Mixed-mode Bytecode Execution

Ole Agesen and David Detlefs

Mixed-mode Bytecode Execution

Ole Agesen and David Detlefs

SMLI TR-2000-87 June 2000

Abstract:

Modern high-performance virtual machines use dynamic compilation. There is a tension between compilation speed and code quality. We argue that a highly-optimizing compiler is best deployed with both a fast, less-optimizing compiler and an interpreter. We present measurements showing that such a system can achieve the same peak performance as a system with just the optimizing compiler, and startup costs similar to a system with just the interpreter and fast compiler.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
agesen@vmware.com
david.detlefs@sun.com

© 2000 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Java, Solaris, and HotSpot are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, email Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.The entire technical report collection is available online at http://www.sun.com/research.

Mixed-mode Bytecode Execution

Ole Agesen[1*] and David Detlefs[+]

* VMware
3145 Porter Drive
Palo Alto, CA 94304
agesen@vmware.com

+ Sun Microsystems Laboratories
One Network Drive
Burlington, MA 01803-0903
david.detlefs@sun.com

1 Introduction

The most common approach to programming language implementation involves *static compilation* of source programs into machine code. This approach tolerates large compilation budgets, since compilation occurs only during program development. Thus, it has become the realm of optimizing compilation,and years of research and development have produced fantastically capable optimizers. Most implementations of C/C++ and Fortran follow this approach.

Another approach translates source programs into an architecture-neutral representation that can be executed by a *virtual machine*. Typical representations include abstract syntax trees and bytecode instruction sets. Most implementations of Smalltalk and the Java™ platform, as well as some Pascal systems, involve virtual machines [5, 11, 12]. Some virtual machines use an interpreter for execution, but to achieve high performance, some form of compilation into machine code seems necessary. This *dynamic compilation*, often called *just-in-time* (*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 high quality code. As a result, many optimization techniques developed for static compilers remain unaffordable in JIT compilers.

The tension between compilation speed and code quality became apparent in an early-generation Self system, one of the first to use an optimizing compiler dynamically [1, 9]. Hölzle shows how to reduce the tension by replacing certain costly static analyses with cheaper dynamic information-gathering techniques. For example, Hölzle's compiler uses a profile-based technique, type feedback [8], to gather type information that had been computed by an expensive analysis in an earlier Self compiler. Even so, many aspects of optimizing compilation remain costly and benefit little from dynamic information.

The Java HotSpot™ performance engine exemplifies a different approach to tolerating JIT compilation overhead [10]. As the name suggests, this system uses on-line profiling to identify and compile a performance-critical subset of the methods, while interpreting the rest. Two-mode execution offers distinct advantages over having only a single interpreted or compiled mode of execution, but, as we shall argue below, the gap between interpretation speed and optimized compiled speed is so large, and the profile of many programs so flat, that perfect performance becomes an unattainable balancing act: either the system under-compiles, causing slow interpretation to dominate performance, or it over-compiles, causing compilation costs to dominate performance.

The preceding observations bring us to the three main contributions of the present work:

- In Section 2, we demonstrate that contemporary Java virtual machines (JVMs) suffer from the compilation cost vs. code quality tension. This unfortunate state of affairs persists despite very high levels of investment into JVMs, calling for a different approach to designing execution engines for virtual machines.
- In Section 3, we propose a 3-mode (bytecode) execution engine, comprised of a simple interpreter, a fast non-optimizing compiler, and a slow optimizing compiler *used in the background*. This design appears attractive from an engineering point of view, in that it reduces demand for extreme performance from any one sub-system; in particular, an unsophisticated interpreter suffices and a slow optimizing compiler can be tolerated.

1. Work done while this author was a member of Sun Microsystems Laboratories.

- In Section 4, we present measurements to evaluate our 3-mode execution engine quantitatively. The system performs surprisingly well. In summary, when an extra processor is available to host the background compilation thread, the system delivers startup times as good as a 2-mode system with an interpreter and a fast compiler (i.e., it avoids excessive compilation overhead), and peak performance as good as the optimizing compiler permits (i.e., it avoids slow-down from interpretation or execution of unoptimized compiled code). Thus, at least for programs of which our suite of benchmarks are representative, fast startup and high peak performance can be achieved simultaneously, using 3-mode execution.

We conclude the paper with a look at related work (Section 5) and a summary of the main results (Section 6).

2 Existing configurations

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 VMs 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 a fixed code pattern for each virtual machine bytecode, doing almost no optimization but running extremely fast. More aggressive JIT compilers introduce some optimizations, but only those that preserve compilation speed. Often this means performing only optimizations that can be accomplished in a single linear code-generation pass. Our measurements will include 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 do not attempt to settle that issue, but rather demonstrate that appropriate dynamic compilation techniques can approach the performance of static compilers with minimal compilation overhead.

Each point in the scatter plot of Figure 1 represents a SPECjvm98 submission (as of the November, 1999). For each submission, the figure plots the worst (i.e., first-run) score against the best-run score. Obviously, all points are above the diagonal line that represents the family of systems with infinitely fast compilers (first = best). The extent to which a point is away from this line indicates the startup penalty imposed by compilation overhead.[2] Generally speaking, higher-performing systems tend to pay higher startup penalties, demonstrating the existence of the startup-speed/peak-performance tension. (This conclusion would be enhanced further if the SPEC ratios were normalized by the speed of the underlying hardware platform: the two points above 60, which challenge our conclusion by offering high performance with relatively low first-run penalties, were run on especially fast hardware.)

3 Description of implementation

In a JVM with three modes of bytecode execution, two transitions must be defined: from interpreter to JIT and from JIT to optimizing compiler. We present detailed measurements in Section 4, but it is useful to have a rough idea of the performance of our JVM's components in isolation to understand our choices when defining these boundaries. We use the SPECjvm98 suite of benchmarks [14] to characterize the interpreter and compilers. SPECjvm98, we have found, measures bytecode execution speed reasonably well. (It is, however, less suitable for measuring memory system or thread performance; the allocation rates and heap sizes of the SPECjvm98 programs are too small to stress a good memory system, and the only program to use more than one thread is mtrt, which uses just two.) Typically, one runs each SPECjvm98 program repeatedly on the same input in one JVM process, until the execution time stabilizes.

2. Note that several systems are quite close to the diagonal, implying that other startup costs, such as class loading, which are approximately equal in both interpreted and compiled systems, are not significant in these measurements.

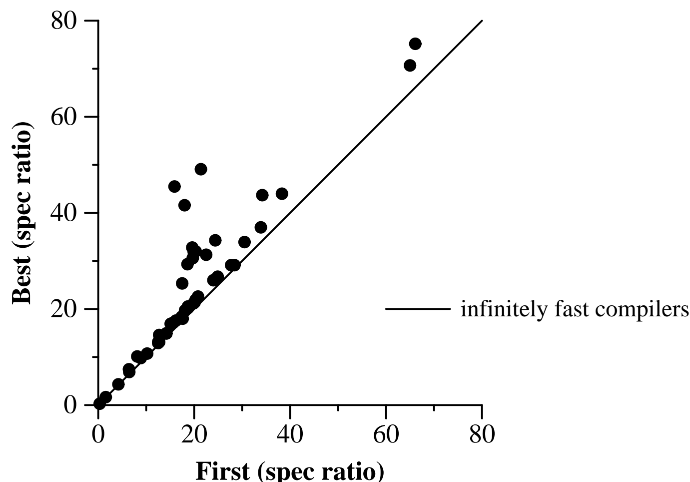

Figure 1. SPECjvm98 submissions: first vs. best scores

This gives two numbers for each program: “first” and “best” run time. Generally, the first run of each sequence includes compilation overhead time, while the subsequent ones do not. 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.

Interpreter. Our bytecode interpreter is implemented using straightforward C code. It fetches one bytecode instruction at a time, performs a `switch` on the opcode to branch to code implementing the operation, and then loops back to repeat the process. Previous versions of the interpreter loop optimized interpretation speed in various ways: one was hand-coded in assembler, another rewrote the bytecode stream into “quick” instructions as described in Chapter 9 of the first edition of the Java Virtual Machine Specification [11], and yet another one used gcc’s computed `gotos` to optimize the looping code. However, once the JIT and interpreter were fully integrated in the JVM, interpretation speed became unimportant for bottom-line performance, and, consequently, we backed out the optimizations to simplify the interpreter. On a system with dual 360 MHz UltraSPARC™ II processors, our interpreter achieves an overall SPECjvm98 ratio of 1.6 in both the first and best runs.

JIT compiler. Our JIT compiler began as a fast template-based compiler [2]. In its initial deployment as an add-on to the 1.0.2 JVM, it completely displaced the interpreter (except for class initializer methods), so a rather heavy compilation load was placed on the compiler. Thus, it was sensible to optimize for compilation speed rather than quality of generated code; averaged over several programs, compilation took 1400 cycles per bytecode instruction (700 cycles per byte of bytecode [2]). Today, the JIT compiler is fully integrated into the JVM, and it performs several optimizations, including: CSE within extended basic blocks, simple usage-count-based register allocation, elimination of array bounds-checking in inner loops, inlining of non-virtual and virtual methods [4], and delay-slot filling. Consequently, compilation speed slowed to 4300 cycles per bytecode instruction (which is still relatively fast). On the aforementioned UltraSPARC system, when compiling all methods, the JIT compiler achieves an overall SPECjvm98 ratio of 22.6 in the first and 23.6 in the best run.

Optimizing compiler. Our optimizing compiler traces its heritage back to a vectorizing and parallelizing compiler for Fortran and C developed at Supercomputer Systems Inc. (SSI) during the years 1987-93. Later, a Chaitin-Briggs-style global register allocator was added at Sun Microsystems. Later still, a front-end for Java class file bytecode (henceforth, Java bytecode) was developed and the compiler was integrated into our JVM. In its present form, the compiler employs two intermediate representations, HF, for high form, and LF, for low form. Optimizations done on HF include method inlining, local and global CSE, dead code elimination, and invariant code hoisting. LF optimizations include global register allocation, delay-slot filling, and branch optimization. While fairly mature for the other languages it supports, the incarnation of this compiler for the Java platform is somewhat immature. Further work on the compiler would probably produce significantly better code; this would increase the advantages we are claiming for 3-

mode execution. The multiple intermediate representations and the extensive optimizations make this compiler quite slow compared with the JIT, but probably not significantly slower than other similarly capable compilers: on average it burns 150,000 cycles per bytecode instruction compiled. Using the optimizing compiler on all methods yields a SPECjvm98 ratio of 18.8 in the first and 27.9 in the best run. The large difference reflects the cost of compilation.

3.1 Interpreter/JIT boundary

Since interpretation speed lags the speed of JIT-compiled code by 23.6 / 1.6 = 15x, the danger of interpreting too much is very real. Let $\alpha \in [0,1]$ be the dynamic fraction of bytecode instructions executed by the interpreter for some program. Then, crudely ignoring compilation time and other fixed costs, the run time of an interpreter/JIT mixed-mode system relative to a pure JIT system will be: $(1-\alpha) + 15\alpha = 1 + 14\alpha$. For example, interpreting just 5% of the dynamically executed bytecode instructions would result in a slow-down of 1.7x compared with executing all bytecode instructions in JIT-compiled form. Clearly, to keep slow interpretation out of the bottom line performance, we should interpret well under 1% of the bytecode instructions executed.

Our system uses the following heuristics. The first time a method is invoked, its bytecode instructions are scanned for back-branches. If a method contains a back-branch, then it may contain a loop; we will call such methods *loop methods*. Then:

- if the method is loop-free, we interpret it, keeping a count of how many times it has been invoked. When the count reaches a certain threshold, we compile the method;
- if the method is a loop method we compile it *immediately*, maintaining a barrier during compilation to prevent other threads from executing them in interpreted mode.

Once compiled, future invocations of a method use the compiled code. Our JVM allows interpreted and compiled invocations of a method to co-exist. While one thread compiles a loop-free method, other threads may continue to invoke it in interpreted form. For recursive methods, the compiling thread itself may have one or more interpreted activations of the method deeper in its stack. The default compilation threshold is 15 invocations. An alternative characterization of this compilation strategy is that loop-free methods have compilation threshold 15, loop methods have threshold one, and compilations at threshold one are special in that they maintain a barrier to prevent interpretation.

This mixed-mode strategy has some interesting properties. First, it is virtually impossible to burn too many cycles in the interpreter loop; only concurrent interpretation during JIT compilation can cause a bytecode instruction to be interpreted more than 15 times. Second, the strategy limits the number of calls from interpreted to compiled code and *vice versa*. This matters because interpreted and compiled calling conventions differ, e.g., the interpreter loop takes extra arguments such as the bytecode stream to interpret, requiring use of costly adapter routines between the two domains. Third, by deferring some compilation, more classes will be loaded and more constant pool entries resolved when compilation finally does occur, helping the JIT compiler produce better code.

In round numbers, one-third of all invoked methods get compiled because they may contain a loop, another third get compiled because they reach the compilation threshold, and the final third remain interpreted. This division shows remarkable stability; varying the compilation threshold in the range from 10 to 100 results in only modest changes in the total run time or total number of methods compiled.

If we increase the loop method compilation threshold to two, however, programs can get stuck in the slow interpreted mode. This problem is neither hypothetical nor insignificant. The first run of the compress benchmark in the SPECjvm98 suite slows from 33 to 78 seconds if loop methods are compiled at the second invocation instead of the first.

At this point, the reader might wonder: Is it at all worthwhile having an interpreter, given that it only reduces the compilation load by one third? In fact, the measurements we will present in Table 1 and Table 2 show little difference in elapsed time between the mode that uses the interpreter and JIT and the mode that runs the JIT compiler on every method before first execution. Still, we believe it is worthwhile to retain an interpreter. The interpreter yields an overall space savings, since compiled code is more voluminous than bytecode. For example, the JIT-all mode running the SPEC javac benchmark (via the provided applet interface, so that user-interface code is invoked) uses 2166 kilobytes of memory for compiled code, while the mode that interprets non-loop methods until their 15th invocation uses only 1555 kilobytes. Also, as mentioned above, interpretation tends to "pre-load" classes, helping the JIT compiler generate better code. This does not greatly effect our results, because our JVM uses a *dynamic patching* technique in which

code generated to do runtime symbol resolution is overwritten dynamically, to approximately the same code that the JIT compiler would have generated had the symbol been resolved at compile time. Other systems might show greater costs for symbols unresolved at compile-time. Finally, avoiding compilation of one-third of all methods allows a 50% increase in the compilation budget for the other methods, giving the JIT compiler time to perform additional optimizations. Effectively, interpretation of the performance-uncritical methods buys us time to optimize the critical methods. The extra compilation time that would result from having no interpreter is not significant in long-running programs (such as SPEC benchmarks), but can be quite significant in short-running programs. For example, "javac Hello.java" (i.e., invoking the source to bytecode compiler javac, which is itself written in the Java programming language, on a small file) runs 20% faster with the interpreter/JIT mixed-mode than with the JIT alone. Such savings, while individually small, can aggregate: consider a "traditional" makefile that invokes the javac compiler on many files individually to rebuild a system.

Another concern might be whether our interpreter is too slow. The Java HotSpot performance engine uses a sophisticated generated interpreter [6]. On the hardware described above, this interpreter delivers a SPECjvm98 score of 2.3, i.e., it is 2.3 / 1.6 = 1.4x faster than our interpreter. However, our system would only benefit marginally from this interpreter speedup: interpreted code would still be 23.6 / 2.3 = 10x slower than JIT-compiled code, so the bottom-line performance *must* be dominated by compiled code.

3.2 JIT/optimizing boundary

At the boundary between the JIT and optimizing compilers, the world looks very different than it does at the boundary between the interpreter and the JIT compiler. Recall that in switching from interpretation to JIT compilation, a compilation overhead of 4300 cycles per bytecode instruction results in an execution speedup of 15x. This speedup eliminates a large number of cycles that would otherwise have been burned in the interpreter loop, easily gaining back the costs of compilation. Progressing from JIT compilation to optimizing compilation incurs an additional 150,000 cycles of compilation overhead per bytecode instruction, and delivers an execution speedup of just 27.9 / 23.6 = 1.18x (again using our crude SPECjvm98 measure). This ratio might increase with more work on the optimizing compiler, but we feel it is unlikely to exceed 2x, and very unlikely to approach 15x. For very long-running programs tolerant of slow startup, and programs with extremely spiked profiles, the optimization investment is worthwhile. But for most programs, even ones that loop extensively like the SPECjvm98 suite, little stands to be gained and much to be lost from running the expensive optimizing compiler.

We tackle these challenges with a combination of two techniques: we use the optimizing compiler very selectively, and we run all optimizing compilations in a "background thread" so that the main computation can continue making progress in JIT-compiled code, while the optimizer slowly produces better code.

Selective optimization. We first observe that applying the optimizing compiler very sparingly is a sound strategy because our foundation from which we optimize is JIT-compiled code, which already runs fairly fast. Maintaining high selectivity on an interpreted foundation would have been much harder. Now consider the mechanism for finding the performance-critical methods. We considered two basic approaches: tick-based sampling, suggested by our colleague Robert Gottlieb, and direct instrumentation of the code produced by the JIT compiler. We chose to implement the latter scheme, although each approach has its merits. For example, ticks can be made to run at an arbitrarily low rate, allowing precise control of the measuring overhead. Instrumentation counters in the compiled code, on the other hand, promise greater determinism (although we still have a slightly nondeterministic effect in threaded programs, resulting from the use of unsynchronized counting code).

In more detail, each compiled method produced by the JIT compiler has an associated "activity counter." Initially, this counter is set to a relatively large positive number, the "3-mode threshold," whose default value is 10,000. For methods that contain no loop, the activity counter measures the number of invocations. These methods subtract one from their activity counter in the method prologue; see Figure 2 in which the activity-counting code is highlighted. Methods that contain loops use the activity counter to measure both the number of invocations and the number of loop iterations. In such methods, a register is reserved to hold the current invocation's activity. The register is set to one in the method prologue to account for method entry and incremented before each (taken or not taken) back-branch; effectively, we count loop iterations plus one for the untaken back-branch when the loop terminates. In the method epilogue, the activity accumulated in the register is subtracted from the activity counter in memory; see Figure 3 (in the code in the figure, the delay-slot filler has replicated the instruction that increments the loop activity register). In

either case, when the counter drops below zero, indicating that the JIT-compiled method has exceeded its allowed activity level, a call to an out-of-line routine adds the method to a recompilation queue.

Java source code:

```
static int factRec(int a) {
  if (a==0) return 1;
  return a*factRec(a-1);
}
```

Java bytecode:

```
  0 iload_0
  1 ifne 6
  4 iconst_1
  6 iload_0
  7 iload_0
  8 iconst_1
  9 isub
 10 invokestatic factRec
 13 imul
 14 ireturn
```

SPARC instructions:

```
fa03db50: save     %sp, -128, %sp          -- push stack frame
fa03db54: sethi    %hi(0xfa03d800), %g3    -- activity count addr.
fa03db58: lduw     [%g3 + 824], %g4        -- load count
fa03db5c: subcc    %g4, 1, %g4             -- dec. count
fa03db60: bg,a,pt  %icc,fa03db70           -- if >0, branch around call
fa03db64: stw      %g4, [%g3 + 824]        -- delay slot: store count
fa03db68: call     putOnCompilationQueue   -- add to compilation queue
fa03db6c: nop                              -- call delay slot
fa03db70: subcc    %i1, 0, %g0             -- test: a==0?
fa03db74: bne,a,pn%icc,fa03db8c            -- branch if no
fa03db78: sub      %i1, 1, %l0             -- in delay slot: %l0 = a-1
fa03db7c: mov      1, %i0                  -- result = 1
fa03db80: ret                              -- return and in delay slot...
fa03db84: restore                          -- pop stack frame
fa03db88: sub      %i1, 1, %l0             -- %l0 = a-1 (dead code)
fa03db8c: stw      %g0, [%sp - 4096]       -- stack overflow probe
fa03db90: lduw     [%fp + 64], %g1         -- load “current thread”
fa03db94: mov      %l0, %o1                -- %o1 = a-1
fa03db98: stw      %g1, [%sp + 64]         -- store “current thread”
fa03db9c: call     factRec                 -- recursive call
fa03dba0: mov      0, %o0                  -- silly calling convention: %o0 = 0
fa03dba4: mov      %o0, %l1                -- %l1 = factRec(a-1)
fa03dba8: smul     %i1, %l1, %i0           -- result = a*factRec(a-1)
fa03dbac: ret                              -- return and in delay slot...
fa03dbb0: restore                          -- pop stack frame
```

Figure 2. A loop-free method, its bytecode, and the SPARC code generated by the JIT compiler.

Java source code:

```
static int factLoop(int a) {
  int res = 1;
  for (int i=1; i<=a; i++)
    res = res*i;
  return res;
}
```

Java bytecode:

```
  0 iconst_1
  1 istore_1
  2 iconst_1
  3 istore_2
  4 goto 14
  7 iload_1
  8 iload_2
  9 imul
 10 istore_1
 11 iinc 2 1
 14 iload_2
 15 iload_0
 16 if_icmple 7
 19 iload_1
 20 ireturn
```

SPARC instructions:

```
fa03dc50: save     %sp, -128, %sp          -- push stack frame
fa03dc54: mov      1, %l0                  -- activity register = 1
fa03dc58: mov      1, %i3                  -- res = 1
fa03dc5c: mov      1, %i2                  -- i = 1
fa03dc60: ba,pt    %icc,fa03dc74           -- branch to bottom of loop
fa03dc64: add      %l0, 1, %l0             -- delay slot: inc. activity register
fa03dc68: smul     %i3, %i2, %i3           -- res = res*i (dead code)
fa03dc6c: add      %i2, 1, %i2             -- i++
fa03dc70: add      %l0, 1, %l0             -- increment activity register
fa03dc74: subcc    %i2, %i1, %g0           -- loop test: i<=a?
fa03dc78: ble,a,pt%icc,fa03dc6c            -- branch back if yes
fa03dc7c: smul     %i3, %i2, %i3           -- delay slot: res = res*i
fa03dc80: mov      %i3, %i0                -- result = res
fa03dc84: sethi    %hi(0xfa03dc00), %g3    -- activity count addr.
fa03dc88: lduw     [%g3 + 56], %g4         -- load count
fa03dc8c: subcc    %g4, %l0, %g4           -- subtract activity register
fa03dc90: bg,a,pt  %icc,fa03dca0           -- if >0, branch around call
fa03dc94: stw      %g4, [%g3 + 56]         -- delay slot: store count
fa03dc98: call     putOnCompilationQueue   -- add to compilation queue
fa03dc9c: nop                              -- call delay slot
fa03dca0: ret                              -- return and in delay slot...
fa03dca4: restore                          -- pop stack frame
```

Figure 3. Loop-containing method, its bytecode, and the SPARC code generated by the JIT compiler.

Our counting code is optimized for RISC processors. We try to place the counting code between instructions that would otherwise incur a load-use stall in the method prologue. We assume a large register file so that committing a register to holding the activity count in loop methods is not detrimental. Often, the simple register increment added before a loop back-branch will be "free" by virtue of executing in an otherwise-unused issue slot (of which there seem to be many in typical superscalar processors). In a pinch, we could reduce the instruction counts slightly compared with the instruction sequences shown in Figures 2 and 3: the subtract, compare, and branch instructions could be combined into a tagged subtraction operation that traps on underflow (`TSUBccTV`). Alternatively, the taken branch (in the common case) around the call, could be replaced with an untaken branch to an out-of-line call; we left out this optimization to avoid a delicate interaction with our garbage collector's stack walking code. By leaving the bulk of the counting code in a consecutive region, we reduce our ability to schedule the code with the surrounding instructions, but we gain the ability to revert it to a consecutive group of `NOP` instructions once the counter reaches zero. We have found that superscalar SPARC™ processors can execute groups of `NOP` instructions quite fast, so this reduces the residual costs in the interim phase after counters have grounded out but before the optimized code is available.

An optional refinement uses two separate counters: an entry counter and a loop iteration counter. When the entry counter hits zero, we schedule the *caller* for recompilation recording a hint to inline the callee. When the loop counter hits zero, we schedule the method itself for recompilation. This idea is inspired by Hölzle's work on adaptive optimization [8].

Background optimization. For a short-running program, waiting for even a single optimizing compilation to complete could result in a measurable slow-down. Ideally, we would compile only the methods for which future execution would recover the entire compilation costs, but this ideal is not attainable since we make optimization decisions online, i.e., without knowledge of the future. However, by moving optimizing compilation into a separate thread, we can control the amount of resources devoted to optimization with some accuracy. We run the optimization thread at low priority for the first 60 seconds to prevent it from draining cycles out of a short-running main computation. On multiprocessors, extra cycles are often available, resulting in background optimization activity even during the first 60 seconds. Equally welcome, on uniprocessors we observe optimization bursts during periods where the main computation blocks waiting for input or other asynchronous activity. On heavily loaded multiprocessors and uniprocessor systems running computationally intensive programs, this simple priority scheme seems to suffice to prevent slow-downs resulting from the optimization thread starving the main computation. Once the initial period is over, we raise the compilation thread's priority back up to normal in the expectation that the program will run for long enough to warrant a more aggressive optimization effort.

Doug Lea has proposed to us the use of a priority-weighted compilation order, which we may consider in the future. As of now, though, we have found that the simple FIFO compilation request queue provides a reasonable ordering. This is not really surprising: methods with high activity levels have their activity counters expire early, so they get on the compilation queue early, too. If the past is a good predictor of the future, these are the very methods that should be optimized soon, to make the most out of the optimized code.

We use a single background compilation thread, although our design allows any number of threads servicing the recompilation request queue. (The main reason for not using more than one thread is that the optimizing compiler itself runs under protection of a global lock. This is not a requirement imposed by the JVM—indeed, multiple JIT-compilations can run concurrently—but a vestige of the optimizing compiler's origins: most static compilers don't need to run concurrently with themselves.) The recompilation thread(s) wait on a condition variable for optimization requests (methods) to become available on a queue. The requests are removed and optimized one-by-one and the new code is installed in the code cache so that future invocations may enter the optimized code instead of the JIT-compiled code. Later, a code GC mechanism reclaims the JIT-compiled versions, as remaining activations of those versions terminate.

It may seem that 3-mode execution involves a substantial implementation effort, requiring a background compilation thread, the need to tolerate multiple compiled versions of each method, code garbage collection, and tracking of compiled code dependencies. In reality, however, most of these components already have to be present in a high-performance JVM. For example, optimistic inlining needs recompilation, a dependency mechanism, and code GC [4]. Thus, to a large degree, the 3-mode execution system was constructed out of existing components.

4 Measurements

Our measurements were done using a version of the Sun Java®2 SDK for the Solaris™ operating system, Production Release,[3] using a Sun Ultra™ 60 Creator3D desktop with two 360 MHz UltraSPARC II processors. We first present the key properties of each execution mode in isolation, using three executions of the SPECjvm98 suite: interpreting all methods, compiling all methods with the JIT, and compiling all methods with the optimizing compiler; see Table 1. All numbers in the table are SPECjvm98 ratios, i.e., speed-ups over the reference system's execution speed, so higher ratios are better. The benchmark execution harness runs each program on its input until the run-to-run difference is sufficiently small. Thus, the "best" run effectively factors out compilation costs, since no compilation occurs after the first iteration, allowing us to gauge the raw speed of each execution mode. As expected, interpretation achieves the same first and best score, the JIT compiler suffers a moderate loss of $(23.6 - 22.6)/23.6 = 4.2\%$ in the first run, and the optimizing compiler suffers a much greater loss of $(27.9 - 18.8)/27.9 = 32.6\%$ in the first run. Indeed, using the optimizing compiler produces a significantly worse first-run score than using the JIT. First-run scores are relevant in some real-world situations: consider again a makefile written in "standard form" that invokes the javac compiler separately on a large number of different files.

Table 1. SPECjvm98 ratios (higher is better) for the three execution modes in isolation.

	interpreter only		**JIT only**		**optimizing compiler only**	
Program	**first**	**best**	**first**	**best**	**first**	**best**
mtrt	2.1	2.1	46.0	48.7	29.5	56.6
jess	1.7	1.7	22.2	24.3	13.6	29.3
compress	1.1	1.1	36.3	36.7	45.4	47.1
db	1.2	1.2	9.1	9.5	9.3	9.6
mpegaudio	1.4	1.4	34.6	34.6	41.3	46.4
jack	2.3	2.3	20.2	20.9	14.5	22.4
javac	1.7	1.7	12.7	13.9	8.1	16.9
geometric mean	1.6	1.6	22.6	23.6	18.8	27.9

Next, we compare configurations of our JVM that use multiple modes of execution; see Table 2. We see several attractive properties in all these mixed-mode systems. Mixed-mode execution does not reduce the peak performance: best ratios equal the best ratios of the single-mode system with only the most optimized execution form available in the mixed-mode system. Generally, mixed-mode execution reduces the first-run penalty. However, the 2-mode interpreter/optimized mixed-mode system still leaves a significant first-run penalty: $(28.5 - 18.9)/28.9 = 33.7\%$. The 3-mode system surpasses or equals the other systems in all these measures. Its first-run score is the best of any system's first-run score; in fact, the 3-mode system's first run outperforms the interpreter/JIT system's best run. The first-run penalty is: $(28.1 - 25.3)/28.1 = 10.0\%$. The best run of the 3-mode system matches the best run of the system that compiles everything with the optimizing compiler.

Table 2. SPECjvm98 ratios (higher is better) for the mixed-mode systems.

	interpreter/JIT		**interpreter/optimizing**		**interpreter/JIT/optimizing**	
Program	**first**	**best**	**first**	**best**	**first**	**best**
mtrt	46.0	49.4	30.1	56.3	45.2	57.3
jess	22.4	24.3	13.7	31.5	25.7	28.5
compress	36.0	36.0	45.7	47.5	46.1	47.3
db	9.1	9.3	9.5	9.8	9.6	9.9
mpegaudio	34.1	34.1	41.5	48.6	46.7	49.0
jack	19.5	20.2	14.3	22.3	20.0	21.9
javac	12.7	13.7	8.2	17.2	14.0	16.7
geometric mean	22.4	23.4	18.9	28.5	25.3	28.1

Now, we take a closer look at two extreme benchmarks in the SPECjvm98 suite: compress with a spiked profile and javac with a flat profile. Previously, using manual off-line profiling, we have demonstrated that compiling as few as 20 methods in compress achieves nearly maximal performance [3]. Our 3-mode JVM handles compress to perfection, selecting just 10 methods for optimization, requiring a total of 0.55 seconds of CPU time in the background compilation thread. Of course, this is not difficult—even the interpreter/optimizing system can achieve nearly zero penalty

3. The non-experimental version of this system is available at http://www.sun.com/solaris/java. The JVM in this system was formerly known as ExactVM.

because this program runs very few methods. In contrast, javac executes many methods and has a flat profile, and therefore presents a tougher challenge. In 3-mode, 327 methods are compiled with the optimizing compiler, taking a total of 22.7 CPU seconds. Suppose the system on which we are running the benchmarks doesn't have any spare cycles and we do nothing to throttle the background compilation thread. Then, compress will still run perfectly well, but the first run of javac would be slowed down by 22.7 seconds of compilation work, resulting in a much *worse* first run than the interpreter/JIT system (the interpreter/JIT system's first-run *time* for javac is 33.4 seconds).

Since we perform these measurements on a machine with an otherwise-unused processor available for background compilation, the system is easily able to process background compilation requests. In a set of 3-mode measurements similar to those summarized in Table 2, all benchmarks performed the majority of optimizing compilations during the first run (of at least 3); the (geometric) average was 69%. The 3-mode strategy can also be used on a uniprocessor, but invocation thresholds must be increased to allocate the processor appropriately between compilation and execution. With settings that keep first-run times competitive, several runs are necessary to complete enough compilation to approach best-run performance. But the essential property still holds: short-lived programs run well, while long-lived programs eventually achieve peak performance.

Our final set of measurements illuminates the cost of the activity counters that select methods for recompilation. For this experiment, we modified the 3-mode system to skip the recompilation step, but leave in the activity counters. Comparing this system with the interpreter/JIT system reveals the costs of the activity counters. The SPECjvm98 geometric means for the counting system are 21.4 for the first run and 22.6 for the best run, giving a counting overhead of approximately 3.5%. Clearly, our 3-mode system more than overcomes this instrumentation overhead, to achieve a net performance advantage.

5 Related work

The introduction discussed dynamic compilation systems for Smalltalk, Self, and Java virtual machines; there are many other such systems with dynamic compilation. Some incarnations of the Self system contained several compilers with different speed/code-quality characteristics. However, Self included neither an interpreter nor background optimizing compilation. Plezbert and Cytron discuss a 2-mode interpreter/JIT system with background compilation [13].

The Self system introduced *on-stack replacement (OSR),* in which an actively executing method (one "on the stack") can to be upgraded from interpreted to compiled mode [7, 8]. OSR aids mixed-mode systems by limiting performance losses resulting from interpreting methods containing long-running loops. Without OSR, such methods must run to completion. Additionally, using OSR in the inverse direction, from compiled execution to interpretation (also called *deoptimization*), allows an optimizer to make optimistic assumptions; should these assumptions become invalid during execution, deoptimization back to interpreted mode provides a bail-out path. Clearly, OSR is an ingenious and powerful mechanism, but it also has potential disadvantages. Its application for deoptimization requires the compilation process to preserve enough information to recover the full source state; this information may consume significant space, and this requirement restricts certain optimizations, such as code motion. In either direction, OSR can be difficult to implement correctly.

In the context of the present work, the upgrade direction of OSR is the most interesting, since it allows increased selectivity in what to optimize. However, we don't believe it relieves the basic tension between startup cost and peak performance enough to overcome the shortcomings of interpreter/optimized 2-mode systems. OSR aids such systems with programs similar to compress, which execute many loop iterations in individual invocations of some methods, so that interpreting even a single invocation to completion dramatically decreases performance. It does not help, however, with programs similar to javac, which spread execution fairly evenly over a large set of methods; here the problem is not when each method is compiled, but rather how much needs to be compiled to avoid a large amount of interpretation.

6 Conclusions

The argument made in this paper is based on two points. First, if high performance is an important goal in a virtual machine design, very little code, dynamically speaking, can be interpreted. The gap between interpretation speed and the speed of code produced by JIT compilers (measured at 15x in our system) is just too large. Further, interpretation

doesn't even contribute very much to decreasing startup time, since the JIT compilers that produce these speedups can be quite fast, so their overhead is quickly amortized — program invocations of extremely short duration excepted, of course. This does not mean that virtual machines should not have interpreters: we have noted several good reasons for doing some interpretation, such as minimizing space costs for compiled code and ensuring that more dynamic information is available when a method is compiled. However, interpretation cannot be a significant execution mode in a high-performance system.

The second point is an intuition: that there are two kinds of compilers, JIT compilers designed to do as much optimization as is compatible with being very fast, and optimizing compilers designed to produce the best code possible, with compilation speed being a secondary concern at best. It could be argued that the relatively small 18% peak-performance difference between the JIT and optimizing compilers in the system we have measured contradicts this intuition. However, we should reiterate that the optimizing compiler, in its application to the Java platform, is immature, and could be improved substantially. The authors have direct experience with the JIT compiler, however, and feel that it is quite mature. Further improvements to the JIT would likely involve introduction of more complicated intermediate forms supporting more expensive analyses; in short, to become more like an optimizing compiler in performance it would have to become more like one in cost. Certainly there will be efforts to increase the quality of JIT compilers while keeping them fast, and to speed up optimizing compilers while keeping them optimizing, and such efforts will have some degree of success. However, we believe the basic distinction will persist.

And perhaps it should. We have shown that 3-mode execution can maintain good first-run speed while allowing judicious application of an expensive compiler to quickly approach peak performance. This should be liberating to designers of dynamic optimizing compilers: optimizations that might have been rejected because of expense can now be considered. Users can "have it all": no need to choose between systems offering fast startup or peak performance, and perhaps better peak performance as well.

Acknowledgments. Dave Cox and Dave Seberger helped us with information about the optimizing compiler.

References

1. Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proc. OOPSLA'91,* pp. 1-15, Phoenix, AZ, October 1991.
2. Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time. *IEEE Micro*, May/June 1997.
3. David Detlefs and Ole Agesen. The Case for Multiple Compilers. In *OOPSLA'99 Workshop on Performance, Portability, and Simplicity in Virtual Machine Design*, http://www.squeak.org/oopsla99_vmworkshop, Denver, CO, November 1999.
4. David Detlefs and Ole Agesen. Inlining of Virtual Methods. In *Proc. ECOOP'99*, pp. 258-278, Lisbon, Portugal, June 1999.
5. Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.
6. Robert Griesemer. Generation of Virtual Machine Code at Startup. *OOPSLA'99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design,* http://www.squeak.org/oopsla99_vmworkshop. Denver, CO, November 1999.
7. Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California, August 1994. Also published as Sun Microsystems Laboratories Technical Report, SMLI TR-95-35, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, March 1995.
8. Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proc. PLDI'94,* pp. 326-336. Published as *SIGPLAN Notices,* 29(6). Orlando, FL, June 1994.
9. Urs Hölzle and David Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proc. OOPSLA'94,* p. 229-243, Portland, OR, October 1994.
10. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html. April 1999.
11. Timothy Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. The Java Series, Addison-Wesley, 1999.
12. Daniel R. Perkins and Dennis Volper. UCSD Pascal on the VAX, Portability and Performance. *Software—Practice and Experience*. 14(5), p. 473-482, 1984.
13. Michael P. Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late than Never"? In *POPL'97*, p. 120-131. Paris, France, January 1997.
14. SPEC jvm98 Benchmarks. http://www.spec.org/osg/jvm98. August 19, 1998 release.

Appendix: Raw Data

This appendix presents the raw data from which the SPEC ratios of Table 1 and Table 2 were calculated, for archival purposes. Each line shows a sequence of times for consecutive runs in the same virtual machine, in seconds. The "first" scores reported in the main text use the best of the first times; the "best" scores were calculated from the best time overall for a configuration.

Pure interpreter:

Benchmark				
mtrt -	220.195,	220.277		
jess -	228.358,	227.347		
compress -	1055.604,	1058.758		
db -	415.15,	411.4		
mpegaudio -	791.677,	786.217		
jack -	195.254,	193.85		
javac -	249.729,	244.556		

Pure JIT:

Benchmark				
mtrt -	10.165,	9.914		
	10.006,	9.454,	9.578	
	9.999,	9.997		
jess -	17.793,	16.233,	16.196	
	17.082,	15.659,	15.93	
	17.098,	15.938,	16.024	
compress -	34.187,	33.62		
	32.648,	32.827		
	32.384,	32.004		
db -	57.323,	55.767		
	55.827,	53.241,	56.559	
	55.752,	54.977		
mpegaudio -	31.811,	33.218		
	33.199,	33.924		
	33.532,	33.554		
jack -	22.521,	21.847,	21.77	
	23.249,	22.532,	22.521	
	23.161,	22.477,	22.408	
javac -	34.482,	30.823,	31.622	
	33.557,	31.509,	31.406	
	33.619,	30.671,	30.927	

Pure JBE:

Benchmark				
mtrt -	15.597,	8.58,	8.541	
	15.752,	8.628,	8.124,	8.27
	15.651,	8.91,	8.548,	8.248
jess -	28.277,	13.13,	13.323	
	27.98,	12.958,	13.165	
	28.469,	13.401,	13.553	
compress -	26.348,	25.434,	26.717	
	25.906,	24.943,	25.348	
	26.119,	25.077,	25.068	
db -	55.965,	54.151,	54.121	
	54.063,	52.671		
	54.186,	53.248		
mpegaudio -	31.114,	27.542,	27.786	
	26.851,	23.703,	25.522	
	26.61,	24.013,	23.449	
jack -	31.665,	20.672,	20.657	
	31.726,	20.687,	20.669	
	31.376,	20.327,	20.299	
javac -	53.799,	26.378,	26.015	
	52.283,	25.779,	25.152	
	53.021,	26.792,	26.455	

Interpreter/JIT:

Benchmark				
mtrt -	9.999,	9.318,	9.629	
	10.091,	9.709,	9.921	
	10.0,	9.349,	9.58	
jess -	17.082,	15.845,	16.074	
	17.206,	15.667,	16.145	
	16.987,	15.945,	16.128	
compress -	33.375,	32.654		
	33.029,	32.906		
	32.675,	33.415		
db -	55.688,	54.328		
	56.683,	55.932		
	59.198,	56.117,	56.618	
mpegaudio -	32.771,	33.467		
	32.65,	32.688		
	32.298,	33.028		
jack -	23.407,	22.625,	22.511	
	23.847,	23.394		
	23.279,	22.6,	22.546	
javac -	33.383,	30.939,	31.006	
	34.245,	32.081,	31.839	
	33.699,	31.174,	31.306	

Interpreter/JBE

Benchmark				
mtrt -	15.291,	8.648,	8.985	
	15.949,	8.285,	8.174	
	16.017,	8.698,	8.18,	8.238
jess -	27.755,	12.047,	13.134	
	28.165,	13.129,	13.275	
	28.858,	13.832,	14.002	
compress -	25.726,	24.93,	25.128	
	25.721,	25.268		
	25.927,	24.719,	24.853	
db -	54.713,	51.34,	51.284	
	55.303,	51.799,	52.463	
	53.185,	52.742		
mpegaudio -	26.487,	24.248,	23.847	
	26.604,	23.696,	22.622,	22.673
	26.603,	24.416,	22.948,	22.638
jack -	31.915,	20.652,	20.448	
	32.475,	20.704,	20.498	
	32.504,	21.258,	21.452	
javac -	51.519,	25.562,	24.73,	25.027
	52.343,	25.468,	25.399	
	51.996,	25.668,	24.89,	25.004

3-mode:

Benchmark				
mtrt -	11.203,	8.073,	8.089	
	11.114,	8.875,	8.24,	8.554
	10.175,	8.602,	8.024,	8.539
jess -	17.258,	13.393,	13.418	
	14.914,	13.445,	13.507	
	14.772,	13.588,	13.33	
compress -	32.927,	25.582,	25.498	
	26.07,	25.458		
	25.499,	24.853		
db -	53.214,	51.427,	51.141	
	53.803,	51.495,	53.231	
	52.702,	50.794,	51.728	
mpegaudio -	24.146,	22.752,	22.456	
	23.748,	23.059		
	23.544,	22.597,	22.648	
jack -	23.338,	20.876,	20.799	
	22.797,	21.692,	21.202	
	23.109,	21.499,	21.116	
javac -	35.894,	26.356,	26.263	
	30.907,	25.488,	25.874	
	30.382,	25.457,	25.478	

About the Authors

Ole Agesen used to work on virtual machines at Sun where he was a member of the Self, Kanban, and Java Technology Research groups. Now he works on a different kind of virtual machine at VMware.

David Detlefs is a Staff Engineer in the Java Technology Research Group in Sun Microsystems Laboratories. He worked previously at the Digital Equipment Corporation's Systems Research Center. He has an S.B. degree from the Massachusetts Institute of Technology and a Ph.D. from Carnegie Mellon University, both in Computer Science.