

O Java, Java! Wherefore Art Thou Java?

Kathryn S McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Stephen M Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Abstract

The experimental systems community relies on benchmarks to evaluate proposed innovations. Therefore, the choice of benchmarks and the programming language in which they are written is a gating function for innovation in our field. While application developers are increasingly embracing managed languages such as Java and C#, systems researchers have not. This disconnect between researchers and application developers may lead to misdirected research. This problem is partially due to a dearth of appropriate benchmarks and appropriate evaluation methodologies. This paper demonstrates and recommends how to perform meaningful experiments with managed languages and contrasts them with methodologies in use for C or C++. The paper also recommends some approaches for encouraging community benchmark and infrastructure development. To further inform systems research, it may behoove some researchers to follow language implementers who “eat their own dog food”, and build systems in managed languages.

I. Introduction

The research community relies on benchmarks to evaluate innovative ideas for improving performance, correctness, reliability, and error detection. Furthermore, workload analysis on existing stems reveals which problems are important to solve at all. The systems community has long embraced this approach. For example, in their classic text, Hennessy and Patterson analyzed workloads on the PDP-11 to motivate and evaluate RISC [11]. As we seek to improve future systems, ideally, benchmarks would capture the workloads of the future, but at the very least, they should represent general or specialized workloads that we care about *now*.

Both researchers and product developers widely use benchmarks from SPEC [18] and the TCP [22] database workloads. These programs are offered with the goal “to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers” [18], i.e., a standardized measure of real, extant, systems. Although researchers use these benchmarks to evaluate and analyze their innovations, they were not designed for this purpose and we should examine whether they are appropriate for designing next generation systems. Yi et al. do exactly that for the SPEC CPU benchmarks [23]. They demonstrate that successive benchmark generations have substantially different bottlenecks, and consequently lead to different ar-

chitecture designs. For example, using SPEC 95 and SPEC 2000 on 2000 era architecture design choices yields different conclusions. They recommend updating benchmark suites and compilers frequently and parameterizing benchmarks to try to project future characteristics. We believe an even worse problem is occurring because architects are not using managed language workloads.

A major trend in software development is the increasing use of *managed* languages such as Java and C#. Managed languages include features such as (1) pointer encapsulation which prevents inadvertent or malicious memory corruption; (2) static and dynamic, type and semantic checking which prevents a class of programming errors; and (3) garbage collection which prevents a class of memory errors and eliminates the need to explicitly program and therefore reason about memory management. These and other features help programmers get their applications working faster. Other features such as dynamic loading help programmers target their applications to dynamic domains such as the web. A recent Gartner report estimates that 80% of new software development will be in Java or C# [10]. For the remainder of this article, we will focus on Java as a representative managed language, but note that systems researchers and developers, and Microsoft, would be well served if there were freely available C# benchmarks. The need for good workloads extends beyond these particular languages, but, the absence of Java and C# among system researchers’ workloads is in contrast to these languages’ growing dominance among real world applications.

Driven by this surging demand, programming language researchers and developers have turned their attention to virtual machine technology such as dynamic compilation and memory management for Java. For example, the most recent ACM SIGPLAN Programming Language Design and Implementation (PLDI 2006) conference included 9 papers out of 36 on technologies for Java, and 5 on automated analysis and transformation of C and C++ programs to provide some of the reliability features found in managed languages. The next logical step in the systems research continuum is to use these as workloads for evaluating and driving future innovations in operating systems and architectures. Yet, the use of managed workloads is not yet evident in architecture research.

A quick survey of the papers in ACM SIGARCH International Symposium on Computer Architecture (ISCA 2006) found the following workloads:

- 20 papers use C and/or C++ applications from SPEC, SPLASH, TPC, MediaBench, and other sources (e.g., a binary translator and industrial web servers).
- 5 papers are orthogonal to the programming language.
- 2 papers use specialized programming languages.

This work is supported by NSF CCR-0311829, NSF CCF-CCR-0311829, NSF CISE infrastructure grant EIA-0303609, ARC DP0452011, DARPA F33615-03-C-4106, DARPA NBCH30390004, IBM, Microsoft, and Intel. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

- 2 papers use Java and C from SPEC.
- 1 paper uses just Java from SPEC.

In this paper, we exhort systems researchers to turn their attention to Java and other workloads in order to make our work more relevant now and in the future, and to participate in developing, maintaining, and expanding the benchmarks and methodologies we all use. We discuss the challenges of measuring managed languages, point out new methodologies researchers can use to make their experiments more meaningful, and discuss existing impediments to participation in benchmarking and general infrastructure building.

Workloads are therefore key to both illuminating systems problems and solutions. In a recent paper, we presented a new benchmark suite for Java called the DaCapo Benchmark Suite [4], [5], named for our research group [21]. The DaCapo benchmarks consist of eleven freely available Java workloads taken from complex, large, and widely used applications. We showed in that paper that the DaCapo benchmarks offered larger, more complex, and more realistic workloads than the dated SPEC Java Benchmarks. We also discussed benchmarking methodologies and methodological issues specific to evaluating managed languages.

Here we focus on the challenges managed languages and benchmarking present to the systems community. We highlight the problem of outmoded and inappropriate benchmarks. We compare benchmarking methodologies for C and C++ to Java, and show the pitfalls, and then make recommendations for how to benchmark Java. We include an evaluation of multiple JVMs (runtimes) on multiple architectures that illustrate the subtleties of measuring managed runtimes. We repeat some prior recommendations, and add some specific ones for architectural research and machine evaluation. For example, we discuss how to control and understand dynamic hotspot recompilation and garbage collection. We show how to control for these factors to produce meaningful comparisons.

Social factors contribute to the current state of the field. We believe that one of the reasons people have not been more critical of the state of benchmarking and participated in improving the state of affairs is lack of funding and publication opportunities. One reason is that agencies and reviewers find it much more appealing to fund a single new idea rather than to fund infrastructure—which might eventually be the basis for many new ideas. We argue that it is not possible to sustain real innovation without substantial, ongoing, investments in benchmarking and infrastructure. As experimental computer scientists, our tools are systems, and if we do not create and maintain useful infrastructure, it becomes very difficult to create and evaluate meaningful novel ideas.

We hope the research community can develop more rewards for key systems infrastructure such as benchmarks, simulators, operating systems, virtual machines, and compilers. One mechanism could be yearly SIG level awards for public systems software. These awards would also provide fodder for the ACM Software System Award. We believe changes like this will help focus the community on meaningful problems and innovative solutions.

II. Methodology Challenges

The next three sections discuss and illustrate the challenges to meaningful and accurate measurements of managed languages. Section V then makes specific recommendations for addressing these challenges.

Due to dynamic loading, dynamic compilation, and garbage collection, the methodologies for evaluating managed languages must differ from those used for traditional imperative languages such as C, C++, and Fortran which use ahead-of-time compilation. For example in managed languages, dynamic compilation triggered by hotspot detection must be controlled as a source of non-determinism and its cost must be measured, as we show in Section IV. The methodologies for managed languages are not well established in the benchmarking culture. However, without well defined methodologies, performance measurements are wide open to accidental or deliberate distortions. Distorted analysis and results can mislead researchers to waste time solving the wrong problems.

SPEC introduced Java benchmarks [20], [19] in 1998. The SPEC jvm98 harness reports the maximum memory usage during each run and uses this to categorize reported results as requiring small (≤ 48 MB), medium (48 to 256 MB), or large (> 256 MB) heaps. When producing publishable results, all benchmarks must run in a single invocation of the JVM. Since the benchmarks vary greatly in their amount of allocation (e.g., mpegaudio allocates 0.7 MB with at most 0.6 MB live at any point; jack allocates 270.7MB with 0.9 MB live [4]), using a single heap metric for all benchmarks ensures that some benchmarks are likely to be significantly over-accommodated, and probably perform no garbage collection. Given the heap requirements of these benchmarks and their memory allocation and live size, even a “small” heap need perform very few collections. Thus, together the reporting and configuration requirements can eliminate garbage collection influences, leaving a defining aspect of the Java language unevaluated.

III. C++ Workloads are Not a Substitute for Java

Although C++ has grown in popularity as well as Java and C#, C++ workloads are unsuitable as a substitute for Java. Systems developers are using C++ as an alternative to C to attain some of the advantages of object-oriented programming while retaining many of the efficiencies of C such as fixed contiguous array memory layouts. However, C++ workloads do not bring to bear the complexities of dynamic loading, dynamic compilation, runtime checks, or garbage collection, which as we show in the following section, influence runtime performance dramatically. Unfortunately, C++ does bring to bear all the limitations and problems of the C type system and pointer model; these features limit compiler optimization opportunities and the opportunity to improve locality by reorganizing heap data and code. These differences make for fundamentally different compiler and runtime workloads. So while C++ differs from C in name, “a rose by any other name would smell as sweet”, or in this case, perhaps not so sweet.

IV. Example Variance

This section shows an example of how architecture, JVM, and dynamic compilation decisions influence performance measurements to illustrate the need for deterministic methodologies for measuring architecture, JVM, and compiler innovations. In addition, this section shows that the *particular iteration* of the benchmark may invert results, and thus that these comparisons need to be made carefully on the same architecture and across architectures. Prior work shows the influence of heap size and architecture for measuring garbage collector innovations [3], [4] and that the choice of JVM and benchmark iteration matter on the same architecture [4].

It is well known that JVMs are often well tuned for a particular architecture and less tuned for others. Thus, the best total performance for a given Java program or benchmark suite is typically a function of both the architecture and the JVM.

The JVMs we used (and most other JVMs) have multiple levels of compilation. When the JVM first encounters a method, it loads the class and either commences interpreting it, or immediately compiles it into machine code using its quickest, most basic compiler, i.e., typically without much optimization. The JVM then performs *hotspot* JIT (re-)compilation on demand for methods that execute frequently. The JVM detects method frequency with performance counters or software instrumentation. When the frequency crosses some threshold, the JVM uses more expensive, and hopefully more effective optimizations. Once a method reaches its highest level of optimization (e.g., after the JVM compiles the method two to four times), the JVM usually ceases recompilation. However, the JVM may re-optimize a method if it detects some assumptions it made are no longer appropriate or have become invalid. For example, an aggressive JIT may speculatively specialize compilation for an observed indirect call, but may need to recompile if the underlying assumption no longer holds, e.g., due to subsequent class loading and compilation.

We now illustrate some of these points with experiments using the following three architectures: a) a 2.2GHz AMD Athlon64 with 2GB of RAM and a 512KB L2 cache, running Linux 2.6.15, b) a 2GHz Intel Pentium M with 1GB of RAM and a 2MB L2 cache, running Linux 2.6.15, and c) a 12-way SunFire V1280, with 12 × 900MHz UltraSparcIIICu, 24 GB RAM and 8 MB off-chip L2, running Solaris 10 3/05.

For these three architectures, we found two widely used commercial JVMs that executed out of the box. We anonymize the JVMs since their identities are not relevant to the results. We execute them with their default settings, which means that in these results, the time-space tradeoff associated with memory management is obscured. Blackburn et al. show that heap size matters [3]. Unlike SPECjvm98, DaCapo programs allocate too much to readily eliminate collection [4], so in the results presented here, the JVMs will exercise their collectors but for brevity we do not explore the associated tradeoffs here.

We execute multiple iterations of the DaCapo Benchmarks (v. dacapo-2006-10-MR1) on each JVM and architecture pair. We measure the first, second, and third iteration of each benchmark running in the same invocation of the virtual machine. Due to the targeted recompilation mechanism described above, we expect the first iteration to be slower. We conduct each

experiment five times and report the fastest result from that set.

The three graphs in Figure 1 plot the performance of the DaCapo Benchmarks for virtual machine “A” and “B” for AMD, Pentium M, and Sparc platforms. We show results for each benchmark, the minimum, max, and geometric mean. In each case we plot six bars, corresponding to three iterations on each of the two JVMs. Each result is normalized with respect to the fastest of the six results for that benchmark (so each cluster has one or more result at 1.0, and none lower than 1.0). We first show results for JVM “A”, each iteration shaded differently. We then show the three iterations for JVM “B”.

The geometric mean results in Figure 1(a) show that on the AMD, JVM “B” is, on average, significantly faster than JVM “A”, and that the performance of JVM “B” gradually improves across each of the three iterations. The performance of JVM “A” also improves, but not as quickly. Figure 1(b) shows a similar story for the Pentium M. However in Figure 1(c) the trend is reversed, and JVM “A” significantly outperforms JVM “B”. In each case, we see gradual improvement across the three iterations. The reversal in Figure 1(c) highlights the sensitivity to architecture and JVM. Although both JVMs are very popular production JVMs they show markedly different performance characteristics on the Sparc compared to the x86-based machines.

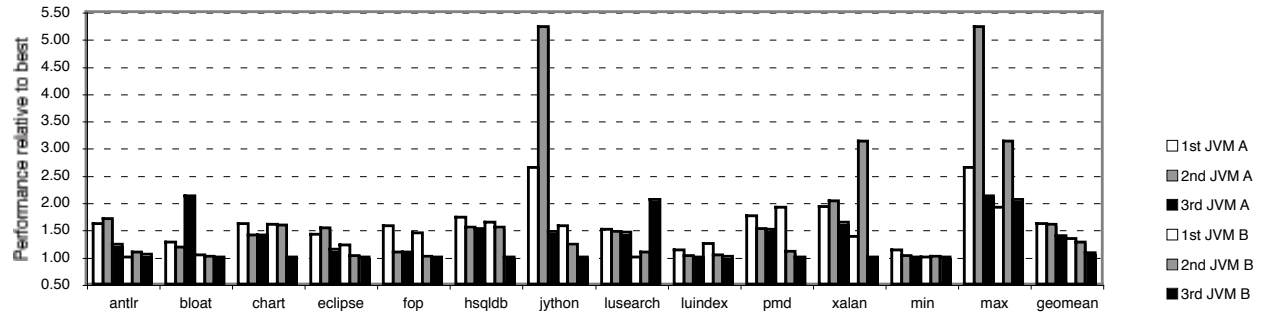
The graphs also show a number of more subtle and surprising results. For example, on the AMD and Pentium M, JVM “A” suffers a two-fold performance degradation from first (white) to second (gray) iteration when executing *lython*. This result is repeatable—recall that each result here is the best of five trials. Note that in the third (black) iteration, JVM “A” performs very well, which suggests that JVM “A” performs expensive optimizations during hotspot recompilation in the second iteration. We investigated further and suspect that one of *lython*’s floating point tests was the source of much of this slowdown. We do not see the same trend on the Sparc, which highlights the architectural sensitivity of such comparisons. JVM “B” suffers a similar anomaly on the third iteration of the *lusearch* benchmark.

V. Suggested Evaluation Methodologies

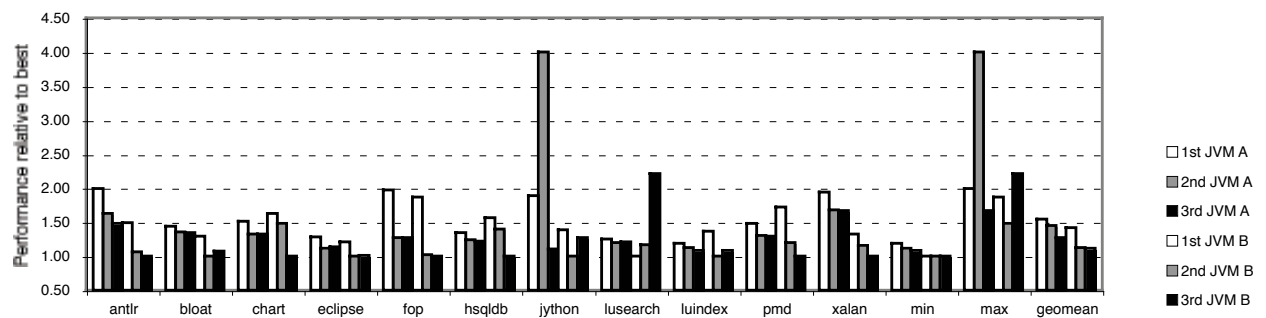
This section recommends methodologies for controlling and reporting results for a variety of architecture, operating system, and compiler experiments.

Throughput versus time.

Two key performance consideration for systems research are execution time and throughput. Throughput measurements can tolerate non-determinism, but total time measurements require deterministic application behavior, or controlled differences, to make meaningful comparisons. For example, given a new branch predictor, we want to fix the workload to measure whether it improves performance. If the workload varies non-deterministically, an improvement may or may not be a result from the architecture change, but rather its influence on timing and recompilation which are orthogonal to the branch predictor. However, even throughput experiments may benefit in some cases from fixing the JVM and JIT as sources of



(a) AMD



(c) Sparc

Fig. 1. Comparison of Two Production JVMs Across Three Architectures, Showing Results For Three Iterations.

non-determinism.

Eliminating non-determinism.

To eliminate non-determinism for architecture, operating system, and compiler execution time measurements and simulations, we recommend replay compilation [17], [13], [16]. Replay compilation has been used successfully to eliminate non-determinism for experiments on garbage collection, compiler optimizations, and compiler debugging. In replay compilation, a profiling run records method compilation decisions, e.g., the level of compilation, and other information such as the branch profile and the call graph profile needed to produce the same version of the JITed method. Then a measurement run ignores the sampling recompilation triggers it normally uses (e.g., the number samples in some method exceeding a threshold that triggers method recompilation). Instead, the replay execution

reads in the profile, and produces the code by applying the compiler as specified by the profile when the method is first executed.

This system must use self-profiling on the same input and thus cannot be adapted when applications have other sources of non-determinism. For example, it is not appropriate for interactive applications. This methodology also eliminates dynamic class loading non-determinism, since it specifies all methods. This functionality is available in Jikes RVM, a freely available Java-in-Java virtual machine [1], and can be adapted into other JVMs [16]. The replay methodology can be configured in a number of ways to produce a mix of unoptimized and optimized code, all optimized, compiler and application, just application, etc. We discuss why and how to create these versions below.

Application quality.

As we pointed out above, modern JVMs first translate to machine code with a non-optimizing compiler or interpreter, and then apply heavier weight optimizations one or more times to hot methods. Therefore, the JVM produces three application loads of interest:

- base: all methods use the non-optimizing compiler or interpreter only.
- hot: a mix of base cold code and optimized compiled hot code based on frequency.
- opt: all methods use the optimizing compiler at its most aggressive level.

The replay methodology makes it possible to create any of these versions with its facility for specifying the optimization level for every method.

The expected execution environment will motivate the choice of code quality. For example, an embedded system may only ever have the space or time to use an interpreter or base compilation. JVMs can be configured to produce all optimized code for long running applications. Client side users will often see optimizations on only the hot code.

How much compilation?

As we showed in the previous section, the compilation load varies between iterations of a benchmark and this load effects total execution time measurably. This feature presents two methodological concerns: does an architecture or operating system optimization improve the JIT system in the JVM (i.e., one important application) or does it improve the Java applications that it executes? Eeckhout et al. show that the JVM can sometimes obscure the application completely [9]. To tease apart the effects, we need measurements that selectively exclude and include the JIT compiler.

- Mix: includes compilation time at one of the levels above, and interleaves JIT compilation and application work.
- Stable: includes only application time with one of the compilation choices above.

Measurements of the first iteration of a benchmark typically capture mix (although JVMs must be configured appropriately). To find a stable application only measurement, one must use a harness that runs the benchmark numerous times until it converges on a stable, *best* execution time, or combine an opt compiler configuration with a replay compiler. By coupling these methodologies with replay compilation to force known deterministic choices out of the compiler, researchers can control and specify the JIT as appropriate for the target execution setting.

Garbage collection and the memory system.

When analyzing architecture or operating system effects on memory systems, researchers must consider its interactions with the garbage collector. To include garbage collection effects in experiments, we recommend heap sizes specific to each benchmark that are proportional to its maximum live size (e.g., two times maximum live size), perhaps a range of heap sizes, and a basic high performance collector, such

as generational with a copying nursery [4]. A heap sized proportional to the live data ensures that some collector effects will be captured. For example, collectors that move objects will do so, and collectors that must interleave objects will do so. If these have positive or negative interactions with, for example, a TLB optimization, these effects will be more likely to be exercised. A copying nursery for newly allocated objects provides excellent locality to programs with high rates of allocation [3]. We refer you to prior work that discusses the subtle and significant interactions of memory system performance and garbage collection in more detail [3], [12], and note in closing that the choice of collector should be customized for target settings such as real-time or embedded systems.

VI. Benchmarking Challenges

Even if we use and specify the appropriate measurement methodologies, a key component of methodology is using appropriate benchmarks. Although the research community continually seeks new benchmarks, there is little funding or community support for maintaining and developing benchmarks. For example, DaCapo benchmark development was recommended at a third year review of a five year grant by an NSF review panel, but it was not funded as a distinct effort. Previous efforts, such as MediaBench [14] introduced a set of media applications. Even though these types of applications are continuing to grow in importance, no recent or updated versions of MediaBench have appeared, highlighting the effort it requires and the lack of rewards in these efforts.

The research community is moving into embedded systems, multicore, transactional memory and heterogeneous processing, but many of these key areas lack benchmarks. Although researchers are proposing new approaches, we still need benchmarks. For example, Berkeley researchers have recently recommended using *dwarfs* for parallel computing that capture a pattern of computation and communication and abstract away from a specific programming language or compilation model [2]. This methodology is designed to explore how to use the programming language to express parallelism, as well as the operating system and architectures that underly these parallel systems with 100s to 1000s of processors. This model tries to skirt both dynamic and traditional compilation, whereas we are recommending methodologies that make dynamic compilation understandable.

VII. Social Challenges

In order for more researchers to spend time on benchmarking and other infrastructure development, there need to be more rewards. The research community for the most part measures value with publications in top research venues, e.g., a paper in ASPLOS, ISCA, OSDI, or PLDI. The papers in these venues present an innovation, often evaluated within the context of an artifact (e.g., compiler, operating systems, and/or simulator). An anecdote in this area is architectural simulators. For example, SimpleScalar is very widely used (100s to perhaps 1000s of publications), but there are three technical reports and one conference paper on the artifact itself [6], [7], [12], [8]. Although a vast majority of architecture researchers use

simulators, there are only a handful of papers that address how to build one [15], and how to validate them [8]. To increase the value to investigators of participating in such efforts, the research and funding community must show they also value creating and sustaining the artifact itself by funding these efforts and creating a venue or space in current high impact venues for papers on these topics. For example, introducing awards for contributions to research infrastructure, or starting a bi-yearly issue on artifacts, rather than one special issue every 10 to 20 years [15]. To change this value system will require a shift in the research community.

VIII. Conclusion

This paper exhorts the community to examine their methodologies and participate in benchmark generation and selection. This request is timely due to the recent shift to managed languages, and the eminent rise of multicore hardware. For example, the community is in particular need of large realistic benchmarks with diverse levels and ways of expressing parallelism, such as locks and transactions. For computer systems researchers and developers to continue to provide increases in performance, their innovations need to derive from current and future workloads. In addition, experiments need to control and explore the interactions workloads such as managed languages present. If we participate in selecting and dissecting workloads, our solutions are more likely to be relevant in the future. If we participate continually in benchmark creation and evaluation, we benefit ourselves, the research community, and users of computer systems. The quality of your workload gates your innovation.

Acknowledgments

We thank the DaCapo Research Group, and in particular, Amer Diwan, Antony Hosking, Eliot Moss, and Darko Stefanovic. Thanks to Mike Bond for suggesting infrastructure prizes. We are grateful to all the developers of Jikes RVM in which we performed many of these experiments. We particularly thank Robin Garner who was instrumental in the development of the DaCapo benchmark suite. Meaningful innovation requires substantial and high performance infrastructure, and we have benefited from and contributed to Jikes RVM for this reason.

References

- [1] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. Flynn Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.
- [2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.
- [3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.
- [4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, October 2006.
- [5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, Dept. of Computer Science, Australian National University, 2006. <http://www.dacapobench.org>.
- [6] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.
- [7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.
- [8] R. Desikan, Doug Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *International Symposium on Computer Architecture*, pages 266–277, June 2001.
- [9] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitecture level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–186, Anaheim, CA, October 2003.
- [10] J. Fenn and A. Lindon. Gartner’s hype cycle special report. Technical report, 2004. <http://www.gartner.com/>.
- [11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.
- [12] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic simplescalar: Simulating java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences, February 2003.
- [13] X. Huang, Z. Wang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, and P. Cheng. The garbage collection advantage: Improving mutator locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, Vancouver, BC, 2004.
- [14] C. Lee, M. Potkousjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. In *International Symposium on Microarchitecture*, pages 330–335, December 1997.
- [15] S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools, guest editors introduction. *IEEE Computer*, 29(12):38–39, February 2002.
- [16] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: improving debuggability of a just-in-time compiler. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–252, October 2006.
- [17] N. Sachindran and J.E.B. Moss. Mark-copy: Fast coping with less space overhead. In *ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 326–343, October 2003.
- [18] Standard Performance Evaluation Corp. SPEC Benchmarks, 2007. <http://www.spec.org>.
- [19] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
- [20] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.
- [21] The DaCapo Research Group, 2007. <http://www.dacapogroup.org>.
- [22] Transaction Processing Performance Council. TCP Benchmarks, 2007. <http://www.tpc.org>.
- [23] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: Designing tomorrow’s processors with yesterday’s tools. In *Proceedings of the 2006 ACM International Conference on Supercomputing*, pages 75–86, Cairns, Queensland, Australia, July 2006.