sum_{l=1}^{4} \beta_{4l} \mathrm{GCV}_{il} + \sum_{m=1}^{4} \beta_{5m} \mathrm{WC}_{im} \\ &+ \beta_7 \mathrm{BMR}_i + \beta_8 \mathrm{L1DLMR}_i + \beta_9 \mathrm{L2DLMR}_i + \beta_{10} \mathrm{PFMR}_i \\ &+ \varepsilon_i \end{aligned} \tag{4.9}$$

In Equations 4.8 and 4.9, the dependent variables P_i and E_i represent mean

power use and energy consumption for observation i, respectively. The categorical predictors are represented through dummy-coded indicator variables corresponding to OpenJDK source version (JSV), GNU GCC build toolchain compiler version (BTCV), garbage collector variant (GCV), and workload category (WC). JVM execution mode (JEM) is represented as a binary indicator variable. For each categorical predictor containing k levels, the corresponding summation extends from 1 to $k-1$, reflecting the use of $k-1$ dummy variables when one level is designated as the reference category.

The remaining predictors correspond to measured microarchitectural performance events observed during benchmark execution. BMR represents the branch misprediction rate, L1DLMR represents the L1 data-load miss rate, L2DLMR represents the L2 data-load miss rate, and PFMR represents the page-fault miss rate. The error terms ε_i^P and ε_i^E represent the unexplained variation in the power use and energy consumption models.

4.11.4 Hierarchical Regression Procedure

To construct the empirical regression models, hierarchical multiple linear regression was used to model variation in average power use and energy consumption. Predictor variables were introduced sequentially according to a theoretically motivated layered systems architecture reflecting progressively deeper levels of system-level influence on observed JRE JVM execution behaviour. This hierarchical modelling strategy permits assessment of the incremental explanatory contribution associated with successive groups of predictors while preserving alignment with the conceptual framework developed earlier in this thesis.

The hierarchical regression procedure was structured using four sequential predictor blocks. The first block represented foundational runtime implementation characteristics and included OpenJDK JRE source version and GNU GCC build toolchain compiler version. These variables define the underlying executable runtime environment prior to program execution and represent lower-level imple-

mentation decisions made during runtime construction.

The second block captured execution strategy characteristics and included JVM execution mode together with garbage collector variant. These variables determine how program instructions are executed and how runtime memory management is performed during execution, thereby representing higher-level runtime configuration decisions governing program execution behaviour.

The third block introduced workload category, representing the external computational demands imposed upon the configured runtime environment. This block was included separately in order to quantify the extent to which differences in workload category account for variation in observed power use and energy consumption beyond the directly specified runtime configuration factors.

The fourth and final block incorporated emergent microarchitectural performance event measures, specifically branch misprediction rate, L1 data cache load miss rate, L2 data cache load miss rate, and page fault rate. Unlike the preceding predictor groups, these variables are not directly specified prior to execution but instead emerge dynamically through the interaction between workload category, runtime configuration, and the underlying hardware architecture.

This hierarchical structure therefore reflects progressively deeper layers of system behaviour, enabling assessment of the extent to which lower-level emergent execution characteristics provide incremental explanatory power beyond the directly specified runtime configuration variables. Separate hierarchical regression models were subsequently estimated for both power use and energy consumption using this common theoretical specification.

4.11.5 Model Diagnostics

The validity and interpretability of ordinary least squares (OLS) regression models depend upon a number of assumptions concerning the structure of the relationship between the predictor variables and dependent variables, the behaviour of model residuals, and the degree of dependency among the predictor variables themselves.

Consequently, prior to the interpretation of any fitted regression model, a series of diagnostic procedures was undertaken to assess the adequacy and robustness of the regression models developed in response to Research Question 3.

The diagnostic process focused on the principal assumptions associated with the Gauss–Markov framework [207] and the practical application of multiple linear regression. Specifically, the analyses examined the adequacy of the linear model specification, residual distribution characteristics, homoscedasticity, multicollinearity, and the presence of influential observations.

The adequacy of the linear model specification was assessed through visual inspection of residual plots and plots of standardised residuals against fitted values. These diagnostics were used to evaluate whether systematic non-linear patterns remained within the residual structure, which would indicate that the assumed linear functional form may provide an inadequate representation of the underlying relationships between the predictors and the dependent variables.

Homoscedasticity was assessed through examination of residual variance patterns across the fitted value range, together with formal statistical assessment using the Breusch–Pagan test. Because ordinary least squares regression assumes constant residual variance, evidence of heteroskedasticity may result in unreliable standard errors and invalid statistical inference when conventional model-based standard error estimates are used.

The distributional characteristics of model residuals were evaluated using quantile–quantile (Q–Q) plots, residual histograms, and the Shapiro–Wilk test. Although ordinary least squares regression does not require the dependent variables themselves to be normally distributed, substantial departures from residual normality may affect the reliability of confidence intervals and statistical significance tests. Given the large sample size employed within this study, greater interpretive emphasis was placed on visual diagnostic assessment than on formal significance testing alone.

Multicollinearity among the predictor variables was assessed through the cal-

culation of Generalised Variance Inflation Factor (GVIF) statistics. Excessive collinearity between predictors may result in unstable coefficient estimates and inflated standard errors, thereby reducing the reliability of coefficient interpretation. Particular attention was given to the microarchitectural performance event measures, as several variables capture partially related aspects of underlying processor execution behaviour.

Potentially influential observations were identified through the examination of Cook's distance statistics together with standardised residuals. These diagnostics were used to determine whether a relatively small number of observations exerted disproportionate influence over model coefficient estimation. Because the dataset incorporates a highly heterogeneous collection of runtime configurations and workload categories, influential observations were interpreted cautiously, recognising that such observations may reflect valid but structurally distinct execution contexts rather than anomalous measurement error.

Where assumption violations were identified, their potential impact on model interpretation was subsequently evaluated. Because heteroskedasticity was anticipated given the heterogeneous structure of the experimental design, coefficient estimates were additionally evaluated using heteroskedasticity-consistent robust standard errors (HC3 variance estimation), which provide more reliable inference under conditions of non-constant residual variance.

To further assess model robustness, sensitivity analysis was performed through re-estimation of the regression models following exclusion of observations identified as influential under Cook's distance diagnostics. This procedure permitted assessment of the extent to which overall explanatory performance and coefficient estimation remained stable when highly influential observations were removed.

Finally, bootstrap resampling procedures were employed to obtain empirical estimates of coefficient variability and confidence interval stability. Stratified non-parametric bootstrap resampling was used in order to preserve the underlying experimental design structure while assessing the extent to which the fitted

regression coefficients remained stable under repeated resampling.

Collectively, these diagnostic and robustness procedures provide evidence regarding the adequacy, stability, and interpretive reliability of the fitted regression models, thereby supporting subsequent interpretation of the explanatory relationships developed in response to Research Question 3.

4.11.6 Bootstrap Estimation of Regression Coefficients and Standard Errors

Ordinary least squares (OLS) regression relies upon a number of assumptions associated with the Gauss–Markov theorem. Under these assumptions, including linearity, independence of observations, homoscedasticity of residuals, and the absence of perfect multicollinearity, the estimated regression coefficients are the Best Linear Unbiased Estimators (BLUE) of the underlying population parameters [207]. Violations of these assumptions can affect the reliability of parameter estimates and, in particular, the accuracy of the associated standard errors, confidence intervals, and statistical significance tests. Consequently, inference based solely upon conventional OLS estimates may be sensitive to departures from the assumptions underpinning the model.

A number of approaches have been proposed to address assumption violations, including transformations of the response and predictor variables, the adoption of robust standard error estimators, and the use of alternative modelling frameworks. An additional and widely adopted approach is bootstrap resampling. The bootstrap, originally introduced by [208], is a non-parametric resampling procedure that approximates the sampling distribution of a statistic directly from the observed data. Rather than relying upon theoretical distributional assumptions, the bootstrap repeatedly samples, with replacement, from the original dataset and recalculates the statistic of interest for each resampled dataset.

In the context of regression modelling, bootstrap estimation can be used to evaluate the stability of regression coefficients and to obtain empirical estimates

of their sampling distributions and standard errors. Fox [209] describes the application of bootstrap procedures within regression analysis and demonstrates how resampling techniques can be used to assess parameter uncertainty when traditional assumptions may be difficult to verify or only approximately satisfied.

For a dataset containing n observations, a bootstrap sample is generated by randomly drawing n observations with replacement from the original dataset. Because sampling is performed with replacement, individual observations may appear multiple times within a given bootstrap sample, while others may be omitted. A regression model is then fitted to the resampled dataset and the coefficients are recorded. This procedure is repeated a large number of times, producing an empirical sampling distribution for each regression coefficient.

Let $\hat{\beta}_j^{(b)}$ denote the estimate of coefficient β_j obtained from bootstrap sample b, where $b = 1, \ldots, B$. The bootstrap estimate of the coefficient standard error is then given by

$$SE_{\text{boot}}(\hat{\beta}_j) = \sqrt{\frac{1}{B-1} \sum_{b=1}^{B} \left(\hat{\beta}_j^{(b)} - \bar{\beta}_j \right)^2} \tag{4.10}$$

whereas

$$\bar{\beta}_j = \frac{1}{B} \sum_{b=1}^{B} \hat{\beta}_j^{(b)} \tag{4.11}$$

represents the mean bootstrap coefficient estimate.

The principal advantage of the bootstrap is that it does not require the sampling distribution of the regression coefficients to conform to a particular theoretical distribution. Instead, the distribution is estimated empirically from the observed data. Consequently, bootstrap estimates often provide more reliable assessments of parameter uncertainty when residual distributions exhibit non-normality, heteroscedasticity, or other departures from classical modelling

assumptions.

Within this study, bootstrap resampling was employed as a supplementary validation procedure for the final regression model specifications. The objective was to assess the stability of the estimated regression coefficients and their associated standard errors, and to evaluate the extent to which the reported regression relationships were sensitive to sampling variability or minor departures from the assumptions underlying ordinary least squares (OLS) regression.

Bootstrap estimation was implemented using the `Boot()` function provided by the `car` package within the R statistical computing environment, which internally employs the resampling framework of the `boot` package [210]. A total of $R = 1000$ bootstrap replications were generated for each model.

For each bootstrap replication, a new dataset of equal size to the original sample was generated through random sampling with replacement. The regression model was subsequently re-estimated and the resulting coefficient estimates recorded. Repeating this procedure 1,000 times produced an empirical sampling distribution for each regression coefficient, from which bootstrap estimates of coefficient means, standard errors, and confidence intervals were obtained.

Because observations were grouped according to the principal experimental factors, namely JRE Source Version (JSV), Build Toolchain Compiler Version (BTCV), JVM Execution Mode (JEM), Garbage Collector Variant (GCV), and Workload Category (WC), a stratified bootstrap procedure was employed using the `strata` argument. This ensured that the relative representation of observations associated with each experimental configuration was preserved throughout the resampling process. Maintaining the original factor structure during resampling reduced the possibility that individual bootstrap samples would over- or under-represent particular runtime configurations or workload categories, thereby producing more stable and representative estimates of coefficient uncertainty.

The resulting bootstrap coefficient estimates and standard errors were compared with their corresponding OLS estimates. Consistency between the bootstrap

and OLS results was interpreted as evidence of coefficient stability and model robustness, providing additional confidence in the reliability of the fitted regression models.

Program Segment 4.7: Bootstrap estimation of regression coefficients and standard errors.

```
m.boot <- Boot(
    lm.model.final,
    R = 1000,
    strata = data.all.df$strata
)
```

Program Segment 4.7 presents the `R` code used to perform the bootstrap procedure. In Program Segment 4.7, the argument `lm.model.final` specifies the final regression model subjected to bootstrap analysis, `R = 1000` defines the number of bootstrap replications, and the `strata` argument ensures that observations are resampled within their respective experimental strata. This approach preserves the representation of the principal experimental factor combinations across bootstrap samples and reduces the likelihood that coefficient estimates are unduly influenced by imbalances introduced during the resampling process.

4.11.7 Analysis Supporting Research Question 3

Research Question 3 seeks to determine the extent to which variation in JRE/JVM power use and energy consumption can be explained through the combined influence of OpenJDK source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and emergent microarchitectural performance measures. Unlike Research Questions 1 and 2, which focus on identifying statistically significant differences and workload-conditioned interaction effects, Research Question 3 is concerned with explanatory modelling of the variation in power use and energy consumption.

To address this question, two hierarchical multiple linear regression models

were developed, one modelling mean power use and the second modelling energy consumption. Hierarchical regression was selected because it permits theoretically related groups of predictors to be introduced sequentially, thereby enabling the incremental contribution of each factor category to explained variance to be evaluated. This approach provides insight not only into the overall explanatory power of the model, but also into the relative importance of individual predictor groups.

The modelling process proceeded through a series of nested regression models. The initial stages evaluated the explanatory contribution of experimentally manipulated factors, namely OpenJDK source version (JSV), GNU GCC build toolchain compiler version (BTCV), JVM execution mode (JEM), garbage collector variant (GCV), and workload category (WC). Subsequent stages incorporated microarchitectural performance event measures (MPE), representing emergent runtime behaviour observed during benchmark execution. Changes in explained variance (R^2) and adjusted explained variance (R^2_{adj}) were evaluated following the introduction of each predictor block in order to assess the additional variation accounted for by the newly introduced variables.

Model adequacy was assessed through standard regression diagnostic procedures, including evaluations of multicollinearity, residual behaviour, homoscedasticity, and influential observations. The hierarchical modelling procedure was then used to evaluate the incremental explanatory contribution associated with each theoretically specified variable group and to assess the extent to which the final model specifications accounted for variation in JRE/JVM power use and energy consumption. Finally, bootstrap resampling was employed to assess the stability of the resulting regression coefficients and associated standard errors.

Collectively, these analyses provide the basis for answering Research Question 3 by quantifying the proportion of variation in JRE/JVM power use and energy consumption that can be explained by the experimental and emergent factors considered throughout this study, and by identifying the factors that contribute

most substantially to that explanatory power.

4.12 Threats to Validity

As with all empirical investigations, the findings reported in this thesis should be interpreted in light of several potential threats to validity. Following established practice in software engineering and empirical systems research, threats are considered from the perspectives of internal validity, construct validity, external validity, and conclusion validity. Where possible, methodological decisions were adopted to mitigate these threats; however, some limitations remain and should be considered when interpreting the results.

4.12.1 Internal Validity

Internal validity concerns the extent to which observed differences in power use and energy consumption can be attributed to the experimental factors under investigation rather than to uncontrolled sources of variation.

Hardware variability.

Although all benchmark executions were conducted on Raspberry Pi 4 Model B devices with identical hardware specifications, minor manufacturing differences between individual boards may influence runtime behaviour. To mitigate this threat, all nodes were configured using identical hardware, operating-system images, storage media, and runtime environments. Furthermore, the experimental reliability assessment demonstrated stable benchmark execution behaviour across cluster nodes, providing evidence that node-to-node variability was limited.

Operating-system and background process interference.

Background operating-system services and network activity may influence processor utilisation, power use, and execution behaviour. This threat was miti-

gated through the use of a lightweight headless installation of Raspbian Buster Lite deployed from a common cloned image across all experimental nodes. Although SSH services and Ethernet connectivity remained enabled for administrative purposes, no active SSH sessions were present during benchmark execution and no benchmark-generated network traffic was exchanged. Benchmark execution therefore occurred in the absence of user-initiated network activity, reducing the likelihood that network communication influenced the observed power use, energy consumption, or microarchitectural performance measurements.

Measurement instrumentation effects.

The collection of microarchitectural performance events introduces execution overhead that could influence power use and energy consumption measurements. To minimise this threat, microarchitectural performance data, and power use and energy consumption measurements were collected during separate benchmark executions. Consequently, the power use and energy consumption observations reported throughout this thesis were obtained without performance-monitoring instrumentation enabled.

Runtime configuration consistency.

Differences in JVM configuration unrelated to the experimental factors could confound observed results. To mitigate this threat, heap allocation parameters, operating-system configuration, benchmark deployment procedures, and runtime initialisation processes were standardised across all benchmark executions.

JRE build process variability.

Because all OpenJDK instances were built from source, uncontrolled variation in the build process could influence resulting runtime behaviour. This threat was mitigated through the use of reproducible build procedures, controlled compiler versions, explicit build configurations, and reliability testing of homogeneous JRE builds.

Thermal effects.

Extended benchmark execution may alter processor temperature and influence both performance and power use. Although all benchmark executions were conducted under comparable laboratory conditions, temperature was not explicitly controlled or incorporated as an experimental factor. Consequently, residual thermal effects remain a potential source of uncontrolled variation.

4.12.2 External Validity

External validity concerns the extent to which the findings may be generalised beyond the specific experimental environment employed within this study.

Hardware platform generalisability.

All experiments were conducted using Raspberry Pi 4 Model B devices based on the ARM Cortex-A72 microarchitecture. Consequently, the findings may not generalise directly to alternative processor architectures, including x86-64 server-class systems, cloud-computing infrastructures, or newer ARM-based platforms.

OpenJDK version range.

The study examined OpenJDK versions 9 through 14. Although these versions encompass a substantial period of JVM evolution, the findings may not generalise directly to earlier legacy releases or to more recent OpenJDK versions beyond those considered in this investigation. OpenJDK development has continued beyond this version range, including subsequent changes to JVM runtime, compiler, garbage collection, memory-management, and concurrency mechanisms. These later developments may alter the relationships between runtime configuration, microarchitectural performance event measures, power use, and energy consumption observed within the present study. Consequently, the findings should not be interpreted as demonstrating that the effects identified across OpenJDK version 9 through 14 necessarily persist unchanged within later JRE/JVM releases.

Assessment of their stability across more recent OpenJDK versions therefore represents an appropriate direction for future empirical investigation.

Compiler version range.

Only four GNU GCC compiler versions were examined. Alternative compiler families, optimisation strategies, or link-time optimisation approaches may produce different runtime characteristics and therefore different power use and energy consumption outcomes.

Benchmark representativeness.

The Renaissance Benchmark Suite was selected because it provides a diverse collection of contemporary JVM workloads. Nevertheless, no benchmark suite can fully represent the complete spectrum of real-world enterprise, scientific, embedded, mobile, and cloud-native JVM applications. Consequently, caution should be exercised when generalising findings beyond the workload classes represented within the benchmark suite.

Runtime deployment environments.

The study was conducted within a controlled experimental environment. Real-world deployments may exhibit additional variability arising from user interaction, operating-system workloads, virtualisation, containerisation, network traffic, and multi-tenant execution environments. These factors may influence observed power use and energy consumption characteristics in practice.

4.12.3 Construct Validity

Construct validity concerns the degree to which the operational measures employed within the study accurately represent the underlying concepts of interest.

Representation of power use and energy consumption.

Power use and energy consumption were operationalised using measurements obtained from a Joulescope precision DC energy analyser. These measurements provide direct observations of electrical power and energy at the Raspberry Pi power boundary. However, they do not incorporate energy consumption associated with the supporting experimental infrastructure, including networking equipment, measurement hosts, and external services. Consequently, the study measures platform-level JRE/JVM power use and energy consumption rather than the energy consumption of the complete experimental infrastructure.

Workload category abstraction.

The workload category factor groups benchmark applications into six broad categories. While these categories provide a useful abstraction for analysis, they may not fully capture all workload categorisations relevant to power use and energy consumption.

Microarchitectural performance events as explanatory variables.

Microarchitectural performance events were treated as emergent explanatory variables rather than directly manipulated factors. Although these measures provide insight into runtime execution behaviour, they represent observable indicators of underlying computational processes rather than direct measures of those processes themselves. Consequently, they should be interpreted as proxies for the mechanisms through which runtime configurations influence power use and energy consumption.

Cold-start execution.

All benchmark workloads were executed under cold-start conditions without JVM warm-up. This design decision was necessary to facilitate direct comparisons between interpretive and tiered execution modes. However, the resulting measurements may not fully represent long-running production deployments in which just-in-time compilation has reached a steady-state operating condition.

4.12.4 Conclusion Validity

Conclusion validity concerns the extent to which the statistical conclusions drawn from the experimental data are justified.

Assumption violations.

Several experimental distributions exhibited departures from normality. To address this threat, non-parametric statistical procedures were employed where appropriate, and regression diagnostics were undertaken prior to model interpretation. Bootstrap estimation was additionally employed to assess the stability of regression coefficients and standard errors under potential assumption violations.

Multiple comparisons.

The large number of factor-level and workload-level comparisons increases the possibility of Type I error. Accordingly, statistical significance was interpreted alongside measures of effect magnitude and practical significance rather than being considered in isolation.

Model specification error.

The regression models developed in this thesis represent simplified approximations of complex runtime behaviour. Additional factors not included within the modelling framework may contribute to the unexplained variation observed within the dependent variables. Consequently, the models should be interpreted as explanatory in accounting for variation rather than strictly causal representations of JRE/JVM power use and energy consumption.

Measurement precision.

Although all physical measurement systems are subject to calibration tolerances and instrument uncertainty, power use and energy consumption measurements were acquired using a Joulescope JS110 precision DC energy analyser that

was factory calibrated by the manufacturer prior to deployment within the experimental platform. The device employs simultaneous voltage and current sampling and provides measurement accuracy substantially greater than the magnitude of the differences observed between many experimental conditions. Consequently, while some degree of measurement uncertainty cannot be eliminated entirely, it is unlikely that instrument error materially influenced the comparative conclusions drawn throughout this study.

Overall, while these threats cannot be eliminated entirely, the controlled experimental design, standardisation of hardware and software environments, reliability assessments, measurement procedures, and statistical validation techniques collectively reduce their potential influence on the findings reported throughout this thesis.

4.13 Chapter Summary

This chapter presented the methodological framework employed to investigate the factors influencing JRE/JVM power use and energy consumption. The chapter began by establishing the theoretical and conceptual foundations underpinning the study, identifying the principal runtime, compiler, memory management, workload, and microarchitectural factors hypothesised to influence the behaviour of managed runtime environments. These factors were subsequently operationalised through an experimental framework that linked the construction of JRE/JVM instances, benchmark execution, data acquisition, and statistical analysis to the research questions guiding the investigation.

The experimental platform was then described in detail, including the Raspberry Pi 4 Model B cluster, the operating-system environment, measurement instrumentation, and supporting software infrastructure. The procedures used to construct the GNU GCC compiler toolchains and OpenJDK JRE/JVM instances were presented, together with the runtime configuration factors defining the experimental treatments. The Renaissance Benchmark Suite was introduced as the

principal workload collection used throughout the study, and a workload category factor were defined to support the subsequent analyses.

The chapter further detailed the data collection procedures used to acquire power use, energy consumption, and microarchitectural performance event measurements. Particular attention was given to benchmark execution procedures, measurement synchronisation, reliability assessment, and dataset construction. The resulting dataset provided the empirical foundation upon which the statistical analyses and modelling activities reported in later chapters were undertaken.

Finally, the statistical analysis framework and empirical modelling framework were presented. The statistical framework defined the descriptive and inferential procedures used to investigate the main effects associated with individual factors and the workload-conditioned interaction effects examined throughout the study. The empirical modelling framework detailed the regression-based approach used to develop explanatory models of JRE/JVM power use and energy consumption, together with the associated diagnostic and bootstrap validation procedures.

Collectively, the methodological components presented in this chapter establish a controlled and reproducible framework for evaluating JRE/JVM power use and energy consumption. The methodology provides the basis for addressing Research Question 1 concerning the main effects associated with JRE source version, compiler version, execution mode, garbage collection strategy, workload category, and microarchitectural behaviour; Research Question 2 concerning workload-conditioned interaction effects; and Research Question 3 concerning the development of empirical models capable of explaining variation in JRE/JVM power use and energy consumption.

The following chapter presents the results associated with Research Question 1, examining the main effects of the experimental factors on JRE/JVM power use and energy consumption.

Chapter 5

Research Question 1: Main Effects

This chapter addresses **Research Question 1 (RQ1)**, which examines how differences in runtime configuration, workload category, and microarchitectural performance event measures influence observed JRE/JVM power use and energy consumption. Specifically, this chapter evaluates the extent to which variation in JRE source version, GNU GCC build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures are associated with measurable differences in JRE/JVM power use and energy consumption.

To systematically evaluate these effects, the chapter adopts a staged analytical approach. We begin by examining the independent effects associated with OpenJDK JRE version §5.2, GNU GCC build compiler version §5.3, garbage collector variant §5.4, and workload category §5.5. These analyses establish baseline characteristics and evaluate the extent to which each predictor independently contributes to observed differences in JRE/JVM power use and energy consumption.

In addition, JVM execution mode is treated throughout this chapter as a cross-sectional analytical dimension. Accordingly, all predictor effects are decomposed according to interpretive and tiered execution to enable direct comparison of execution mode effects across differing runtime conditions.

The final section examines microarchitectural performance event measures

§5.6, focusing on branch misprediction rates, cache miss activity, and page fault behaviour. These analyses seek to characterise lower-level processor execution characteristics associated with observed differences in power use and energy consumption, while evaluating the extent to which microarchitectural performance event measures are statistically associated with the observed outcome measures.

Across all analyses, descriptive assessments of the underlying distributions are presented first, followed by inferential statistical analyses examining differences between predictor groups and execution modes. Given the distributional properties of the observed measures, non-parametric statistical procedures are employed where appropriate.

Collectively, the analyses presented in this chapter establish which runtime configuration factors, workload category, and execution characteristics exert the strongest influence on observed JRE/JVM power use and energy consumption, while identifying the runtime conditions under which interpretive and tiered execution differ most substantially.

5.1 How this chapter addresses RQ1

This chapter addresses Research Question 1 (RQ1), which asks:

> ***RQ1****: How do variations in JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category affect JRE JVM power use and energy consumption, and what relationships exist between these outcomes and observed microarchitectural performance characteristics?*

RQ1 is addressed through a structured inferential statistical analysis in which each predictor variable is examined independently, while all analyses are additionally decomposed according to execution mode to assess whether observed effects differ between interpretive and tiered execution contexts. Specifically, this chapter evaluates whether statistically significant differences in JRE JVM power use and

energy consumption emerge as a function of JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, and workload category. In addition, relationships between microarchitectural performance measures and observed power use and energy consumption are assessed.

Each section therefore contributes evidence toward answering RQ1 by determining whether a given system-level factor exhibits a statistically significant effect, the magnitude and direction of that effect, and whether such effects remain stable across interpretive and tiered execution contexts. Collectively, the analyses establish which factors materially influence JRE JVM power use and energy consumption and under what runtime conditions these effects occur.

5.1.1 Statistical Assumption Assessment

Prior to inferential analysis, distributional assumptions were systematically assessed across all experimental conditions using the Shapiro-Wilk test. In the majority of cases, normality assumptions were violated across both pooled observations and execution-mode decomposed subsets.

Accordingly, non-parametric statistical procedures were adopted as the primary inferential framework throughout this chapter. Kruskal-Wallis tests were used for multi-group comparisons, Wilcoxon rank-sum tests were used for pairwise execution-mode comparisons, and post-hoc pairwise Wilcoxon comparisons with Bonferroni adjustment were conducted where omnibus significance was observed.

Individual subsection-level assumption testing results are only reported where deviations from this general pattern occurred.

5.2 OpenJDK JRE Version Effects

This section contributes to answering **RQ1** by evaluating whether OpenJDK JRE source version is associated with statistically significant differences in JRE JVM power use and energy consumption. Specifically, the analyses assess whether

observed differences between JRE versions are sufficiently systematic to indicate that JRE source version contributes meaningfully to variation in power use and energy consumption. In addition, all analyses are decomposed according to interpretive and tiered execution modes to examine the extent to which execution mode influences these observed relationships.

Principally, this section addresses sub-research question **SRQ1.1**, which examines the extent to which OpenJDK JRE source version influences JRE JVM power use and energy consumption:

> **SRQ1.1:** ***What effect does JRE source version have on JRE JVM power use and energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this section also contributes evidence toward addressing **SRQ1.6**, which examines how differences between interpretive and tiered execution influence JRE JVM power use and energy consumption across differing runtime conditions:

> **SRQ1.6:** ***How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions?***

In this section, we first examine the effect of OpenJDK JRE source version on power use, beginning with a descriptive analysis of the pooled and execution-mode-specific distributions, followed by inferential statistical analyses examining differences between JRE version medians. We then apply the same analytical structure to energy consumption, assessing whether JRE source version is associated with measurable differences in energy consumption across all observations and within interpretive and tiered execution modes. Finally, we examine the relative differences between interpretive and tiered execution by analysing power use and energy consumption ratios. The section concludes by summarising the

principal JRE version effects observed in relation to **SRQ1.1**, while also identifying the extent to which the observed execution mode differences contribute toward addressing **SRQ1.6**.

5.2.1 Power Use

This subsection principally addresses sub-research question **SRQ1.1a**, which examines the extent to which OpenJDK JRE source version influences observed JRE JVM power use:

> **SRQ1.1a:** ***What effect does JRE source version have on JRE JVM power use?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6a**, which examines how differences between interpretive and tiered execution influence observed JRE JVM power use across differing runtime conditions:

> **SRQ1.6a:** ***How do differences between interpretive and tiered execution affect JRE JVM power use across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the power use distributions associated with each OpenJDK JRE version. Second, we conduct inferential statistical analyses to evaluate whether the observed differences between JRE versions are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in power use.

Descriptive Analysis

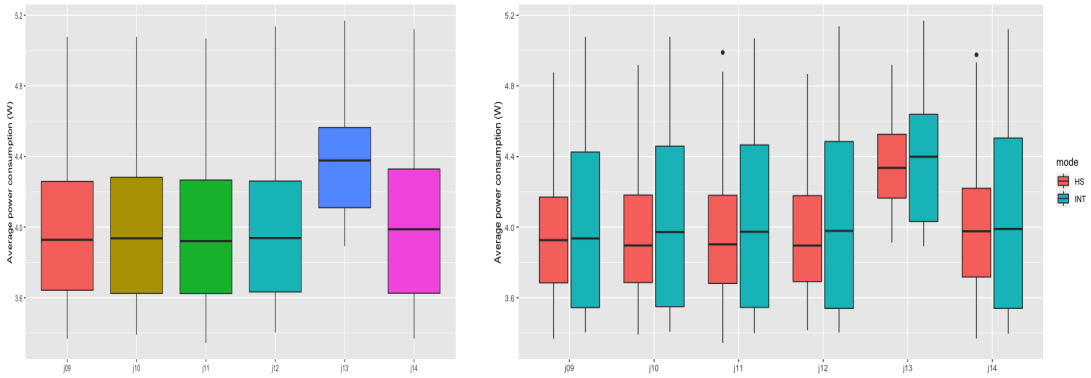
To provide an initial assessment of JRE version effects on power use, we begin by examining the distributional properties of the observed measurements. Particular attention is given to differences in location, variability, and distributional overlap between OpenJDK JRE versions. These characteristics are first examined for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes. Visual summaries of the distributions are presented using box-and-whisker plots, Figure 5.1, with detailed numerical summaries provided in Table 5.1.

In Figure 5.1 (a), we present box-and-whisker plots depicting the power use distributions associated with each OpenJDK JRE JVM version. We give a more detailed overview of each distribution's point estimates in Table 5.1. Across all OpenJDK JRE versions, average power use was recorded at $4.05W$, with average median power use recorded at $4.02W$. OpenJDK JRE version 13 was associated with the highest average and median power use, with associated mean and median values of $4.36W$ and $4.38W$, respectively.

Individual standard deviations were relatively large in comparison to the differences between distribution means, resulting in substantial overlap between the OpenJDK JRE version distributions. We observed that average power use differed by no more than approximately $0.39W$ between OpenJDK JRE versions.

In addition to exhibiting the highest average power use, OpenJDK JRE version 13 also displayed the lowest variability, with an associated standard deviation of $0.29W$ and an interquartile range of $0.45W$. Consequently, the elevated power use associated with OpenJDK JRE version 13 appears relatively consistent across workload executions.

Excluding OpenJDK JRE version 13, the remaining JRE versions demonstrated highly similar power use characteristics, with mean power use values ranging between $3.97W$ and $4.00W$.



(a) (b)

Figure 5.1: Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE power use distributions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b).

Table 5.1: Power Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use.

	Pooled Observations (W)							Execution Mode (W)							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	3.37	5.08	3.97	3.93	0.39	0.62	864	HS	3.37	4.88	3.94	3.93	0.32	0.49	480
								INT	3.40	5.08	4.01	3.94	0.47	0.88	384
j10	3.39	5.08	3.98	3.94	0.42	0.66	768	HS	3.39	4.92	3.94	3.90	0.32	0.50	384
								INT	3.41	5.08	4.02	3.97	0.49	0.91	384
j11	3.34	5.07	3.98	3.92	0.41	0.64	768	HS	3.34	4.99	3.94	3.90	0.32	0.50	384
								INT	3.40	5.07	4.02	3.97	0.48	0.92	384
j12	3.40	5.14	3.98	3.94	0.42	0.63	736	HS	3.42	4.87	3.94	3.90	0.30	0.49	368
								INT	3.40	5.14	4.03	3.98	0.50	0.95	368
j13	3.89	5.17	4.36	4.38	0.29	0.45	736	HS	3.91	4.92	4.35	4.34	0.21	0.36	368
								INT	3.89	5.17	4.38	4.40	0.35	0.61	368
j14	3.37	5.12	4.00	3.99	0.43	0.70	736	HS	3.37	4.98	3.97	3.98	0.33	0.50	368
								INT	3.40	5.12	4.03	3.99	0.51	0.97	368
Avg.	**3.46**	**5.11**	**4.05**	**4.02**	**0.39**	**0.62**	**768**	**HS Avg.**	**3.47**	**4.93**	**4.01**	**3.99**	**0.30**	**0.47**	**392**
						Total:	**4,608**	**INT Avg.**	**3.48**	**5.11**	**4.08**	**4.04**	**0.47**	**0.87**	**376**
														HS Total:	**2,352**
														INT Total:	**2,256**

The results from an analysis of power use differences between interpretive and tiered execution at the OpenJDK JRE version level are presented in Figure 5.1 (b) and Table 5.1. The horizontal axis in Figure 5.1 (b) lists each of the six OpenJDK JRE versions with clustering by execution mode.

For interpretive execution, we recorded the average median power use at $4.08W$. The absolute mean difference between OpenJDK JRE version power use magnitudes was approximately $0.14W$. However, this average difference is largely driven by the elevated power use associated with OpenJDK JRE version

13. The difference between OpenJDK JRE version 13 and all other versions was approximately $0.40W$. Excluding OpenJDK JRE version 13, the absolute mean difference between interpretive execution power use magnitudes was approximately $0.02W$, and no greater than $0.04W$.

For tiered execution, we recorded the average median power use at $4.01W$. We observed the largest difference between interpretive and tiered execution for OpenJDK JRE version 12 at $0.08W$. The absolute mean difference between OpenJDK JRE version power use magnitudes was approximately $0.13W$. Similar to interpretive execution, this average difference was largely attributable to the elevated power use associated with OpenJDK JRE version 13. The difference between OpenJDK JRE version 13 and all other versions was approximately $0.34W$.

Across all OpenJDK JRE versions, interpretive execution was marginally higher average power use than tiered execution. The differences were small, ranging between approximately $0.03W$ and $0.09W$, but the direction of the effect was consistent across all JRE versions.

While average power use differed only modestly between execution modes, interpretive execution exhibited considerably greater variability. Across OpenJDK JRE versions, interpretive execution was associated with larger standard deviations and interquartile ranges than tiered execution, indicating greater fluctuation in observed power use.

Although OpenJDK JRE version 13 exhibited the highest power use, it simultaneously displayed the lowest variability. This relationship was consistently observed across the pooled analysis and within both the interpretive and tiered execution mode analyses.

Inferential Statistical Analysis

The descriptive analyses identified apparent differences in power use between OpenJDK JRE versions and between execution modes. To determine whether

those observed differences are statistically significant, we next conduct a series of inferential statistical analyses. We first assess the distributional assumptions associated with the observed power use measurements. Given the resulting distributional characteristics, non-parametric statistical procedures are employed to evaluate differences between OpenJDK JRE versions, differences between interpretive and tiered execution, and the extent to which any observed effects are attributable to individual JRE versions.

OpenJDK JRE Version Effects:

Considering all pooled observations, a Kruskal-Wallis H test indicated statistically significant differences in median power use between OpenJDK JRE versions, $\chi^2 = 539.75$, $df = 5$, $p < .001$. Pairwise comparisons identified OpenJDK JRE version 13 as having significantly higher median power use than all other OpenJDK JRE versions (all $p < .001$). No other pairwise differences were observed. The average difference between OpenJDK JRE version 13 and the remaining OpenJDK JRE versions was approximately $0.435W$, with an associated standard deviation of $0.026W$. All remaining pairwise comparisons indicated no statistically significant differences in median power use (all $p > .05$).

Execution Mode Effects:

For each OpenJDK JRE version, independent-samples Mann-Whitney tests were conducted to assess differences between interpretive and tiered execution. The results indicated no statistically significant differences in median power use between execution modes for any OpenJDK JRE version (all $p > .05$).

To assess differences between OpenJDK JRE versions within interpretive execution, a Kruskal-Wallis H test was performed. The results indicated statistically significant differences in median power use, $\chi^2 = 163.44$, $df = 5$, $p < .001$. Post hoc Mann-Whitney tests showed that all statistically significant differences were attributable to comparisons involving OpenJDK JRE version 13 (all $p < .001$).

For tiered execution, a Kruskal-Wallis H test also indicated statistically signifi-

cant differences between OpenJDK JRE versions, $\chi^2 = 454.80$, $df = 5$, $p < .001$. As with interpretive execution, only pairwise comparisons involving OpenJDK JRE version 13 were statistically significant (all $p < .001$).

Although descriptive median estimates were consistently lower for tiered execution than for interpretive execution, we found no statistically significant evidence to suggest that median power use differed between execution modes at the OpenJDK JRE version level (all $p > .05$).

Sensitivity Analysis: Excluding OpenJDK JRE Version 13:

To determine whether the observed effects were exclusively attributable to OpenJDK JRE version 13, additional analyses were conducted with version 13 removed. Considering all pooled observations, no statistically significant differences remained between OpenJDK JRE versions, $\chi^2 = 1.426$, $df = 4$, $p = .840$. Similarly, no statistically significant differences were observed for tiered execution, $\chi^2 = 0.155$, $df = 2$, $p = .925$, or for interpretive execution, $\chi^2 = 0.005$, $df = 2$, $p = .998$.

Summary

In relation to **SRQ1.1a**, the findings indicate that JRE source version exerts a limited effect on JRE JVM power use. While statistically significant differences in power use were observed between OpenJDK JRE versions, the observed effect was primarily associated with a single JRE release, indicating that JRE source version does not constitute a major determinant of runtime power use within the experimental environment considered in this study.

The findings further indicate that variation in power use associated with JRE source version is comparatively small relative to the overall variation present within the observed power use measurements. The observed differences were principally attributable to OpenJDK JRE version 13, which consistently demonstrated higher power use relative to all other assessed JRE versions, while OpenJDK JRE versions 9, 10, 11, 12, and 14 remained broadly comparable in their power use

characteristics.

In relation to **SRQ1.6a**, the results indicate that execution mode exerts only a limited influence on observed JRE JVM power use across differing runtime conditions. Across all assessed JRE versions, interpretive execution was consistently associated with slightly higher power use than tiered execution. However, despite the consistency in direction, no statistically significant differences between execution modes were observed within any individual JRE version.

Research Question Findings

SRQ1.1a: OpenJDK JRE source version exerts a limited but statistically significant effect on JRE JVM power use, with the observed effect being principally attributable to the elevated power use associated with OpenJDK JRE version 13.

SRQ1.6a: Differences between interpretive and tiered execution do not constitute a statistically significant determinant of JRE JVM power use under the runtime conditions examined in this study.

5.2.2 Energy Consumption

This subsection principally addresses sub-research question **SRQ1.1b**, which examines the extent to which OpenJDK JRE source version influences observed JRE JVM energy consumption:

> **SRQ1.1b:** ***What effect does JRE source version have on JRE JVM energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6b**, which examines how differences between interpretive and tiered execution influence observed JRE JVM energy consumption across differing runtime conditions:

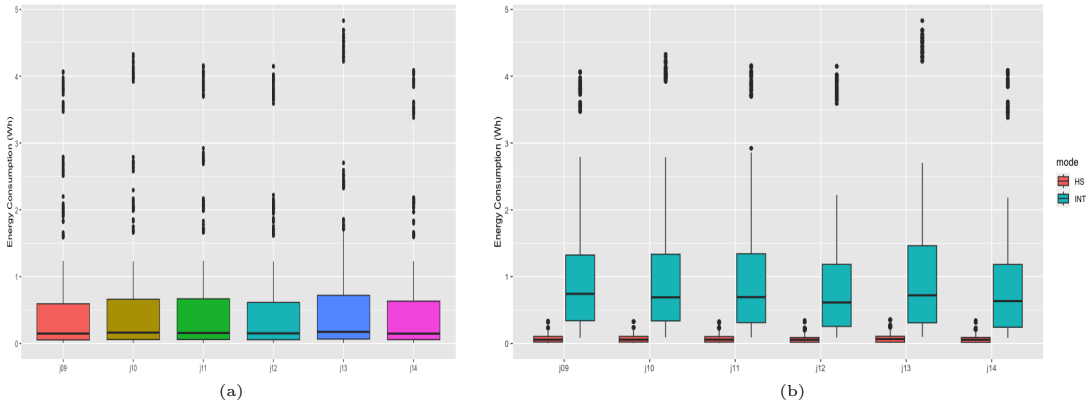
SRQ1.6b: ***How do differences between interpretive and tiered execution affect JRE JVM energy consumption across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the energy consumption distributions associated with each OpenJDK JRE version, with particular attention given to differences in location, variability, distributional shape, and the relationship between interpretive and tiered execution modes. Second, we conduct inferential statistical analyses to evaluate whether the observed differences in energy consumption between JRE versions are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in energy consumption.

Descriptive Analysis

To provide an initial assessment of JRE version effects on energy consumption, we begin by examining the distributional properties of the observed measurements. These characteristics are first considered for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes. Visual summaries of the distributions are presented using box-and-whisker plots, Figure 5.2, while detailed numerical summaries are provided in Table 5.2.

Focusing on pooled energy consumption observations for each OpenJDK JRE version, Figure 5.2 (a) and Table 5.2 present the corresponding energy consumption distributions and descriptive statistics. Median energy consumption across OpenJDK JRE versions ranged between $0.15Wh$ and $0.18Wh$. Across all pooled observations, average energy consumption was recorded at $0.56Wh$, with an average median energy consumption of $0.16Wh$. OpenJDK JRE version 13 recorded the highest mean and median energy consumption, with associated values of $0.62Wh$ and $0.18Wh$, respectively. However, the overall differences



(a) (b)

Figure 5.2: Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across all measurements pooled (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b).

between OpenJDK JRE versions were comparatively small, with median energy consumption differing by no more than approximately $0.03Wh$ between versions.

Substantial variability was evident within each OpenJDK JRE version distribution. Average standard deviations exceeded the corresponding median energy consumption values, while interquartile ranges were of a similar magnitude to the median estimates themselves. Consequently, considerable overlap was observed between the energy consumption distributions associated with each OpenJDK JRE version. In contrast to the power use distributions, no OpenJDK JRE version demonstrated clear separation from the remaining versions.

Inspection of Figure 5.2 (a) further indicates pronounced positive skewness across all distributions. In each case, median energy consumption values were substantially lower than their corresponding means, reflecting the presence of a relatively small number of workload executions associated with considerably larger energy consumption magnitudes. This pattern is also reflected in the extensive upper tails and outlying observations visible within the box-and-whisker plots.

Decomposing the pooled distributions according to execution mode, Figure 5.2 (b) and Table 5.2 reveal markedly different energy consumption characteristics for interpretive and tiered execution. Across all OpenJDK JRE versions, average median energy consumption for tiered execution was approximately $0.06Wh$, compared with approximately $0.69Wh$ for interpretive execution. Consequently,

interpretive execution was consistently associated with substantially greater energy consumption than tiered execution.

Table 5.2: Energy Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with energy consumption.

	Pooled Observations (Wh)							**Execution Mode (Wh)**							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	0.01	4.07	0.52	0.15	0.85	0.54	864	HS	0.01	0.34	0.08	0.06	0.07	0.08	480
								INT	3.40	5.08	1.06	0.74	1.04	0.98	384
j10	0.01	4.33	0.59	0.16	0.94	0.60	768	HS	0.01	0.33	0.08	0.06	0.07	0.08	384
								INT	3.41	5.08	1.10	0.69	1.12	1.00	384
j11	0.01	4.16	0.58	0.16	0.93	0.61	768	HS	0.01	0.33	0.08	0.06	0.07	0.08	384
								INT	3.40	5.07	1.09	0.70	1.09	1.03	384
j12	0.01	4.15	0.54	0.15	0.88	0.56	736	HS	0.01	0.34	0.07	0.06	0.07	0.07	368
								INT	3.40	5.14	1.01	0.62	1.04	0.93	368
j13	0.01	4.83	0.62	0.18	1.01	0.65	736	HS	0.01	0.36	0.08	0.07	0.07	0.09	368
								INT	3.89	5.17	1.16	0.72	1.21	1.15	368
j14	0.01	4.09	0.53	0.15	0.86	0.58	736	HS	0.01	0.34	0.07	0.06	0.06	0.07	368
								INT	3.40	5.12	1.00	0.64	1.02	0.94	368
Avg.	**0.01**	**4.27**	**0.56**	**0.16**	**0.91**	**0.59**	**768**	**HS Avg.**	**0.01**	**0.34**	**0.08**	**0.06**	**0.07**	**0.08**	**392**
						Total:	**4,608**	**INT Avg.**	**3.48**	**5.11**	**1.07**	**0.69**	**1.09**	**1.01**	**376**
													HS Total:		**2,352**
													INT Total:		**2,256**

In addition to exhibiting higher energy consumption, interpretive execution also displayed considerably greater variability. Across OpenJDK JRE versions, interpretive execution was associated with larger standard deviations and interquartile ranges than tiered execution, indicating greater variation in energy consumption across workload executions. Despite these differences between execution modes, the relative ordering of OpenJDK JRE versions remained broadly similar, with OpenJDK JRE version 13 generally exhibiting the highest energy consumption and OpenJDK JRE version 14 among the lowest.

In contrast to the power use distributions, where OpenJDK JRE version 13 exhibited clear separation from the remaining JRE versions, no OpenJDK JRE version showed comparable separation in terms of energy consumption. Across OpenJDK JRE versions, median energy consumption associated with interpretive execution was approximately eleven times greater than that associated with tiered execution.

Within both interpretive and tiered execution modes, OpenJDK JRE version 13 was associated with the highest median energy consumption. However, the

magnitude of these differences remained comparatively small relative to the variability observed within the distributions.

In regard to outliers, see Figure 5.2, the outlying energy consumption magnitudes are representative of seven workload applications, namely, `akka-uct`, `db-shootout`, `movie-lens`, `naive-bayes`, `neo4j-analytics`, `page-rank`, and `reactors`, none of which are from the `functional`, `scala`, or `web` application categories.

Inferential Statistical Analysis

OpenJDK JRE Version Effects:

Considering all pooled observations, a Kruskal–Wallis H test indicated that there was no evidence of statistically significant differences in median energy consumption between OpenJDK JRE versions at the conventional 5% significance level, $\chi^2 = 9.985$, $df = 5$, $p = .076$. Although this result approaches statistical significance at the less conservative 10% significance level, the observed effect remains comparatively weak. In contrast to the power use analysis, where OpenJDK JRE version 13 was associated with significantly elevated power use, no comparably strong JRE version effect was observed for energy consumption. Consequently, energy consumption appeared broadly stable across OpenJDK JRE version releases.

Execution Mode Effects:

When analysed separately by execution mode, there was no evidence of statistically significant differences in median energy consumption between OpenJDK JRE versions for tiered execution, $\chi^2 = 7.449$, $df = 5$, $p = .189$. Similarly, no statistically significant differences were observed between OpenJDK JRE versions for interpretive execution, $\chi^2 = 7.742$, $df = 5$, $p = .171$. These findings are consistent with the descriptive analyses, which indicated substantial overlap between the energy consumption distributions associated with each OpenJDK JRE version.

In contrast, comparisons between interpretive and tiered execution modes

revealed statistically significant differences in median energy consumption for every OpenJDK JRE version (all $p < .001$). In each case, tiered execution was associated with significantly lower energy consumption than interpretive execution.

The absence of JRE version effects, coupled with the consistently large differences observed between interpretive and tiered execution, suggests that execution mode exerts a substantially greater influence on energy consumption than JRE source version. While the superior execution time performance associated with tiered execution would lead one to anticipate lower energy consumption, the magnitude and consistency of the observed differences demonstrate that execution mode represents the principal contributor of energy consumption across all assessed OpenJDK JRE versions. A detailed summary of the energy consumption distributions associated with each execution mode is presented in Table 5.2.

Summary

In relation to **SRQ1.1b**, the findings indicate that JRE source version has no statistically significant effect on JRE JVM energy consumption. The observed differences suggest that energy consumption remains broadly stable across OpenJDK JRE versions, indicating that JRE source version does not constitute an important determinant of total runtime energy consumption within the experimental environment considered in this study.

The findings further indicate that any variation in energy consumption associated with JRE source version is comparatively small relative to the overall variation present within the observed energy consumption measurements. These results suggest that JRE source version contributes minimally to variation in total runtime energy consumption.

In relation to **SRQ1.6b**, the results indicate that execution mode represents the principal determinant of observed JRE JVM energy consumption across differing runtime conditions. Across all assessed OpenJDK JRE versions, interpretive execution was consistently associated with substantially higher energy consumption

than tiered execution, while the relative differences between JRE versions remained negligible irrespective of the JRE version employed.

Collectively, these findings indicate that OpenJDK JRE source version does not exert a statistically significant influence on JRE JVM energy consumption. In contrast, the substantial and consistent differences observed between interpretive and tiered execution demonstrate that execution mode constitutes a far more influential determinant of observed JRE JVM energy consumption under the runtime conditions examined in this study.

Research Question Findings

SRQ1.1b: OpenJDK JRE source version does not exert a consistent statistically significant effect on JRE JVM energy consumption. Although weak evidence of a possible JRE version effect was observed when considering all pooled observations, this effect was not sustained once analyses were decomposed according to execution mode, indicating that energy consumption remained broadly stable across OpenJDK JRE versions under the experimental conditions examined in this study.

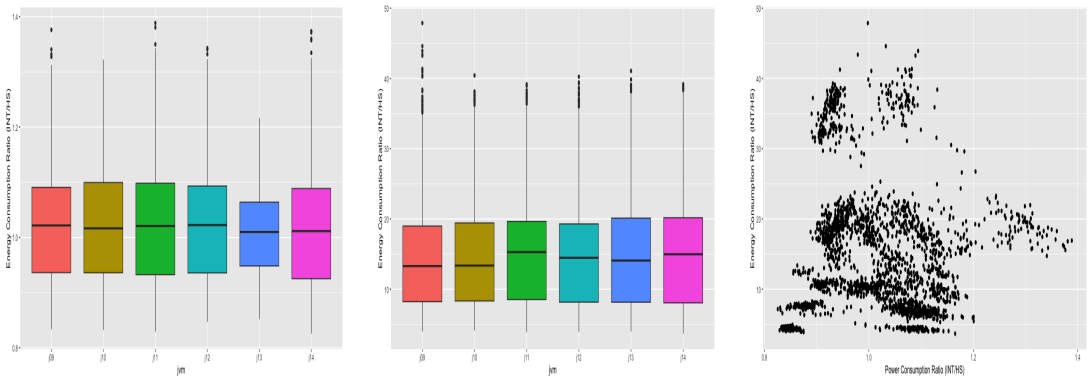
SRQ1.6b: Differences between interpretive and tiered execution constitute a major determinant of JRE JVM energy consumption, with interpretive execution consistently exhibiting substantially greater energy consumption than tiered execution across all assessed OpenJDK JRE versions.

5.2.3 Interpretive Execution Relative to Tiered Execution

This subsection examines the proportional differences between interpretive and tiered execution across OpenJDK JRE versions. Ratios were calculated for both power use and energy consumption, allowing the relative performance of the two execution modes to be compared independently of absolute measurement

magnitudes. We first examine power use ratios, followed by energy consumption ratios, before assessing the relationship between the two measures.

We present those differences through a series of box-and-whisker plots in Figure 5.3 (a). In addition, we present a summary of power use and energy consumption ratio distribution point estimates in Table 5.3.



(a) (b) (c)

Figure 5.3: Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting power use ratios for interpretive execution relative to tiered execution across all JRE JVM version (a), energy consumption ratios for interpretive execution relative to tiered execution (b), and a scatterplot illustrating the association between the corresponding power use and energy consumption ratios (c).

Table 5.3: Power and Energy Ratios: Descriptive statistics associated with the ratio of interpretive execution relative to tiered execution.

	Power Ratios							Energy Ratios						
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	0.83	1.38	1.02	1.02	0.10	0.15	384	3.98	47.88	15.56	13.30	9.43	10.77	384
j10	0.83	1.32	1.02	1.02	0.10	0.16	384	4.14	40.44	15.59	13.37	8.98	11.11	384
j11	0.83	1.39	1.02	1.02	0.11	0.17	384	3.89	39.16	15.73	15.29	9.09	11.12	384
j12	0.85	1.34	1.02	1.02	0.11	0.16	368	3.95	40.24	15.50	14.48	9.12	11.13	368
j13	0.85	1.22	1.01	1.01	0.08	0.12	368	3.98	41.07	15.78	14.08	9.34	11.97	368
j14	0.83	1.37	1.02	1.01	0.11	0.16	368	3.68	39.18	15.73	14.98	9.32	12.11	368
Avg.	**0.84**	**1.34**	**1.02**	**1.02**	**0.10**	**0.15**	**376**	**3.94**	**41.33**	**15.65**	**14.25**	**9.21**	**11.37**	**376**
						Total:	**2,256**						**Total:**	**2,256**

Power Use Ratios

Descriptive Analysis

The median power use ratio was approximately 1.02, indicating that the typical workload execution consumed slightly more power under interpretive execution than under tiered execution. However, the broad spread of the distributions around unity demonstrates that this relationship was not universal

across workloads. Ratios both above and below 1.00 were observed, indicating that the relative power use advantage of either execution mode varied according to workload category.

Inferential Statistical Analysis

Shapiro–Wilk tests indicated significant departures from normality for all power use ratio distributions (all $p < .001$). Visual inspection of the distributions indicated a moderate positive skew and the presence of several outlying observations, indicating that power use ratios were not constant across workloads. In fact, a significant degree of variation in those ratios is evident. A summary of the variation indicated that observed power use ratios ranged from approximately 0.83 to 1.39.

Assessing for differences between OpenJDK JRE versions, there was no evidence to indicate that power use ratios differed between versions, $\chi^2 = 5.326$, $df = 5$, $p = .377$. This finding suggests that the relative power use relationship between interpretive and tiered execution remained largely unchanged across OpenJDK JRE releases. Inspection of the ratio distributions indicates that the observed variation was primarily attributable to workload-specific effects rather than JRE version effects. Across all OpenJDK JRE versions, the `naive-bayes` benchmark was identified as the principal outlier. Furthermore, OpenJDK JRE version 13 has the narrowest ratio distribution, indicating less workload-dependent variation in power use ratios than the remaining OpenJDK JRE versions.

Energy Consumption Ratios

Descriptive Analysis

In regard to the relative difference between interpretive energy consumption and tiered execution energy consumption we present energy consumption ratio distributions for each OpenJDK JRE, in Figure 5.3 (b). In addition, we present detailed point-estimate summaries of the energy consumption ratios in Table 5.3.

Mean energy consumption ratios ranged from 15.50 to 15.78 across OpenJDK

JRE versions, while median ratios ranged from 13.37 to 15.29. These values indicate that interpretive execution typically consumed between approximately thirteen and fifteen times more energy than tiered execution. Considerable variability was evident within the ratio distributions, with standard deviations ranging from 8.98 to 9.43 and interquartile ranges ranging from 10.77 to 12.11.

Across all OpenJDK JRE versions, interpretive execution consumed substantially more energy than tiered execution. Observed energy consumption ratios ranged from approximately 3.89 to 47.88, indicating that interpretive execution consumed between approximately four and forty-eight times as much energy as tiered execution, depending on workload category.

Inferential Statistical Analysis

Shapiro–Wilk tests indicated significant departures from normality for all energy consumption ratio distributions (all $p < .001$). Visual inspection indicated positively skewed distributions and the presence of several outlying observations.

The outlying observations evident in Figure 5.3 (b) were associated primarily with the `philosophers` and `mnemonics` benchmarks. These workloads belong to the `scala` and `functional` workload categories, respectively, suggesting that variation in interpretive-to-tiered execution energy consumption ratios may be associated with workload category.

Assessing for differences between OpenJDK JRE versions, there was no evidence to indicate that energy consumption ratios differed between versions, $\chi^2 = 0.534$, $df = 5$, $p = .991$. This finding indicates that the relative energy consumption relationship between interpretive and tiered execution remained effectively unchanged across OpenJDK JRE releases. Consequently, although the magnitude of the energy consumption advantage associated with tiered execution varied according to workload category, that advantage was remarkably consistent across JRE versions.

Relationship Between Power and Energy Ratios

To assess whether the relative differences in power use between interpretive and tiered execution were associated with the corresponding differences in energy consumption, we examined the relationship between power use ratios and energy consumption ratios to determine if the large energy consumption differences between interpretive and tiered execution are a consequence of differences in power use. The results indicated no evidence of a meaningful relationship, $R^2 = .008$, $F(1, 2254) = 19.233$, $p < .001$. Although the relationship attained statistical significance owing to the large sample size, the model explained less than 1% of the observed variation in energy consumption ratios.

Visual inspection of the scatterplot presented in Figure 5.3 (c) supports this conclusion. No discernible linear relationship is evident, with substantial variation in energy consumption ratios observed across almost all levels of the power use ratio variable. Consequently, the large differences in energy consumption observed between interpretive and tiered execution cannot be attributed primarily to differences in power use.

Taken together, these findings suggest that the energy consumption advantage associated with tiered execution is driven principally by its shorter execution time rather than by reductions in power use.

Summary

The proportional comparison of interpretive and tiered execution demonstrates that execution mode exerted markedly different effects on power use and energy consumption. Although workload-dependent variation was evident, the relative power use relationship between interpretive and tiered execution remained stable across OpenJDK JRE versions. Furthermore, no statistically significant differences in power use ratios were observed between JRE versions, indicating that the relative power use effects associated with execution mode were largely independent of JRE release.

In contrast, interpretive execution was consistently associated with substantially greater energy consumption than tiered execution. While the magnitude of this advantage varied across workloads, the relative energy consumption relationship between interpretive and tiered execution remained effectively unchanged across OpenJDK JRE versions. Consequently, the energy consumption benefits associated with tiered execution appear to represent a persistent characteristic of the JVM execution model rather than a feature of any specific OpenJDK JRE release.

Finally, no meaningful relationship was observed between power use ratios and energy consumption ratios. This finding indicates that the substantial energy consumption advantage associated with tiered execution cannot be explained primarily by differences in power use. Instead, the results suggest that the observed energy consumption differences are driven principally by the superior execution time performance associated with tiered execution.

5.2.4 Summary

The results indicate that OpenJDK JRE source version had different effects on JRE JVM power use and energy consumption. In relation to power use, statistically significant differences were observed between OpenJDK JRE versions; however, these differences were attributable primarily to OpenJDK JRE version 13. When OpenJDK JRE version 13 was excluded, no statistically significant differences remained between the other assessed JRE versions. This indicates that JRE source version generally exerted only a limited influence on power use, with OpenJDK JRE version 13 representing a clear exception.

In relation to energy consumption, there was no evidence of a statistically significant JRE version effect at the conventional 5% significance level. Although weak evidence of a possible pooled JRE version effect was observed at the 10% level, this effect was not sustained when the data were decomposed according to interpretive and tiered execution modes. Consequently, JRE source version

did not appear to exert a consistent or meaningful effect on JRE JVM energy consumption.

The comparison of interpretive and tiered execution further demonstrated that execution mode had different implications for power use and energy consumption. Power use differences between interpretive and tiered execution were modest and remained broadly stable across OpenJDK JRE versions. In contrast, tiered execution was consistently associated with substantially lower energy consumption than interpretive execution. The ratio analysis further indicated that this energy consumption advantage was not primarily explained by differences in power use, but was instead principally associated with the superior execution time performance of tiered execution.

Taken together, these findings answer **SRQ1.1** by showing that OpenJDK JRE source version had a limited effect on JRE JVM power use and no consistent statistically significant effect on energy consumption. The principal exception was OpenJDK JRE version 13, which was associated with elevated power use. The findings also contribute to answering **SRQ1.6** by showing that differences between interpretive and tiered execution exerted only limited effects on power use, but constituted a major determinant of energy consumption across assessed OpenJDK JRE versions.

Section Research Question Findings

SRQ1.1: OpenJDK JRE source version exerts a limited effect on JRE JVM power use and no consistent statistically significant effect on JRE JVM energy consumption. The main exception is OpenJDK JRE version 13, which was associated with elevated power use relative to the remaining assessed JRE versions.

SRQ1.6: Differences between interpretive and tiered execution exert only limited effects on JRE JVM power use, but constitute a major determinant of JRE JVM energy consumption. Tiered execution consistently reduced energy consumption relative to interpretive execution, principally because of its shorter execution time rather than lower power use.

5.3 Build Toolchain Compiler Effects

This section contributes to answering **RQ1** by evaluating whether GNU GCC build toolchain compiler version is associated with statistically significant differences in JRE JVM power use and energy consumption. Specifically, the analyses assess whether observed differences between build toolchain compiler versions are sufficiently systematic to indicate that compiler version contributes meaningfully to variation in power use and energy consumption. In addition, all analyses are decomposed according to interpretive and tiered execution modes to examine the extent to which execution mode influences these observed relationships.

Principally, this section addresses sub-research question **SRQ1.2**, which examines the extent to which GNU GCC build toolchain compiler source version influences JRE JVM power use and energy consumption:

> **SRQ1.2:** ***What effect does GNU GCC build toolchain compiler source version have on JRE JVM power use and energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this section also contributes evidence toward addressing **SRQ1.6**, which examines how differences between interpretive and tiered execution influence JRE JVM power use and energy consumption across differing runtime conditions:

> **SRQ1.6:** ***How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions?***

In this section, we first examine the effect of GNU GCC build toolchain compiler source version on power use, beginning with a descriptive analysis of the pooled and execution-mode-specific distributions, followed by inferential statistical analyses examining differences between GNU GCC version medians. We then apply the same analytical structure to energy consumption, assessing whether GNU GCC build toolchain compiler source version is associated with measurable differences in energy consumption across all observations and within interpretive and tiered execution modes. Finally, we examine the relative differences between interpretive and tiered execution by analysing power use and energy consumption ratios. The section concludes by summarising the principal GNU GCC build toolchain compiler version effects observed in relation to **SRQ1.2**, while also identifying the extent to which the observed execution mode differences contribute toward addressing **SRQ1.6**.

5.3.1 Power Use

This subsection principally addresses sub-research question **SRQ1.2a**, which examines the extent to which GNU GCC build toolchain compiler source version influences observed JRE JVM power use:

> **SRQ1.2a:** ***What effect does GNU GCC build toolchain compiler source version have on JRE JVM power use?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6a**, which examines how differences between interpretive and tiered execution influence observed JRE JVM power use across differing runtime conditions:

> **SRQ1.6a:** ***How do differences between interpretive and tiered execution affect JRE JVM power use across differing runtime conditions?***

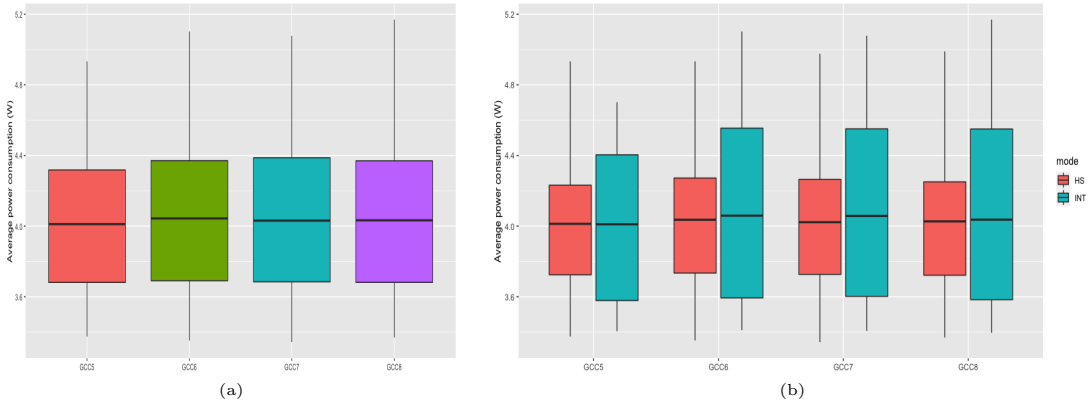
The analysis proceeds in two stages. First, we present a descriptive assessment of the power use distributions associated with each GNU GCC build toolchain compiler version. Second, we conduct inferential statistical analyses to evaluate whether the observed differences between GNU GCC versions are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in power use.

Descriptive Analysis

To provide an initial assessment of GNU GCC build toolchain compiler effects on power use, we begin by examining the distributional properties of the observed measurements. Particular attention is given to differences in location, variability, and distributional overlap between GNU GCC versions. These characteristics are first examined for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes. Visual summaries of the distributions are presented using box-and-whisker plots, Figure 5.4, with detailed numerical summaries provided in Table 5.4.

Considering all pooled observations, Figure 5.4 (a) and Table 5.4 indicate highly similar power use characteristics across GNU GCC versions. Average power use across compiler versions was approximately $4.04W$, while median power use

ranged between $4.01W$ and $4.04W$. GNU GCC version 5 exhibited the lowest average and median power use, with corresponding values of $4.00W$ and $4.01W$, respectively. GNU GCC versions 6 and 7 exhibited the highest average power use, both recording mean values of $4.06W$.



(a) (b)

Figure 5.4: Power Use Distributions: Box-and-whisker plots depicting JRE JVM power use distributions across GNU GCC versions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b).

The absolute difference between average power use magnitudes associated with GNU GCC versions was approximately $0.03W$. In contrast, standard deviations ranged between $0.38W$ and $0.44W$, indicating that the variation observed within each compiler version distribution was substantially greater than the differences observed between compiler versions. Consequently, considerable overlap was evident between the GNU GCC power use distributions.

Inspection of Figure 5.4 (a) further demonstrates that the distributions were broadly similar in shape and spread. No GNU GCC version exhibited the degree of separation previously observed for OpenJDK JRE version 13. Rather, the distributions formed a tightly clustered group, suggesting that compiler version is associated with only relatively small differences in JRE JVM power use

Within interpretive execution, average power use differed by approximately $0.04W$ between GNU GCC versions. GNU GCC version 5 was associated with the lowest average power use at $4.02W$, while GNU GCC versions 6, 7, and 8 exhibited highly similar average power use values, with a corresponding value

of $4.10W$. Although differences were small, GNU GCC version 5 consistently exhibited lower power use than the remaining compiler versions.

For tiered execution, differences between GNU GCC versions were even less pronounced. Average power use differed by approximately $0.01W$ across compiler versions, with mean values ranging between $3.99W$ and $4.02W$. The corresponding distributions exhibited substantial overlap, indicating little practical separation between compiler versions under tiered execution.

Table 5.4: Power Distributions: Descriptive statistics associated with each of the four GNU GCC versions, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use.

	Pooled Observations (W)							**Execution Mode (W)**							
JRE	**Min**	**Max**	**M**	$\mathbf{M}_d$	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	$\mathbf{M}_d$	**SD**	**IQR**	**n**
GCC5	3.37	4.93	4.00	4.01	0.38	0.64	1152	HS	3.37	4.93	3.99	4.01	0.34	0.51	588
								INT	3.41	4.70	4.02	4.01	0.41	0.83	564
GCC6	3.35	5.10	4.06	4.04	0.43	0.68	1152	HS	3.35	4.93	4.02	4.04	0.33	0.54	588
								INT	3.41	5.10	4.10	4.06	0.50	0.96	564
GCC7	3.34	5.08	4.06	4.03	0.43	0.70	1152	HS	3.34	4.98	4.01	4.02	0.34	0.54	588
								INT	3.41	5.08	4.10	4.06	0.50	0.95	564
GCC8	3.37	5.17	4.05	4.03	0.44	0.69	1152	HS	3.37	4.99	4.01	4.03	0.34	0.53	588
								INT	3.40	5.17	4.10	4.04	0.52	0.97	564
Avg.	**3.36**	**5.07**	**4.04**	**4.03**	**0.42**	**0.68**	**1152**	**HS Avg.**	**3.36**	**4.96**	**4.01**	**4.03**	**0.34**	**0.53**	**588**
						Total:	**4,608**	**INT Avg.**	**3.41**	**5.01**	**4.08**	**4.04**	**0.48**	**0.93**	**564**
														HS Total:	**2,352**
														INT Total:	**2,256**

The results from an analysis of power use differences between interpretive and tiered execution at the GNU GCC version level are presented in Figure 5.4 (b) and Table 5.4. Across all GNU GCC versions, interpretive execution exhibited higher average power use than tiered execution. For interpretive execution, average power use was approximately $4.08W$, compared with approximately $4.01W$ for tiered execution.

Across all GNU GCC versions, the difference between interpretive and tiered execution ranged from approximately $0.03W$ to $0.09W$. While interpretive execution consistently exhibited higher power use, the magnitude of these differences was comparatively small. However, interpretive execution was associated with greater variability, exhibiting larger standard deviations and interquartile ranges than tiered execution across all GNU GCC versions. This pattern indicates greater fluctuation in observed power use under interpretive execution than under tiered

execution.

Inferential Statistical Analysis

The descriptive analyses identified only modest differences in power use between GNU GCC versions. To determine whether those observed differences are statistically significant, we next conduct a series of inferential statistical analyses. We first assess the distributional assumptions associated with the observed power use measurements. Given the resulting distributional characteristics, non-parametric statistical procedures are employed to evaluate differences between GNU GCC versions, differences between interpretive and tiered execution, and the extent to which any observed effects are attributable to individual compiler versions.

GNU GCC Version Effects:

Considering all pooled observations, a Kruskal–Wallis H test indicated no statistically significant differences in median power use between GNU GCC versions at the 5% level of significance, $\chi^2 = 7.737$, $df = 3$, $p = .052$. However, the effect attained statistical significance at the 10% level. Post hoc comparisons indicated that GNU GCC version 5 was associated with lower median power use than GNU GCC versions 6, 7, and 8 (all $p < .10$). The magnitude of the effect was small, with average power use differing by approximately $0.02W$.

Execution Mode Effects:

To assess differences between GNU GCC versions within interpretive execution, a Kruskal–Wallis H test was performed. The results indicated statistically significant differences in median power use, $\chi^2 = 13.572$, $df = 3$, $p = .004$. Post hoc comparisons showed that GNU GCC version 5 differed significantly from GNU GCC versions 6, 7, and 8 (all $p < .05$). GNU GCC version 5 was associated with lower power use, exhibiting reductions of up to approximately 1.2% relative to the remaining compiler versions.

For tiered execution, a Kruskal–Wallis H test indicated no statistically signifi-

cant differences in median power use between GNU GCC versions, $\chi^2 = 2.010$, $df = 3$, $p = .570$.

Independent-samples Mann–Whitney tests were then conducted to assess differences between interpretive and tiered execution for each GNU GCC version. At the 5% level of significance, a statistically significant difference was observed only for GNU GCC version 7, $U = 152839$, $p = .022$. For GNU GCC version 7, tiered execution was associated with approximately $0.04W$ lower power use than interpretive execution.

At the 10% level of significance, additional evidence of execution-mode differences was observed for GNU GCC versions 6 and 8, $U = 154883$, $p = .053$, and $U = 154897$, $p = .053$, respectively. In both cases, tiered execution was associated with marginally lower power use than interpretive execution. No statistically significant differences between execution modes were observed for GNU GCC version 5 ($p > .10$).

Summary

In relation to **SRQ1.2a**, the findings indicate that GNU GCC build toolchain compiler source version exerts a modest effect on JRE JVM power use. While no statistically significant differences were observed between GNU GCC versions at the conventional 5% significance level when considering all pooled observations, weak evidence of a compiler version effect was observed at the 10% significance level. The observed differences were comparatively small in magnitude, indicating that compiler version does not constitute a major determinant of runtime power use within the experimental environment considered in this study.

The findings further indicate that variation in power use associated with GNU GCC compiler version was principally associated with GNU GCC version 5, which consistently exhibited lower power use relative to GNU GCC versions 6, 7, and 8. However, the absolute magnitude of these differences remained comparatively small, with observed reductions in power use typically amounting to only a few

hundredths of a watt.

In relation to **SRQ1.6a**, the results indicate that execution mode exerts only a limited influence on observed JRE JVM power use across differing runtime conditions. Across all GNU GCC versions, interpretive execution was generally associated with slightly higher power use and greater variability than tiered execution. However, these differences remained modest, with statistically significant execution mode differences observed only for GNU GCC version 7 at the conventional 5% significance level, with weaker evidence of comparable differences for GNU GCC versions 6 and 8 at the 10% significance level.

Collectively, these findings indicate that GNU GCC build toolchain compiler source version exerts only a limited influence on JRE JVM power use, with the principal effect being associated with the comparatively lower power use observed under GNU GCC version 5. In contrast, differences between interpretive and tiered execution do not constitute a strong or consistent determinant of JRE JVM power use under the runtime conditions examined in this study.

Research Question Findings

SRQ1.2a: GNU GCC build toolchain compiler source version exerts a limited effect on JRE JVM power use. Although only weak pooled evidence of compiler version differences was observed, GNU GCC version 5 consistently exhibited marginally lower power use relative to GNU GCC versions 6, 7, and 8.

SRQ1.6a: Differences between interpretive and tiered execution do not constitute a strong or consistent determinant of JRE JVM power use. Although interpretive execution generally exhibited slightly higher power use, execution mode effects remained comparatively small and were only observed under limited compiler-version conditions.

5.3.2 Energy Consumption

This subsection principally addresses sub-research question **SRQ1.2b**, which examines the extent to which GNU GCC build toolchain compiler source version influences observed JRE JVM energy consumption:

> **SRQ1.2b:** ***What effect does GNU GCC build toolchain compiler source version have on JRE JVM energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6b**, which examines how differences between interpretive and tiered execution influence observed JRE JVM energy consumption across differing runtime conditions:

> **SRQ1.6b:** ***How do differences between interpretive and tiered execution affect JRE JVM energy consumption across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the energy consumption distributions associated with each GNU GCC build toolchain compiler version, with particular attention given to differences in location, variability, distributional shape, and the relationship between interpretive and tiered execution modes. Second, we conduct inferential statistical analyses to evaluate whether the observed differences in energy consumption between GNU GCC versions are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in energy consumption.

Descriptive Analysis

To provide an initial assessment of GNU GCC build toolchain compiler effects on energy consumption, we begin by examining the distributional properties of

the observed measurements. Particular attention is given to differences in location, variability, and distributional overlap between GNU GCC versions. These characteristics are first examined for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes. Visual summaries of the distributions are presented using box-and-whisker plots, Figure 5.5, with detailed numerical summaries provided in Table 5.5.

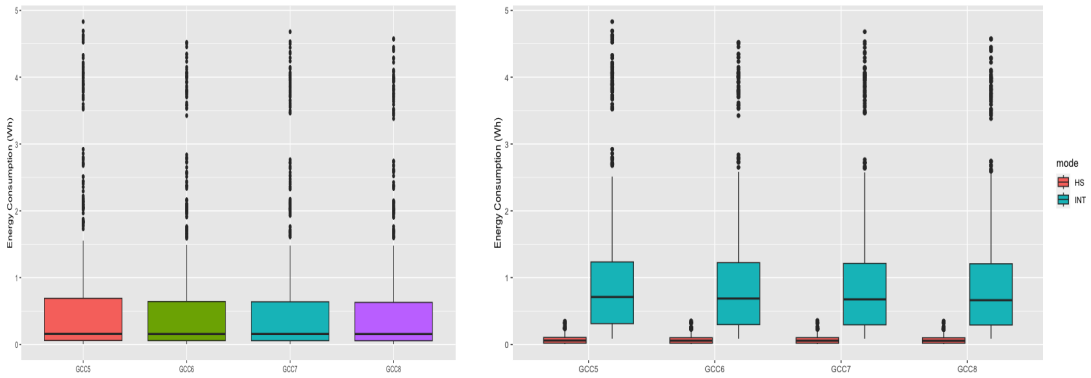


Figure 5.5: Energy Consumption Distributions: Box-and-whisker plots depicting JRE JVM energy consumption distributions across GNU GCC versions for all pooled measurements (a), and grouped according to tiered (HS) and interpretive (INT) execution modes (b).

Considering all pooled observations, Figure 5.5 (a) and Table 5.5 indicate highly similar energy consumption characteristics across GNU GCC versions. Average energy consumption was approximately $0.56Wh$ across all compiler versions, while median energy consumption remained constant at approximately $0.16Wh$. GNU GCC version 5 exhibited the highest average energy consumption at $0.57Wh$, while GNU GCC versions 6, 7, and 8 each exhibited average energy consumption of approximately $0.56Wh$.

The differences between GNU GCC versions were extremely small. Average energy consumption differed by no more than approximately $0.01Wh$ between compiler versions. In contrast, standard deviations were approximately $0.90Wh$ across all GNU GCC versions and substantially exceeded the observed differences between distribution means. Consequently, considerable overlap was evident between the GNU GCC energy consumption distributions.

Inspection of Figure 5.5 (a) further indicates that all GNU GCC energy consumption distributions exhibited pronounced positive skewness. In each case, median energy consumption values were substantially lower than their corresponding means, reflecting the presence of a relatively small number of workload executions associated with considerably greater energy consumption. This pattern is also evident in the upper tails and outlying observations visible within the box-and-whisker plots.

Within interpretive execution, average energy consumption differed only marginally between GNU GCC versions, ranging from approximately $1.06Wh$ to $1.09Wh$. GNU GCC version 5 exhibited the highest average energy consumption, while GNU GCC version 8 exhibited the lowest. However, the corresponding distributions exhibited substantial overlap, indicating little practical separation between compiler versions.

For tiered execution, differences between GNU GCC versions were even less pronounced. Average energy consumption ranged between approximately $0.07Wh$ and $0.08Wh$, while median energy consumption remained constant at approximately $0.06Wh$ across all GNU GCC versions. As with the pooled observations, the tiered execution distributions exhibited extensive overlap.

The results from an analysis of energy consumption differences between interpretive and tiered execution at the GNU GCC version level are presented in Figure 5.5 (b) and Table 5.5. Across all GNU GCC versions, interpretive execution exhibited substantially greater energy consumption than tiered execution. Average energy consumption for interpretive execution was approximately $1.07Wh$, compared with approximately $0.08Wh$ for tiered execution. Interpretive execution energy consumption is approximately 13 times greater than tiered execution.

Although energy consumption remained broadly stable across GNU GCC versions, substantial differences were evident between execution modes. Across all GNU GCC versions, interpretive execution consumed approximately thirteen times more energy than tiered execution on average. In addition, interpretive execution

Table 5.5: Energy Distributions: Descriptive statistics associated with each of the four GNU GCC versions, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with energy consumption.

	Pooled Observations (Wh)							Execution Mode (Wh)							
JRE	**Min**	**Max**	**M**	$\mathbf{M}_d$	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	$\mathbf{M}_d$	**SD**	**IQR**	**n**
GCC5	0.01	4.83	0.57	0.16	0.93	0.63	1152	HS	0.01	0.35	0.08	0.06	0.07	0.09	588
								INT	3.41	4.70	1.09	0.71	1.12	0.92	564
GCC6	0.01	4.52	0.56	0.16	0.91	0.59	1152	HS	0.01	0.35	0.08	0.06	0.07	0.08	588
								INT	3.41	5.10	1.07	0.69	1.09	0.93	564
GCC7	0.01	4.68	0.56	0.16	0.91	0.58	1152	HS	0.01	0.36	0.07	0.06	0.07	0.08	588
								INT	3.41	5.08	1.07	0.68	1.09	0.92	564
GCC8	0.01	4.58	0.56	0.16	0.90	0.57	1152	HS	0.01	0.35	0.08	0.06	0.07	0.08	588
								INT	3.40	5.17	1.06	0.66	1.08	0.92	564
Avg.	**0.01**	**4.65**	**0.56**	**0.16**	**0.91**	**0.59**	**1152**	**HS Avg.**	**0.01**	**0.35**	**0.08**	**0.06**	**0.07**	**0.08**	**588**
						Total:	**4,608**	**INT Avg.**	**3.41**	**5.01**	**1.07**	**0.69**	**1.10**	**0.92**	**564**
														HS Total:	**2,352**
														INT Total:	**2,256**

exhibited considerably greater variability, with larger standard deviations and interquartile ranges than tiered execution across all compiler versions. This pattern indicates greater fluctuation in energy consumption under interpretive execution than under tiered execution.

Taken together, the descriptive analyses suggest that GNU GCC build toolchain compiler version is associated with only relatively small differences in JRE JVM energy consumption. In contrast, the most pronounced differences in the energy consumption distributions were associated with execution mode rather than compiler version.

Inferential Statistical Analysis

The descriptive analyses indicated highly similar energy consumption characteristics across GNU GCC versions. To determine whether the observed differences are statistically significant, we next conduct a series of inferential statistical analyses. We first assess the distributional assumptions associated with the observed energy consumption measurements. Given the resulting distributional characteristics, non-parametric statistical procedures are employed to evaluate differences between GNU GCC versions and between interpretive and tiered execution modes.

GNU GCC Version Effects:

Considering all pooled observations, a Kruskal–Wallis H test indicated no evidence of statistically significant differences in median energy consumption between GNU GCC versions, $\chi^2 = 0.228$, $df = 3$, $p = .973$. These findings are consistent with the descriptive analyses, which indicated highly similar energy consumption characteristics and substantial overlap between the GNU GCC energy consumption distributions. Consequently, energy consumption appeared stable across the assessed GNU GCC compiler versions.

Execution Mode Effects:

Independent-samples Mann–Whitney tests were conducted to assess differences between interpretive and tiered execution for each GNU GCC version. The results indicated statistically significant differences in median energy consumption between execution modes for all GNU GCC versions (all $p < .001$). In each case, tiered execution was associated with substantially lower energy consumption than interpretive execution.

To assess differences between GNU GCC versions within interpretive execution, a Kruskal–Wallis H test was performed. The results indicated no evidence of statistically significant differences in median energy consumption between GNU GCC versions, $\chi^2 = 0.308$, $df = 3$, $p = .958$. Median energy consumption values were highly similar across compiler versions, indicating that compiler version exerted little influence on interpretive execution energy consumption.

For tiered execution, a Kruskal–Wallis H test likewise indicated no evidence of statistically significant differences in median energy consumption between GNU GCC versions, $\chi^2 = 0.491$, $df = 3$, $p = .921$. Similar to the interpretive execution results, energy consumption remained highly stable across compiler versions.

The absence of GNU GCC version effects, coupled with the consistently large differences observed between interpretive and tiered execution, suggests that execution mode exerts a substantially greater influence on JRE JVM energy consumption than build toolchain compiler version. While energy consumption remained effectively unchanged across GNU GCC releases, tiered execution was

consistently associated with markedly lower energy consumption than interpretive execution.

Summary

In relation to **SRQ1.2b**, the findings indicate that GNU GCC build toolchain compiler source version has no statistically significant effect on JRE JVM energy consumption. Across all assessed GNU GCC versions, energy consumption remained highly stable, with both descriptive and inferential analyses indicating negligible differences between compiler versions. These findings suggest that compiler source version does not constitute an important determinant of total runtime energy consumption within the experimental environment considered in this study.

The findings further indicate that any variation in energy consumption associated with GNU GCC compiler version is comparatively small relative to the overall variation present within the observed energy consumption measurements. Across all pooled observations, average energy consumption differed by no more than approximately $0.01Wh$ between compiler versions, while substantial overlap was observed across all corresponding distributions. These results suggest that GNU GCC compiler version contributes minimally to variation in total runtime energy consumption.

In relation to **SRQ1.6b**, the results indicate that execution mode represents the principal determinant of observed JRE JVM energy consumption across differing runtime conditions. Across all GNU GCC versions, interpretive execution was consistently associated with substantially higher energy consumption than tiered execution, with interpretive execution consuming approximately thirteen times more energy than tiered execution on average. These differences remained highly consistent irrespective of the compiler version employed.

Collectively, these findings indicate that GNU GCC build toolchain compiler source version does not exert a statistically significant influence on JRE JVM

energy consumption. In contrast, the substantial and consistent differences observed between interpretive and tiered execution demonstrate that execution mode constitutes a far more influential determinant of observed JRE JVM energy consumption under the runtime conditions examined in this study.

Research Question Findings

SRQ1.2b: GNU GCC build toolchain compiler source version does not exert a statistically significant effect on JRE JVM energy consumption, indicating that energy consumption remains broadly stable across GNU GCC compiler versions under the experimental conditions examined in this study.

SRQ1.6b: Differences between interpretive and tiered execution constitute a major determinant of JRE JVM energy consumption, with interpretive execution consistently exhibiting substantially greater energy consumption than tiered execution across all assessed GNU GCC compiler versions.

5.3.3 Interpretive execution relative to tiered execution

To assess the relative difference in power use between interpretive and tiered execution across GNU GCC versions, we examine the distribution of power use ratios, calculated as interpretive execution power use relative to tiered execution power use. Ratios greater than unity indicate higher power use under interpretive execution, whereas ratios less than unity indicate higher power use under tiered execution.

We present those differences through a series of box-and-whisker plots in Figure 5.6 (a) and (b), for power use and energy consumption, respectively. In addition, we present a summary of power use and energy consumption ratios distribution point estimates in Table 5.6.

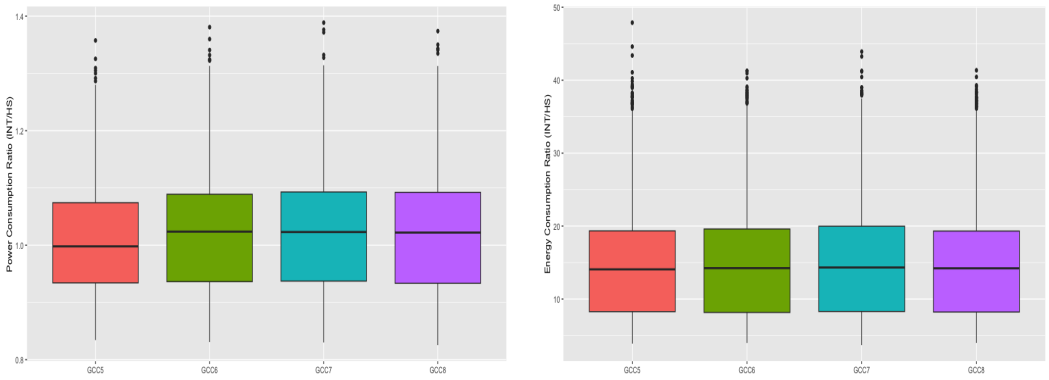


Figure 5.6: Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution across all GCC versions (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b).

Table 5.6: Power Energy Ratios: Descriptive statistics associated with each of the four GNU GCC versions, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution.

	Power Ratios							**Energy Ratios**						
GCC	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
GCC5	0.83	1.36	1.01	1.00	0.10	0.14	564	3.89	47.88	15.61	14.07	9.22	11.07	564
GCC6	0.83	1.38	1.02	1.02	0.10	0.15	564	3.98	41.28	15.75	14.23	9.30	11.44	564
GCC7	0.83	1.39	1.02	1.02	0.10	0.16	564	3.68	43.92	15.7	14.33	9.20	11.69	564
GCC8	0.83	1.37	1.02	1.02	0.11	0.16	564	3.98	41.35	15.54	14.21	9.13	11.10	564
Avg.	**0.83**	**1.38**	**1.02**	**1.02**	**0.10**	**0.15**	**564**	**3.88**	**43.61**	**15.65**	**14.21**	**9.21**	**11.37**	**564**
						Total:	**2,256**						**Total:**	**2,256**

Power Use Ratios

Descriptive Analysis

Inspection of Figure 5.6 (a) and Table 5.6 indicates that the relative power use relationship between interpretive and tiered execution remained broadly consistent across GNU GCC versions. Mean power use ratios ranged between approximately 1.01 and 1.02, while median ratios ranged between 1.00 and 1.02. Consequently, the typical workload execution exhibited only modest differences in power use between interpretive and tiered execution.

Despite the similarity in central tendency, considerable variation was evident within the ratio distributions. Power use ratios ranged from approximately 0.83 to 1.39, indicating that the relative power use performance of interpretive and tiered

execution varied substantially across workloads. Ratios below unity indicate lower power use under interpretive execution, whereas ratios above unity indicate lower power use under tiered execution.

Across GNU GCC versions, interpretive execution exhibited lower power use for approximately 54.3% of workload comparisons, with values ranging between approximately 49.0% and 56.0%.

Visual inspection of Figure 5.6 (a) indicates substantial overlap between the GNU GCC ratio distributions. GNU GCC version 5 exhibited the lowest mean and median power use ratios, suggesting a marginally smaller difference between interpretive and tiered execution than was observed for GNU GCC versions 6, 7, and 8.

Inferential Statistical Analysis

Power use ratio distributions departed significantly from normality (all $p < .001$) and exhibited substantial workload-dependent variation. The `naive-bayes` workload was identified as the principal outlying observation across the ratio distributions.

A Kruskal–Wallis H test indicated statistically significant differences in median power use ratios between GNU GCC versions, $\chi^2 = 9.119$, $df = 3$, $p = .028$. Post hoc comparisons indicated that GNU GCC version 5 was associated with significantly lower median power use ratios than GNU GCC versions 6, 7, and 8 (all $p < .05$). This finding suggests that GNU GCC version 5 was associated with slightly smaller interpretive-to-tiered execution power use ratios than GNU GCC versions 6, 7, and 8.

Energy Consumption Ratios

Descriptive Analysis

Inspection of the box-and-whisker plots indicates that the ratio between interpretive and tiered execution energy consumption was not constant across workload executions. All ratio distributions exhibited significant positive skewness

(all $p < .001$), indicating lower variation in energy consumption associated with tiered execution.

At the lower end of the observed range, interpretive execution consumed approximately 3.68 times more energy than tiered execution. At the upper end of the range, interpretive execution consumed approximately 47.88 times more energy than tiered execution. Consequently, while tiered execution consistently exhibited lower energy consumption, the magnitude of that advantage varied considerably across workloads.

Despite this variation, the central tendency of the distributions remained remarkably stable across GNU GCC versions. Mean energy consumption ratios ranged from 15.54 to 15.75, while median ratios ranged from 14.07 to 14.33, indicating that interpretive execution typically consumed approximately fourteen times more energy than tiered execution regardless of compiler version.

Inferential Statistical Analysis

Inspection of the box-and-whisker plots indicates that the ratio between interpretive and tiered execution energy consumption was not constant across workload executions. Shapiro–Wilk tests indicated significant departures from normality for all ratio distributions (all $p < .001$). Visual inspection indicated positively skewed distributions and substantial variation in the relative energy consumption advantage associated with tiered execution.

The outlying observations evident in Figure 5.6 (b) were associated primarily with the `philosophers` and `mnemonics` benchmarks. These workloads belong to the `scala` and `functional` workload categories, respectively, suggesting that variation in interpretive-to-tiered execution energy consumption ratios was more strongly associated with workload category than with compiler version.

Assessing for differences between GNU GCC versions, there was no evidence that energy consumption ratios differed between compiler versions, $\chi^2 = 0.324$, $df = 3$, $p = .956$. This finding indicates that the relative energy consumption relationship between interpretive and tiered execution remained effectively un-

changed across GNU GCC releases. Consequently, although the magnitude of the energy consumption advantage associated with tiered execution varied according to workload category, that advantage remained highly consistent across compiler versions.

Summary

The proportional comparison of interpretive and tiered execution demonstrates that execution mode exerted markedly different effects on power use and energy consumption across GNU GCC versions. Although workload-dependent variation was evident, the relative power use relationship between interpretive and tiered execution remained broadly consistent across compiler versions. GNU GCC version 5 exhibited marginally lower power use ratios than GNU GCC versions 6, 7, and 8, indicating a slightly smaller difference between execution modes. However, the magnitude of this effect was small, and power use ratios remained close to unity across all GNU GCC versions.

In contrast, interpretive execution was consistently associated with substantially greater energy consumption than tiered execution. While the magnitude of this advantage varied across workloads, the relative energy consumption relationship between interpretive and tiered execution remained highly stable across compiler versions. No statistically significant differences in energy consumption ratios were observed between GNU GCC versions, indicating that the energy consumption advantage associated with tiered execution was largely independent of compiler version.

Taken together, these findings suggest that GNU GCC build toolchain compiler version exerted only a limited influence on the relative relationship between interpretive and tiered execution. Although modest differences were observed in power use ratios, the dominant finding was the substantial and highly consistent energy consumption advantage associated with tiered execution across all assessed GNU GCC versions.

5.3.4 Summary

The results indicate that GNU GCC build toolchain compiler source version had a limited effect on JRE JVM power use and no statistically significant effect on JRE JVM energy consumption. In relation to power use, no statistically significant differences were observed between GNU GCC versions at the conventional 5% significance level when considering all pooled observations. However, weak evidence of a compiler-version effect was observed at the 10% significance level, with GNU GCC version 5 consistently exhibiting slightly lower power use than GNU GCC versions 6, 7, and 8. Within interpretive execution, this effect was more clearly observed, although the absolute magnitude of the differences remained comparatively small.

In relation to energy consumption, no statistically significant differences were observed between GNU GCC versions, either for pooled observations or when interpretive and tiered execution modes were analysed separately. Energy consumption distributions exhibited substantial overlap across compiler versions, indicating that changes in GNU GCC build toolchain compiler source version did not meaningfully alter JRE JVM energy consumption under the experimental conditions examined in this study.

The comparison of interpretive and tiered execution further demonstrated that execution mode had different implications for power use and energy consumption. Power use differences between interpretive and tiered execution were modest, with limited and inconsistent statistical evidence of execution-mode differences across GNU GCC versions. In contrast, tiered execution was consistently associated with substantially lower energy consumption than interpretive execution across all GNU GCC versions. The ratio analysis further indicated that this energy consumption advantage remained highly stable across compiler versions and was not meaningfully attributable to differences in power use.

Taken together, these findings answer **SRQ1.2** by showing that GNU GCC build toolchain compiler source version was associated with only limited differences

in JRE JVM power use and no measurable differences in JRE JVM energy consumption. The principal compiler-version effect was associated with GNU GCC version 5, which consistently exhibited slightly lower power use relative to later compiler versions. The findings also contribute to answering **SRQ1.6** by showing that differences between interpretive and tiered execution had only limited effects on power use, but represented a major determinant of energy consumption across all assessed GNU GCC versions.

Section Research Question Findings

SRQ1.2: GNU GCC build toolchain compiler source version was associated with only limited differences in JRE JVM power use and no statistically significant differences in JRE JVM energy consumption. The principal observed compiler-version effect was associated with GNU GCC version 5, which consistently exhibited slightly lower power use than GNU GCC versions 6, 7, and 8.

SRQ1.6: Differences between interpretive and tiered execution had only limited and inconsistent effects on JRE JVM power use, but represented a major determinant of JRE JVM energy consumption. Tiered execution consistently reduced energy consumption relative to interpretive execution across all assessed GNU GCC versions.

5.4 Garbage Collector Variant Effects

This section contributes to answering **RQ1** by evaluating whether garbage collector variant is associated with statistically significant differences in JRE JVM power use and energy consumption. Specifically, the analyses assess whether observed differences between garbage collector variants are sufficiently systematic to indicate that garbage collector selection contributes meaningfully to variation in power use and energy consumption. In addition, all analyses are decomposed according to interpretive and tiered execution modes to examine the extent to which execution

mode influences these observed relationships.

Principally, this section addresses sub-research question **SRQ1.3**, which examines the extent to which garbage collector variant influences JRE JVM power use and energy consumption:

> **SRQ1.3:** ***What effect does garbage collector variant have on JRE JVM power use and energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this section also contributes evidence toward addressing **SRQ1.6**, which examines how differences between interpretive and tiered execution influence JRE JVM power use and energy consumption across differing runtime conditions:

> **SRQ1.6:** ***How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions?***

Across the following subsections, five garbage collector variants are considered: the serial garbage collector (SGC), the parallel garbage collector (PGC), the concurrent mark and sweep collector (CGC), the garbage-first collector (G1GC), and the concurrent mark and sweep collector combined with parallel young-generation collection (CPGC).

In this section, we first examine the effect of garbage collector variant on power use, beginning with a descriptive analysis of the pooled and execution-mode-specific distributions, followed by inferential statistical analyses examining differences between garbage collector variant medians. We then apply the same analytical structure to energy consumption, assessing whether garbage collector variant is associated with measurable differences in energy consumption across all observations and within interpretive and tiered execution modes. Finally, we examine the relative differences between interpretive and tiered execution by analysing power use and energy consumption ratios. The section concludes by

summarising the principal garbage collector variant effects observed in relation to **SRQ1.3**, while also identifying the extent to which the observed execution mode differences contribute toward addressing **SRQ1.6**.

5.4.1 Power Use

This subsection principally addresses sub-research question **SRQ1.3a**, which examines the extent to which garbage collector variant influences observed JRE JVM power use:

> **SRQ1.3a:** ***What effect does garbage collector variant have on JRE JVM power use?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6a**, which examines how differences between interpretive and tiered execution influence observed JRE JVM power use across differing runtime conditions:

> **SRQ1.6a:** ***How do differences between interpretive and tiered execution affect JRE JVM power use across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the power use distributions associated with each garbage collector variant. Second, we conduct inferential statistical analyses to evaluate whether the observed differences between garbage collector variants are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in power use.

Descriptive Analysis

We begin by examining the descriptive characteristics of the power use distributions associated with each garbage collector variant. The objective of this analysis is to identify differences in central tendency, dispersion, and distributional shape that may indicate potential associations between garbage collector selection and JRE JVM power use. Descriptive statistics are presented in Figure 5.7(a) and Table 5.7.

Across all pooled observations, average power use was approximately $4.04W$. Mean power use varied only modestly between garbage collector variants, with an average absolute deviation of $0.05W$. The largest observed difference occurred between the G1GC and CPGC collectors, where mean power use differed by approximately $0.10W$.

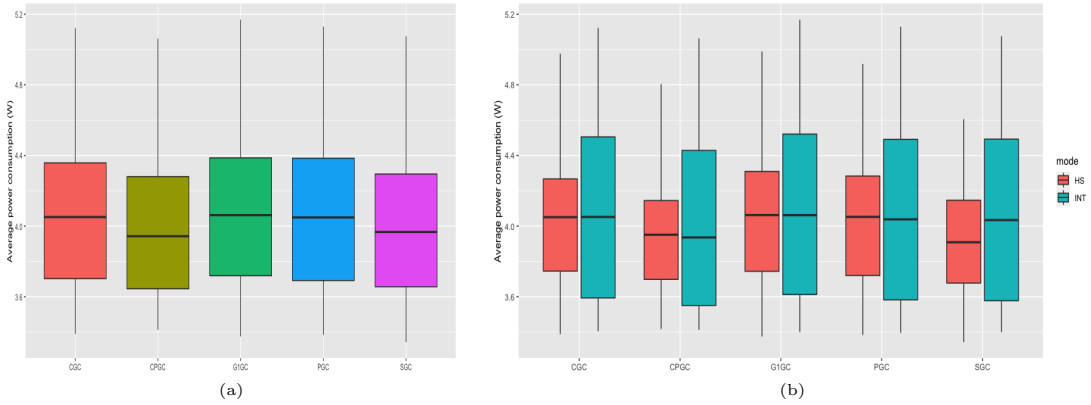


Figure 5.7: Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE average power use performance across garbage collector implementations, all measurements pooled (a), and clustered based on tiered (HS) and interpretive (INT) modes of execution (b).

Among the five garbage collector variants, the G1GC collector exhibited the greatest mean power use ($M = 4.08W$), followed by both the CGC and PGC collectors ($M = 4.06W$). Lower mean power use was observed for the SGC collector ($M = 4.00W$), while the lowest mean power use was associated with the CPGC collector ($M = 3.98W$). Median power use values closely mirrored the corresponding means, ranging from $3.94W$ for CPGC to $4.06W$ for G1GC, suggesting limited influence from extreme observations.

Table 5.7: Power Distributions: Descriptive statistics associated with each of the five garbage collector variants, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use.

	Pooled Observations (W)							Execution Mode (W)							
GC	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
CGC	3.39	5.12	4.06	4.05	0.42	0.66	1128	HS	3.39	4.98	4.03	4.05	0.33	0.52	564
								INT	3.4	5.12	4.08	4.05	0.49	0.91	564
CPGC	3.41	5.06	3.98	3.94	0.41	0.64	192	HS	3.42	4.8	3.96	3.95	0.32	0.45	96
								INT	3.41	5.06	4.01	3.94	0.48	0.88	96
G1GC	3.37	5.17	4.08	4.06	0.42	0.67	1032	HS	3.37	4.99	4.06	4.06	0.35	0.57	564
								INT	3.4	5.17	4.1	4.06	0.49	0.91	468
PGC	3.38	5.13	4.06	4.05	0.43	0.69	1128	HS	3.38	4.92	4.03	4.05	0.35	0.56	564
								INT	3.4	5.13	4.08	4.04	0.49	0.91	564
SGC	3.34	5.08	4.00	3.97	0.41	0.64	1128	HS	3.34	4.61	3.92	3.91	0.3	0.47	564
								INT	3.4	5.08	4.07	4.03	0.48	0.92	564
Avg.	**3.38**	**5.11**	**4.04**	**4.01**	**0.42**	**0.66**	**921**	**HS Avg.**	**3.38**	**4.86**	**4.00**	**4.00**	**0.33**	**0.51**	**470**
						Total:	**4,608**	**INT Avg.**	**3.40**	**5.11**	**4.07**	**4.02**	**0.49**	**0.91**	**451**
														HS Total:	**2,352**
														INT Total:	**2,256**

The distributions exhibited comparable levels of dispersion. Standard deviations ranged from $0.41W$ to $0.43W$, while interquartile ranges varied between $0.64W$ and $0.69W$. Visual inspection of Figure 5.7(a) indicates substantial overlap between the power use distributions associated with each garbage collector variant, suggesting that any differences in central tendency are relatively small in magnitude when compared with the overall variability observed within each distribution.

Inferential Statistical Analysis

To determine whether the differences in power use observed between garbage collector variants were sufficiently systematic to indicate an association between garbage collector selection and JRE JVM power use, we undertook a series of inferential statistical analyses. The analysis proceeded in three stages. First, we assessed the distributional assumptions associated with the power use observations. Second, we evaluated whether statistically significant differences existed between garbage collector variants using both pooled observations and observations decomposed by execution mode. Finally, we examined whether differences existed between interpretive and tiered execution within individual garbage collector variants.

Garbage Collector Variant Effects:

Considering all pooled observations, we observed statistically significant differences in median power use between garbage collector variants, $\chi^2(4) = 27.871$, $p < .001$. Post hoc comparisons indicated that the CPGC collector exhibited lower median power use than the CGC, G1GC, and PGC collectors (all $p < .05$). Differences were approximately $0.12W$. Similarly, the SGC collector exhibited lower median power use compared to the CGC, G1GC, and PGC collectors (all $p < .05$). Differences were approximately $0.08W$. No statistically significant differences were observed between the CGC, G1GC, and PGC collectors (all $p > .05$). Collectively, these results suggest that the pooled differences in power use were principally attributable to the comparatively lower power use associated with the CPGC and SGC collectors. The results are presented in Figure 5.7 (a) and Table 5.7.

Execution Mode Effects:

We next examined whether these relationships persisted when interpretive and tiered execution were considered separately.

For interpretive execution, no statistically significant differences in median power use were observed between garbage collector variants, $\chi^2 = 2.191$, $df = 4$, $p = .701$. Consequently, there was no evidence that garbage collector selection influenced power use during interpretive execution.

In contrast, statistically significant differences in median power use were observed between garbage collector variants during tiered execution, $\chi^2 = 53.107$, $df = 4$, $p < .001$. Post hoc comparisons were consistent with those observed for the pooled analysis. Specifically, the CPGC collector exhibited approximately $0.10W$ lower power use than the CGC, G1GC, and PGC collectors, while the SGC collector exhibited approximately $0.14W$ lower power use than the CGC, G1GC, and PGC collectors. These findings suggest that the differences observed between garbage collector variants in the pooled analysis were primarily attributable to variation arising during tiered execution.

Finally, to evaluate whether execution mode influenced power use within individual garbage collector variants, we conducted a series of Mann–Whitney tests comparing interpretive and tiered execution at each collector level. Statistically significant differences between execution modes were observed only for the SGC collector, $U = 133362$, $p < .001$. For the SGC collector, interpretive execution exhibited higher power use than tiered execution, with a median difference of approximately $0.12W$. No statistically significant differences between interpretive and tiered execution were observed for the CGC, CPGC, G1GC, or PGC collectors (all $p > .05$).

Summary

In relation to **SRQ1.3a**, the findings indicate that garbage collector variant had a statistically significant but comparatively modest association with JRE JVM power use. Statistically significant differences were observed between garbage collector variants when considering all pooled observations, indicating that garbage collector selection was associated with measurable variation in runtime power use. However, the absolute magnitude of these differences remained comparatively small relative to the overall variation observed within the dataset.

The findings further indicate that the observed differences in power use were not uniformly distributed across garbage collector variants. The CPGC and SGC collectors were consistently associated with comparatively lower power use relative to the CGC, G1GC, and PGC collectors. Although these differences were statistically significant, the magnitude of the observed reductions remained relatively modest, suggesting that garbage collector selection had only a limited association with overall runtime power use under the experimental conditions examined in this study.

In relation to **SRQ1.6a**, the results indicate that execution mode contributed only partially to the observed garbage collector effects on JRE JVM power use. The statistically significant pooled effects were primarily attributable to differ-

ences arising under tiered execution, while no statistically significant differences between garbage collector variants were observed during interpretive execution. Comparisons between interpretive and tiered execution within individual garbage collector variants were similarly limited, with statistically significant execution-mode differences observed only for the SGC collector.

Collectively, these findings indicate that garbage collector variant was associated with statistically significant differences in JRE JVM power use, although the overall magnitude of these differences remained modest. The principal observed effects were associated with the comparatively lower power use exhibited by the CPGC and SGC collectors. In contrast, differences between interpretive and tiered execution did not represent a strong or consistent determinant of JRE JVM power use across garbage collector variants under the runtime conditions examined in this study.

Research Question Findings

SRQ1.3a: Garbage collector variant was associated with statistically significant but comparatively modest differences in JRE JVM power use. The principal observed effects were associated with the CPGC and SGC collectors, both of which consistently exhibited lower power use relative to the CGC, G1GC, and PGC collectors.

SRQ1.6a: Differences between interpretive and tiered execution did not represent a strong or consistent determinant of JRE JVM power use across garbage collector variants. Although garbage collector differences were primarily observed under tiered execution, execution mode effects within individual garbage collector variants remained limited.

5.4.2 Energy Consumption

This subsection principally addresses sub-research question **SRQ1.3b**, which examines the extent to which garbage collector variant influences observed JRE JVM energy consumption:

> **SRQ1.3b:** ***What effect does garbage collector variant have on JRE JVM energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6b**, which examines how differences between interpretive and tiered execution influence observed JRE JVM energy consumption across differing runtime conditions:

> **SRQ1.6b:** ***How do differences between interpretive and tiered execution affect JRE JVM energy consumption across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the energy consumption distributions associated with each garbage collector variant, with particular attention given to differences in location, variability, distributional shape, and the relationship between interpretive and tiered execution modes. Second, we conduct inferential statistical analyses to evaluate whether the observed differences in energy consumption between garbage collector variants are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in energy consumption.

Descriptive Analysis

We begin by examining the descriptive characteristics of the energy consumption distributions associated with each garbage collector variant. The objective of this

analysis is to identify differences in central tendency, dispersion, and distributional shape that may indicate potential associations between garbage collector selection and JRE JVM energy consumption. Descriptive statistics are presented in Figure 5.8 and Table 5.8.

Across all pooled observations, average energy consumption was approximately $0.56Wh$, while the average median energy consumption was approximately $0.16Wh$. Mean energy consumption varied only modestly between garbage collector variants, ranging from $0.53Wh$ for G1GC to $0.58Wh$ for PGC. Median energy consumption was highly consistent across all garbage collector variants, varying between $0.15Wh$ and $0.16Wh$.

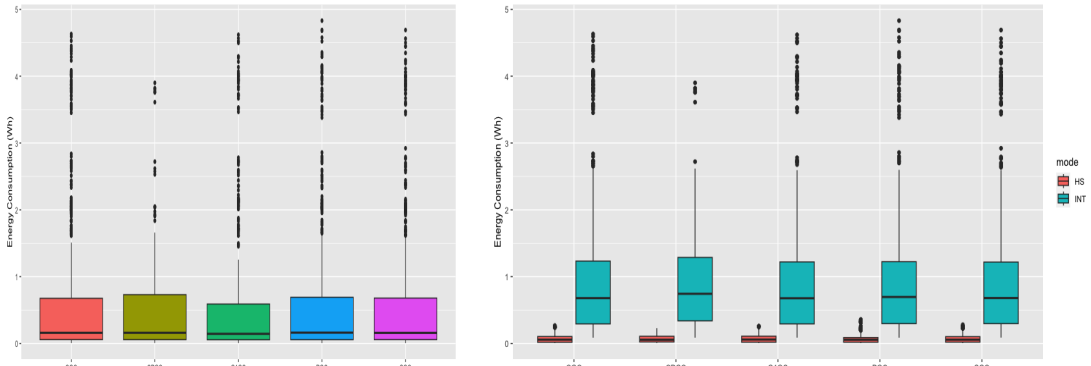


Figure 5.8: Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across garbage collector implementations, all measurements pooled (a), and clustered based on tiered (HS) and interpretive (INT) modes of execution (b).

Visual inspection of Figure 5.8(a) indicates substantial overlap between the energy consumption distributions associated with each garbage collector variant. Standard deviations ranged from $0.89Wh$ to $0.93Wh$, while interquartile ranges varied between $0.54Wh$ and $0.67Wh$, indicating comparable levels of dispersion across collector variants.

When execution modes were considered separately, substantially different patterns emerged. Across all garbage collector variants, average energy consumption during tiered execution was approximately $0.08Wh$, compared with approximately $1.07Wh$ during interpretive execution. Despite this large difference between ex-

ecution modes, energy consumption remained highly consistent across garbage collector variants within each execution mode. For tiered execution, median energy consumption was approximately $0.06Wh$ for all collector variants, while interpretive median energy consumption ranged from $0.68Wh$ to $0.74Wh$. This indicates that energy consumption under interpretive execution was substantially greater than under tiered execution, by approximately a factor of eleven to twelve across all garbage collector variants.

Table 5.8: Energy Distributions: Descriptive statistics associated with each of the five garbage collector variants, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with power use.

	Pooled Observations (Wh)							Execution Mode (Wh)							
GC	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
CGC	0.01	4.63	0.57	0.16	0.92	0.62	1128	HS	0.01	0.27	0.08	0.06	0.06	0.09	564
								INT	0.01	0.28	1.07	0.68	1.09	0.94	564
CPGC	0.01	3.90	0.57	0.16	0.89	0.67	192	HS	0.01	0.23	0.08	0.06	0.07	0.09	96
								INT	0.01	0.28	1.07	0.74	1.05	0.95	96
G1GC	0.01	4.62	0.53	0.15	0.90	0.54	1032	HS	0.01	0.26	0.08	0.06	0.06	0.09	564
								INT	0.01	0.28	1.08	0.68	1.10	0.93	468
PGC	0.01	4.83	0.58	0.16	0.93	0.64	1128	HS	0.01	0.36	0.08	0.06	0.08	0.07	564
								INT	0.01	0.28	1.07	0.70	1.10	0.93	564
SGC	0.01	4.69	0.57	0.16	0.91	0.62	1128	HS	0.01	0.28	0.07	0.06	0.06	0.08	564
								INT	0.01	0.28	1.07	0.68	1.08	0.92	564
Avg.	**0.01**	**4.53**	**0.56**	**0.16**	**0.91**	**0.62**	**921**	**HS Avg.**	**0.01**	**0.28**	**0.08**	**0.06**	**0.07**	**0.08**	**470**
						Total:	**4,608**	**INT Avg.**	**0.01**	**0.28**	**1.07**	**0.70**	**1.08**	**0.93**	**451**
														HS Total:	**2,352**
														INT Total:	**2,256**

Inferential Statistical Analysis

To determine whether the differences identified in the descriptive analysis were sufficiently systematic to indicate an association between garbage collector selection and JRE JVM energy consumption, we undertook a series of inferential statistical analyses. The analysis proceeded in three stages. First, we assessed the distributional assumptions associated with the energy consumption observations. Second, we evaluated whether statistically significant differences existed between garbage collector variants using both pooled observations and observations decomposed by execution mode. Finally, we examined whether differences existed between interpretive and tiered execution within individual garbage collector variants.

Garbage Collector Version Effects:

Considering all pooled observations, a Kruskal–Wallis H test indicated no evidence of statistically significant differences in median energy consumption between garbage collector variants, $\chi^2 = 3.177$, $df = 4$, $p = 0.529$. Consequently, energy consumption appeared stable across the assessed GNU GCC compiler versions.

Execution Mode Effects:

Focusing on interpretive execution, the results of a Kruskal–Wallis H test showed no statistically significant difference in median energy consumption, $\chi^2 = 0.0412$, $df = 4$, $p = 1.000$. A similar result was observed in regard to differences in energy consumption between garbage collector variants for tiered execution, $\chi^2 = 1.334$, $df = 4$, $p = 0.856$.

Independent-samples Mann–Whitney tests were conducted to assess differences in energy consumption between interpretive and tiered execution within each garbage collector variant. For all garbage collector variants, statistically significant differences in median energy consumption were observed between execution modes (all $p < .001$). In each case, tiered execution was associated with substantially lower energy consumption than interpretive execution. These findings were consistent with the descriptive analysis, which indicated that median energy consumption under interpretive execution was approximately eleven to twelve times greater than under tiered execution across all garbage collector variants.

Summary

In relation to **SRQ1.3b**, the findings indicate that garbage collector variant has no statistically significant effect on JRE JVM energy consumption. Across all assessed garbage collector variants, both descriptive and inferential analyses indicated highly similar energy consumption distributions, with no statistically significant differences observed when considering all pooled observations or when observations were decomposed according to execution mode. These findings indicate that

garbage collector selection does not represent an important determinant of total runtime energy consumption within the experimental environment considered in this study.

The findings further indicate that any variation in energy consumption associated with garbage collector selection was comparatively small relative to the overall variation present within the observed energy consumption measurements. Mean energy consumption differed only marginally between collector variants, median energy consumption remained highly consistent across all configurations, and substantial overlap was observed between all corresponding distributions. Collectively, these results indicate that garbage collector variant contributes minimally to variation in overall JRE JVM energy consumption.

In relation to **SRQ1.6b**, the results indicate that execution mode represents the principal determinant of observed JRE JVM energy consumption across differing runtime conditions. Across all garbage collector variants, interpretive execution was consistently associated with substantially greater energy consumption than tiered execution, with median energy consumption under interpretive execution being approximately eleven to twelve times greater than that observed under tiered execution. These differences remained highly consistent irrespective of the garbage collector variant employed.

Collectively, these findings indicate that garbage collector variant does not meaningfully influence observed JRE JVM energy consumption under the runtime conditions examined in this study. In contrast, the substantial and highly consistent differences observed between interpretive and tiered execution demonstrate that execution mode constitutes a far more influential determinant of JRE JVM energy consumption than garbage collector selection.

Research Question Findings

SRQ1.3b: Garbage collector variant does not have a statistically significant effect on JRE JVM energy consumption, indicating that energy consumption remains broadly stable across all assessed garbage collector variants under the experimental conditions examined in this study.

SRQ1.6b: Differences between interpretive and tiered execution constitute a major determinant of JRE JVM energy consumption, with interpretive execution consistently exhibiting substantially greater energy consumption than tiered execution across all assessed garbage collector variants.

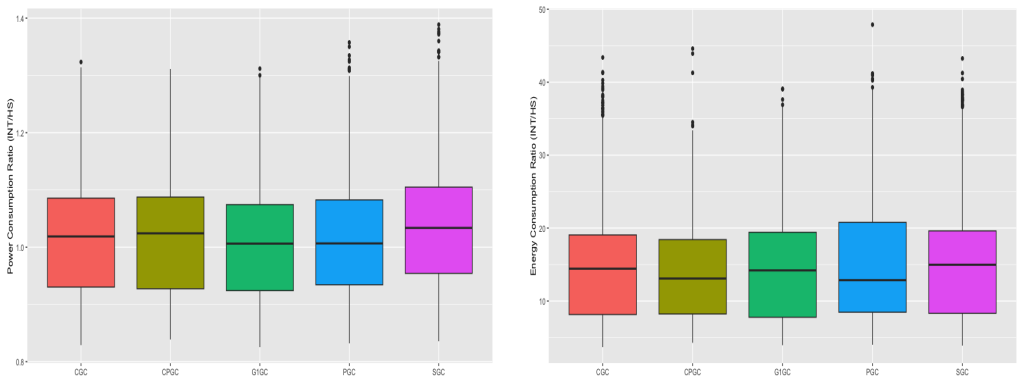
5.4.3 Interpretive Execution Relative to Tiered Execution

This subsection examines the differences between interpretive and tiered execution in terms of power use and energy consumption across garbage collector variants. To facilitate comparison, we analyse the ratio of interpretive execution to tiered execution for both measures. Ratio values greater than unity indicate that interpretive execution was associated with higher power use or energy consumption than tiered execution, whereas values less than unity indicate the opposite. The analysis proceeded in two stages. First, we present a descriptive summary of the power use and energy consumption ratios. Second, we undertake an inferential assessment to determine whether those ratios differ significantly between garbage collector variants.

The distributions of interpretive-to-tiered execution power use and energy consumption ratios are presented in Figure 5.9, while descriptive statistics are reported in Table 5.9.

Power Use Ratios

Descriptive Analysis



(a) (b)

Figure 5.9: Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution across all garbage collector variants (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b).

Table 5.9: Power Energy Ratios: Descriptive statistics associated with each garbage collector variant, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution.

	Power Ratios							**Energy Ratios**						
GC	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
CGC	0.83	1.32	1.01	1.02	0.10	0.16	564	3.68	43.38	15.48	14.43	9.00	10.91	564
CPGC	0.84	1.31	1.02	1.02	0.10	0.16	96	4.27	44.60	15.13	13.09	9.12	10.19	96
G1GC	0.83	1.31	1.01	1.01	0.10	0.15	468	3.95	39.10	15.32	14.20	8.75	11.64	468
PGC	0.83	1.36	1.01	1.01	0.10	0.15	564	4.00	47.88	16.07	12.86	9.84	12.32	564
SGC	0.84	1.39	1.04	1.03	0.10	0.15	564	3.89	43.25	15.75	14.97	9.14	11.29	564
Avg.	**0.83**	**1.34**	**1.02**	**1.02**	**0.10**	**0.15**	**451**	**3.96**	**43.64**	**15.55**	**13.91**	**9.17**	**11.27**	**451**
						Total:	**2,256**						**Total:**	**2,256**

For power use, ratio distributions were highly consistent across garbage collector variants. Mean and median ratios were approximately 1.02 across all collectors, indicating that interpretive execution generally consumed slightly more power than tiered execution. The SGC collector exhibited the largest median power use ratio ($M_d = 1.03$), while the G1GC and PGC collectors exhibited the smallest median ratios ($M_d = 1.01$). Dispersion was also highly comparable across collector variants, with standard deviations of approximately 0.10 and interquartile ranges between 0.15 and 0.16.

Inspection of the ratio ranges indicates that interpretive execution was not uniformly associated with higher power use. Across garbage collector variants, observed power use ratios ranged from approximately 0.83 to 1.39. Consequently,

while interpretive execution generally exhibited slightly greater power use than tiered execution, a subset of workload executions exhibited lower power use under interpretive execution, as reflected by ratio values below unity.

Inferential Statistical Analysis

Prior to evaluating differences in interpretive-to-tiered execution ratios between garbage collector variants, the ratio distributions were assessed for normality using the Shapiro–Wilk test. All distributions deviated significantly from normality (all $p < .001$). Inspection of the distributions identified only a small number of outlying observations. A limited number of outliers were associated with the CGC, CPGC, and G1GC collectors, while for both the PGC and SGC collectors the workload `naive-bayes` was identified as the principal outlying observation.

To determine whether interpretive-to-tiered execution power use ratios differed between garbage collector variants, a Kruskal–Wallis test was undertaken. The results indicated statistically significant differences in median power use ratios between garbage collector variants, $\chi^2(4) = 36.149$, $p < .001$. Post hoc comparisons revealed that the SGC collector was associated with significantly larger power use ratios than the CGC, PGC, and G1GC collectors (all $p < .05$). Consequently, the SGC collector exhibited slightly larger interpretive-to-tiered execution power use ratios than the CGC, PGC, and G1GC collectors. However, the practical magnitude of these differences was limited, with median ratios differing by no more than 0.02 across collector variants.

Energy Consumption Ratios

Descriptive Analysis

For energy consumption, ratio distributions were also highly consistent across garbage collector variants. Mean ratios ranged from 15.13 for CPGC to 16.07 for PGC, while median ratios ranged from 12.86 for PGC to 14.97 for SGC. These ratios indicate that interpretive execution typically consumed between approximately thirteen and fifteen times more energy than tiered execution,

irrespective of the garbage collector variant employed.

Although central tendencies were broadly comparable across collector variants, considerable variability was observed within the ratio distributions. Standard deviations ranged from 8.75 to 9.84, while interquartile ranges varied between 10.19 and 12.32.

Inspection of the ratio ranges indicates that interpretive execution was consistently associated with higher energy consumption than tiered execution. Across all garbage collector variants, observed energy consumption ratios ranged from approximately 3.68 to 47.88, with all ratio values exceeding unity. Consequently, all observed interpretive-to-tiered execution energy consumption ratios exceeded unity, indicating higher energy consumption under interpretive execution across all workload executions and garbage collector variants.

Inferential Statistical Analysis

Consistent with the power use ratio analysis, all energy consumption ratio distributions deviated significantly from normality (all $p < .001$). Consequently, non-parametric procedures were employed throughout.

In contrast, no statistically significant differences were observed between garbage collector variants in terms of their interpretive-to-tiered execution energy consumption ratios, $\chi^2(4) = 2.645$, $p = .619$. Although considerable variability was observed within the ratio distributions, the relative increase in energy consumption associated with interpretive execution was statistically comparable across garbage collector variants. Although median energy consumption ratios varied between 12.86 and 14.97, these differences were not statistically significant.

Summary

Analysis of the interpretive-to-tiered execution ratios indicated that garbage collector variant exerted only a limited influence on the relative differences between execution modes. For power use, statistically significant differences in interpretive-to-tiered execution ratios were observed between garbage collector variants, with

the SGC collector exhibiting the largest relative increase in power use under interpretive execution. However, the magnitude of these differences was small, with median power use ratios remaining close to unity across all garbage collector variants.

In contrast, no statistically significant differences were observed between garbage collector variants in terms of their interpretive-to-tiered execution energy consumption ratios. Across all garbage collector variants, interpretive execution consumed substantially more energy than tiered execution, with median energy consumption ratios ranging from approximately thirteen to fifteen. However, considerable variability was observed within the ratio distributions. Observed energy consumption ratios ranged from 3.68 to 47.88, indicating that while interpretive execution was consistently associated with higher energy consumption, the magnitude of the energy consumption penalty varied substantially across workload executions. Nevertheless, all observed energy consumption ratios exceeded unity, demonstrating that every workload execution consumed more energy under interpretive execution than under tiered execution, irrespective of the garbage collector variant employed.

Collectively, these findings suggest that although garbage collector variant may influence absolute levels of power use and energy consumption, it has comparatively little effect on the relative differences observed between interpretive and tiered execution modes. While the relative power use effects of interpretive execution varied across workloads and garbage collector variants, interpretive execution was consistently associated with greater energy consumption than tiered execution, although the magnitude of that increase varied substantially between workload executions.

5.4.4 Summary

This section evaluated the effect of garbage collector variant on JRE JVM power use and energy consumption. The analysis considered five garbage collector

variants: CGC, CPGC, G1GC, PGC, and SGC. Results were examined using pooled observations, observations decomposed by execution mode, and interpretive-to-tiered execution ratios.

In relation to power use, statistically significant differences were observed between garbage collector variants when observations were pooled across execution modes. These differences were primarily associated with the comparatively lower power use observed under the CPGC and SGC collectors relative to the CGC, G1GC, and PGC collectors. However, the magnitude of these differences was modest, and the corresponding distributions substantially overlapped. When execution modes were considered separately, garbage collector effects were observed during tiered execution but not during interpretive execution, indicating that the pooled power use differences were primarily associated with variation arising under tiered execution.

In relation to energy consumption, no statistically significant differences were observed between garbage collector variants. This was the case both for pooled observations and for observations decomposed by interpretive and tiered execution. Energy consumption distributions remained broadly stable across collector variants, indicating that garbage collector selection did not meaningfully alter total runtime energy consumption under the experimental conditions examined in this study.

The comparison of interpretive and tiered execution further demonstrated that execution mode had different implications for power use and energy consumption. Power use ratios were close to unity across all garbage collector variants, indicating that interpretive execution generally exhibited only slightly higher power use than tiered execution. However, the observed ratio range showed that interpretive execution was not uniformly associated with higher power use, as some workload executions exhibited lower power use under interpretive execution. In contrast, all energy consumption ratios exceeded unity, ranging from 3.68 to 47.88, indicating that interpretive execution always consumed more energy than tiered execution, although the magnitude of this increase varied substantially across workload

executions.

Taken together, these findings answer **SRQ1.3** by showing that garbage collector variant was associated with statistically significant but modest differences in JRE JVM power use, primarily under tiered execution, but was not associated with statistically significant differences in JRE JVM energy consumption. The findings also contribute to answering **SRQ1.6** by showing that differences between interpretive and tiered execution had only limited and inconsistent effects on power use, but represented a major determinant of energy consumption across all assessed garbage collector variants.

Section Research Question Findings

SRQ1.3: Garbage collector variant was associated with statistically significant but modest differences in JRE JVM power use, primarily under tiered execution, but was not associated with statistically significant differences in JRE JVM energy consumption. The principal observed power use differences were associated with the CPGC and SGC collectors, both of which exhibited lower power use relative to the CGC, G1GC, and PGC collectors.

SRQ1.6: Differences between interpretive and tiered execution had only limited and inconsistent effects on JRE JVM power use across garbage collector variants, but represented a major determinant of JRE JVM energy consumption. Interpretive execution consistently consumed substantially more energy than tiered execution across all assessed garbage collector variants.

5.5 Workload Effects

This section contributes to answering **RQ1** by evaluating whether workload category is associated with statistically significant differences in JRE JVM power use and energy consumption. Specifically, the analyses assess whether observed

differences between workload categories are sufficiently systematic to indicate that workload category contributes meaningfully to variation in power use and energy consumption. In addition, all analyses are decomposed according to interpretive and tiered execution modes to examine the extent to which execution mode influences these observed relationships.

Principally, this section addresses sub-research question **SRQ1.4**, which examines the extent to which workload category influences JRE JVM power use and energy consumption:

> **SRQ1.4:** ***What effect does workload category have on JRE JVM power use and energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this section also contributes evidence toward addressing **SRQ1.6**, which examines how differences between interpretive and tiered execution influence JRE JVM power use and energy consumption across differing runtime conditions:

> **SRQ1.6:** ***How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions?***

Across the following subsections, six workload categories are considered: `apache`, `concurrency`, `database`, `functional`, `scala`, and `web`.

In this section, we first examine the effect of workload category on power use, beginning with a descriptive analysis of the pooled and execution-mode-specific distributions, followed by inferential statistical analyses examining differences between workload-category medians. We then apply the same analytical structure to energy consumption, assessing whether workload category is associated with measurable differences in energy consumption across all observations and within interpretive and tiered execution modes. Finally, we examine the relative differences between interpretive and tiered execution by analysing power use and

energy consumption ratios. The section concludes by summarising the principal workload category effects observed in relation to **SRQ1.4**, while also identifying the extent to which the observed execution mode differences contribute toward addressing **SRQ1.6**.

5.5.1 Power Use

This subsection principally addresses sub-research question **SRQ1.4a**, which examines the extent to which workload category influences observed JRE JVM power use:

> **SRQ1.4a:** ***What effect does workload category have on JRE JVM power use?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6a**, which examines how differences between interpretive and tiered execution influence observed JRE JVM power use across differing runtime conditions:

> **SRQ1.6a:** ***How do differences between interpretive and tiered execution affect JRE JVM power use across differing runtime conditions?***

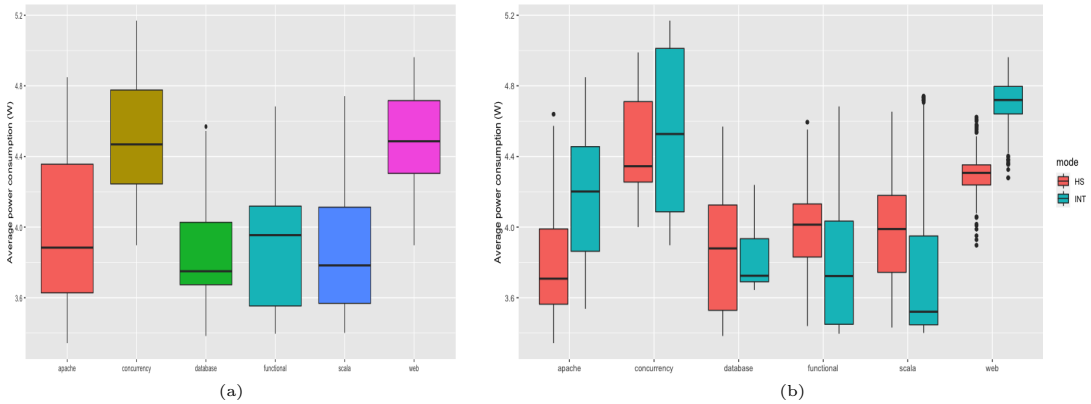
The analysis proceeds in two stages. First, we present a descriptive assessment of the power use distributions associated with each workload category, with particular attention given to differences in location, variability, distributional overlap, and the relationship between interpretive and tiered execution modes. Second, we conduct inferential statistical analyses to evaluate whether the observed differences in power use between workload category are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the

extent to which execution mode contributes to the observed variation in power use.

Descriptive Analysis

To provide an initial overview of the relationship between workload category and power use, we first examine the distributions of power use associated with each workload category before undertaking formal statistical tests of differences between categories.

In Figure 5.11 (a), we present the distributions of power use associated with each benchmark workload category, irrespective of execution mode. In Table 5.10, we present a more precise summary of the descriptive statistics associated with each workload category distribution.



(a) (b)

Figure 5.10: Power Use Distributions: Box-and-whisker plots depicting OpenJDK JRE average power use across workload categories, all measurements pooled (a), and grouped based on tiered (HS) and interpretive (INT) modes of execution (b).

Table 5.10: Power Distributions: Descriptive statistics associated with each of the six workload categories, reported for all pooled observations and grouped by execution mode, detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and sample size (**n**) associated with power use.

	Pooled Observations (W)							Execution Mode (W)							
Category	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
apache	3.37	5.12	3.98	3.88	0.38	0.73	1372	HS	3.34	4.64	3.79	3.71	0.29	0.43	700
								INT	3.54	4.85	4.18	4.20	0.37	0.59	672
concurrency	3.37	5.12	4.50	4.47	0.33	0.53	588	HS	4.00	4.99	4.44	4.35	0.26	0.46	300
								INT	3.90	5.17	4.55	4.53	0.39	0.93	288
database	3.37	5.12	3.84	3.75	0.26	0.35	296	HS	3.38	4.57	3.86	3.88	0.33	0.60	152
								INT	3.64	4.24	3.82	3.73	0.17	0.24	144
functional	3.37	5.12	3.90	3.95	0.32	0.57	980	HS	3.44	4.59	3.99	4.01	0.23	0.30	500
								INT	3.40	4.68	3.81	3.72	0.37	0.59	480
scala	3.37	5.12	3.88	3.78	0.37	0.55	980	HS	3.43	4.65	3.99	3.99	0.26	0.44	500
								INT	3.40	4.74	3.76	3.52	0.43	0.50	480
web	3.37	5.12	4.49	4.49	0.24	0.41	392	HS	3.90	4.62	4.30	4.31	0.14	0.11	200
								INT	4.28	4.96	4.69	4.72	0.14	0.17	192
Avg.	**3.37**	**5.12**	**4.10**	**4.05**	**0.32**	**0.52**	**768**	**HS Avg.**	**3.58**	**4.68**	**4.06**	**4.04**	**0.25**	**0.39**	**392**
						Total:	**4,608**	**INT Avg.**	**3.69**	**4.77**	**4.14**	**4.07**	**0.31**	**0.50**	**376**
														HS Total:	**2,352**
														INT Total:	**2,256**

From an analysis of the workload distributions presented in Figure 5.10 (a), we observe that power use differs systematically across workload categories. In particular, a clear distinction emerges between the `concurrency` and `web` workloads and the remaining four workload categories.

The descriptive statistics suggest the presence of two broad workload groups. The `concurrency` and `web` workloads exhibit the highest average power use, both averaging approximately $4.5W$. In contrast, the `apache`, `database`, `functional`, and `scala` workloads form a lower-power group, averaging approximately $3.9W$.

Across all workloads, we recorded the absolute average deviation in power use at $0.35W$. The largest difference in power use was observed between the `database` and `concurrent` workloads, which we recorded at $0.66W$. In general, the `concurrency` and `web` categories of workload exhibit similar average and median power use characteristics, with the remaining categories of workload having average power use magnitudes ranging within approximately $0.142W$ of each other.

Comparing interpretive execution to tiered execution at the workload category level, depicted in Figure 5.10 (b) and detailed in Table 5.10, differences in power use are evident. Across the six workload categories, the average absolute difference between interpretive and tiered execution was approximately $0.22W$. However,

the magnitude and direction of these differences varied according to workload category. The largest differences were observed for the `apache` and `web` workloads, where the average difference approached $0.4W$, whereas smaller differences were observed for the remaining workload categories.

Overall, the descriptive analyses suggest the presence of two broad workload groups characterised by relatively higher and lower power use, while indicating that the relative differences between interpretive and tiered execution vary according to workload category.

Inferential Statistical Analysis

The descriptive analyses indicated apparent differences in power use across workload categories, as well as differences between interpretive and tiered execution within individual workload classes. To determine whether these observed differences are statistically significant, we next conduct a series of inferential statistical analyses. We first assess the distributional characteristics of the observed power use measurements. Given the resulting distributional properties, non-parametric statistical procedures are then employed to evaluate differences between workload categories and differences between interpretive and tiered execution within each workload class.

Workload Category Effects:

Assessments for differences in population medians indicate statistically significant differences in power use magnitudes, $\chi^2 = 1393.7$, $Df = 5$, $p < .001$. All pairwise comparisons, except for the `concurrency` workloads compared to the `web` workloads and the `database` workloads compared to the `scala` workloads showed statistically significant differences in median power use magnitudes (all $p < .001$).

Execution Mode Effects:

Testing for differences in population medians between interpretive and tiered

execution power use, the results of Mann-Whitney U tests indicated statistically significant differences in medians for each workload category (all $p < .001$). For three workload categories, namely `database`, `functional`, and `scala`, interpretive execution was associated with lower median power use. In contrast, tiered execution was associated with lower median power use for the `apache`, `concurrency`, and `web` workload categories. We recorded the average absolute difference in power use at $0.22W$. The most significant difference was approximately $0.4W$ for the `apache` and `web` workload categories.

Summary

In relation to **SRQ1.4a**, the findings indicate that workload category has a statistically significant effect on JRE JVM power use. Statistically significant differences were observed across workload categories, indicating that workload category represents a major contributor to variation in runtime power use within the experimental environment considered in this study. Certain workload categories consistently imposed substantially greater power demands than others, demonstrating that the characteristics of the executing workload strongly influence observed power use.

The findings further indicate that workload-related differences in power use were not uniformly distributed across the benchmark suite, with two broad workload groupings emerging according to their relative power demands. The `concurrency` and `web` workload categories consistently exhibited substantially higher power use than the remaining workload categories, while the `apache`, `database`, `functional`, and `scala` workloads formed a comparatively lower-power group. These findings indicate that workload category represents an important source of variation in observed JRE JVM power use.

In relation to **SRQ1.6a**, the results indicate that the relative effects of interpretive and tiered execution on JRE JVM power use vary according to workload category. Neither interpretive nor tiered execution was consistently associated

with lower power use across all workload categories. Instead, both the magnitude and direction of execution mode differences varied according to workload category, indicating that execution mode effects on power use are workload dependent rather than globally consistent.

Collectively, these findings indicate that workload category is strongly associated with observed differences in JRE JVM power use and represents one of the most influential determinants of runtime power use identified within this study. In contrast to other experimental factors previously examined, workload category produced substantial and highly consistent differences in power use behaviour across the benchmark suite, while the relative effects of interpretive and tiered execution were shown to vary according to workload category.

Research Question Findings

SRQ1.4a: workload category has a statistically significant and substantial effect on JRE JVM power use. The strongest effects were associated with the `concurrency` and `web` workload categories, both of which consistently exhibited substantially higher power use relative to the remaining workload categories.

SRQ1.6a: Differences between interpretive and tiered execution do not exhibit a consistent global relationship with JRE JVM power use, with both the magnitude and direction of execution mode effects varying according to workload category.

5.5.2 Energy Consumption

This subsection principally addresses sub-research question **SRQ1.4b**, which examines the extent to which workload category influences observed JRE JVM energy consumption:

> **SRQ1.4b:** ***What effect does workload category have on JRE JVM energy consumption?***

In addition, because all analyses are decomposed according to interpretive and tiered execution modes, this subsection also contributes evidence toward addressing **SRQ1.6b**, which examines how differences between interpretive and tiered execution influence observed JRE JVM energy consumption across differing runtime conditions:

> **SRQ1.6b:** ***How do differences between interpretive and tiered execution affect JRE JVM energy consumption across differing runtime conditions?***

The analysis proceeds in two stages. First, we present a descriptive assessment of the energy consumption distributions associated with each workload category, with particular attention given to differences in location, variability, distributional overlap, and the relationship between interpretive and tiered execution modes. Second, we conduct inferential statistical analyses to evaluate whether the observed differences in energy consumption between workload category are statistically significant. In each case, results are presented for all pooled observations and subsequently decomposed according to interpretive and tiered execution modes to assess the extent to which execution mode contributes to the observed variation in energy consumption.

Descriptive Analysis

An examination of the workload energy consumption distributions presented in Figure 5.11 (a) indicates substantial differences in energy consumption across workload categories. In contrast to the power use results, the observed differences in energy consumption are considerably larger, suggesting that workload category exert a stronger influence on total energy consumption than on instantaneous power use.

The descriptive statistics presented in Table 5.11 indicate clear separation between workload categories. The `database` workloads exhibit the highest average energy consumption, averaging approximately $1.87Wh$, substantially exceeding

all remaining workload categories. The `apache` and `concurrency` workloads form a second grouping, each averaging approximately $0.8Wh$, while the `functional`, `scala`, and `web` workloads exhibit substantially lower average energy consumption, averaging between $0.20Wh$ and $0.22Wh$.

Decomposing the results according to execution mode, shown in Figure 5.11 (b) and detailed in Table 5.11, substantial differences emerge between interpretive and tiered execution. Across all workload categories, tiered execution is associated with substantially lower average energy consumption than interpretive execution. The largest differences are observed for the `database`, `apache`, and `concurrency` workloads, while comparatively smaller but still substantial differences are observed for the `functional`, `scala`, and `web` workload categories.

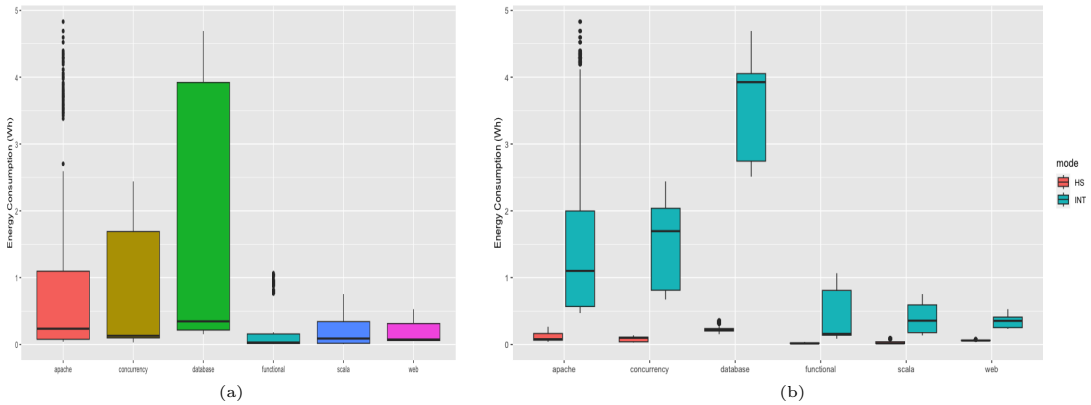


Figure 5.11: Energy Consumption Distributions: Box-and-whisker plots depicting OpenJDK JRE energy consumption across workload categories, all measurements pooled (a), and grouped based on tiered (HS) and interpretive (INT) modes of execution (b).

Table 5.11: Energy Distributions: Descriptive statistics associated with each of the six workload categories, reported for all pooled observations and grouped by execution mode, detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and sample size (**n**) associated with power use.

	Pooled Observations (Wh)							Execution Mode (Wh)							
Category	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
apache	0.04	4.83	0.79	0.24	1.03	1.02	1372	HS	0.04	0.27	0.11	0.08	0.06	0.10	700
								INT	3.54	4.85	1.49	1.10	1.10	1.43	672
concurrency	0.03	2.44	0.80	0.13	0.83	1.59	588	HS	0.03	0.14	0.08	0.10	0.03	0.07	300
								INT	3.90	5.17	1.54	1.70	0.56	1.23	288
database	0.15	4.69	1.87	0.35	1.76	3.70	296	HS	0.15	0.36	0.23	0.22	0.05	0.04	152
								INT	3.64	4.24	3.61	3.93	0.68	1.31	144
functional	0.01	1.07	0.22	0.03	0.33	0.14	980	HS	0.01	0.04	0.02	0.02	0.01	0.01	500
								INT	3.40	4.68	0.43	0.16	0.37	0.67	480
scala	0.01	0.75	0.21	0.09	0.23	0.32	980	HS	0.01	0.10	0.03	0.02	0.03	0.03	500
								INT	3.40	4.74	0.39	0.36	0.21	0.41	480
web	0.05	0.53	0.20	0.08	0.15	0.25	392	HS	0.05	0.08	0.06	0.06	0.01	0.01	200
								INT	4.28	4.96	0.34	0.36	0.08	0.16	192
Avg.	**0.05**	**2.39**	**0.68**	**0.15**	**0.72**	**1.17**	**768**	**HS Avg.**	**0.05**	**0.17**	**0.09**	**0.08**	**0.03**	**0.04**	**392**
						Total:	**4,608**	**INT Avg.**	**3.69**	**4.77**	**1.30**	**1.27**	**0.05**	**0.87**	**376**
														HS Total:	**2,352**
														INT Total:	**2,256**

Considering interpretive execution in isolation, substantial variation in energy consumption is evident across workload categories. Under interpretation, the `database` workloads remain clearly separated from all other workload categories, while the `apache` and `concurrency` workloads form a second higher-energy grouping, with the remaining workload categories exhibiting comparatively lower energy consumption.

In contrast, tiered execution exhibits a more compressed energy distribution across workload categories. Although the `database` workloads continue to exhibit the highest average energy consumption under tiered execution, the absolute differences between workload categories are considerably smaller, with the `functional` and `scala` workloads exhibiting the lowest average energy consumption.

Overall, the descriptive analyses suggest that workload category influences energy consumption to a greater extent than power use, while indicating that execution mode constitutes a substantial contributor to the observed energy consumption differences. Furthermore, although workload-related differences are evident under both execution modes, these differences become considerably more pronounced during interpretive execution.

Inferential Statistical Analysis

To determine whether the descriptive differences observed between workload categories reflect statistically meaningful differences in JRE JVM energy consumption, a series of inferential statistical analyses were conducted. The analyses evaluate whether workload category is associated with systematic variation in energy consumption across the six benchmark workload categories, while also examining the extent to which execution mode moderates those observed relationships.

The inferential analysis proceeds in two stages. First, the distributional characteristics of the observed energy consumption measurements are assessed to determine whether the assumptions required for parametric statistical procedures are satisfied. Finally, differences in energy consumption between workload categories are evaluated to determine whether workload category is associated with statistically significant variation in total runtime energy consumption.

Workload Category Effects:

Assessments of differences between workload categories indicate statistically significant differences in median energy consumption, $\chi^2 = 996.800$, $df = 5$, $p < .001$. Post hoc pairwise comparisons indicated statistically significant differences in median energy consumption between all workload category comparisons (all $p < .001$).

The results indicate clear separation between workload categories, with the `database`, `apache`, and `concurrency` workloads consistently associated with higher energy consumption relative to the `functional`, `scala`, and `web` workloads. These findings suggest that workload category exerts a substantial influence on total energy consumption.

Execution Mode Effects:

For each workload categories, statistically significant differences in median energy consumption were observed between interpretive and tiered execution (all $p < .001$). For each workload categories, tiered execution was consistently

associated with lower energy consumption than interpretive execution.

The magnitude of the observed differences between interpretive and tiered execution varied substantially across workload categories. For example, for the `database` workload category, median energy consumption under interpretive execution ($M_d = 3.93$) was approximately 18 times greater than under tiered execution ($M_d = 0.22$), whereas for the `web` workload category the corresponding difference was approximately 6 times greater.

In addition, the magnitudes of variation presented in Table 5.11 differ substantially between interpretive and tiered execution observations, both with respect to standard deviation (column SD) and interquartile range (column IQR). The comparatively smaller dispersion observed under tiered execution indicates that energy consumption remains more stable under tiered execution than under interpretive execution.

Focusing exclusively on interpretive execution, statistically significant differences in median energy consumption were again observed across workload categories, $\chi^2 = 605.1$, $df = 5$, $p < .001$. The results indicate substantial variation in energy consumption across workload categories, with the `database` workload category consistently associated with the highest median energy consumption, while the `functional`, `scala`, and `web` workload categories were associated with comparatively lower median energy consumption.

A comparable pattern was observed under tiered execution, where statistically significant differences in median energy consumption were again observed across workload categories (all $p < .001$). Although workload-related differences remained evident under tiered execution, the absolute differences between workload categories were considerably smaller than those observed under interpretive execution.

Summary

In relation to **SRQ1.4b**, the findings indicate that workload category had a statistically significant effect on JRE JVM energy consumption. Statistically significant differences were observed between all workload categories, demonstrating that workload category was associated with substantial variation in total runtime energy consumption. These results indicate that workload category represents one of the most influential determinants of energy consumption within the experimental environment examined in this study.

The findings further indicate that workload-related differences in energy consumption were considerably larger than those observed for power use. Clear separation was observed between workload categories, with the `database` workloads consistently exhibiting the greatest energy consumption, followed by the `apache` and `concurrency` workloads, while the `functional`, `scala`, and `web` workloads were associated with substantially lower energy consumption. Collectively, these results demonstrate that workload category were strongly associated with variation in total runtime energy consumption.

In relation to **SRQ1.6b**, the results indicate that execution mode constituted a major contributor to the observed workload-related differences in JRE JVM energy consumption. Across all workload categories, tiered execution was consistently associated with substantially lower energy consumption than interpretive execution, with statistically significant differences observed for every workload category. However, the magnitude of these differences varied considerably according to workload class. In particular, workload-related differences became substantially more pronounced under interpretive execution, where variation between workload categories was considerably amplified relative to tiered execution.

Collectively, these findings indicate that workload category had a substantial and statistically significant association with JRE JVM energy consumption, while execution mode further amplified those observed differences. Although workload category was an important determinant of overall energy consumption,

the substantial and highly consistent differences observed between interpretive and tiered execution demonstrate that execution mode also represented a major contributor to variation in JRE JVM energy consumption under the runtime conditions examined in this study.

Research Question Findings

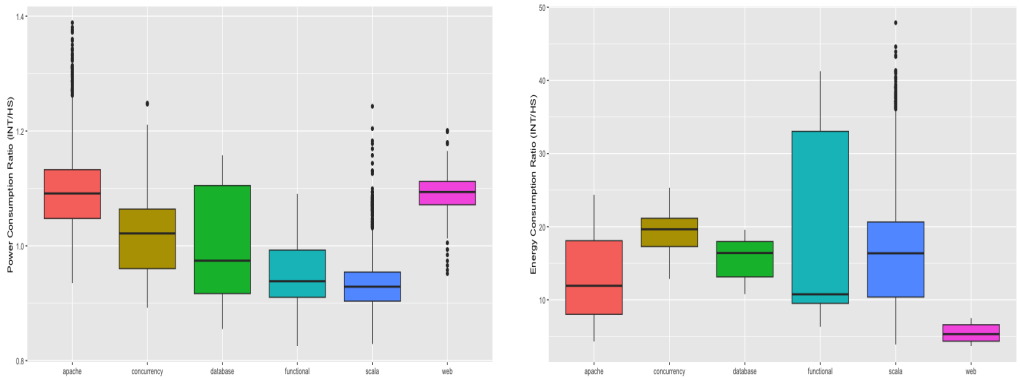
SRQ1.4b: workload category was associated with substantial and statistically significant differences in JRE JVM energy consumption, with workload category representing one of the strongest determinants of overall runtime energy consumption observed in this study.

SRQ1.6b: Differences between interpretive and tiered execution constituted a major determinant of JRE JVM energy consumption across all workload categories, with tiered execution consistently exhibiting substantially lower energy consumption and workload-related differences becoming considerably more pronounced under interpretive execution.

5.5.3 Interpretive Execution Relative to Tiered Execution

This subsection examines the differences between interpretive and tiered execution in terms of power use and energy consumption across benchmark workload categories. To facilitate comparison, we analyse the ratio of interpretive execution to tiered execution for both measures. Ratio values greater than unity indicate that interpretive execution was associated with higher power use or energy consumption than tiered execution, whereas values less than unity indicate the opposite. The analysis proceeded in two stages. First, we present a descriptive summary of the power use and energy consumption ratios. Second, we undertake an inferential assessment to determine whether those ratios differ significantly between workload categories.

The distributions of interpretive-to-tiered execution power use and energy consumption ratios are presented in Figure 5.12, while descriptive statistics are reported in Table 5.12.



(a) (b)

Figure 5.12: Interpretive-to-Tiered Execution Ratios: Box-and-whisker plots depicting the ratio of power use for interpretive execution relative to tiered execution (a), the ratio of energy consumption for interpretive execution relative to tiered execution (b).

Table 5.12: Power Energy Ratios: Descriptive statistics associated with each of the six workload categories, detailing the mean (**M**), the median ($\mathbf{M}_d$), the standard deviation (**SD**), and the interquartile range (**IQR**) associated with the ratio of interpretive power use relative to tiered execution, and the ratio of interpretive energy consumption relative to tiered execution.

	Power Use Ratios							Energy Consumption Ratios						
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
apache	0.94	1.39	1.103	1.091	0.086	0.085	672	4.31	24.35	13.06	11.91	5.21	10.07	672
concurrency	0.89	1.25	1.025	1.022	0.072	0.104	288	12.86	25.32	19.08	19.64	2.77	3.87	288
database	0.86	1.16	0.997	0.974	0.094	0.188	144	10.78	19.57	15.8	16.4	2.48	4.84	144
functional	0.83	1.09	0.952	0.938	0.057	0.082	480	6.32	41.25	19.46	10.75	12.47	23.52	480
scala	0.83	1.24	0.942	0.929	0.074	0.051	480	3.89	47.88	17.42	16.36	11.07	10.26	480
web	0.95	1.20	1.091	1.094	0.040	0.041	192	3.68	7.49	5.48	5.31	1.16	2.24	192
Avg.	**0.88**	**1.22**	**1.02**	**1.01**	**0.07**	**0.09**	**376**	**6.97**	**27.64**	**15.05**	**13.40**	**5.86**	**9.13**	**376**
						Total:	**2,256**						**Total:**	**2,256**

Inferential Statistical Analysis

Prior to evaluating differences in interpretive-to-tiered execution ratios between workload categories, the ratio distributions were assessed for normality using the Shapiro–Wilk test. All distributions deviated significantly from normality (all $p < .001$). Inspection of the distributions was undertaken to identify potential outlying observations.

To determine whether interpretive-to-tiered execution power use ratios differed between workload categories, a Kruskal–Wallis test was undertaken. Where statistically significant differences were observed, post hoc pairwise comparisons were conducted to identify the workload categories associated with those differences.

Descriptive Analysis

For power use, interpretive-to-tiered execution ratios varied more substantially across workload categories than was previously observed for JRE source version, GNU GCC compiler version, or garbage collector variant. Mean power use ratios ranged from 0.942 for the `scala` workload category to 1.103 for the `apache` workload category, while median ratios ranged from 0.929 for `scala` to 1.094 for `web`. These results indicate that workload category exerts a comparatively greater influence on the relative power use differences observed between interpretive and tiered execution.

The observed ratio values indicate that interpretive execution was not uniformly associated with higher power use. The `apache`, `concurrency`, and `web` workload categories exhibited median ratio values exceeding unity, indicating slightly higher power use under interpretive execution relative to tiered execution. In contrast, the `database`, `functional`, and `scala` workload categories exhibited median ratio values below unity, indicating lower power use under interpretive execution for those workload categories.

Differences in variability were also evident across workload categories. The `database` workload category exhibited the greatest dispersion, with an interquartile range of 0.188, while the `web` workload category exhibited the smallest variability, with an interquartile range of only 0.041. Across all workload categories, observed ratio values ranged from approximately 0.83 to 1.39, indicating substantial heterogeneity in the relative power use differences observed between interpretive and tiered execution.

Inferential Statistical Analysis

Prior to evaluating differences in interpretive-to-tiered execution power use ratios between workload categories, the ratio distributions were assessed for normality using the Shapiro–Wilk test. All workload category distributions deviated significantly from normality (all $p < .001$). Consequently, non-parametric statistical procedures were employed throughout the analysis.

1154.8, df = 5

To determine whether interpretive-to-tiered execution power use ratios differed between workload categories, a Kruskal–Wallis test was undertaken. The results indicated statistically significant differences in median power use ratios between workload categories, $\chi^2(5) = 1154$, $df = 5$, $p < .001$. Post hoc comparisons were subsequently conducted to identify the workload categories associated with those differences.

The results indicate that workload category influences the relative differences in power use observed between interpretive and tiered execution. In particular, workload categories associated with `apache` and `web` workloads exhibited comparatively larger power use ratios, whereas the `functional` and `scala` workload categories were associated with comparatively smaller power use ratios. These findings suggest that workload category moderates the relative power use differences observed between execution modes.

Energy Consumption Ratios

Descriptive Analysis

For energy consumption, substantially larger differences were observed between interpretive and tiered execution across all workload categories. Mean energy consumption ratios ranged from 5.48 for the `web` workload category to 19.46 for the `functional` workload category, while median ratios ranged from 5.31 for `web` to 19.64 for `concurrency`. These ratio values indicate that interpretive execution consistently consumed substantially more energy than tiered execution across all workload categories.

Unlike the power use ratios, all observed energy consumption ratios exceeded unity, indicating that every workload execution consumed more energy under interpretive execution irrespective of workload category. However, considerable variation was observed in the magnitude of the relative energy consumption differences between workload categories.

The `concurrency`, `database`, `functional`, and `scala` workload categories exhibited particularly large energy consumption ratios, indicating that interpretive execution consumed between approximately sixteen and twenty times more energy than tiered execution for these workload categories. In contrast, the `web` workload category exhibited substantially smaller ratios, with a median value of only 5.31, indicating comparatively smaller relative differences between execution modes.

Considerable variability was also observed within the ratio distributions. The `functional` workload category exhibited the greatest variability, with a standard deviation of 12.47 and an interquartile range of 23.52, whereas the `web` workload category exhibited the smallest dispersion, with a standard deviation of only 1.16 and an interquartile range of 2.24.

Inferential Statistical Analysis

Consistent with the power use ratio analysis, prior to evaluating differences in interpretive-to-tiered execution energy consumption ratios between workload categories, the ratio distributions were assessed for normality using the Shapiro–Wilk test. All distributions deviated significantly from normality (all $p < .001$). Inspection of the distributions was undertaken to identify potential outlying observations.

To determine whether interpretive-to-tiered execution energy consumption ratios differed between workload categories, a Kruskal–Wallis test was undertaken. Where statistically significant differences were observed, post hoc pairwise comparisons were conducted to identify the workload categories associated with those differences. The `scala` collection of workloads was the only workload category with significant outliers.

Consistent with the power use ratio analysis, the energy consumption ratio distributions were assessed for normality using the Shapiro–Wilk test. All workload category distributions deviated significantly from normality (all $p < .001$). Consequently, non-parametric procedures were employed throughout.

To determine whether interpretive-to-tiered execution energy consumption ratios differed between workload categories, a Kruskal–Wallis test was undertaken. The results indicated statistically significant differences in median energy consumption ratios between workload categories, $\chi^2(5) = 611.750$, $df = 5$, $p < .001$. Post hoc comparisons were subsequently conducted to identify the workload categories associated with those differences. The only cases for whicj no difference was observed was for comparisons between the `database` and `scala` workloads ($p = .482$), and between the `functional` and `scala` workloads ($p = .729$).

The results indicate substantial differences between workload categories in the relative energy consumption differences observed between execution modes. In particular, the `concurrency`, `database`, `functional`, and `scala` workload categories exhibited substantially larger energy consumption ratios than the `web` workload category, indicating that workload category exerts a strong moderating influence on the magnitude of the energy consumption differences observed between interpretive and tiered execution.

Summary

The findings indicate that workload category exerts a substantial effect on the relative energy consumption differences observed between interpretive and tiered execution. Statistically significant differences were observed between workload categories, indicating that workload category influences the magnitude of the relative energy consumption differences observed between execution modes.

The findings further indicate that considerable variation exists in the relative energy consumption penalty associated with interpretive execution across workload categories. While interpretive execution was consistently associated with higher

energy consumption than tiered execution across all workload categories, the magnitude of those differences varied substantially according to workload category, suggesting that workload category contribute meaningfully to variation in the relative energy consumption differences observed between execution modes.

In addition, the results demonstrate that interpretive execution was universally associated with substantially higher energy consumption than tiered execution irrespective of workload category. All observed energy consumption ratios exceeded unity, indicating that every workload execution consumed more energy under interpretive execution, although the magnitude of the observed energy consumption differences varied considerably between workload categories.

Collectively, these findings indicate that workload category moderates the relative energy consumption differences observed between execution modes. While interpretive execution remained consistently associated with substantially greater energy consumption than tiered execution across all benchmark workload categories, workload category exerted a strong influence on the magnitude of that relative energy consumption penalty.

5.5.4 Summary

In relation to **SRQ1.4**, the findings indicate that workload category had a statistically significant and substantial association with both JRE JVM power use and energy consumption. Across the six workload categories, statistically significant differences were observed for both measures, demonstrating that the workload executed by the JRE JVM represents one of the principal sources of variation in runtime power use and total energy consumption within the experimental environment considered in this study.

The findings further indicate that workload category was more strongly associated with variation in energy consumption than with variation in power use. For power use, workload-related differences were statistically significant and formed two broad groupings, with the `concurrency` and `web` workload categories generally

associated with higher power use than the remaining workload classes. In contrast, energy consumption exhibited substantially greater separation between workload categories, with the `database` workloads consistently associated with the highest energy consumption, followed by the `apache` and `concurrency` workloads, while the `functional`, `scala`, and `web` workloads were associated with comparatively lower energy consumption.

In relation to **SRQ1.6**, the results demonstrate that execution mode had different implications for power use and energy consumption across workload categories. For power use, neither interpretive nor tiered execution was consistently associated with lower values across all workload categories, indicating that both the direction and magnitude of execution-mode differences depended on the workload being executed. For energy consumption, however, tiered execution was consistently associated with substantially lower energy consumption than interpretive execution across all workload categories.

The interpretive-to-tiered execution ratio analysis reinforced these findings. Power use ratios remained comparatively close to unity, indicating that relative power use differences between execution modes were generally small, although their direction varied by workload category. In contrast, all energy consumption ratios exceeded unity, demonstrating that every workload execution consumed more energy under interpretive execution than under tiered execution. However, the magnitude of this relative energy consumption difference varied substantially across workload categories.

Taken together, these findings answer **SRQ1.4** by showing that workload category was associated with substantial and statistically significant differences in both JRE JVM power use and energy consumption, with the strongest workload-related differences observed for energy consumption. The findings also contribute to answering **SRQ1.6** by showing that execution mode had workload-dependent effects on power use, but consistently and substantially affected energy consumption, with tiered execution reducing energy consumption relative to interpretive

execution across all workload categories.

Section Research Question Findings

SRQ1.4: workload category was associated with substantial and statistically significant differences in both JRE JVM power use and energy consumption. The strongest workload-related differences were observed for energy consumption, with the `database`, `apache`, and `concurrency` workload categories generally associated with higher energy consumption than the `functional`, `scala`, and `web` categories.

SRQ1.6: Differences between interpretive and tiered execution had workload-dependent effects on JRE JVM power use, but represented a major and consistent determinant of JRE JVM energy consumption. Tiered execution consistently reduced energy consumption relative to interpretive execution across all workload categories, although the magnitude of this reduction varied substantially by workload category.

5.6 Microarchitectural Performance Event Relationships

Unlike the preceding sections, which examined the influence of categorical experimental factors on observed JRE JVM power use and energy consumption, this section examines the relationship between lower-level microarchitectural performance event measures and observed runtime power use and energy consumption. Specifically, the analyses evaluate whether variation in branch misprediction rates, L1 data load miss rates, L2 data load miss rates, and page fault rates is associated with differences in observed power use and energy consumption. In addition, variation in these microarchitectural performance event measures across OpenJDK

JRE versions and execution modes is examined to contextualise the relationships observed.

In addition, the purpose of the microarchitectural analysis presented in this section is not to comprehensively evaluate the influence of all experimental factors on processor-level execution characteristics. Rather, this analysis examines whether variation observed between OpenJDK JRE versions, particularly the elevated power use previously observed under OpenJDK JRE version 13, is accompanied by systematic differences in microarchitectural performance events.

Principally, this section addresses sub-research question **SRQ1.5**, which examines the relationship between observed microarchitectural performance event measures and JRE JVM power use and energy consumption:

> **SRQ1.5:** ***What relationship exists between observed microarchitectural performance event measures and JRE JVM power use and energy consumption?***

In addition, because the microarchitectural distributions and their relationships with power use and energy consumption are decomposed according to interpretive and tiered execution modes, this section also contributes evidence toward addressing **SRQ1.6**. In this context, SRQ1.6 is addressed by examining whether execution mode is associated with systematic differences in microarchitectural event rates and whether those differences help explain the observed separation between interpretive and tiered execution in power use and energy consumption:

> **SRQ1.6:** ***How do differences between interpretive and tiered execution affect JRE JVM power use and energy consumption across differing runtime conditions?***

In this section, we examine four microarchitectural performance event measures: branch misprediction rate, L1 data load miss rate, L2 data load miss rate, and page fault rate. For each measure, we first assess whether the event rate varies systematically across OpenJDK JRE versions and execution modes. We then

examine the relationship between the corresponding microarchitectural event rate and observed JRE JVM power use, followed by the relationship between that event rate and observed JRE JVM energy consumption. Where relevant, analyses are further decomposed according to execution mode to assess whether pooled relationships are influenced by aggregation across interpretive and tiered execution.

5.6.1 Branch Misprediction Relationships

JRE Effects

Descriptive Analysis

To evaluate the relationship between JRE JVM version and branch misprediction rate, we first examine the distribution of branch misprediction rates observed across the six OpenJDK JRE JVMs. Figure 5.13 presents box-and-whisker plots depicting the distribution of branch misprediction rates, while Table 5.13 summarises the corresponding descriptive statistics for all pooled observations, with results further decomposed according to interpretive and tiered execution.

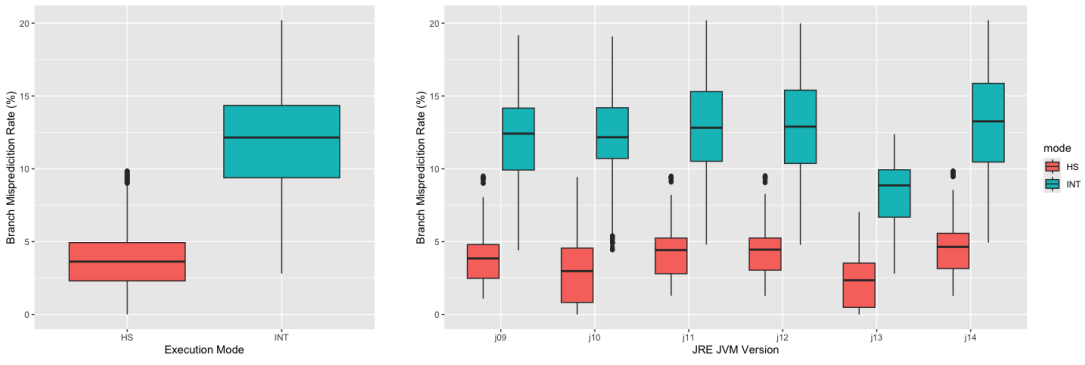


Figure 5.13: Branch misprediction rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b).

Across all pooled observations, branch misprediction rates exhibited substantial variation between JRE versions. Mean branch misprediction rates ranged from a low of 5.26% for OpenJDK JRE version 13 to a high of 8.69% for OpenJDK JRE version 14, representing an absolute difference of approximately 3.43 percentage

points. Median branch misprediction rates followed a similar pattern, varying from 4.36% under OpenJDK JRE version 13 to 6.31% under OpenJDK JRE version 14. With the exception of OpenJDK JRE version 13, the remaining JRE versions exhibited broadly similar branch misprediction distributions, with mean values ranging between 7.45% and 8.69%, and median values ranging between 5.39% and 6.31%.

Substantial variability was observed within each JRE distribution. Standard deviation values ranged from 3.65 percentage points for OpenJDK JRE version 13 to 5.41 percentage points for OpenJDK JRE version 10, while interquartile ranges varied between 6.51 and 9.18 percentage points. This indicates a relatively wide dispersion of branch misprediction rates across benchmark executions. OpenJDK JRE version 13 exhibited both the lowest central tendency and the narrowest overall distribution, suggesting comparatively lower branch misprediction rates relative to the remaining JRE versions.

Table 5.13: Branch Misprediction Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with branch misprediction rates.

	Pooled Observations (%)							**Execution Mode (%)**							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	1.09	19.17	7.45	5.39	4.88	8.61	864	HS	1.09	9.51	3.89	3.85	1.83	2.32	480
								INT	4.40	19.17	11.90	12.42	3.72	4.25	384
j10	0.00	19.09	7.56	5.86	5.41	9.18	768	HS	0.00	9.44	3.05	2.98	2.42	3.73	384
								INT	4.43	19.09	12.06	12.17	3.47	3.48	384
j11	1.29	20.19	8.44	6.03	5.22	8.40	768	HS	1.29	9.51	4.22	4.42	1.89	2.45	384
								INT	4.80	20.19	12.65	12.82	3.91	4.79	384
j12	1.28	19.98	8.50	6.05	5.23	8.44	736	HS	1.28	9.53	4.33	4.45	1.90	2.21	368
								INT	4.78	19.98	12.67	12.89	4.03	5.03	368
j13	0.00	12.37	5.26	4.36	3.65	6.51	736	HS	0.00	7.04	2.32	2.35	1.79	3.04	368
								INT	2.81	12.37	8.20	8.86	2.48	3.25	368
j14	1.27	20.21	8.69	6.31	5.24	8.61	736	HS	1.27	9.87	4.51	4.64	1.97	2.42	368
								INT	4.93	20.21	12.87	13.26	4.01	5.40	368
Avg.	0.82	18.50	7.65	5.67	4.94	8.29	768	HS Avg.	0.82	9.15	3.72	3.78	1.97	2.69	392
						Total:	**4608**	INT Avg.	4.36	18.50	11.73	12.07	3.60	4.37	376
													HS Total:		**2,352**
													INT Total:		**2,256**

When decomposed according to execution mode, substantial differences emerged between tiered execution (HS) and interpretive execution (INT). Across all JRE

versions, branch misprediction rates under tiered execution remained consistently low, with mean values ranging between 2.32% for OpenJDK JRE version 13 and 4.51% for OpenJDK JRE version 14. Median values under tiered execution were similarly stable, ranging between 2.35% and 4.64%. Overall, the average branch misprediction rate under tiered execution was 3.72%, with an associated median of 3.78%.

In contrast, interpretive execution consistently exhibited substantially greater branch misprediction rates across all JRE versions. Mean branch misprediction rates ranged from 8.20% under OpenJDK JRE version 13 to 12.87% under OpenJDK JRE version 14, while median values ranged between 8.86% and 13.26%. Across all observations, the average branch misprediction rate under interpretive execution was 11.73%, with an associated median of 12.07%.

Comparing execution modes, interpretive execution consistently produced branch misprediction rates approximately three times greater than those observed under tiered execution across all JRE versions. This separation between execution mode distributions was substantially greater than the variation observed between individual JRE versions, suggesting that execution mode constituted the dominant source of variation in branch misprediction rates within the observed dataset.

Inferential Statistical Analysis

In regard to all OpenJDK JRE JVM versions, the results of Shapiro-Wilk tests of normality indicated that branch misprediction distributions statistically significantly deviated from normality in all cases (all $p < .001$). Consequently, a non-parametric Kruskal-Wallis rank sum test was conducted to assess whether branch misprediction rates differed significantly between JRE versions.

The results of the Kruskal-Wallis test indicated statistically significant differences in branch misprediction distributions between JRE versions, $\chi^2 = 242.100$, $df = 5$, $p < .001$. To identify the source of these differences, pairwise Wilcoxon rank sum tests were subsequently conducted between all JRE version combinations. The pairwise comparisons revealed that OpenJDK JRE version 13 exhibited

statistically significant differences in branch misprediction rates relative to all remaining JRE versions ($p < .001$ in all cases), consistently exhibiting lower branch misprediction rates than all other JRE versions.

Beyond the differences associated with OpenJDK JRE version 13, additional statistically significant differences were observed between OpenJDK JRE versions 9 and 11 ($p < .001$), versions 9 and 12 ($p < .001$), versions 9 and 14 ($p < .001$), versions 10 and 11 ($p < .001$), versions 10 and 12 ($p < .001$), and versions 10 and 14 ($p < .001$). In contrast, no statistically significant differences were observed between OpenJDK JRE versions 9 and 10 ($p = .921$), versions 11 and 12 ($p = .728$), versions 11 and 14 ($p = .105$), or versions 12 and 14 ($p = .211$).

Collectively, these pairwise comparisons indicate that branch misprediction rates formed three statistically distinguishable groupings across JRE versions. OpenJDK JRE version 13 consistently exhibited the lowest branch misprediction rates, OpenJDK JRE versions 9 and 10 formed an intermediate grouping characterised by moderate branch misprediction rates, while OpenJDK JRE versions 11, 12, and 14 exhibited comparatively elevated branch misprediction rates. These findings indicate that branch misprediction rate varied systematically across JRE versions rather than being driven solely by the anomalous characteristics previously observed under OpenJDK JRE version 13.

To assess whether execution mode influenced branch misprediction rates within each OpenJDK JRE JVM version, pairwise Wilcoxon rank sum tests were conducted comparing tiered execution and interpretive execution distributions separately for each JRE version.

The results indicated statistically significant differences in branch misprediction rates between execution modes for all six OpenJDK JRE versions (all $p < .001$). In all cases, interpretive execution exhibited substantially greater branch misprediction rates relative to tiered execution. Across all observations, average branch misprediction rates under interpretive execution were approximately three times greater than those observed under tiered execution.

Collectively, these findings demonstrate that execution mode constituted a consistent source of variation in branch misprediction rate across all OpenJDK JRE versions, with the observed differences between execution modes being substantially greater than many of the differences observed between individual JRE versions themselves.

Summary

The descriptive analysis indicated that branch misprediction rates varied systematically across OpenJDK JRE JVM versions. Although branch misprediction distributions remained broadly comparable across most JRE versions, OpenJDK JRE version 13 consistently exhibited the lowest branch misprediction rates, together with the narrowest overall distribution. In contrast, OpenJDK JRE versions 11, 12, and 14 exhibited comparatively elevated branch misprediction rates relative to the remaining JRE versions. When decomposed according to execution mode, substantial differences were observed between execution modes, with interpretive execution consistently exhibiting substantially greater branch misprediction rates relative to tiered execution across all JRE versions.

Inferential analysis supported these observations. Statistically significant differences in branch misprediction distributions were observed between JRE versions, with pairwise comparisons indicating that OpenJDK JRE version 13 differed significantly from all remaining JRE versions. Additional pairwise differences were observed between OpenJDK JRE versions 9 and 11, versions 9 and 12, versions 9 and 14, versions 10 and 11, versions 10 and 12, and versions 10 and 14, suggesting that branch misprediction rates formed three broad groupings across JRE versions. Statistically significant differences were also observed between interpretive and tiered execution for all six JRE versions, with interpretive execution consistently exhibiting branch misprediction rates approximately three times greater than those observed under tiered execution.

Collectively, these findings indicate that branch misprediction rate varied systematically across OpenJDK JRE JVM versions and was strongly influenced

by execution mode, with execution mode constituting the dominant source of variation in observed branch misprediction rates.

Branch Misprediction and Power Use Relationships

To assess whether branch misprediction rates were associated with observed JRE JVM power use, we first visually examine the relationship between both variables through the scatterplot presented in Figure 5.14. In addition, the distributional characteristics of both variables were examined prior to inferential testing. Shapiro-Wilk tests of normality indicated that both branch misprediction rates and observed power use statistically significantly deviated from normality, branch misprediction rate ($W = 0.938$, $p < .001$), and power use ($W = 0.967$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between branch misprediction rate and observed power use.

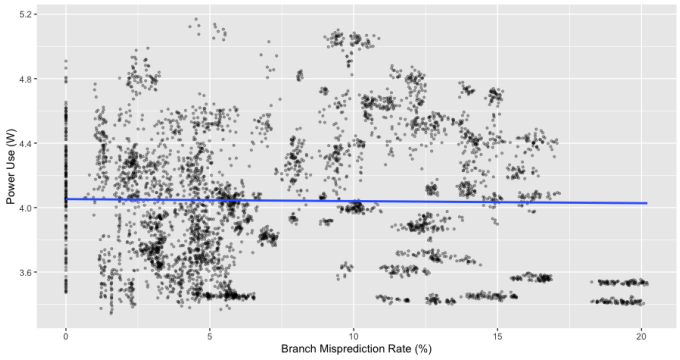


Figure 5.14: Branch Misprediction and Power Use Relationship: Scatterplot depicting the relationship between observed branch misprediction rates and measured JRE JVM power use across all pooled observations.

The results indicated no statistically significant monotonic relationship between branch misprediction rates and observed power use, $\rho = -0.014$, $p = .335$. The magnitude of the observed correlation coefficient was negligible, indicating that variation in branch misprediction rate was not associated with meaningful variation in observed JRE JVM power use.

These findings suggest that variation in branch misprediction rate does not

meaningfully contribute to observed variation in instantaneous JRE JVM power use. Despite measurable differences in branch misprediction rates across JRE versions and execution modes, these differences were not associated with practically meaningful variation in instantaneous power use within the observed experimental environment.

Branch Misprediction and Energy Consumption Relationships

To assess whether branch misprediction rates were associated with observed JRE JVM energy consumption, we first examine the relationship between both variables through the scatterplot presented in Figure 5.15. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both branch misprediction rates and observed energy consumption statistically significantly deviated from normality, branch misprediction rate ($W = 0.938$, $p < .001$), and energy consumption ($W = 0.629$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between branch misprediction rate and observed energy consumption.

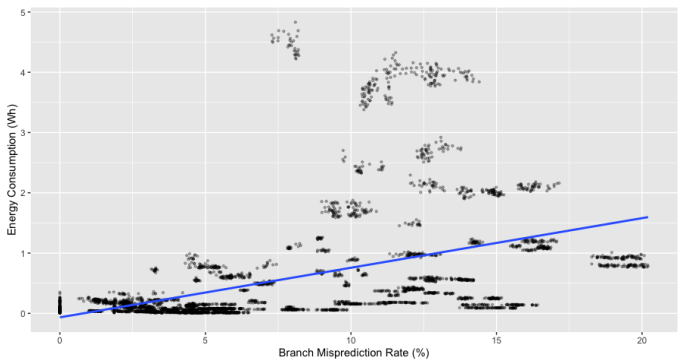


Figure 5.15: Branch Misprediction and Energy Consumption Relationship: Scatterplot depicting the relationship between observed branch misprediction rates and measured JRE JVM energy consumption across all pooled observations.

The results indicated a statistically significant positive monotonic relationship between branch misprediction rates and observed energy consumption, $\rho = 0.606$, $p < .001$, suggesting that observations characterised by greater branch mispredic-

tion rate also tended to exhibit greater cumulative energy consumption.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, substantially different relationships emerged. Under tiered execution, a statistically significant moderate negative monotonic relationship was observed between branch misprediction rates and observed energy consumption, $\rho = -0.327$, $p < .001$, indicating that observations characterised by comparatively greater branch misprediction rate were associated with lower measured cumulative energy consumption within the tiered execution environment.

In contrast, under interpretive execution, a statistically significant but comparatively weak positive monotonic relationship was observed, $\rho = 0.140$, $p < .001$, indicating that greater branch misprediction rate was associated with only modest increases in cumulative energy consumption under interpretive execution.

Collectively, these findings indicate that the strong positive relationship observed across pooled observations was substantially influenced by execution mode aggregation effects. Because interpretive execution simultaneously exhibited both substantially greater branch misprediction rates and substantially greater overall energy consumption relative to tiered execution, pooling observations across execution modes artificially inflated the apparent overall association between branch misprediction rate and energy consumption.

These findings suggest that branch misprediction rate does not independently constitute a strong determinant of observed JRE JVM energy consumption. Rather, execution mode appears to represent the dominant underlying factor influencing both branch misprediction rate and cumulative energy consumption, with the strong positive pooled relationship primarily reflecting the substantial structural differences between tiered and interpretive execution.

Summary

The descriptive analysis indicated that branch misprediction rates varied systematically across OpenJDK JRE JVM versions. Although branch misprediction distributions remained broadly comparable across most JRE versions, OpenJDK JRE version 13 consistently exhibited substantially lower branch misprediction rates relative to all remaining JRE versions, together with the narrowest overall distribution. When decomposed according to execution mode, substantial differences were observed between tiered and interpretive execution, with interpretive execution consistently exhibiting branch misprediction rates approximately three times greater than those observed under tiered execution across all JRE versions.

Inferential statistical analysis supported these observations. Statistically significant differences in branch misprediction rates were observed between JRE versions, with pairwise comparisons indicating that OpenJDK JRE version 13 differed significantly from all remaining JRE versions. Additional statistically significant differences were observed between several other JRE version pairings, indicating that branch misprediction rate varied systematically across JRE implementations rather than being solely attributable to the anomalous characteristics observed under OpenJDK JRE version 13. Statistically significant differences were also observed between interpretive and tiered execution for all six JRE versions, confirming that execution mode constituted the dominant source of variation in branch misprediction rate.

Correlation analysis further indicated that branch misprediction rate exhibited no statistically significant relationship with observed power use. Although pooled observations initially indicated a strong positive monotonic relationship between branch misprediction rate and observed energy consumption, further decomposition according to execution mode revealed substantially different underlying relationships. Under tiered execution, branch misprediction rate exhibited a moderate negative relationship with energy consumption, whereas under interpretive execution only a comparatively weak positive relationship was observed. These

findings indicate that the strong pooled association between branch misprediction rate and energy consumption was substantially influenced by execution mode aggregation effects.

Collectively, these findings suggest that branch misprediction rate does not independently constitute a strong determinant of either instantaneous JRE JVM power use or cumulative energy consumption. Rather, execution mode appears to represent the dominant underlying factor influencing both branch misprediction rate and overall energy consumption, with the observed pooled relationships primarily reflecting the substantial structural differences between tiered and interpretive execution.

These findings contribute, in part, to addressing the broader microarchitectural research questions concerning the extent to which processor-level execution characteristics influence observed JRE JVM power use and energy consumption.

5.6.2 L1 Data Cache Load Miss Rate Relationships

JRE Effects

Descriptive Analysis

To evaluate the relationship between JRE JVM version and L1 data cache load miss rates, we first examine the distribution of L1 data cache load miss rates observed across the six OpenJDK JRE JVMs. Figure 5.16 presents box-and-whisker plots depicting the distribution of L1 data cache load miss rates, while Table 5.14 summarises the corresponding descriptive statistics for all pooled observations, with results further decomposed according to interpretive and tiered execution.

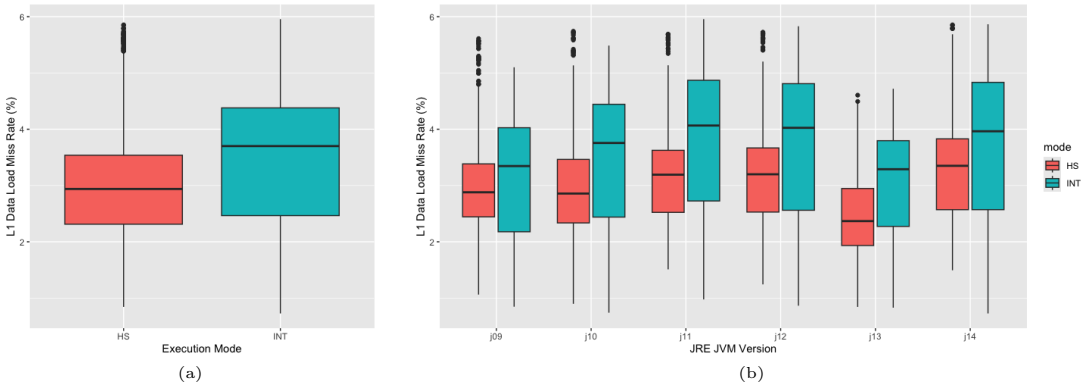


Figure 5.16: L1 Data Cache Load Miss Rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b).

Across all pooled observations, L1 data cache load miss rates exhibited moderate variation between JRE versions. Mean L1 data cache load miss rates ranged from a low of 2.80% for OpenJDK JRE version 13 to a high of 3.55% for OpenJDK JRE version 14, representing an absolute difference of approximately 0.75 percentage points. Median values followed a similar pattern, varying from 2.75% under OpenJDK JRE version 13 to 3.56% under OpenJDK JRE version 14. Consistent with the previously observed branch misprediction results, OpenJDK JRE version 13 exhibited noticeably lower miss rates relative to the remaining JRE versions, while the other five JRE versions remained comparatively similar, with mean values ranging between 3.08% and 3.55%.

Variability within each JRE distribution remained comparatively modest. Standard deviation values ranged from 0.90 percentage points for OpenJDK JRE version 13 to 1.20 percentage points for OpenJDK JRE version 14, while interquartile ranges varied between 1.30 and 1.81 percentage points. Overall, this indicates comparatively compact L1 data cache load miss rate distributions across benchmark executions. OpenJDK JRE version 13 again exhibited both the lowest central tendency and the narrowest overall distribution, suggesting comparatively reduced L1 data cache load miss activity relative to the remaining JRE versions.

Table 5.14: L1 Data Cache Load Miss Rate Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with L1 data cache load miss rates.

	Pooled Observations (%)							Execution Mode (%)							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	0.85	5.61	3.08	3.01	1.03	1.30	864	HS	1.06	5.61	3.00	2.88	0.94	0.94	480
								INT	0.85	5.10	3.18	3.35	1.12	1.85	384
j10	0.74	5.74	3.24	3.13	1.15	1.57	768	HS	0.90	5.74	2.97	2.86	1.04	1.13	384
								INT	0.74	5.49	3.50	3.76	1.20	2.00	384
j11	0.98	5.96	3.54	3.44	1.19	1.76	768	HS	1.51	5.69	3.22	3.19	0.99	1.10	384
								INT	0.98	5.96	3.85	4.07	1.28	2.14	384
j12	0.86	5.83	3.54	3.48	1.19	1.81	736	HS	1.24	5.72	3.23	3.20	1.02	1.14	368
								INT	0.86	5.83	3.85	4.03	1.27	2.25	368
j13	0.83	4.72	2.80	2.75	0.90	1.51	736	HS	0.84	4.61	2.48	2.37	0.77	1.01	368
								INT	0.83	4.72	3.11	3.29	0.92	1.52	368
j14	0.73	5.87	3.55	3.56	1.20	1.81	736	HS	1.49	5.85	3.29	3.35	1.04	1.26	368
								INT	0.73	5.87	3.82	3.96	1.30	2.26	368
Avg.	0.83	5.62	3.29	3.23	1.11	1.63	768	HS Avg.	1.18	5.54	3.03	2.97	0.97	1.10	392
						Total:	**4608**	INT Avg.	0.83	5.49	3.55	3.74	1.18	2.01	376
														HS Total:	**2,352**
														INT Total:	**2,256**

When decomposed according to execution mode, differences emerged between tiered execution (HS) and interpretive execution (INT). Across all JRE versions, L1 data cache load miss rates under tiered execution remained relatively consistent, with mean values ranging between 2.48% for OpenJDK JRE version 13 and 3.29% for OpenJDK JRE version 14. Median values under tiered execution ranged from 2.37% to 3.35%. Across all tiered execution observations, the average L1 data cache load miss rate was 3.03%, with an associated median of 2.97%.

Interpretive execution exhibited consistently greater L1 data cache load miss rates across all JRE versions. Mean miss rates ranged from 3.11% under OpenJDK JRE version 13 to 3.85% under OpenJDK JRE versions 11 and 12, while median values ranged between 3.29% and 4.07%. Across all observations, interpretive execution exhibited an average L1 data cache load miss rate of 3.55%, with an associated median of 3.74%.

Comparing execution modes, interpretive execution consistently produced moderately greater L1 data cache load miss rates relative to tiered execution across all JRE versions. However, unlike the previously observed branch misprediction

results, the magnitude of separation between execution mode distributions was comparatively modest, with the observed differences between execution modes being substantially smaller than those previously observed for branch misprediction rates. These findings suggest that while execution mode contributed to variation in observed L1 data cache load miss rates, the magnitude of this effect was comparatively less pronounced than that observed for branch misprediction activity.

Inferential Statistical Analysis

In regard to all OpenJDK JRE JVM versions, the results of Shapiro-Wilk tests of normality indicated that L1 data cache miss rate distributions statistically significantly deviated from normality in all cases (all $p < .001$). Consequently, a non-parametric Kruskal-Wallis rank sum test was conducted to assess whether L1 data cache miss rates differed significantly between JRE versions.

The results of the Kruskal-Wallis test indicated statistically significant differences in L1 data cache load miss rate distributions between JRE versions, $\chi^2 = 259.560$, $df = 5$, $p < .001$. To identify the source of these differences, pairwise Wilcoxon rank sum tests were subsequently conducted between all JRE version combinations. The pairwise comparisons revealed that OpenJDK JRE version 13 exhibited statistically significant differences in L1 data cache miss rates relative to all remaining JRE versions ($p < .001$ in all cases), consistently exhibiting lower L1 data cache load miss rates than all other JRE versions.

Beyond the differences associated with OpenJDK JRE version 13, additional statistically significant differences were observed between OpenJDK JRE versions 9 and 11 ($p < .001$), versions 9 and 12 ($p < .001$), versions 9 and 14 ($p < .001$), versions 10 and 11 ($p < .001$), versions 10 and 12 ($p < .001$), and versions 10 and 14 ($p < .001$). In contrast, no statistically significant differences were observed between OpenJDK JRE versions 9 and 10 ($p = .098$), versions 11 and 12 ($p = 1.000$), versions 11 and 14 ($p = 1.000$), or versions 12 and 14 ($p = 1.000$).

Collectively, these pairwise comparisons indicate that L1 data cache load

miss rates formed three broad statistically distinguishable groupings across JRE versions. OpenJDK JRE version 13 consistently exhibited the lowest L1 data cache load miss rates and differed significantly from all remaining JRE versions. OpenJDK JRE versions 9 and 10 formed an intermediate grouping, with no statistically significant difference observed between these versions. In contrast, OpenJDK JRE versions 11, 12, and 14 exhibited comparatively elevated L1 data cache load miss rates and did not significantly differ from one another. These findings indicate that L1 data cache load miss rate varied systematically across JRE versions rather than reflecting variation attributable solely to the anomalous characteristics previously observed under OpenJDK JRE version 13.

To assess whether execution mode influenced L1 data cache miss rates within each OpenJDK JRE JVM version, pairwise Wilcoxon rank sum tests were conducted comparing tiered execution and interpretive execution distributions separately for each JRE version.

The results indicated statistically significant differences in L1 data cache miss rates between execution modes for all six OpenJDK JRE versions (all $p < .001$). In all cases, interpretive execution exhibited consistently greater L1 data cache load miss rates relative to tiered execution, although the magnitude of these differences remained comparatively modest. Across all observations, average L1 data cache load miss rates under interpretive execution were approximately 17% greater than those observed under tiered execution.

Collectively, these findings demonstrate that execution mode constituted a consistent source of variation in L1 data cache miss rate across all OpenJDK JRE versions, with interpretive execution consistently exhibiting moderately greater miss activity relative to tiered execution under every JRE implementation examined.

L1 Data Cache Load Miss Rate and Power Use Relationships

To assess whether L1 data cache load miss rates were associated with observed JRE JVM power use, we first examine the relationship between both variables through the scatterplot presented in Figure 5.17. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both L1 data cache load miss rate and observed power use statistically significantly deviated from normality, L1 data cache load miss rate ($W = 0.981$, $p < .001$), and power use ($W = 0.967$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between L1 data cache load miss rate and observed power use.

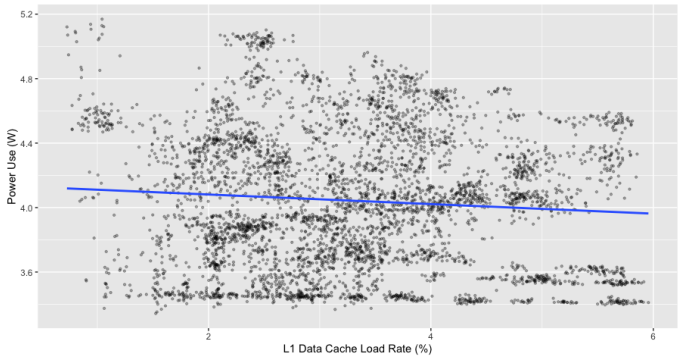


Figure 5.17: L1 Data Cache Load Miss Rate and Power Use Relationship: Scatterplot depicting the relationship between observed L1 data cache load miss rates and measured JRE JVM power use across all pooled observations.

The results indicated a statistically significant negative monotonic relationship between L1 data cache load miss rate and observed power use, $\rho = -0.044$, $p = .003$. However, the magnitude of the observed correlation coefficient was negligible, indicating that variation in L1 data cache load miss rate was associated with only an extremely small proportion of the overall variation observed in JRE JVM power use.

Although the observed relationship reached statistical significance, this result is likely attributable to the large sample size rather than reflecting a practically meaningful association between both variables. The negative direction of the

relationship indicates that observations characterised by comparatively greater L1 data cache load miss rates tended to exhibit marginally lower observed power use, although the magnitude of this effect was extremely small.

These findings suggest that variation in L1 data cache load miss rates does not meaningfully contribute to observed variation in instantaneous JRE JVM power use. Despite measurable differences in L1 data cache load miss rates across JRE versions and execution modes, these differences do not correspond to practically meaningful variation in measured power use. This indicates that failed L1 data cache accesses do not independently impose a sufficiently large processor-level penalty to measurably influence instantaneous power use within the observed experimental environment.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, substantially different relationships emerged. Under interpretive execution, a statistically significant negative monotonic relationship was observed between L1 data cache load miss rate and observed power use, $\rho = -0.207$, $p < .001$, indicating that observations characterised by comparatively greater L1 data cache load miss rates tended to exhibit lower measured power use.

In contrast, under tiered execution, a statistically significant but comparatively weak positive monotonic relationship was observed, $\rho = 0.101$, $p < .001$, indicating that observations characterised by comparatively greater L1 data cache load miss rates tended to exhibit slightly greater measured power use.

Collectively, these findings indicate that the pooled relationship substantially masks structurally different relationships between L1 data cache load miss behaviour and observed power use across execution modes. Under interpretive execution, increased L1 data cache load miss activity appears associated with modest reductions in instantaneous power use, potentially reflecting differences

in processor execution activity associated with memory access behaviour. Under tiered execution, the weak positive relationship suggests that greater L1 cache activity may coincide with slightly greater sustained processor activity.

These findings suggest that L1 data cache load miss rate does not independently constitute a strong global determinant of observed JRE JVM power use. Rather, execution mode appears to moderate the relationship substantially, with the weak pooled relationship primarily reflecting the aggregation of two fundamentally different execution environments.

L1 Data Cache Load Miss Rate and Energy Consumption Relationships

To assess whether L1 data cache load miss rates were associated with observed JRE JVM energy consumption, we first examine the relationship between both variables through the scatterplot presented in Figure 5.18. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both L1 data cache load miss rate and observed energy consumption statistically significantly deviated from normality, L1 data cache load miss rate ($W = 0.981$, $p < .001$), and energy consumption ($W = 0.629$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between L1 data cache load miss rate and observed energy consumption.

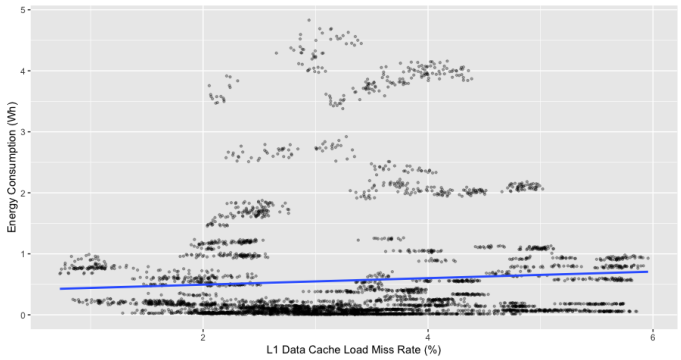


Figure 5.18: L1 Data Cache Load Miss Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed L1 data cache miss rates and measured JRE JVM energy consumption across all pooled observations.

Across all pooled observations, the results indicated a statistically significant positive monotonic relationship between L1 data cache load miss rate and observed energy consumption, $\rho = 0.057$, $p < .001$. However, the magnitude of the observed correlation coefficient was negligible, indicating that variation in L1 data cache load miss rate was associated with only minimal variation in observed cumulative JRE JVM energy consumption.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, substantially different relationships emerged. Under interpretive execution, a statistically significant weak negative monotonic relationship was observed between L1 data cache load miss rate and observed energy consumption, $\rho = -0.106$, $p < .001$, indicating that observations characterised by comparatively greater L1 data cache load miss rates tended to exhibit slightly lower measured cumulative energy consumption.

Under tiered execution, a substantially stronger statistically significant negative monotonic relationship was observed, $\rho = -0.345$, $p < .001$, indicating that observations characterised by comparatively greater L1 data cache load miss rates were associated with comparatively lower measured cumulative energy consumption.

Collectively, these findings indicate that the weak positive relationship observed across pooled observations substantially masks fundamentally different underlying relationships once execution mode is controlled for. Although pooled observations initially suggested a positive relationship between L1 data cache load miss rate and observed energy consumption, decomposition according to execution mode revealed consistently negative relationships under both execution environments.

These findings suggest that L1 data cache load miss rate does not exhibit a strong relationship with cumulative JRE JVM energy consumption. Rather, the positive pooled relationship appears primarily attributable to execution mode

aggregation effects. Execution mode appears to represent the dominant structural factor influencing both observed L1 data cache load miss rates and cumulative energy consumption, with the pooled association largely reflecting the substantial differences between interpretive and tiered execution environments.

Summary

The descriptive analysis indicated that L1 data cache load miss rates varied systematically across OpenJDK JRE JVM versions. Although the magnitude of variation between JRE versions was comparatively modest, OpenJDK JRE version 13 consistently exhibited the lowest L1 data cache load miss rates together with the narrowest overall distribution. This pattern closely mirrors the branch misprediction results previously observed, providing further evidence that OpenJDK JRE version 13 exhibited distinct underlying execution characteristics relative to the remaining JRE implementations. When decomposed according to execution mode, interpretive execution consistently exhibited moderately greater L1 data cache load miss rates than tiered execution across all JRE versions, although the magnitude of these differences was substantially smaller than those previously observed for branch misprediction activity.

Inferential statistical analysis supported these observations. Statistically significant differences in L1 data cache load miss rate were observed between JRE versions, with pairwise comparisons indicating that OpenJDK JRE version 13 consistently differed from the remaining JRE versions by exhibiting comparatively reduced miss activity. Statistically significant differences were also observed between interpretive and tiered execution across all JRE versions, confirming that execution mode constituted a consistent source of variation in observed L1 data cache load miss rates.

Correlation analysis indicated that L1 data cache load miss rates exhibited no practically meaningful global relationship with observed JRE JVM power use. Although pooled observations indicated a statistically significant negative relationship between L1 data cache load miss rate and power use, the magnitude of

the observed association was negligible. When decomposed according to execution mode, structurally different relationships emerged, with interpretive execution exhibiting a weak negative relationship and tiered execution exhibiting a weak positive relationship. These findings indicate that the pooled relationship masks substantially different underlying execution mode behaviours.

A similar pattern was observed for energy consumption. Although pooled observations initially suggested a weak positive relationship between L1 data cache load miss rate and cumulative energy consumption, decomposition according to execution mode revealed consistently negative relationships under both interpretive and tiered execution. In particular, the substantially stronger negative relationship observed under tiered execution indicates that the weak positive pooled relationship was primarily attributable to aggregation effects introduced by combining fundamentally different execution environments.

Collectively, these findings suggest that L1 data cache load miss rate does not exhibit a strong relationship with either instantaneous JRE JVM power use or cumulative energy consumption. While L1 data cache load miss behaviour varied systematically across JRE implementations and execution modes, the observed relationships with power use and energy consumption were weak, inconsistent, and highly dependent upon execution mode decomposition. Similar to the previously observed branch misprediction results, these findings indicate that execution mode represents the dominant structural factor governing both observed microarchitectural performance event patterns and the corresponding runtime power use and energy consumption characteristics observed within the experimental environment examined in this study.

These findings contribute toward addressing the broader microarchitectural research question by demonstrating that lower-level processor execution characteristics cannot be interpreted independently of the execution environment under which they occur, with execution mode consistently emerging as the dominant factor governing the observed relationships.

5.6.3 L2 Data Cache Load Miss Rate Relationships

JRE Effects

Descriptive Analysis

To evaluate the relationship between JRE JVM version and L2 data cache load miss rates, we first examine the distribution of L2 data cache load miss rates observed across the six OpenJDK JRE JVMs. Figure 5.19 presents box-and-whisker plots depicting the distribution of L2 data cache load miss rates, while Table 5.15 summarises the corresponding descriptive statistics for all pooled observations, with results further decomposed according to interpretive and tiered execution.

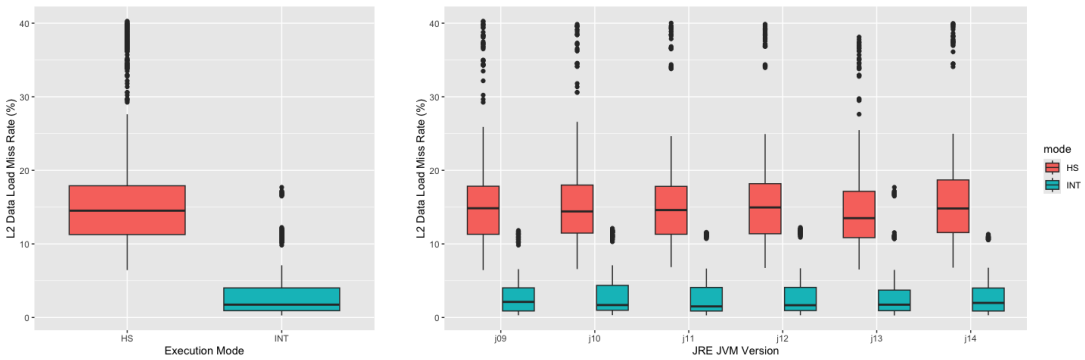


Figure 5.19: L2 Data Cache Load Miss Rate distributions across execution mode categories (a), and JRE version clustered by execution mode (b).

Across all pooled observations, L2 data cache load miss rates exhibited relatively limited variation between JRE versions. Mean L2 data cache load miss rates ranged from a low of 9.18% for OpenJDK JRE version 13 to a high of 10.20% for OpenJDK JRE version 9, representing an absolute difference of approximately 1.02 percentage points. Median L2 data cache load miss rates followed a similar pattern, ranging from 8.38% under OpenJDK JRE version 13 to 9.62% under OpenJDK JRE version 9. In contrast to branch misprediction and L1 data load miss rates, differences between JRE versions were comparatively small, with all six JRE versions exhibiting broadly similar L2 data cache load miss rate distributions.

Substantial variability was observed within each JRE distribution. Standard

deviation values ranged from 8.22 percentage points for OpenJDK JRE version 13 to 8.94 percentage points for OpenJDK JRE version 14, while interquartile ranges varied between 12.55 and 13.25 percentage points. This indicates considerable dispersion in L2 data cache load miss rates across benchmark executions. Although OpenJDK JRE version 13 exhibited the lowest observed central tendency, the magnitude of these differences remained comparatively small and did not materially distinguish OpenJDK JRE version 13 from the remaining JRE implementations.

Table 5.15: L2 Data Load Miss Rate Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with L2 data cache load miss rates.

	Pooled Observations (%)							**Execution Mode (%)**							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	0.30	40.28	10.20	9.62	8.69	13.09	864	HS	6.45	40.28	16.01	14.85	7.28	6.54	480
								INT	0.30	11.83	2.94	2.12	2.91	3.14	384
j10	0.32	39.85	9.47	8.45	8.47	12.71	768	HS	6.59	39.85	15.90	14.42	7.17	6.53	384
								INT	0.32	12.12	3.03	1.69	3.03	3.38	384
j11	0.29	40.01	9.52	8.86	8.79	13.08	768	HS	6.85	40.01	16.18	14.60	7.54	6.51	384
								INT	0.29	11.58	2.86	1.51	3.00	3.20	384
j12	0.31	39.87	9.66	8.87	8.85	13.25	736	HS	6.74	39.87	16.32	14.96	7.64	6.81	368
								INT	0.31	12.24	3.01	1.67	3.10	3.15	368
j13	0.28	38.12	9.18	8.38	8.22	12.55	736	HS	6.53	38.12	15.22	13.50	6.94	6.29	368
								INT	0.28	17.70	3.14	1.74	3.76	2.80	368
j14	0.31	39.96	9.68	8.69	8.94	12.73	736	HS	6.78	39.96	16.41	14.82	7.78	7.15	368
								INT	0.31	11.32	2.95	1.99	2.92	3.12	368
Avg.	0.30	39.68	9.62	8.81	8.66	12.90	768	HS Avg.	6.65	39.68	16.01	14.53	7.39	6.64	392
						Total	**4608**	INT Avg.	0.30	12.80	2.99	1.79	3.12	3.13	376
														HS Total:	**2352**
														INT Total:	**2256**

When decomposed according to execution mode, substantial differences emerged between tiered execution (HS) and interpretive execution (INT), although the pattern observed differed substantially from that previously observed for branch misprediction and L1 data cache load miss rates. Whereas interpretive execution previously exhibited elevated branch misprediction and L1 data cache load miss rates, L2 data cache load miss rates exhibited the opposite pattern, with tiered execution consistently producing substantially greater L2 cache miss rates across all JRE versions.

Across all JRE versions, L2 data cache load miss rates under tiered execution

remained consistently elevated, with mean values ranging between 15.22% for OpenJDK JRE version 13 and 16.41% for OpenJDK JRE version 14. Median values under tiered execution ranged from 13.50% to 14.96%. Across all tiered execution observations, the average L2 data load miss rate was 16.01%, with an associated median of 14.53%.

In contrast, interpretive execution consistently exhibited substantially lower L2 data cache load miss rates across all JRE versions. Mean miss rates ranged from 2.86% under OpenJDK JRE version 11 to 3.14% under OpenJDK JRE version 13, while median values ranged between 1.51% and 2.12%. Across all observations, interpretive execution exhibited an average L2 data load miss rate of 2.99%, with an associated median of 1.79%.

Comparing execution modes, tiered execution consistently produced substantially greater L2 data cache load miss rates relative to interpretive execution across all JRE versions, with average miss rates approximately five times greater than those observed under interpretive execution. This separation between execution mode distributions was considerably larger than the relatively minor differences observed between individual JRE versions, indicating that execution mode contributed substantially to the overall variation observed in L2 data cache load miss rates within the experimental dataset.

Inferential Statistical Analysis

In regard to all OpenJDK JRE JVM versions, the results of Shapiro-Wilk tests of normality indicated that L2 data cache load miss rate distributions statistically significantly deviated from normality in all cases (all $p < .001$). Consequently, a non-parametric Kruskal-Wallis rank sum test was conducted to assess whether L2 data cache load miss rates differed significantly between JRE versions.

The results of the Kruskal-Wallis test indicated that no statistically significant differences were observed between JRE versions, $\chi^2 = 8.084$, $df = 5$, $p = .152$. Pairwise Wilcoxon rank sum tests similarly indicated no statistically significant

pairwise differences between any JRE version combinations following Bonferroni adjustment (all $p > .05$).

These findings indicate that L2 data cache load miss rates remained statistically stable across all OpenJDK JRE JVM versions. Unlike the previously observed branch misprediction and L1 data cache load miss rate results, no evidence was observed to suggest systematic variation in L2 data cache load miss rates between JRE implementations.

Collectively, these findings suggest that OpenJDK JRE JVM version was not associated with statistically meaningful differences in observed L2 data cache load miss rates. Although OpenJDK JRE version 13 exhibited marginally lower average L2 data cache load miss rates, these differences remained small relative to the overall spread of observations and were insufficient to produce statistically significant differences.

L2 Data Cache Load Miss Rate and Power Use Relationships

To assess whether L2 data cache load miss rates were associated with observed JRE JVM power use, we first examine the relationship between both variables through the scatterplot presented in Figure 5.20. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both L2 data cache load miss rate and observed power use statistically significantly deviated from normality, L1 data cache load miss rate ($W = 0.872$, $p < .001$), and power use ($W = 0.967$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between L1 data cache load miss rate and observed power use. The results indicated a statistically significant positive monotonic relationship between L2 data cache load miss rate and observed JRE JVM power use, $\rho = 0.085$, $p < .001$, suggesting that observations characterised by comparatively greater L2 data cache load miss rates also tended to exhibit marginally greater measured power use.

However, the magnitude of the observed correlation coefficient remained extremely small, indicating that variation in L2 data cache load miss rate was associated with only a negligible proportion of the overall variation observed in JRE JVM power use.

Although the observed relationship reached statistical significance, this result is likely attributable primarily to the large sample size rather than reflecting a practically meaningful association between both variables. While the positive direction of the relationship suggests that observations characterised by comparatively greater L2 data cache load miss rates tended to exhibit slightly greater observed power use, the magnitude of this effect remained extremely limited. These findings suggest that L2 data cache load miss rate ddoes not exhibit a strong relationship with instantaneous JRE JVM power use.

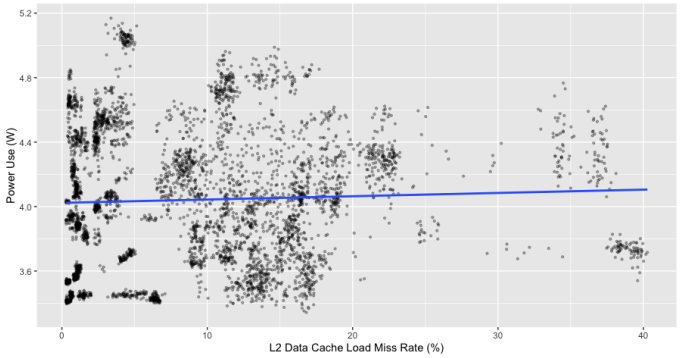


Figure 5.20: L2 Data Cache Load Miss Rate and Power Use Relationship: Scatterplot depicting the relationship between observed L2 data cache load miss rates and measured JRE JVM power use across all pooled observations.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, substantially different relationships emerged. Under interpretive execution, a statistically significant moderate positive monotonic relationship was observed between L2 data cache load miss rate and observed power use, $\rho = 0.318$, $p < .001$, indicating that observations characterised by comparatively greater L2 data cache load miss rates tended to

exhibit greater measured power use.

In contrast, under tiered execution, a statistically significant but comparatively weak positive monotonic relationship was observed, $\rho = 0.093$, $p < .001$, indicating that observations characterised by comparatively greater L2 data cache load miss rates tended to exhibit only marginally greater measured power use.

Collectively, these findings indicate that the weak pooled relationship masks important differences between execution environments. Although pooled observations suggested only a negligible positive association between L2 data cache load miss rate and observed power use, decomposition according to execution mode revealed a substantially stronger positive relationship under interpretive execution. This indicates that the relationship between L2 data cache load miss rate and power use is more pronounced under interpretive execution than is apparent from the pooled analysis alone.

These findings suggest that L2 data cache load miss rate may represent a more meaningful microarchitectural correlate of instantaneous JRE JVM power use than previously examined microarchitectural performance event measures. While the overall pooled relationship remained weak, the moderate positive relationship observed under interpretive execution indicates that deeper memory hierarchy access patterns may be more closely associated with observed power use than the front-end execution characteristics previously examined in this study.

L2 Data Cache Load Miss Rate and Energy Consumption Relationships

To assess whether L2 data cache load miss rates were associated with observed JRE JVM energy consumption, we first examine the relationship between both variables through the scatterplot presented in Figure 5.21. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both L2 data cache load miss rate and observed energy consumption statistically significantly deviated from normality, L2 data cache load miss rate ($W = 0.872$, $p < .001$),

and energy consumption ($W = 0.629$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between L2 data cache load miss rate and observed energy consumption.

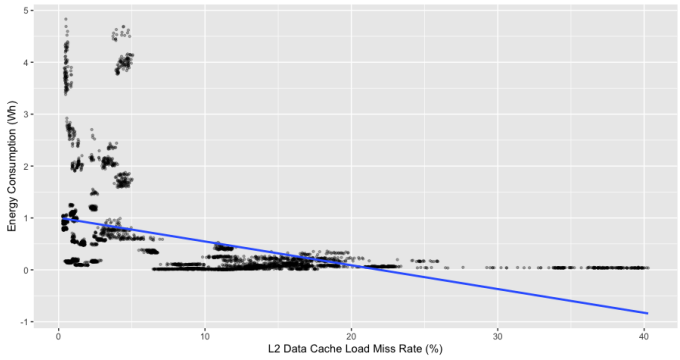


Figure 5.21: L2 Data Cache Load Miss Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed L2 data cache load miss rates and measured JRE JVM energy consumption across all pooled observations.

The results indicated a statistically significant strong negative monotonic relationship between L2 data cache load miss rate and observed JRE JVM energy consumption, $\rho = -0.604$, $p < .001$, suggesting that observations characterised by comparatively greater L2 data cache load miss rates also tended to exhibit lower measured cumulative energy consumption.

At first inspection, this result appears to suggest that greater L2 data cache load miss rates are associated with reduced cumulative energy consumption. However, given the substantial structural differences previously observed between interpretive and tiered execution, additional analysis was conducted to determine whether execution mode contributed to the observed pooled relationship.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations. Shapiro-Wilk tests again indicated that L2 data cache load miss rate distributions statistically significantly deviated from normality under both interpretive execution ($W = 0.753$, $p < .001$) and tiered execution ($W = 0.824$, $p < .001$).

When decomposed according to execution mode, substantially different relationships emerged. Under interpretive execution, a statistically significant but negligible negative monotonic relationship was observed between L2 data cache load miss rate and observed energy consumption, $\rho = -0.047$, $p = .026$, indicating little practically meaningful relationship between both variables.

In contrast, under tiered execution, a statistically significant moderate positive monotonic relationship was observed, $\rho = 0.390$, $p < .001$, indicating that observations characterised by comparatively greater L2 data cache load miss rates tended to exhibit greater measured cumulative energy consumption.

Collectively, these findings indicate that the strong negative pooled relationship substantially masks fundamentally different underlying relationships once execution mode is controlled for. Although pooled observations initially suggested that greater L2 data cache load miss rates were associated with reduced energy consumption, decomposition according to execution mode revealed that this relationship was primarily attributable to structural differences between execution environments. In particular, tiered execution simultaneously exhibited substantially greater L2 data cache load miss rates together with substantially lower cumulative energy consumption relative to interpretive execution, artificially producing the strong negative pooled association.

These findings suggest that L2 data cache load miss rate does not independently constitute a globally consistent determinant of cumulative JRE JVM energy consumption. Rather, the relationship between L2 data cache load miss rate and energy consumption appears highly dependent upon execution mode, with tiered execution exhibiting a comparatively stronger positive within-mode relationship that is obscured when observations are aggregated across fundamentally different execution environments.

Summary

The descriptive analysis indicated that L2 data cache load miss rates remained broadly stable across OpenJDK JRE JVM versions, with only relatively

minor differences observed between JRE implementations. Although OpenJDK JRE version 13 exhibited the lowest average and median L2 data cache load miss rates, these differences remained comparatively small relative to the overall spread of observations and did not materially distinguish OpenJDK JRE version 13 from the remaining JRE versions. This contrasts with the previously observed branch misprediction and L1 data cache load miss rate results, both of which exhibited clearer systematic differences across JRE versions.

When decomposed according to execution mode, substantial differences emerged between tiered and interpretive execution.

Unlike branch misprediction and L1 data cache load miss rates, where interpretive execution consistently produced comparatively higher miss rates, L2 data cache load miss rates exhibited the opposite pattern, with tiered execution producing substantially greater L2 cache miss rates across all JRE versions. On average, L2 data cache load miss rates under tiered execution were approximately five times greater than those observed under interpretive execution, indicating that execution mode constituted the dominant source of variation in observed L2 data cache load miss rates.

Inferential statistical analysis supported these observations. No statistically significant differences in L2 data cache load miss rates were observed between JRE versions, confirming that OpenJDK JRE JVM version was not associated with meaningful differences in L2 cache miss activity. This indicates that, unlike branch misprediction and L1 data cache load miss behaviour, L2 cache miss activity remained comparatively stable across JRE implementations.

Correlation analysis indicated that L2 data cache load miss rates exhibited only a weak positive relationship with observed JRE JVM power use when all observations were pooled. However, decomposition according to execution mode revealed substantially different within-mode relationships, with interpretive execution exhibiting a moderate positive relationship and tiered execution exhibiting only a weak positive relationship. These findings indicate that the pooled analysis

masked a stronger underlying association between L2 data cache load miss rate and instantaneous power use under interpretive execution.

A substantially more complex pattern was observed for energy consumption. While pooled observations indicated a strong negative relationship between L2 data cache load miss rate and cumulative energy consumption, decomposition according to execution mode revealed that this relationship was largely attributable to structural differences between tiered and interpretive execution environments. Within interpretive execution, only a negligible relationship was observed, whereas tiered execution exhibited a moderate positive relationship. This indicates that the strong pooled negative association primarily reflected structural differences between execution mode categories rather than a globally consistent relationship between L2 data cache load miss rate and cumulative energy consumption.

Collectively, these findings suggest that L2 data cache load miss rate exhibits more substantial within-mode associations with observed JRE JVM power use and energy consumption than the previously examined branch misprediction and L1 data cache load miss rate measures. However, similar to those earlier results, the interpretation of these relationships remains strongly dependent upon execution mode. These findings further reinforce the broader observation emerging from this study that microarchitectural performance event measures cannot be interpreted independently of execution environment, as pooled analyses may obscure fundamentally different underlying execution-mode-specific relationships.

5.6.4 Page Fault Relationships

JRE Effects

Descriptive Analysis

To evaluate the relationship between JRE JVM version and page fault activity, we first examine the distribution of page fault rates observed across the six OpenJDK JRE JVMs. Table 5.16 presents descriptive statistics associated with page fault rates, reported for all pooled observations and further decomposed

according to execution mode.

Across all pooled observations, page fault rates remained consistently low across all JRE versions, with relatively limited variation observed between distributions. Mean page fault rates ranged from a low of 0.06% for OpenJDK JRE versions 10 and 13 to a high of 0.09% for OpenJDK JRE version 9, representing an absolute difference of approximately 0.03 percentage points. Median page fault rates followed a similarly constrained distribution, ranging between 0.04% and 0.06%. Overall, all six JRE versions exhibited highly comparable page fault distributions, indicating minimal variation in page fault activity across JRE versions.

Variability within each JRE distribution remained comparatively low. Standard deviation values ranged between 0.06 and 0.13 percentage points, while interquartile ranges varied only marginally between 0.07 and 0.09 percentage points. This indicates a relatively compact distribution of page fault rates across benchmark executions. Although OpenJDK JRE version 9 exhibited the highest observed variability, largely due to several comparatively large outlying observations, the overall magnitude of variation remained small across all JRE versions.

Table 5.16: Page Fault Distributions: Descriptive statistics associated with each of the six OpenJDK JRE JVMs, reported for all pooled observations, and grouped by execution mode. Detailing the mean (**M**), median ($\mathbf{M}_d$), standard deviation (**SD**), interquartile range (**IQR**), and the sample size (**n**) associated with page fault rates.

	Pooled Observations (%)							**Execution Mode (%)**							
JRE	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**	**Mode**	**Min**	**Max**	**M**	**M_d**	**SD**	**IQR**	**n**
j09	0.00	2.47	0.09	0.05	0.13	0.09	864	HS	0.01	2.47	0.10	0.07	0.16	0.11	480
								INT	0.00	0.34	0.06	0.03	0.07	0.07	384
j10	0.00	0.37	0.06	0.04	0.07	0.07	768	HS	0.01	0.37	0.07	0.05	0.06	0.06	384
								INT	0.00	0.31	0.06	0.04	0.07	0.08	384
j11	0.00	0.35	0.07	0.05	0.07	0.07	768	HS	0.01	0.35	0.07	0.05	0.06	0.07	384
								INT	0.00	0.30	0.06	0.03	0.07	0.08	384
j12	0.00	0.37	0.07	0.04	0.07	0.07	736	HS	0.01	0.37	0.07	0.05	0.07	0.07	368
								INT	0.00	0.29	0.06	0.03	0.07	0.07	368
j13	0.00	0.37	0.06	0.04	0.06	0.07	736	HS	0.01	0.37	0.06	0.05	0.06	0.06	368
								INT	0.00	0.29	0.06	0.04	0.07	0.07	368
j14	0.00	0.38	0.08	0.06	0.08	0.09	736	HS	0.01	0.38	0.09	0.06	0.08	0.08	368
								INT	0.00	0.32	0.08	0.05	0.08	0.12	368
Avg.	0.00	0.72	0.07	0.05	0.08	0.08	768	HS Avg.	0.01	0.72	0.08	0.06	0.08	0.08	392
						Total:	**4608**	INT Avg.	0.00	0.31	0.06	0.04	0.07	0.08	376
													HS Total:		**2352**
													INT Total:		**2256**

When decomposed according to execution mode, relatively small differences emerged between tiered execution (HS) and interpretive execution (INT). Across all JRE versions, page fault rates under tiered execution were marginally greater, with mean values ranging between 0.06% for OpenJDK JRE version 13 and 0.10% for OpenJDK JRE version 9. Median values under tiered execution ranged between 0.05% and 0.07%. Across all tiered execution observations, the average page fault rate was 0.08%, with an associated median of 0.06%.

Interpretive execution exhibited slightly lower page fault rates across most JRE versions. Mean page fault rates ranged between 0.06% and 0.08%, while median values ranged from 0.03% to 0.05%. Across all observations, interpretive execution exhibited an average page fault rate of 0.06%, with an associated median of 0.04%.

Comparing execution modes, tiered execution consistently produced marginally greater page fault rates relative to interpretive execution across all JRE versions. However, unlike the substantial execution mode differences previously observed for branch misprediction rates and L2 data load miss rates, the absolute magnitude of separation between tiered and interpretive execution distributions remained comparatively small. Overall, page fault rates exhibited the lowest degree of variation across both JRE versions and execution modes among all microarchitectural performance event measures examined in this study.

Inferential Statistical Analysis

In regard to all OpenJDK JRE JVM versions, the results of Shapiro-Wilk tests of normality indicated that page fault rate distributions statistically significantly deviated from normality in all cases (all $p < .001$). Consequently, non-parametric Kruskal-Wallis rank sum tests were conducted to assess whether page fault rates differed significantly between JRE versions.

When observations were pooled across both execution modes, the results of the Kruskal-Wallis test indicated statistically significant differences in page fault rate distributions between JRE versions, $\chi^2 = 42.398$, $df = 5$, $p < .001$.

Subsequent pairwise Wilcoxon rank sum tests indicated that OpenJDK JRE version 9 exhibited statistically significant differences relative to OpenJDK JRE versions 10 ($p = .030$), 12 ($p = .030$), and 13 ($p = .004$), while OpenJDK JRE version 14 exhibited statistically significant differences relative to OpenJDK JRE versions 10 ($p < .001$), 11 ($p < .001$), 12 ($p < .001$), and 13 ($p < .001$). No statistically significant differences were observed between OpenJDK JRE versions 9 and 11 ($p = .088$), versions 9 and 14 ($p = 1.000$), or among OpenJDK JRE versions 10, 11, 12, and 13 (all $p = 1.000$).

When interpretive execution was considered independently, statistically significant differences in page fault rate distributions were again observed between JRE versions. Pairwise Wilcoxon rank sum tests indicated that OpenJDK JRE version 14 exhibited statistically significant differences relative to OpenJDK JRE versions 9 ($p = .006$), 11 ($p = .041$), 12 ($p = .007$), and 13 ($p = .009$). No statistically significant differences were observed between OpenJDK JRE versions 9 through 13 (all $p = 1.000$), while the comparison between OpenJDK JRE versions 10 and 14 approached but did not reach statistical significance ($p = .060$). These findings indicate a high degree of consistency in page fault activity across OpenJDK JRE versions under interpretive execution, with divergence emerging only for OpenJDK JRE version 14.

When tiered execution was considered independently, statistically significant differences were more widely observed across JRE versions. Pairwise Wilcoxon rank sum tests indicated that OpenJDK JRE version 9 differed significantly from OpenJDK JRE versions 10 ($p < .001$), 11 ($p < .001$), 12 ($p < .001$), and 13 ($p < .001$), while OpenJDK JRE version 14 differed significantly from OpenJDK JRE versions 10 ($p = .002$), 11 ($p = .024$), 12 ($p = .034$), and 13 ($p < .001$). No statistically significant differences were observed between OpenJDK JRE versions 9 and 14 ($p = 1.000$), or among OpenJDK JRE versions 10, 11, 12, and 13 (all $p = 1.000$).

Collectively, these findings indicate that page fault rates exhibited statistically

significant but comparatively modest variation across JRE versions. Under interpretive execution, page fault activity remained highly stable across OpenJDK JRE versions 9 through 13, with statistically significant divergence observed only for OpenJDK JRE version 14. Under tiered execution, greater variation was observed, with OpenJDK JRE versions 9 and 14 exhibiting statistically significant differences relative to intermediate JRE versions. However, despite the presence of statistically significant differences, the absolute magnitude of these variations remained small, suggesting that page fault activity is broadly consistent across JRE implementations and unlikely to represent a substantively important driver of the power use differences observed between JRE versions.

Page Fault Rate and Power Use Relationships

To assess whether page fault rates were associated with observed JRE JVM power use, we first examine the relationship between both variables through the scatterplot presented in Figure 5.22. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both page fault rate and observed power use statistically significantly deviated from normality, page fault rate ($W = 0.647$, $p < .001$), and power use ($W = 0.967$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between page fault rate and observed power use.

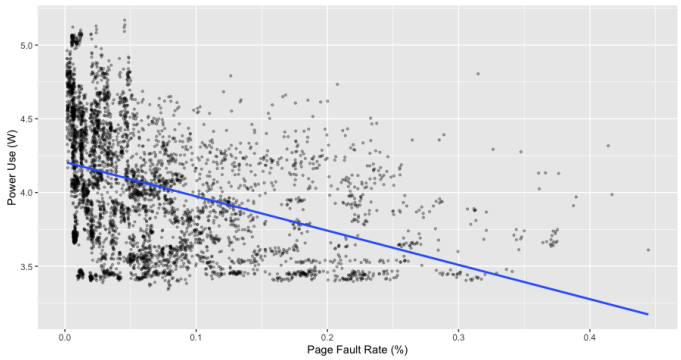


Figure 5.22: Page Fault Rate and Power Use Relationship: Scatterplot depicting the relationship between observed page fault rates and measured JRE JVM power use across all pooled observations.

The results indicated a statistically significant negative monotonic relationship between page fault rate and observed JRE JVM power use, $\rho = -0.479$, $p < .001$, suggesting that observations characterised by comparatively greater page fault rates also tended to exhibit comparatively lower measured power use.

The magnitude of the observed correlation coefficient indicates a moderate association between both variables, suggesting that variation in page fault rate was associated with a non-trivial proportion of the overall variation observed in JRE JVM power use.

Unlike the previously examined microarchitectural performance event measures, the observed relationship between page fault rate and power use was comparatively substantial. However, the negative direction of the relationship suggests that observations characterised by comparatively greater page fault activity tended to exhibit lower instantaneous power use, indicating that this relationship may reflect broader execution environment differences rather than a direct causal relationship between page fault activity and power use. These findings therefore suggest that page fault behaviour may be associated with runtime conditions that systematically influence observed power use rather than independently determining instantaneous JRE JVM power consumption.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, similarly strong negative relationships remained evident under both execution environments. Under interpretive execution, a statistically significant moderate negative monotonic relationship was observed between page fault rate and observed power use, $\rho = -0.589$, $p < .001$, indicating that observations characterised by comparatively greater page fault rates tended to exhibit comparatively lower measured power use.

Under tiered execution, a statistically significant but comparatively weaker negative monotonic relationship was also observed, $\rho = -0.325$, $p < .001$, again

indicating that observations characterised by comparatively greater page fault rates tended to exhibit comparatively lower measured power use.

Collectively, these findings indicate that the negative relationship between page fault rate and observed power use was not solely attributable to pooled differences between execution environments, but remained consistently observable under both interpretive and tiered execution conditions. The substantially stronger relationship observed under interpretive execution indicates that the association between page fault activity and power use is more pronounced when execution occurs without sustained JIT compilation optimisation.

These findings suggest that page fault activity represents a more substantial correlate of instantaneous JRE JVM power use than several of the previously examined microarchitectural performance event measures. Unlike branch misprediction rates, L1 data cache load miss rates, and L2 data cache load miss rates, elevated page fault activity was consistently associated with lower observed instantaneous power use. This suggests that execution conditions characterised by greater operating system-level virtual memory activity may coincide with reduced processor execution intensity, thereby contributing to comparatively lower observed power consumption.

Page Fault Rate and Energy Consumption Relationships

To assess whether page fault rates were associated with observed JRE JVM energy consumption, we first examine the relationship between both variables through the scatterplot presented in Figure 5.23. To determine the appropriate inferential procedure, the distributional characteristics of both variables were subsequently assessed. Shapiro-Wilk tests of normality indicated that both page fault rate and observed energy consumption statistically significantly deviated from normality, page fault rate ($W = 0.647$, $p < .001$), and energy consumption ($W = 0.629$, $p < .001$). Consequently, a non-parametric Spearman rank-order correlation test was conducted to evaluate the relationship between page fault rate and observed

energy consumption.

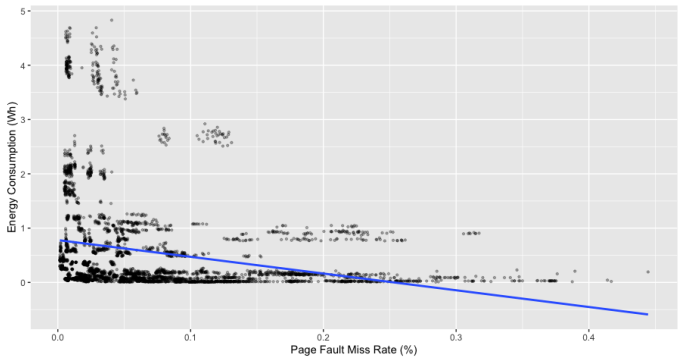


Figure 5.23: Page Fault Rate and Energy Consumption Relationship: Scatterplot depicting the relationship between observed page fault rates and measured JRE JVM energy consumption across all pooled observations.

The results indicated a statistically significant negative monotonic relationship between page fault rate and observed JRE JVM energy consumption, $\rho = -0.371$, $p < .001$, suggesting that observations characterised by comparatively greater page fault rates also tended to exhibit comparatively lower measured energy consumption.

The magnitude of the observed correlation coefficient indicates a moderate association between both variables, suggesting that variation in page fault rate was associated with a non-trivial proportion of the overall variation observed in JRE JVM energy consumption.

Unlike the previously examined microarchitectural performance event measures, the observed relationship between page fault rate and energy consumption was comparatively substantial. The negative direction of the relationship suggests that observations characterised by comparatively greater page fault activity tended to coincide with comparatively lower total energy consumption, indicating that runtime conditions associated with elevated page fault activity may also correspond with reduced overall computational demand during execution.

To determine whether execution mode contributed to the observed pooled relationship, additional Spearman rank-order correlation tests were conducted separately for tiered and interpretive execution observations.

When decomposed according to execution mode, similarly consistent negative relationships remained evident under both execution environments. Under interpretive execution, a statistically significant moderate negative monotonic relationship was observed between page fault rate and observed energy consumption, $\rho = -0.360$, $p < .001$, indicating that observations characterised by comparatively greater page fault rates tended to exhibit comparatively lower measured energy consumption.

Under tiered execution, a statistically significant moderate negative monotonic relationship was also observed, $\rho = -0.432$, $p < .001$, again indicating that observations characterised by comparatively greater page fault rates tended to exhibit comparatively lower measured energy consumption.

Collectively, these findings indicate that the negative relationship between page fault rate and observed energy consumption remained consistently observable across both interpretive and tiered execution environments. Unlike several previously examined microarchitectural performance event measures, decomposition according to execution mode did not materially alter either the strength or direction of the observed relationship, suggesting that execution mode did not substantially mediate the relationship between page fault activity and energy consumption.

These findings suggest that page fault activity represents one of the more substantial correlates of observed JRE JVM energy consumption among the microarchitectural performance event measures examined in this study. Elevated page fault activity was consistently associated with comparatively lower observed energy consumption, suggesting that runtime conditions characterised by greater operating system-level virtual memory activity may systematically coincide with execution conditions associated with lower total energy expenditure during program execution.

Summary

The descriptive analysis indicated a high degree of consistency in page fault

activity across OpenJDK JRE versions. Across all pooled observations, page fault rates remained uniformly low, with only minimal absolute variation observed between JRE versions. Although tiered execution consistently exhibited marginally greater page fault rates relative to interpretive execution, the magnitude of these execution mode differences remained comparatively small when contrasted with the substantially larger execution mode differences previously observed for branch misprediction rates and L2 data cache load miss rates.

Inferential analysis broadly supported these observations. Although statistically significant differences in page fault rate distributions were identified between certain JRE versions, the absolute magnitude of these differences remained limited. Under interpretive execution, page fault activity remained highly stable across OpenJDK JRE versions 9 through 13, with statistically significant divergence observed only for OpenJDK JRE version 14. Under tiered execution, statistically significant differences were more widely observed, primarily reflecting comparatively elevated page fault rates for OpenJDK JRE versions 9 and 14 relative to intermediate JRE versions. Collectively, these findings indicate that page fault activity remains broadly consistent across JRE implementations and is unlikely to represent a substantively important source of the JRE version differences previously observed in measured power use.

In contrast to the comparatively weak relationships previously observed for branch misprediction rates, L1 data cache load miss rates, and L2 data cache load miss rates, page fault activity exhibited substantially stronger relationships with both observed power use and energy consumption. For power use, a statistically significant moderate negative monotonic relationship was observed across pooled observations, with elevated page fault rates consistently associated with comparatively lower measured power use. This relationship remained evident under both execution environments and was particularly pronounced under interpretive execution, where the strongest inverse relationship was observed.

A similarly consistent pattern emerged for energy consumption. Elevated page

fault rates were associated with comparatively lower measured energy consumption, with moderate negative relationships observed across pooled observations and under both interpretive and tiered execution. Unlike several previously examined microarchitectural performance event measures, decomposition according to execution mode did not materially alter the strength or direction of the observed energy relationship, indicating that the inverse association between page fault activity and energy consumption remained structurally consistent across both execution environments.

In relation to the broader microarchitectural analyses conducted in this study, these findings indicate that page fault activity exhibits substantially stronger associations with both power use and energy consumption than several processor-level execution characteristics previously examined. Unlike branch misprediction rates and cache miss rates, which generally exhibited comparatively weak relationships with observed power use and energy consumption, elevated page fault activity was consistently associated with lower observed power use and lower total energy consumption. This suggests that operating system-level memory management behaviour may represent a more meaningful correlate of JRE JVM runtime power and energy characteristics than several lower-level processor execution behaviours examined throughout this study.

5.6.5 Summary

Across all microarchitectural performance event analyses conducted within this section, the results demonstrated substantial variation in the extent to which individual processor-level execution characteristics were associated with observed JRE JVM power use and energy consumption. Although statistically significant differences were identified across several microarchitectural performance event distributions, the magnitude, consistency, and practical importance of these relationships varied considerably between the four microarchitectural performance event measures examined.

For branch misprediction rates, descriptive and inferential analyses indicated systematic variation across OpenJDK JRE JVM versions, with OpenJDK JRE version 13 consistently exhibiting substantially lower branch misprediction activity relative to all remaining JRE versions. Substantial differences were also observed between interpretive and tiered execution, with interpretive execution consistently producing branch misprediction rates approximately three times greater than those observed under tiered execution. However, despite these substantial differences in branch misprediction behaviour, branch misprediction rate exhibited no meaningful relationship with observed power use, while the initially strong pooled relationship with energy consumption was shown to be largely attributable to execution mode aggregation effects.

A broadly similar pattern was observed for L1 data cache load miss rates. OpenJDK JRE version 13 again consistently exhibited comparatively reduced L1 data cache miss activity relative to the remaining JRE versions, while interpretive execution consistently produced moderately greater L1 miss rates than tiered execution. However, relationships between L1 data cache load miss rates and both observed power use and energy consumption remained comparatively weak and were highly dependent upon decomposition according to execution mode. These findings indicate that L1 data cache load miss behaviour does not independently constitute a strong determinant of either observed power use or cumulative energy consumption.

In contrast, L2 data cache load miss rates exhibited substantially different execution characteristics. Unlike branch misprediction and L1 data cache miss activity, no statistically significant differences were observed between OpenJDK JRE JVM versions. Furthermore, tiered execution consistently exhibited substantially greater L2 data cache load miss rates relative to interpretive execution, producing average miss rates approximately five times greater than those observed under interpretive execution. Although pooled relationships between L2 data cache load miss rate and observed power use and energy consumption remained compara-

tively weak, decomposition according to execution mode revealed a substantially stronger positive relationship between L2 miss activity and observed power use under interpretive execution. These findings suggest that deeper memory hierarchy access behaviour may represent a more meaningful correlate of JRE JVM runtime power characteristics than previously examined front-end execution measures.

Page fault activity exhibited the most distinct relationship patterns observed among all microarchitectural performance event measures examined. Although page fault rates remained uniformly low across all OpenJDK JRE JVM versions and exhibited comparatively limited variation between both JRE versions and execution modes, correlation analyses revealed consistent moderate negative relationships with both observed power use and energy consumption. Unlike branch misprediction rates, L1 data cache load miss rates, and L2 data cache load miss rates, decomposition according to execution mode did not materially alter the observed relationships between page fault activity and either power use or energy consumption. Elevated page fault rates were consistently associated with comparatively lower observed power use and lower cumulative energy consumption under both interpretive and tiered execution.

Collectively, these findings indicate that microarchitectural performance event measures do not contribute equally to explaining observed variation in JRE JVM power use and energy consumption. Front-end processor execution characteristics, including branch misprediction rates and L1 data cache load miss rates, generally exhibited comparatively weak and inconsistent relationships with both outcome variables. In contrast, deeper memory hierarchy behaviour and operating system-level memory management activity, represented by L2 data cache load miss rates and page fault activity respectively, exhibited comparatively stronger and more consistent associations with observed runtime power and energy characteristics.

More broadly, decomposition according to execution mode repeatedly demonstrated that interpretive and tiered execution frequently altered both observed microarchitectural behaviour and the corresponding relationships between microar-

chitectural activity and measured power use and energy consumption. For several microarchitectural performance event measures, execution mode consistently emerged as an important structural factor governing the observed relationships. However, these effects were not uniform across all measures examined, indicating that the influence of execution mode depends substantially upon the underlying execution characteristic being observed.

In relation to **SRQ1.5**, the findings indicate that microarchitectural performance event measures exhibited substantially different relationships with observed JRE JVM power use and energy consumption. Branch misprediction rates and L1 data cache load miss rates exhibited comparatively weak and inconsistent relationships with both outcome variables, suggesting limited explanatory importance. In contrast, L2 data cache load miss rates exhibited stronger execution mode dependent relationships, while page fault activity exhibited the strongest and most consistent associations with both power use and energy consumption. Collectively, these findings indicate that deeper memory hierarchy behaviour and operating system-level memory management activity represent more meaningful correlates of JRE JVM runtime power and energy characteristics than front-end processor execution characteristics.

In relation to **SRQ1.6**, the findings demonstrate that interpretive and tiered execution consistently produced substantially different microarchitectural performance event measures across multiple performance event measures. Interpretive execution exhibited substantially greater branch misprediction rates and moderately greater L1 data cache load miss rates, while tiered execution consistently produced substantially greater L2 data cache load miss rates and marginally elevated page fault activity. Furthermore, decomposition according to execution mode frequently altered the strength and direction of the observed relationships between microarchitectural performance event measures and both power use and energy consumption. Collectively, these findings indicate that execution mode represents an important structural factor influencing both observed processor-

level execution behaviour and the extent to which individual microarchitectural performance characteristics are associated with JRE JVM power use and energy consumption under differing runtime conditions.

Section Research Question Findings

SRQ1.5: Microarchitectural performance event measures exhibited heterogeneous relationships with JRE JVM power use and energy consumption. Branch misprediction rates and L1 data cache load miss rates demonstrated comparatively weak and inconsistent relationships with both outcome variables, while L2 data cache load miss rates exhibited stronger execution mode dependent relationships with power use. Page fault activity exhibited the strongest and most consistent relationships, with elevated page fault rates consistently associated with lower observed power use and lower energy consumption.

SRQ1.6: Differences between interpretive and tiered execution substantially influenced observed microarchitectural behaviour and frequently altered the relationships between microarchitectural performance event measures and JRE JVM power use and energy consumption. Interpretive execution consistently exhibited substantially greater branch misprediction rates and L1 data cache load miss rates, whereas tiered execution consistently exhibited substantially greater L2 data cache load miss rates. However, the influence of execution mode was not uniform across all measures, with page fault activity exhibiting comparatively stable relationships across both execution environments.

5.7 How this Chapter Answers RQ1

This chapter addressed **Research Question 1 (RQ1)**, which examined how variations in JRE source version, build toolchain compiler version, JVM execution mode, garbage collector variant, workload category, and microarchitectural performance event measures influence JRE JVM power use and energy consumption. Collectively, the analyses presented throughout this chapter demonstrate that these factors do not contribute equally to observed variation in power use and energy consumption. Instead, a clear hierarchy of influence emerged, with workload category and execution mode representing the most substantial determinants, while several system-level configuration variables exerted comparatively limited effects.

Across the system-level configuration variables examined, **OpenJDK JRE source version exhibited only limited influence on overall runtime behaviour**. Statistically significant differences in power use were observed between JRE versions, but these differences were almost entirely attributable to the anomalous behaviour associated with OpenJDK JRE version 13. Once this version was excluded, no statistically significant differences remained between the remaining JRE releases. In contrast, JRE source version demonstrated no consistent statistically significant effect on overall energy consumption, indicating that variation between JRE releases contributed minimally to observed differences in total runtime energy consumption.

Similarly, **GNU GCC build toolchain compiler version exerted comparatively limited effects on both power use and energy consumption**. Although modest differences in power use were observed, particularly for GCC version 5, these effects were comparatively small and inconsistent across execution modes. No statistically significant differences in energy consumption were observed across compiler versions, indicating that compiler source version represented only a minor determinant of overall runtime power and energy characteristics.

The analyses also demonstrated that **garbage collector variant exerted**

relatively modest influence on observed runtime behaviour. While statistically significant differences in power use were observed between garbage collector variants, these differences were comparatively small in magnitude and primarily associated with specific collector implementations under tiered execution. In contrast, no statistically significant differences in energy consumption were observed between garbage collector variants, indicating that memory management strategy contributed only marginally to overall variation in runtime energy consumption within the experimental environment considered.

In contrast to these comparatively weak system-level effects, **workload category emerged as one of the strongest determinants of both power use and energy consumption**. Statistically significant differences were observed across all workload categories for both outcome variables, with workload-specific computational characteristics strongly influencing overall runtime behaviour. Particularly pronounced differences were observed for energy consumption, where database-oriented workloads consistently exhibited substantially greater energy consumption than workloads associated with the functional, scala, and web workload categories. These findings indicate that workload category represent a major source of variation in both power use and energy consumption.

The analyses further demonstrated that **differences between interpretive and tiered execution constituted the principal determinant of observed energy consumption and an important secondary determinant of power use**. Across all experimental conditions, tiered execution consistently exhibited substantially lower energy consumption than interpretive execution, with interpretive execution frequently consuming an order of magnitude more total energy than tiered execution. In contrast, differences in power use between execution modes were comparatively modest and frequently statistically non-significant. Ratio analyses demonstrated that the substantial energy consumption advantage associated with tiered execution could not be explained by reductions in power use alone, indicating that the energy consumption advantage was principally

attributable to reduced execution time rather than lower instantaneous power draw.

Finally, the microarchitectural analyses demonstrated that **lower-level computational characteristics were differentially associated with power use and energy consumption**. Branch misprediction rates, cache load miss behaviour, and page fault activity varied systematically across JRE versions and execution modes, indicating that differences in runtime configuration influenced underlying processor-level execution characteristics. However, an important distinction emerged between front-end and memory-hierarchy related execution behaviour. Measures associated with front-end execution activity, particularly branch misprediction rates and L1 data cache load miss rates, exhibited comparatively weak relationships with observed power use and energy consumption. In contrast, deeper memory hierarchy behaviour, particularly L2 data cache load miss rates, demonstrated substantially stronger associations with both outcome variables, suggesting that memory subsystem behaviour represented a more important lower-level predictor of runtime power use and energy consumption characteristics.

aken collectively, these findings demonstrate that **JRE/JVM power use and energy consumption are more strongly associated with workload-dependent computational characteristics and JVM execution mode than with differences associated with JRE source version, build toolchain compiler version, or garbage collector variant**. Among all factors examined, workload category and execution mode emerged as the dominant determinants of observed variation, while microarchitectural analyses indicated that deeper memory hierarchy behaviour provides important explanatory insight into the lower-level computational mechanisms associated with these observed differences.

In answer to **RQ1**, the findings indicate that while software configuration choices exert measurable but generally limited effects on JRE JVM power use and energy consumption, workload category and JVM execution strategy constitute the principal determinants of overall runtime power and energy consumption be-

haviour. Furthermore, observed differences in these outcomes are associated more strongly with deeper memory subsystem activity than with front-end execution characteristics, indicating that microarchitectural performance event measures provide an important explanatory mechanism linking software execution behaviour to observed power and energy consumption outcomes.

Overall Research Question Findings

RQ1: JRE JVM power use and energy consumption were influenced primarily by workload category and JVM execution mode, with workload-dependent computational characteristics representing the strongest determinant of observed variation. OpenJDK JRE source version, GNU GCC compiler version, and garbage collector variant exerted comparatively limited effects, with OpenJDK JRE version 13 representing the principal exception. Tiered execution consistently reduced energy consumption relative to interpretive execution, primarily due to reduced execution time rather than lower power use. Microarchitectural analyses further indicated that deeper memory hierarchy activity, particularly L2 data cache load miss activity, exhibited stronger associations with observed power use and energy consumption than branch misprediction and L1 data cache load miss activity.

REFERENCES

[1] Raspberry Pi Ltd, "Raspberry Pi 4 Model B Specifications," [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/, [Accessed: Sep. 15, 2026].

[2] Jetperch LLC, "Joulescope DC Energy Analyser," [Online]. Available: https://www.joulescope.com, [Accessed: Sep. 28, 2022].

[3] TP-Link Technologies Co., Ltd., *TL-SG116 16-Port Gigabit Desktop Switch Datasheet*, TP-Link Technologies Co., Ltd., 2026, [Accessed: Jun. 4, 2026]. [Online]. Available: https://www.tp-link.com/us/support/download/tl-sg116/

[4] J. Lambert, "PhD Data Joulescope Acquisition Script," [Online]. Available: https://github.com/jonathan-lambert/, 2026, gitHub repository. [Accessed: Jun. 7, 2026].

[5] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the JVM," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 31–47. [Online]. Available: https://doi.org/10.1145/3314221.3314637

[6] B. Venners, *Inside the Java Virtual Machine*. McGraw-Hill, 1998.

[7] K. J. Gough and D. Corney, "Evaluating the Java Virtual Machine as a Target for Languages Other Than Java," in *Modular Programming Languages*, W. Weck and J. Gutknecht, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 278–290.

[8] Cloud Foundry Foundation, "Top languages for enterprise application development," Cloud Foundry Research Report. [Online]. Available: https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf, 2018, [Accessed: Apr. 24, 2021].

[9] Azul Systems, Inc., "Java at speed: The Azul Zing JVM optimized for modern hardware," [Online]. Available: https://www.azul.com/resources-hub/whitepapers, [Accessed: Apr. 27, 2021].

[10] S. Cass, "Top programming languages 2020," IEEE Spectrum. [Online]. Available: https://spectrum.ieee.org/top-programming-language-2020, 2020, [Accessed: Apr. 29, 2021].

[11] TIOBE Software BV, "TIOBE Index for April 2021," [Online]. Available: https://www.tiobe.com/tiobe-index/, 2021, [Accessed: Apr. 23, 2021].

[12] W. Gu, N. A. Burns, M. T. Collins, and W. Y. P. Wong, "The evolution of a high-performing Java Virtual Machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 135–150, 2000.

[13] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, p. 97–113, 06 2003. [Online]. Available: https://doi.org/10.1145/857076.857077

[14] D. Gregg, J. F. Power, and J. Waldron, "Platform independent dynamic Java Virtual Machine analysis: the Java Grande Forum benchmark suite," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 459–484, 2003. [Online]. Available: http://mural.maynoothuniversity.ie/10224/

[15] S. L. P. Jones, N. Ramsey, and F. Reig, "C–: A portable assembly language that supports garbage collection," in *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, ser. Lecture Notes in Computer Science, G. Nadathur, Ed., vol. 1702. Berlin, Heidelberg: Springer, 1999, pp. 1–28. [Online]. Available: https://doi.org/10.1007/10704567_1

[16] C. Daly, J. Horgan, J. Power, and J. Waldron, "Platform independent dynamic Java Virtual Machine analysis: the Java grande forum benchmark suite," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, 2001, pp. 106–115.

[17] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java Just in Time," *IEEE Micro*, vol. 17, no. 3, p. 36–43, 05 1997. [Online]. Available: https://doi.org/10.1109/40.591653

[18] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. mei W. Hwu, "A study of the cache and branch performance issues with running

Java on current hardware platforms," in *In Proceedings of COMPCON*. Piscataway, NJ, USA: IEEE, 1997, pp. 211–216.

[19] D. Gollapudi, "Java Platform, Standard Edition JRockit to HotSpot Migration Guide," Oracle. [Online]. Available: https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/index.html, Jun. 2015, release 8, E63235-01. [Accessed: Apr. 20, 2022].

[20] P. Costanza, C. Herzeel, and W. Verachtert, "A comparison of three programming languages for a full-fledged next-generation sequencing tool," *BMC Bioinformatics*, vol. 20, no. 1, p. 301, 2019. [Online]. Available: https://doi.org/10.1186/s12859-019-2903-5

[21] B. Oancea, I. G. Rosca, T. Andrei, and A. I. Iacob, "Evaluating Java performance for linear algebra numerical computations," *Procedia Computer Science*, vol. 3, pp. 474–478, 2011, world Conference on Information Technology. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050910004552

[22] L. Lin, Z. Xu, H. Huan, Z. Jian, and L. Li-Xin, "An empirical comparison of implementation efficiency of iterative and recursive algorithms of fast fourier transform," in *Machine Learning and Intelligent Communications*, M. Guan and Z. Na, Eds. Cham: Springer International Publishing, 2021, pp. 73–81.

[23] M. Källén and T. Wrigstad, "Performance of an OO compute kernel on the JVM: Revisiting Java as a language for scientific computing applications," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 144–156. [Online]. Available: https://doi.org/10.1145/3357390.3361026

[24] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 2 - high-performance embedded computing," in *Embedded Computing for High Performance*, J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, Eds. Boston: Morgan Kaufmann, 2017, pp. 17 – 56. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128041895000028

[25] M. Wolf, "Chapter 2 - CPUs," in *High-Performance Embedded Computing (Second Edition)*, second edition ed., M. Wolf, Ed. Boston: Morgan Kaufmann, 2014, pp. 59 – 138. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780124105119000022

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[27] G. Polito, N. Palumbo, S. Labsari, S. Ducasse, and P. Tesone, "Interpreter register autolocalisation: Improving the performance of efficient interpreters," in *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–5. [Online]. Available: https://doi.org/10.1145/3532512.3532518

[28] K. R. Bowers and D. R. Kaeli, "Characterizing the SPEC JVM98 benchmarks on the Java Virtual Machine," Northeastern University, Department of Electrical and Computer Engineering, Boston, MA, USA, Tech. Rep. ECE-CEG-98-026, Sep. 1998, [Online]. Available: https://ece.northeastern.edu/groups/nucar/publications/javafin.pdf.

[29] J. Lambert, R. Monahan, and K. Casey, "Repositioning tiered HotSpot execution performance relative to the interpreter," [Online]. Available: https://arxiv.org/abs/2303.04157, 2023.

[30] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 11 2013. [Online]. Available: https://doi.org/10.1093/comjnl/bxt129

[31] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[32] T. L. Floyd, *Principles of Electric Circuits: Conventional Current Version*, 9th ed. Upper Saddle River, NJ: Pearson, 2010.

[33] M. A. Hardy, *Regression with Dummy Variables*, ser. Quantitative Applications in the Social Sciences. Newbury Park, CA, USA: SAGE Publications, 1993, no. 93.

[34] R. F. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[35] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[36] C. Herder and J. Dujmovic, "Workload characterization using metrics based on instruction grouping," *International Journal of Computer and Information Sciences*, vol. 5, no. 1, 2004.

[37] H. McGhan and M. O'Connor, "PicoJava: A direct execution engine for Java bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, 1998.

[38] J. Waldron and O. Harrison, "Analysis of virtual machine stack frame usage by Java methods." in *IMSA*. Citeseer, 1999, pp. 271–274.

[39] J. Waldron, "Dynamic bytecode usage by object oriented Java programs," in *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*, 1999, pp. 384–393.

[40] J. Waldron, C. Daly, D. Gray, and J. Horgan, "Comparison of factors influencing bytecode usage in the Java Virtual Machine," in *Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK*, 2000.

[41] S. Byrne, C. Daly, D. Gregg, and J. Waldron, "Dynamic analysis of the Java Virtual Machine method invocation architecture," *Database*, vol. 96, no. 8.3, pp. 91–7, 2002.

[42] T. Dowling, J. F. Power, and J. Waldron, "Relating static and dynamic measurements for the Java Virtual Machine instruction set," *Recent Advances in Simulation, Computational Methods and Soft Computing*, 2002.

[43] D. Gregg, J. Power, and J. Waldron, "Platform independent dynamic Java Virtual Machine analysis: the Java Grande Forum benchmark suite," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 459–484, 2003.

[44] D. O'Donoghue, A. Leddy, J. F. Power, and J. Waldron, "Bi-gram analysis of Java bytecode sequences," in *Proceedings Second Workshop on Intermediate Representation Engineering for the Java Virtual Machine*. National University of Ireland Maynooth, 2002, pp. 187–192.

[45] D. O'Donoghue and J. F. Power, "Identifying and evaluating a generic set of superinstructions for embedded Java programs," in *International Conference on Embedded Systems and Applications (ESA '04)*, June 2004, pp. 192–198.

[46] D. Gregg, J. Power, and J. Waldron, "A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 7-8, pp. 757–773, 2005.

[47] J. Lambert and J. F. Power, "An analysis of basic blocks within SPECjvm98 applications," Department of Computer Science, National University of Ireland Maynooth, Maynooth, Co. Kildare, Ireland, Technical Report NUIM-CS-TR-2005-15, Nov. 2005. [Online]. Available: https://norma.ncirl.ie/1395/1/NUIM-CS-TR-2005-15.pdf

[48] J. Park, J. Park, W. Song, S. Yoon, B. Burgstaller, and B. Scholz, "Treegraph-based instruction scheduling for stack-based virtual machines," *Electronic Notes in Theoretical Computer Science*, vol. 279, no. 1, pp. 33–45, 2011.

[49] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic Press, 2013.

[50] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, "A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo Scala, and SPECjvm2008," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 3–14. [Online]. Available: https://doi.org/10.1145/3030207.3030211

[51] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification Java SE 20 Edition*. Oracle, 2023.

[52] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification Java SE 20 Edition*. Oracle, 2023.

[53] Oracle, "Java Platform, Standard Edition & Java Development Kit Version 14 API Specification," [Online]. Available: https://docs.oracle.com/en/java/javase/14/docs/api/index.html, 2020, [Accessed: Sep. 20, 2023].

[54] K. Casey, A. Ertl, and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 6:37, 2007. [Online]. Available: http://mural.maynoothuniversity.ie/10188/

[55] E. Gagnon and L. Hendren, "Effective inline-threaded interpretation of Java bytecode using preparation sequences," in *International Conference on Compiler Construction*. Springer, 2003, pp. 170–184.

[56] B. Davis and J. Waldron, "A survey of optimisations for the Java Virtual Machine," in *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '03. USA: Computer Science Press, Inc., 2003, p. 181–183.

[57] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ client compiler for Java 6," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 1, pp. 1–32, 2008.

[58] S. Oaks, *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* " O'Reilly Media, Inc.", 2014.

[59] V. Ivanov, "JVM JIT-compiler Overview," [Online]. Available: https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf, Oracle Corp., [Accessed: Mar. 5, 2023].

[60] J. Rose, "OpenJDK Wiki: Performance Tactic Index," [Online]. Available: https://wiki.openjdk.org/display/HotSpot/PerformanceTacticIndex, 2009, [Accessed: Sep. 4, 2023].

[61] Oracle, "Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide," [Online]. Available: https://docs.oracle.com/javase/9/gctuning/toc.htm, 2017, [Accessed: Sep. 4, 2023].

[62] B. J. Evans, *Java: The Legend*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

[63] M. Reinhold, "JDK enhancement proposal index," [Online]. Available: https://openjdk.java.net/jeps/0, [Accessed: May 26, 2022].

[64] S. Johansson, "JDK enhancement proposal 344: Parallel full GC for G1," [Online]. Available: https://openjdk.java.net/jeps/344, [Accessed: May 26, 2022].

[65] E. Helin, "JDK enhancement proposal 344: Abortable mixed collections for G1," [Online]. Available: https://openjdk.java.net/jeps/344, [Accessed: May 26, 2022].

[66] R. S. Rodrigo Bruno, Thomas Schatzl, "JDK enhancement proposal 346: Promptly return unused committed memory from G1," [Online]. Available: https://openjdk.java.net/jeps/346, [Accessed: May 26, 2022].

[67] S. Kim, "JDK enhancement proposal 345: NUMA-aware memory allocation for G1," [Online]. Available: https://openjdk.java.net/jeps/345, [Accessed: May 26, 2022].

[68] I. Lam, "JDK enhancement proposal 310: Application class-data sharing," [Online]. Available: https://openjdk.java.net/jeps/310, [Accessed: May 26, 2022].

[69] I. L. Jiangli Zhou, Calvin Cheung, "JDK enhancement proposal 341: Default CDS archives," [Online]. Available: https://openjdk.java.net/jeps/341, [Accessed: May 26, 2022].

[70] C. C. Jiangli Zhou, Ioi Lam, "JDK enhancement proposal 346: Dynamic CDS archives," [Online]. Available: https://openjdk.java.net/jeps/350, [Accessed: May 26, 2022].

[71] M. C. Carlisle and A. Rogers, "Software caching and computation migration in Olden," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 29–38, 1995.

[72] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in Java," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 280–291.

[73] reprogrammer, "The JOlden Benchmark Suite," GitHub repository. [Online]. Available: https://github.com/reprogrammer/jolden, [Accessed: Sep. 15, 2026].

[74] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A benchmark suite for high performance Java," *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 375–388, 2000.

[75] Roldan Pozo and Bruce R. Miller, "SciMark 2.0: A Java benchmark for scientific and numerical computing," National Institute of Standards and Technology (NIST), 2000, [Online]. Available: https://math.nist.gov/scimark2/.

[76] Standard Performance Evaluation Corporation, "SPECjbb2015," [Online]. Available: https://www.spec.org/jbb2015/index.html, 2015, [Accessed: Sep. 7, 2023].

[77] Standard Performance Evaluation Corporation, "SPECjEnterprise 2018 Web Profile," [Online]. Available: https://www.spec.org/jenterprise2018web/, 2018, [Accessed: Sep. 15, 2026].

[78] Standard Performance Evaluation Corporation, "SPECjvm2008," [Online]. Available: https://www.spec.org/jvm2008/index.html, 2008, [Accessed: Sep. 7, 2023].

[79] I. Gouy, "The Computer Language Benchmarks Game," [Online]. Available: https://sschakraborty.github.io/benchmark/index.html, [Accessed: Sep. 4, 2023].

[80] B. Goel, S. A. McKee, and M. Själander, "Chapter two - techniques to measure, model, and manage power," in *Advances in Computers*, A. Hurson and A. Memon, Eds. Elsevier, 2012, vol. 87, pp. 7 – 54. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978012396528800002X

[81] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, "Portable, scalable, per-core power estimation for intelligent resource management," in *International Conference on Green Computing*, 2010, pp. 135–146.

[82] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 4 - source code analysis and instrumentation," in *Embedded Computing for High Performance*, J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, Eds. Boston: Morgan Kaufmann, 2017, pp. 99 – 135. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128041895000041

[83] J. Nunez-Yanez and G. Lore, "Enabling accurate modeling of power and energy consumption in an ARM-based system-on-chip," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 319 – 332, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933113000021

[84] Standard Performance Evaluation Corporation (SPEC), "Power and Performance Benchmark Methodology V2.2," [Online]. Available: https://www.spec.org/power_ssj2008/docs/SPECpower-Methodology.pdf, 7001 Heritage Village Plaza, Suite 225 Gainesville, VA 20155, USA, 2014, [Accessed: Jun. 3, 2020].

[85] Standard Performance Evaluation Corporation, "SPECpower_ssj2008 Design Overview," [Online]. Available: https://www.spec.org/power/docs/SPECpower_ssj2008-Design_Overview.pdf, May 2012, [Accessed: Sep. 15, 2026].

[86] Y. Mao, "Detailed power measurement with ARM embedded boards," Master's thesis, University of Maine, Orono, ME, USA, 2018, [Accessed:

Sep. 15, 2026]. [Online]. Available: https://digitalcommons.library.umaine.edu/etd/2878

[87] M. Tanner, R. Eckel, and I. Senevirathne, "Enhanced low current, voltage, and power dissipation measurements via Arduino Uno microcontroller with modified commercially available sensors," in *APS March Meeting Abstracts*, 2016.

[88] B. Dezfouli, I. Amirtharaj, and C.-C. C. Li, "EMPIOT: An energy measurement platform for wireless IoT devices," *Journal of Network and Computer Applications*, vol. 121, pp. 135 – 148, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804518302479

[89] P. S. Munoz, N. Tran, B. Craig, B. Dezfouli, and Y. Liu, "Analyzing the resource utilization of AES encryption on IoT devices," in *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2018, pp. 1200–1207.

[90] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *International Green Computing Conference*. Piscataway, NJ, USA: IEEE, Nov 2014, pp. 1–6.

[91] J. J. Merelo-Guervós, I. Blancas-Alvarez, P. A. Castillo, G. Romero, P. García-Sánchez, V. M. Rivas, M. García-Valdez, A. Hernández-Águila, and M. Román, "Ranking the performance of compiled and interpreted languages in genetic algorithms," in *International Conference on Evolutionary Computation Theory and Applications*, vol. 2. SCITEPRESS, 2016, pp. 164–170.

[92] G. G. Magalhaes, A. L. Sartor, A. F. Lorenzon, P. O. A. Navaux, and A. C. Schneider Beck, "How programming languages and paradigms affect performance and energy in multithreaded applications," in *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2016, pp. 71–78.

[93] H. Eichhorn, J. L. Cano, F. McLean, and R. Anderl, "A comparative study of programming languages for next-generation astrodynamics systems," *CEAS Space Journal*, vol. 10, no. 1, pp. 115–123, 2018.

[94] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software*

Language Engineering, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[95] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022

[96] H. Rich, "What you need to know about the JITServer from IBM Semeru Runtimes – The basics," [Online]. Available: https://developer.ibm.com/blogs/all-you-need-to-know-about-the-openj9-jitserver/, Jun 2022, [Accessed: Feb. 28, 2023].

[97] A. G. Graham, J.-P. Legault, H. Soontiens, J. Brown, S. A. MacKay, G. W. Dueck, K. B. Kent, K. Konno, and D. Maier, "Evaluating the performance of the Eclipse OpenJ9 JVM JIT compiler on AArch64," in *2022 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2022, pp. 411–418.

[98] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the jalapeño JVM," *SIGPLAN Not.*, vol. 35, no. 10, p. 47–65, oct 2000. [Online]. Available: https://doi.org/10.1145/354222.353175

[99] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded Java environment," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*. Piscataway, NJ, USA: IEEE, 2002, pp. 92–103.

[100] C. S. Ballapuram, *Semantics-oriented low power architecture*. Georgia Institute of Technology, 2008.

[101] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis, "What are your programming language's energy-delay implications?" in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–313. [Online]. Available: https://doi.org/10.1145/3196398.3196414

[102] D. Mossé, N. Aboughazaleh, B. Childers, and R. Melhem, "Energy conservation in memory hierarchies using power-aware cached-DRAM," IDepartment of Computer Science, University of Pittsburgh, Tech. Rep. TR-05-123, 2005.

[103] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin, "Energy behavior of Java applications from the memory perspective," in *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[104] S. Lafond and J. Lilius, "An energy consumption model for an embedded Java Virtual Machine," in *Architecture of Computing Systems - ARCS 2006*, W. Grass, B. Sick, and K. Waldschmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 311–325.

[105] T. Cao, S. M. Blackburn, and K. S. McKinley, "Virtual machine services: An opportunity for hardware customization," *Technology*, vol. 32, no. 32nm, p. 45nm, 2011.

[106] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "Runtime monitoring of software energy hotspots," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 160–169.

[107] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: A reflection on the state of the art," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. IEEE, 2015, pp. 29–35.

[108] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 345–360.

[109] G. Rocha, F. Castor, and G. Pinto, "Comprehending energy behaviors of Java I/O APIs," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[110] M. Longo, A. V. Rodriguez, C. M. Mateos Diaz, and A. O. Zunino Suarez, "Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings," *Computer Science And Information Systems*, vol. 16, no. 2, pp. 541–564, 2023.

[111] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. P. Fernandes, "Improving energy-efficiency by recommending Java collections," *Empirical Software Engineering*, vol. 26, pp. 1–45, 2021.

[112] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating the impact of Java Virtual Machines on energy consumption," in *Proceedings*

of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2021, pp. 1–11.

[113] M. Shimchenko, M. Popov, and T. Wrigstad, "Analysing and predicting energy consumption of garbage collectors in OpenJDK," in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, 2022, pp. 3–15.

[114] D. C. Bree and M. Ó. Cinnéide, "Removing decorator to improve energy efficiency," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 902–912.

[115] G. Chen, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Adaptive garbage collection for battery-operated environments." in *Java Virtual Machine Research and Technology Symposium*, 2002, pp. 1–12.

[116] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 225–236, 2012.

[117] R. Kumar and P. K. Singh, "An approach for compiler optimization to exploit instruction level parallelism," in *Advanced Computing, Networking and Informatics- Volume 2*, M. Kumar Kundu, D. P. Mohapatra, A. Konar, and A. Chakraborty, Eds. Cham: Springer International Publishing, 2014, pp. 509–516.

[118] Marcon, Caterina, Smirnova, Oxana, and Muralidharan, Servesh, "Impact of different compilers and build types on Geant4 simulation execution time," *EPJ Web Conf.*, vol. 245, p. 05037, 2020. [Online]. Available: https://doi.org/10.1051/epjconf/202024505037

[119] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, F. Behner, L. Bellagamba, J. Boudreau, L. Broglia, A. Brunengo, H. Burkhardt, S. Chauvie, J. Chuma, R. Chytracek, G. Cooperman, G. Cosmo, P. Degtyarenko, A. Dell'Acqua, G. Depaola, D. Dietrich, R. Enami, A. Feliciello, C. Ferguson, H. Fesefeldt, G. Folger, F. Foppiano, A. Forti, S. Garelli, S. Giani, R. Giannitrapani, D. Gibin, J. Gómez Cadenas, I. González, G. Gracia Abril, G. Greeniaus, W. Greiner, V. Grichine, A. Grossheim, S. Guatelli, P. Gumplinger, R. Hamatsu, K. Hashimoto, H. Hasui, A. Heikkinen, A. Howard, V. Ivanchenko, A. Johnson, F. Jones, J. Kallenbach, N. Kanaya,

M. Kawabata, Y. Kawabata, M. Kawaguti, S. Kelner, P. Kent, A. Kimura, T. Kodama, R. Kokoulin, M. Kossov, H. Kurashige, E. Lamanna, T. Lampén, V. Lara, V. Lefebure, F. Lei, M. Liendl, W. Lockman, F. Longo, S. Magni, M. Maire, E. Medernach, K. Minamimoto, P. Mora de Freitas, Y. Morita, K. Murakami, M. Nagamatu, R. Nartallo, P. Nieminen, T. Nishimura, K. Ohtsubo, M. Okamura, S. O'Neale, Y. Oohata, K. Paech, J. Perl, A. Pfeiffer, M. Pia, F. Ranjard, A. Rybin, S. Sadilov, E. Di Salvo, G. Santin, T. Sasaki, N. Savvas, Y. Sawada, S. Scherer, S. Sei, V. Sirotenko, D. Smith, N. Starkov, H. Stoecker, J. Sulkimo, M. Takahata, S. Tanaka, E. Tcherniaev, E. Safai Tehrani, M. Tropeano, P. Truscott, H. Uno, L. Urban, P. Urban, M. Verderi, A. Walkden, W. Wander, H. Weber, J. Wellisch, T. Wenaus, D. Williams, D. Wright, T. Yamada, H. Yoshida, and D. Zschiesche, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168900203013688

[120] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: an overview," in *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, 1994, pp. 38–39.

[121] D. Branco and P. R. Henriques, "Impact of GCC optimization levels in energy consumption during program execution," *Acta Electrotechnica et Informatica*, vol. 16, no. 1, pp. 20–26, 2016.

[122] P. Yuan, Y. Guo, and X. Chen, "Rethinking compiler optimizations for the Linux kernel," in *Proceedings of the 2015 Workshop on Programming Languages and Operating Systems (PLOS '15)*. Association for Computing Machinery, 2015.

[123] M. Ugur, C. Jiang, A. Erf, T. Ahmed Khan, and B. Kasikci, "One profile fits all: Profile-guided Linux kernel optimizations for data center applications," *SIGOPS Oper. Syst. Rev.*, vol. 56, no. 1, p. 26–33, Jun. 2022. [Online]. Available: https://doi.org/10.1145/3544497.3544502

[124] M. Palkowski and M. Gruzewski, "Time and energy benefits of using automatic optimization compilers for NPDP tasks," *Electronics*, vol. 12, no. 17, 2023. [Online]. Available: https://www.mdpi.com/2079-9292/12/17/3579

[125] K. Venugopal, G. Manjunath, and V. Krishnan, "{sEc}: A portable interpreter optimizing technique for embedded Java Virtual Machine," in *2nd*

Java Virtual Machine Research and Technology Symposium (Java VM 02), 2002.

[126] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: application initialization at build time," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[127] J. Lambert, R. Monahan, and K. Casey, "Accidental choices—How JVM choice and associated build tools affect interpreter performance," *Computers*, vol. 11, no. 6, p. 96, 2022.

[128] R.-G. Urma, "Alternative languages for the JVM. a look at eight features from eight JVM languages," [Online]. Available: https://www.oracle.com/technical-resources/articles/{J}ava/architect-languages.html, Jul 2014, [Accessed: Apr. 23, 2021].

[129] Oracle, "Oracle GraalVM Enterprise Edition 22: Truffle Language Implementation Framework," [Online]. Available: https://docs.oracle.com/en/graalvm/enterprise/22/docs/graalvm-as-a-platform/language-implementation-framework/, 2022, [Accessed: Apr. 20, 2022].

[130] A. Prokopec, A. Rosa, D. Leopoldseder, G. Duboscq, P. Tuma, M. Studener, L. Bulej, Y. Zheng, A. Villazon, D. Simon *et al.*, "On evaluating the Renaissance benchmarking suite: Variety, performance, and complexity," *arXiv preprint arXiv:1903.10267*, 2019.

[131] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon *et al.*, "Renaissance: A modern benchmark suite for parallel applications on the JVM," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2019, pp. 11–12.

[132] T. A. Proebsting, "Optimizing an ANSI C interpreter with superoperators," in *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 1995, pp. 322–332.

[133] M. Maierhofer and M. A. Ertl, "Local Stack Allocation," in *CC '98: Proceedings of the 7th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 1998, pp. 189–203.

[134] M. A. Ertl and D. Gregg, "The behavior of efficient virtual machine interpreters on modern architectures," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, ser. Euro-Par '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 403–412.

[135] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–25, 2003.

[136] M. Berndl, B. Vitale, M. Zaleski, and A. Brown, "Context threading: a flexible and efficient dispatch technique for virtual machine interpreters," in *International Symposium on Code Generation and Optimization.* Piscataway, NJ, USA: IEEE, 2005, pp. 15–26.

[137] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, E. Seckler, C. Li, S. Brunthaler, P. Larsen, and M. Franz, "Efficient hosted interpreters on the JVM," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, feb 2014. [Online]. Available: https://doi.org/10.1145/2532642

[138] S. Ritter, "A Matter of Interpretation: From Bytecodes to Machine Code in the JVM," [Online]. Available: https://www.{A}zul.com/blog/a-matter-of-interpretation-from-bytecodes-to-machine-code-in-the-jvm/, Azul, 2022, [Accessed: Mar. 7, 2023].

[139] V. Simonis, "Template- vs. C++-Interpreter shootout," [Online]. Available: https://web.archive.org/web/20090214031632/https://weblogs.{J}ava.net/blog/simonis/archive/2007/11/template_vs_cin.html, 2007, [Accessed: Mar. 7, 2023].

[140] D. Gregg, M. A. Ertl, and A. Krall, "Implementing an efficient Java interpreter," in *High-Performance Computing and Networking*, B. Hertzberger, A. Hoekstra, and R. Williams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 613–620.

[141] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl, "Virtual machine showdown: stack versus registers," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005, pp. 153–163.

[142] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, pp. 1–36, 2008.

[143] J. Rose, "Performance tactic index," [Online]. Available: https://wiki.openjdk.org/display/HotSpot/PerformanceTacticIndex, 2009, [Accessed: Apr. 26, 2023].

[144] L. Dykstra, W. Srisa-an, and J. M. Chang, "An analysis of the garbage collection performance in sun's HotSpot/sup TM/Java Virtual Machine," in *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference (Cat. No. 02CH37326)*. IEEE, 2002, pp. 335–339.

[145] S. Tavakolisomeh, R. Bruno, and P. Ferreira, "BestGC: An automatic GC selector," *IEEE Access*, 2023.

[146] G. Contreras and M. Martonosi, "Techniques for real-system characterization of Java Virtual Machine energy and power behavior," in *2006 IEEE International Symposium on Workload Characterization*. IEEE, 2006, pp. 29–38.

[147] S. Bhattacharya, K. Rajamani, and M. Gupta, "Power-performance implications of software runtime bloat: A case study with the SPECpower_ssj2008 benchmark," IBM Research, Tech. Rep., 2010.

[148] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in Java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019, pp. 1677–1682.

[149] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 46–57.

[150] J. W. Davidson, J. R. Rabung, and D. B. Whalley, "Relating static and dynamic machine code measurements," *IEEE Transactions on Computers*, vol. 41, no. 4, pp. 444–454, 1992.

[151] D. Gregg, J. Power, and J. Waldron, "Benchmarking the Java virtual architecture: The SPEC JVM98 benchmark suite," *Java Microarchitectures*, pp. 1–18, 2002.

[152] J. Horgan, J. Power, and J. Waldron, "Measurement and analysis of runtime profiling data for Java programs," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2001, pp. 122–130.

[153] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," in *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, 2003, pp. 41–49.

[154] L. Ardito and M. Torchiano, "Creating and evaluating a software power model for Linux single board computers," in *2018 IEEE/ACM 6th International Workshop on Green And Sustainable Software (GREENS)*, 2018, pp. 1–8.

[155] J. J. Dujmović, "Evaluation and design of benchmark suites," *State-of-the-Art in Performance Modeling and Simulation: Theory, Techniques, and Tutorials*, pp. 287–323, 1998.

[156] J. J. Dujmovic and I. Dujmovic, "Evolution and evaluation of SPEC benchmarks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 2–9, 1998.

[157] J. J. Dujmovic, "Universal benchmark suites," in *MASCOTS'99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.* IEEE, 1999, pp. 197–205.

[158] J. J. Dujmovic and C. Herder, "Java workload characterization using bytecodes," in *Int. CMG Conference*, 2001, pp. 205–216.

[159] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The structure and performance of interpreters," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. New York, NY, USA: Association for Computing Machinery, 1996, p. 150–159. [Online]. Available: https://doi.org/10.1145/237090.237175

[160] D. Gregg, M. A. Ertl, and A. Krall, "A fast Java interpreter," in *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES), Genoa.* Citeseer, 2001.

[161] O. Papadakis, A. Andronikakis, N. Foutris, M. Papadimitriou, A. Stratikopoulos, F. Zakkak, P. Xekalakis, and C.-E. Kotselidis, "A multi-faceted memory analysis of Java benchmarks," in *20th International Conference on Managed Programming Languages & Runtimes*, 2023.

[162] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan, "Java runtime systems: Characterization and architectural implications," *IEEE Transactions on computers*, vol. 50, no. 2, pp. 131–146, 2001.

[163] T. Li, R. Bhargava, and L. K. John, "Adapting branch-target buffer to improve the target predictability of Java code," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 2, pp. 109–130, 2005.

[164] E. Rohou, B. N. Swamy, and A. Seznec, "Branch prediction and the performance of interpreters—don't trust folklore," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 103–114.

[165] C. Kim, S. Kim, H. G. Cho, D. Kim, J. Kim, Y. H. Oh, H. Jang, and J. W. Lee, "Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 291–303, 2016.

[166] M. A. Ertl, "Stack caching for interpreters," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995, pp. 315–327.

[167] S. Hu *et al.*, "Cache performance in Java Virtual Machines: A study of constituent phases," in *2002 IEEE International Workshop on Workload Characterization.* IEEE, 2002, pp. 81–90.

[168] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, Java," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400689

[169] S. Crooijmans, M. Lubbers, and P. Koopman, "Reducing the power consumption of IoT with task-oriented programming," in *Trends in Functional Programming*, W. Swierstra and N. Wu, Eds. Cham: Springer International Publishing, 2022, pp. 80–99.

[170] I. Ashraf, M. Umer, R. Majeed, A. Mehmood, W. Aslam, M. N. Yasir, and G. S. Choi, "Home automation using general purpose household electric appliances with Raspberry Pi and commercial smartphone," *PloS one*, vol. 15, no. 9, pp. e0 238 480–e0 238 480, 2020.

[171] G. Coleman, W. Salter, and M. Walsh, "OpenWeedLocator (OWL): an open-source, low-cost device for fallow weed detection," *Scientific Reports*, vol. 12, no. 1, 2022, cited by: 4; All Open Access, Gold Open Access, Green Open Access. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85122560652&doi=10.1038%2fs41598-021-03858-9&partnerID=40&md5=e1960f2fae049106c8ecb0feabc5c1ed

[172] L. Chechlinski, B. Siemiatkowska, and M. Majewski, "A system for weeds and crops identification-reaching over 10 FPS on Raspberry Pi with the usage of MobileNets, DenseNet and custom modifications," *Sensors (Basel, Switzerland)*, vol. 19, no. 17, 2019.

[173] K. Joshitha, S. Nishanthini, B. Harshini, and T. Sadhana, "Raspberry Pi based crop and weed classification using machine learning algorithm," in *IOP Conference Series: Earth and Environmental Science*, vol. 980, 2022. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85125841008&doi=10.1088%2f1755-1315%2f980%2f1%2f012029&partnerID=40&md5=2906d2fc9da4e754b064c165f83dd2ad

[174] J. Y. Kim, H. Abdel-Haleem, Z. Luo, and A. Szczepanek, "Open-source electronics for plant phenotyping and irrigation in controlled environment," *Smart Agricultural Technology*, vol. 3, p. 100093, 2023.

[175] M. F. Kabir and S. Roy, "Real-time vehicular accident prevention system using deep learning architecture," *Expert systems with applications*, vol. 206, p. 117837, 2022.

[176] L. Skolrood, Y. Wang, S. Zhang, and Q. Wei, "Single-molecule and particle detection on true portable microscopy platforms," *Sensors and actuators reports*, vol. 4, p. 100063, 2022.

[177] Z. Wu, Y. Chen, B. Zhao, X. Kang, and Y. Ding, "Review of weed detection methods based on computer vision," *Sensors*, vol. 21, no. 11, 2021. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85106240294&doi=10.3390%2fs21113647&partnerID=40&md5=8dbf08a24aec888f938ced63282733d9

[178] M. Lupión, A. Polo-Rodríguez, J. Medina-Quero, J. F. Sanjuan, and P. M. Ortigosa, "On the limits of conditional generative adversarial neural networks to reconstruct the identification of inhabitants from IoT low-resolution thermal sensors," *Expert Systems with Applications*, vol. 203, p. 117356, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422007084

[179] S. D. Icagic and G. S. Kvascev, "A smart alcoholmeter sensor based on deep learning visual perception," *Sensors*, vol. 22, no. 19, 2022.

[180] J. C. Tovar, J. S. Hoyer, A. Lin, A. Tielking, S. T. Callen, S. Elizabeth Castillo, M. Miller, M. Tessman, N. Fahlgren, J. C. Carrington, D. A. Nusinow, and M. A. Gehan, "Raspberry Pi–powered imaging for plant phenotyping," *Applications in plant sciences*, vol. 6, no. 3, pp. e1031–n/a, 2018.

[181] F. Astudillo-Salinas, D. Barrera-Salamea, A. Vázquez-Rodas, and L. Solano-Quinde, "Minimizing the power consumption in Raspberry Pi to use as a remote WSN gateway," in *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*, 2016, pp. 1–5.

[182] J. M. Paniego, L. Libutti, M. Pi Puig, F. Chichizola, L. C. De Giusti, M. Naiouf, and A. E. De Giusti, "Modelado estadístico de potencia usando contadores de rendimiento sobre Raspberry Pi," in *XXIV Congreso Argentino de Ciencias de la Computación (La Plata, 2018).*, 2018.

[183] J. M. Paniego, L. Libutti, M. Pi Puig, F. Chichizola, L. C. De Giusti, M. Naiouf, and A. E. De Giusti, "Modelado de potencia en placas SBC: integración de diferentes generaciones Raspberry Pi," in *XXV Congreso Argentino de Ciencias de la Computación (CACIC 2019, Universidad Nacional de Río Cuarto)*, 2019.

[184] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, p. 63–73, Sep 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[185] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. USA: IEEE Computer Society, 2009, p. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[186] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728

[187] J. Somarribas, E. Meneses, and K. Olivas, "Predictive power consumption model for compute intensive applications in clustered ARM A53 embedded systems," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[188] J. M. Paniego, L. Libutti, M. P. Puig, F. Chichizola, L. De Giusti, M. Naiouf, and A. De Giusti, "Unified power modeling design for various Raspberry Pi generations analyzing different statistical methods," in *Computer Science – CACIC 2019*, P. Pesado and M. Arroyo, Eds. Cham: Springer International Publishing, 2020, pp. 53–65.

[189] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 185–194. [Online]. Available: https://doi.org/10.1145/1168857.1168881

[190] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," in *39th Annual IEEE Conference on Local Computer Networks*, 2014, pp. 236–243.

[191] G. Bekaroo and A. Santokhee, "Power consumption of the Raspberry Pi: A comparative analysis," in *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech)*, 2016, pp. 361–366.

[192] M. Cloutier, C. Paradis, and V. Weaver, "A Raspberry Pi cluster instrumented for fine-grained power measurement," *Electronics*, vol. 5, no. 4, p. 61, Sep 2016. [Online]. Available: http://dx.doi.org/10.3390/electronics5040061

[193] W. Hajji and F. P. Tso, "Understanding the performance of low power Raspberry Pi cloud for big data," *Electronics*, vol. 5, no. 2, p. 29, 2016.

[194] S. E. Alves Filho, A. M. F. Burlamaqui, R. V. Aroca, and L. M. G. Gonçalves, "NPi-cluster: A low power energy-proportional computing cluster architecture," *IEEE Access*, vol. 5, pp. 16 297–16 313, 2017.

[195] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL: A portable implementation of the high-performance Linpack benchmark for distributed-memory computers," [Online]. Available: https://www.netlib.org/benchmark/hpl/, 2008, [Accessed: Sep. 15, 2026].

[196] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[197] S. Plimpton, P. Crozier, and C. Trott, "miniMD: Molecular dynamics mini-app," GitHub repository. [Online]. Available: https://github.com/Mantevo/miniMD, [Accessed: Sep. 15, 2026].

[198] T. Shizukuishi and K. Matsubara, "An efficient tinification of the Linux kernel for minimizing resource consumption," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1228–1237. [Online]. Available: https://doi.org/10.1145/3341105.3373913

[199] I. Amirtharaj, T. Groot, and B. Dezfouli, "Profiling and improving the duty-cycling performance of Linux-based IoT devices," *CoRR*, vol. abs/1808.10097, 2018. [Online]. Available: http://arxiv.org/abs/1808.10097

[200] Raspberry Pi Ltd, "BCM2711 ARM Peripherals," [Online]. Available: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf, 2022, [Accessed: Sep. 15, 2026].

[201] SanDisk, "SanDisk Ultra microSDHC UHS-I Card, 32 GB, A1, Class 10 (SDSQUA4-032G-GN6MA)," [Online]. Available: https://www.sandisk.com/en-ie/products/memory-cards/microsd-cards/sandisk-ultra-uhs-i-microsd?sku=SDSQUA4-032G-GN6MA, [Accessed: Jun. 6, 2026].

[202] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, 1st ed. John Wiley & Sons, 1991.

[203] OpenJDK, "Archived releases," [Online]. Available: https://wiki.openjdk.org/spaces/JDKUpdates/pages/42598676/Archived+Releases, 2026, [Accessed: Jun. 9, 2026].

[204] H. Oi, "A comparative study of JVM implementations with SPECjvm2008," in *2010 Second International Conference on Computer Engineering and Applications*, vol. 1, 2010, pp. 351–357.

[205] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*, 3rd ed. John Wiley & Sons, 2014.

[206] G. W. Corder and D. I. Foreman, *Nonparametric Statistics: A Step-by-Step Approach*, 1st ed. Hoboken, NJ: Wiley, 2009.

[207] W. D. Berry, *Understanding regression assumptions*. Sage, 1993, vol. 92.

[208] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. Chapman and Hall/CRC, 1994.

[209] J. Fox and S. Weisberg, "Bootstrapping regression models in R," Appendix to *An R Companion to Applied Regression*, 3rd ed., 2018. [Online]. Available: https://www.john-fox.ca/Companion/appendices/Appendix-Bootstrapping.pdf

[210] A. Canty and B. Ripley, *boot: Bootstrap Functions*, 2025, r package version 1.3-32. [Online]. Available: https://CRAN.R-project.org/package=boot

[211] OpenJDK, "JDK-8233466: aarch64: Remove unnecessary load of mdo when profiling return and parameter types," OpenJDK JDK Bug System. [Online]. Available: https://bugs.openjdk.org/browse/JDK-8233466, 2019, [Accessed: Aug. 29, 2026].