Bigram Analysis of Java Bytecode Sequences

Diarmuid O'Donoghue[1] Aine Leddy[1] James Power[1]
John Waldron[2]

[1] Dept. of Computer Science, National University of Ireland, Maynooth, Ireland.
[2] Dept. of Computer Science, Trinity College Dublin, Ireland.

1 Introduction

Much research has been conducted in the analysis of Java bytecodes in order to gain a better understanding of how Java programs behave. One branch of this research has focused on analysing bytecode usage within the Java Virtual Machine (JVM), with particular emphasis on analysing bytecodes associated with various benchmark programs.

Previous research has focused on the frequencies of the individual bytecodes at the *static* class-file level [2]. Another branch examines *dynamic* bytecodes, as executed by the JVM itself at run-time [4, 6]. This project follows on from previous dynamic bytecode analysis, analysing streams of Java bytecodes produced at the platform independent level. It differs from previous projects, in that it is not concentrating on the occurrences of the individual bytecodes, but in the occurrences of *bigrams*, or bytecode pairs.

We report on a project that performed a bigram analysis of dynamic bytecode sequences. The objective was to identify the most commonly used bytecode pairs, and to examine the relative frequency of occurrence of these bytecodes. In all, 12 large Java programs were analysed, taken from the Java Grande and SPEC benchmark suites. Our findings are of relevance to research into instruction set design and implementation, as well as JVM optimisation.

2 Bigrams

Bigrams [7] are widely used in statistical "natural language" processing, and most significantly play a vital role in Hidden Markov models. Applications abound, but include context sensitive spell checking, voice-to-text systems and grammar checking. Bigram analysis typically uses a corpus of text to learn the probability of various word pairs, and these probabilities are later used in recognition. However, in this project we are only interested in the data collection phase of bigram usage.

The joint probability of a word (bytecode) sequence can be expressed as the product of individual word (bytecode) probabilities, each conditioned on the preceding word (bytecode). If we assume that a word sequence is valid (i.e. a sentence), then intuitively we say that the word sequence *"the black cat"* is more likely to be followed by the word *"purred"* than by the word *"crystalography"*.

Assume that $P(b_2|b_1)$ is the probability of bytecode b_2 given bytecode b_1. Then we wish to record the observed sequence frequencies for all bytecode pairs $P(b_2|b_1)$. Note that this

Accepted for the Second Workshop on Intermediate Representation Engineering for the Java Virtual Machine Trinity College, Dublin, Ireland, June 13-14, 2002, pp. 187-192

Java Grande		
Program	**bytecodes executed**	
	total	unique
euler	1.46e+10	143
moldyn	7.60e+09	141
monte	2.63e+09	152
raytr	1.18e+10	149
search	7.10e+09	140

SPEC JVM98		
Program	**bytecodes executed**	
	total	unique
cmprs	1.25e+10	122
jess	1.91e+09	143
db	3.77e+09	119
javac	2.43e+09	144
mpeg	1.15e+10	153
mtrt	2.20e+09	141
jack	1.50e+09	131

Figure 1: The 12 programs used in our analysis were taken from the Java Grande and SPEC JVM98 benchmark suites. Here we show the total number of bytecodes executed for each of the applications, along with the number of unique bytecode instructions actually used.

follows directly on from previous work [4, 5], which recorded the frequencies of individual dynamic bytecodes. From a different perspective, we also interested in identifying, for example, any non-occurring bigrams where $P(b_2|b_1)$ is 0.

3 Results

This corpus of test programs consisted of two Java benchmarks suites, the Java Grande Benchmark Suite [3] containing five benchmark-programs, and the SPEC JVM98 Benchmark Suite [9] which contains seven programs. Version 2.0 of the Java Grande benchmark suite (Size A) was used, and the programs in this suite were compiled using SUN's *javac* compiler, Standard Edition (JDK build 1.3.0-C). The programs in the SPEC suite were distributed as bytecode files, and were not recompiled.

The advantage of using these two benchmark suites is that the dynamic frequencies of the individual bytecodes have already been analysed for these suites in [4] and [5, 6] respectively. Further, both suites provide examples of long sequences of instructions - ranging from 1.5×10^9 bytecodes (*jack*, in the SPEC suite) to 1.5×10^{10} bytecodes (*euler*, in the Java Grande suite). A brief summary of these programs is given in Figure 1; more information can be found in [3, 9].

The results in this section we compiled by running the Grande and SPEC benchmark suites on an instrumented JVM. The Kaffe Virtual Machine, version 1.0.6, was used, with JIT compilation switched off. Each executed bytecode was recorded and measured, so that these results reflect not only bytecode from the benchmark programs themselves, but also from the Kaffe class library. It should be noted that the Kaffe class library is not 100% compliant with SUN's JDK, and may differ from other Java class libraries.

3.1 Frequently occurring bigrams

The table in Figure 2 shows the top ten most frequently occurring bigrams across all 12 benchmark programs. Note that these constitute a *weighted* average, to smooth differences between the total bytecode counts for individual programs.

Rank (R*)	Bigram		Overall Frequency	Rank AVG	Rank STDEV
1	aload_0	getfield	9.21 %	3.8	8.9
2	invokevirtual	aload_0	2.35 %	40.7	62.0
3	iload	aaload	1.91 %	199.3	254.8
4	aload_1	getfield	1.40 %	152.6	156.5
5	getfield	iload	1.40 %	68.9	68.4
6	dload	dload	1.38 %	114.7	340.0
7	getfield	aload_0	1.29 %	37.8	44.9
8	putfield	aload_0	1.13 %	32.4	27.7
9	aaload	iload	1.06 %	67.1	161.1
10	getfield	aload_1	0.95 %	154.8	163.1

Figure 2: Top ten most frequently used bytecode bigrams, over all 12 benchmark programs. The overall frequency and rank reflects a weighted average of the 12 individual program traces.

We can see that the most common bigram in the combined results is made up of the bytecodes `aload_0` and `getfield`. The bytecode `aload_0`, is a load reference from the local variable at index 0, which would hold the `this`-pointer in an instance method. The `aload_0` instruction occurs in three more of the top ten instructions. The `getfield` instruction occurs in four more of the top ten bigrams.

The bigram at rank seven is made up of the same bytecodes as the top ranked bigram - but in reverse order. This is interesting as it has been previously discovered in [4] that the these two bytecodes were in the top four most frequently executed bytecodes for four out of the five Java Grande benchmark programs. In analysis of the SPEC JVM98 benchmark suite [6], these two bytecodes had the highest frequency on average. From the bigram results it can be seen that these bytecode are most often executed together *in sequence*.

The combined rank hides much variation that occurs within the data for each individual benchmark program. We compare the ranking and percentage frequency of each bigram rather than its occurrence count, as this avoided the tendency of a large program to overwhelm the statistics generated by shorter programs. The top ranked bigram (`aload_0` and `getfield`) was also the top ranked bigram in eight of the ten benchmark programs. On the other two programs this bigram was ranked 3^{rd} and 32^{nd}, and the standard deviation of the rank value was under 9. Figure 3 shows the rankings of the overall top 10 bigrams in each of the individual benchmark programs. We note that *moldyn*, a translation of a Fortran program, is the only program not to show a high frequency of occurrence of the `aload_0 getfield` bigram. The methods in this program are mainly static, and hence do not have a `this` pointer at position 0.

3.2 Bigram coverage

In this section we examine how many of the total possible number of bigrams were actually used by the benchmark programs. In a previous studies of bytecode usage, it was found that, taken together, the Grande suite actually only uses 169 of the possible bytecodes, whereas the SPEC suite uses only 186 of the bytecodes - or 85% and 93% of the 199 possible bytecodes, respectively. The number of unique instructions used for each program in both of the suites is given in Figure 1.

R*	monte		search		moldyn		raytr		euler	
	R	F %	R	F %	R	F %	R	F %	R	F %
1	1	12.14	1	6.84	32	0.10	1	13.07	3	8.05
2	8	1.87	71	0.39	91	0.00	15	1.83	215	0.01
3	312	0.00	137	0.18	5	4.24	31	0.90	1	14.63
4	124	0.02	432	0.00	226	0.00	2	9.94	65	0.18
5	72	0.38	5	1.60	132	0.00	30	0.90	4	6.33
6	3	2.33	0	0.00	1	14.00	133	0.01	52	0.25
7	10	1.55	111	0.21	43	0.03	13	1.93	12	1.67
8	19	1.38	60	0.45	44	0.03	12	2.00	8	1.96
9	0	0.00	0	0.00	0	0.00	0	0.00	2	10.15
10	140	0.01	442	0.00	248	0.00	3	7.24	64	0.19

R*	jack		jess		mtrt		javac		db		cmprs		mpeg	
	R	F %	R	F %	R	F %	R	F %	R	F %	R	F %	R	F %
1	1	12.10	1	8.43	1	13.40	1	10.94	1	10.35	1	9.43	1	5.68
2	5	1.78	2	4.49	2	11.90	2	2.92	34	0.69	11	1.65	32	0.63
3	855	0.00	78	0.33	3	0.06	196	0.12	426	0.00	342	0.00	6	2.51
4	82	0.22	6	1.60	418	0.00	129	0.20	3	4.12	303	0.00	41	0.56
5	188	0.10	59	0.39	187	0.06	94	0.29	5	3.42	47	0.59	4	2.76
6	0	0.00	1187	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
7	4	2.43	95	0.28	123	0.11	3	1.94	11	2.31	9	2.16	19	0.87
8	2	2.78	31	0.67	9	1.60	5	1.58	87	0.16	59	0.51	53	0.40
9	0	0.00	502	0.00	296	0.03	0	0.00	0	0.00	0	0.00	5	2.59
10	90	0.19	35	0.61	62	0.26	33	0.48	9	2.43	273	0.00	459	0.01

Figure 3: The top-ten bigrams for each benchmark program. Here we show the rank, R, and frequency, F %, of each of the overall top 10 bigrams, for each program in the suite. These results are sorted based on the first column, R*,the overall bigram rank.

When we extend this analysis to bigrams, we find that the coverage is much lower. Figure 4 compares the number of unique bigrams used in each benchmark program, against the maximum possible number of bigrams. The maximum possible number of bigrams for a program is the square of the number of unique bytecode instructions used, as given in Figure 1.

Figure 5 illustrates another key finding of our study. That is, that the frequency of usage of bigrams obeys the power law. A very small number of bigrams occur *very* frequently, while most bigrams occur with a small degree of frequency. These results are most startling when we remember that our previous results indicate that only a small percentage of possible bigrams occur in the first place. A similar distribution is observed when the benchmark programs are considered individually.

The first part of the most common bigram is `aload_0`. As previously stated, the most common successor is the `getfield` instruction, which accounts for 9.6% of all identified bigrams. The second most common bigram starting with the `aload_0` instruction is `aload_1` accounting for 0.69% of identified bigrams. Interestingly, the distribution of the successors of `aload_0` also follows the power law.

4 Conclusion

We have examined bytecode usage within the JVM for the Java Grande and SPEC benchmark suites. We recorded and analysed the bytecode bigram usage - that is pairs of bytecodes executed in sequence.

Our results illustrate that there are a very small number of bytecode pairs that are used with very high frequency, most noticeably the bigram `aload_0` followed by `getfield`. An interesting result was that the frequency of bigram usage obeyed the power law, with logarithmically decreasing frequency across the less frequent bigrams. This indicates that

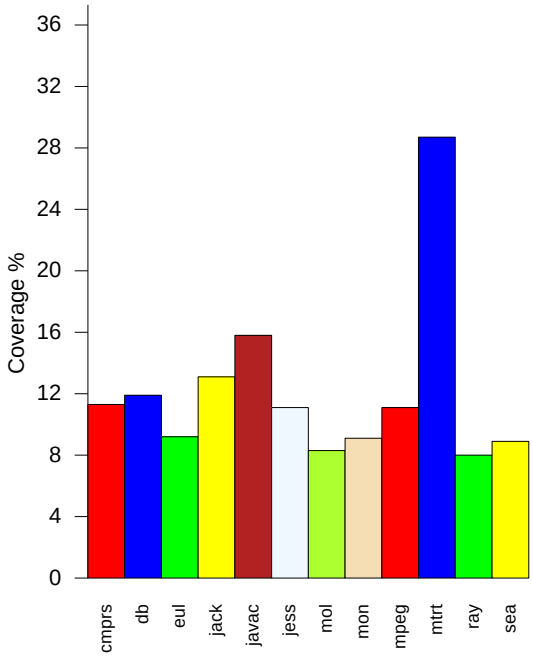

Figure 4: Observed Coverage of Possible Bigrams. Here we show the number of unique bigrams used as a percentage of the square of the number of unique bytecodes used for each program.

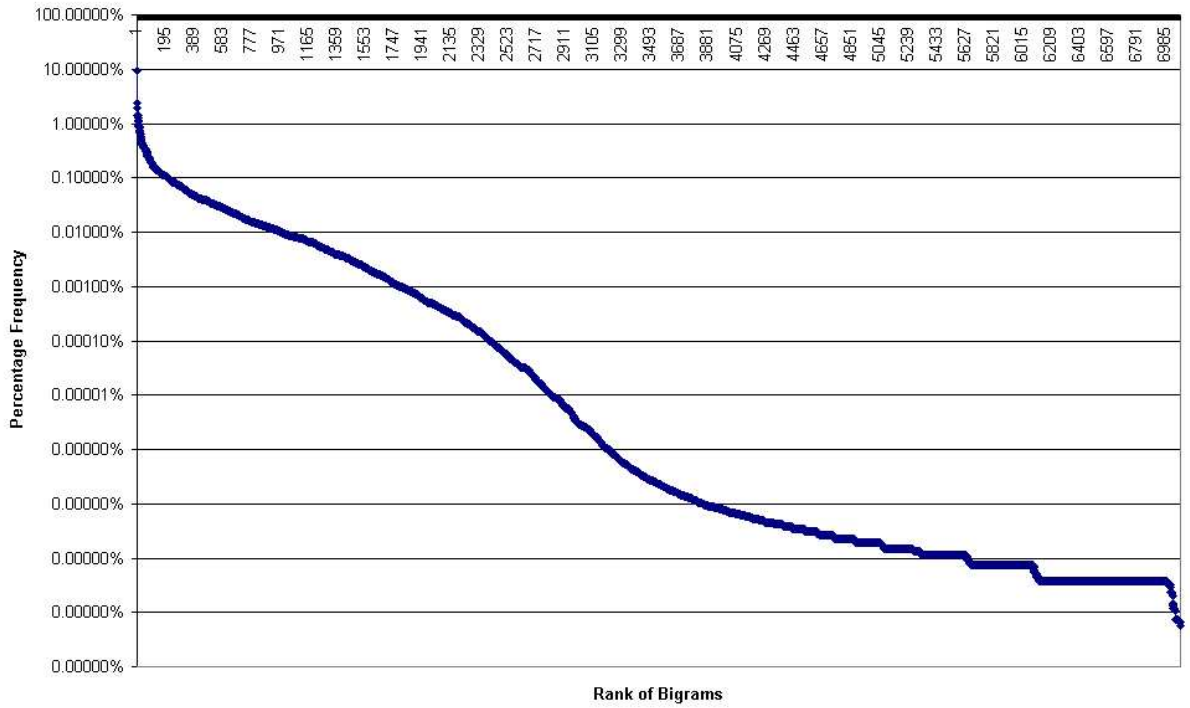

Figure 5: Bigram usage Frequencies. This graph plots the overall usage frequencies for the observed bigrams, and shows them conforming to a power distribution.

optimising should focus on these most frequently used bytecode bigrams.

We note that the most common bigram, `aload_0` followed by `getfield`, accounted for nearly 10% of the bigrams executed in most programs. Many standard JVMs will apply extensive optimisations to the bytecode including, at least, register allocation to eliminate many of the load and store operations. However, JVMs which directly interpret the code, such as those operating in restricted environments, could save up to 10% in the instruction fetch/decode cycle if `aload_0` and `getfield` were combined into a single instruction.

We believe that a study of the dynamic behaviour patterns of bytecode programs is an important foundation for the future design of intermediate representations. The work presented above is an initial study; in future work we hope to measure the impact of optimisations based on this data.

References

[1] L.A. Adamic. *Zipf, Power-laws, and Pareto - a ranking tutorial.* `http://ginger.hpl.hp.com/shl/papers/ranking/ranking.html` Internet Ecologies Area, Xerox Palo Alto Research Center, Palo Alto, CA 94304

[2] D.N. Antonioli and M. Pilz. *Analysis of the Java Class File Format.* Technical Report ifi-98.04, Dept. of Computer Science, University of Zurich, 1998.

[3] M. Bull, L. Smith, M. Westhead, D. Henty and R. Davey. *Benchmarking Java Grande Applications.* In Proc. Conference on the Practical Applications of Java, Manchester, UK, April 2000.

[4] C. Daly, J. Horgan, J. Power, J. Waldron. *Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite.* Joint ACM Java Grande - ISCOPE 2001 Conference, Stanford University, USA, June 2-4, 2001.

[5] D. Gregg, J. Power, J. Waldron. *Benchmarking the Java Virtual Architecture - The SPEC JVM98 Benchmark Suite.* Chapter 1 of *Java Microarchitectures*, Editors N. Vijaykrishnan and M. Wolczko, Kluwer Academic, 2002.

[6] C. Herder and J.J. Dujmovic. *Frequency Analysis and Timing of Java Bytecodes.* Technical Report SFSU-CS-TR-00.02, Computer Science Dept, San Francisco State University, USA, 2000.

[7] D. Jurafsky and J. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing.* In Speech Recognition and Computational Linguistics, Prentice-Hall, 2000.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[9] Standard Performance Evaluation Corporation. . *The SPEC JVM98 Benchmark Suite*, `http://www.spec.org/`, August 1998.

[10] T.J. Wilkinson. *KAFFE, A Virtual Machine to run Java code.* `http://www.kaffe.org/`, 2000.