The DaCapo Benchmarks: Java Benchmarking Development and Analysis

Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Elliot B Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, Ben Wiedermann

ARC DP0452011, DARPA F33615-03-C-4106, DARPA NBCH30390004, IBM, and Intel. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

Abstract
Since benchmarks drive computer science research and industry product development, which ones we use and how we evaluate them are key questions for the community. Despite complex run-time tradeoffs due to dynamic compilation and garbage collection required for Java programs, many evaluations still use methodologies developed for SPEC, the dominant purveyor of benchmarks, compounded this problem by institutionalizing these methodologies for their Java benchmark suite. This paper recommends benchmarking selection and evaluation methodologies, and introduces the DaCapo benchmarks, a set of open source, client-side Java benchmarks. We demonstrate that the complex interactions of (1) architecture, (2) compiler, (3) virtual machine, (4) memory management, and (5) application require more extensive evaluation than C, C++, and Fortran which stress (4) much less, and do not require (3). We use and introduce new value, time-series, and statistical metrics for static and dynamic properties such as code complexity, code size, heap composition, and pointer mutations. No benchmark suite is definitive, but these metrics show that DaCapo improves over SPEC Java in a variety of ways, including more complex code, richer object behaviors, and more demanding memory system requirements. This paper takes a step towards improving methodologies for choosing and evaluating benchmarks to foster innovation in system design and implementation for Java and other managed languages.

Categories and Subject Descriptors C.4 [Measurement Techniques]
General Terms Measurement, Performance
Keywords methodology, benchmark, DaCapo, Java, SPEC

1. Introduction
When researchers explore new system features and optimizations, they typically evaluate them with benchmarks. If the idea does not improve a set of interesting benchmarks, researchers are unlikely to submit the idea for publication, or if they do, the community is unlikely to accept it. Thus, benchmarks set standards for innovation and can encourage or stifle it.

For Java, industry and academia typically use the SPEC Java benchmarks (the SPECjvm98 benchmarks and SPECjbb2000 [37, 38]). When SPEC introduced these benchmarks, their evaluation rules and the community’s evaluation metrics glossed over some of the key questions for Java benchmarking. For example, (1) SPEC reporting of the “best” execution time is taken from multiple iterations of the benchmark within a single execution of the virtual machine, which will typically eliminate compile time. (2) In addition to steady state, application performance, a key question for Java virtual machines (JVMs) is the tradeoff between compile and application time, yet SPEC does not require this metric, and the community often does not report it. (3) SPEC does not require reports on multiple heap sizes and thus does not explore the space-time tradeoff automatic memory management (garbage collection) must make. SPEC specifies three possible heap sizes, all of which over-provision the heap. Some researchers and industry evaluations of course do vary and report these metrics, but many do not.

This paper introduces the DaCapo benchmarks, a set of general purpose, realistic, freely available Java applications. This paper also recommends a number of methodologies for choosing and evaluating Java benchmarks, virtual machines, and their memory management systems. Some of these methodologies are already in use. For example, Eeckhout et al. recommend that hardware vendors use multiple JVMs for benchmarking because applications vary significantly based on JVM [19]. We recommend and use this methodology on three commercial JVMs, confirming none is a consistent winner and benchmark variation is large. We recommend here a deterministic methodology for evaluating compiler optimizations that holds the compiler workload constant, as well as the standard steady-state stable performance methodology. For evaluating garbage collectors, we recommend multiple heap sizes and deterministic collector configurations. We also suggest new and previous methodologies for selecting benchmarks and comparing them. For example, we recommend time-series data versus single values, including heap composition and pointer distances for live objects as well as allocated objects. We also recommend principal component analysis [13, 18, 19] to assess differences between benchmarks.

We use these methodologies to evaluate and compare DaCapo and SPEC, finding that DaCapo is more complex in terms of static and dynamic metrics. For example, DaCapo benchmarks have much richer code complexity, class structures, and class hierarchies than SPEC according to the Chidamber and Kemerer metrics [12]. Furthermore, this static complexity produces a wider variety and more complex object behavior at runtime, as measured by data structure complexity, pointer source/target heap distances, live and allocated object characteristics, and heap composition. Principal component analysis using code, object, and architecture behavior metrics differentiates all the benchmarks from each other.

The main contributions of this paper are new, more realistic Java benchmarks, an evaluation methodology for developing benchmark suites, and performance evaluation methodologies. Needless to say,
the DaCapo benchmarks are not definitive, and they may or may not be representative of workloads that vendors and clients care about most. Regardless, we believe this paper is a step towards a wider community discussion and eventual consensus on how to select, measure, and evaluate benchmarks, VMs, compilers, runtimes, and hardware for Java and other managed languages.

2. Related Work

We build on prior methodologies and metrics, and go further to recommend how to use them to select benchmarks and for best practices in performance evaluation.

2.1 Java Benchmark Suites

In addition to SPEC (discussed in Section 3), prior Java benchmarks suites include Java Grande [26], Jolden [11, 34], and Ashes [17]. The Java Grande Benchmarks include programs with large demands for memory, bandwidth, or processing power [26]. They focus on array intensive programs that solve scientific computing problems. The programs are sequential, parallel, and distributed. They also include microbenchmark tests for language and communication features, and some cross-language tests for comparing C and Java. DaCapo also focuses on large, realistic programs, but not on parallel or distributed programs. The DaCapo benchmarks are more general purpose, and include both client and server side applications.

The Jolden benchmarks are single-threaded Java programs rewritten from parallel C programs that use dynamic pointer data structures [11, 34]. These programs are small kernels (less than 600 lines of code) intended to explore pointer analysis and parallelization, not complete systems. The Soot project distributes the Ashes benchmarks with their Java compiler infrastructure, and include the Jolden benchmarks, a few more realistic benchmarks such as their compiler, and some interactive benchmarks [17]. The DaCapo benchmarks contain many more realistic programs, and are more ambitious in scope.

2.2 Benchmark Metrics and Characterization

Dufour et al. recommend characterizing benchmarks with architecture independent value metrics that summarize: (1) size and structure of program, (2) data structures, (3) polymorphism, (4) memory, and (5) concurrency into a single number [17]. We do not consider concurrency metrics to limit the scope of our efforts. We use metrics from the first four categories and add metrics, such as filtering for just the live objects, that better expose application behavior. Our focus is on continuous metrics, such as pointer distributions and heap composition graphs, rather than single values. Dufour et al. show how to use these metrics to drive compiler optimization explorations, whereas we show how to use these metrics to develop methodologies for performance and benchmark evaluation.

Prior work studied some of the object properties we present here [4, 15, 21, 39], but not for the purposes of driving benchmark selection and evaluation methodologies. For example, Dieckmann and Hötzle [15] measure object allocation properties, and we add to their analysis live object properties and pointer demographics. Stefanović pioneered the use of heap composition graphs which we use here to show inherent object lifetime behaviors [39].

2.3 Performance Evaluation Methodologies

Eeckhout et al. study SPECjvm98 and other Java benchmarks using a number of virtual machines on one architecture, AMD’s K7 [19]. Their cluster analysis shows that methodologies for designing new hardware should include multiple virtual machines and benchmarks because each widely exercises different hardware aspects. One limitation of their work is that they use a fixed heap size, which as we show masks the interaction of the memory manager’s space-time tradeoff in addition to its influence on mutator locality. We add to Eeckhout et al.’s good practices in methodology that the hardware designers should include multiple heap sizes and memory management strategies. We confirm Eeckhout et al.’s finding. We present results for three commercial JVMs on one architecture that show a wide range of performance sensitivities. No one JVM is best across the suite with respect to compilation time and code quality, and there is a lot a variation. These results indicate there is plenty of room for improving current commercial JVMs.

Many recent studies examine and characterize the behavior of Java programs in simulation or on hardware [19, 21, 23, 28, 29, 30, 31, 32, 33]. This work focuses on workload characterization, application behavior on hardware, and key differences with C programs. For example, Hauswirth et al. mine application behavior to understand performance [21]. The bulk of our evaluation focuses on benchmark properties that are independent of any particular hardware or virtual machine implementation, whereas this prior work concentrates on how applications behave on certain hardware with one or more virtual machines. We extend these results to suggest that these characteristics can be used to separate and evaluate the benchmarks in addition to the software and hardware running them. Much of this Java performance analysis work either disables garbage collection [15, 35], which introduces unnecessary memory fragmentation, or holds the heap size and/or garbage collector constant [19, 28], which may hide locality effects.

A number of researchers examine garbage collection and its influence on application performance [3, 4, 20, 22, 28, 40]. For example, Kim and Hsu use multiple heap sizes and simulate different memory hierarchies with a whole heap mark-sweep algorithm, assisted by occasional compaction [28]. Kim and Hsu, and Rajan et al. [33] note that a mark-sweep collector has a higher miss rate than the application itself because the collector touches reachable data that may not be in the program’s current working set. Blackburn et al. use the methodology we recommend here for studying the influence of copying, mark-sweep, and reference counting collectors, and their generational variants on three architectures [4]. They show a contiguously allocating generational copying collector delivers better mutator cache performance and total performance than a whole-heap mark-sweep collector with a free-list. A few studies explore heap size effects on performance [9, 10, 28], and as we show here, garbage collectors are very sensitive to heap size, and in particular to tight heaps. Diwan et al. [16, 41], Hicks et al. [22], and others [7, 8, 24] measure detailed, specific mechanism costs and architecture influences [16], but do not consider a variety of collection algorithms. Our work reflects these results and methodologies, but makes additional recommendations.

3. Benchmark and Methodology Introduction

This section describes SPEC Java and SPEC execution rules, how we collected DaCapo benchmarks, and our execution harness.

3.1 SPEC Java Benchmarks.

We compare the DaCapo suite to SPECjvm98 [37] and a modified version of SPECjbb2000 [38], and call them the SPEC Java benchmarks, or SPEC for short. We exclude SPECjAppServer because it requires multiple pieces of hardware and software to execute. The original SPECjbb2000 is a server-side Java application and reports its score as work done over a fixed time rather than elapsed time for a fixed work load. Although throughput (measuring work done over a fixed time) is one important criteria for understanding applications such as transaction processing systems, most applications are not throughput oriented. Superficially, the difference between fixing the time and workload is minor, however a variable workload is methodologically problematic. First, throughput workloads force a repetitive loop into the benchmark, which influences JIT optimization strategies and opportunities for parallelism, but is not representative of the wide range of non-repetitive workloads. Furthermore,
variable workloads make performance hard to analyze and reason about. For example, the level and number of classes optimized and re-optimized at higher levels and the number of garbage collect-
tions vary with the workload, leading to complex cascading effects on overall performance. We therefore modify SPECjbb2000, creat-
ing pseudojbb, which executes a fixed workload (by default, 70,000 transactions execute against a single warehouse).

SPEC benchmarking rules discourage special casing the vir-
tual machine, compiler, and/or architecture for a specific SPEC Java benchmark. They specify the largest input size (100), se-
quencing through the benchmarks, no harness caching, and no pre-
compilation of classes. The SPECjvm98 harness runs all the bench-
marks multiple times, and intersperses untimed and timed execu-
tions. Benchmarkers may run all the programs as many times as they like, and then report the best and worst results using the same virtual machine and compiler configurations. SPEC indicates that reporting should specify the memory sizes: 48MB, 48–256MB, and greater than 256MB, but does not require reporting all three. All these sizes over provision the heap. Excluding the virtual machine, SPEC programs allocate up to 271MB, and have at most 8MB live in the heap at any time, except for pseudojbb with 21MB live (see Section 7). Since 2000, none of the vendors has published results for the smaller heaps.

The SPEC committee is currently working on collecting a new set of Java benchmarks. The SPEC committee consists of industrial representatives and a few academics. One of their main criteria is representativeness, which industry is much better to judge than academia. When SPEC releases new benchmark sets, they include a performance comparison point. They do not include or describe any measured metrics on which they based their selection. This paper suggests methodologies for both selecting and evaluating Java Benchmarks, which are not being used or recommended in current industrial standards, SPEC or otherwise.

3.2 DaCapo Benchmarks

We began the DaCapo benchmarking effort in mid 2003 as the re-
sult of an NSF review panel in which the panel and the DaCapo research group agreed that the existing Java benchmarks were lim-
iting our progress. What followed was a two-pronged effort to iden-
tify suitable benchmarks, and develop a suite of analyses to char-
acterize candidate benchmarks and evaluate them for inclusion. We began with the following criteria.

1. Diverse real applications. We want applications that are widely used to provide a compelling focus for the community’s innovation and optimizations, as compared to synthetic benchmarks.

2. Ease of use. We want the applications to be relatively easy to use and measure.

We implemented these criteria as follows.

1. We chose only open source benchmarks and libraries.

2. We chose diverse programs to maximize coverage of application domains and application behaviors.

3. We focused on client-side benchmarks that are easy to measure in a completely standard way, with minimal dependences outside the scope of the host JVM.

4. We excluded GUI applications since they are difficult to benchmark systematically. In the case of eclipse, we exercise a non-GUI subset.

5. We provide a range of inputs. With the default input sizes, the programs are timely enough that it takes hours or days to execute thousands of invocations of the suite, rather than weeks. With the exception of eclipse, which runs for around a minute, each benchmark executes for between 5 and 20 seconds on contemporary hardware and JVMs.

We considered other potential criteria, such as long running, GUI, and client-server applications. We settled on the above characteristics because their focus is similar to the existing SPEC benchmarks, while addressing some of our key concerns. Around 20 stu-
dents and faculty at six institutions then began an iterative process of identifying, preparing, and experimenting with candidate benchmarks. Realizing the difficulty of identifying a good benchmark suite, we made the DaCapo benchmark project open and transpar-
ent, inviting feedback from the community [14]. As part of this process, we have released three beta versions.

We identified a broad range of static and dynamic metrics, including some new ones, and developed a framework in Jikes RVM [1] for performing these detailed analyses. Sections 6, 7, and 8 describe these metrics. We systematically analyzed each can-
didate to identify ones with non-trivial behavior and to maximize the suite’s coverage. We included most of the benchmarks we eval-
uated, excluding only a few that were too trivial or whose license agreements were too restrictive, and one that extensively used exceptions to avoid explicit control flow.

The Constituent Benchmarks

We now briefly describe each benchmark in the final pre-release of the suite (beta-2006-08) that we use throughout the paper. More detailed descriptions appear in Figures 4 through 14. The source code and the benchmark harness are available on the DaCapo benchmark web site [14].

antlr A parser generator and translator generator.

bloat A bytecode-level optimization and analysis tool for Java.

chart A graph plotting toolkit and pdf renderer.
eclipse An integrated development environment (IDE).
fop An output-independent print formatter.

hsqldb An SQL relational database engine written in Java.

jython A python interpreter written in Java.
luindex A text indexing tool.
lusearch A text search tool.
pmd A source code analyzer for Java.
xalan An XSLT processor for transforming XML documents.

The benchmark suite is packaged as a single jar file containing a harness (licensed under the Apache Public License [2]), all the benchmarks, the libraries they require, three input sizes, and input data (e.g., luindex, lusearch and xalan all use the works of Shake-
ppeare). We experimented with different inputs and picked repre-
sentative ones.

The Benchmark Harness

We provide a harness to invoke the benchmarks and perform a validity check that insures each benchmark ran to completion correctly. The validity check performs checksums on err and out streams during benchmark execution and on any generated files after benchmark execution. The harness passes the benchmark if its checksums match pre-calculated values.

The harness supports a range of options, including user-specified hooks to call at the start and end of the benchmark and/or after the benchmark warm-up period, running multiple benchmarks, and printing a brief summary of each benchmark and its origins. It also supports workload size (small, default, large), which iteration (first, second, or nth), or a performance-stable iteration for reporting ex-
ecution time. To find a performance-stable iteration, the harness takes a window size $w$ (number of executions) and a convergence target $\nu$, and runs the benchmark repeatedly until either the coeffi-
cient of variation, $\frac{\sigma}{w}$, of the last $w$ runs drops below $\nu$, or reports failure if the number of runs exceeds a maximum $m$ (where $\sigma$ is the standard deviation and $m$ is the arithmetic mean of the last $w$ exe-
cution times). Once performance stabilizes, the harness reports the execution time of the next iteration. The harness provides defaults for $w$ and $\nu$, which the user may override.
mixed hot methods and baseline compiled code. Compiling all the methods at the highest level with static profile information or using all baseline code is also deterministic, but it does not provide a realistic code base. These methodologies provide compiler and memory management researchers a way to control the virtual machine and compiler, holding parts of the system constant to tease apart the influence of proposed improvements. We highly recommend these methodologies together with a clear specification and justification of which methodology is appropriate and why. In Section 5.1, we provide an example evaluation that uses the deterministic stable methodology which is appropriate because we compare garbage collection algorithms across a range of architectures and heap sizes, and thus want to minimize variation due to sampling and JIT compilation.

5. Benchmarking Methodology
This section argues for using multiple architectures and multiple heap sizes related to each program’s maximum live size to evaluate the performance of Java, memory management, and its virtual machines. In particular, we show a set of results in which the effects of locality and space efficiency trade off, and therefore heap size and architecture choice substantially affect quantitative and qualitative conclusions. The point of this section is to demonstrate that because of the variety of implementation issues that Java programs encompass, measurements are sensitive to benchmarks, the underlying architecture, the choice of heap size, and the virtual machine. Thus presenting results without varying these parameters is at best unhelpful, and at worst, misleading.

5.1 How not to cook your books.
This experiment explores the space-time tradeoff of two full heap collectors: SemiSpace and MarkSweep as implemented in Jikes RVM [1] with MMTk [4, 5] across four architectures. We experimentally determine the minimum heap size for each program using MarkCompact in MMTk. (These heap sizes, which are specific to MMTk and Jikes RVM version 2.4++, can be seen in the top x-axes of Figures 2(a)–(d).) The virtual machine triggers collection when the application exhausts the heap space. The SemiSpace
Table 1. Cross JVM Comparisons

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>First iteration</th>
<th>Second iteration</th>
<th>Third iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B/A</td>
<td>C/A</td>
</tr>
<tr>
<td>201_compress</td>
<td>7.2</td>
<td>0.76</td>
<td>0.75</td>
</tr>
<tr>
<td>202_jess</td>
<td>3.2</td>
<td>0.80</td>
<td>0.58</td>
</tr>
<tr>
<td>205_raytrace</td>
<td>2.6</td>
<td>0.86</td>
<td>0.54</td>
</tr>
<tr>
<td>209_db</td>
<td>8.0</td>
<td>1.07</td>
<td>1.00</td>
</tr>
<tr>
<td>213_javac</td>
<td>6.0</td>
<td>0.48</td>
<td>0.69</td>
</tr>
<tr>
<td>222_mpeg</td>
<td>3.8</td>
<td>1.73</td>
<td>1.22</td>
</tr>
<tr>
<td>227_mir</td>
<td>2.9</td>
<td>0.72</td>
<td>0.51</td>
</tr>
<tr>
<td>228_jack</td>
<td>5.7</td>
<td>1.08</td>
<td>0.61</td>
</tr>
<tr>
<td>geomean</td>
<td>0.94</td>
<td>0.74</td>
<td>0.68</td>
</tr>
</tbody>
</table>

DaCapo

| antlr | 6.0 | 0.53 | 0.68 | 0.53 | 3.4 | 0.60 | 0.94 | 0.69 | 0.66 | 3.2 | 0.69 | 0.97 | 0.69 | 0.64 |
| xalan | 12.0 | 0.98 | 1.03 | 0.98 | 9.7 | 1.28 | 1.20 | 1.00 | 0.93 | 9.1 | 1.24 | 1.28 | 1.00 | 0.88 |
| jart | 12.2 | 1.97 | 1.47 | 0.97 | 9.5 | 1.30 | 1.68 | 1.00 | 0.90 | 9.2 | 0.73 | 1.71 | 1.00 | 0.75 |
| eclipse | 61.7 | 1.28 | 0.96 | 0.96 | 39.4 | 1.60 | 1.17 | 1.00 | 0.74 | 23.8 | 1.60 | 1.94 | 1.00 | 0.54 |
| top | 7.1 | 0.40 | 0.40 | 0.40 | 4.8 | 0.33 | 0.35 | 0.33 | 0.63 | 5.1 | 0.31 | 0.35 | 0.31 | 0.66 |
| hsqldb | 12.0 | 0.82 | 0.47 | 0.47 | 7.7 | 0.67 | 0.66 | 0.66 | 0.66 | 7.3 | 0.86 | 0.68 | 0.68 | 0.67 |
| luindex | 15.5 | 1.04 | 0.94 | 0.94 | 9.8 | 1.41 | 1.41 | 1.00 | 0.80 | 9.1 | 1.55 | 1.49 | 1.00 | 0.79 |
| lusearch | 13.1 | 0.74 | 0.90 | 0.74 | 10.6 | 0.92 | 1.06 | 0.92 | 0.91 | 10.5 | 1.56 | 1.07 | 1.00 | 1.10 |
| jython | 16.5 | 0.52 | 0.68 | 0.52 | 8.3 | 0.92 | 2.87 | 0.92 | 1.09 | 7.9 | 0.92 | 0.83 | 0.83 | 0.59 |
| msql | 10.4 | 1.04 | 0.95 | 0.95 | 7.5 | 1.50 | 1.15 | 1.00 | 0.89 | 6.9 | 1.18 | 1.27 | 1.00 | 0.76 |
| xalan | 8.3 | 0.87 | 0.90 | 0.87 | 5.3 | 1.53 | 1.20 | 1.00 | 0.86 | 5.0 | 1.24 | 1.26 | 1.00 | 0.76 |
| geomean | 0.84 | 0.85 | 0.76 | 0.76 | 1.10 | 1.25 | 0.86 | 0.83 | 0.83 | 1.08 | 1.17 | 0.84 | 0.74 |

This section explores the sensitivity of performance results to JVMs. Table 1 presents execution times and differences for three leading commercial Java 1.5 JVMs running one, two, and three iterations of the SPEC Java and DaCapo benchmarks. We have made the JVMs anonymous (‘A’, ‘B’, & ‘C’) in deference to license agreements and because JVM identity is not pertinent to the point. We use a 2GHz Intel Pentium M with 1GB of RAM and a 2MHz 1.2 cache, and in each case we run JVM ‘B’ out of the box, with no special command line settings. Columns 2–4 show performance for a single iteration, which will usually be most impacted by compilation time. Columns 6–8 present performance for a single iteration, which will usually be most impacted by compilation time. Columns 6–8 present performance for a single iteration, which will usually be most impacted by compilation time. Columns 6–8 present performance for a single iteration, which will usually be most impacted by compilation time. Columns 6–8 present performance for a single iteration, which will usually be most impacted by compilation time.

Some interesting result is that no JVM is uniformly best on all configurations. The results show there is a lot of room for overall JVM improvements. For DaCapo, potential improvements range from 14% (second iteration) to 25% (first iteration). Even for SPEC potential improvements range from 6% (second iteration) to 32% (first iteration). If a single JVM could achieve the best performance on DaCapo across the benchmarks, it would improve performance by a geometric mean of 24% on the first iteration, 14% on the second iteration, and 16% on the third iteration (the geomean row). Among the notable results are that JVM C slows down significantly for the second iteration of jython, and then performs best.
on the third iteration, a result we attribute to aggressive hotspot compilation during the second iteration. The eclipse benchmark appears to take a long time to warm up, improving considerably in both the second and third iterations. On average, SPEC benchmarks speed up much more quickly than the DaCapo benchmarks, which is likely a reflection on their smaller size and simplicity. We demonstrate this point quantitatively in the next section.

These results reinforce the importance of good methodology and the choice of benchmark suite, since we can draw dramatically divergent conclusions by simply selecting a particular iteration, virtual machine, heap size, architecture, or benchmark.

6. Code Complexity and Size

This section shows static and dynamic software complexity metrics which are architecture and virtual machine independent. We present Chidamber and Kemerer’s software complexity metrics [12] and a number of virtual machine and architecture independent dynamic metrics, such as, classes loaded and bytecode compiled. Finally, we present a few virtual machine dependent measures of dynamic behavior, such as, methods bytecode compiled the compiler detects as frequently executed (hot), and instruction cache misses. Although we measure these features with Jikes RVM, Eckhout et al. [19] show that for SPEC, virtual machines fairly consistently identify the same hot regions. Since the DaCapo benchmarks are more complex than SPEC, this trend may not hold as well for them, but we believe these metrics are not overly influenced by our virtual machine. DaCapo and SPEC differ quite a bit; DaCapo programs are more complex, object-oriented, and exercise the instruction cache more.

6.1 Code Complexity

To measure the complexity of the benchmark code, we use the Chidamber and Kemerer object-oriented programming (CK) metrics [12] measured with the ckjm software package [36]. We apply the CK metrics to classes that the application actually loads during execution. We exclude standard libraries from this analysis as they are heavily duplicated across the benchmarks (column two of Table 3 includes all loaded classes). The average DaCapo program loads more than twice as many classes during execution as SPEC. The following explains what the CK metrics reveal and the results for SPEC and DaCapo.

WMC Weighted methods per class. Since ckjm uses a weight of 1, WMC is simply the total number of declared methods for the loaded classes. Larger numbers show that a program provides more behaviors, and we see SPEC has substantially lower WMC values than DaCapo, except for _213_javac, which as the table shows is the richest of the SPEC benchmarks and usually falls in the middle or top of the DaCapo program range of software complexity. Unsurprisingly, fewer methods are declared (WMC in Table 2) than compiled (Table 3), but this difference is only dramatic for eclipse.

DIT Depth Inheritance Tree. DIT provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1. Except for _213_javac and _202_jess, DaCapo programs typically have deeper inheritance trees.

NOC Number of Children. NOC is the number of immediate subclasses of the class. Table 2 shows that in SPEC, only _213_javac has any interesting behavior, but hasqldb, luindex, lusearch and pmd in DaCapo also have no superclass structure.

CBO Coupling between object classes. CBO represents the number of classes coupled to a given class (effort couplings). Method calls, field accesses, inheritance, arguments, return types, and exceptions all couple classes. The interactions between objects and classes is substantially more complex for DaCapo compared to SPEC. However, both _202_jess and _213_javac have relatively high CBO values.

RFC Response for a Class. RFC measures the number of different methods that may execute when a method is invoked. Ideally, we would find for each method of the class, the methods that class will call, and repeat for each called method, calculating the transitive closure of the method’s call graph. Ckjm calculates a rough approximation to the response set by inspecting method calls within the class’s method bodies. The RFC metric for DaCapo shows a factor of around five increase in complexity over SPEC.

LCOM Lack of cohesion in methods. LCOM counts methods in a class that are not related through the sharing of some of the class’s fields. The original definition of this metric (used in ckjm) considers all pairs of a class’s methods, subtracting the number of method pairs that share a field access from the number of method pairs that do not. Again, DaCapo is more complex, e.g., eclipse and pmd have LCOM metrics at least two orders of magnitude higher than any SPEC benchmark.

In summary, the CK metrics show that SPEC programs are not very object-oriented in absolute terms, and that the DaCapo benchmarks are significantly richer and more complex than SPEC. Furthermore, DaCapo benchmarks extensively use object-oriented features to manage their complexity.

6.2 Code Size and Instruction Cache Performance

This section presents program size metrics. Column 2 of Table 3 shows the total number of classes loaded during the execution of each benchmark, including standard libraries. Column 3 shows the total number of declared methods in the loaded classes (compare to column 2 of Table 2, which excludes standard libraries). Columns 4 and 5 show the number of methods compiled (executed at least

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WMC</th>
<th>DIT</th>
<th>NOC</th>
<th>CBO</th>
<th>RFC</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DaCapo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>3094</td>
<td>202</td>
<td>1011</td>
<td>202</td>
<td>201</td>
<td>202</td>
</tr>
<tr>
<td>jython</td>
<td>487</td>
<td>1419</td>
<td>202</td>
<td>1011</td>
<td>202</td>
<td>202</td>
</tr>
<tr>
<td>xalan</td>
<td>971</td>
<td>397</td>
<td>202</td>
<td>1011</td>
<td>202</td>
<td>202</td>
</tr>
<tr>
<td>geomean</td>
<td>1857</td>
<td>139</td>
<td>109</td>
<td>935</td>
<td>442</td>
<td>2256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WMC</th>
<th>Weighted methods/class</th>
</tr>
</thead>
<tbody>
<tr>
<td>DaCapo</td>
<td>SPEC</td>
</tr>
<tr>
<td>154</td>
<td>19</td>
</tr>
<tr>
<td>614</td>
<td>97</td>
</tr>
<tr>
<td>330</td>
<td>33</td>
</tr>
<tr>
<td>152</td>
<td>12</td>
</tr>
<tr>
<td>1011</td>
<td>186</td>
</tr>
<tr>
<td>367</td>
<td>40</td>
</tr>
<tr>
<td>332</td>
<td>33</td>
</tr>
<tr>
<td>375</td>
<td>46</td>
</tr>
<tr>
<td>541</td>
<td>35</td>
</tr>
<tr>
<td>min</td>
<td>152</td>
</tr>
<tr>
<td>max</td>
<td>1011</td>
</tr>
<tr>
<td>geomean</td>
<td>366</td>
</tr>
</tbody>
</table>

Table 2. CK Metrics for Loaded Classes (Excluding Libraries)
### Bytecodes Compiled and Instruction Cache Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Classes Loaded</th>
<th>Methods Declared</th>
<th>Methods &amp; Bytecodes Compiled</th>
<th>I-Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Methods</td>
<td>L1 I-cache, ITLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>/ms norm, /ms norm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BC KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BC KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Methods</td>
<td>Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BC KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BC KB</td>
<td></td>
</tr>
<tr>
<td>SPEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_201_compress</td>
<td>157</td>
<td>1118</td>
<td>254</td>
<td>23.9</td>
</tr>
<tr>
<td>_202_jess</td>
<td>92</td>
<td>1777</td>
<td>655</td>
<td>42.4</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>177</td>
<td>1316</td>
<td>381</td>
<td>32.4</td>
</tr>
<tr>
<td>_209_db</td>
<td>149</td>
<td>1108</td>
<td>249</td>
<td>23.7</td>
</tr>
<tr>
<td>_213_javac</td>
<td>302</td>
<td>2261</td>
<td>978</td>
<td>89.0</td>
</tr>
<tr>
<td>_222_mpegadio</td>
<td>200</td>
<td>1407</td>
<td>425</td>
<td>68.3</td>
</tr>
<tr>
<td>_227_mtrnt</td>
<td>178</td>
<td>1318</td>
<td>379</td>
<td>32.4</td>
</tr>
<tr>
<td>_228_jack</td>
<td>202</td>
<td>1392</td>
<td>488</td>
<td>53.6</td>
</tr>
<tr>
<td>pseudojb</td>
<td>238</td>
<td>2622</td>
<td>824</td>
<td>69.7</td>
</tr>
<tr>
<td>mm</td>
<td>149</td>
<td>1108</td>
<td>249</td>
<td>23.7</td>
</tr>
<tr>
<td>max</td>
<td>302</td>
<td>2622</td>
<td>978</td>
<td>89.0</td>
</tr>
<tr>
<td>DaCapo</td>
<td>204</td>
<td>1523</td>
<td>464</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.** Bytecodes Compiled and Instruction Cache Characteristics

once) and the corresponding KB of bytecodes (BC KB) for each benchmark. We count bytecodes rather than machine code, as it is not virtual machine, compiler, or ISA specific. The DaCapo benchmarks average more than twice the number of classes, three times as many declared methods, four times as many compiled methods, and four times the volume of compiled bytecodes, reflecting a substantially larger code base than SPEC. Columns 6 and 7 show how much code is optimized by the JVM’s adaptive compiler over the course of two iterations of each benchmark (which Eeckhout et al.’s results indicate is probably representative of most hotspot finding virtual machines [19]). Columns 8 and 9 show that the DaCapo benchmarks have a much lower proportion of methods which the adaptive compiler regards as hot. Since the virtual machine selects these methods based on frequency thresholds, and these thresholds are tuned for SPEC, it may be that the compiler should be selecting warm code. However, it may simply reflect the complexity of the benchmarks. For example, eclipse has nearly four thousand methods compiled, of which only 14 are regarded as hot (0.4%). On the whole, this data shows that the DaCapo benchmarks are substantially larger than SPEC. Combined with their complexity, they should present more challenging optimization problems.

We also measure instruction cache misses per millisecond as another indicator of dynamic code complexity. We measure misses with the performance counters on a 2.0 GHz Pentium M with a 32KB level 1 instruction cache and a 2MB shared level two cache, each of which are 8-way with 64 byte lines. We use Jikes RVM and only report misses during the mutator portion of the second iteration of the benchmarks (i.e., we exclude garbage collection). Columns 10 and 11 show L1 instruction misses, first as misses per millisecond, and then normalized against the geometric mean of the SPEC benchmarks. Columns 12 and 13 show ITLB misses using the same metrics. We can see that on average DaCapo has L1 I-cache misses nearly six times more frequently than SPEC, and ITLB misses about eight times more frequently than SPEC. In particular, none of the DaCapo benchmarks have remarkably few misses, whereas SPEC benchmarks _201_compress, 202_jess, and 209_db hardly ever miss the IL1. All DaCapo benchmarks have misses at least twice that of the geometric mean of SPEC.

### Objects and Their Memory Behavior

This section presents object allocation, live object, lifetime, and lifetime time-series metrics. We measure allocation demographics suggested by Dieckmann and Hölszle [15]. We also measure lifetime and live object metrics, and show that they differ substantially from allocation behaviors. Since many garbage collection algorithms are most concerned with live object behaviors, these demographics are more indicative for designers of new collection mechanisms. Other features, such as the design of per-object metadata, also depend on the demographics of live objects, rather than allocated objects.

The data described in this section and Section 8 is presented in Table 4, and in Figures 4(a) through 14(a), of which contains data for one of the DaCapo benchmarks, ordered alphabetically. In a companion technical report [6], we show these same graphs for SPEC Java. For all the metrics, DaCapo is more complex and varied in its behavior, but we must exclude SPEC here due to space limitations. Each figure includes a brief description of the benchmark, key attributes, and metrics. It also plots time series and summaries for (a) object size demographics (Section 7.2), (b) heap composition (Section 7.3), and (c) pointer distances (Section 8). Together this data shows that the DaCapo suite has rich and diverse object lifetime behaviors.

Since Jikes RVM is written in Java, the execution of the JIT compiler normally contributes to the heap, unlike most other JVMs, where the JIT is written in C. In these results, we exclude the JIT compiler and other VM objects by placing them into a separate, excluded heap. To compute the average and time-series object data, we modify Jikes RVM to keep statistics about allocations and to compute statistics about live objects at frequent snapshots, i.e., during full heap collections.
7.1 Allocation and Live Object Behaviors

Table 4 summarizes object allocation, maximum live objects, and their ratios in MB (megabytes) and objects. The table shows that DaCapo allocates substantially more objects than the SPEC benchmarks, by nearly a factor of 20 on average. The live objects and memory are more comparable; but still DaCapo has on average three times the live size of SPEC. DaCapo has a much higher ratio of allocation to maximum live size, with an average of 147 compared to SPEC’s 23 measured in MB. Two programs stand out; *python* with a ratio of 8104, and *xalan* with a ratio of 2364. The DaCapo benchmarks therefore put significantly more pressure on the underlying memory management policies than SPEC.

Nursery survival rate is a rough measure of how closely a program follows the generational hypothesis which we measure with a 4MB bounded nursery and report in the last column of Table 4. Note that nursery survival needs to be viewed in the context of heap turnover (column seven of Table 4). A low nursery survival rate may suggest low total GC workload, for example, *222_mpegaudio* and *hsqldb* in Table 4. A low nursery survival rate and a high heap turnover ratio instead suggests a substantial GC workload, for example, *eclipse* and *luindex*. SPEC and DaCapo exhibit a wide range of nursery survival rates. Blackburn et al. show that even programs with high nursery survival rates and large turnover benefit from generational collection with a copying bump-pointer nursery space [4]. For example, *213_javac* has a nursery survival rate of 26% and performs better with generational collectors. We confirm this result for all DaCapo benchmarks, even on *hsqldb* with its 63% nursery survival rate and low turnover ratio.

Table 4 also shows the average object size. The benchmark suites do not substantially differ with respect to this metric. A significant outlier is *201_compress*, which compresses large arrays of data. Other outliers include *222_mpegaudio*, *lusearch* and *xalan*, all of which also operate over large arrays.

7.2 Object Size Demographics

This section improves the above methodology for measuring object size demographics. We show that these demographics vary with time and when viewed from of perspective of allocated versus live objects. Allocation-time size demographics inform the structure of the allocator. Live object size demographics impact the design of per-object metadata and elements of the garbage collection algorithm, as well as influencing the structure of the allocator. Figures 4(a) through 14(a) each use four graphs to compare size demographics for each DaCapo benchmark. The object size demographics are measured both as a function of all allocations (top) and as a function of live objects seen at heap snapshots (bottom). In each case, we show both a histogram (left) and a time-series (right).

The allocation histogram plots the number of objects on the y-axis in each object size (x-axis in log scale) that the program allocates. The live histogram plots the average number of live objects of each size over the entire program. We color every fifth bar black to help the eye correlate between the allocation and live histograms. Consider *antlr* in Figure 4(a) and *bleat* in Figure 5(a). For *antlr*, the allocated versus live objects in a size class show only modest differences in proportions. For *bleat* however, 12% of its allocated objects are 38 bytes whereas essentially no live objects are 38 bytes, which indicates they are short lived. On the other hand, less than 1% of *bleat*’s allocated objects are 52 bytes, but they make up 20% of live objects, indicating they are long lived. Figure 14(a) shows that for *xalan* there is an even more marked difference in allocated and live objects, where 50% of allocated

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Heap Volume (MB)</th>
<th>Heap Objects</th>
<th>Mean Object Size</th>
<th>Nursery Survival %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alloc</td>
<td>Live</td>
<td>Alloc/Live</td>
<td>Alloc</td>
</tr>
<tr>
<td>201_compress</td>
<td>105.4</td>
<td>6.3</td>
<td>16.8</td>
<td>3,944</td>
</tr>
<tr>
<td>202_jess</td>
<td>262.0</td>
<td>1.2</td>
<td>221.3</td>
<td>7,955</td>
</tr>
<tr>
<td>205_raytrace</td>
<td>133.5</td>
<td>3.8</td>
<td>35.1</td>
<td>6,397</td>
</tr>
<tr>
<td>209_db</td>
<td>74.6</td>
<td>8.5</td>
<td>8.8</td>
<td>3,218</td>
</tr>
<tr>
<td>213_javac</td>
<td>178.3</td>
<td>7.2</td>
<td>24.8</td>
<td>5,911</td>
</tr>
<tr>
<td>222_mpegaudio</td>
<td>0.7</td>
<td>0.6</td>
<td>1.1</td>
<td>3,022</td>
</tr>
<tr>
<td>227_mirt</td>
<td>140.5</td>
<td>7.2</td>
<td>19.5</td>
<td>6,689</td>
</tr>
<tr>
<td>228_jack</td>
<td>270.7</td>
<td>0.9</td>
<td>292.7</td>
<td>9,393</td>
</tr>
<tr>
<td>pseudojbb</td>
<td>207.1</td>
<td>21.1</td>
<td>9.8</td>
<td>6,158</td>
</tr>
<tr>
<td>min</td>
<td>0.7</td>
<td>0.6</td>
<td>1.1</td>
<td>3,022</td>
</tr>
<tr>
<td>max</td>
<td>270.7</td>
<td>21.1</td>
<td>292.7</td>
<td>9,393</td>
</tr>
<tr>
<td>geomean</td>
<td>86.5</td>
<td>3.8</td>
<td>23.0</td>
<td>1,180,850</td>
</tr>
</tbody>
</table>

Table 4. Key Object Demographic Metrics
objects are 12 bytes, but none stay live. In fact, 65% of live objects
are 2 Kbytes, whereas they make up only 2% of allocated objects.
How well these large objects are handled will thus in large part
determine the performance of the collector on xalan.

For each allocation histogram, we also present a time series
graph in Figures 4(a) through 14(a). Each line in the time series
graph represents an object size class from the histogram on the
left. We color every fifth object size black, stack them, and place
the smallest size classes at the bottom of the graphs. The distance
between the lines indicates the cumulative number of objects allo-
cated or live of the corresponding size, as a function of time (in
bytes of allocation by convention).

Together, the histogram and time-series data show marked dif-
fferences between allocated and live object demographics. For
example, the allocation histograms for bloat, fop, and xalan (Fig-
ures 5(a), 8(a), 14(a)) are similar, but the time series data shows
many differences. The xalan program has eight allocation phases
that are self-similar and mirrored in the live data, although in dif-
f erent size class proportions. Whereas, in bloat allocation and live
object show much less phase behavior, and phases are not self-
correlated. Comparing live and allocated time-series for fop shows
a different pattern. There is a steady increase in the live objects
of each size (and consequently, probably growing data structures),
whereas fop allocates numerous sizes in a several distinct allocation
phases. Thus, the allocation and live graphs are very different. This
shows that live and allocation time series analysis can reveal com-
plexity and opportunities that a scalar metric will never capture.

7.3 Heap Composition Graphs

Figures 4(b) through 14(b) each plot heap composition in lines of
constant allocation as a function of time, measured in allocations
(top) and pointer mutations (bottom). Like the live object time
series graphs, these graphs expose the heap composition but show
object lifetime behaviors rather than object size. Since both graphs
show live objects, their shapes are similar. The heap composition
graphs group objects into cohorts based on allocation time. We
choose cohort sizes as a power of two ($2^n$) such that there are
between 100 and 200 cohorts, shown as a line in each graph. The
top line corresponds to the oldest cohort and indicates the total
volume of live objects in the heap. The gaps between each of the
lines reflects the amount in each cohort, and when objects in a
cohort die, adjacent lines move closer together or if they all die,
the lines merge. It is not uncommon for programs to immediately
allocate long lived data, indicated by a gap between the top line and
the other cohorts; bloat, hsqldb, python, and lusearch all show this
behavior in Figures 5(b), 9(b), 10(b), and 12(b).

Qualitatively, the complexity of the graphs in Figures 4(b)
through 14(b) reflect the object lifetime behaviors of each of the
benchmarks. With the exception of python and lusearch, the
DaCapo benchmarks show much richer lifetime behaviors than
SPEC [6]; python is an interpreter, which leads to a highly regular
execution pattern. Although python allocates more than any of the
SPEC benchmarks, its behavior is highly regular. We experimented
with a number of interpreted workloads and found very similar,
highly regular behavior, suggesting that the interpreter rather than
the interpreted program dominates. The programs chart and xalan
show distinct self-similar phases with respect to object lifetimes
in Figures 6(b) and 14(b). The programs fop and hsqldb show
regular, steady heap growth in Figures 8(b) and 9(b). On the other
hand, bloat, eclipse, lunit, and pmd show irregular, complex
object lifetime patterns in Figures 5(b), 7(b), 11(b), and 13(b).

8. Reference Behavior in Pointer Distances

Java programs primarily use pointer-based data structures. This
section provides statistics that describe the connectivity of the data
structures created and manipulated by the DaCapo benchmarks.

We measure pointer distance between its source and target objects
by the relative ages of the objects, for both static snapshots of the
heap, and dynamically as pointers change. These properties
influence aspects of memory performance, such as temporal and
spatial locality and the efficacy of generational garbage collectors.

Figures 4(c) through 14(c) show the relative distances between
the sources and targets of pointers in the heap for each benchmark.
Pointer distance is measured by the difference between the target
and source object positions within (a close approximation to) a per-
fec tly compacted heap. We approximate a continuously perfectly
compacted heap by tracking cohort sizes and the logical position
of each object within each cohort during frequent garbage collec-
tions. The youngest object has a heap position of 0 and the oldest
has a heap position equal to the volume of live objects in the heap.
Thus, positive values are old to young object pointers, and negative
values are young to old.

We include both a 'static' snapshot measure of pointer distance,
and a 'dynamic' mutation measure. Snapshot pointer distance is
established by examining all pointers in the live object graph at a
garbage collection–measuring the state of the object graph. Mu-
tation distance is established by examining every pointer as it is
created–measuring the activity over the object graph. We express
these metrics as aggregate histograms for the execution of the en-
tire benchmark, and as a time series to reflect the changing shape
of the histogram over time (measured in mutations).

We first consider snapshot pointer distance, the top histogram
and time series in Figures 4(c) through 14(c). The most striking
feature of these graphs is the wide range of behaviors displayed
by the benchmarks. Several programs show very irregular time-
varying behavior, e.g., antlr, chart, eclipse, and lunit; whereas
bloat hsqldb, and pmd are more stable, but still vary a bit; and xalan
shows a very complex, but exactly repeated pattern.

The mutation pointer distance graphs have the same axes and
are shown below each snapshot pointer distance figure. These
graphs are computed by tracking pointer distances at all pointer
stores (in a write barrier), rather than at static snapshots of the
heap. These graphs show a wider range of irregularity and patterns
than the heap snapshots.

To illustrate the differences between these metrics, consider
bloat in Figure 5(c). Many pointers point from old to new objects
(positive numbers in the snapshot graphs in the top half of Fig-
ure: 5(c)), but almost all pointer mutations install new to old point-
ers (negative numbers in the mutation graphs). The snapshot graphs
indicate that around 40% of pointers will point from old to new at
any given snapshot (see the top-most line in the time series) and
about 60% will point from new to old (the bottom-most line). On
the other hand, the mutation graphs show that for most of the execu-
tion of the benchmark, nearly 100% of pointer mutations are in the
new to old direction. The divergence of the snapshot and mutation
data, and the time-varying nature of each highlight the limitations
of single value summaries of benchmark behaviors.

9. Principal Components Analysis

Previous sections demonstrate that DaCapo is more object oriented,
more complex, and larger than SPEC, whereas this section demon-
strates that all the constituent programs differ from each other, us-
ing principal component analysis (PCA) [18]. This result indicates
that we satisfy our goal of program diversity. It also confirms that
DaCapo benchmarks differ from SPEC, which is unsurprising by
now given the results from the preceding sections.

PCA is a multivariate statistical technique that reduces a large
N dimensional space into a lower dimensional uncorrelated space.
PCA generates a positive or negative weight (factor loading) associ-
ated with each metric. These weights transform the original higher
dimension space into P principal components using linear equa-
tions. We follow the PCA methodology from prior work [13, 19],
but use different constituent metrics. Table 5 shows these metrics which cover architecture, code, and memory behavior. We include architecture metrics to expand over the code and memory metrics presented and explored in depth by previous sections, and to see which of these differentiate the benchmarks. Our code metrics include the i-cache miss rate for each benchmark, the number of methods compiled and the volume of bytecodes compiled. The memory metrics include static and dynamic pointer distances, and incoming and outgoing pointer distributions.

Following prior work [27], our architecture metrics are micro-architecture neutral, meaning that they capture key architectural characteristics such as instruction mix, branch prediction, and register dependencies, but do so independently of the underlying micro-architecture. We gather these metrics using a modified version of Simics v. 3.0.11 [42]. We use our harness to measure stable performance in Sun’s HotSpot JVM, v. 1.5.0_07-b03 running on a simulated Sun Ultra-5,10 with Solaris 9.

PCA computes four principal components (PC1, PC2, PC3, and PC4) which in our case account for 70% of the variance between benchmarks. PCA identifies principal components in order of significance; PC1 is the most determinative component and PC4 is the least. Table 5 shows the relative ranks of each of the metrics for PC1–PC4. The absolute value of the numbers in columns 2–5 indicates the rank significance of the metric, while the sign indicates whether the contribution is negative or positive. We bold the ten most significant contributions to PC1 cover each of the three metric categories.

Scatter plots in Figure 3 show how the benchmarks differ in two-dimensional space. Figure 3 plots each program’s PC1 value against its PC2 value in the top graph, and Figure 3 plots PC3 and PC4 in the bottom graph. Intuitively, the further the distance between two benchmarks, the further apart they are with respect to the metrics. The benchmarks differ if they are apart in either graph. Since the programs are well distributed in these graphs, the benchmarks differ.

10. Conclusion

Benchmarks play a strategic role in computer science research and development by creating a common ground for evaluating ideas and products. The choice of benchmarks and benchmarking methodology can therefore have a significant impact on a research field, potentially accelerating, retarding, or misdirecting energy and innovation. Prompted by concerns among ourselves and others about the state-of-the-art, we spent thousands of hours at eight separate institutions examining and addressing the problems of benchmarking Java applications. The magnitude of the effort surely explains why so few have developed benchmark suites.

This paper makes two main contributions: 1) it describes a range of methodologies for evaluating Java, including a number of new analyses, and 2) it presents the DaCapo benchmark suite. We show that good methodology is essential to drawing meaningful conclusions and highlight inadequacies prevalent in current methodology. A few of our specific methodology recommendations are:

- When selecting benchmarks for a suite, use PCA to quantify benchmark differences with metrics that include static and dynamic code and data behavior.
- When evaluating architectures use multiple JVMs. Evaluating new architecture features will also benefit from multiple JVMs.
- When evaluating JVM performance, use multiple architectures with mix and stable methodologies, reporting first and/or second iterations as well as steady-state to explore the compile and runtime tradeoffs in the JVM.
- When measuring memory performance, use and report heap sizes proportional to the minimums.
- When measuring GC and JIT compilation performance use mix and stable methodologies, and use constant workload (rather than throughput) benchmarks.
- When measuring GC or compile time overheads use deterministic stable and mix methodologies.

This paper uses these methodologies to demonstrate that the DaCapo benchmarks are larger, more complex and richer than the commonly used SPEC Java benchmarks. The DaCapo benchmarks are publicly available, evolving, and have and will remain open to public feedback [14].

<table>
<thead>
<tr>
<th>Metric</th>
<th>Architecture Rank</th>
<th>Code Rank</th>
<th>Memory Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction mix – ALU</td>
<td>-9</td>
<td>-15</td>
<td>7</td>
</tr>
<tr>
<td>Instruction mix – branches</td>
<td>-10</td>
<td>-4</td>
<td>16</td>
</tr>
<tr>
<td>Instruction mix – memory</td>
<td>1</td>
<td>13</td>
<td>-11</td>
</tr>
<tr>
<td>Branch mispred/instruction for a PPM predictor</td>
<td>8</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Register dependence distance up to 16</td>
<td>-5</td>
<td>-12</td>
<td>-5</td>
</tr>
<tr>
<td>Register dependence distance between 16 and 64</td>
<td>-15</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>Register dependence distance above 64</td>
<td>15</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Instruction cache misses in misses/msec</td>
<td>-14</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>Bytecode compiled in KB</td>
<td>-7</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Methods compiled</td>
<td>-2</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5. Metrics Used for PC Analysis and Their PC Rankings.
Acknowledgments

We thank Andrew Appel, Randy Chow, Frans Kaashoek, and Bill Pugh who encouraged this project at our three year ITR review. We thank Mark Wegman who initiated the public availability of Jikes RVM, and the developers of Jikes RVM. Fahad Gilani wrote the original version of the measurement infrastructure for his ANU Masters Thesis.

References


Benchmark Description and Origin

Short Description A parser generator and translator generator
Long Description ANTLR parses one or more grammar files and generate a parser and lexical analyzer for each.
Threads Single threaded
Repeats Two iterations, each parses 44 distinct grammar files
Version 2.7.2
Copyright Public Domain
Author Terence Parr
License Public Domain

Benchmark Characteristics

Total Allocation (MB) 237.9
(Obj) 4,208,403
Maximum Live (MB) 1.0
(Obj) 15,566
Pointer Mutations (M) 3.91
Classes Loaded 126

Figure 4. Benchmark Characteristics: antlr
Benchmark Description and Origin

Short Description A Bytecode-level optimization and analysis tool for Java
Long Description BLOAT analyzes and optimizes some of its own class files
Threads Single threaded
Repeats Single iteration, transitively optimizes classes referenced by a single root class
Version 1.0
Copyright Copyright (c) 1997-2001 Purdue Research Foundation of Purdue University
Author Nathaniel Nystrom and David Whitlock
License BSD-style

Benchmark Characteristics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Allocation (MB)</td>
<td>1,222.5</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>6.2</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>257.84</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>281</td>
</tr>
</tbody>
</table>

Figure 5. Benchmark Characteristics: bloat
### Benchmark Description and Origin

<table>
<thead>
<tr>
<th>Short Description</th>
<th>A graph plotting toolkit and pdf renderer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Description</td>
<td>JFreeChart plots a number of complex line graphs and renders them as pdf via JText</td>
</tr>
<tr>
<td>Threads</td>
<td>Single threaded</td>
</tr>
<tr>
<td>Repeats</td>
<td>Single iteration plots 14 distinct graphs</td>
</tr>
<tr>
<td>Version</td>
<td>0.9.21, 1.0b</td>
</tr>
<tr>
<td>Copyright</td>
<td>(C)opyright 2000-2004, by Object Refinery Limited and Contributors; Copyright 2000, 2001, 2002 by Bruno Lowagie</td>
</tr>
<tr>
<td>Author</td>
<td>David Gilbert, Bruno Lowagie and Paulo Soares</td>
</tr>
<tr>
<td>License</td>
<td>LGPL and MPL</td>
</tr>
</tbody>
</table>

### Benchmark Characteristics

- **Total Allocation (MB):** 742.8
- **Maximum Live (MB):** 9.5
- **Pointer Mutations (M):** 19.64
- **Classes Loaded:** 219

![Figure 6. Benchmark Characteristics: chart](chart)

(a) Allocated (above) and Live (below) Object Size Histograms and Time-series

(b) Heap Composition Time-series, in Allocations (above) and Mutations (below)
**Benchmark Description and Origin**

<table>
<thead>
<tr>
<th>Short Description</th>
<th>An integrated development environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Description</td>
<td>Run a series of eclipse jdt (non-gui) performance tests</td>
</tr>
<tr>
<td>Threads</td>
<td>Workload is single threaded, but Eclipse uses multiple threads internally</td>
</tr>
<tr>
<td>Repeats</td>
<td>Single iteration, performs multiple distinct Eclipse tasks</td>
</tr>
<tr>
<td>Version</td>
<td>3.1.2</td>
</tr>
<tr>
<td>Copyright</td>
<td>Eclipse Foundation</td>
</tr>
<tr>
<td>Author</td>
<td>Eclipse Foundation</td>
</tr>
<tr>
<td>License</td>
<td>Eclipse Public License</td>
</tr>
</tbody>
</table>

**Benchmark Characteristics**

<table>
<thead>
<tr>
<th>Total Allocation (MB)</th>
<th>5,582.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Obj)</td>
<td>104,162,353</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>30.0</td>
</tr>
<tr>
<td>(Obj)</td>
<td>470,333</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>335.49</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>795</td>
</tr>
</tbody>
</table>

![Figure 7](image)
**Benchmark Description and Origin**

**Short Description**
An output-independent print formatter

**Long Description**
fop takes an XSL-FO file, parses it and formats it, generating an encrypted pdf file.

**Threads**
Single threaded

**Repeats**
Single iteration, renders a single XSL-FO file

**Version**
0.20.5

**Copyright**
Copyright (C) 1999-2003 The Apache Software Foundation

**Author**
Apache Software Foundation

**License**
Apache Public License

---

**Benchmark Characteristics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Allocation (MB)</td>
<td>100.3</td>
</tr>
<tr>
<td>(Obj)</td>
<td>2,402,403</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>6.9</td>
</tr>
<tr>
<td>(Obj)</td>
<td>177,718</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>3.85</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>231</td>
</tr>
</tbody>
</table>

---

**Figure 8.** Benchmark Characteristics: fop
**Benchmark Description and Origin**

**Short Description**
An SQL relational database engine written in Java

**Long Description**
HSQLDB executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application.

- **Threads**: 20 client threads
- **Repeats**: 40 transactions per client
- **Version**: 1.8.0.4
- **Copyright**: Copyright (c) 2001-2002, The HSQL Development Group
- **Author**: The HSQLDB Development Group
- **License**: The HSQLDB license.

**Benchmark Characteristics**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Allocation (MB)</td>
<td>142.7</td>
</tr>
<tr>
<td>(Obj)</td>
<td>4,514,965</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>72.0</td>
</tr>
<tr>
<td>(Obj)</td>
<td>3,223,276</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>19.31</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>131</td>
</tr>
</tbody>
</table>

---

**Figure 9. Benchmark Characteristics: hsqldb**

(a) Allocated (above) and Live (below) Object Size Histograms and Time-series

(b) Heap Composition Time-series, in Allocations (above) and Mutations (below)

(c) Snapshot (above) and Mutation (below) Pointer Distance Histograms and Time-series
**Benchmark Description and Origin**

- **Short Description**: A python interpreter written in Java
- **Long Description**: jython executes (interprets) the pybench benchmark or a small python program
- **Threads**: Single threaded
- **Repeats**: Single iteration runs a single iteration of the pybench python benchmark
- **Version**: 2.1
- **Copyright**: Copyright (c) Python Software Foundation
- **Author**: Jim Hugunin and Barry Warsaw
- **License**: Jython Software License.

**Benchmark Characteristics**

- Total Allocation (MB): 1,183.4
- (Obj): 25,940,819
- Maximum Live (MB): 0.1
- (Obj): 2,788
- Pointer Mutations (M): 82.96
- Classes Loaded: 251

---

**Figure 10. Benchmark Characteristics: jython**

(a) Allocated (above) and Live (below) Object Size Histograms and Time-series

(b) Heap Composition Time-series, in Allocations (above) and Mutations (below)

(c) Snapshot (above) and Mutation (below) Pointer Distance Histograms and Time-series
Benchmark Description and Origin

**Short Description**: A text indexing tool

**Long Description**: Indexes a set of documents, the works of Shakespeare and the King James Bible

**Threads**: Single threaded

**Repeats**: Single iteration indexes two multi-file documents

**Version**: 1.9.1

**Copyright**: Copyright (C) The Apache Software Foundation

**Author**: Lucene Project Management Committee

**License**: Apache Public License

Benchmark Characteristics

<table>
<thead>
<tr>
<th>Total Allocation (MB)</th>
<th>201.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Obj)</td>
<td>7,202,623</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>1.0</td>
</tr>
<tr>
<td>(Obj)</td>
<td>18,566</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>21.75</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>128</td>
</tr>
</tbody>
</table>

(a) Allocated (above) and Live (below) Object Size Histograms and Time-series

(b) Heap Composition Time-series, in Allocations (above) and Mutations (below)

(c) Snapshot (above) and Mutation (below) Pointer Distance Histograms and Time-series

Figure 11. Benchmark Characteristics: luindex
**Benchmark Description and Origin**

Short Description: A text search tool
Long Description: Text search of keywords over a corpus of data comprising the works of Shakespeare and the King James bible
Threads: 32 threads
Repeats: Each thread searches a large index for about 3500 distinct words
Version: 1.9.1
Copyright: Apache Software Foundation, Apache license v2.0
Author: Lucene Project Management Committee
License: Apache Public License

**Benchmark Characteristics**

<table>
<thead>
<tr>
<th>Total Allocation (MB)</th>
<th>1,780.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Obj)</td>
<td>15,780,651</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>10.9</td>
</tr>
<tr>
<td>(Obj)</td>
<td>34,792</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>64.77</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>118</td>
</tr>
</tbody>
</table>

---

**Figure 12.** Benchmark Characteristics: lusearch
Benchmark Description and Origin

Short Description  A source code analyzer for Java
Long Description  pmd analyzes a list of Java classes for a
range of source code problems
Threads         Single threaded
Repeats         Single iteration checks a single large
source file against 18 coding rules
Version         1.8
Copyright       Copyright (c) 2003, InfoEther, LLC
Author          Tom Copeland
License         BSD-style

Benchmark Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Allocation (MB)</td>
<td>779.7</td>
</tr>
<tr>
<td>(Obj)</td>
<td>34,137,722</td>
</tr>
<tr>
<td>Maximum Live (MB)</td>
<td>13.7</td>
</tr>
<tr>
<td>(Obj)</td>
<td>419,789</td>
</tr>
<tr>
<td>Pointer Mutations (M)</td>
<td>105.52</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>325</td>
</tr>
</tbody>
</table>

Figure 13. Benchmark Characteristics: pmd
**Benchmark Description and Origin**

**Short Description**
An XSLT processor for transforming XML documents.

**Long Description**
Xalan transforms an XML document (either a test case or the works of Shakespeare) and transforms the document into html.

**Threads**
Single threaded

**Repeats**
8 iterations, each transforms a single large XML document (the works of Shakespeare)

**Version**
2.4.1

**Copyright**
Copyright (C) 1999-2003 The Apache Software Foundation

**Author**
Apache Software Foundation

**License**
Apache Software License

**Benchmark Characteristics**

- Total Allocation (MB): 60,235.6
- Maximum Live (MB): 25.5
- Pointer Mutations (M): 278.20
- Classes Loaded: 244

---

**Figure 14. Benchmark Characteristics:** xalan

(a) Allocated (above) and Live (below) Object Size Histograms and Time-series

(b) Heap Composition Time-series, in Allocations (above) and Mutations (below)

(c) Snapshot (above) and Mutation (below) Pointer Distance Histograms and Time-series