A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites

David Gregg[1], James Power[2,*] and John Waldron[1]

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

SUMMARY

In this paper we seek to provide a foundation for the study of the level of use of object-oriented techniques in Java programs in general, and scientific applications in particular. In particular we investigate the profiles of Java programs from a number of perspectives, including the use of class library methods, the size of methods called, the mode of invoke instruction used and the polymorphicity of call sites. We also present a categorisation of the nature of small-sized methods used in Java programs. We compare the Java Grande and SPEC JVM98 benchmark suites, and note a significant difference in the nature and composition of these suites, with the programs from the Java Grande suite demonstrating a less object-oriented approach.

KEY WORDS: Benchmark suites, Java Virtual Machine, dynamic profiling

1. Introduction

The Java programming language [15] has become established as a general-purpose programming language, with applications in most aspects of computer science and software engineering. Despite its relatively poor speed performance compared to C and C++, it is also becoming increasingly popular in the domain of scientific computing, since it facilitates the construction of portable, robust and reliable applications.

In attempting to improve the performance of Java programs, there has been much concentration on compiler design techniques to increase the speed and efficiency of the Java Virtual Machine (JVM) [19]. As well as the standard range of compiler optimisations,

*Correspondence to: james.power@may.ie.
This version prepared on September 1, 2003 at 15:19.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

JVM optimisation techniques also deal with object-oriented issues such as class hierarchy analysis, stack-based object allocation, wrapping basic types as objects, and speeding up or eliminating dynamically-bound virtual method calls. Since many JVMs implement optimisations dynamically, as the program executes, the choice of the exact nature and level of optimisation is crucial to improving program performance. Many JVMs apply an adaptive strategy, profiling the application as it runs, and choosing suitable optimisations.

Since choosing and applying the correct optimisations is so important for the performance of Java applications, it is reasonable to ask whether the same techniques that apply to standard object-oriented applications also apply to scientific applications. It is arguable that many scientific applications may not be written in an object-oriented style, or may not fully exploit all the features of Java, and thus may not be amenable to the same kind of optimisation.

In this paper we seek to provide a foundation for the study of the level of use of object-oriented techniques in Java programs in general, and scientific applications in particular. In particular we investigate the profiles of Java programs from a number of perspectives, including the use of class library methods, the size of methods called, the mode of invoke instruction used and the polymorphicity of call sites. We also present a novel categorisation of small methods used in Java programs, and describe their composition and level of use.

Unlike many existing techniques which are based on static analysis and static metrics, we study the behaviour of Java programs dynamically, as they execute on the virtual machine. This allows us to build a picture of the behaviour, rather than the architecture of the programs and directly parallels modern run-time optimisation techniques.

In order to measure examples of Java programs we compare the results from the analysis of two benchmark suites. We use the Java Grande benchmark suite to provide an example of larger, scientific applications, and the SPEC JVM98 benchmark suite to represent more standard Java applications. It is not entirely obvious what constitutes a "standard" scientific Java application, and we hope that the results presented here can be used to calibrate the Java Grande and SPEC JVM98 suites, and measure the degree to which they are in fact representative of such applications.

In Section 2 we introduce the two benchmark suites, and briefly give an overview of some of the existing work using these suites. In Section 3 we compare the Java Grande and SPEC JVM98 suites by analysing the method-call frequency, in particular the number of class library methods called, and the nature of the method calls. In section 4 we present a static and dynamic analysis of the sizes of methods called in both suites, highlighting the differences between Java Grande and SPEC JVM98 programs. Section 5 concludes the paper.

2. Background and Related Work

In this section we give an overview of the two benchmarks suites used in this study, and we review some of the background to method-level analyses of Java programs.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Table I. The benchmark suites compared in this analysis. *Five of the programs were taken from section 3 of the Java Grande Benchmark suite, and seven from the SPEC JVM98 benchmark suite.*

The Java Grande Forum Benchmark Suite Section 3: 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 (SPEC) JVM98 Benchmarks

cmprs	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

2.1. The Benchmark Suites

For the purposes of our study, we have used programs from two Java benchmark suites: the SPEC JVM98 benchmark suite [26] and the Java Grande Forum Benchmark Suite [8]. The first set of programs is taken from section 3 of the Java Grande suite version 2.0, and contains five large-scale applications, designed as examples of real-world applications. The second set, consisting of seven programs, is from the SPEC JVM98 suite. The SPEC JVM98 suite was designed as an industry-standard benchmark suite for measuring the performance of client-side Java applications. The benchmarks included in this work are shown in Table I; in the remainder of this paper we shall refer to these as the *Grande* and *SPEC* suites respectively.

Studies of the Grande and SPEC suites have typically concentrated on performance issues for various JVMs. Studies of the Grande suite include performance-related measures [8, 7] as well as dynamic byte-code level views [11]. The SPEC JVM98 suite is perhaps more commonly used to measure the speed and effectiveness of Java compilers and virtual machines [5, 6, 23, 27, 31]. Other views include those discussing lower-level architectural issues relating to the SPEC programs [20, 24], allocation and heap behaviour [12], as well as measuring the performance of various Java microarchitectures [22] .

2.2. Technical Details

The Java Grande Forum Benchmark suite is distributed in source code format and was compiled using the Java compiler from SUN's JDK, version 1.3. The SPEC suite is distributed in bytecode format, and the maximum size, *s100* was used in the results presented here. None of the programs were optimised in any way. The Kaffe Virtual Machine [32], version 1.0.6, was

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

used to collect data on the dynamic operation of all the programs. Each application was run without modification on the instrumented Kaffe VM. It is important to note for this study that the Kaffe VM used here was run in *interpretive* mode. That is, in order to measure the scope for optimisation, no method inlining or method compilation was performed during the running of the programs.

It should also be noted that all measurements in this chapter were made with the Kaffe class library. This library is not 100% compliant with SUN's JDK, and may, of course, differ from other Java class libraries. In subsequent sections we will distinguish between *Library* code from the Kaffe class library, and *Application* code, consisting of those bytecodes from the benchmark suites themselves.

2.3. Related Work

Ishizaki et al. notes that typical object-oriented programs have smaller methods, and method calls occur frequently [18]. Their JIT inlines static calls to small methods, and uses class hierarchy analysis [17] to devirtualise dynamic calls. The test suite they use includes the SPEC benchmarks as well as some more GUI-based programs, and Ishizaki et al. report good scope for devirtualisation across the suite, in particular for mtrt which makes extensive use of small methods.

Both the Hotspot [28] and Jalapeño [1] virtual machines include method inlining as part of their repertoire of adaptive optimisations. Indeed, the Hotspot White Paper [28] notes that method inlining is important not just because it eliminates the overhead of a method call, but also because it produces larger blocks of code to which further optimisations can be applied. Arnold et al. describe an approach to optimisation in Jalapeño based on statistical sampling, and notes a good performance improvement through most of the programs in the SPEC suite [4].

Sundaresan et al. compare a number of call-graph analysis techniques for the purposes of method call resolution, showing a good performance improvement for the SPEC programs [29]. These optimisations, among others, have been implemented in the Soot tool [30]. Arnold et al. present a cost-benefit analysis of three inlining techniques, using programs from the SPEC suite to compare their performance [3].

The trade-offs involved in deciding whether a method should be inlined are discussed by Scheifler [25] as well as Chang et al. [9] in the context of procedural languages, and applied to Java by Arnold et al. [3]. While a number of techniques can be employed to estimate the cost-benefit ratio for a given inlining, in this paper we take the simplest approach to both. Following both Scheifler and Arnold et al. we estimate the cost in terms of the method size, and we take the node-based approach of Arnold et al. in estimating benefit by counting the number of calling sites.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Table II. Measurements of the number of methods invoked by Java Grande and SPEC JVM98 applications. *These figures refer to the number of dynamic method calls whose source was in the application part of the benchmark suites.*

Java Grande Methods Calls from the Application

	invoke virtual	invoke special	invoke static	invoke interface	total invokes
eul	22.8	39.6	37.6	0.0	32.9e+6
mol	78.9	0.8	20.3	0.0	0.5e+6
mon	35.3	0.7	64.0	0.0	31.3e+6
ray	48.3	2.6	49.2	0.0	457.7e+6
sea	100.0	0.0	0.0	0.0	71.2e+6
ave	*57.1*	*8.7*	*34.2*	*0.0*	*119e+06*

SPEC JVM98 Methods Calls from the Application

	invoke virtual	invoke special	invoke static	invoke interface	total invokes
cmprs	91.3	8.7	0.0	0.0	225.9e+6
db	83.1	0.2	0.1	16.5	90.2e+6
jack	62.5	10.9	11.8	14.8	28.9e+6
javac	78.5	15.2	2.1	4.2	80.1e+6
jess	85.0	9.2	5.1	0.7	107.4e+6
mpeg	72.1	26.6	1.2	0.2	109.7e+6
mtrt	94.5	5.4	0.1	0.0	284.7e+6
ave	*81.0*	*10.9*	*2.9*	*5.2*	*132e+06*

3. Method Calls

In this section we describe the distribution of method calls in the programs studied. The results presented here are more coarse-grained than those presented in the next section, but even here important differences emerge between the two benchmark suites.

3.1. Method call sites

Table II gives a summary of the nature of method call sites in the programs from the Grande and SPEC suites. Here we have counted each method call that took place in application code when the program was run. In Table II we have partitioned the method calls that take place in the application code between virtual, special, static and interface methods. Calls *to* application and library methods are included here, but only calls *from* application methods are counted, since the nature of the application cannot directly determine method calls internal to the class library.

For example, from the first row of Table II we can see that of the 32.9×10^6 method calls in application code when the `eul` program was run, 22.8% were the result of an `invokevirtual` instruction, 39.6% were the result of an `invokespecial` instruction, 37.6% resulted from an `invokestatic` instruction, and 0.0% resulted from an `invokeinterface` instruction. The

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Table III. Measurements of total number of dynamic methods called in the Grande and SPEC suites. *Also shown is the percentage of the methods called whose target was in the Class Library, and percentage of total calls targeted at native methods in the Class Library.*

Java Grande Methods Called Dynamically

Program	Total methods	Library %	Library & native %
eul	33.4e+06	58.0	12.6
mol	0.5e+06	22.7	19.8
mon	80.7e+06	98.8	37.3
ray	457.7e+06	3.1	1.6
sea	71.2e+06	0.0	0.0
ave	*129e+06*	*36.5*	*14.3*

SPEC JVM98 Methods Called Dynamically

Program	Total methods	Library %	Library & native %
cmprs	226.0e+06	0.0	0.0
db	123.6e+06	98.7	0.1
jack	115.8e+06	93.6	1.1
javac	152.5e+06	63.0	1.4
jess	134.9e+06	32.5	0.0
mpeg	109.7e+06	1.3	0.0
mtrt	288.4e+06	3.2	0.1
ave	*164e+06*	*41.8*	*0.4*

`invokespecial` instruction is typically used in a constructor to call a constructor from a super class.

From the percentages in Table II we can see a sharp difference between the benchmark suites. The calls to virtual, interface and "special" methods could be considered the hallmark of an object-oriented program, and form the predominant part of the method calls in SPEC. The Grande programs, in contrast have a much higher average of calls to static methods, apart from the `sea` application.

Since many JVM optimisations are directed toward devirtualisation and eliminating the overhead of dynamically-bound method calls, it is reasonable to suggest that such optimisations will have a reduced impact on the programs from the Grande suite when compared to the SPEC suite.

3.2. Method call targets

In Table III we take the symmetric view to Table II, and look instead at the *target* of method calls. While a certain degree of usage of the Java class library is unavoidable in Java programs, intensive use of the library suggests a better fit between the application and the methods provided by that library. Thus Table III summarises the distribution of methods called between application code, library code and native methods. The totals and percentages in Table III

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

are dynamic counts, where each method called has been counted during the running of the program. Since there are no native methods in the applications themselves, all native methods called come from the class library. Method calls emanating from native methods have not been included in these totals.

For example, from the first row of Table III we can see that 33.4×10^6 methods were called when the eul program was run. Of these, 58.0% were directed at methods in the class library, with 12.6% of the total methods called being directed at *native* methods in the class library.

As can be seen from Table III there is considerable variance across the programs with regard to the number of library methods called. While the programs from the SPEC suite have a slightly higher average number of library methods called, this is clearly not uniform across the applications. However, it is notable that the Grande programs with a high number of library methods called also have a high proportion of *native* methods called, particularly in the case of mon, where almost all library methods called are of this type. In this context, it is fair to suggest that the number of Java methods called in the class library is greater on average for the SPEC suite.

Radhakrishnan et al. note a consistent distribution in the dynamic sizes of methods called in the SPEC suite, and attributes this to the influence of methods from the class library [24] (both Sun's JDK and Kaffe seem to have been used in the study). Table III appears to contradict this view, since at least three SPEC applications direct a very small number of method calls to the class library, whereas only two make significant use of the library. This cannot be the determining factor across all SPEC applications.

3.3. Polymorphicity of virtual method calls

We have presented results measuring the proportion of methods calls directed at virtual methods in Table II, since these are considered typical of the object-oriented programming style. However, it is possible to program in a "pseudo" object oriented style, where methods are defined as virtual, but are not actually overridden in the program code. Such method call sites are prime targets for optimisation, since the target of most, if not all their calls can be calculated statically [29, 17].

In the context of our discussion then, it is important to address the issue of the degree to which the virtual method calls in a program are targeted at different methods at run-time. To do this, we have tracked the target of the `invokevirtual` instruction for the Grande and SPEC suites and noted, for each call site, the number of different methods called at run-time. We would expect that in programs written in an object-oriented style, `invokevirtual` instructions will often have many targets. Conversely, if the program is written in a procedural style with little use of inheritance, we would expect that these instructions would usually have only a single target.

Table IV shows the percentage of executed `invokevirtual` instructions with varying number of targets. For each program in each suite we show the percentage of `invokevirtual` instructions directed at 1,2,...,10 and >10 different methods. For example, from the first data line of the table we see that 98.77% of the `invokevirtual` instructions in the eul program were instructions that were directed at a single method. This percentage is a proportion of the total number of dynamically-executed `invokevirtual` instructions for eul. It should thus

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Table IV. Different method call targets for Grande and SPEC programs - totals. *For each program, these figures show the percentage of dynamically-executed* `invokevirtual` *instructions directed to 1,2,...10 different virtual methods.*

Java Grande virtual method calls directed at multiple targets (all virtual method calls)

Program	1	2	3	4	5	6	7	8	9	10	> 10
eul	98.77	0.04	0.05	0.00	0.54	0.59	0.00	0.00	0.00	0.00	0.01
mol	99.67	0.04	0.11	0.05	0.00	0.00	0.02	0.00	0.01	0.00	0.10
mon	99.28	0.18	0.00	0.24	0.09	0.21	0.00	0.00	0.00	0.00	0.00
ray	50.23	0.00	0.00	0.00	0.00	0.00	2.60	0.00	0.00	0.00	47.18
sea	46.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	53.50

SPEC JVM98 virtual method calls directed at multiple targets (all virtual method calls)

Program	1	2	3	4	5	6	7	8	9	10	> 10
cmprs	99.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
db	40.84	58.58	0.00	0.05	0.00	0.05	0.03	0.00	0.36	0.00	0.09
jack	8.80	0.10	0.07	0.10	0.10	0.10	0.10	0.02	0.34	0.01	90.36
javac	30.17	0.04	0.10	0.08	0.52	0.01	0.19	0.25	0.00	4.54	64.09
jess	55.71	2.08	0.06	0.00	0.00	0.06	0.01	0.20	0.00	0.00	41.88
mpeg	65.69	4.57	0.02	12.84	0.00	0.00	0.00	0.00	0.00	0.00	16.88
mtrt	0.31	0.57	4.22	3.07	1.25	0.37	42.86	0.00	0.36	0.00	47.00

Table V. Different method call targets for Grande and SPEC programs - application only. *For each program, these figures show the percentage of dynamically-executed* `invokevirtual` *instructions directed to 1,2,...10 different virtual methods in the application part of the programs.*

Java Grande virtual method calls directed at multiple targets (application only)

Program	1	2	3	4	5	6	7	8	9	10	> 10
eul	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mol	99.99	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mon	97.64	0.73	0.00	1.09	0.00	0.54	0.00	0.00	0.00	0.00	0.00
ray	50.23	0.00	0.00	0.00	0.00	0.00	2.60	0.00	0.00	0.00	47.18
sea	46.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	53.50

SPEC JVM98 virtual method calls directed at multiple targets (application only)

Program	1	2	3	4	5	6	7	8	9	10	> 10
cmprs	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
db	39.89	60.04	0.00	0.00	0.00	0.05	0.03	0.00	0.00	0.00	0.00
jack	15.49	0.19	0.16	0.23	0.23	0.23	0.04	0.04	0.77	0.02	82.61
javac	40.07	0.05	0.15	0.13	0.76	0.02	0.01	0.39	0.00	0.01	58.42
jess	63.24	2.36	0.07	0.00	0.00	0.07	0.00	0.03	0.00	0.01	34.23
mpeg	65.71	4.57	0.00	12.84	0.00	0.00	0.00	0.00	0.00	0.00	16.88
mtrt	0.04	0.57	4.26	3.10	1.26	0.38	43.21	0.00	0.36	0.00	46.84

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Table VI. Different method call targets for Grande and SPEC programs - library only. *For each program, these figures show the percentage of dynamically-executed* `invokevirtual` *instructions directed to 1,2,...10 different virtual methods in the Class Library.*

Java Grande virtual method calls directed at multiple targets (class library only)

Program	1	2	3	4	5	6	7	8	9	10	> 10
eul	32.82	2.18	2.72	0.15	29.60	32.13	0.05	0.01	0.04	0.00	0.32
mol	80.84	2.24	6.39	2.69	0.10	0.00	1.01	0.14	0.54	0.00	6.05
mon	99.75	0.03	0.00	0.00	0.12	0.11	0.00	0.00	0.00	0.00	0.00
ray	75.51	4.07	7.83	0.33	0.23	3.60	0.00	0.16	0.00	0.22	8.06
sea	80.22	4.72	5.45	1.40	0.00	1.21	0.00	0.17	0.50	0.24	6.10

SPEC JVM98 virtual method calls directed at multiple targets (class library only)

Program	1	2	3	4	5	6	7	8	9	10	> 10
cmprs	49.72	2.19	12.13	0.18	0.74	0.07	10.56	0.09	0.00	0.12	24.20
db	78.88	0.01	0.03	2.18	0.03	0.01	0.03	0.00	14.98	0.04	3.82
jack	3.67	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	96.29
javac	11.13	0.02	0.00	0.00	0.07	0.00	0.53	0.00	0.00	13.24	75.01
jess	0.16	0.00	0.01	0.01	0.00	0.00	0.06	1.40	0.00	0.00	98.36
mpeg	16.04	0.88	52.86	0.07	0.33	0.03	2.31	0.03	0.00	0.05	27.41
mtrt	33.39	0.01	0.04	0.01	0.01	0.12	0.03	0.00	0.26	0.00	63.44

be borne in mind that, as shown earlier in Table II, this is of variable importance in each program, particularly for those in the Grande suite.

Interestingly, the number of `invokevirtual`s with only a single target is very large. However, there is a large difference in behaviour between SPEC and Grande applications. On average 45% of SPEC `invokevirtual`s have only a single target, whereas the corresponding figure for Grande is 78%. Clearly, the cmprs benchmark from the SPEC suite does not exhibit this feature of object-oriented programs, but all others have large numbers of highly polymorphic `invokevirtual`s. In contrast, three of the five Grande programs tend toward a single target, and the other two use a large number of single target `invokevirtual`s. Clearly, on this measure, SPEC programs are significantly more object-oriented than Grande ones.

Table V shows the percentage of executed `invokevirtual` instructions with varying number of targets for just the application part of the benchmark programs, with call targets in the Class Library excluded. For the Grande benchmarks the figures are almost identical to the total figures, because only a tiny percentage of executed `invokevirtual` instructions are in the Class Library. In the case of the SPEC programs, the difference is larger, especially for jack and jess. The corresponding figures for the `invokevirtual` instructions in the Class Library only are shown in Table VI. The Class Library uses much more inheritance than the programs in the benchmark suite, many of which contain less than five classes, so the `invokevirtual` instructions tend to be much more polymorphic.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

4. Size of Methods Called

In this section we consider the size of the methods called in each of the suites. This is significant for two reasons. First, small methods are a prime target for code inlining, an important optimisation technique in many JVMs. Second, small methods are regarded as standard in object-oriented programs [18], and a desirable result of code refactoring [14].

Henderson-Sellers [16, §6.3.1] suggests that method size can be indicative of the "object-orientedness" of code. He notes that object-oriented code tends towards smaller method sizes, and ascribes larger method sizes to a "traditional" (as opposed to object-oriented) mind-set. Lorenz and Kidd [21] point out a tendency toward shorter methods on average in object-oriented programs, and suggest that longer methods indicate a higher likelihood that "function-oriented" code is being written.

In this section we present average figures for each of the Grande and SPEC suites, rather than results for each program. It should be noted that the data in this section are not average *counts*, but rather average *frequencies*, expressed as a percentage for each program. Thus, differences in program size will not cause one program to unduly effect the overall results for a benchmark suite.

4.1. Method Size Distribution

The size of a method is a common metric used in the estimation of the cost of method inlining, since repeated inlining of a large method can lead to "code bloat", and produces the familiar size versus speed trade-off. In particular, a significant increase in code size due to inlining may prove prohibitive in the case of Java applications running in constrained environments.

There are a number of ways of measuring the size of a method, including simple lines-of-code, cyclomatic complexity and volume metrics - see Fenton and Pfleeger [13] for a survey. In this study we use the simplest metric, namely the number of *bytecode instructions* the method contains. Antonioli and Pilz describe an extensive study of the static composition of Java source file where it is noted that the average size of a bytecode instruction, i.e. the operator plus the operands, is just under 2 bytes [2].

Figure 1 summarises the distribution of method sizes in the Grande and SPEC suites, based on counting the number of bytecode instructions in the corresponding class files. The approach of statically measuring the program source is typical for most standard program metrics used in software engineering, and static method sizes and call counts form the basis of metrics such as *weighted method per class* and the *response for a class* [10].

As can be seen from Figure 1 the method sizes for the Library methods used by both programs has a similar distribution, with a large number of small methods (less than 10 bytecode instructions), and a decreasing frequency as the method size gets larger. The SPEC application code follows a similar distribution, with a peak occurring for some large methods used in the programs themselves. However, the Grande suite is somewhat different, with a reduction in the overall presence of very small methods immediately noticeable. In fact 25.9% of the Grande application methods, and 41.5% of the SPEC application methods are of size less than 10 instructions.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

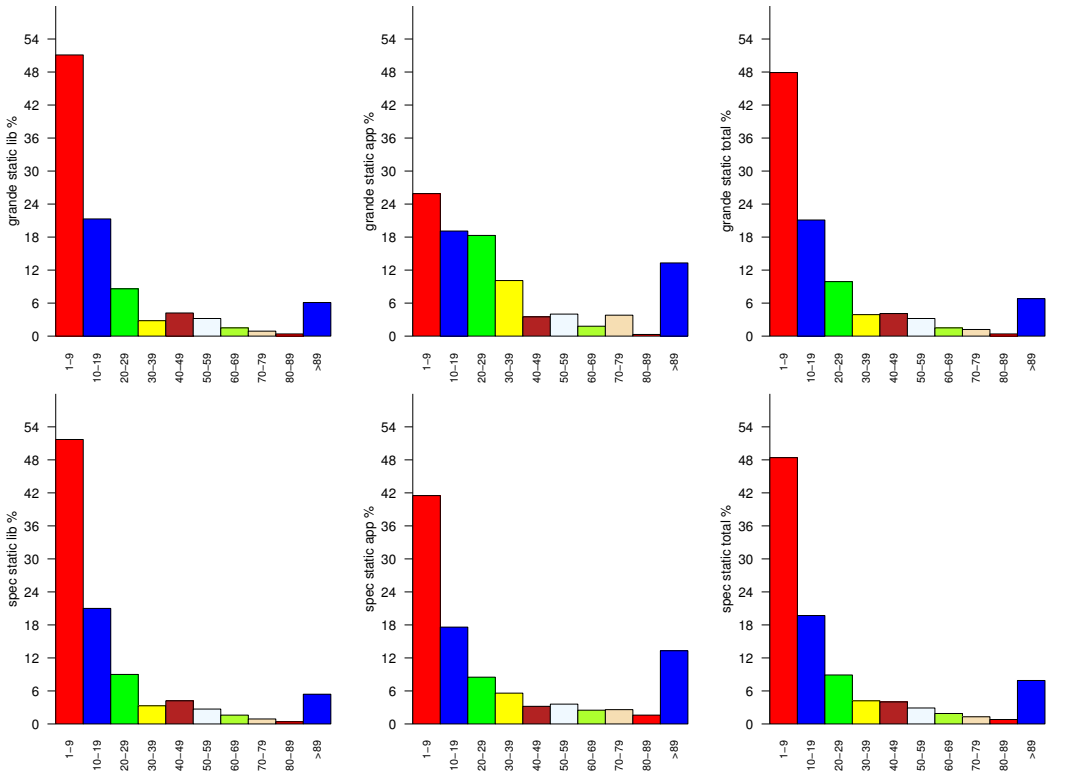

Figure 1. Distribution of method calls, calculated statically. *These charts compare the distribution of method calls in the bytecode source for the Grande and SPEC benchmark suites. The horizontal axis indicates method size in terms of number of bytecode instructions.*

Figure 2 measures the frequency of method calls that occur dynamically, during the program execution. We can see that for methods in the class library used by both benchmarks, the frequency of a method call varies inversely with its size, as predicted by its static occurrence frequency. However, the distribution of dynamic method calls in the applications differs markedly between the SPEC and Grande programs, with the SPEC programs maintaining the distribution of the class library, and the Grande programs giving noticeably less weight to smaller methods. Indeed, small methods, of less than 10 bytecode instructions, count for 9.8% of the Grande application methods called, but 38.6% of the SPEC application methods called.

The use of small methods in the class library as well as in the SPEC suite suggests that these programs have been written in the traditional object-oriented style. The absence of this distribution in the Grande suite suggests that these programs have been written using a more "procedural" style, with less emphasis on reducing method size.

While the elimination of a method call overhead is the most obvious benefit of inlining, another advantage is that larger blocks of code are created, and these may then be subjected to

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

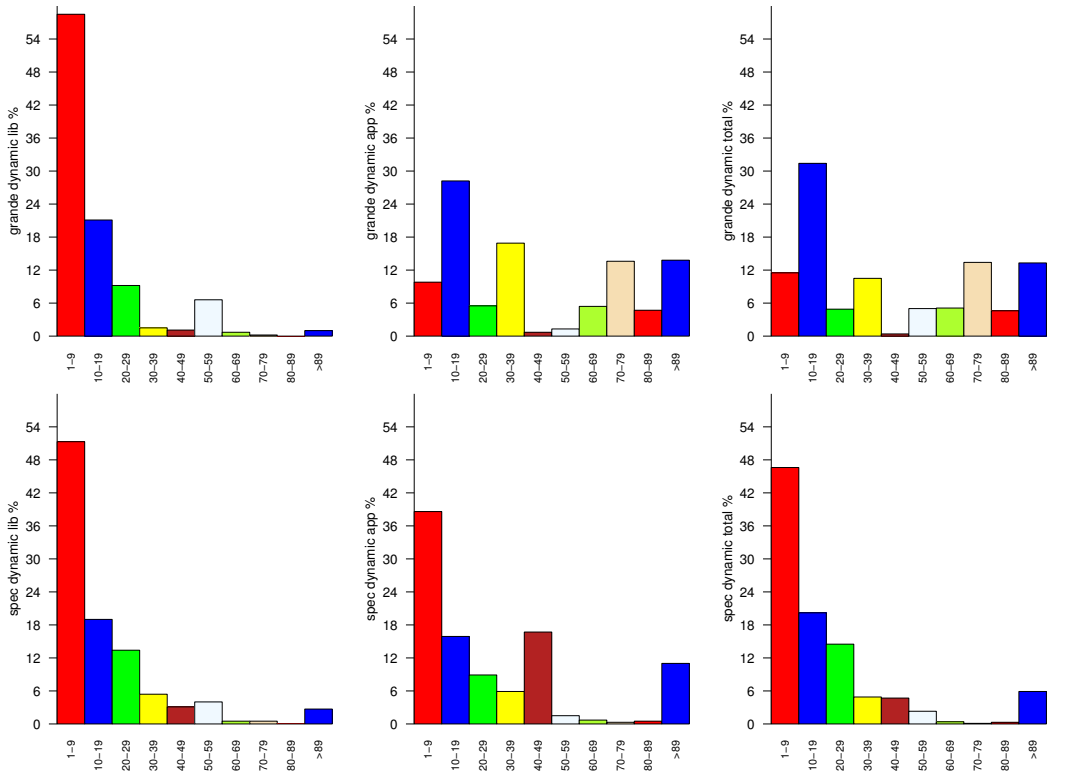

Figure 2. Distribution of method calls, calculated dynamically. *These charts compare the distribution of method calls during the execution of programs from the Grande and SPEC benchmark suites. The horizontal axis indicates method size in terms of number of bytecode instructions.*

more wide-ranging optimisation. Figure 3 presents another view of the distribution of method sizes, where the frequencies are based on the number of bytecode instructions that the method contains.

For example, from Figure 3 we can see that methods of size less than 10 instructions accounted for 40.7% of all library instructions executed in the Grande suite, and 17.6% of all library instructions executed in the SPEC suite. Such a measure is inherently biased against small methods, but helps to give an impression of the proportion of bytecodes that would be affected by inlining these methods.

As can be seen in Figure 3, the instructions in small methods still form a significant proportion of the instructions executed in the class library. However, in the application code this is reversed, with small methods making a negligible impact on the total number of bytecodes executed in the Grande applications, but accounting for 13.5% of all application instructions executed in the SPEC suite.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

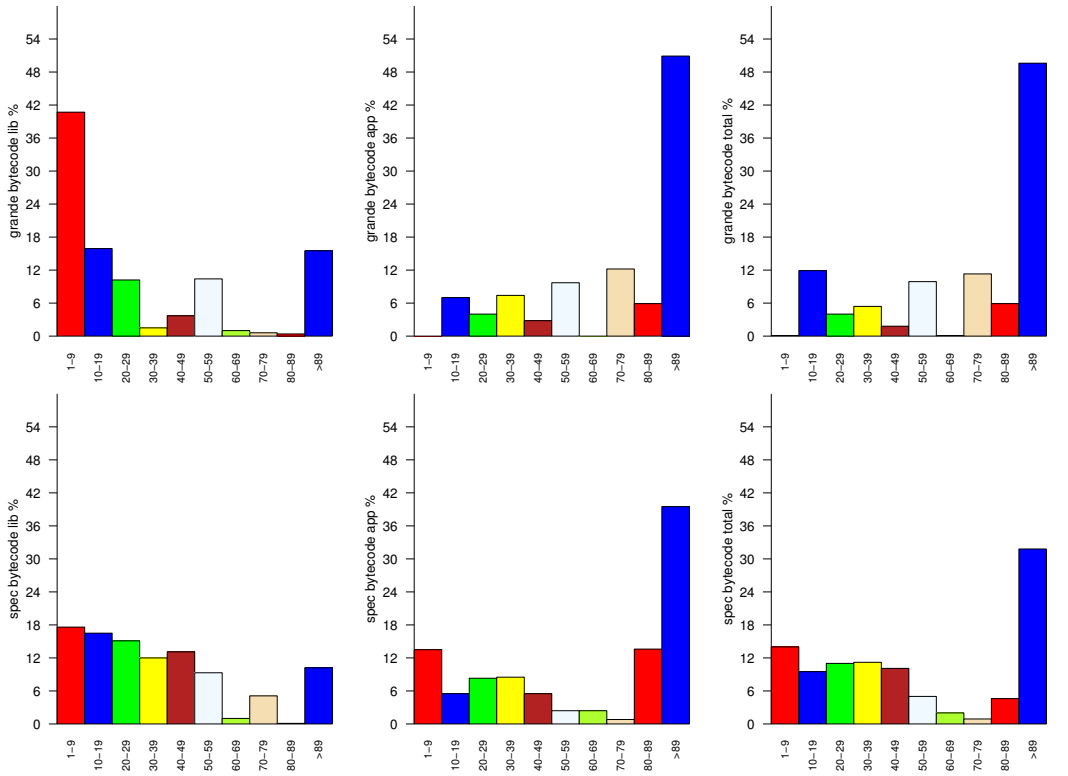

Figure 3. Distribution of method sizes, calculated by bytecode size. *These charts compare the distribution of total bytecodes executed between the methods executed from the Grande and SPEC benchmark suites. The horizontal axis indicates method size in terms of number of bytecode instructions.*

Radhakrishnan et al. note a consistent tri-nodal distribution in the dynamic sizes of methods called in the SPEC suite, with most of the methods being either 1, 9 or 26 bytecodes long [24]. None of the figures presented above reflect such a distribution, and it is unclear how the distribution reported by Radhakrishnan et al. could have been calculated.

4.2. Distribution of Small Methods

Since smaller methods are seen as reflecting an object-oriented programming style, this subsection studies their composition in some detail. It should be remembered throughout this subsection that, as shown in Figure 2, smaller methods represent a significantly lesser proportion of the Grande suite compared to the SPEC suite.

Figure 4 shows the distribution between methods containing less than 10 instructions in the Grande and SPEC suites. Here each method has been weighted by the number of times it was called dynamically, in order to get a better view of its effect on the suite. For the purpose of

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

Categories of small methods called

A	simple	load/store and constant access only
B	field	As for A, but also has dynamic field access
C	s/field	As for A, but also with static field access
D	invoke	As for A, but also invokes dynamic methods
E	s/invoke	As for A, but also invokes static methods
F	field & invoke	As for A, with dynamic field access and dynamic method calls
G	field & s/invoke	As for A, with field access and static method calls
H	invoke & s/field	As for A, with static field access and dynamic method calls
I	invoke & s/invoke	As for A, with static and dynamic method calls
J	s/field & s/invoke	As for A, with static field access and method calls
K	control	Contains a control instruction, such as `if` or `goto`
L	others	All methods not in the above categories

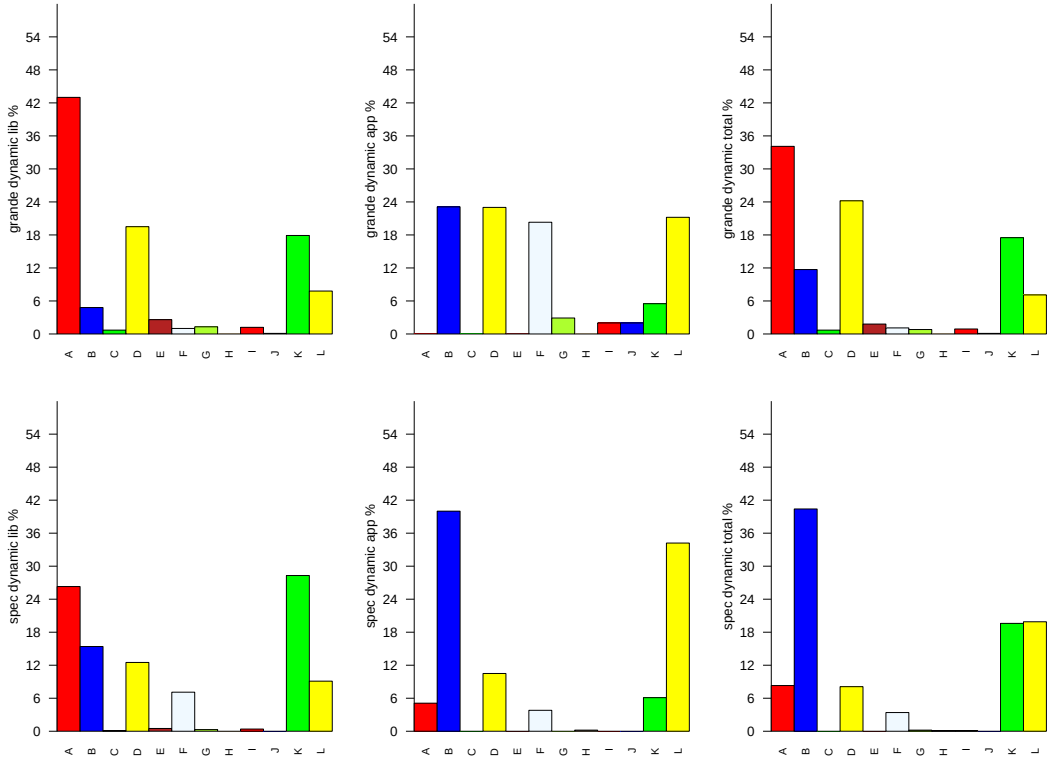

Figure 4. Profile of the small methods in the Grande and SPEC suites. *The table partitions methods containing less than 10 instructions into various categories based on the instructions they contain. The bar charts shows he proportion of small methods that belong to each category, based on methods called dynamically, for the Grande and SPEC suites.*

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

this study, we have partitioned this set of small methods into 12 separate categories, based on their general functionality. To do this, we have ignored stack-manipulation instructions, and concentrated on method calls, field accesses and control instructions.

The simple methods in Category A thus contain only variable load and store instructions, as well as constant accesses. A very small proportion of these methods contain just a single `return` instruction. Radhakrishnan et al. identifies methods containing only 1 instruction as "wrapper methods", consisting of control transfer instructions [24]. However, the single instruction in such methods must be a `return` instruction. As can be seen from Figure 4, these form a negligible part of the application methods called in either the Grande or SPEC suite (0% and 5% respectively). They do, however, account for a significant proportion of class library methods called in both Grande, at 43%, and SPEC, at 26%, where they are chiefly responsible for returning constant values.

Methods that simply access a static field, either changing or returning its value are prime candidates for inlining and subsequent constant propagation. As can be seen from Figure 4, these methods, constituting category C, do not have a significant role in either suite. Similarly, categories E and J which contain static method calls account for a low proportion of the total in both suites.

Methods in category B are the classic *get* and *set* methods, that just access or change an instance variable. These play a relatively small role in the class library methods called in both suites, but account for 40% of application methods in the SPEC suite, almost twice as high a proportion as for the Grande suite at 23%. While the higher usage of such methods in the SPEC suite is consistent with the object-oriented nature of the programs, as noted in previous sections, it is interesting to see that this is not in turn reflected in the class libraries.

The methods in category D consist of the actual "wrapper" methods that mainly dispatch a call to another (non-static) method. These account for a fairly consistent proportion through the Grande and SPEC class library methods (20% and 13% respectively) and in the corresponding Grande and SPEC application methods (23% and 10% respectively). The correspondence between library and application programs, as well as the lower usage in SPEC application programs suggest that such methods may not be typical of object-oriented programs.

The methods in category K of Figure 4 are the only ones to contain control instructions, such as `if` or `goto` branches, or `throw` instructions. As such, they are the only methods that can correspond to conditional or iterative statements in the corresponding Java code. This category of method plays a very small part in the small-method application code for either the Grande or SPEC suites, 6% in each case, supporting our thesis that these small methods deal chiefly with encapsulation and interface issues, rather than with actual computation. However category K methods play a significant role in the library methods called for the Grande and SPEC suites, 20% and 28% respectively, suggesting a heavier reliance on class library routines that perform computations.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

5. Conclusions

In this paper we have analysed programs from a number of different Java applications by measuring a number of method-based characteristics. These included the nature of the method calls, the use of the class library methods, the polymorphicity of method call sites, and the sizes of the methods called. These figures are based on the dynamic analysis of running Java programs, and effectively simulate some of the work done by a run-time compiler. We believe that the data presented here can provide a solid, quantitative foundation for the study of optimisation techniques in such compilers.

We have presented a comparative study of applications from two standard benchmark suites for Java programs: the SPEC JVM98 and Java Grande benchmark suites. We have exposed significant differences between these suites using our dynamic, method-based measurements, and suggest that run-time optimisation strategies would benefit from a study of these differences. We note that the programs from the Grande suite demonstrate a measurably less object-oriented approach than SPEC applications, including making less use of the class library and making greater use of longer methods.

As we have noted, our work differs in a number of important points from that presented by Radhakrishnan et al. [24]. First, they note a tri-nodal distribution in method sizes for the SPEC suite, where most methods contain 1, 9 or 26 bytecodes. While a number of different approaches can be taken to measuring method size, none of the three approaches analysed in this paper appear to yield this distribution. Second, they suggest that the similarity in distribution is due primarily to the influence of methods from the class library. We observe no such similarity between programs in the SPEC suite, and we note in Table III that they vary widely in their use of library methods. Third, Radhakrishnan et al. state that methods of size 1 contain a "control transfer instruction" which we note must more specifically be a *return* instruction

We have presented a novel classification of small methods in Java programs, with a view to understanding their composition and usage. We note that few small methods contain branching or control instructions, and we describe the distribution of these methods between *get* and *set* methods, as well as "wrapper" methods. As before, a quantifiable difference between SPEC and Grande applications emerges as a consistent feature of our study.

The work presented in this paper raises two important question in relation to the performance of Grande applications. First, do existing compiler optimisations, targeted specifically at object-oriented programs, deliver satisfactorily for scientific applications that may not make such heavy use of these techniques? Second, are the programs in the Grande suite representative of scientific applications, or will such applications come to resemble object-oriented programs as more of the code base is moved to Java? We believe that the analysis presented in this paper provides a framework in which an answer to these questions can be developed, particularly as more scientific applications in Java become available for study.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

REFERENCES

1. 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.
2. 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.
3. M. Arnold, S. Fink, V. Sarkar, and P.F. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 52 – 64, Boston, MA, USA, January 18 2000.
4. M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Object Oriented Programming Systems Languages and Applications*, pages 47 – 65, Minneapolis, Minnesota, USA, October 15-19 2000.
5. K.R. Bowers and D. Kaeli. Characterising the SPEC JVM98 benchmarks on the Java virtual machine. Technical report, Northeastern University Computer Architecture Research Group, Dept. of Electrical and Computer Engineering, Boston Massachusetts 02115, USA, 1998.
6. T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Object-Oriented Programming Systems, Languages and Applications*, pages 353 – 366, Tampa, Florida, USA, October 14-18 2001.
7. 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.
8. 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.
9. P. Chang, S.A. Mahlke, W.Y. Chen, and W.W. Hwu. Profile-guided automatic inline expansion for C programs. *Software - Practice and Experience*, 22(5):349 – 369, May 1992.
10. S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
11. C. Daly, J. Horgan, J. Power, and J. 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.
12. S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPEC JVM98 Java benchmarks. In *13th European Conference on Object Oriented Programming*, pages 92–115, Lisbon, Portugal, June 1999.
13. N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 1996.
14. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
15. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
16. Brian Henderson-Sellers. *Object-Oriented Metrics*. Prentice Hall, 1996.
17. 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.
18. K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. 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.
19. 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.
20. T. Li, L.K. John, V. Narayanan, A. Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPEC JVM98 benchmarks. In *International Conference on Supercomputing*, pages 22–33, Santa Fe, NM, USA, May 2000.
21. Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.
22. V. Narayanan and M.I. Wolczko, editors. *Java Microarchitectures*. Kluwer Academic Publishers, 2002.
23. I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Object-Oriented Programming Systems, Languages and Applications*, pages 195 – 210, Tampa, Florida, USA, October 14-18 2001.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.

24. 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.
25. R.W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647 – 654, September 1977.
26. SPEC. 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.
27. T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java Just-In-Time compiler. In *Object-Oriented Programming Systems, Languages and Applications*, pages 180 – 194, Tampa, Florida, USA, October 14-18 2001.
28. Sun Microsystems. The Java HotSpot virtual machine. Technical White Paper, 2001. http://java.sun.com/products/hotspot/.
29. V. Sundaresan, L. Hendren, and C. Razafimahefa. Practical virtual method call resolution for Java. In *Object Oriented Programming Systems Languages and Applications*, pages 264 – 280, Minneapolis, Minnesota, USA, October 15-19 2000.
30. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *9th NRC/IBM Center for Advanced Studies Conference*, pages 125–135, Toronto, Canada, November 1999.
31. J. Whaley. Partial method compilation using dynamic profile information. In *Object-Oriented Programming Systems, Languages and Applications*, pages 166 – 179, Tampa, Florida, USA, October 14-18 2001.
32. T.J. Wilkinson. KAFFE, a virtual machine to run Java code, July 2000. http://www.kaffe.org.

Published in: Concurrency and Computation: Practice and Experience
Vol. 17, No. 7-8, June-July, 2005, pp. 757-773.