Superinstructions and Replication in the Cacao JVM interpreter

M. Anton Ertl [*] Christian Thalinger Andreas Krall

TU Wien

Abstract

Dynamic superinstructions and replication can provide large speedups over plain interpretation. In a JVM implementation we have to overcome two problems to realize the full potential of these optimizations: the conflict between superinstructions and the quickening optimization; and the non-relocatability of JVM instructions that can throw exceptions. In this paper, we present solutions for these problems. We also present empirical results: We see speedups of up to a factor of 4 on SpecJVM98 benchmarks from superinstructions with all these problems solved. The contribution of making potentially throwing JVM instructions relocatable is up to a factor of 2. A simple way of dealing with quickening instructions is good enough, if superinstructions are generated in JIT style. Replication has little effect on performance.

1. Introduction

Virtual machine interpreters are a popular programming language implementation technique, because they combine portability, ease of implementation, and fast compilation. E.g., while the Mono implementation of .NET has JIT compilers for seven architectures, it also has an interpreter in order to support other architectures (e.g., HP-PA and Alpha). Mixed-mode systems (such as Sun's HotSpot JVM) employ an interpreter at the start to avoid the overhead of compilation, and use the JIT only on frequently-executed code.

The main disadvantage of interpreters is their run-time speed. There are a number of optimizations that reduce this disadvantage. In this paper we look at dynamic superinstructions (see Section 2.1) and replication (see Section 2.2), in the context of the Cacao JVM interpreter.

While these optimizations are not new, they pose some interesting implementation problems in the context of a JVM implementation, and their effectiveness might differ from that measured in other contexts. The main contributions of this paper are:

- We present a new way of combining dynamic superinstructions with the quickening optimization (Section 3).
- We show how to avoid non-relocatability for VM instruction implementations that may throw exceptions (Section 4).
- We present empirical results for various variants of dynamic superinstructions and replication combined with different approaches to quickening and to throwing JVM instructions (Section 5). This shows which of these issues are important and which ones are not.

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

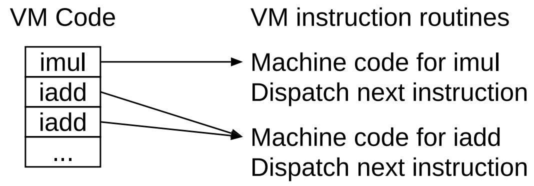

Figure 1. Threaded-code representation of VM code

2. Background

This section explains some of the previous work on which the work in this paper is built.

2.1 Dynamic Superinstructions

This section gives a simplified overview of dynamic superinstructions [RS96, PR98, EG03].

Normally, the implementation of a virtual machine (VM) instruction ends with the dispatch code that executes the next instruction. A particularly efficient representation of VM code is threaded code [Bel73], where each VM instruction is represented by the address of the real-machine code for executing the instruction (Fig. 1); the dispatch code then consists just of fetching this address and jumping there.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: .NET Technologies 2006, Plzen, Czech Republic

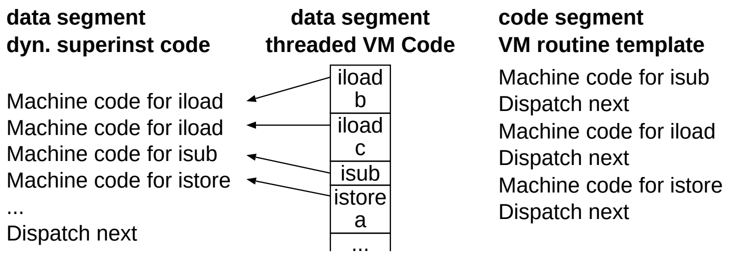

Figure 2. A dynamic superinstruction for a simple JVM sequence

A VM superinstruction is a VM instruction that performs the work of a sequence of simple VM instructions. By replacing the simple VM instructions with the superinstruction, the number of dispatches can be reduced and the branch prediction accuracy of the remaining dispatches can be improved [EG03].

One approach for implementing superinstructions is dynamic superinstructions: Whenever the front end[1] of the interpretive system compiles a VM instruction, it copies the real-machine code for the instruction from a template to the end of the current dynamic superinstruction; if the VM instruction is a VM branch, it also copies the dispatch code, ending the superinstruction; the VM branch has to perform a dispatch in order to perform its control flow, otherwise it would just fall through to the code for the next VM instruction. As a result, the real-machine code for the dynamic superinstruction is the concatenation of the real-machine code of the component VM instructions (see Fig. 2).

In addition to the machine code, the front end also produces threaded code; the VM instructions are represented by pointers into the machine code of the superinstruction.

During execution of the code in Fig. 2, a branch to the `iload b` performs a dispatch through the first VM instruction slot, resulting in the execution of the dynamic superinstruction code starting at the first instance of the machine code for `iload`, and continues the execution of the dynamic superinstruction until it finally executes another dispatch through another VM instruction slot to another dynamic superinstruction.

As a result, most of the dispatches are eliminated, and the rest have a much better prediction accuracy on CPUs with branch target buffers, thus eliminating most of the dispatch overhead. In particular, there is no dispatch overhead in straight-line code.[2]

[1] More generally, the subsystem that generates threaded code, e.g., the loader or, in the case of the Cacao interpreter, the JIT compiler.

[2] Some people already consider this to be a simple form of JIT compilation. In this paper we refer to it as an interpreter technique, for the following reasons: 1) It can be added with relatively little effort and very portably (with fall-back to plain threaded code if necessary) to an existing threaded-code interpreter; 2) The executed machine code still accesses the VM code for immediate arguments and for control flow.

There is one catch, however: Not all VM instruction implementations work correctly when executed in another place. E.g., if a piece of code contains a relative address for something outside the piece of code (e.g., a function call on the IA32 architecture), that relative address would refer to the wrong address after copying; therefore this piece of code is not relocatable for our purposes.[3] The way to deal with this problem is to end the current dynamic superinstruction before the non-relocatable VM instruction, let the VM instruction slot for the non-relocatable VM instruction point to its original template code (which works correctly in this place), and start the next superinstruction only afterwards.

Dynamic superinstructions can provide a large speedup at a relatively modest implementation cost (a few days even with the additional issues discussed in this paper). It does require a bit of platform-specific code for flushing the instruction cache (usually one line of code per platform), but if this code is not available for a platform, one can fall back to the plain threaded-code interpreter on that platform.

2.2 Replication

As described above, two equal sequences of VM instructions result in two copies of the real-machine code for the superinstruction (replication [EG03]).

An alternative is to check, after generating a superinstruction, whether its real-machine code is the same as that for a superinstruction that was created earlier[4]; if so, the threaded-code pointers can be directed to the first instance of the real-machine code, and the new instance could be freed (non-replication).

Replication is good for indirect branch prediction accuracy on CPUs with branch target buffers (BTBs) and is easier to implement, whereas non-replication reduces the real-machine code size significantly and can reduce the I-cache misses.

[3] Why do we not support a more sophisticated way of relocating code that does not have this problem? Because that relocation method would be architecture-specific, and thus we would lose the portability advantage of interpreters; it would also make the implementation significantly more complex, reducing the simplicity advantage.

[4] Of course, instead of checking the real-machine code afterwards, one could also check the virtual-machine code beforehand, but that is an implementation detail.

ACONST	INVOKESPECIAL
ARRAYCHECKCAST	INVOKESTATIC
CHECKCAST	INVOKEVIRTUAL
GETFIELD_CELL	MULTIANEWARRAY
GETFIELD_INT	NATIVECALL
GETFIELD_LONG	PUTFIELD_CELL
GETSTATIC_CELL	PUTFIELD_INT
GETSTATIC_INT	PUTFIELD_LONG
GETSTATIC_LONG	PUTSTATIC_CELL
INSTANCEOF	PUTSTATIC_INT
INVOKEINTERFACE	PUTSTATIC_LONG

Figure 3. Instructions in the (JVM-derived) Cacao interpreter VM that reference the constant pool

2.3 Cacao interpreter

For the present work, we revitalized the Cacao interpreter [GEK01, EGKP02] and adapted it to the changes in the Cacao system since the original implementation (in particular, quickening, and OS-supported threads).

The most unusal thing about the Cacao interpreter is that it does not interpret the JVM byte code directly; instead, a kind of JIT compiler (actually a stripped-down variant of the normal Cacao JIT compiler) translates the byte code into threaded code for a very JVM-like VM, which is then interpreted. The advantage of this approach is that the interpreter can use the fast threaded-code dispatch, and the immediate arguments of the VM instructions can be accessed much faster, because they are properly aligned and byte-ordered for the architecture at hand. Moreover, this makes it easier to implement dynamic superinstructions and enables some minor optimizations.

The Cacao interpreter is implemented using Vmgen [EGKP02], which supports a number of optimizations (e.g., keeping the top-of-stack in a register), making our baseline interpreter already quite fast.

3. Quickening

This section discusses one of the problems of the JVM and .NET when implementing dynamic superinstructions.

3.1 The problem

A number of JVM instructions (see Fig. 3) refer to the constant pool, and through it to components of (possibly) other classes. A class should be loaded and must be initialized exactly when the first instruction refering to it is executed.

Performing all the overhead of checking whether the class is already loaded and initialized, and resolving the class and component information into the actual information needed (an offset in the case of `getfield`) on every execution is very expensive: in experiments with an old version of Kaffe[5] we found that optimizing this overhead away gives a speedup on the SpecJVM98 benchmarks by about a factor of 3.

The original Java interpreter optimizes such *slow instructions* by rewriting them and their immediate operand(s) in the VM code into *quick instructions* when they are executed the first time [LY97, Chapter 9]. This optimization is called *quickening*.

The immediate operand of the quick instruction is the result of resolving the operand of the slow instruction. E.g., for the slow `getfield` instruction the immediate operand is a constant-pool reference to the field of a class, whereas the immediate operand of `getfield_quick` is the offset of the field. In our examples (e.g., Fig. 4) as well as in our implementation, we have separate slots for these two operands; in the examples, this makes it easier to show what is happening; in the implementation, this reduces the need for locking [GH03].

This approach does not work with dynamic superinstructions: In general, rewriting the VM code is not enough; we would also have to rewrite or patch the real-machine code generated for the superinstruction; and the difficulties there are that the real-machine code of the slow and the quick instruction usually have a different length; moreover, the slow instruction and its quick equivalent might not be both relocatable (usually, the slow instruction is not relocatable, and the quick instruction is).

3.2 A simple solution

A simple solution is to exclude slow instructions from being integrated into dynamic superinstructions (just as it is done for non-relocatable instructions). A preceding dynamic superinstruction would end right before the slow instruction and dispatch to the slow instruction as usual in threaded code. The slow instruction could then rewrite itself into the quick instruction, as in a plain threaded-code interpreter (see Fig. 4).

The disadvantages of this solution are:

- Usually two additional VM instruction dispatches are performed per execution of the quick instruction that would not be performed if it was integrated in the dynamic superinstruction: One for ending the preceding superinstruction, and one by the quick instruction itself. This hurts mainly CPUs without BTBs (branch target buffers) or similar indirect-branch predictors.
- The quick instruction is not replicated, leading to a low prediction accuracy for the dispatch by the quick instruction on CPUs with BTBs. This disadvantage could be eliminated in, e.g., the following way: When the slow instruction rewrites itself into the quick instruction, it replicates the quick instruc-

[5] http://www.complang.tuwien.ac.at/java/kaffe-threaded/

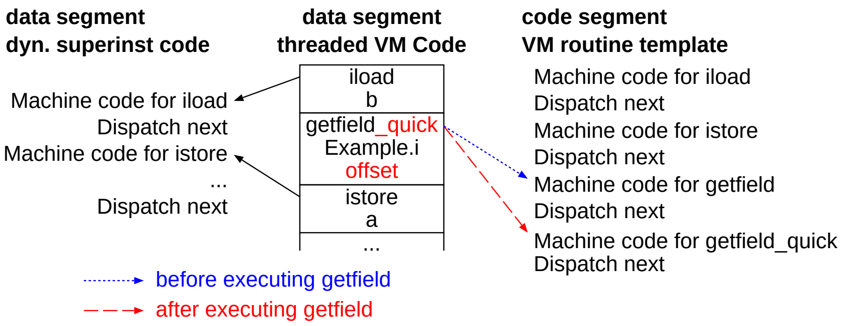

Figure 4. Simple solution: Exclude slow and quick instructions from dynamic superinstructions.

tion (including its dispatch) and lets the instruction slot point to the new replica. However, this approach will lead to less spatial locality and thus more I-cache misses than the normal arrangement of dynamic superinstructions with replication.

- When applying additional optimizations, such as static superinstructions [EGKP02] or static stack caching [EG04a], the natural approach to take would be to also exclude the to-be-quickened instructions from these optimizations. Everything else would require additional implementation costs similar to more sophisticated approaches for this problem.

These disadvantages lead to significant slowdowns (compared to more sophisticated approaches) when all slow instructions are treated in this way [GH03].

However, the Cacao interpreter translates the JVM bytecode into threaded code using a JIT compiler with method granularity. If the JIT compiler encounters a slow JVM instruction that references a class that has already been loaded and initialized, it translates it into a quick instruction without the intermediate state of a slow threaded-code instruction. These quick instructions can be integrated into dynamic superinstructions without a problem.

So, in the Cacao interpreter, only a subset of the slow instructions from the original JVM code have the problems mentioned above even with this simple solution. If the parts of the code containing this subset are only executed rarely, the performance disadvantage of the simple solution is negligible. Our results (see Section 5) indicate that this is indeed the case.

However, we did not know this from the start, so we also looked into more sophisticated approaches. Moreover, more sophisticated approaches do have their merits in settings (like SableVM) where no JIT translation into threaded code with superinstructions is used: At least our sophisticated approach is simpler to implement than a JIT translator.

3.3 Previous sophisticated solutions

Like the Cacao interpreter, SableVM translates the JVM bytecode into threaded code with dynamic superinstructions. One difference is that SableVM keeps only the instruction slot for the first VM instruction in a superinstruction, whereas the Cacao interpreter keeps all the VM instruction slots around (even though only the first one is used when the superinstruction is executed); to avoid confusion, we show the same approach in all the examples: all VM instruction slots are kept.

SableVM deals with quickening by creating an out-of-line preparation sequence in VM code (see Fig. 5), as well as the superinstruction (which incorporates the quick versions of any instructions to be quickened instruction). On first execution the VM code jumps to the preparation sequence, which performs the first execution (including a variant of the slow instruction that patches the operand elsewhere), and rewrites the `goto` to the preparation sequence into an invocation of the superinstruction; finally, the preparation sequence jumps to the first (super)instruction behind the VM code covered by the superinstruction. On the next execution, the VM code just executes the superinstruction.

Casey et al. [CEG05, Section 5.4] have also implemented dynamic superinstructions in a JVM interpreter. They treat the slow instruction as non-relocatable, as in the simple solution, but leave space in the real-machine-code area for the (real-machine code of the) corresponding quick instruction; on quickening, they copy the real-machine code for the quick instruction into that space, resulting in a dynamic superinstruction that includes the quick instruction. This solution requires that all VM instruction slots are kept around.

3.4 Our sophisticated solution

Figure 6 shows our approach: When we generate the threaded code for a block, we also generate the real-machine code for the superinstruction; however, if we encounter a slow instruction, we generate the real-machine code for the appropriate quick instruction.

However, if there is a slow instruction in the block, we do not let the threaded code point to the dynamic superinstruction right away. Instead, we first generate conventional threaded code, which does not reference the dynamic superinstruction in any way, and that code

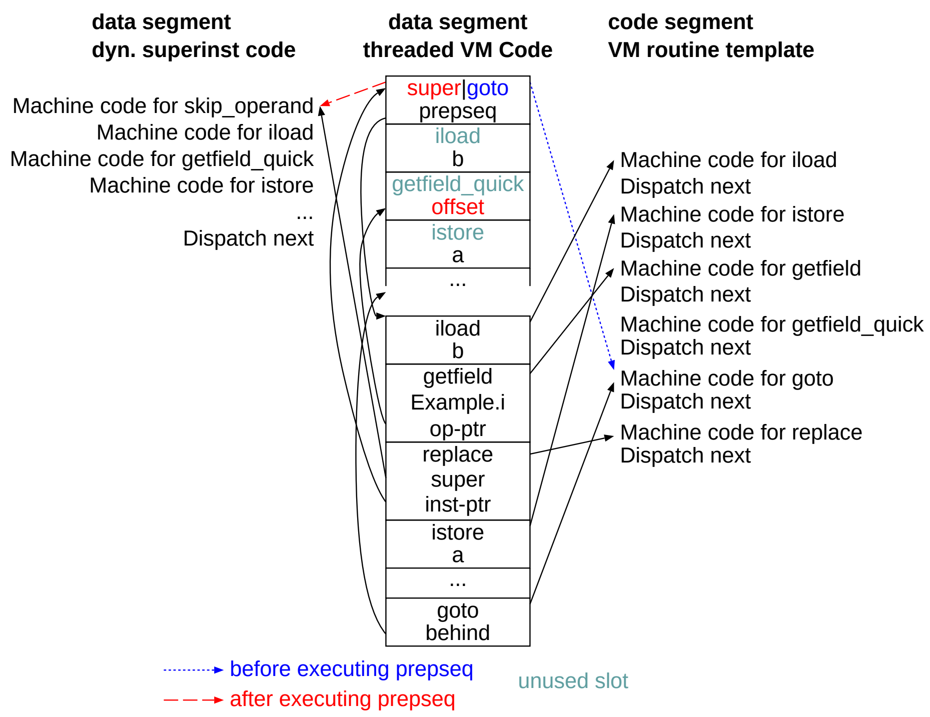

Figure 5. SableVM's preparation sequence for the first execution and the dynamic superinstruction including a quick instruction for subsequent executions

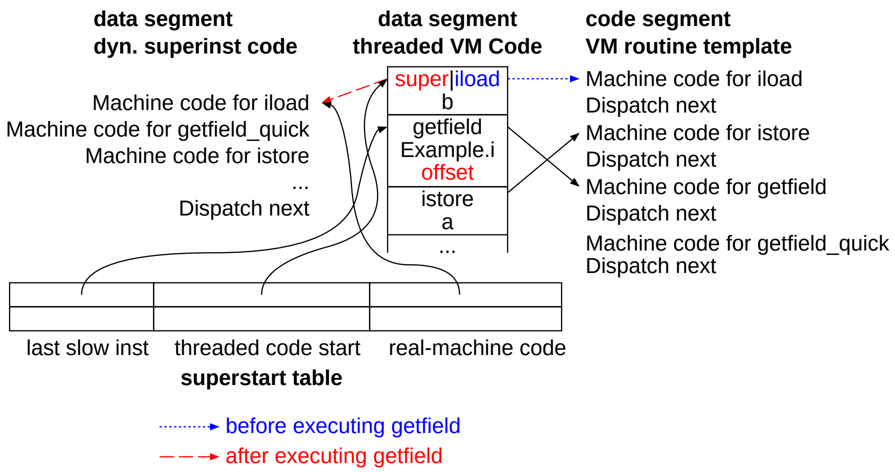

Figure 6. Cacao's sophisticated solution: first execute threaded code; the last slow instruction rewrites the first instruction in the sequence into the superinstruction.

contains the slow instructions. We also record what the last slow instruction in the block is, and use this in a table called superstart: the last slow instruction is the lookup key, and it also contains a pointer to the superinstruction real-machine code for the block, and the first threaded-code word in the block.

When a slow instruction is executed, it first performs all the necessary loading and initialization work. Then it looks itself up in the superstart table, and patches the threaded-code word at the block start to point to the real-machine code for the superinstruction.[6] The next time the basic block is executed, it will use the dynamic superinstruction.

We did not define above what we mean by *block*: It is the VM code covered by a dynamic superinstruction. It is essentially the same as a basic block, with one additional boundary condition: If there is a VM instruction

[6] We need to patch only the first threaded-code word, because, once we are executing the dynamic superinstruction, the other threaded-code words are not used.

with non-relocatable real-machine code, that also terminates the superinstruction (and thus the block); the next superinstruction starts after the non-relocatable VM instruction.

In earlier work [EG03] we let superinstructions continue straight-line across control-flow joins. We cannot do this here; consider the case of a superinstruction consisting of two basic blocks, with each basic block containing one slow VM instruction:

- When the first slow instruction is reached, this is not the last slow instruction in the superinstruction, so we cannot do the patching; if we did, we would get a race condition: another thread could execute the quick instruction implementation in the superinstruction before this thread has performed the necessary class loading and initializations.
- When the second slow instruction is reached, it does not know if it can patch the start of the first basic block, because it does not know if that basic block and its slow instruction has been executed, and the appropriate initializations done.

As a result, the part of the superinstruction for the first basic block would never be used.

In earlier work we also let superinstructions continue across fall-through edges of conditional branches. We also do not do this here: If there is a slow instruction in the fall-through path, but the branch is always taken, the superinstruction might never be activated.

One could work around these issues, but that would require significant complexity.

Note that the simple solution (Section 3.2) does not have these restrictions and thus can be better than our sophisticated solution (depending on the dynamic frequencies of originally-slow instructions vs. basic block ends and not-taken conditional branches).

Another thing worth noting is that our solution requires that the superinstruction keeps all the VM instruction slots, because the first time the code is executed as plain threaded code. In terms of the SableVM solution, we use the original sequence combined with the entry in the superstart table as preparation sequence. So keeping all the slots leads to a significant simplification here, as well as in other contexts, such as superinstructions across basic block boundaries [EG03] and optimal selection of static superinstructions.

Finally, one of the advantages of our sophisticated approach over the simple solution and over the solution of Casey et al. [CEG05] is that our solution is easier to adapt to situations where dynamic superinstructions are combined with static stack caching and/or static superinstructions: While generating the dynamic superinstruction, we use static stack caching or static superinstructions without having to consider complications from quickening, and the threaded code for the first execution need not use these optimizations.

IALOAD	GETFIELD_CELL
LALOAD	GETFIELD_INT
AALOAD	GETFIELD_LONG
BALOAD	PUTFIELD_CELL
CALOAD	PUTFIELD_INT
SALOAD	PUTFIELD_LONG
IASTORE	INVOKEVIRTUAL
LASTORE	INVOKESPECIAL
BASTORE	INVOKEINTERFACE
CASTORE	ARRAYLENGTH
IDIV	CHECKNULL
IREM	

Figure 7. Instructions in the (JVM-derived) Cacao interpreter VM that can throw exceptions

4. Relocatability and exceptions

Only relocatable real-machine code can be used in dynamic superinstructions (see Section 2.1). In order to be relocatable, a code fragment must not contain relative references to targets outside the code fragment, nor absolute references to targets inside the code fragment.

There are a number of JVM instructions that can throw an exception, but usually don't (see Fig. 7); e.g., `getfield` (and its quick variants) can throw a null pointer exception.

The code for throwing an exception is quite complex, so we don't want to replicate it with frequently occuring instructions like `getfield`. Moreover, it involves a function call, which makes the code non-relocatable on most architectures (it is a relative reference to code outside the fragment).

What we actually would like to do is to keep the throw code common, and jump to it from the various potentially exception-generating VM instructions. Unfortunately, when implemented directly, this usually still makes the exception-generating VM instructions non-relocatable, because the direct jump uses relative addressing on most architectures.

Our way to deal with this is to use an indirect jump instead of the direct jump. Since exceptions are rarely thrown and, when thrown, cost a lot of time anyway, the additional cost of the indirect jump is negligible.

We implement the indirect jump by putting the addresses of the throw code in a local variable, and then jumping to it with `goto *`. We have to take some care to confuse the constant propagation[7], otherwise gcc will "optimize" the indirect branch back into a direct branch. An additional problem is that we have to work around the bugs that recent gccs have in this area: PR15242 and PR25285.

[7] We make the local variable appear to be non-constant by having an assignment of another value to it in some code fragment that appears to be reachable.

Both SableVM [GH03] and Casey et al.'s work [CEG05] solve this problem in a way similar to our's[8], but they do not discuss it in their papers, and do not provide data about the effectiveness of this work.

5. Empirical results

5.1 Setup

The hardware we used in our experiments was a 2.2GHz Athlon 64 X2 4400+, a 2.26GHz Pentium 4, and a 1GHz Pentium III. The main difference between these CPUs for our experiments is in the size of the instruction cache: while the I-cache of the Athlon 64 is relatively large (64KB), it is much smaller in the Pentium III (16KB) and the Pentium 4 (12K microinstructions); so, negative effects of replication should become visible on the latter CPUs first.

All systems were running under Linux 2.6.13 or 2.6.14. We used SpecJVM98 as benchmark; we ran each benchmark three times, and report the median result.

5.2 Superinstructions

We benchmarked a threaded-code Cacao interpreter without any kind of superinstructions (**plain**), and the Cacao interpreter with dynamic superinstructions with all combinations of the following variants:

throw Instructions that can throw an exception cannot (-) or can (+) be integrated in a dynamic superinstruction (Section 4).

simple/soph The approach used for dealing with quickening: simple (Section 3.2) or our sophisticated solution (Section 3.4).

replication Without or with replication (Section 2.2).

We compiled the Cacao interpreter with gcc-2.95. We used GNU Classpath 0.19 as Java library for Cacao.

One thing worth noting is that the performance of the Cacao interpreter is strongly influenced by how many VM registers end up in real-machine registers. In the present case we managed to get the most important VM registers (ip, sp, TOS) in real-machine registers, but with more recent gcc versions, or when compiling the interpreter into a dynamically linkable library, the results are significantly worse. We used the same interpreter executable for all these measurements, with the variants determined by command-line options. This ensures that all the variants use the same register allocation.

Figure 8, 9 show the timing results; for space reasons we do not show the Pentium III results, but they are similar to the Athlon 64 X2 results.

We see that the best variant of dynamic superinstructions provides a huge speedup over plain threaded code, comparable to the effects we saw for Forth [EG03]. The speedup is even bigger on the Pentium 4 (which we did not measure earlier), probably because this CPU has a relatively higher branch misprediction penalty.

Looking at the variations, we see that **throw** has a large performance effect. By contrast, both replication and our sophisticated quickening usually have a small and not consistently positive effect on performance.

Our result for our sophisticated quickening is remarkable because the results for SableVM show a large speedup of sophisticated quickening over simple quickening [GH03]. Our explanation for this is that Cacao converts many instructions (and apparently most of the frequently-executed ones) into quick instructions already during the translation from bytecode (so there is no need to quicken them at run-time and they can be integrated into superinstructions like ordinary instructions), whereas SableVM goes through the slow-instruction stage for all slow instructions in the bytecode.

Another interesting result is that, despite Java's reputation for bloat, replication does not hurt much on any of the benchmarks, not even on the Pentium 4 and III with their small I-caches. So at least the SpecJVM98 benchmarks have good temporal locality. Implementing the non-replication option cost only three hours of work, so it may still be worthwhile (as an option) for CPUs that do not predict indirect branches with BTBs.

5.3 Other systems

Figure 10 shows the performance of various other JVM systems, both interpreters and JIT/mixed-mode systems compared to the Cacao interpreter with dynamic superinstructions.

The first interesting result is that already the **plain** Cacao interpreter (without superinstructions) is quite competetive. Surprisingly, it regularly beats even SableVM (which does use dynamic superinstructions), probably thanks to better register allocation.

The Cacao interpreter with dynamic superinstructions (**+throw soph +repl**) is quite a bit faster, as discussed above.

JIT and mixed-mode systems are generally even faster (except, usually, Kaffe). The most comparable of these is, of course, the Cacao JIT compiler, which provides speedups by up to a factor of 3.3. So, dynamic superinstructions provide performance that is halfway between plain threaded code and a JIT compiler, for much less than half the effort.

6. Related work

The work most closely related to our work is the work on dynamic superinstructions in the JVM in SableVM [GH03] and by Casey et al. [CEG05, Section 5.4]. Both papers discuss the problem of combining quickening with dynamic superinstructions; the sophisticated solu-

[8] Email communications with Etienne Gagnon and David Gregg.

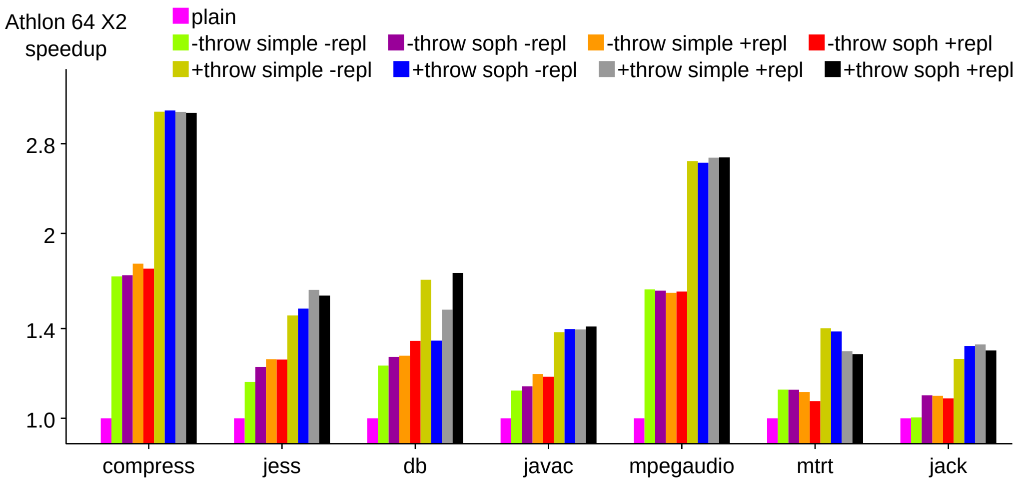

Figure 8. Speedup of dynamic superinstruction variants over *plain* on an Athlon 64 X2 4400+ (log. scale)

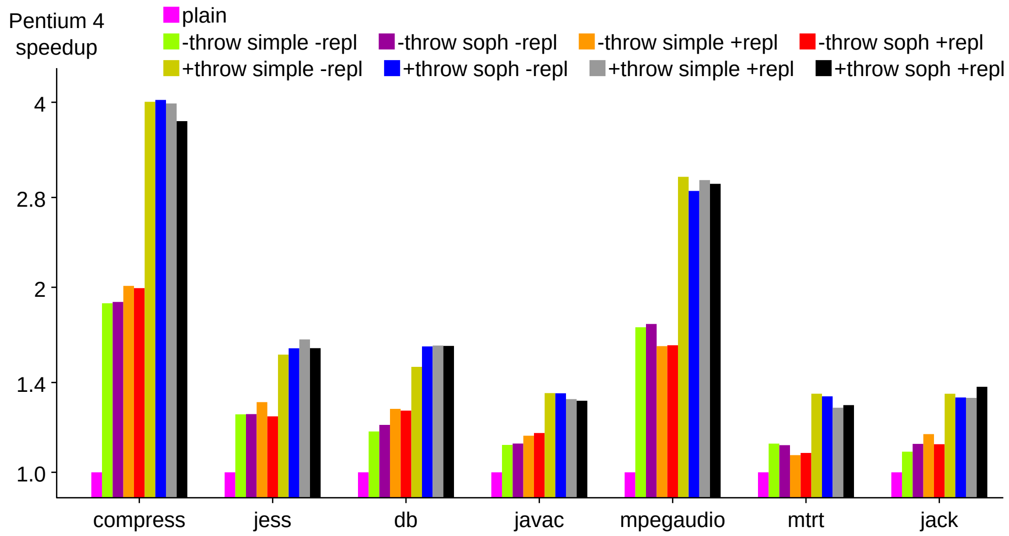

Figure 9. Speedup of dynamic superinstruction variants over *plain* on a 2.26GHz Pentium 4 (log. scale)

tions they present are more complex than our sophisticated solution (for a more detailed discussion, read Section 3.3). One significant difference is that we use a JIT-style translation, which allows us to use quick instructions right from the start in many cases, and this makes the simple approach competetive, whereas SableVM always goes through the slow instructions, and sees a big slowdown from the simple approach. Another difference between our work and the previous ones is that we discuss the issue of the relocatability of instructions that can throw exceptions, and we present results.

Choi et al. [CGHS99] point out the large effect that potential exception-throwing instructions have on a JIT compiler and present some solutions in that context, but do not discuss or solve the problems that are addressed in the present paper.

With static superinstructions the set of superinstructions is fixed at interpreter build time (or earlier). Static superinstructions, and the related, but more complex concepts of supercombinators [Hug82] and superoperators [Pro95, HATvdW99] have been used for a long time in interpreters. This includes an earlier version of the Cacao interpreter [EGKP02]; in that work we did not encounter the conflict between superinstructions and quickening, because that version of Cacao (incorrectly) initialized classes on compiling, not on first execution. So one of the advances of this work over the earlier work is a proper solution for this conflict. The other important difference between these earlier works and this work is that in this work we look at dynamic superinstructions.

Dynamic superinstructions [RS96, PR98] (also known as selective inlining) are a relatively recent invention. Replication [EG03] was developed to improve BTB indirect branch prediction accuracy, and combines nicely with dynamic superinstructions for improved perfor-

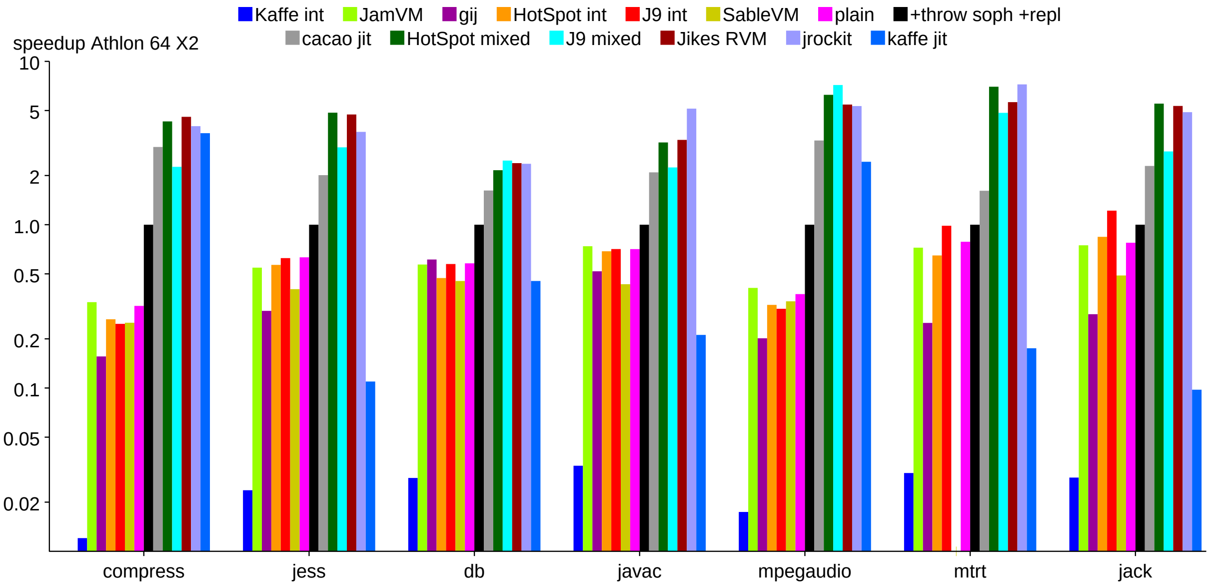

Figure 10. Relative speeds of various JVM systems on an Athlon 64 X2 4400+ (log. scale)

mance with reduced implementation effort. The present work applies these concepts in the context of the JVM, and solves the problems that arise in this context: combining dynamic superinstructions with the quickening optimization; and ensuring that VM instructions that can throw exceptions can be included in superinstructions.

As a further step after dynamic superinstructions with replication, one can generate code that includes immediate arguments and performs control flow directly instead of through the threaded code, turning the system into a simple native-code compiler. The work based on Forth [EG04b] showed a nice speedup, but the work based on Tcl [VA04] did not show a speedup over the baseline interpreter (without superinstructions or replication) for many application benchmarks, because it led to a large increase in I-cache misses. This problem would certainly also arise in an implementation of dynamic superinstructions with replication (where the resulting code is typically a little bit larger than for the more advanced technique above). Therefore, we were a little worried, how well dynamic superinstructions and especially replication would work for the JVM; we answer these questions in the present work.

7. Conclusion

Applying dynamic superinstructions and replication to the JVM poses two challenges, which we solve in this paper:

- These optimizations conflict with the *quickening* optimization for the first-execution resolution of constant-pool references. A simple approach just excludes slow instructions from dynamic superinstructions. As our empirical results show, this method works well enough in the context of a JIT-style compiler with method granularity, because it usually translates slow instructions to quick instructions already when generating the dynamic superinstruction.

  We also present a more sophisticated approach that is easier to implement than previous sophisticated approaches and is useful if the system does not use JIT-style translation to threaded code.

- Instructions that can throw an exception would normally have non-relocatable real-machine code and could not be included in dynamic superinstructions, leaving a lot of the speedup potential from dynamic superinstructions unused. We solve this problem by converting the direct branches to the throwing code (which are the cause of the non-relocatability) into indirect branches (which are relocatable).

We also present empirical results on a number of platforms: The overall speedup we see is quite large, up to a factor of 4, with a factor of about 2 being more typical. The effect of making instructions that can throw exceptions relocatable is also quite large (up to a factor of 2). Replication has a relatively small effect. A simple approach to quickening combined with a JIT-style translation into threaded code with dynamic superinstructions usually works about as well as a sophisticated approach to quickening.

Acknowledgments

The anonymous reviewers provided comments that helped improve this paper.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[CEG05] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. Submitted to ACM TOPLAS, 2005.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Program Analysis for Software Tools and Engineering (PASTE'99)*, 1999.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[EG04a] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *IVME '04 Proceedings*, pages 7–14, 2004.

[EG04b] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient Java interpreter. In *High-Performance Computing and Networking (HPCN Europe 2001)*, pages 613–620. Springer LNCS 2110, 2001.

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 170–184. Springer, 2003.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[Hug82] R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, first edition edition, 1997.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.