

# The Case for Multiple Compilers

David Detlefs and Ole Agesen

Sun Microsystems Laboratories\*  
1 Network Drive  
Burlington, MA 01803-0902, USA  
{david.detlefs,ole.agesen}@sun.com

## 1 Introduction

For virtual machine implementations to achieve high performance, some form of translation of the virtual machine's input language into the native code of the host machine seems necessary. This translation process is often called *just-in-time* (a.k.a. *JIT*) compilation, or sometimes *dynamic* compilation. The use of JIT compilation introduces a tension in virtual machine design: compilation time adds to the run time of the application, so compilation must be fast, but minimizing compilation time makes it difficult to generate excellent code. In this paper we present measurements that quantify this tradeoff, and suggest an approach that satisfies both concerns.

## 2 Different Compilers

JIT compilers form a continuum, from very fast compilers producing mediocre code to slow compilers producing highly optimized code. Still, it is possible to divide this continuum roughly in two, calling one side *true JITs* and the other *traditional compilers*. This naming conveys the generally true fact that fast JIT compilers are written from scratch for use as dynamic compilers in virtual machines. Compilation speed is usually the paramount concern for such compilers. On the other hand, many of the slower, highly optimizing compilers in use in VM's are actually traditional static compilers adapted and pressed into use as JIT compilers.

The continuum continues, of course, within these categories. Template-based JIT compilers produce the a fixed code pattern for each virtual machine byte-code, doing almost no optimizations but running extremely fast. More aggressive JIT compilers introduce some optimizations, but only those that preserve compilation speed. Often this means performing only optimization that can be accomplished in a single linear code-generation pass. Our measurements will include

---

\* Sun, Sun Microsystems, Java, HotSpot, Solaris, and Ultra are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

such a compiler, and will show that they can sometimes achieve surprisingly good performance in a limited compilation budget.

The idea of deploying a traditional compiler as a dynamic JIT compiler would have been unthinkable five years ago. However, the inexorable march of Moore's law has made compilation fast enough to be considered acceptable overhead in the execution of at least some applications. But not all applications; some argue for the traditional static compilation model, saying that for many applications there is little good reason to pay any compilation overhead on every execution. We won't attempt to settle that issue, but rather argue that appropriate dynamic compilation techniques can approach the performance of static compilers with minimal compilation overhead.

### 3 Measurements

The meat of this paper is in figure 1. Each point in this figure represents a set of runs of the programs in the SPECjvm98 benchmark suite with a given execution configuration of an experimental version of the Sun Java<sup>®</sup>2 SDK for the Solaris<sup>™</sup> operating system, Production Release.<sup>1</sup> The runs were performed on a Sun Ultra<sup>™</sup> 60 Creator 3D desktop, which has 2 360 MHz UltraSPARC<sup>™</sup> II processors.

There are seven programs in this suite. Each program was run three times in succession on the same input set, in one Java virtual machine process. Thus, the first run of each sequence includes any JIT compilation time, while the second and third (generally) do not, since compilation has already been done. These runs yield a "first" and a "best" time. We convert each time into the corresponding "spec ratio," the ratio of the execution speed implied by the time to that of a reference platform. For example, if a benchmark ran in 12.6 seconds, and the reference platform time for that benchmark was 380 seconds, then the spec ratio of that run is 380/12.6, or 30.2. For each program, then, we have a first and best spec ratio. These numbers are combined across a suite of programs by taking geometric means. Figure 1 uses the combined numbers for the seven programs of the benchmark suite.

Each point, therefore, has a first-run spec ratio as its x-coordinate, and a best-run spec ratio as its y-coordinate. All points are obviously above the 45° line that represents the family of execution engines with infinite compilation speed. The farther from this line, the greater the difference between first and best runs.

There are many execution configurations. The underlying Java virtual machine has three modes of execution: an interpreter, a fairly sophisticated JIT compiler that produces relatively good code in a tight compile-time budget (referred to henceforth as **fast**), and a traditional compiler adapted for use as a JIT compiler (henceforth **opt**). Here are the configurations:

---

<sup>1</sup> The non-experimental version of this system is available at <http://www.sun.com/solaris/java>.

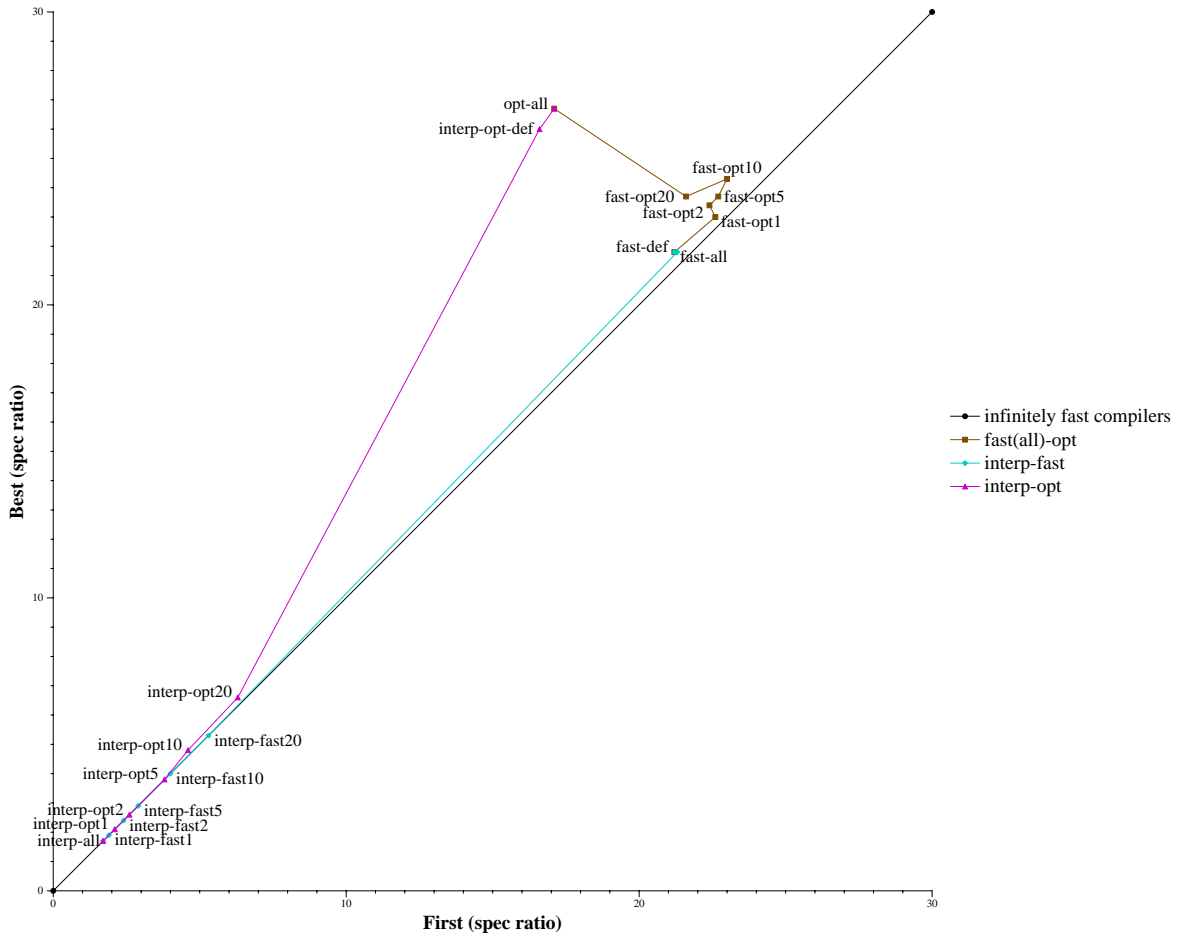


Fig. 1. SPECjvm98 runs: different compilation configurations

- **interp-all**  
No compilation is used; all execution is via the interpreter.
- **interp- $\{\mathbf{fast}, \mathbf{opt}\}n$**   
Compile the top  $n$  methods (as determined by a cpu sampling of run using the **fast** compiler) using the indicated compiler. Interpret all other methods.
- **$\{\mathbf{fast}, \mathbf{opt}\}$ -def**  
Use the indicated compiler to compile all loop-containing methods at their first invocation, and non-loop-containing methods at their 15<sup>th</sup> invocation. **Fast-def** is the default configuration of the released system.
- **fast-opt $n$**   
Compile all methods with the **fast** compiler, except for the top  $n$  methods, which are compiled with **opt**.

- **opt-all**  
**opt**-compile all methods before first execution.

The lines connect three families of runs, two trading off interpreted and compiled execution, and one balancing **opt** execution against **fast** execution. We draw several conclusions from this graph:

- The difference between the speed of compiled and the speed of interpretation is large; a large number of methods must be compiled to prevent the overhead of interpreting the remainder from swamping the performance gains from compilation. This is shown by the difference in speed between compiling the top 20 methods, and compiling all methods, with either compiler. (This system, because of its fast JIT, has deliberately chosen to use a relatively slow interpreter written in a portable high-level language. We believe, however, that even a doubling of interpretation speed would not materially change these results).
- When a large number of methods are compiled with a slow compiler, the compilation time becomes a significant fraction of first-run time. This is shown by the distance of the **opt-all** point from the diagonal.
- A compiler like **fast** can produce best scores within 25% of the best scores of a highly optimizing compiler. Selective application of the highly optimizing compiler can achieve most of the remaining performance difference, while keeping compilation time low: see the point **fast-opt10**.

## 4 Different applications

The measurements of section 3 present the composite of several applications. Applications can have very different characteristics: some have very “steep” profiles, with almost all execution time spent in a handful of methods, and others have very flat profiles, with execution time distributed approximately evenly over a large number of methods. Below we discuss results for individual benchmarks that illustrate these differences, and their effects on compilation tradeoffs.

Figure 2 shows the same graph as figure 1, but restricted to the **compress** benchmark. This benchmark executes a fairly small amount of code, allowing all the code to be compiled by **opt** without impacting total runtime too adversely. Further, it has a steep profile within the code it executes: optimizing just the single top method achieves nearly peak performance.

In contrast, the **javac** benchmark, whose results are shown in figure 3, executes a relatively large number of methods, and has a flat profile. It is significantly more expensive to use **opt** on first runs of this program than it is to use **fast**. For both benchmarks, however, using **fast** as the default and applying **opt** to a small number of important methods yields near-maximal performance with small compile-time penalties.

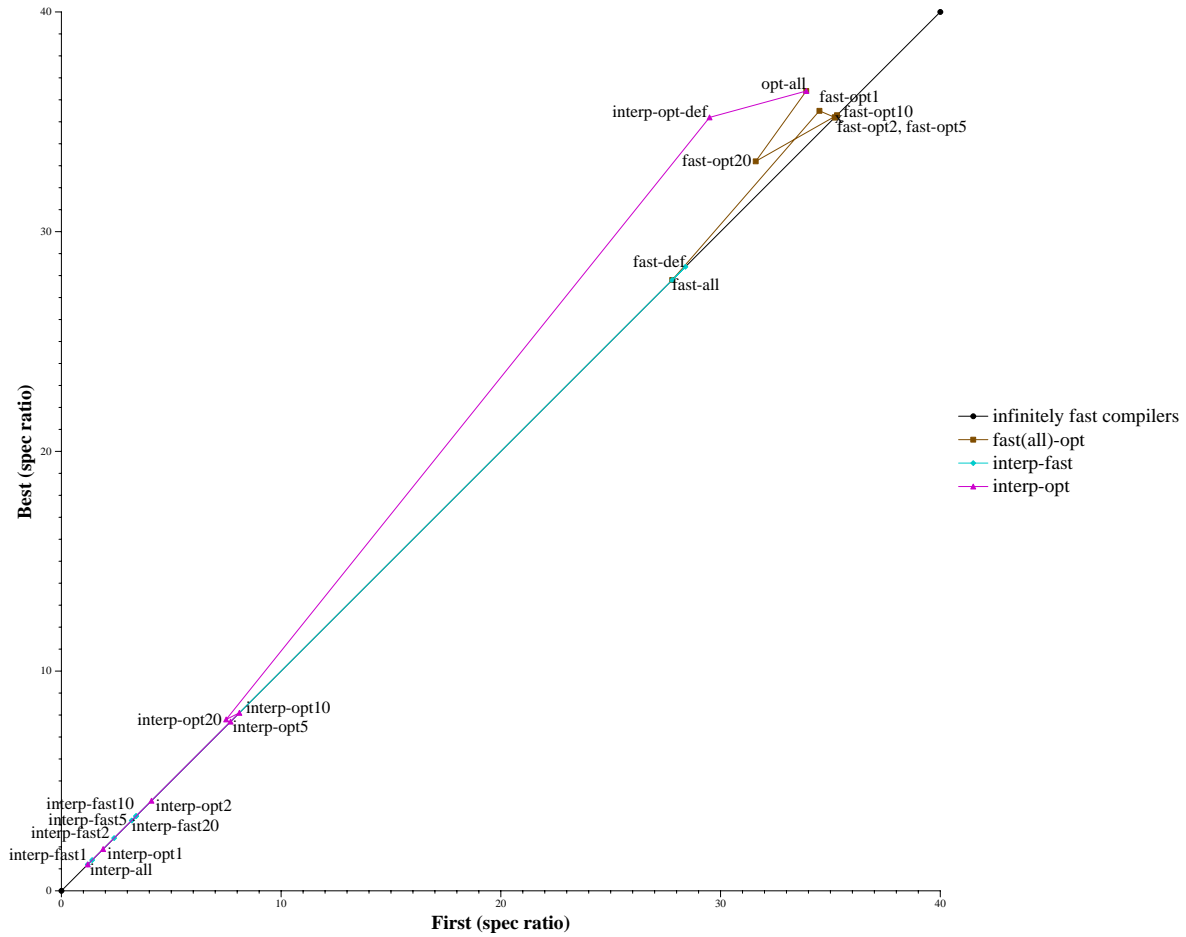


Fig. 2. SPECjvm98 \_201\_compress benchmark: different compilation configurations

## 5 Counterarguments

Our colleague Cliff Click made an excellent observation about these measurements. He noted that the top 20 methods of the **compress** benchmark account for well in excess of 99% of the cumulative cycles, so even if one assumed compiled code to be 50 times faster than interpreted code, compiling the top 20 methods should still yield at least  $2/3$  the fully-compiled best-run speed, while we observe much less than that, approximately  $1/4$  the fully-compiled best-run speed. This led to a small experiment pointing out a component of the cost we had neglected: the cost of transitions from compiled to interpreted code. We created a small program where one method called another small method in a tight loop, and restricted the system to compile the caller containing the loop, but interpret the callee. On our system, this configuration was approximately

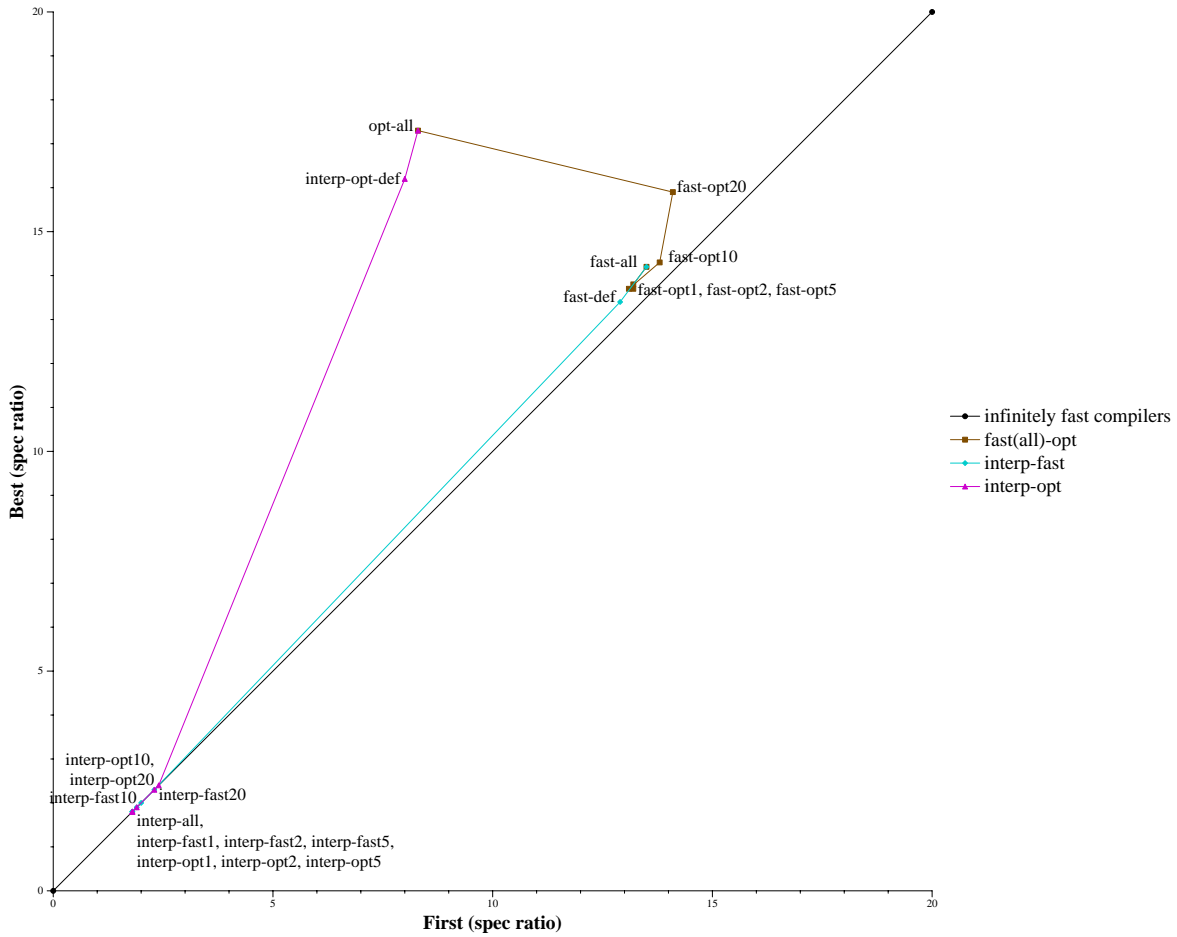


Fig. 3. SPECjvm98 \_213\_javac benchmark: different compilation configurations

three times as expensive as interpreting both methods! The slowdown is due to the cost of entering and exiting the interpreter loop.

In a system such as ours with a fast JIT, where almost all bytecodes are executed in compiled form, this transition is not taken very often, so it is not one that we have optimized well. Indeed, comparisons with a system based on the Java HotSpot™ performance engine on this simple experiment show a difference of approximately a factor of five in cost of interpreter entry. This casts some doubts on our arguments: the poor performance of our system in **interp-opt** mode could be blamed on excessive cost for interpreter entry. There is some validity to this argument, but not enough to negate our basic point. We performed further measurements using the Java HotSpot performance engine as an example of an **interp-opt** configuration with an optimized compiled-to-interpreted path. We indeed find that compiling the top 20 methods achieves peak performance

(in fact, just the top 10 suffice), with little compilation overhead. However, other programs have flatter profiles. Running the **javac** benchmark, for example, this system achieves only about 1/4 of peak performance when compiling only the top 20 methods. And achieving peak performance has an associated compilation cost: the official SPECjvm98 submission for this system for **javac** shows a worst-run (i.e., first-run) speed only 53% the best-run speed.<sup>2</sup> The ratio for the suite as a whole is 72%. These numbers are by no means atypical of high-performance systems: the IBM system with the highest current official SPEC rating has a worst-run/best-run speed ratio of 67% for **javac**, and 78% for the entire suite.

## 6 Related work

Any discussion of dynamic compilation techniques must acknowledge the early work in the Smalltalk community, such as Deutsch and Schiffman's [2]. The Self project at Stanford [3] continued investigation of these issues. Obviously, many groups are building new or improving existing static and dynamic compilers for Java virtual machines; we will make no attempt to list them all. Very little of this work, however, is investigating the mixed use of compilers at different ends of the speed/code-quality continuum. One exception is the Jalapeño compiler project at IBM Research [1]. However, available descriptions of this work include only measurement of "best" times, not of "first" times (to use the terminology of the current paper), making it difficult to evaluate the extent to which they are able to simultaneously optimize both measures.

## 7 Conclusions/Opinions/Future work

For some applications, such as server applications that run (if fate is kind) for weeks at a time, compilation time (within reason) doesn't matter, since the application will run long enough to amortize even a small improvement in code quality. For other applications, code quality doesn't matter: speed beyond some minimum threshold is irrelevant, and perhaps even pure interpretation will be sufficient. Embedded real-time systems, and some simple applications such as editors or spreadsheets on sufficiently fast processors may fall in this category. We claim, however, that there are many applications between these two extremes, for which both compilation speed and code quality matter significantly.

No doubt compilation technology, driven by market pressures, will continue to improve in all dimensions. JIT compilers will gain new optimization techniques that will increase their code quality but make them a little slower. New attention will be paid to speeding up traditional compilers deployed in JIT settings, decreasing their compilation-time cost. We predict, however, that such efforts will reach a point of diminishing returns – the two kinds of compilers won't "meet in the middle." A system with only a JIT-like compiler will always trail a system with a full-blown traditional compiler in code quality, but a system

---

<sup>2</sup> See <http://www.spec.org/osg/jvm98/results/res99q2/jvm98-19990505-03292.g.html>

with only an expensive compiler will impose significant compilation overhead on short-running applications.

We believe that the right solution is to mix the two approaches in a near-optimal way. If the expensive compiler is applied to only a small number of the most important methods, and the fast compiler is applied to the rest, we get much of the benefit of the expensive compiler with little of the cost. In fact, we believe that this approach will approximate the execution speed of statically compiled code with little compilation overhead.

Of course, the measurements in this paper assumed an “oracle” (in the form of profile-feedback) that identified the most important methods to compile with an expensive compiler. The problem of choosing these important methods dynamically, without unduly slowing down execution speed with extra instrumentation, remains an interesting one. Another interesting problem is that of replacing the compiled code for a method with more optimized code produced by a more expensive compiler. This problem is especially interesting when done while the original code is executing. In this case, it is sometimes called *on-stack replacement* [4], since the stack frame format corresponding to the original compilation must be converted to the format corresponding to the new compilation. We expect such problems to occupy us profitably in the coming year.

**Acknowledgments.** We would like to thank the compiler team on the Solaris Software Java Technology Group for making this research possible. We’d like to thank Dave Spott and Dave Cox especially for interesting discussions about the coordination of multiple-compiler systems. Finally, Cliff Click was extremely helpful in aiding us to make measurements of the Java HotSpot system.

## References

1. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño dynamic optimizing compiler for java. private communication, 1999.
2. L. Peter Deutsch and Allan Schiffman. Efficient implementation of a Smalltalk-80 system. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, pages 297–302, Salt Lake City, 1984. ACM SIGPLAN.
3. Urs Hölzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Ph.D. Thesis CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.
4. Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In Christopher W. Fraser, editor, *Proceedings of the ACM SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pages 32–43, San Francisco, CA, June 1992. ACM Press.