Dynamic Analysis of the Java Virtual Machine Method Invocation Architecture

SIOBHÁN BYRNE
Ericsson Telecom, Dun Laoghaire,
Co.Dublin, IRELAND.

CHARLES DALY
Computer Applications, Dublin City University,
Dublin 9, IRELAND.

DAVID GREGG
Computer Science, Trinity College Dublin,
Dublin 2, IRELAND.

JOHN WALDRON
Computer Science, Trinity College Dublin,
Dublin 2, IRELAND.

Abstract: - Platform independent dynamic analysis has been shown to be an important technique for performance analysis and workload characterization of programs that run on the Java Virtual Machine. In this paper we explore how this methodology can me used to study method invocation. We identify differences in program behaviour and propose a metric to predict dynamic compilation efficiency.

Key-Words: - Java Virtual Machine, method invocation, performance analysis, workload characterization

1 Introduction

The Java paradigm for executing programs is a two stage process. Firstly the source is converted into a platform independent intermediate representation, consisting of bytecode and other information stored in class files [15]. The second stage of the process involves hardware specific conversions, perhaps by a JIT or hotspot compiler for the particular hardware in question, followed by the execution of the code. This research extends existing research involving dynamic analysis at the platform independent bytecode level [7], and investigates how the method invocation architecture is dynamically used by real programs from two widely used benchmark suites — the SPEC JVM98 benchmark suite [20] and the Java Grande Forum Benchmark Suite [5].

The increasing prominence of internet technology, and the widespread use of the Java programming language has given the Java Virtual Machine (JVM) an important position in the study of compilers and related technologies. To date, much of this research has concentrated in two main areas:

- Static analysis of Java class files, for purposes such as optimisation [22], compression [3], software metrics [6], or the extraction of object models [11]
- The performance of the bytecode interpreter, yielding techniques such as Just-In-Time (JIT) compilation [1] and hotspot-centered compilation [21]. See [12] for a survey.

2 Benchmark Suites

The SPEC JVM98 [20] suite was designed as an industry-standard benchmark suite for measuring the performance of client-side Java applications, and we have used the seven main programs from this suite (see Figure 1). Some of the programs from the SPEC

The Java Grande Forum Benchmark Suite Large Scale Applications	
eul	Computational Fluid Dynamics
mol	Molecular Dynamics simulation
mon	Monte Carlo simulation
ray	3D Ray Tracer
sea	Alpha-beta pruned search

Standard Performance Evaluation Corporation JVM98 Benchmarks	
compress	Modified Lempel-Ziv method (LZW)
db	Performs multiple database functions on memory resident database
jack	A Java parser generator that is based on PCCTS
javac	This is the Java compiler from the JDK 1.0.2.
jess	Java Expert Shell System is based on NASA's CLIPS expert shell system
mpeg	Decompresses ISO MPEG Layer-3 audio files
mtrt	A raytracer with two threads each rendering a scene

Figure 1: *The programs used in this analysis*. There were 12 programs in total, taken from the Java Grande Forum sequential benchmarks and from the SPEC JVM98 benchmarks.

suite are distributed in bytecode format only. The Java Grande Forum Benchmark suite is distributed in source code format. For this study, these programs were compiled using the Java compiler from SUN's JDK, version 1.3 (see Figure 1).

We believe the suites chosen are as close as possible to "standard" based on published results. Many other suites of benchmark programs for Java exist, including micro-benchmarks such as CaffeineMark [17] Richards and DeltaBlue [24]. As these measure small, repetitive operations, it was felt that their results would not be typical of Java applications. For the same reason larger suites, designed to test Java's threads or server-side applications, such as SPEC's Java Business Benchmarks (SPECjbb2000), the Java Grande Forum's Multi-threaded Benchmarks, IBM's Java Server benchmarks [4] or VolanoMark [16] have not been included at this point.

Studies of the SPEC and Grande suites have concentrated on performance issues for various JVMs. Studies of the Grande suite include performance-related measures such as [5], as well as dynamic bytecode level views [7]. There have been a number of studies of the SPEC JVM98 benchmark suite. [20] provides speed comparisons of the suite using different Java Platforms, and [10] examines the speed impact of various optimisations. [9] uses the SPEC JVM98 suite in an examination the prediction rate achieved by invoke-target and other predictors. Both [14] and [18] discuss low-level timing and cache performance for the suite. [19] also looks at cache misses, but from the perspective of the SPEC JVM98 programs' memory behaviour. This theme is investigated in depth in [8], which studies the allocation behaviour of the SPEC JVM98 suite from the perspective of memory management. Both [19] and [26] note that the SPEC JVM98 suite may not be suitable for assessing all types of Java applications.

Program	Classes loaded	Program %	API %
Compress	95	17.9	82.1
JESS	236	58.9	41.1
Database	96	8.3	91.7
javac	237	61.2	38.8
mpegaudio	127	37.8	62.2
mtrt	112	26.8	73.2
jack	140	37.1	62.9
ave	149	35.4	64.6

Table 1: *Measurements of total number of classes loaded by SPEC JVM98 applications Also shown is the percentage of the total which are in the API and the program itself.*

3 Methodology

The data presented in this paper were gathered by running each of the programs independently on a modified JVM. The JVM used was Kaffe [23], an

Program	Classes loaded	Program %	API %
eul	96	7.3	92.7
mol	96	7.3	92.7
mon	104	15.4	84.6
ray	95	15.8	84.2
sea	87	9.2	90.8
ave	95	11.0	89.0

Table 2: *Measurements of total number of classes loaded by Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C). Also shown is the percentage of the total which are in the API and the program itself.*

Program	Methods invoked	Program %	API %	native %
Compress	343	16.6	83.4	9.6
JESS	798	52.4	47.6	5.0
Database	398	12.6	87.4	8.5
javac	1136	65.5	34.5	3.5
mpegaudio	506	43.9	56.1	6.9
mtrt	509	34.8	65.2	8.6
jack	613	46.7	53.3	5.4
ave	614	38.9	61.1	6.8

Table 3: *Measurements of total number of methods invoked at least once by SPEC JVM98 applications. Also shown is the percentage of the total which are in the API, native and the program itself.*

Program	Methods invoked	Program %	API %	native %
eul	379	10.0	90.0	12.1
mol	362	9.7	90.3	11.3
mon	438	24.0	76.0	10.0
ray	356	18.3	81.7	10.7
sea	319	12.2	87.8	11.0
ave	370	14.8	85.2	11.0

Table 4: *Measurements of total number of methods invoked at least once by Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C). Also shown is the percentage of the total which are in the API, native and the program itself.*

independent cleanroom implementation of the JVM, distributed under the GNU Public License. Version 1.0.6 of Kaffe was used. Other approaches to tracing the execution of Java programs include bytecode-level instrumentation [13], and special-purpose JVMs such as SUN's Tracing JVM [25] and IBM's Jikes Research Virtual Machine, a development of the Jalapeño Virtual Machine [2].

While Kaffe can be built to emit debugging information, we modified its source to collect information more directly suited to our purposes. We added a hash table dictionary with full method names as keys, and dynamically counted each invocation and bytecode executed by for all methods.

We distinguished methods in the Kaffe library by noting the name beginning with *java/* or *kaffe/*. It should be noted that all measurements in this chapter were made with the Kaffe class library. This library may not be 100% compliant with SUN's JDK, but runs all the programs studied successfully. Platform independent dynamic analysis does not depend on the virtual machine used, but measures the methods and bytecodes executed by the both the program and the standard library. The API measurements reported here may therefore differ from other Java class libraries. In subsequent sections we will distinguish between code from the (Kaffe) class library and "program" i.e. those bytecodes from the SPEC JVM98 or Grande benchmark suites.

4 Analysis and Conclusions

Tables 1 and 2 show measurements of total number of classes loaded by SPEC JVM98 and Grande applications. Also shown is the percentage of the total which are in the API and the particular program itself. Tables 3 and 4 show measurements of total number of methods invoked at least once by SPEC JVM98 and Grande applications. Also shown is the percentage of the total which are in the API and the particular program itself. On average close to 4 methods are invoked at least once for each class loaded.

The first thing that stands out from Tables 2 and 4 is that while 89% of the classes loaded and 85% of the methods invoked at least once by the Grande applications are in the API, only 7.7% of the bytecodes, as has been shown in [7], are executed in the API. This

	invoke virtual	invoke special	invoke static	invoke interface	total
API					
Compress	46.5	24.2	29.2	0.1	2.8e+04
JESS	45.1	36.4	18.5	0.0	2.7e+07
Database	5.6	26.9	67.5	0.0	3.3e+07
javac	45.3	22.7	21.2	10.9	7.2e+07
mpegaudio	61.9	21.4	16.6	0.0	5.1e+04
mtrt	61.1	28.8	10.1	0.0	3.6e+06
jack	27.2	46.9	14.1	11.8	8.7e+07
ave	41.8	29.6	25.3	3.3	3.2e+07
non-API					
Compress	91.3	8.7	0.0	0.0	2.3e+08
JESS	85.0	9.2	5.1	0.7	1.1e+08
Database	83.1	0.2	0.1	16.5	9.0e+07
javac	78.5	15.2	2.1	4.2	8.0e+07
mpegaudio	72.1	26.6	1.2	0.2	1.1e+08
mtrt	94.5	5.4	0.1	0.0	2.8e+08
jack	62.5	10.9	11.8	14.8	2.9e+07
ave	81.0	10.9	2.9	5.2	1.3e+08
Total					
Compress	91.3	8.7	0.0	0.0	2.3e+08
JESS	76.9	14.7	7.8	0.5	1.3e+08
Database	62.2	7.4	18.3	12.1	1.2e+08
javac	62.8	18.7	11.1	7.4	1.5e+08
mpegaudio	72.1	26.6	1.2	0.2	1.1e+08
mtrt	94.1	5.7	0.2	0.0	2.9e+08
jack	36.0	37.9	13.6	12.6	1.2e+08
ave	70.8	17.1	7.5	4.7	1.6e+08

Table 5: *Measurements of total number of methods invoked by SPEC JVM98 applications. Also shown is the percentage of the total which are in the API and the program itself.*

	invoke virtual	invoke special	invoke static	invoke interface	total
API					
eul	32.8	54.6	12.6	0.0	4.3e+05
mol	63.5	21.6	14.8	0.1	1.4e+04
mon	78.1	0.6	21.4	0.0	4.9e+07
ray	62.6	21.6	15.7	0.1	1.3e+04
sea	63.3	21.4	15.2	0.1	1.2e+04
ave	60.1	24.0	15.9	0.1	9.9e+06
non-API					
eul	22.8	39.6	37.6	0.0	3.3e+07
mol	78.9	0.8	20.3	0.0	5.4e+05
mon	35.3	0.7	64.0	0.0	3.1e+07
ray	48.3	2.6	49.2	0.0	4.6e+08
sea	100.0	0.0	0.0	0.0	7.1e+07
ave	57.1	8.7	34.2	0.0	1.2e+08
Total					
eul	22.9	39.8	37.3	0.0	3.3e+07
mol	78.5	1.3	20.1	0.0	5.5e+05
mon	61.5	0.6	37.9	0.0	8.1e+07
ray	48.3	2.6	49.2	0.0	4.6e+08
sea	100.0	0.0	0.0	0.0	7.1e+07
ave	62.2	8.9	28.9	0.0	1.3e+08

Table 6: *Measurements of total number of methods invoked by Grande applications compiled using SUNs javac compiler, Standard Edition (JDK build 1.3.0-C). Also shown is the percentage of the total which are in the API and the program itself.*

shows dramatically different dynamic behaviour between Java API library code and Grande programs. It suggest that techniques such as dynamic [21] or JIT [1, 10] compilation may be much more effective when applied to Grande program code than to API code. There is a significant variation in the ratio of methods invoked at least once to bytecodes executed percentages (Method to Bytecode Dynamic percentage, MBDP) exhibited by different categories of code, and we believe this platform independent MBDP ratio with indicate quantitatively the possible benefits (or losses) that can be achieved during any platform specific compilation phases.

The same MBDP variation, to a lesser extent, can also be observed for SPEC JVM98 programs in Tables 1 and 3, where that average number of classes loaded from the API by the SPEC suite is 65%, methods invoked at least once is 61%, but bytecodes executed 34%.

Tables 5 and 6 show measurements of total number of methods invoked by SPEC JVM98 and Grande applications. Also shown is the percentage of the total which are in the API and the program itself. It's interesting to see that non-API static methods are insignificant for each of the SPEC programs, but are quite significant for each of the Grande programs, except sea. Since there's been a considerable amount of work done on optimizing virtual method calls, this suggests that any resulting speed improvements will not be as well reflected in the Grande suite.

It is possible to categorize invokespecial and invokestatic together, since both of these types of call can be bound as soon as the class is loaded. On the other hand, both invokevirtual and invokeinterface can only be bound when the call is made. It seems that the API method call figures are then much the same for both SPEC and Grande.

References

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-In-Time Java compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Canada, June 1998.

[2] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] D. Antonioli and M. Pilz. Analysis of the Java class file format. Technical Report 98.4, Dept. of Computer Science, University of Zurich, Switzerland, April 1988.

[4] S.J. Baylor, M. Devarakonda, S.J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[5] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, April 2000.

[6] T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, November 2000.

[7] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark Suite. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, pages 106–115, Stanford, CA, USA, June 2001.

[8] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *13th European Conference on Object Oriented Programming*, pages 92–115, Lisbon, Portugal, June 1999.

[9] Karel Driesen, Patrick Lam, Jerome Miecznikowski, Feng Qian, and Derek Rayside. On the predictability of invoke targets in Java byte code. In *2nd Workshop on Hardware*

Support for Objects and Microarchitecture for Java, Austin, Texas, 17 September 2000.

[10] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation and evaluation of optimisations in a Just-In-Time compiler. In *ACM 1999 Java Grande Conference*, pages 119–128, San Francisco, CA, USA, June 1999.

[11] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering*, 27(2):194–202, Feb 2001.

[12] I.H. Kazi, H.H. Chan, B. Stanley, and D.J. Lilja. Techniques for obtaining high perfromance in Java programs. *ACM Computing Surveys*, 32(3):213–240, September 2000.

[13] Han Bok Lee. BIT: A tool for instrumenting Java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, pages 73–82, Monterey, California, U.S.A., December 1997.

[14] Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *International Conference on Supercomputing*, pages 22–33, Santa Fe, NM, USA, May 2000.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[16] John Neffenger. The Volano report: Which Java platform is fastest, most scalable? *JavaWorld*, March 1999.

[17] Pendragon. *CaffeineMark 3.0*. Pendragon Software Corporation, http://www.pendragon-software.com/pendragon/cm3/, 13 May 1999.

[18] 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*, 50(2):131–146, February 2001.

[19] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Joint International Conference on Measurements and Modeling of Computer Systems*, pages 194–205, Cambridge, MA, USA, June 2001.

[20] SPEC. SPEC releases SPECjvm98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, 19 August 1998. http://www.specbench.org/osg/jvm98/-press.html.

[21] Sun Microsystems. The Java HotSpot virtual machine. Technical White Paper, http://java.sun.com/products/hotspot/, 7 November 2001.

[22] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *9th NRC/IBM Center for Advanced Studies Conference*, pages 125–135, Toronto, Canada, November 1999.

[23] T.J. Wilkinson. *KAFFE, A Virtual Machine to run Java Code*. http://www.kaffe.org, July 2000.

[24] Mario Wolczko. *Benchmarking Java with Richards and DeltaBlue*. Sun Microsystems Laboratories, http://www.sun.com/research/-people/mario/java_benchmarking/, 2001.

[25] Mario Wolczko. *The Tracing JVM*. Sun Microsystems Laboratories, http://www.experimentalstuff.com/-Technologies/TracingJVM, 19 April 2001.

[26] Xiaolan Zhang and Margo I. Seltzer. HBench: Java: an application-specific benchmarking framework for Java virtual machines. In *ACM Java Grande Conference*, pages 62–70, San Francisco, CA, USA, June 2000.