Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century

Stephen M Blackburn[α], Kathryn S McKinley[β], Robin Garner[α], Chris Hoffmann[γ], Asjad M Khan[γ], Rotem Bentzur[δ], Amer Diwan[ε], Daniel Feinberg[δ], Daniel Frampton[α], Samuel Z Guyer[ζ], Martin Hirzel[η], Antony Hosking[θ], Maria Jump[β], Han Lee[ι], J Eliot B Moss[γ], Aashish Phansalkar[β], Darko Stefanović[δ], Thomas VanDrunen[κ], Daniel von Dincklage[ε], Ben Wiedermann[β]

[α]Australian National University, [β]University of Texas at Austin, [γ]University of Massachusetts at Amherst, [δ]University of New Mexico, [ε]University of Colorado, [ζ]Tufts University, [η]IBM, [θ]Purdue University, [ι]Intel, [κ]Wheaton College

Abstract

Evaluation methodology underpins all innovation in experimental computer science. It requires relevant *workloads*, appropriate *experimental design*, and rigorous *analysis*. Unfortunately, methodology is not keeping pace with the changes in our field. The rise of managed languages such as Java, C#, and Ruby in the past decade and the imminent rise of commodity multicore architectures for the next decade pose new methodological challenges that are not yet widely understood. This paper explores the consequences of our collective inattention to methodology on innovation, makes recommendations for addressing this problem in one domain, and provides guidelines for other domains. We describe benchmark suite design, experimental design, and analysis for evaluating Java applications. For example, we introduce new criteria for measuring and selecting diverse applications for a benchmark suite. We show that the complexity and nondeterminism of the Java runtime system make experimental design a first-order consideration, and we recommend mechanisms for addressing complexity and nondeterminism. Drawing on these results, we suggest how to adapt methodology more broadly. To continue to deliver innovations, our field needs to significantly increase participation in and funding for developing sound methodological foundations.

1. Introduction

Methodology is the foundation for judging innovation in experimental computer science. It therefore directs and *misdirects* our research. Flawed methodology can make good ideas look bad or bad ideas look good. Like any infrastructure, such as bridges and power lines, methodology is often mundane and thus vulnerable to neglect. While systemic misdirection of research is not as dramatic as a bridge collapse [11] or complete power failure [10], the scientific and economic cost may be considerable. Sound methodology includes using appropriate *workloads*, principled *experimental design*, and rigorous *analysis*. Unfortunately, many of us struggle to adapt to the rapidly changing computer science landscape. We use archaic benchmarks, out-dated experimental designs, and/or inadequate data analysis. This paper explores the methodological gap, its consequences, and some solutions. We use the commercial uptake of *managed languages* over the past decade as the driving example.

Many developers today choose managed languages, which provide: (1) memory and type safety, (2) automatic memory management, (3) dynamic code execution, and (4) well-defined boundaries between type-safe and unsafe code (e.g., JNI and Pinvoke). Many such languages are also object-oriented. Managed languages include Java, C#, Python, and Ruby. C and C++ are not managed languages; they are compiled-ahead-of-time, not garbage collected, and unsafe. Unfortunately, managed languages add at least three new degrees of freedom to experimental evaluation: (1) a *space-time tradeoff* due to garbage collection, in which heap size is a control variable; (2) *nondeterminism* due to adaptive optimization and sampling technologies; and (3) system *warm-up* due to dynamic class loading and just-in-time (JIT) compilation.

Although programming language researchers have embraced managed languages, many have not evolved their evaluation methodologies to address these additional degrees of freedom. As we shall show, weak methodology leads to incorrect findings. Equally problematic, most architecture and operating systems researchers do not use appropriate workloads. Most ignore managed languages entirely, despite their commercial prominence. They continue to use C and C++ benchmarks, perhaps because of the significant cost and challenges of developing expertise in new infrastructure. Regardless of the reasons, the current state of methodology for managed languages often provides bad results or no results.

To combat this neglect, computer scientists must be vigilant in their methodology. This paper describes how we addressed some of these problems for Java and makes recommendations for other domains. We discuss how benchmark designers can create *forward-looking and diverse workloads* and how researchers should use them. We then present a set of *experimental design guidelines* that accommodate complex and nondeterministic workloads. We show that managed languages make it much harder to produce meaningful results, and suggest how to identify and explore control variables. Finally, we discuss the importance of *rigorous analysis* [8] for complex nondeterministic systems that are not amenable to trivial empirical methods.

We address neglect in one domain, at one point in time, but the broader problem is widespread and growing. For example, researchers and industry are pouring resources into and exploring new approaches for embedded systems, multicore architectures, and concurrent programming models. However, without consequent investments in methodology, how can we confidently evaluate these approaches? The community must take responsibility for methodology. For example, many Java evaluations still use SPECjvm98, which is badly out of date. Out-of-date benchmarks are problematic because they pose last year's problems and can lead to different conclusions [17]. To ensure a solid foundation for future innovation, the community must make continuous and substantial investments. Establishing community standards and sustaining these investments requires open software infrastructures containing the consequent artifacts.

For our part, we developed a new benchmark suite and new methodologies. We estimate that we have spent 10 000 personhours to date developing the DaCapo suite and associated infrastructure, none of it directly funded. Such a major undertaking would be impossible without a large number of contributing institutions and individuals. Just as NSF and DARPA have invested in networking infrastructure to foster the past and future generations of the Internet, our community needs foundational investment in methodological infrastructure to build next-generation applications, software systems, and architectures. Without this investment, what will be the cost to researchers, industry, and society in lost opportunities?

2. Workload Design and Use

The DaCapo research group embarked on building a Java benchmark suite in 2003 after we highlighted the dearth of realistic Java benchmarks to an NSF review panel. The panel suggested we solve our own problem, but our grant was for dynamic optimizations. NSF did not provide additional funds for benchmark development, but we forged ahead regardless. The standard workloads at the time, SPECjvm98 and SPECjbb2000 [14, 15], were out of date. For example, SPECjvm98 and SPECjbb2000 make meager use of Java language features, and SPECjvm98 has a tiny code and memory footprint (SPEC measurements are in a technical report [3]). We therefore set out to create a suite suitable for research, a goal that adds new requirements beyond SPEC's goal of product comparisons. Our goals were:

Relevant and diverse workload: A diverse, widely used set of nontrivial applications that provide a compelling platform for innovation.

Suitable for research: A controlled, tractable workload amenable to analysis and experiments.

We selected the following benchmarks for the initial release of the DaCapo suite, based on criteria described below.

antlr	A parser generator and translator generator
bloat	A Java bytecode-level optimization and analysis tool
chart	A graph-plotting toolkit and PDF renderer
eclipse	An integrated development environment (IDE)
fop	An output-device–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 transformer for XML documents

2.1 Relevance and Diversity

No workload is definitive, but a narrow scope makes it possible to attain some coverage. We limited the DaCapo suite to nontrivial, actively maintained real-world Java applications. We solicited and collected candidate applications. Because source code supports research, we considered only open-source applications. We first packaged candidates into a prototype DaCapo harness and tuned them with inputs that produced *tractable* execution times suitable for experimentation, that is, around a minute on 2006 commodity hardware. Section 2.2 describes how the DaCapo packaging provides tractability and standardization.

We then quantitatively and qualitatively evaluated each candidate. Table 1 lists the static and dynamic metrics we used to ensure that the benchmarks were relevant and diverse. Our original paper [4] presents the DaCapo metric data and our companion technical report [3] adds SPECjvm98 and SPECjbb200. We compared against SPEC as a reference point, and compared candidates with each other to ensure diversity.

We used new and standard metrics. Our standard metrics included the static CK metrics, which measure code complexity of object-oriented programs [6]; dynamic heap composition graphs, which measure time-varying lifetime properties of the heap [16]; and architectural characteristics such as branch misprediction rates and instruction mix. We introduced new metrics to capture domain-specific characteristics of Java such as allocation rate, ratio of allocated to live memory, and heap mutation rate. These new metrics included summaries and time series of allocated and live object size demographics, summaries and time series of pointer distances, and summaries and time series of mutation distances. Pointer distance and mutation distance time-series metrics summarize the lengths of the edges that form the application's object graph. We designed these metrics and their means of collection to be abstract, so that the measurements are VM-neutral [4].

Figure 1 qualitatively illustrates the temporal complexity of heap composition and pointer distance metrics for two benchmarks, *_209_db* and *eclipse*. With respect to our metrics, *eclipse* from DaCapo is qualitatively richer than *_209_db* from SPECjvm98. Our original paper explains how to read these graphs and includes dozens of graphs, representing mountains of data [4]. Furthermore, it shows that the DaCapo benchmarks substantially improve over SPECjvm98 on all measured metrics. To confirm the diversity of the suite, we applied principal component analysis (PCA) [7] to the summary metrics. PCA is a multivariate statistical technique for reducing a large N-dimensional space into a lower-dimensional uncorrelated space. If the benchmarks are uncorrelated in lower-dimensional space, then they are also uncorrelated in the higher-dimensional space. The analysis shows that the DaCapo benchmarks are diverse, nontrivial real-world applications with significant memory load, code complexity, and code size.

Because the applications come from active projects, they include unresolved performance anomalies, both typical and unusual programming idioms, and bugs. Although not our intention, their rich use of Java features uncovered bugs in some commercial JVMs. The suite notably omits Java application servers, embedded Java applications, and numerically intensive applications. Only a few benchmarks are explicitly concurrent. To remain relevant, we plan to update the DaCapo Benchmarks every two years to their latest version, add new applications, and delete applications that have become less relevant. This relatively tight schedule should reduce the extent to which vendors may tune their products to the benchmarks (which is standard practice, notably for SPECjbb2000 [1]).

As far as we know, we are the first to use quantitative metrics and PCA analysis to ensure that our suite is diverse and nontrivial. The designers of future suites should choose additional aggregate and time-varying metrics that directly address the domain of interest. For example, metrics for concurrent or embedded applications might include a measure of the fraction of time spent executing purely sequential code, maximum and time-varying degree of parallelism, and a measure of sharing between threads.

2.2 Suitable for Research

We decided that making the benchmarks tractable, standardized, and suitable for research was a high priority. While not technically deep, good packaging is extremely time consuming and affects usability. Researchers need tractable workloads because they often run thousands of executions for a single experiment. Consider comparing four garbage collectors over 16 heap sizes–that is 64 combinations we need to measure. Teasing apart the performance differences with multiple hardware performance monitors may add eight or more differently instrumented runs per combination. Using five trials to ensure statistical significance requires a grand total of 2560 test runs. If a single benchmark test run takes as long as 20 minutes (the time limit is 30 minutes on SPECjbb [15]), we would need over

Metric	Description
	Code Metrics
CK metrics [6]	Object-oriented programming metrics measuring source code complexity
Code size	Numbers of classes loaded, methods declared, total bytecodes compiled
Code footprint	Instruction cache and I-TLB misses
Optimization	Number of methods compiled, number optimized, percentage hot
	Heap Metrics
Allocation	Total bytes/objects allocated, average object size
Heap footprint	Maximum live bytes/objects, nursery survival rate
Fan-out/fan-in	Mean incoming and outgoing pointers per object
Pointer distance	Mean distance in bytes of each pointer encountered in a snapshot traversal of an age-ordered heap
Mutation distance	Mean distance in bytes of each pointer dynamically created/mutated by the application in an age-ordered heap
	Architecture Metrics
Instruction mix	Mix of branches, ALU, and memory instructions
Branches	Branch mispredictions per instruction for PMM predictor
Register dependence	Register dependence distances

Table 1. Quantitative selection metrics

a month on one machine for just one benchmark comparison–and surely we should test the four garbage collectors on many benchmarks, not just one.

Moreover, time-limited workloads do not hold work constant, so they are analytically inconvenient for reproducibility and controlling load on the JIT compiler and the garbage collector. Cycle-accurate simulation, which slows execution down by orders of magnitude, further amplifies the need for tractability. We therefore provide work-limited benchmarks with three input sizes; small, default, and large. For some of the benchmarks, large and default are the same. The largest ones typically executed in around a minute on circa 2006 commodity high-performance architectures.

We make simplicity our priority for packaging; we ship the suite as a single self-contained Java jar file. The file contains all benchmarks, a harness, input data, and checksums for correctness. The harness checksums the output of each iteration and compares it to a stored value. If the values do not match, the benchmark fails. We provide extensive configuration options for specifying the number of iterations, the ability to run to convergence with customized convergence criteria, and callback hooks before and after every iteration. For example, the user-defined callbacks can turn hardware performance counters on and off, or switch a simulator in and out of detailed simulation mode. We use these features extensively and are heartened to see others using them [12].

For standardization and analytical clarity, our benchmarks require only a single host and we avoid components that require user-configuration. By contrast, SPEC jAppServer, which models real-world application servers, requires multiple hosts and depends on third-party–configurable components such as a database. Here we traded some relevance for control and analytical clarity.

We provide a separate ‘source’ jar to build the entire suite from scratch. For licensing reasons, the ‘source’ jar automatically downloads the Java code from the licensor. With assistance from our users [5], our packaging now facilitates static whole program analysis, which is not required for standard Java implementations. Since the entire suite and harness are open-source, we happily accept contributions from our users.

2.3 The Researcher

Appropriate workload selection is a task for the community, consortia, the workload designer, and the researcher. Researchers make a workload selection, either implicitly or explicitly, when they conduct an experiment. This selection is often automatic: “Let’s use the same thing we used last time!” Since researchers invest heavily in their evaluation methodology and infrastructure, this path offers the least resistance. Instead, we need to identify the workloads and methodologies that best serve the research evaluation. If there is no satisfactory answer, it is time to form or join a consortium and create new suitable workloads and supporting infrastructure.

Do Not Cherry-Pick! A well-designed benchmark suite reflects a range of behaviors and should be used as a whole. Perez et al. demonstrate with alarming clarity that cherry-picking changes the results of performance evaluation [13]. They simulate 12 previously published cache architecture optimizations in an apples-to-apples evaluation on a suite of 26 SPECcpu benchmarks. There is one clear winner with all 26 benchmarks. There is a choice of 2 different winners with a suitable subset of 23 benchmarks, 6 winners with subsets of 18, and 11 winners with 7. When methodology allows researchers a choice among 11 winners from 12 candidates, the risk of incorrect conclusions, by either mischief or error, is too high. Section 3.1 shows that Java is equally vulnerable to subsetting.

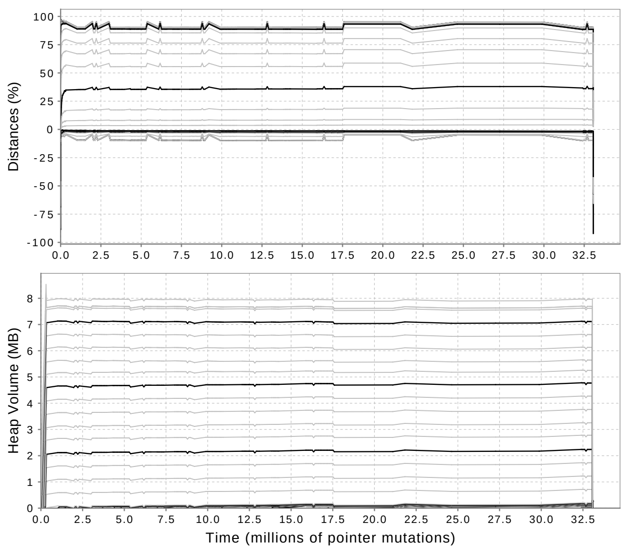
Run every benchmark. If it is impossible to report results for every benchmark because of space or time constraints, bugs, or relevance, explain why. For example, if you are proposing an optimization for multi-threaded Java workloads, you may wish to exclude benchmarks that do not exhibit concurrency. In this case, we recommend reporting all the results but highlighting the most pertinent. Otherwise readers are left guessing as to the impact of the “optimization” on the omitted workloads—with key data omitted, readers and reviewers should *not* give researchers the benefit of the doubt.

3. Experimental Design

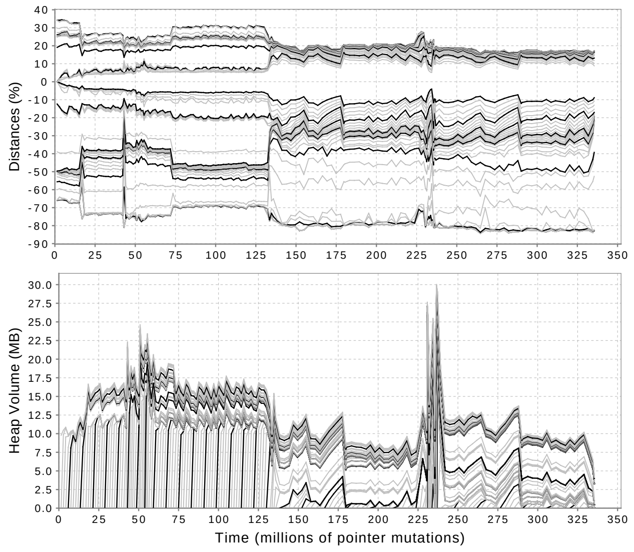
Sound experimental design requires a meaningful baseline and comparisons that control key parameters. Most researchers choose and justify a baseline well, but identifying which parameters to control and how to control them is challenging.

3.1 Gaming Your Results

The complexity and degrees of freedom inherent in these systems make it easy to produce misleading results through errors, omissions, or mischief. Figure 2 presents four results from a detailed comparison of two garbage collectors. The JVM, architecture, and other evaluation details appear in the original paper [4]. More garbage collector implementation details are in Blackburn et al. [2]. Each graph shows normalized time (lower is better) across a range of heap sizes that expose the space-time tradeoff for implementations of two canonical garbage collector designs, SemiSpace and MarkSweep.



(a) SPECjvm98 *_209_db*



(b) DaCapo *eclipse*

Figure 1. Two time-varying selection metrics. Pointer distance (top) and heap composition (bottom) as a function of time.

Subsetting Figure 2 badly misleads us in at least three ways: (1) Figure 2(c) shows that by selecting a single heap size rather than plotting a continuum, the results can produce diametrically opposite conclusions. At 2.1×, MarkSweep performs much better than SemiSpace, while at 6.0×, SemiSpace performs better. Figures 2(a) and 2(d) exhibit this same dichotomy, but have different crossover points. Unfortunately, some researchers are still evaluating the performance of garbage-collected languages *without* varying heap size. (2) Figures 2(a) and 2(b) confirm the need to use an entire benchmark suite. Although *_209_db* and *hsqldb* are established in-memory database benchmarks, SemiSpace performs better for *_209_db* in large heaps, while MarkSweep is always better for *hsqldb*. (3) Figures 2(c) and 2(d) show that the architecture significantly impacts conclusions at these heap size ranges. MarkSweep is better at more heap sizes for AMD hardware as shown in Figure 2(c). However, Figure 2(d) shows SemiSpace is better at more heap sizes for PowerPC (PPC) hardware. This example of garbage collection evaluation illustrates a small subset of the pitfalls in evaluating the performance of managed languages.

3.2 Control in a Changing World

Understanding what to control and how to control it in an experimental system is clearly important. For a classic comparison of Fortran, C, or C++ systems, there are at least two degrees of freedom to control: (a) the *host platform* (hardware and operating system), and (b) the *language runtime* (compiler and associated libraries). Over the years, researchers have evolved solid methodologies for evaluating compiler, library, and architectural enhancements that target these languages. Consider a compiler optimization for improving cache locality. Accepted practice is to compile with and without the optimization and report how often the compiler applied the optimization. To eliminate interference from other processes, one runs the versions stand-alone on one or more architectures and measures miss rates with either performance counters or a simulator. This methodology evolved, but is now extremely familiar. Once researchers invest in a methodology, the challenge is to notice when the world has changed and to figure out *how* to adapt.

Modern managed runtimes such as Java add at least three more degrees of freedom: (c) the *heap size*, (d) the *nondeterminism*, and (e) *warm-up* of the runtime system.

Heap Size Managed languages use garbage collection to detect unreachable objects, rather than relying on the programmer to explicitly delete objects. Garbage collection is fundamentally a space-time tradeoff between the efficacy of space reclamation and time spent reclaiming objects; heap size is the key control variable. The smaller the heap size, the more often the garbage collector will be invoked and the more work it will perform.

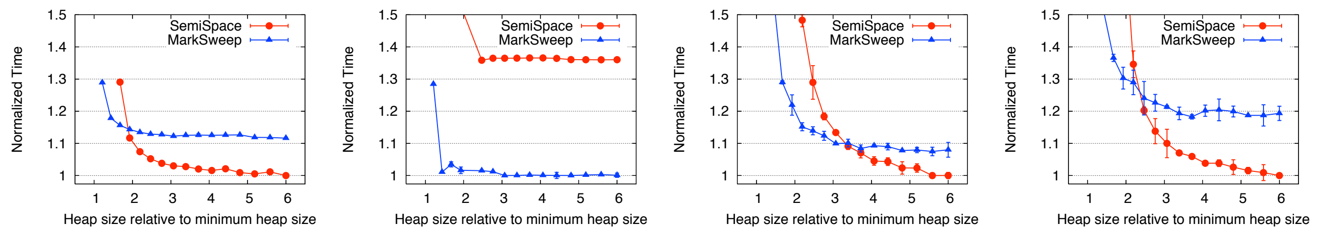
Nondeterminism Deterministic profiling metrics are expensive. High-performance JVMs therefore use approximate execution frequencies computed by low-overhead dynamic sampling to select which methods the JIT compiler will optimize and how. For example, a method may happen to be sampled N times in one invocation and $N+3$ in another; if the optimizer uses a hot-method threshold of $N+1$, it will make different choices. Due to this nondeterminism, code quality usually does not reach the *same* steady state on a deterministic workload across independent JVM invocations.

Warm-Up A single invocation of the JVM will often execute the same application repeatedly. The first iteration of the application usually includes the largest amount of dynamic compilation. Later iterations usually have both less compilation and better application code quality. Eventually, code quality may reach a steady state. Code quality thus "warms-up." Steady state is the most frequent use-case. For example, application servers run their code many, many times in the same JVM invocation and thus care most about steady-state performance. Controlling for code warm-up is an important aspect of experimental design for high-performance runtimes.

3.3 A Case Study

We consider performance evaluation of a new garbage collector as an example of experimental design. We describe the context and then show how to control the factors described above to produce a sound experimental design.

Two key context-specific factors for garbage collection evaluation are: (a) the *space-time tradeoff* as discussed above, and (b) the



(a) *_209_db*, Pentium-M (b) *hsqldb*, Pentium-M (c) *pseudojbb*, AMD (d) *pseudojbb*, PPC

Figure 2. Gaming your results. Four ways to compare two garbage collectors.

relationship between the collector and *mutator* (the term for the application itself in the garbage-collection literature). For simplicity, we consider a *stop-the-world* garbage collector, in which the collector and the mutator never overlap in execution. This separation eases measurement of the mutator and collector. Some collector-specific code mixes with the mutator: object allocation and *write barriers*, which identify pointers that cross between independently collected regions. This code impacts both the mutator and the JIT compiler. Furthermore, the collector greatly affects mutator locality, due to the allocation policy and any movement of objects at collection time.

Meaningful Baseline Comparing against the state of the art is ideal, but practical only when researchers make their implementations publicly available. Researchers can then implement their approaches using the same tools or control for infrastructure differences to make apples-to-apples comparisons. Garbage-collection evaluations often use generational MarkSweep collectors as a baseline because these collectors are widely used in high-performance VMs and perform well.

Host Platform Garbage collectors exhibit architecture-dependent performance properties that are best revealed with an evaluation across multiple architectures, as shown in Figures 2(c) and 2(d). These properties include locality, the cost of write barriers, and the cost of synchronization instructions.

Language Runtime The language runtime, libraries, and JIT compiler directly affect memory load, and so should be controlled. Implementing various collectors in a common toolkit factors out common shared mechanisms and focuses the comparison on the algorithmic differences between the collectors.

Heap Size Garbage collection evaluations should compare performance across a range of benchmark-specific relative heap sizes, starting at the smallest heap in which any of the measured collectors can run, as shown by Figure 2. Each evaluated system must experience the same memory load which requires forcing collections between iterations to normalize the heap, and controlling the JIT compiler.

Nondeterminism Nondeterministic JIT optimization plans lead to nondeterministic mutator performance. JIT optimization of collector-specific code, optimizations that elide allocations, and the fraction of time spent in collection may affect mutator behavior in ways that cannot be predicted or repeated. For example in Jikes RVM, a Java-in-Java VM widely used by researchers, JIT compiler activity directly generates garbage collection load because the compiler allocates and executes in the same heap as the application. These effects make nondeterminism even more acute.

Warm-Up For multi-iteration experiments, as the system warms up, mutator speeds increase and JIT compiler activity decreases, the *fraction* of time spent in collection typically grows. Steady-state execution therefore accentuates the impact of the garbage collector as compared to start-up. Furthermore, the relative impact of collector-specific code will change as the code is more aggressively optimized. Evaluations must therefore control for code quality and warm-up.

3.4 Controlling Nondeterminism

Of the three new degrees of freedom outlined in Section 3.2, we find dealing with nondeterminism to be the most methodologically challenging. Over time, we have adopted and recommend three different strategies: (a) use deterministic *replay* of optimization plans, which requires JVM support, (b) take multiple measurements in a single JVM invocation, after reaching steady state and turning off the JIT compiler, and (c) generate sufficient data points and apply suitable statistical analysis [8]. Depending on the experiment, the researcher will want to perform one, two, or all of these experiments. The first two reduce nondeterminism for analysis purposes by controlling its sources. Statistical analysis of results from (a) and (b) will reveal whether differences from the remaining nondeterminism are significant. The choice of (c) accommodates larger factors of nondeterminism (see Section 4) and may be more realistic, but requires significantly more data points, at the expense of other experiments.

Replay Compilation Replay compilation collects profile data and a compilation plan from one or more training runs, forms an optimization plan, and then replays it in subsequent, independent timing invocations [9]. This methodology deterministically applies the JIT compiler, but requires modifications to the JVM. It isolates the JIT compiler activity, since replay eagerly compiles to the plan's final optimization level instead of lazily relying on dynamic recompilation triggers. Researchers can measure the first iteration for deterministic characterization of start-up behavior. Replay also removes most profiling overheads associated with the adaptive optimization system, which is turned off. As far as we are aware, production JVMs do not support replay compilation.

Multi-Iteration Determinism An alternative approach that does not depend on runtime support is to run multiple measurement iterations of a benchmark in a single invocation, *after* the runtime has reached steady state. Unlike replay, this approach does not support deterministic measurement of warm-up. We use this approach when gathering data from multiple hardware performance counters, which requires multiple distinct measurements of the same system. We first perform $N-1$ unmeasured iterations of a benchmark while the JIT compiler warms up the code. We then turn the JIT compiler off and execute the Nth iteration unmeasured to drain any JIT work queues. We measure the next K iterations. On each iteration, we gather different performance counters of interest. Since the code quality has reached steady state, it should be a representative mix of optimized and unoptimized code. Since the JIT compiler is turned off, the variation between the subsequent iterations should be low. The variation can be measured and verified.

3.5 Experimental Design in Other Settings

In each experimental setting, the relative influence of the degrees of freedom, and how to control them, will vary. For example, when evaluating a *new compiler optimization*, researchers should hold the garbage collection activity constant to keep it from obscuring the effect of the optimization. Comparing on multiple architectures is best, but is limited by the compiler back-end. When evaluating a *new architecture*, vary the garbage collection load and JIT compiler activity, since both have distinctive execution profiles. Since architecture evaluation often involves very expensive simulation, eliminating nondeterminism is particularly important.

4. Analysis

Researchers use data analysis to identify and articulate the significance of experimental results. This task is more challenging when systems and their evaluation become more complex, and the sheer volume of results grows. The primary data analysis task is one of aggregation: (a) across repeated experiments to defeat experimental noise, and (b) across diverse experiments to draw conclusions.

Aggregating data across repeated experiments is a standard technique for increasing confidence in a noisy environment [8]. In the limit, this approach is in tension with *tractability*, because researchers have only finite resources. Reducing sources of nondeterminism with sound experimental design improves tractability. Since noise cannot be eliminated altogether, multiple trials are inevitably necessary. Researchers must aggregate data from multiple trials and provide evidence such as confidence intervals to reveal whether the findings are significant. Georges et al. [8] use a survey to show that current practice lacks statistical rigor and explain the appropriate tests for comparing alternatives.

Section 2.3 exhorts researchers not to cherry-pick benchmarks. Still, researchers need to convey results from diverse experiments succinctly, which necessitates aggregation. We encourage researchers (a) to include complete results, and (b) to use appropriate summaries. For example, using the geometric mean dampens the skewing effect of one excellent result. Although industrial benchmarks will often produce a single aggregate score over a suite, this methodology is brittle because the result depends entirely on vagaries of the suite composition [18]. For example, while it is tempting to cite your best result—*"we outperform X by up to 1000%"*—stating an aggregate together with the best and worst results is more honest and insightful.

5. Conclusion

Methodology plays a strategic role in experimental computer science research and development by creating a common ground for evaluating ideas and products. Sound methodology relies on *relevant workloads*, the use of *principled experimental design*, and *rigorous analysis*. Evaluation methodology can therefore have a significant impact on a research field, potentially accelerating, retarding, or misdirecting energy and innovation. However, we work within a fast-changing environment and our methodologies must adapt to remain sound and relevant. Prompted by concerns among ourselves and others about the state of the art, we spent thousands of hours at eight institutions examining and addressing the problems of evaluating Java applications. The lack of direct funding, the perception that methodology is mundane, and the magnitude of the effort surely explain why these efforts are uncommon.

We address neglect of evaluation methodology concretely, in one domain at one point in time, and draw broader lessons for experimental computer science. The development and maintenance of the DaCapo benchmark suite and associated methodology have brought some much-needed improvement to our evaluations and to our particular field. However, experimental computer science cannot expect the upkeep of its methodological foundations to fall to ad hoc volunteer efforts. We encourage stakeholders such as industry and granting agencies to be forward-looking and make a systemic commitment to stem methodological neglect. Invest in the foundations of our innovation.

Acknowledgments

We thank Andrew Appel, Randy Chow, Frans Kaashoek, and Bill Pugh, who encouraged this project at our three year NSF ITR review. We thank Mark Wegman, who initiated the public availability of Jikes RVM, and the developers of Jikes RVM. We gratefully acknowledge Fahad Gilani, who wrote the original version of the measurement infrastructure for his ANU Masters thesis; Xianglong Huang and Narendran Sachindran, who helped develop the replay methodology; and Jungwoo Ha and Magnus Gustafsson, who helped developed the multi-iteration replay methodology. We thank Tom Horn for his proof reading, and Guy Steele for his careful reading and suggestions.

References

[1] A. Adamson, D. Dagastine, and S. Sarne. SPECjbb2005 – A year in the life of a benchmark. In *2007 SPEC Benchmark Workshop*. SPEC, Jan. 2007.

[2] 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.

[3] 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.

[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 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[5] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *21st European Conference on Object-Oriented Programming, July 30th–August 3rd 2007, Berlin, Germany*, number 4609 in Lecture Notes in Computer Science, pages 525–549. Springer Verlag, 2007.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] G. H. Dunteman. *Principal Components Analysis*. Sage Publications, 1989.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 57–76, Montreal, Quebec, Canada, 2007.

[9] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, 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.

[10] B. Leyland. Auckland central business district supply failure. *Power Engineering Journal*, 12(3):109–114, June 1998.

[11] National Transportation Safety Board. NTSB urges bridge owners to perform load capacity calculations before modifications; I-35W investigation continues. SB-08-02. `http://www.ntsb.gov/Pressrel/2008/080115.html`, Jan. 2008.

[12] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ACM/IEEE International Symposium on Computer Architecture*, pages 174–185, New York, NY, USA, 2007. ACM.

[13] D. G. Perez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *International Symposium on Microarchitecture*, pages 43–54, Portland, OR, Dec. 2004.

[14] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[15] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[16] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, Dec. 1998.

[17] 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 *International Conference on Supercomputing*, pages 75–86, Cairns, Queensland, Australia, July 2006.

[18] R. M. Yoo, H.-H. S. Lee, H. Lee, and K. Chow. Hierarchical means: Single number benchmarking with workload cluster analysis. In *IISWC 2007. IEEE 10th International Symposium on Workload Characterization*, pages 204–213. IEEE, 2007.