A Coverage Analysis of Java Benchmark Suites

Stephen Brown, Áine Mitchell and James F. Power

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

The IASTED International Conference on Software Engineering,
Innsbruck, Austria, February 15-17, 2005

Note: This is a slightly extended version of the paper that appears in The IASTED International Conference on Software Engineering, 2005.

A COVERAGE ANALYSIS OF JAVA BENCHMARK SUITES

Stephen Brown, Áine Mitchell and James F. Power*
Department of Computer Science,
National University of Ireland,
Maynooth, Co. Kildare, Ireland.

ABSTRACT
The Java programming language provides an almost ideal environment for both static and dynamic analysis, being easy to parse, and supporting a standardised, easily-profiled virtual environment.

In this paper we study the relationship between results obtainable from static and dynamic analysis of Java programs, and in particular the difficulties of correlating static and dynamic results. As a foundation for this study, we focus on various criteria related to run-time code coverage, as commonly used in test suite analysis.

We have implemented a dynamic coverage analysis tool for Java programs, and we use it to evaluate several standard Java benchmark suites using line, instruction and branch coverage criteria. We present data indicating a considerable variance in static and dynamic analysis results between these suites, and even between programs in these suites.

KEY WORDS
Benchmarking, software testing, dynamic analysis.

1 Introduction and Motivation

In this paper we provide a foundation for the study of the relationship between results obtainable from static and dynamic analysis of Java programs. This paper is motivated by the following observations:

1. The Java programming language and its associated virtual machine provides a fertile environment for the static and dynamic analysis of object-oriented programs.
2. A significant body of research already exists on the analysis and manipulation of Java code, and a number of papers refer to both static and dynamic analysis
3. Fundamental to relating static and dynamic analysis is a knowledge of the degree to which the analysed source code corresponds to the code that is actually executed, yet there is little existing research that compares results obtained from static and dynamic analyses.

In this paper we seek to address this apparent gap in the literature and in research by examining Java programs from a static and dynamic perspective, and using coverage criteria to provide a framework for comparison. We posit that poor coverage results may hinder the comparison of static and dynamic analyses and should, in any case, be a measured, recorded factor in such a comparison.

The remainder of this paper is structured as follows. Section 2 discusses the background to our work, and seeks to connect static analysis metrics with dynamic coverage analysis. Sections 3 and 4 use this connection to evaluate some common benchmark suites for Java. Section 5 summarises these results and concludes the paper.

2 Background and Related Work

A significant amount of the research on the analysis and manipulation of Java programs has sought to combine static and dynamic data, or to manipulate the dynamic behaviour of Java programs through static code transformations. Recent examples from the last year alone include work on conflict analysis [1], super-instruction selection [2], static and dynamic slicing [3], and, of course, program optimisation [4].

Recent work by the Sable group [5] has sought to record the results of running various programs from different benchmark suites. Many of the measurements they record, such as program size, polymorphism and memory usage are of interest to the Java performance community. However, the title of their work, "Dynamic Metrics" seems to draw a parallel with the corresponding field of *static* software metrics.

Much of the research that performs both static and dynamic analysis of Java code concentrates on a particular aspect of a program's behaviour. However, we suggest that any such comparison should be viewed in the context of some *overall* perspective of the relationship between the static and dynamic data. To this end, this paper analyses a series of Java programs using four basic measurements, lines of code, cyclomatic complexity, instruction coverage and branch coverage. These measurements are summarised in Figure 1.

To quantify the static aspects of a program we have used the two most basic metrics available. The lines of code (LOC) metric is frequently used as a general guide to the size of a program. It is just as frequently criticised as being subject to many arbitrary variations depending on the language used, source code layout etc. Yet despite its

*Contact author: James F. Power `<jpower@cs.may.ie>`

Accepted for The IASTED International Conference on Software Engineering
Innsbruck, Austria, February 15-17, 2005.

Static	Dynamic
Lines of Code (LOC)	Line/Instruction Coverage
McCabe Cyclomatic Complexity	Branch/Edge Coverage

Figure 1. Two basic static metrics and their dynamic equivalents.

many limitations it is still the universal first step in program measurement. The McCabe cyclomatic complexity of a program quantifies the number of decisions in the source code, and is often used as a rough guide to gauge the effort required for testing or maintenance.

Many other more complex or more specific metrics exist [6]. The purpose of choosing such basic measures is that their use is commonly accepted, and is not usually deemed to be biased toward any particular area of application. In Figure 1 we also identify two *dynamic* counterparts to these metrics, drawn from the domain of software testing. These are discussed in the following subsection.

2.1 The Role of Coverage Criteria

Dynamic coverage measures are typically used in the field of software testing [7, 8, 9, 10] as an estimate of the effectiveness of a test suite. The basis of software testing is that software functionality is characterised by its execution behaviour. Clearly, improved test coverage leads to improved fault coverage and improved software reliability [11]. From our perspective, higher execution coverage by a program in a benchmark suite (measured as test coverage) means that execution of the program provides a better characterisation of that program's behaviour.

There are a number of established analyses for determining test coverage in a program from a control-flow viewpoint. These include statement coverage, decision coverage, condition coverage, decision/condition coverage, multiple condition coverage [7], and modified condition/-decision coverage (MC/DC) [12]. As is the case with some static software metrics, there can often be subtle differences in the definition of these coverage measures. However, each of them seeks to measure, typically as a percentage, the degree to which a certain aspect of the program source code has been exercised at run-time.

The most fundamental form of coverage, statement coverage, is achieved if every source language statement in the program is executed at least once. This can be measured at run time by instrumenting every basic block, or vertex in the control-flow graph [13]. This relates directly to the size of the program, and a low statement coverage implies that large sections of the program are not being exercised during the dynamic analysis.

The other coverage analyses (decision coverage, condition coverage, multiple decision coverage, and decision/condition coverage) can be measured by instrumenting every branch in the source code, corresponding to tracing edges in the control flow graph [13]. It is important to differentiate between different branches to the same destination in order to measure full edge coverage of the predicate graphs [12]. This measurement is different from profiling and tracing as defined in [13] in that neither the sequencing, nor the count of edge executions is relevant; for measuring control-based test coverage, just a boolean flag indicating execution of each edge is sufficient.

For many programming languages, attempts to compare run-time behaviour to static data is complicated by the difference between the compiled code and the original source code. With Java, however, source code is compiled into bytecode, which retains many of the features of the original code. In particular, the simplicity of bytecode means that we can unambiguously identify the quantities being measured. Using bytecode-level measures may sometimes result from necessity: it is not always the case that the original Java source code is made available for analysis.

Thus, in the remainder of this paper we focus on the static manipulation, and the static and dynamic analysis of Java programs at the bytecode level. In particular, we use the bytecode instruction as our unit of measurement for LOC and line coverage. Similarly we equate the conditional jump instruction with predicates or decision-points in the program, and use these as a basis for calculating the cyclomatic complexity and the run-time decision coverage. Working at the bytecode level also considerably facilitates the identification of edges in the control-flow graph. To avoid confusion with the numerous methods of calculating decision coverage mentioned above, we call this *edge coverage*.

At run-time, edge coverage at the bytecode level lies between condition/decision coverage and modified condition/decision coverage, at the source code level. Short-circuit evaluation of multiple conditions in boolean expressions limits the number of possible edges (at the object level) to 2N (short-circuit operator extensions to MC/DC). But impossible branches, due to coupled conditions, are not easily identified at the bytecode level. This form of coverage is discussed as “object code coverage” in [12].

Deciding what percentage constitutes an acceptable level of coverage is a necessarily empirical task, but estimates tend toward the highest quartile. Piwowarski et al. [14] deem 70% sufficient for statement coverage, with anything under 50% regarded as generally insufficient. Malaiya et al. [15] note that 80% branch coverage “often produces acceptable results”, whereas Grady [16] considers 85% branch coverage to be an appropriate figure for ensuring good characterisation. Devanbu et al. [17] suggest a range of 80%-90% branch coverage has a high likelihood of exposing software faults. It is not obvious however that all these estimates are based on objective quantitative data, nor that they are all independent.

```
S1.   count = 0;
S2.   n = fileptr.read();
P3.   while( fileptr.ready() ) {
P4.       if (n>=0)
S5.           return(error);
          else {
S6.           numarray[count] = n;
S7.           count++;
          }
S8.       n = fileptr.read();
      }
S9.   avg = calcavg(numarray,count);
S10.  return(avg);
```

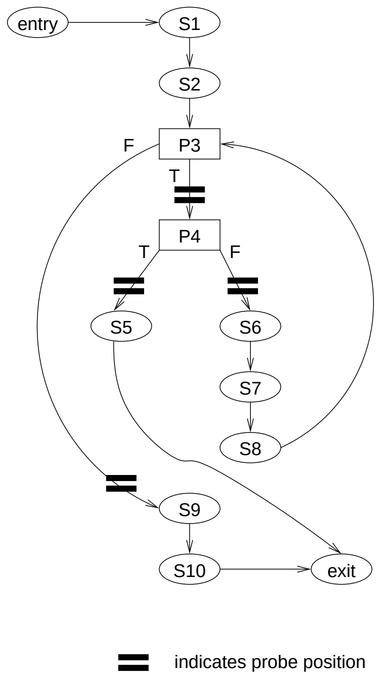

Figure 2. A piece of Java code, along with the corresponding control-flow graph, indicating the location of the edge-coverage probes.

3 Implementation Details

As has been previously noted [18], the Java runtime system provides an ideal environment for dynamic analysis. Typically, such analysis can be conducted at three main levels of granularity:

1. *Instrumenting a JVM*: There are several open-source implementations of the JVM available, such as Kaffe, Jalapeño or the Sable VM, and access to their source code means that all aspects of a running Java program can be observed.

2. *Using the Java Virtual Machine Debug Interface (JVMDI)*: Version 1.4 and later of the Java SDK support a debugging interface that provides event notification for low-level JVM operations. A trace program that handles these events can thus record information on the execution of the Java program.

3. *Instrumenting the bytecode*: This involves statically manipulating the bytecode to insert probes, or other tracing mechanisms, that record information at runtime.

The last approach, involving bytecode manipulation, provides the simplest method of dynamic analysis since it does not require low-level knowledge of JVM internals, and imposes little overhead on the running program.

In what follows we have run the programs from a number of benchmark suites and collected line, instruction and edge coverage data. The data was collected by instrumenting the programs at the bytecode level, inserting probes to collect relevant events. The basic framework used the Byte Code Engineering Library [19], along with the Gretel Residual Test Coverage Tool [20] to perform line and instruction coverage.

The Gretel tool statically determines the basic blocks in a Java class file and inserts a probe consisting of small sequence of bytecode instructions at each basic block. Whenever the basic block is executed, the probe code records a "hit" as a simple boolean value. The number of bytecode instructions in the basic block can then be used to calculate instruction coverage. Furthermore, if the class file contains a line number table, mapping bytecode instructions to line numbers in the original Java program, line coverage can also be deduced from this data. We note that line coverage corresponds to lines of code in the Java source program, and that this may not correspond to statement coverage.

3.1 Implementing Edge Coverage

We have written a complimentary tool to Gretel which manipulates bytecode, inserting probes to calculate edge coverage. Each conditional jump in the bytecode is associated with a probe-pair, recording the true and false evaluations of the conditional. The probe to record the false edge is simply inserted immediately after the jump instruction, while the probe to record the true edge is inserted at the target of the jump instruction. The probe at the jump target is preceded by a goto instruction to ensure that it is not executed under any other circumstances, such as a "fall-through" from a source-level `if` or `switch` statement. We note that unconditional jumps are not instrumented, since they do not correspond to a predicate in the source code.

We note that this approach does not correspond exactly to counting conditions in the original Java code. In particular, short-circuit boolean operators (&& and `||`) can cause an early exit from evaluation, represented by a conditional jump in the bytecode. Thus a boolean expression of the form `(a && b)` would count for two possibilities in the original Java code, but counts for four possibilities in our calculation.

Our implementation of edge coverage also deals with loops, since these are implemented by conditional jumps in the bytecode. For each loop we register two possibilities depending on whether it is executed zero or more times for a `while` and `for` loop, or one or more times for a `do-while` loop. Such calculations could be affected by intensive bytecode optimisations, where conditional code might be moved or merged. However, most Java compilers carry out very little such optimisation by default [21].

To illustrate the functionality of our edge-coverage tool, Figure 2 shows an example of some Java code, adapted from an example by Rothermel and Harrold [22]. Here, probes are inserted to record the outcomes of both predicates, $P3$ and $P4$, so that running the program will produce an edge coverage as a fraction of 4. This example also highlights the difference in granularity between statement and edge coverage. For example, the two simplest paths through this program are:

Path 1: $entry \rightarrow S1 \rightarrow S2 \rightarrow P3 \rightarrow S9 \rightarrow S10 \rightarrow exit$
Path 2: $entry \rightarrow S1 \rightarrow S2 \rightarrow P3 \rightarrow P4 \rightarrow S5 \rightarrow exit$

Both paths 1 and 2 correspond to 50% line coverage (counting the predicates), but edge coverage can distinguish the cases, yielding 25% for path 1 and 50% for path 2.

The McCabe cyclomatic complexity of a program can be defined as the number of decisions occurring statically in the program code, plus one [6]. Since each probe-pair we insert is associated with a single decision point, the number of probe-pairs occurring statically in the code provides a measure of the cyclomatic complexity. Specifically, if a program has a McCabe cyclomatic complexity value of m, the "edge count" (i.e. number of probes) recorded below corresponds to $(m-1)*2$, since we record two possible outcomes for each decision.

4 Coverage Results

In this section we examine several benchmark suites for Java programs, and detail instruction and branch counts, along with the associated coverage data.

The instruction counts reported below correspond to `size.appLoad.value` of the Sable group's dynamic metrics [23], and our coverage percentage corresponds to `size.appRun.value` expressed as a percentage of `size.appLoad.value`. Despite the fact that edge coverage is regarded as one of the most fundamental coverage criteria, we are not aware of any work to date that has applied them to Java benchmark suites.

4.1 Experimental Environment

In this section we report results from a variety of Java programs, taken from a number of popular benchmark suites. We use version 1.03_05 of the SPEC JVM98 benchmark suite [24], version 3.0 of the CaffeineMark suite [25], and section 3 of version 2.0 of the Java Grande benchmark suite [26]. There appears to be just one version of the JOlden suite, dated March 6, 2003 [27].

All the programs except those in the SPEC and CaffeineMark suites were compiled using the *javac* compiler from Sun's SDK version 1.4.2, and all benchmarks were run using the client virtual machine from this SDK. The programs in the SPEC and CaffeineMark suites are distributed in class file format, and were not recompiled or otherwise modified. We note (in accordance with the licence) that the SPEC programs were run individually, and thus none of these results are comparable with the standard SPEC JVM98 metric. All benchmark suites include not just the programs themselves, but a test harness to ensure that results from different executions are comparable.

In all cases, only the bytecode corresponding to the benchmark programs was instrumented, and thus the results presented below do not include bytecode instructions executed in the Java class library. While excluding the class libraries may not always be appropriate for performance-based analysis, it is clearly a sensible approach from a coverage perspective.

4.2 Results for Application Programs

The instruction and edge coverage results for the programs in the SPEC JVM98 benchmark suite are given in Table 1. For each program we show the instruction count, corresponding to the number of bytecode instructions occurring statically in the class file, and the edge count, which is twice the number of conditional jumps occurring statically in the bytecode. We do not give line number coverage details for the SPEC suite since line number tables were not available for four of the programs, and may be unreliable for the others. We note that for most of the programs in the other suites, instruction coverage seems to approximate line coverage fairly well.

The columns labelled 1, 10 and 100 correspond to the instruction and edge coverage percentages when the SPEC programs are run at sizes 1, 10 and 100. Size 100 is the standard, reportable size for the benchmarks, and the other sizes represent 1% and 10% of this size. As we can see from Table 1, none of the SPEC programs have complete instruction coverage and three programs, `jess`, `db` and `javac` have particularly low coverage. One consequence

spec98 Program	Instr Count	% Instr Coverage			Edge Count	% Edge Coverage		
		size 1	size 10	size 100		size 1	size 10	size 100
_201_compress	2045	63	63	64	172	72	72	73
_202_jess	19309	35	34	36	1836	36	36	39
_205_raytrace	5839	80	80	81	446	76	77	80
_209_db	1867	41	39	36	215	47	47	58
_213_javac	43614	16	42	45	5535	14	48	51
_222_mpegaudio	45760	92	93	93	683	60	62	61
_228_jack	18480	75	75	75	1545	67	67	67

Table 1. Instruction and edge coverage percentages for the SPEC JVM98 benchmark suite, run at sizes 1, 10 and 100.

Grande Program	Line Count	% Line Coverage		Instr Count	% Instr Coverage		Edge Count	% Edge Coverage	
		size A	size B		size A	size B		size A	size B
Euler	490	85	85	9601	88	88	148	90	90
MolDyn	198	98	98	2003	97	97	90	98	98
MonteCarlo	675	55	55	4070	53	53	154	36	36
RayTracer	277	89	89	2424	81	81	72	91	91
Search	181	79	79	1754	81	81	164	78	78

Table 2. Line, instruction and edge coverage percentages for section 3 of the Java Grande benchmark suite, run at sizes A and B.

CaffeineMark Program	Instr Count	% Instr Coverage	Edge Count	% Edge Coverage
Dialog	310	99	4	75
Float	34	94	12	91
Graphics	87	97	8	87
Image	382	71	16	68
Logic	75	70	108	61
Loop	107	98	10	90
Method	68	89	12	83
Sieve	50	96	16	87
String	85	97	6	83

Table 3. Instruction and edge coverage percentages for the CaffeineMark benchmark suite, run individually, at the standard size.

JOlden Program	Line Count	% Line Coverage	Instr Count	% Instr Coverage	Edge Count	% Edge Coverage
bh	278	78	1978	73	160	69
bisort	100	79	624	84	60	70
em3d	115	74	878	71	80	68
health	157	85	1082	88	106	79
mst	120	76	749	81	74	63
perimeter	129	87	871	81	116	83
power	169	87	2213	91	96	81
treeadd	54	38	378	58	32	59
tsp	154	87	988	86	79	78
voronoi	262	60	1840	58	104	50

Table 4. Line, instruction and edge coverage percentages for the JOlden benchmark suite, run with the default parameters.

of such low instruction coverage is that it will be difficult to correlate the results of static and dynamic analyses, since such a high proportion of the static code is not involved in the execution.

The results for edge coverage in Table 1 are somewhat poorer than those for line coverage, with only one program, `raytrace`, even making it as high as 80%. Thus, by Grady's criteria mentioned in Section 2.1 above, none of these programs would offer satisfactory coverage from a testing standpoint.

One notable feature of the SPEC JVM98 coverage data in Table 1 is the relatively small difference between the different run sizes. For all programs except `javac` there is virtually no difference in coverage between the run sizes suggesting that, at least for some analyses, the overhead of running the benchmarks at size 100 could be avoided. Indeed, for the `db` program the instruction coverage actually decreases for the larger input size, falling from 41% to 36% as we move from size 1 to size 100. The change in this case is due to a help method containing 112 instructions which is called for sizes 1 and 10, but not for size 100.

Table 2 gives the line, instruction and edge coverage for the five applications in section 3 of the Java Grande sequential benchmark suite. Since the Grande programs are distributed in source code format, line numbers are available, but we note that in any case, line coverage corresponds closely with instruction coverage. Overall, the line and instruction coverage for the Java Grande programs are a little better than for the SPEC programs, being above 80% for all except the `MonteCarlo` program. As with the SPEC suite, using the smaller versions with size A inputs makes little relative difference to the coverage results for the programs.

4.3 Results for Smaller Programs

The programs in the SPEC JVM98 and Java Grande section 3 suites are intended to model "real world" applications, and consist of full-fledged programs. However, a number of benchmark suites are designed to model specific aspects of Java programs, or specific types of operations. In this subsection we analyse two such benchmark suites: CaffeineMark, a set of micro-benchmarks for Java, and JOlden, a suite of programs intended to model memory-intensive programs.

Table 3 presents the instruction and edge coverage for the programs in the CaffeineMark suite. Since these programs were distributed as class files we do not include line numbers here. As can be seen from this table, these programs present a markedly different coverage profile to the SPEC and Java Grande suites, with instruction coverage at or over 89% for seven of the nine programs. As before, edge coverage is lower than instruction coverage, but is still over 85% for four programs, and at 83% for two others. Clearly, the simple, single-task design of these programs contributes to their higher coverage results.

Table 4 presents the line, instruction and edge coverage data for the programs in the JOlden benchmark suite. Again, these results appear much better overall then either the SPEC or Java Grande suites, with all but three of the programs having line coverage over 80%. We note, however, that the coverage results are not as good for the programs in the JOlden suite as for the CaffeineMark programs. This can be attributed to their increased complexity, measured in terms of the edge count. Using the relationship with the cyclomatic complexity defined above, we note that the cyclomatic complexity of the JOlden programs ranges from 17 (`treeadd`) to 81 (`bh`), whereas all but one of the CaffeineMark programs have a cyclomatic complexity under 9.

4.4 Going beyond the standard suites

As well as using the standard benchmark suites, many researchers use mixtures or subsets of these suites and, occasionally, augment these suites with programs that better reflect the problems encountered in their work. An example of this latter approach is the work by Ishizaki et al. [28], and Suganuma et al. [29] which uses the programs from the SPEC suite, but augments these with a number of more graphically-based programs of their own choosing.

We stress that we do not seek to criticise either of these works, and that we are addressing the use of these programs from a different perspective than the original authors. In addition, once standard benchmark programs are not being used, it becomes difficult to reproduce the exact environment of an experiment. Thus, the results we report in the remainder of this subsection should be considered to be indicative, and should not be directly correlated with the results reported in either [28] or [29]. In particular, the versions of the software used here are typically later than those cited in [28] and [29].

In Table 5 we list four of the programs used in both papers, which we have assessed using line, instruction and edge coverage criteria. The first two columns are the description gleaned from [28] and [29], the remaining column lists the version information for the programs used in our trial. Two of the programs are fully-fledged applications, whereas two are demo programs from the standard Java SDK distribution.

Table 5 shows the coverage results for running the four programs under these conditions. It is notable that the coverage results for all programs are relatively poor, with the edge coverage in particular falling considerably behind the benchmark suites. Also, as might be expected, the two demo programs, `Java2D` and `SwingSet` are exercised more fully then either `ICE` or `Jfig`.

Most of the poor coverage for the two applications can be accounted for by the minimal tasks performed; indeed, the `ICE` application only exercised 276 of 896 classes, whereas `Jfig` only exercised 58 out its 233 classes. When the non-used classes are excluded, the `ICE` instruction usage increases to 38%, and that for `Jfig` increases to 34%,

Program	Description	Version
ICE Browser	Simple Internet browser. Run the application and open a web page	version 5_4_3_1
Java2D	2D graphics library. Run the demo program as an application with options -runs=1, -delay=0	From the Java 2 SDK, version 1.4.2
Jfig	A Java version of the xfig drawing program. Run the application and open a document	version 2.20
SwingSet	GUI component. Run the demo program as an application to bring up the initial window	From the Java 2 SDK, version 1.4.2

Program	Line Count	% Line Covg.	Instr Count	% Instr Covg.	Edge Count	% Edge Covg.
ICE	34576	17	310029	20	50381	17
Java2D	4510	64	43544	73	2309	44
Jfig	19809	11	189873	13	8903	9
SwingSet	2904	67	30173	76	697	24

Table 5. Some of the programs used by Ishizaki et al., and their running conditions, along with line, instruction and edge coverage percentages.

which nonetheless is still a long way from satisfactory coverage.

5 Conclusion

In this paper we have noted the importance of comparing static and dynamic analyses of Java programs. In order to provide the foundation for such analysis, we have proposed that common static metrics should be evaluated for coverage in a dynamic context. We have presented the results of an analysis of several Java benchmark suites, showing the relationship between static and dynamic data.

The results presented in this paper lead to the following conclusions:

- The larger suites designed to measure "real-world" applications had poor instruction and edge coverage, so we conclude that static analysis will prove a poor predictor of dynamic behaviour.
- The suites with smaller programs had better coverage, but this probably resulted from much lower complexity, rendering the programs less suitable for wider analyses.
- The ad hoc addition of programs to benchmark suites is a perilous activity, and could benefit from a full analysis of the relevant metrics.

This paper is a first step in comparing static and dynamic analyses using coverage criteria. Some experimental results have indicated that testing to achieve branch coverage performs more poorly than testing to achieve data-flow coverage [30], [31]. It is likely that data-flow or path coverage techniques would provide a better indicator of the coverage provided by the execution of Java benchmark suites. We believe it important that existing and future benchmark suites should be evaluated using not just the usual static metrics, but also using the fullest possible range of coverage criteria.

In a recent paper, Sim et al. issued a challenge to the software engineering community to define benchmarks for use in their field [32] . They cited the importance of benchmark suites in other areas of computer science, and identified the importance of benchmark suites in achieving consensus on research goals and allowing for more a rigorous examination of research results. We hope that the work presented in this paper can complement any such movement toward the increased use of benchmark programs.

The Java programming language provides an apparently fertile environment for conducting dynamic analysis. Java programs can be readily analysed statically or dynamically, and at various levels of granularity. The availability of numerous Java programs, including many benchmark suites, seems to furnish an ideal basis for dynamic analysis.

This paper outlines and demonstrates a basis for making a comparison between static and dynamic analyses, through recording and comparing some base line measurements. The results in this paper suggest that caution must be exercised in the choice of a benchmark suite, or of programs from a benchmark suite, given the wide variance among the coverage results.

6 Acknowledgements

This work is partly funded by the Embark initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

References

[1] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Im-*

plementation, pages 115–128, San Diego, California, USA, June 9-11 2003.

[2] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Conference on Programming Language Design and Implementation*, pages 278–288, San Diego, California, USA, June 9-11 2003.

[3] F. Umemori, K. Konda, R. Yokomori, and K. Inoue. Design and implementation of bytecode-based java slicing system. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 108– 117, Amsterdam, The Netherlands, Sept. 26-27 2003.

[4] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 187–204, Anaheim, CA, USA, October 26-30 2003.

[5] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, 2003.

[6] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, 1996.

[7] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons, February 1979.

[8] M. Roper. *Software Testing*. McGraw Hill, September 1994.

[9] R. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison Wesley, October 1999.

[10] J.D. McGregor and D.A. Sykes. *A Practical Guide to Testing Object-oriented Software*. Addison Wesley, March 2001.

[11] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51(4):420–426, December 2002.

[12] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[13] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[14] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *15th International Conference on Software Engineering*, pages 287–301, Baltimore, Maryland, USA, May 17-21 1997.

[15] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical Report CS-96-128, Colorado State University, 1996.

[16] R. E. Grady. *Practical Software Metrics for Project Management and Process improvement*. Prentice-Hall, 1992.

[17] P.T. Devanbu and S.G. Stubblebine. Cryptographic verification of test coverage claims. *IEEE Transactions on Software Engineering*, 26(2):178–192, February 2000.

[18] Claire Knight. Smell the coffee! uncovering Java analysis issues. In *International Workshop on Source Code Analysis and Manipulation*, pages 161 – 167, Florence, Italy, November 10 2001.

[19] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, April 3 2001.

[20] C. Howells. Gretel: An open-source residual test coverage tool, June 2002. http://www.cs.uoregon.edu/research/-perpetual/Software/Gretel/.

[21] 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*, 15(3-5):459–484, March 2003.

[22] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[23] Sable Research Group, McGill University. Dynamic software metrics, April 24 2003. http://www.sable.mcgill.ca/metrics/.

[24] Standard Performance Evaluation Corporation. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.specbench.org/osg/jvm98/-press.html.

[25] Pendragon Software Corporation. Caffeinemark 3.0, May 13 1999. http://www.benchmarkhq.ru/cm30/info.html.

[26] J.M. Bull, L.A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *ACM Java Grande / ISCOPE Conference*, pages 97 – 105, Stanford University, California, USA, June 2-4 2001.

[27] B. Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona Spain, September 8-12 2001.

[28] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Object Oriented Programming Systems Languages and Applications*, pages 294 – 310, Minneapolis, Minnesota, USA, October 15-19 2000.

[29] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 180–194, Tampa, Florida, USA, October 14-18 2001.

[30] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch tetsing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[31] M. Hutchins, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 16-21 1994.

[32] S.E. Sim, S.M. Easterbrook, and R.C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering*, pages 74–83, Portland, Oregon, USA, May 3-10 2003.