Benchmarking the Performance of Application Startup for AspectJ-aware Java Virtual Machines

by

Peter Selby

BSc Computer Science, University of Winnipeg, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

July 2013

© Peter Selby, 2013
Abstract

In [12] and [13], Golbeck implemented AspectJ Virtual Machine (AJVM), an in-VM, lazy aspect weaver to improve the startup performance of applications containing aspects. In this work, it was important to clearly show an improvement in performance over earlier aspect weavers at startup. However, the lack of large applications using aspects and the complexity of startup benchmarking make such performance evaluation difficult. This thesis examines these challenges and presents our approach to application startup benchmarking. Using this approach, it clearly identifies previously unknown performance differences between aspect-aware Virtual Machine (VM) implementations.
Preface

This dissertation is original, unpublished work by the author, Peter Selby. This thesis was part of a larger project to compare different approaches to aspect weaving and their performance during application startup. Data from this thesis was therefore used in [13] and [11].
# Table of Contents

Abstract .......................................................... ii
Preface .............................................................. iii
Table of Contents .................................................... iv
List of Tables ......................................................... vi
List of Figures ....................................................... vii
Glossary ............................................................. viii

1 Introduction ....................................................... 1
   1.1 Problem ...................................................... 3
   1.2 Workload .................................................... 4

2 Startup Benchmarking .......................................... 7
   2.1 Benchmarking Approaches - Strengths and Weaknesses .... 8
      2.1.1 Microbenchmarks ....................................... 8
      2.1.2 Synthetic Benchmarks .................................. 11
      2.1.3 Real-world Benchmarks ................................ 13

3 Our Approach to Startup Benchmarking ....................... 15
   3.1 Statistical Rigour .......................................... 16
   3.2 Benchmarking Language Features .......................... 17
   3.3 Startup Microbenchmarks .................................. 21
   3.4 Startup Macrobenchmarks - DaCapo ....................... 23
   3.5 Results and Interpretation ................................. 29

4 Threats to Validity ............................................. 33
List of Tables

Table 3.1 Summation of all language feature microbenchmark results 19
Table 3.2 Summation of all language feature microbenchmark results (continued) 20
Table 3.3 Summary of startup characteristics of Eclipse, Tomcat and the DaCapo benchmark suite 25
Table A.1 Measurements from Eclipse startup and DaCapo eclipse benchmark 44
List of Figures

Figure 3.1  Language Feature Microbenchmarks ................. 21
Figure 3.2  Startup Microbenchmarks ............................ 24
Figure 3.3  DaCapo - No Aspect ................................. 27
Figure 3.4  DaCapo - Non-Matching Aspect ...................... 28
Figure 3.5  Estimated Eclipse Runtime - Non-Matching Aspect .... 32
Glossary

API  application programming interface  
AJVM  AspectJ Virtual Machine  
JIT  Just-in-Time Compiler  
VM  Virtual Machine  
JVM  Java Virtual Machine  
RVM  Jikes Research JVM  
AOP  Aspect-Oriented Programming  
GUI  Graphical User Interfaces  
JVMTI  Java Virtual Machine Tool Interface
Chapter 1

Introduction

Application startup performance is an important, and often overlooked, aspect of overall performance of a program. Poor application startup performance can be a serious problem especially during the edit-compile-run cycle of software development in particular, as it can dramatically slow the pace of development, where changes in code force the recompilation of projects, followed by application startup.

Various techniques for procedural or object-oriented code have been developed to minimize this startup cost. Usually, it is not necessary to recompile a complete project; it is sufficient to compile those parts of the program which have been changed.

Aspect-Oriented programming, as introduced in [16, 18], allows programming within alternative program decompositions. AspectJ ([17]) provides an alternative to the primary object-based decomposition of Java, allowing code to be woven, or injected, at specified join points in code. The advantage to this approach is that it can improve the modularity of cross-cutting code, or code which is not isolated in the primary decomposition, thus reducing the complexity and redundancy of code.

Due to the fact that aspects in Aspect-Oriented code are by definition cross-cutting, they are not amenable to the trivial optimizing techniques mentioned above. Modification of a single aspect can affect a large number of classes or procedures, requiring a significant amount of weaving or recompiling, and thus dramatically slowing application startup and the edit-compile-run development cycle. This problem has led to the avoidance of Aspect-Oriented Programming (AOP), or to special-case rather than general, systemic use of aspects.
One potential solution to this issue is to perform aspect weaving at run time. There are several points during program execution at which aspect weaving might occur. In [13], Golbeck implemented both class-granularity load-time and lazy method-granularity execution-time weaving within the Jikes research Java virtual machine (JikesRVM). We contrasted these approaches with the AJC VM-external class load-time weaver on the same VM.

These approaches to aspect weaving stand to significantly improve the performance of application startup, for two reasons: first, implementing the weaver within the VM stands to eliminate the redundant datastructures required by the VM-external approach, and the associated performance penalty, by sharing the VM’s internal datastructures; second, the lazy method-level weaver need only weave code which is actually executed by the VM, rather than all loaded code, in the case of the class-level weavers, or all code, in the case of the AJC compile-time weaver. As only a fraction of loaded code is generally executed at startup, this saving could significantly improve startup performance.

Thus, we expected the lazy in-VM weaver to outperform AJC’s load-time weaver for application startup. At the same time, we did not expect any performance degradation in either long-term steady-state performance or in the execution of standard Java code; once weaving is completed, the resulting bytecode should be essentially identical, and in all approaches, no significant work is required if no aspects are present, though some extra loading may occur.

In order to quantify our claims, we wished to show a clear improvement in application startup for real-world programs containing AspectJ aspects, even in the presence of unwoven aspect changes. Additionally, we wanted to demonstrate that neither steady-state nor unmodified Java code suffered a deterioration in performance.

Achieving the goal of accurately showing an improvement in application startup for Java programs containing aspects is not, however, as straightforward as it might first appear.

First, there are no large, real-world Java applications utilizing aspects systematically with which to benchmark and compare the various approaches to weaving. This is clearly a general problem with the development of new language features: obviously, they will not have widely-used code suitable for real-world benchmarks.

For benchmarking new language features, it is usually sufficient to use
microbenchmarks. These are easy to design, focused, and easy to interpret, although extrapolating overall performance from microbenchmarks is problematic, as discussed in 1.2. In our case, however, we are focused specifically on startup performance for large, complex, real-world applications. Golbeck’s improvements do not lend themselves to simple feature-by-feature comparison, and the size, complexity, and variety of the startup workload for large applications makes extrapolation from simple microbenchmarks especially troublesome.

Even in the case of standard Java programs, benchmarking application startup is difficult, due to the complexity introduced by garbage collection, Just-in-Time Compiler (JIT) compilation, and the performance characteristics of the VM itself [8-10, 19, 20]. As discussed in 1.2 these factors can make performance analysis very difficult; overly simple benchmarks can lead to very inaccurate conclusions.

Generally, startup benchmarks measure the performance of startup for typical widely-used applications. Given that there are no large, representative applications with aspects that might serve as an effective startup benchmark for our implementations, much less typical or widely-used ones, we were left to consider other approaches to benchmarking which would allow us to make quantifiable claims about startup performance, preferably by allowing us to approach our ideal benchmark.

1.1 Problem

Given the lack of appropriate applications with which to perform startup benchmarks, we needed an alternative approach to accurately estimate what the difference in performance between the various weaving approaches would be on such an application.

Among the approaches we considered were the following:

1. Create a large program, synthetic or otherwise, with aspects
2. Add aspects to an existing, widely-used code base
3. Restrict ourselves to microbenchmarks
4. Use an industry-standard benchmark suite

We discuss the advantages and disadvantages of each approach in 2.1.
Our case was further complicated by the fact that our in-VM weaving implementations were implemented in the (Jikes Research JVM (RVM)), a metacircular Java VM programmed mostly in Java and intended for research projects, which does not completely support new Java language features. Thus, many Java programs are unable to run on the RVM, complicating the use of standard benchmarking applications.

1.2 Workload

We were interested in the startup performance of applications on different Java VMs. Application startup can be defined as the time between the initial execution of an application and the point at which it is able to perform useful work in response to user input. It differs in several significant ways from steady-state or long-running workloads, which are the typical target of benchmarking suites:

- The garbage collector, which plays a significant role in stable-state performance, plays a smaller role in startup, which is characterized instead by heavy loading, data structure initialization, and memory allocation [5].

- All code is initially unoptimized; the JIT compiler works heavily [1, 6, 20], and over time code is optimized and performance improves.

- Heavy class loading occurs; of the loaded code, generally speaking, only a fraction will be run, and of that only a fraction will be JIT-compiled due to heavy utilization; some example figures can be found in section 3.4. This profile differs from many common benchmarks, especially microbenchmarks, where a larger proportion of loaded code is executed and JIT-compiled. The VMs we wished to compare differed during class loading, and immediately before code is executed; thus we expected the largest performance differences to occur for startup workloads.

- Generally, the VM engages in a varied workload relative to most steady-state execution, and especially relative to simple microbenchmarks or benchmarks which focus on computationally-intensive tasks such as the SPEC benchmark suite [4] [5]. Likewise, a startup workload will generally involve a wider variety of tasks than a steady-state workload:
code is loaded from disk, aspects woven, code JIT-compiled, unoptimized code executed, and data structures initialized. During steady-state performance little code is loaded, aspects are already woven into the loaded bytecode and the resulting bytecode optimized by the JIT compiler, and most accessed data structures are already initialized.

With difficulty, these characteristics might be achieved using synthetic benchmarks; however, a large body of work regarding benchmarks emphasizes the importance of using large, complex, realistic benchmark workloads. Simulating such workloads would be prohibitively difficult; ‘success’ would be difficult to measure, and the possibility would remain that any perceived difference in performance between VMs may be due to overlooked issues with the synthesized code.

[8] points out the danger of scaling small benchmarks up to create large benchmarks, or extrapolating the performance of complex real-world benchmarks based on the result of small or simple benchmarks. Both small benchmarks and simple benchmarks (i.e. computationally-intense, well-defined tasks) tend to be dominated by the performance of the VM, overemphasizing differences between VMs, which would have a much smaller effect on large and complex workloads. Section 2.1.1 discusses the extent to which a VM can affect the performance of a simple benchmark.

The DaCapo benchmarks [4] were designed to provide a realistic workload according to the criteria suggested by [7], including the size and complexity of code and data structures, depth of inheritance and complexity of polymorphism, scope and complexity of memory usage, and other criteria. They argue the importance of such workloads, pointing out that the DaCapo benchmarks, and in particular the eclipse benchmark (Sec 5.2), warm up slowly relative to the SPEC benchmark suite and other widely-used benchmarks. This supports the assertion that computationally-intense but simple benchmarks will be more affected by the VM (that is, the JIT compiler) than more balanced workloads. Such benchmarks can still be very useful in examining performance during steady-state, which also tends to be strongly influenced by the VM, and in particular the optimizing capabilities of the JIT compiler. During startup, however, heavy class loading, weaving, JIT-compilation, and work by the garbage collector have more influence than the

---

1Where complexity is measured by metrics such as methods-per-class, depth-of-inheritance, coupling-between-objects, lack-of-cohesion-in-methods, and other parameters
execution of optimized code. It is these elements of VM execution that Golbeck’s implementation targets, so realistic benchmarks are especially vital in our case.

Since our ideal workload is the startup performance of a complex real-world application—in particular Eclipse, which [4] points out is particularly slow to optimize—it was important for us to choose the closest possible workload available. This point was underlined by [3], in which the authors argue that the best determinant of large program performance is not the VM, garbage collector, code or design quality, or other external factors, but instead is the purpose of the software in question; that is to say, inherent properties of a task will significantly determine the performance of a program. This point is further emphasized in [19]. This makes the choice of benchmark workload especially important. The fact that we were able to use the actual codebase of the Eclipse project (albeit without Graphical User Interfaces (GUI)) using the DaCapo eclipse benchmark is encouraging.

[3] points out that “There are no quantitative, testable, predictive theories about the internal structures of large scale systems”, which makes the synthesis of a characteristic workload even less appealing, and suggests that it is difficult to know with any certainty whether two different code bases can be treated as equivalent for the purposes of performance.

[4], [8], [3], and [7] all emphasize the importance of using a variety of workloads for benchmarking. While we were particularly focused on the startup performance of the eclipse benchmark, we tested all VMs with several other working DaCapo benchmarks, as well as idealized synthetic startup benchmarks. This gave us a good indication of how the various VMs performed for a variety of different workloads.
Chapter 2

Startup Benchmarking

Our ideal benchmark for comparing the performance of our VM-internal aspect weaving implementations with the traditional AJC aspect weavers would be a large, complex, industry-standard application making extensive and typical use of AspectJ language features. No such program exists. This is in part because AspectJ is itself quite new. Furthermore, the performance cost of weaving complex aspects across large amounts of code has served as a deterrent to systemic aspect use.

Since our ideal benchmark was not available, we considered a number of approaches to approximating or replacing it with another approach or combination of approaches, including:

Creating or modifying an appropriate program We might have written or modified a program to use AspectJ language features, and used it for benchmarking; however, time constraints made this unrealistic, and in any case such a program could no longer be considered industry-standard, and ensuring that it could be considered representative of typical applications would be difficult, particularly in view of the issues raised in 1.2. Aspects inserted into an existing program would have to be ‘realistic’, in number, scope, cross-cutting extent and behaviour, all of which are difficult to quantify.

Large synthetic benchmark We considered generating large synthetic programs to emulate a startup workload, but the difficulty of simulating a real-world workload is too great for this approach to be feasible; and even given apparently realistic results, the possibility remains that
any difference in performance (especially relative performance) is due to an unrealistic workload.

**Microbenchmarks** Microbenchmarks are widely used to test new language features, as they can give clear and unambiguous results. However, they cannot capture the complexity of real-world application startup. Performance on microbenchmarks is not a good predictor of performance on large-scale benchmarks, and especially give little indication of the degree of any change in performance for complex macro programs [8].

**Industry-standard benchmarks** There are several widely-used benchmark suites which are designed to test VMs using a realistic workload. These have the advantage of widespread use and acceptance, and are well-understood according to accepted metrics of complexity in real-world code, thus increasing the relevance of results; however, none of the widely-used benchmark suites use AspectJ language features, and the focus of such benchmarks is not startup performance, but steady-state performance.

The following sections will investigate the strengths and weaknesses of each approach, and finally present and defend our approach, which we believe captures the most important aspects of startup performance, while introducing no serious sources of error or misleading measurements.

### 2.1 Benchmarking Approaches - Strengths and Weaknesses

#### 2.1.1 Microbenchmarks

Microbenchmarks are the most common form of benchmark for new language features, because they are easily designed and implemented, and can target the performance of specific features. They give clean and unambiguous results to very specific performance questions. These attributes make microbenchmarks effective for the measurement of individual AspectJ language features, and comparison between the performance of said features in different VM implementations.
While we did want to measure the performance of individual language features to ensure that it did not deteriorate, either immediately or in steady-state, we did not expect to see a significant improvement in most cases. Our primary concern, instead, was with application startup, and in particular with class loading and its interaction with the JIT compiler and other elements of performance.

For this purpose, microbenchmarks are much less suited. As discussed in 1.2, observed performance differences in microbenchmarks do not necessarily correspond to performance of complex real-world code, for many reasons. The execution of such code, and in particular the startup workload, involves a wide range of activities, including class loading and disk I/O, data structure initialization, execution of large amounts of unoptimized code and much higher levels of JIT-optimization, and so on. Apparent benefits revealed by microbenchmarks may disappear due to relative insignificance in a larger, more complex workload. More complex code might exhibit smaller improvements in performance than simple code, or even, in pathological cases, deteriorate. Features may not scale efficiently to large or complex problems or data sets.

Modern VMs and computer systems both exacerbate the performance differences between microbenchmarks and large applications. The size and behaviour of various system caches, as well as CPU optimizations such as out-of-order operations and pipelining, can have a dramatic effect on program performance; if data structures used in microbenchmarks are either unrealistically large and complex, small and simple, or if they are simply not diverse enough, or if they are not used in a representative way (i.e. using random or linear access, or a combination thereof), caches might have an undue effect on performance. The same is true of the effect of CPU optimizations on unrealistically simple or complex code and calculations.

The Java Virtual Machine (JVM) complicates accurate performance measurement, primarily due to the JIT compiler and the garbage collector, both of which can significantly alter application performance in a manner analogous to system caches and CPUs. Overly-simple code may be dramatically optimized by the JIT compiler; in extreme cases, the JIT compiler may actually eliminate significant benchmark work, for example removing loops which have no effect on program output. Short-running programs may spend an unrealistic amount of their total runtime optimizing code without benefiting from the performance benefits of those optimizations; long-running benchmarks may amortize the cost of optimization to irrelevance. The garbage
collector, like system memory, displays very different performance for different data structures; in simple benchmarks, for example, it is quite possible that no garbage collection will occur. Large applications tend to load far more code and data than they will use, particularly during application startup; depending on the application, significant amounts of data may be loaded, and data structures initialized. All of these factors mean that the performance of microbenchmarks are of limited use in predicting startup (or, indeed, any) large application performance, and it is important not to extrapolate too much from micro results.

The effect of the JIT compiler and garbage collector on simple benchmarks can be seen in our iterated microbenchmarks, in which we observe a degree of improvement much greater than is observed in realistic applications, represented here by DaCapo benchmarks. Furthermore, the benchmarks varied more from VM to VM, confirming the result from [8] that simple benchmarks cluster by VM, whereas more complex benchmarks cluster by task.

[8] confirms, after thoroughly analyzing the interaction between applications and Java VMs, that “using short-running [benchmarks] as representative of long-running [benchmarks] is clearly bad practise.”

[3] analyzes the behaviour of programs, and comes to the conclusion that it is the task of a program, rather than any other factors, which determine overall performance characteristics, due to the amount and nature of computation performed, the size and layout of data structures, and the patterns with which they are accessed, and other similar factors. This suggests that recreating a desired workload would be more complicated than it might initially seem; certainly, scaling up simple microbenchmarks is not sufficient to simulate a desired workload.

In our case, we are concerned with weaving, which always occurs prior to JIT-optimization; therefore, the time spent optimizing code and the performance of the resulting code are only a secondary concern; they do not directly affect the difference in performance of the various tested VMs. However, we wish to know the significance of this change in performance relative to other tasks. Code which is either too simple, resulting in fast optimization and unrealistic performance improvement, or too complex to be effectively optimized, would leave us with an inaccurate approximation.

The difference in relative performance between our idealized startup benchmarks (Fig 3.2) and the benchmarks of the DaCapo suite (Fig 3.3 and Fig 3.4) clearly illustrate the danger of extrapolating large application performance from micro- or synthetic benchmarks.
One problem with microbenchmarks, and many benchmarking suites, for analyzing benchmark startup, especially in our case, is that most benchmarks use a significant portion of their loaded code.

Microbenchmarks have previously been used to benchmark the performance of AspectJ language features ([15], [2]), with a focus on steady-state performance. All such microbenchmarks are very focused on specific features, which is in line with our approach.

2.1.2 Synthetic Benchmarks

Initially, we considered the possibility of synthesizing a startup workload, combining the desired characteristics of large industry-standard applications with widespread usage of aspects. By using this approach, we could generate a code base comparable in size to Eclipse or other desired benchmarks, with aspects interwoven throughout the code. Using code generation would allow us to approach realistic sizes for our code base much faster than either writing a new application from scratch or modifying an existing application.

However, this approach is not as straightforward or as sound as it might first appear. Further investigation led us to abandon the possibility of large synthetic benchmarks, due to the points raised below.

[3] argues that the primary determinant of application performance is the underlying purpose or task of the application. Additionally, the authors point out that there exists no effective theory regarding the complexity and internal structure of large, complex computer systems. This means that there is no effective metric by which to judge whether a synthetic benchmark approaches the complexity of a real-world application.

Since the task of an application is the primary determinant of its performance, and there are no metrics for identifying differences between different applications, attempting to simulate a particular application workload is reduced to guesswork. Any observed difference in performance might be due to unrealistic internal characteristics.

[7] lays out metrics against which the DaCapo suite benchmarks are measured, and it includes among code characteristics to be measured: methods-per-class, depth-of-inheritance, coupling-between-objects, and lack-of-cohesion-in-methods, along with the size and complexity of data structures (which would themselves require similar metrics). Falling short in these metrics (or, indeed, exceeding them) could easily lead to inaccurate or biased results. On the other hand, they are hardly comprehensive; while they give
an indication of the complexity of real-world code, they do not cover all possible sources of complexity within applications.

The JIT compiler complicates the generation of synthetic benchmarks, because in order for a benchmark to resemble an actual application, it would have to contain a realistic amount of optimizable code; code which is too simple has a dual risk of being over-optimizable (for example, if large sections of code had no effect on output they may be removed) or far too difficult to optimize (e.g. organized in such a way that no simplifications are possible). This adds another dimension to the problem of complexity—not only must the code and data resemble, in complexity, that of the real application, but the code should also be similar in complexity from the perspective of JIT optimizations. A similar argument can be made for the performance of the garbage collector, and indeed for the system CPU and system caches.

It might be argued that even if synthetic benchmarks don’t capture the full complexity of target applications, the relative performance of different VMs on the same benchmarks is still informative. However, differences in performance may emerge from particular patterns of execution; as an extreme example, a simple benchmark might load very little computationally-intensive code, JIT-compile most of it, and execute it repeatedly over a long period of time, spending only a small amount of time on allocation, garbage collection, or running unoptimized code, all of which are essential to application startup. The performance of such a benchmark would likely differ dramatically across different VMs, depending on the optimizations used by the JIT-compiler.

A more concrete and revealing example is supplied by [4], in which the authors point out that the SPEC benchmark suite warms up faster, and differs more in performance across VMs, than the DaCapo benchmarks. The SPEC benchmarks consist of real-world code performing actual work, but they are relatively simple, both according to the metrics suggested by [7] and in terms of tasks, where they focus on repetitive and CPU-intensive benchmarks. The DaCapo benchmarks, which strive for real-world complexity and use larger code bases and more varied workloads, warm up less quickly and differ less across different VMs. If real code can differ thus due to differences in optimization, the danger for synthetic code is clear.

Our initial reason for considering the creation of synthetic benchmarks was to approach our ideal benchmark, that is, a large industry-standard application (or a reasonable simulacrum thereof) containing a large code base with a realistic number of aspects. However, since achieving the right balance
of code and data structure complexity is difficult and the optimizations of the JIT, garbage collector, CPU and cache might all benefit unrealistically from any fundamental simplicity in code or data, and on the other hand the cost of scanning and weaving aspects is relatively static, the ratio of time spent on weaving versus time spent executing code is very likely to be significantly incorrect. Since the VMs differ primarily in approaches to weaving, this could greatly exaggerate the difference in performance between VMs, in a way that is difficult to isolate. Thus, synthetic benchmarks could tell us only very little about the performance of our VM implementations on complex code with aspects.

Benchmarks using real-world code, like the DaCapo suite, will also not perfectly report the difference in performance between VMs-especially since they lack aspects—but there is no danger of a significant ‘collapse’ in complexity due to the fact that the code is from real applications. Simple aspects allow us to judge the cost of scanning and weaving simple aspects, which, though not a perfect stand-in for actual aspects, give us a good idea of the cost of real aspects without any possibility of deceptive results.

2.1.3 Real-world Benchmarks

Finally, we considered using an industry standard benchmark suite to measure the performance of our VM implementations and that of the AJC weaver. This has the advantage of making our results more comparable and relevant to other approaches, while increasing the trustworthiness of our results. Industry-standard benchmarks are designed to provide realistic, real-world tests for Java virtual machines, and most have been analyzed and studied to ensure that they provide useful and relevant results.

However, there are two major drawbacks to industry-standard benchmarks:

- There are no such benchmarks utilizing AspectJ aspects in a systemic way

- Most Java benchmark suites are focused on steady-state performance; no benchmarks focus especially on the startup workload, which is where we expect to see a difference in performance

In spite of these drawbacks, we believe that industry-standard benchmarks offer us the best insight into performance available. By applying
simple aspects to benchmark code, we can establish the cost of weaving on complex code; and by considering the significant and relevant differences between the benchmark and startup workloads, we can estimate the performance difference for real-world code on different VMs. However, we would like our selected benchmarks to closely resemble actual applications, to capture as many performance characteristics relevant to startup as possible.

- The DaCapo Benchmark Suite
  The DaCapo benchmark suite was introduced in [4] with the intention of providing realistic, real-world code with which to benchmark Java VMs. All benchmarks were designed in accordance with [7] to ensure that the resulting benchmarks retained the complexity and depth of real-world code.

  The authors point out that many preceding benchmark suites focus on computation-heavy benchmarks which are, from a workload standpoint, simple; real applications tend to be much more complex than such benchmarks. They illustrate this point by comparing the DaCapo benchmark suite with the common SPEC benchmark suite, showing that the SPEC benchmarks 'warm up' significantly faster than the DaCapo benchmarks; that is to say, they reach their optimal performance more quickly. Their code is significantly simpler according to the metrics in [7]; they load less code and execute a greater portion. Thus, while the SPEC benchmarks are a useful tool for comparing the performance of one VM against another for certain use cases, they are not as useful at predicting the performance of actual applications; this is especially true for the startup workload.

  As previously mentioned, one industry-accepted startup benchmark is the Eclipse IDE. The DaCapo suite includes an eclipse benchmark, which uses the Eclipse code base to execute a series of benchmarks. See Appendix A for a brief summary of the relevant characteristics of the eclipse benchmark as compared to startup of the Eclipse IDE.

  The DaCapo benchmarks have the advantage, for us, of being developed on the JikesRVM virtual machine; though not all the benchmarks are able to run on our modified JVMs, we were able to use several, including the eclipse benchmark.
Chapter 3

Our Approach to Startup Benchmarking

In order to compare the performance of our VM implementations with the existing AJC aspect weaving approaches, our ideal benchmark was a widely-used, representative program utilizing AspectJ language features in a systemic way; however, as discussed above, no such program exists.

Our goal, therefore, was to approximate a startup workload, or capture the most relevant characteristics of such a workload, allowing us to confidently claim a performance improvement, and predict the degree of improvement that might be expected in the ideal benchmark.

Taking into account the arguments presented in 1.2 and 2.1 we formulated a combination of microbenchmarks to benchmark the performance, at startup and in steady-state, of AspectJ language features on each VM implementation; class-loading microbenchmarks to isolate the startup costs of class loading and the benefits of lazy weaving, and the overhead of weaving infrastructure in each VM; DaCapo benchmarks to benchmark the performance of each VM on standard Java code; and DaCapo benchmarks in the presence of simple AspectJ aspects to force the scanning of code to estimate the per-aspect cost of scanning and weaving for each VM.

Using this combination of benchmarks, we were able to show that:

- The performance of AspectJ language features is statistically indistinguishable across VMs, except in isolated cases where an in-VM approach allows for optimizations unavailable to external weavers; this remains true after JIT-optimization
• Our startup microbenchmarks clearly illustrate the benefits of both the in-VM approach, which eliminates costly redundant data structures, and the benefits of lazy execution-time weaving.

• Performance on the unmodified DaCapo benchmarks remains the same within margin of error, demonstrating that our modifications to the VM do not impact the performance of standard Java code.

• With the introduction of simple aspects to the DaCapo benchmarks which force the scanning of all loaded or executed code (depending on the implementation) to approximate the impact of a single widely cross-cutting aspect, our in-VM outperforms the AJC load time weaver, and approaches the performance of the compile-time AJC weaver.

3.1 Statistical Rigour

Benchmarking has become more complicated over time, as programs and environments get more complex; non-determinism is introduced by more advanced chips, memory and caching, and by the virtual machine. However, benchmarking methodology has, for the most part, remained unchanged. Benchmark results that are commonly given include single-run, second run, average over multiple runs, first- or second-best or -worst run, or combinations of these approaches. While all report useful information, they can omit or conceal significant differences. [9] argues that none of these approaches is rigorous enough to be useful for valid comparisons. Instead, the authors recommend using a statistically sound approach to benchmarking and reporting of results.

In order to achieve statistical soundness, we followed the recommendations of [9] and ran our benchmarks multiple times, collecting the results and calculating the mean time and the 95% confidence interval (using the Student’s T-distribution with n - 1 degrees of freedom) over all runs each time. In most cases, we continued until the confidence interval was less than a specified margin of error, which is shown or reported with the benchmarks. In the case of some long-running DaCapo benchmarks, we ran each benchmark a specified number of times and reported the resulting confidence interval. This was necessary because the number of runs required to reach a desired margin of error is unbounded, and the complexity of these benchmarks meant.
that the results were less uniform, and thus the confidence converged very slowly, if at all.

In all benchmarks, we ignored the first result because it was significantly slower than average due to class and library loading from disk. This initialization cost is, of course, very often an element of application startup; however, this cost is determined mostly by the hardware and software environment of the VM, putting it out of our scope of concern. The VM implementation influences this aspect of initialization performance only in the relative size of the code and data to be loaded, and the difference in size between the tested VMs was insignificant.

3.2 Benchmarking Language Features

In order to verify that the performance of AspectJ language features was equivalent in VM-internal and -external approaches, we compared the performance of language features on each VM with a suite of microbenchmarks.

Each benchmark is designed to target the performance of an individual language feature. Individual benchmarks are run independently of one another and iterate a specified number of times.

All benchmarks consist of a harness class, which instantiates the benchmark class, which in turn loops for the given number of iterations, calling a simple method repeatedly each time. The difference benchmarks vary only in the supplied advice, provided in a separate (and uncompiled) aspect file, which applies to the repeatedly-executed method.

In order to capture both the startup (or first-run) performance and the steady-state post-optimization performance of each feature, we executed each benchmark repeatedly, starting with a single iteration and doubling the iteration count each time, to a specified benchmark-dependent maximum. At each step we repeatedly executed the benchmark, collecting the measured run-time from execution to completion (including VM initialization), until we had a specified 95% confidence interval for the mean (see 3.1). For the microbenchmarks, we used a 2 ms margin of error. The resulting graph allows us to extrapolate both first-run and steady-state performance; the first-run counting VM initialization is simply the first measurement with 1 iteration; the cost of the first iteration can be measured by subtracting the 1st run from the second. By measuring the slope between the final two points, the steady-state per-iteration performance can be measured.
With JIT compilation disabled, performance increased linearly with the number of iterations; when enabled, per-iteration performance improved with more iterations. As expected, in both cases all VMs behave identically within error, with a few exceptions where the in-VM approach allowed for optimization.

Startup performance was much better for the VM-internal approaches. All implementations perform the same weaving, since there are no unused methods or aspects, so this difference in performance is not due to lazy weaving; rather, it is due to the necessary redundant data structures required for VM-external weaving and other initialization costs.

These results are summarized in 3.1, which shows the startup time (i.e. the performance for 1 iteration of a benchmark), the runtime of the last run, and the slope between the final two points, which illustrates the steady-state performance. The VMs measured are the AJC load-time weaver and our class-level and method-level (or lazy-weaving) implementations. Benchmarks where the different VMs showed a statistically-significant performance difference in steady state are emphasized.

The same data is summarized in 3.1.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FR</th>
<th>LR</th>
<th>$m_1$</th>
<th>FR</th>
<th>LR</th>
<th>$m_2$</th>
<th>$\frac{m_2}{m_1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpbeforerecall</td>
<td>969.25</td>
<td>1836.73</td>
<td>0.426</td>
<td>199.25</td>
<td>1655.30</td>
<td>0.417</td>
<td>0.98</td>
</tr>
<tr>
<td>interpflownoState</td>
<td>994.60</td>
<td>2306.31</td>
<td>14.2</td>
<td>199.66</td>
<td>595.40</td>
<td>2.97</td>
<td>0.21</td>
</tr>
<tr>
<td>interpstate01</td>
<td>986.40</td>
<td>1177.13</td>
<td>0.538</td>
<td>197.00</td>
<td>390.09</td>
<td>0.491</td>
<td>0.91</td>
</tr>
<tr>
<td>interpstate02</td>
<td>999.00</td>
<td>1113.37</td>
<td>3.27</td>
<td>197.50</td>
<td>322.00</td>
<td>2.72</td>
<td>0.83</td>
</tr>
<tr>
<td>interpunadvised</td>
<td>849.00</td>
<td>1058.60</td>
<td>0.342</td>
<td>194.50</td>
<td>417.33</td>
<td>0.410</td>
<td>1.20</td>
</tr>
<tr>
<td>cflow01</td>
<td>996.75</td>
<td>2304.16</td>
<td>11.7</td>
<td>200.25</td>
<td>583.00</td>
<td>2.75</td>
<td>0.23</td>
</tr>
<tr>
<td>thisTargetArgs01</td>
<td>992.50</td>
<td>1175.58</td>
<td>0.392</td>
<td>200.00</td>
<td>386.84</td>
<td>0.552</td>
<td>1.41</td>
</tr>
<tr>
<td>thisTargetArgs02</td>
<td>990.50</td>
<td>1230.46</td>
<td>0.854</td>
<td>199.50</td>
<td>506.15</td>
<td>1.84</td>
<td>2.15</td>
</tr>
<tr>
<td>thisTargetArgs03</td>
<td>998.50</td>
<td>1304.70</td>
<td>2.20</td>
<td>200.00</td>
<td>391.40</td>
<td>0.650</td>
<td>0.30</td>
</tr>
<tr>
<td>thisTargetArgs04</td>
<td>1001.16</td>
<td>1345.17</td>
<td>2.61</td>
<td>202.00</td>
<td>520.15</td>
<td>2.04</td>
<td>0.78</td>
</tr>
<tr>
<td>thisTargetArgs05</td>
<td>990.25</td>
<td>1244.35</td>
<td>1.52</td>
<td>196.50</td>
<td>391.35</td>
<td>0.533</td>
<td>0.35</td>
</tr>
<tr>
<td>thisTargetArgs06</td>
<td>994.00</td>
<td>1265.57</td>
<td>1.34</td>
<td>200.00</td>
<td>477.39</td>
<td>1.54</td>
<td>1.15</td>
</tr>
<tr>
<td>thisTargetArgs08</td>
<td>988.20</td>
<td>1174.26</td>
<td>0.445</td>
<td>198.66</td>
<td>424.26</td>
<td>0.656</td>
<td>1.48</td>
</tr>
<tr>
<td>thisTargetArgs10</td>
<td>1001.00</td>
<td>1280.19</td>
<td>1.74</td>
<td>203.27</td>
<td>443.63</td>
<td>0.723</td>
<td>0.42</td>
</tr>
<tr>
<td>thisTargetArgs12</td>
<td>989.00</td>
<td>1243.23</td>
<td>1.36</td>
<td>199.00</td>
<td>455.73</td>
<td>1.87</td>
<td>1.37</td>
</tr>
<tr>
<td>thisTargetArgs14</td>
<td>992.00</td>
<td>1229.81</td>
<td>0.819</td>
<td>201.50</td>
<td>471.00</td>
<td>1.26</td>
<td>1.53</td>
</tr>
<tr>
<td>thisTargetArgs16</td>
<td>996.25</td>
<td>1317.15</td>
<td>2.23</td>
<td>201.00</td>
<td>487.92</td>
<td>1.45</td>
<td>0.65</td>
</tr>
<tr>
<td>thisTargetArgs18</td>
<td>992.00</td>
<td>1252.20</td>
<td>1.64</td>
<td>200.00</td>
<td>465.21</td>
<td>1.62</td>
<td>0.99</td>
</tr>
<tr>
<td>trivial01</td>
<td>966.66</td>
<td>1154.66</td>
<td>0.464</td>
<td>201.00</td>
<td>373.66</td>
<td>0.360</td>
<td>0.78</td>
</tr>
<tr>
<td>trivial03</td>
<td>900.00</td>
<td>1044.00</td>
<td>0.478</td>
<td>199.75</td>
<td>354.59</td>
<td>0.584</td>
<td>1.22</td>
</tr>
<tr>
<td>trivial05</td>
<td>979.50</td>
<td>1235.84</td>
<td>1.06</td>
<td>202.50</td>
<td>446.20</td>
<td>0.833</td>
<td>0.76</td>
</tr>
<tr>
<td>trivial07</td>
<td>903.00</td>
<td>1104.75</td>
<td>1.32</td>
<td>199.20</td>
<td>320.17</td>
<td>0.614</td>
<td>0.47</td>
</tr>
</tbody>
</table>

**Table 3.1:** Summation of all language feature microbenchmark results. Show first run time, last run time, slope ($m_n$, scaled by $10^{-4}$), and the slope ratio of $m_n/m_1$, where $m_1$ is the slope of ajc ltw. Table 1 compares load time weaving to class weaving, table 2 compares against method weaving. Note that slope is calculated not from $LR - FR$, but from $LR - LR_{-1}$. Where ajvm performance is statistically different from ajc ltw, the slope and benchmark name have been emphasized. (Continued on following page)
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ajc ltw FR</th>
<th>ajc ltw LR</th>
<th>m&lt;sub&gt;1&lt;/sub&gt;</th>
<th>ajvm method FR</th>
<th>ajvm method LR</th>
<th>m&lt;sub&gt;3&lt;/sub&gt;</th>
<th>m&lt;sub&gt;3&lt;/sub&gt;/m&lt;sub&gt;1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpbeforecall</td>
<td>969.25</td>
<td>1836.73</td>
<td>0.426</td>
<td>190.33</td>
<td>1035.00</td>
<td>4.00</td>
<td>0.94</td>
</tr>
<tr>
<td>interpflownoState</td>
<td>994.60</td>
<td>2306.31</td>
<td>14.2</td>
<td>192.66</td>
<td>594.64</td>
<td>3.02</td>
<td>0.21</td>
</tr>
<tr>
<td>interpstate01</td>
<td>986.40</td>
<td>1177.13</td>
<td>0.538</td>
<td>190.33</td>
<td>383.31</td>
<td>0.620</td>
<td>1.15</td>
</tr>
<tr>
<td>interpstate02</td>
<td>999.00</td>
<td>1113.37</td>
<td>3.27</td>
<td>193.50</td>
<td>323.25</td>
<td>3.83</td>
<td>1.17</td>
</tr>
<tr>
<td>interpunadvised</td>
<td>849.00</td>
<td>1058.60</td>
<td>0.342</td>
<td>188.00</td>
<td>405.29</td>
<td>0.359</td>
<td>1.05</td>
</tr>
<tr>
<td>cflow01</td>
<td>996.75</td>
<td>2304.16</td>
<td>11.7</td>
<td>191.66</td>
<td>587.75</td>
<td>2.90</td>
<td>0.25</td>
</tr>
<tr>
<td>thisTargetArgs01</td>
<td>992.50</td>
<td>1175.58</td>
<td>0.392</td>
<td>195.00</td>
<td>386.92</td>
<td>0.747</td>
<td>1.91</td>
</tr>
<tr>
<td>thisTargetArgs02</td>
<td>990.50</td>
<td>1230.46</td>
<td>0.854</td>
<td>195.50</td>
<td>502.51</td>
<td>0.197</td>
<td>2.31</td>
</tr>
<tr>
<td>thisTargetArgs03</td>
<td>998.50</td>
<td>1304.70</td>
<td>2.20</td>
<td>190.00</td>
<td>387.34</td>
<td>0.744</td>
<td>0.34</td>
</tr>
<tr>
<td>thisTargetArgs04</td>
<td>1001.16</td>
<td>1345.17</td>
<td>2.61</td>
<td>196.00</td>
<td>516.92</td>
<td>1.99</td>
<td>0.76</td>
</tr>
<tr>
<td>thisTargetArgs05</td>
<td>990.25</td>
<td>1244.35</td>
<td>1.52</td>
<td>193.50</td>
<td>388.39</td>
<td>6.96</td>
<td>0.46</td>
</tr>
<tr>
<td>thisTargetArgs06</td>
<td>994.00</td>
<td>1265.57</td>
<td>1.34</td>
<td>192.00</td>
<td>465.14</td>
<td>1.38</td>
<td>1.03</td>
</tr>
<tr>
<td>thisTargetArgs08</td>
<td>988.20</td>
<td>1174.26</td>
<td>0.445</td>
<td>193.00</td>
<td>422.90</td>
<td>0.73</td>
<td>1.67</td>
</tr>
<tr>
<td>thisTargetArgs10</td>
<td>1001.00</td>
<td>1280.19</td>
<td>1.74</td>
<td>193.66</td>
<td>434.96</td>
<td>0.740</td>
<td>0.43</td>
</tr>
<tr>
<td>thisTargetArgs12</td>
<td>989.00</td>
<td>1243.23</td>
<td>1.36</td>
<td>195.00</td>
<td>449.25</td>
<td>1.73</td>
<td>1.28</td>
</tr>
<tr>
<td>thisTargetArgs14</td>
<td>992.00</td>
<td>1229.81</td>
<td>0.819</td>
<td>193.50</td>
<td>475.26</td>
<td>1.54</td>
<td>1.88</td>
</tr>
<tr>
<td>thisTargetArgs16</td>
<td>996.25</td>
<td>1317.15</td>
<td>2.23</td>
<td>194.00</td>
<td>486.24</td>
<td>1.55</td>
<td>0.69</td>
</tr>
<tr>
<td>thisTargetArgs18</td>
<td>992.00</td>
<td>1252.20</td>
<td>1.64</td>
<td>193.66</td>
<td>453.50</td>
<td>1.51</td>
<td>0.92</td>
</tr>
<tr>
<td>trivial01</td>
<td>966.66</td>
<td>1154.66</td>
<td>0.464</td>
<td>198.28</td>
<td>378.33</td>
<td>0.510</td>
<td>1.10</td>
</tr>
<tr>
<td>trivial03</td>
<td>900.00</td>
<td>1044.00</td>
<td>0.478</td>
<td>194.50</td>
<td>345.52</td>
<td>0.603</td>
<td>1.26</td>
</tr>
<tr>
<td>trivial05</td>
<td>979.50</td>
<td>1235.84</td>
<td>1.06</td>
<td>195.25</td>
<td>438.99</td>
<td>0.799</td>
<td>0.76</td>
</tr>
<tr>
<td>trivial07</td>
<td>903.00</td>
<td>1104.75</td>
<td>1.32</td>
<td>193.25</td>
<td>322.23</td>
<td>0.761</td>
<td>0.58</td>
</tr>
</tbody>
</table>

**Table 3.2:** Summation of all language feature microbenchmark results (continued).
3.3 Startup Microbenchmarks

As discussed in [1,2] microbenchmarks are a very poor predictor for the performance of a given VM on complex, real-world applications; extrapolating overall performance on real-world code based on such benchmarks would be a serious mistake, allowing for the possibility of significant error. However, it is still possible to confirm performance predictions and isolate elements of overall performance using targeted microbenchmarks. We used several such benchmarks to confirm and clarify our picture of VM performance.

Of primary interest are the four benchmarks included in 3.2. Each was selected to illustrate an aspect of relative performance of the different implementations under consideration.

staticLoad measures the startup performance of each VM for a trivial one-operation task. The Java code comprising the benchmark is compiled, and an empty AspectJ file is included; this forces each of the lazy weavers to initialize and check for aspects, but does not necessitate loading, processing, or data structure initialization associated with aspects, thus exposing the fundamental cost of initialization for each VM.
**staticLoadWithInfr** is identical to **staticLoad**, except that the included aspect file is not empty; rather, it contains a single empty aspect. This forces each of the late weavers to initialize data structures to load the empty aspect; however, no matching or weaving will occur. This isolates the cost of VM initialization, plus the cost of initializing aspect-related data structures.

**1000MethodsLoaded** loads a large amount of simple code, in the form of 100 classes consisting of 10 simple methods each, with an aspect containing *before* advice which applies to a single method in each class. For each class, a single instance is created. The approaches which weave at class load-time must perform scanning and weaving on all classes, but the method-level lazy weaver need only weave those methods which are executed.

**500MethodsExec** is identical to **1000MethodsLoaded**, except that half of the loaded methods in each class, totalling 500, are executed. Again, the class load time weavers must perform all scanning and weaving at load time, whereas the method-level lazy weaver only scans those methods which are executed—roughly half of all loaded code.

The aspect used in both **1000MethodsLoaded** and **500MethodsExec** applies only to a single method in each class, and consists of a single integer operation followed by a test to prevent the JIT optimizer from removing the associated bytecode.

We executed each of these four benchmarks on the following four VMs.

- **ajc** Aspects are woven at compile-time, and executed on an unmodified JVM (Jikes 3.1.0); this performance can be considered ideal. All classes must be woven.

- **ajc ltw** Aspects are woven, external to the VM, at class load time; the resulting code is executed on an unmodified JVM. Only loaded classes are woven.

- **ajvm - class level** Aspects are woven within the modified VM at class load time; this approach is very similar to the **ajc** load time weaver, but internal to the VM. Only loaded classes are woven.
**ajvm - method level** Aspects are woven on a per-method basis, immediately prior to execution. Only executed methods are woven.

Each of the two AJVM implementations has two variants, with and without Type Pattern Tables (TPTs), which are intended to improve aspect matching performance.

![Figure 3.2: Startup Microbenchmarks](image)

### 3.4 Startup Macrobenchmarks - DaCapo

While the startup microbenchmarks in 3.3 are useful for isolating elements of startup performance, using them to extrapolate performance on real-world applications might lead to very inaccurate conclusions.

Our ideal target benchmark would be a large, industry-standard application making systemic use AspectJ language features. Since no such application exists, we opted to use industry standard benchmarks to give us a more realistic picture of the relative performance of different aspect weaving approaches on real-world code.

Section 1.2 discusses in detail some of the pitfalls and complications involved in benchmarking workload. Since our ideal benchmark was not available, and in order to maximize the relevance of our comparisons, we wished to remain as close as possible to industry-standard benchmarks.
We chose the DaCapo benchmarks because of their emphasis on real-world code and complex workloads. The authors measured their benchmarks according to the metrics presented in [7], according to which the benchmarks closely resemble similar real-world code, in terms of code complexity, structure, hierarchy, size, and so on. According to these metrics, other benchmark suites, including the popular SPEC benchmark suite, often fall far short of real-world code in complexity and depth. Because of the complexity of the startup workload, these metrics are of particular importance to us.

In [4], the authors compare the performance of the DaCapo suite with that of the SPEC suite, and discover that the DaCapo benchmarks, and particularly the eclipse benchmark, are significantly slower to warm up; that is, the JIT-optimizer takes longer to reach a steady state on the DaCapo benchmarks, due to their increased complexity. The DaCapo benchmarks have more classes, methods, and bytecode, and the proportion of executed and ‘hot’ methods (frequently-executed methods) is much smaller in the DaCapo suite than in the SPEC suite\(^1\). Again, this is more similar to real-world code. Table 3.3 illustrates this point, showing the amount of code loaded, executed, and JIT-optimized (or ‘hot’) in the DaCapo benchmarks and in actual startup of Eclipse and Tomcat.

In measured memory use, the difference between the two suites is even more significant, with the average DaCapo benchmark allocating more than 10 times as much memory. These allocations also tend to be smaller, meaning that from the perspective of AspectJ, that there may be significantly more join points in the same bytecode.

The eclipse benchmark stands out among the DaCapo benchmarks as particularly complex; it has more methods and classes, both absolutely and relative to the size of it’s own bytecode, than other benchmarks. It has a relatively high cache miss rate, and the proportion of optimized and ‘hot’ methods is relatively low compared to loaded and executed code. It allocates and uses more memory than average for the DaCapo benchmarks, and dramatically more than the SPEC benchmarks. In all of these measurements, it resembles the real Eclipse startup workload, and differs from typical benchmark workloads.

Since our benchmark target workload is, generally, the startup of a complex real-world program, and in particular the Eclipse startup workload, we focused particularly on the DaCapo eclipse benchmark. For this reason, we

\(^1\)See [4], table 3.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th># loaded methods</th>
<th># executed methods (exec/loaded)</th>
<th># JIT compiled methods (compiled/exec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>81663</td>
<td>35445 (43.4%)</td>
<td>3386 (9.5%)</td>
</tr>
<tr>
<td>Tomcat</td>
<td>13797</td>
<td>6380 (46.2%)</td>
<td>584 (9.2%)</td>
</tr>
<tr>
<td>antlr</td>
<td>6122</td>
<td>3438 (56.2%)</td>
<td>851 (24.8%)</td>
</tr>
<tr>
<td>avrora</td>
<td>7279</td>
<td>3956 (54.3%)</td>
<td>659 (16.7%)</td>
</tr>
<tr>
<td>bloat</td>
<td>7727</td>
<td>4105 (53.1%)</td>
<td>888 (21.6%)</td>
</tr>
<tr>
<td>chant</td>
<td>2156</td>
<td>1111 (51.5%)</td>
<td>29 (2.6%)</td>
</tr>
<tr>
<td>derby</td>
<td>18892</td>
<td>8188 (43.3%)</td>
<td>1805 (22.0%)</td>
</tr>
<tr>
<td>eclipse</td>
<td>9591</td>
<td>5076 (52.9%)</td>
<td>788 (15.5%)</td>
</tr>
<tr>
<td>hsqldb</td>
<td>8140</td>
<td>3555 (43.7%)</td>
<td>585 (16.5%)</td>
</tr>
<tr>
<td>jython</td>
<td>11369</td>
<td>5382 (47.3%)</td>
<td>1337 (24.8%)</td>
</tr>
<tr>
<td>luindex</td>
<td>5835</td>
<td>2999 (51.4%)</td>
<td>867 (28.9%)</td>
</tr>
<tr>
<td>lusearch</td>
<td>5746</td>
<td>2934 (51.1%)</td>
<td>498 (17.0%)</td>
</tr>
<tr>
<td>pmd</td>
<td>9388</td>
<td>4573 (48.7%)</td>
<td>1059 (23.2%)</td>
</tr>
<tr>
<td>xalan</td>
<td>2596</td>
<td>1370 (52.8%)</td>
<td>35 (2.6%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>190301</strong></td>
<td><strong>88512 (46.5%)</strong></td>
<td><strong>13371 (15.1%)</strong></td>
</tr>
</tbody>
</table>

Table 3.3: Summary of startup characteristics of Eclipse, Tomcat and the DaCapo benchmark suite. Shows the number of methods loaded, executed, and compiled along with the ratio of executed to loaded methods, and compiled to executed methods for each benchmark. These numbers were obtained using JVMTI on IBM J9 VM.

performed a more thorough analysis on the eclipse benchmark, and compared the resulting figures with Eclipse startup itself, as measured on the Sun Hotspot VM. All figures were captured with the JVMTI interface.

We discovered the following:

1. DaCapo’s eclipse benchmark loads 80% as much code as Eclipse startup, of which 40% is shared between the two. Most of the remaining code is Eclipse code or Java libraries, which the benchmark exercises. The Eclipse GUI accounts for much of the code loaded by Eclipse but not by the DaCapo benchmark.
2. The benchmark executes 48% of loaded methods, as compared to 44.3% by Eclipse. The total size, in bytes, of executed bytecode (prior to JIT-compilation) differs by less than 3%. Executed code accounts for 49% of loaded code in Eclipse, and 59% of code loaded by the benchmark.

3. 12.9% of the benchmark’s code is JIT-optimized, as opposed to only 2.4% of Eclipse code; in fact, the top 10 most executed method in the benchmark account for 90% of bytecode executed (as opposed to only 53% in the case of Eclipse startup).

4. The benchmark’s execution time is approximately 10 times as long as Eclipse startup, due to repetitive, long-running benchmark methods.

More detail on the comparison can be found in Appendix A.

Since all of the VMs tested are variants of JikesRVM, we expect the long term steady-state performance of all variants to converge; as we will see in 3.5, the microbenchmark results confirm that the bytecode generated by each aspect weaver tends to converge in performance, before and after JIT-optimization, across VMs. In any case, our modifications to the DaCapo benchmarks introduced only minimal bytecode changes.

Thus, the difference in execution time between the DaCapo eclipse benchmark and Eclipse startup is not a significant problem, and can mostly
be ignored, because most of this time is spent repeatedly executing a small number of methods. This cost will be shared equally between all implementations. More important, to us, is the amount of loaded and executed code. It is this difference that will most significantly affect the difference in performance, since the tested VMs differ at the point of class loading and method execution. Here, we find that the amount of loaded code is similar, and the amount of executed code is very close. Given that much of the code is identical, and the code is similar in method size, class size, method and class count, and other metrics, we can presume that the density of join points is similar.

Given that the VMs share most of their code, differing only in their approach to weaving, we can say with confidence that, variance aside, the difference in performance occurs almost exclusively at the point of class loading and method execution, and any difference must be attributable to the modifications in the VMs. Finally, given the similarities above, we would expect a very similar difference in performance for the normal Eclipse startup workload. Given that Eclipse loads about 25% more code, and executes a slightly smaller proportion (44.3% as opposed to 48.0%), we would actually expect the difference in performance between the lazy (or method level) weaving approach and the other, class-level approaches to grow slightly for actual Eclipse startup, while the relative performance between the VM-internal and -external class-level weavers should remain unchanged, as they would perform

---

**Figure 3.4:** DaCapo - Non-Matching Aspect
the same work.

3.5 Results and Interpretation

Our approach to benchmarking application startup consists of several parts: measuring AspectJ language feature performance with microbenchmarks, isolating components of startup performance with synthetic startup benchmarks, verifying the performance of unmodified Java code with DaCapo macrobenchmarks, and estimating the performance of complex application startup, particularly Eclipse, in presence of aspects, with the DaCapo benchmarks modified to include some simple aspects.

The results from the AspectJ language feature microbenchmarks clearly confirmed that the bytecode generated by all VM implementations was equivalent, from a performance perspective. This was true both before and after JIT-optimization. The only exceptions occurred, as expected, where an in-VM approach permitted optimization. There remains the possibility that performance may diverge in special cases or for more complex aspects; however, we believe this is unlikely.

Our synthetic startup benchmarks (Fig 3.2) allow us to isolate components of the startup workload and estimate their impact. Initialization of the VM does vary; our in-VM approaches, including the aspect-weaving infrastructure (see 3.3) take 1.9 times as long as an unmodified JikesRVM to load. Both the unmodified JVM and our implementations are significantly faster to load than JikesRVM with the AJC VM-external aspect weaver, which takes 5.4 times longer than the unmodified JikesRVM, and 2.9 times longer than our in-VM approaches to load.

The difference in performance between staticLoad and staticLoadWith-Infr illustrate the cost of initializing the aspect-weaving infrastructure; this turns out to be insignificant for the in-VM approaches (and nonexistent for the unmodified VM, as it performs no extra initialization), but causes an 84% slowdown for AJC LTW. The ability of the in-VM approaches to reuse VM code and data structures accounts for their relative efficiency in this case.

However, this difference in load time is relatively small compared to the overall execution time of a complex application, and the cost of initializing the aspect weaver varies little relative to the program size; the cost of actual scanning and weaving is much more significant, as the cost scales with the program size.
The next two benchmarks in Fig 3.2 illustrate the cost of scanning and weaving code, and the costs associated with different approaches to weaving. Each VM must first initialize, and thus pay the same startup cost as in \texttt{staticLoadWithInfr}; all additional time is spent on scanning code, weaving, or executing. As established above, generated bytecode is identical in performance, and so time spent in execution, after weaving, should be essentially identical; and, as established by the prewoven ‘ajc’ result, execution accounts for only 180 – 270 ms of the total execution time.

The difference in performance between \texttt{ajc ltw} and \texttt{ajvm-class} is due to the use, by \texttt{ajvm}, of VM-internal data structures. Both are class-level weavers, having similar scanning and weaving performance; however, \texttt{ajc ltw} creates its own data structures for use in weaving, while \texttt{ajvm} uses internal data structures which the VM itself will utilize during execution. As illustrated by the difference in performance between the two, the benefit of sharing the VM’s internal data structures is significant.

The difference between the \texttt{ajvm-class} and \texttt{ajvm-method} benchmark results is due to the amount of code which must be scanned and woven; the class-level weaver scans the entirety of each loaded class, as soon as it is loaded. The method-level weaver performs weaving on a per-method basis, and only when a method is actually executed. In this way, it further amortizes weaving time over execution, and it avoids weaving unused methods. This is clearly seen in \texttt{1000MethodsLoaded}, in which \texttt{ajvm-method} is able to perform almost as well as the pre-woven \texttt{ajc}, simply because it is able to avoid almost all scanning and weaving, as only a tiny fraction of code is actually executed.

In a more realistic case, \texttt{500MethodsExec}, where 50% of the loaded code is executed (in line with both regular startup and the DaCapo benchmarks, seen in 3.4), \texttt{ajvm-method}’s performance suffers significantly relative to \texttt{ajc}, but it is still able to complete the benchmark in 63% of the time of the class-level weaver. This illustrates the potential benefits of a lazy approach to aspect weaving; the performance of the lazy weaver is closely proportional to the amount of code executed by the benchmark. Since the startup workload in most real-world applications executes only a fraction of total code, this suggests that the benefits of lazy weaving could be significant.

The DaCapo benchmarks, however, suggest that the actual benefits of lazy weaving are not as significant as the \texttt{500MethodsExec} benchmark would suggest. The advantage of using internal data structures can still be clearly seen in the difference in performance between \texttt{ajc ltw} and all other implementations.
In the case of the DaCapo eclipse benchmark, it is important to remember that while the amount of code loaded and executed by the benchmark, relative to Eclipse startup, is very similar (at 80% and 103%, respectively), the DaCapo benchmark runs much longer, due to repeated execution of benchmark code. For actual Eclipse startup, we would expect ajc ltw and ajvm-class to perform slightly slower (due to the additional 25% of code loaded) and ajc and ajvm-method to perform almost exactly the same; that is, we would expect the relative performance of the two class-level weavers to suffer compared to the pre-compiled and method-level weavers. However, the total runtime would be reduced to less than $\frac{1}{10}$; this reduction would be shared by all implementations, since it occurs during execution of code after weaving.

An estimate of the startup time of Eclipse in the presence of an equivalent aspect is given in Fig 3.5; this estimate was made by subtracting the execution cost shared by all VM implementations (that is, the runtime of ajc, which performs neither scanning nor weaving at runtime) and adding the approximate execution cost of Eclipse startup, as measured using Sun’s HotSpot VM. Note that since ajc performs scanning at compile time and will find no matching join points and thus perform no weaving, the generated bytecode, and thus the measured runtime, will be identical with or without the chosen aspect. The error remains the same as in the original graph.

![Figure 3.5: Estimated Eclipse Runtime - Non-Matching Aspect](image)

Additionally, the results in 3.4 include only the cost of scanning, and not of weaving; the aspect applies to a non-existent method. In an actual implementation, one would expect a larger number of aspects, and weaving to occur. In the case of the class-level weavers, all loaded code would have to be both scanned and woven; the method-level weaver would avoid
both scanning and weaving unexecuted code. We would therefore expect the implementations to diverge further due to weaving cost.

Based on the observed results, we can clearly state that using VM-internal data structures for scanning and weaving aspects at runtime results in a clear and significant performance gain for startup workloads; according to our estimate above, for the Eclipse startup workload in the presence of a single non-matching aspect causing all code to be scanned, it saves over 20% of total runtime.

The benefits of lazy weaving are less clear from these results; though the potential benefits are clear from our synthetic startup benchmarks (Fig 3.2), the magnitude of the advantage shrinks significantly in the DaCapo microbenchmarks, to the point that the two results are often the same within error, or very similar. However, since the lazy weaver can avoid both scanning and weaving for each aspect included in the system, we would expect the two to diverge in large systems containing many aspects.

Regardless, it is clear from these results that the biggest gain in performance in both real-world and synthetic benchmarks is due to the reuse of existing, internal data structures.
Chapter 4

Threats to Validity

The most significant threat to the validity of our approach to benchmarking the performance of runtime aspect weaving during application startup on various VM implementations is that we were unable to use our ideal benchmark: an industry standard program using aspects in a systemic way. While we believe that our analysis provides enough data to confidently predict how the different implementations would perform on such a benchmark, there remains the possibility that an unforeseen effect might cause performance on real-world applications to degrade due to interaction with the JIT compiler, memory usage patterns, or other problem.

We were able, on all implementations, to directly benchmark:

- Simple AspectJ language features
- Idealized startup benchmarks
- Weaver and data structure initialization
- Complex, real-world benchmarks using unmodified Java code
- Scanning and running of complex, real-world benchmarks

The two most obvious shortcomings with our analysis are:

- The DaCapo benchmarks we used to estimate startup performance are not identical to real startup workloads
- We did not benchmark scanning and weaving in the presence of both complex aspects and complex code
As discussed in §3.4, we believe that the two workloads are similar in most critical aspects. The amount of code loaded and executed is very similar, and the DaCapo benchmarks are subject, at the beginning of their execution, to the idiosyncrasies which characterize the startup workload: heavy allocation with relatively light garbage collection, heavy JIT-compilation and execution of unoptimized bytecode, heavy I/O, and so on. The critical differences occur due to the repeated execution of optimized code on the part of the benchmark. Since this occurs after aspect scanning and weaving has occurred, it will not vary between different implementations, assuming that the generated bytecode is equivalent. We demonstrated that this was indeed the case in §3.2, at least for isolated language features.

A more thorough analysis of the DaCapo eclipse benchmark, as compared to actual Eclipse startup, does reveal some significant differences between the two; this comparison can be found in Appendix A. While we believe these differences are, for the most part, orthogonal to our analysis, it is possible they may play a larger role than we believe.

While we have attempted to capture all of the salient aspects of our ideal workload for benchmarking, there is one which we were unable to target: complex aspects interacting with complex code. Such a benchmark would effectively be our ideal benchmark. As aspect weaving is relatively straightforward and well-understood, and our approach does not significantly differ in its approach to weaving from existing approaches, we believe performance will not suffer due to the interaction of complex aspects and code.

One important reason for the dearth of complex programs using aspects systematically is that the costs of doing so can drastically slow the development process. The cross-cutting nature of aspects means that small changes to an aspect can force the scanning and weaving of a large portion of program code; this is quite unlike modifications to standard Java code which are, by their nature, local. Load-time weaving was itself intended to address this problem. It may be hoped that improved startup performance in presence of aspects might allow for the wider adoption of aspects.

The lack of large programs using aspects systematically even makes it difficult to speak of programs using aspects in a ‘typical’ way, as it is not clear what use is typical, or what would be typical if not for performance constraints. Even were this not the case, aspects allow for many possible decompositions of problems domains, so such targets remain elusive. It is for this reason that we focused on a simple aspect which forced all executed code to be scanned, rather than attempting to create a typical aspect for the
DaCapo benchmarks; this gives a reliable and reproduceable result, which is relevant to all applications and problem domain decompositions.
Chapter 5

Comments on Alternative Aspect Approaches

This paper has focused on contrasting the VM-internal AspectJ weaving approaches introduced in [13] with the AspectJ load-time and compile-time weavers. While there are other aspect weaving approaches for AspectJ and AspectJ-like languages, their goals and implementation were largely orthogonal to ours, limiting the use of direct comparisons. In fact, several such approaches could complement the in-VM weaving and late weaving examined herein.

Our particular focus on startup performance sets our work apart in aspect-related research; typically, it is steady-state programming which is targeted instead. This is true of Steamloom and IBM’s J9 VM.

Several research efforts have focused on run-time aspect weaving, but these tend to focus on dynamic loading and unloading of aspects. JAsCo, ALIA, PROSE and Nu fit into this category. While these might see the same benefits of in-VM weaving as the approaches above, performance is not a primary goal of these implementations, and due to the requirements of dynamic loading and unloading, the languages implemented are substantially different than AspectJ.

In [13], we also introduced Immad Nassir’s interpreter-based approach to run-time aspect weaving; the intention was to minimize startup time by using the interpreter for initial weaving to avoid initialization cost before switching to a JIT weaver for better steady-state performance. Our benchmark comparisons suggested that it was doubtful that an interpreter-based approach could rival the performance of a lazy weaver. Due to the fact that
this approach does not currently support the JIT compiler, a fair comparison on more complex benchmarks is not feasible.
Chapter 6

Future Work

As discussed in the introduction, startup performance is an important, and frequently neglected, aspect of overall performance; this is especially true during the development cycle, when an application may be launched repeatedly. This paper has emphasized the need for benchmarks which specifically target the startup performance of complex applications, focusing on a realistic workload. The startup workload is sufficiently distinct from steady-state that we believe it warrants customized benchmarks.

Likewise, the lack of comprehensive AspectJ benchmarks makes it difficult to quantitatively compare different AspectJ compiler and weaver approaches. While microbenchmarks are sufficient for simple language feature comparisons, extrapolation from such basic benchmarks is problematic, as discussed in 1.2. A suite of complex, real-world benchmarks using aspects systematically, targeting AspectJ implementations, would be very helpful for more substantive comparisons. However, as discussed above, it would be difficult to synthesize such benchmarks, so this might first require ‘typical’ large AspectJ programs.
Chapter 7

Conclusion

This thesis has identified and examined real, quantifiable differences in the performance of different approaches to runtime aspect weaving of AspectJ code for Java virtual machines.

We clearly showed, with a combination of microbenchmarks, simple synthetic benchmarks, and large benchmark utilizing real-world code that there is a significant performance difference between VM-internal and VM-external approaches to runtime weaving. By isolating the components of VM startup, we showed that this difference is primarily due to the initialization of external data structures, which outweighs other performance factors.

We also showed a perceptible, but less clear, difference in performance between class-level and method level late aspect weaving in realistic benchmarks in which only a fraction of all loaded code is executed.
Bibliography


Appendix A

Eclipse and DaCapo eclipse

The following table includes some key values from our analysis of Eclipse startup and the DaCapo eclipse benchmark. Though they differ significantly in some ways, the amount of code loaded and executed, the two most important values in our case, are very similar. Generally, the benchmark loads less code, uses more of loaded code, and uses some methods (those being exercised by the benchmark) extensively. The benchmark runs for approximately 10 times as long as Eclipse startup; this time is mostly spent executing a small number of methods.

Note that measurement of bytes gives only a rough idea of execution time; most measurements are of method size prior to JIT compilation. However, these values are accurate with respect to weaving; the number of bytes loaded and executed are exactly the number that require scanning in the class-level and method-level VMs, respectively.

These figures were gathered using Sun's HotSpot Client VM (build 17.1-b03) using the JVMTI application programming interface (API).
Table A.1: Measurements from Eclipse startup and DaCapo *eclipse* benchmark

<table>
<thead>
<tr>
<th>Measurement</th>
<th>eclipse</th>
<th>dacapo-eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>% exec (methods)</td>
<td>44.3%</td>
<td>48.0%</td>
</tr>
<tr>
<td>% exec (bytes)</td>
<td>49.1%</td>
<td>59.7%</td>
</tr>
<tr>
<td>% JIT (methods)</td>
<td>2.4%</td>
<td>12.9%</td>
</tr>
<tr>
<td>% JIT (bytes)</td>
<td>3.5%</td>
<td>20.0%</td>
</tr>
<tr>
<td>Code executed (bytes)</td>
<td>1020701</td>
<td>1049755</td>
</tr>
<tr>
<td>Size (total bytes, no jit)</td>
<td>2135529</td>
<td>1709476</td>
</tr>
<tr>
<td>Relative size</td>
<td>125%</td>
<td>80.0%</td>
</tr>
<tr>
<td>Shared methods</td>
<td>22.2%</td>
<td>37.8%</td>
</tr>
<tr>
<td>Shared called methods</td>
<td>21.4%</td>
<td>39.5%</td>
</tr>
<tr>
<td>Total methods</td>
<td>42685</td>
<td>25085</td>
</tr>
<tr>
<td>Total methods exec</td>
<td>18909</td>
<td>12041</td>
</tr>
<tr>
<td>Total classes</td>
<td>4294</td>
<td>2262</td>
</tr>
<tr>
<td>Shared classes</td>
<td>964</td>
<td>964</td>
</tr>
<tr>
<td>Shared classes %</td>
<td>22.4%</td>
<td>42.6%</td>
</tr>
<tr>
<td>Avg method size (loaded)</td>
<td>50.0 B</td>
<td>68.1 B</td>
</tr>
<tr>
<td>Avg JIT method size</td>
<td>72.8 B</td>
<td>105.4 B</td>
</tr>
<tr>
<td>Avg non-JIT (but executed) method size</td>
<td>57.6 B</td>
<td>71.8 B</td>
</tr>
<tr>
<td>Avg methods per class</td>
<td>9.9</td>
<td>11.1</td>
</tr>
<tr>
<td>Avg methods per class (executed)</td>
<td>5.2</td>
<td>6.3</td>
</tr>
<tr>
<td>Avg methods per used class</td>
<td>10.8</td>
<td>11.9</td>
</tr>
<tr>
<td>Top 10 method bytecode / total</td>
<td>52.5%</td>
<td>89.9%</td>
</tr>
<tr>
<td>% bytecode executed</td>
<td>49.1%</td>
<td>59.7%</td>
</tr>
<tr>
<td>% exec bytecode jitted</td>
<td>7.1%</td>
<td>33.5%</td>
</tr>
</tbody>
</table>