# Optimizations for a Java Interpreter Using Instruction Set Enhancement

Kevin Casey
Dept.of Computer Science
University of Dublin
Trinity College
Dublin 2, Ireland
Kevin.Casey@cs.tcd.ie

M. Anton Ertl
Institut für Computersprachen
TU Wien
Argentinierstraße 8
A-1040 Wien, Austria
anton@complang.tuwien.ac.at

David Gregg
Dept.of Computer Science
University of Dublin
Trinity College
Dublin 2, Ireland
David.Gregg@cs.tcd.ie

## ABSTRACT

Several methods for optimizing Java interpreters have been proposed that involve augmenting the existing instruction set. In this paper we describe the design and implementation of three such optimizations for an efficient Java interpreter. Specialized instructions are new versions of existing instructions with commonly occurring operands hardwired into them, which reduces operand fetching. Superinstructions are new Java instructions which perform the work of common sequences of instructions. Finally, instruction replication is the duplication of existing instructions with a view to improving branch prediction accuracy. We describe our basic interpreter, the interpreter generator we use to automatically create optimised source code for enhanced instructions, and discuss Java specific issues relating to these optimizations. Experimental results show significant speedups (up to a factor of 3.3, and realistic average speedups of 30%-35%) are attainable using these techniques.

## Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

## Keywords

Interpreter, Virtual Machine, Java

## 1. MOTIVATION

Interpreters have a number of advantages that make them attractive for implementing programming languages. These advantages include simplicity, ease of construction and maintanence, trivial retargetability, low memory requirements, and a fast modify-compile-run cycle. The main weakness of interpreters is that they are slow. Even very efficient interpreters are typically about ten times slower than compiled code [11]. The goal of our work is to narrow that gap, by applying speed optimizations to Java virtual machine (VM) interpreters.

One such class of optimizations are techniques that augment the VM's instruction set with new instructions that require less interpretive overhead. In this paper we describe the design and implementation of three such optimizations for an efficient Java interpreter. Specialized instructions are new versions of existing instructions with commonly occurring operands hardwired into them, which reduces operand fetching. Superinstructions are new Java instructions which perform the work of common sequences of instructions. Finally, instruction replication is the duplication of existing instructions to improve branch prediction accuracy.

Traditionally, adding new VM instructions to an interpreter made it much less maintainable, because they increased the size of the source code. We use an interpreter generator to automate the profiling of VM code to identify likely candidate instructions, to produce optimized source code for these augmented VM instructions from a specification of the unoptimized version, and to automatically rewrite VM code at run-time to take advantage of these augmented instructions.

This paper describes the design and implementation of our interpretive system, and presents extensive experimental results on the performance of these optimizations. Many of these optimizations have been used on an *ad hoc* basis for many years to speed up interpreters. Our contributions are to study them in a systematic way, to show the extent to which such optimizations to be automated, to present our experience of making them work, and to present experimental results on their actual, rather than presumed, effectiveness.

The rest of this paper is organized as follows. In section 2 we describe our basic interpreter system. Section 3 describes how instructions can be specialized for common immediate operands to reduce operand fetch. In section 4 we describe our system of automatically generating superinstructions — frequently occurring sequences of instructions that are merged together to form a new VM instruction. Section 5 evaluates replicating the code to implement VM instructions in order to reduce branch mispredictions. Finally, section 6 places our research in the context of other work in the area.

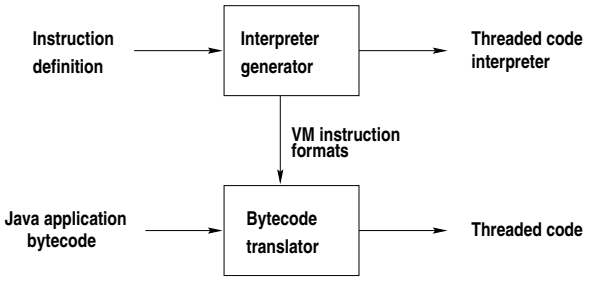


**Figure 1: Structure of the Interpreter System**

```
IADD ( iValue1 iValue2 -- iResult )
{
  iResult = iValue1 + iValue2;
}
```

**Figure 2: Definition of `IADD` VM instruction**

## 2. INTERPRETIVE SYSTEM

Figure 1 shows the structure of our interpreter system. Our system is based around tiger[1] a generator that reads in a simple description of the VM instruction set, and generates optimized C source code for a VM interpreter.

The *instruction definition* describes the behavior of each VM instruction. The definition of an instruction consists of a specification of the effect on the stack, followed by C code to implement the instruction. Figure 2 shows the definition of `IADD`. The instruction takes two operands from the stack (`iValue1,iValue2`), and places the result (`iResult`) on the stack. This relatively high-level description of the VM instruction allows our generator to perform a lot of optimizations, such as the ones described in this paper.

Our generated interpreter is a direct threaded-code interpreter [1]. That is, VM instructions are represented in memory as the addresses of the routines that implement them. In addition to being more efficient than a bytecode interpreter, it also allows us to experiment with a wide range of optimizations, because more than 256 VM instructions can be used. However, the bytecode must be translated to threaded code before it is executed. In the process, we perform a number of optimizations, such as combining multiple-byte immediate operands into a single operand.

Our interpreter has been built into the CVM, an implementation of the Java 2 Micro Edition (J2ME), which provides a core set of class libraries, and is intended for use on devices with up to 2MB of memory. Our new interpreter replaces the existing interpreter in CVM.

To act as reference point for experimental results in following sections, table 1 shows a comparison of the absolute running times of the SPECjvm98 [14] benchmarks on three different JVMs. The first is our base interpreter without any of the optimizations described in this paper. We also show running times for Sun's HotSpot 1.4.2 mixed-mode interpreter and JIT compiler, and for HotSpot using only the

[1]Tiger further develops the ideas in Ertl et al's `vmgen` [6] (see section 6).

| Benchmark | Our Base interpreter | Hotspot interpreter | Hotspot mixed-mode |
|---|---|---|---|
| javac | 33.45 | 27.92 | 6.85 |
| jack | 18.12 | 16.62 | 3.15 |
| mpeg | 90.51 | 81.89 | 5.47 |
| jess | 27.7 | 21.48 | 3.10 |
| db | 66.29 | 47.68 | 15.05 |
| compress | 110.46 | 88.91 | 7.94 |
| mtrt | 31.27 | 28.93 | 2.24 |

**Table 1: Comparison of running time of our base interpreter with the Sun HotSpot Client VM Interpreter, and mixed mode interpreter–JIT compiler on the SPECjvm98 benchmark programs.**

```
GETFIELD_QUICK ( aObjectPtr #iOffset -- iResult )
{
  if ( aObjectPtr != NULL )
    iResult = *(aObjectPtr + iOffset);
  else
    NULL_POINTER_EXCEPTION();
}
```

**Figure 3: Definition of `IGETFIELD` VM instruction**

interpreter. The modified version of CVM used for these tests was compiled under GCC 2.96. The hardware used to run the benchmarks was a Pentium 4 2.66 Ghz based PC with 512 MB of memory, running Mandrake Linux 9.2.

Overall, our interpreter is on average 15% slower than the Hotspot interpreter. There are two main reasons for this. Firstly, Hotspot has a much faster run time system than CVM. Secondly the Hotspot interpreter is faster than our interpreter. Its dynamically-generated, highly-tuned assembly language interpreter is able to execute bytecodes more quickly than our portable interpreter written in C. Finally, the mixed-mode compiler- interpreter provides a reference point to compare our interpreter against compiled native code. The following sections describe how our interpreter can be further optimized by enhancing the instruction set with new opcodes.

## 3. INSTRUCTION SPECIALIZATION

Many JVM instructions take immediate operands from the instruction stream when executing. Fetching these operands from memory is part of the overhead of interpretation. Specialized instructions are new versions of existing instructions with commonly occurring operands hardwired into them, to reduce operand fetching. Typically, the machine code for a specialized instruction can be much more efficient, not only because it usually eliminates a load, but also because the compiler can optimize the code for that particular constant.

### 3.1 Implementation

Figure 3 shows the tiger definition for `GETFIELD_QUICK`, which is used to fetch the value of in field of an object. This instruction takes an immediate argument from the instruction stream which specifies the offset of the field within the object. `GETFIELD_QUICK` is one of the most frequently executed instructions in our JVM (around 8% of all instructions). The offsets are most commonly one of just a few values, so

```
LABEL(GETFIELD_QUICK_0) /* start label  */
{
int aObjectPtr;     /* declaration of.. */
int iOffset;        /*  ...stack items  */
int iResult;
aObjectPtr = *sp;   /* fetch stack items */
iOffset = 0;        /* immediate operand */
{               /* user provided C code */
  if ( aObjectPtr != NULL )
    iResult = *(aObjectPtr + iOffset);
  else
    NULL_POINTER_EXCEPTION();
}
*sp = iResult;   /* store stack result */
ip++;            /* update VM ip */
}
NEXT;            /* dispatch next instr. */
```

**Figure 4: Simplified tiger output for `GETFIELD_QUICK` VM instruction specialized with the immediate operand 0.**

it may make sense to generate specialized versions for each of these constants.

Figure 4 shows the generated C code for `GETFIELD_QUICK`, specialized with the immediate operand 0 (a particularly common case). Whereas the generated code would normally load the offset from the instruction stream, in the specialized version it is simply set to the chosen constant. Although this code looks long and complicated, the C compiler will optimize it well. In particular, constant propagation will eliminate the add of the offset entirely in this case.

Interestingly, the JVM instruction set includes quite a number of already specialized instructions. For example, there are four versions of each of the load and store instructions for local variables, for each of the first four local variables. However, we have chosen to convert these specialized instructions into their generic form, and implement our own generic system for creating specialized instructions. There are three main reasons for this decision.

First, by converting to generic versions, we increase the opportunities for using superinstructions. For example the sequence `ILOAD_0 IADD` could use the superinstruction `ILOAD-IADD`, whereas it is not practical to have large numbers of specialized superinstructions, such as `ILOAD_0-IADD`.

Secondly, the standard specialized instructions in the JVM appear to have been chosen on an *ad hoc* basis, with little attention how often it appears in real code. For example, the instruction `FSTORE_0` does not appear even once in in all the SPECjvm98 benchmarks [16]. We would like to choose the instructions to specialize based on real measurements rather than presumed usefulness.

Finally, the immediate operand is not known for many VM instructions until the first time they are executed, so they cannot be specialized in JVM bytecode. For example, the offset for a `GETFIELD` instruction may not be known until the first time it is executed, because it may access another class which may not yet be loaded. Given that field access instructions account for about 16% of executed instructions in the SPECjvm benchmarks, this greatly reduces the potential for exploiting specialization.

Tiger supports specialized instruction in three ways. First, it allows us to automatically generate a version of the interpreter which profiles the value of all immediate operands for each instruction as it executes. Based on this profiling information, we can choose the best combinations of instructions and immediate operands to specialize. Secondly, tiger generates C source code for to implement specialized instructions, from the instruction definitions and the output of the profiler. Finally, tiger also generates C routines to automatically replace VM instructions and their operands with specialized versions. Thus, almost the entire process of creating specialized instructions is automated.

## 3.2 Evaluation

An important question is how specialized instructions should be chosen. If we want to customize the interpreter for a particular program, we just choose the most commonly executed combinations in a test run of that program. If, on the other hand, the program is not available, then we would like to choose a representative set from profiles of several other programs.

We evaluated instruction specialization using our interpreter system and the specJVM98 benchmarks. We selected specialized instructions using three strategies: (1) the dynamically most frequently executed combinations for this particular program, (2) the dynamically most frequently executed combinations in all specJVM98 programs except this program, and (3) the most frequent combinations appearing statically in the code of all other programs.

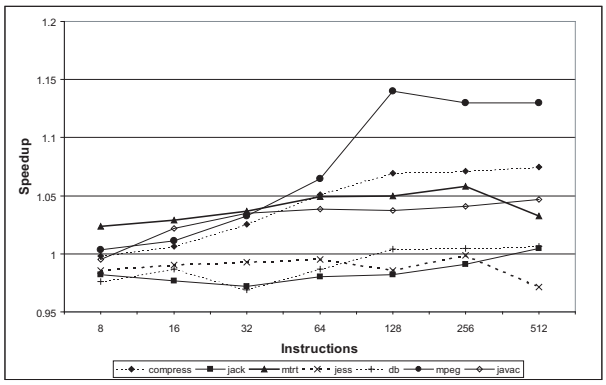


**Figure 5: Speedup from varying numbers of specialized instructions chosen based static frequency in other programs.**

Surprisingly, adding small numbers of specialized instructions to our interpreter actually makes it slower. We used the Pentium 4's hardware performance counters to investigate this. As expected, we found that specialized VM instructions reduce the number of x86 native machine instructions needed to execute the interpreter. Normally, we would expect a corresponding reduction in execution time. However, we also found that specialized instructions also impact on indirect branch prediction rates, which has a much large effect on running time.

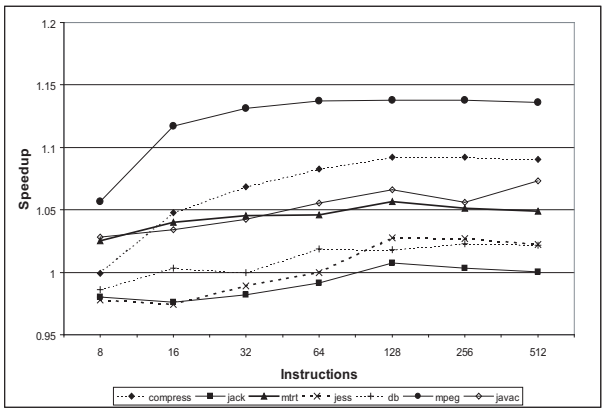


**Figure 6: Speedup from varying numbers of specialized instructions chosen specifically for a program.**

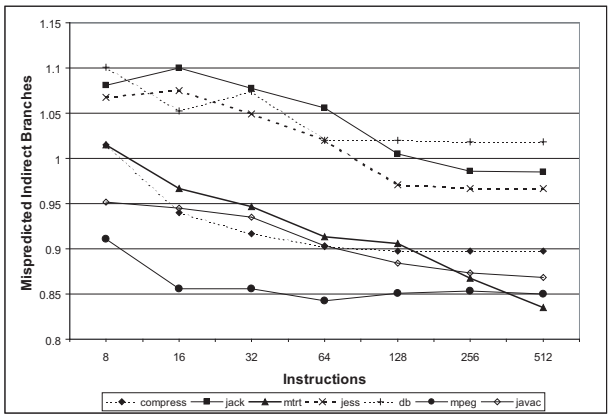


**Figure 7: Percentage change in indirect branch mispredictions from using specialized instructions chosen specifically for a program.**

Figure 7 shows the ratio of the number of indirect branch mispredictions for various configurations compared to the interpreter without specialized instructions. Small numbers of specialized instructions result in more mispredicted indirect branches. Given the high cost of branch mispredictions (around 20 cycles on the P4) and that dispatching each VM instruction involves executing an indirect branch, a reduction in misprediction accuracy has a much larger effect on execution time than a small reduction in executed instructions.

Further investigation showed the source of the problem. Unlike original JVM bytecode, our base interpreter uses no specialized instructions, and it does not have separate versions of VM instructions for different types, except where they require different code. We made a deliberate decision to minimise the number of VM instructions to facilitate superinstructions. The result is that JVM instructions such as `ALOAD`, `ILOAD` and `FLOAD_1` are all implemented with the same code. Given that local loads account for around 35% of all executed instructions [16], this code is an extremely frequent target for the indirect branches that implement VM instruction dispatch. When we introduce specialized versions of this local load, we no longer have a single common target, and indirect branch prediction accuracies fall.

However, another effect is also in play. There is a separate indirect branch at the end of the code to implement each VM instruction. When we specialize an instruction, we introduce a new implementation, with its own indirect branch. It is important to recall that current processors use a branch target buffer (BTB) to predict indirect branches, which simply predicts that the target address will be the same as on the previous execution of the same branch. By having a separate indirect branch (and thus a separate BTB entry) for each specialized local load instruction, we capture some context about the program. For example, there may be several `LOAD` instructions in a method, but only `LOAD_6` is followed by `IADD`. As the number of specialized instructions rises, the benefit of more separate indirect branches outweighs the cost of a larger number of targets, and the net effect is positive. We also measured an increase in instruction cache (trace cache) misses, but the effect was much lower than that on indirect branch prediction.

In summary, instruction specialization is used by many VM interpreters to reduce the overhead of inline immediate operands. However, the main performance impact of specialized instructions is their effect on indirect branch prediction accuracy, rather than reduced operand fetches. Thus, the overall effect is somewhat unpredictable.

# 4. SUPERINSTRUCTIONS

A superinstruction is a new virtual machine instruction that consists of a sequence of several existing VM instructions. There are a number of benefits associated with this. One is that superinstructions reduce the number of VM instruction dispatches required to perform a certain sequence of instructions. This is quite important since instruction dispatch has been shown to be a particular bottleneck in interpreters [5].

Another benefit superinstructions provide is the opportunity to optimise the interpreter source code. For example, it is

common that the result written to the stack by one instruction will be read from the stack by the following one. When generating C source code to implement superinstructions, tiger eliminates the stack read and writes, and instead keeps the value in a local variable between the two component instructions. A third benefit associated with superinstructions is that combining the source code for instructions together exposes a larger "window" of code to the C compiler, which allows greater opportunities for optimisation.

There are two main ways that superinstructions are used. The first is to generate an interpreter that is optimized for a particular program. In this case we will select superinstructions specific to that program. Selecting the optimal set of superinstructions for a given program is NP-hard [13]. We experimented with a number of heuristics, such as finding the most frequently executed (sub)sequences of VM instructions, in a scheme similar to Proebsting's [13]. Eventually we found that by far the best scheme is to simply select the $n$ most frequently executed basic blocks in the program to be superinstructions.

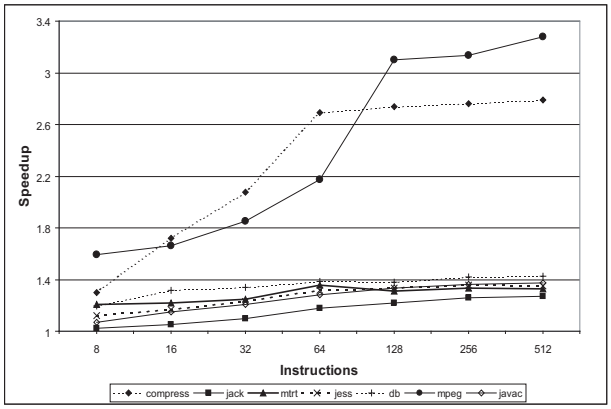


**Figure 8: Adding individually tailored superinstructions to cvm.**

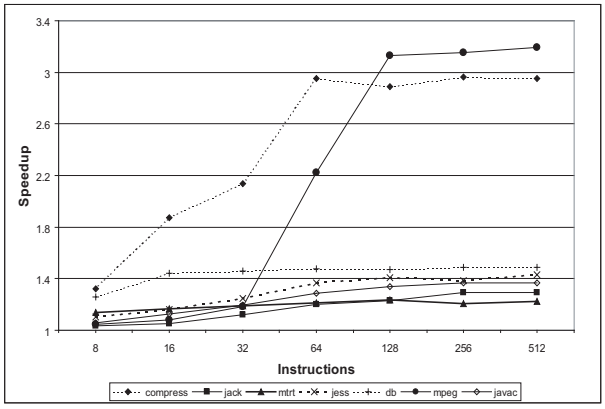


**Figure 9: Adding individually tailored superinstructions across basic blocks to CVM.**

Figure 8 shows the speedups achieved using varying numbers of superinstructions. The figures are quite impressive for some benchmarks, in particular compress and mpegaudio, with speedups of 2.8 and 3.3 respectively at 512 superinstructions. Other benchmarks do not benefit so well from this approach but nonetheless the minimum speedup is 1.27 (jack) at 512 superinstructions which is quite reasonable. It is worthwhile noting that, even with only 32 superinstructions, the minimum speedup across all benchmarks is 1.1.

| Scheme | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|
| static | 26.3 | 27.7 | 30.7 | 33.0 | 38.7 | 41.6 | 44.2 |
| static norm. | 26.3 | 27.3 | 30.7 | 33.6 | 38.8 | 40.8 | 44.0 |
| static ldiv | 23.7 | 28.1 | 30.3 | 34.5 | 39.6 | 44.5 | 48.3 |
| static ldiv norm. | 23.7 | 27.8 | 29.8 | 34.4 | 38.7 | 44.2 | 47.4 |
| static lmul | 16.7 | 17.9 | 19.0 | 19.1 | 19.7 | 21.0 | 21.3 |
| static lmul norm. | 19.3 | 20.4 | 21.3 | 23.5 | 24.6 | 25.1 | 25.8 |
| dynamic | 24.1 | 26.0 | 30.2 | 31.8 | 33.2 | 34.9 | 36.5 |
| dynamic norm. | 23.8 | 28.8 | 32.4 | 35.4 | 37.7 | 42.3 | 43.4 |
| dynamic ldiv | 23.8 | 26.4 | 29.7 | 33.6 | 37.4 | 41.2 | 44.1 |
| dyn. ldiv norm. | 23.8 | 26.7 | 30.9 | 35.7 | 40.8 | 44.4 | 47.2 |
| dyn. lmul | 1.3 | 1.3 | 1.3 | 2.5 | 2.6 | 2.9 | 3.0 |
| dyn. lmul norm. | 15.5 | 15.9 | 17.3 | 18.5 | 18.6 | 19.8 | 20.4 |

**Table 2: Comparison of superinstruction selection strategies.**

A more generally useful way to use superinstructions is to try to select a generic set of superinstructions which will be useful across a wide range of currently unknown programs. Perhaps the best approach is to examine the profiles of several programs and identify the most common sequences of instructions. For each program, we selected superinstructions based on the profiles of all other programs. For these experiments we used only the actual benchmark's program code, and not any library code, which would be common to all programs.

We tested several criteria for selecting superinstructions (see table 2). We tested measuring the *static* number of times that each sequence appears in the code as well as its *dynamic* execution frequency. In order to avoid one large program dominating the others, we *normalized* the frequencies to percentages of total static/dynamic instructions in the program. Originally, we felt that longer sequences were more desirable, because they eliminate more dispatches, so we tried multiplying the frequencies by the length of the superinstruction minus one (*lmul*). It also occurred to us that shorter superinstructions might be easier to reuse, so we also tried dividing by the number of dispatches removed by the superinstruction (*ldiv*).

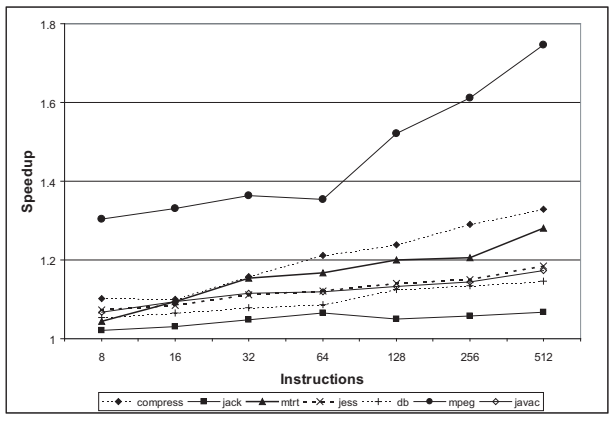


**Figure 10: Adding statically selected superinstructions to CVM.**

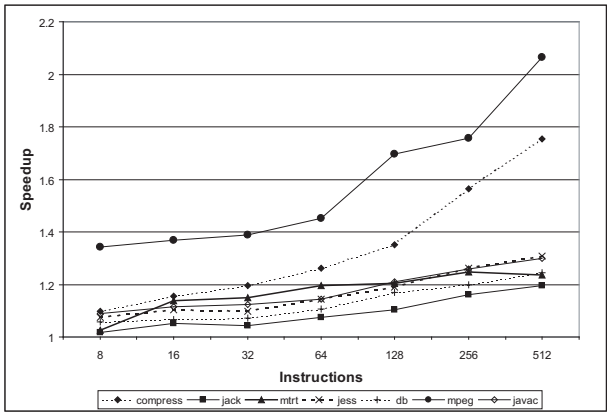


**Figure 11: Adding statically selected short superinstrunctions to CVM.**

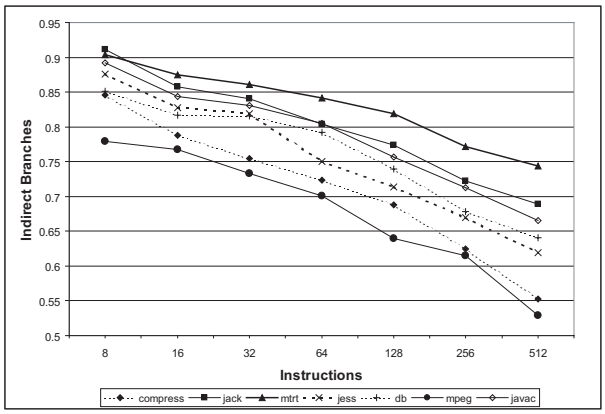


**Figure 12: Indirect branch reduction due to statically selected short superinstructions.**

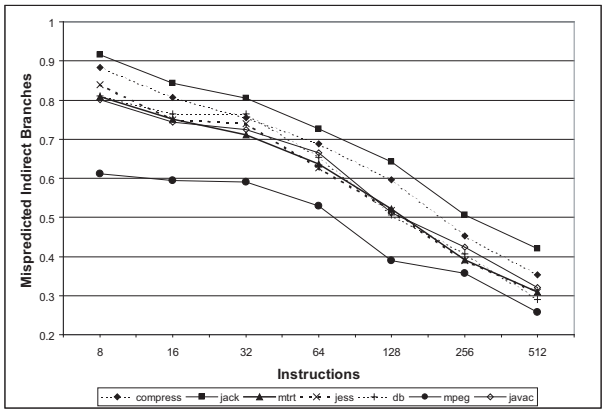


**Figure 13: Mispredicted indirect branch reduction due to statically selected short superinstructions.**

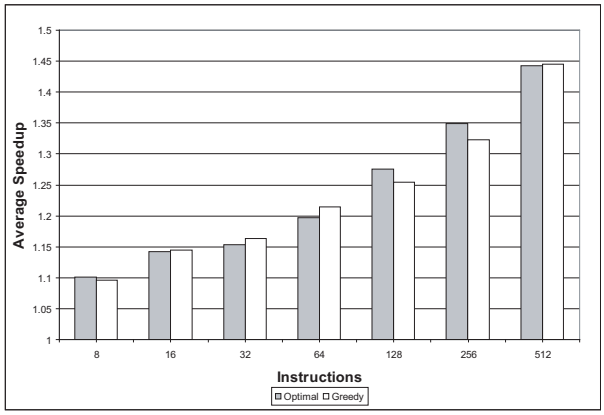


**Figure 14: Comparison of optimal vs greedy parsing strategies for statically selected superinstructions.**

Table 2 shows the average reduction in VM instruction dispatches across all benchmarks using different combinations of the selection strategies and varying numbers of superinstructions. A number of trends are clear. Dynamic frequency performs worse than static, because it is biased very strongly in favor of the inner loops of the programs.

We use the interpreter generator `tiger` based on `vmgen` [6] to allow us to generate superinstructions using profiling information. `tiger` takes in an instruction definition, and outputs an interpreter in C which implements the definition. The interpreter generator translates the stack specification of the instruction definition into pushes and pops of the stack, adds code to invoke following instructions, and makes it easy to apply optimizations to all virtual machine instructions, without modifying the code for each separately. `tiger` allows for some extra functionality over `vmgen` including support for superinstructions across basic blocks.

The main determinant of the usefulness of superinstructions is whether the sequences we choose to make into superinstructions account for a large proportion of the running time of the programs that run on the interpreter. The set of superinstructions must be chosen when the interpreter is constructed, most likely at a time when one doesn't know which programs will be run on the interpreter. Thus, one must somehow guess which superinstructions are likely to be useful for a set of programs that one has never seen.

The most common way to make guesses at the behaviour of unseen programs is to measure the behaviour of a set of standard benchmarks programs, and hope that these benchmarks resemble the real programs. A question remains, however, as to how the benchmarks should be measured to identify useful superinstructions. Gregg and Waldron [10] tested a wide range of strategies for choosing superinstructions for Forth programs. They found, perhaps surprisingly, that the best strategy was to simply choose those sequences that appear most frequently in the static code. This was the next strategy we chose. More specifically, for each benchmark X in our set of benchmarks, we used profiling data from every other benchmark in the set (excluding benchmark X) to generate a list of most commonly occurring basic blocks.

These basic blocks were used to create superinstructions to be added to a JVM to be tested on benchmark X.

Results are presented in figure 10 for this strategy. The results are graphed similarly to before, with results presented for each benchmark with a varying number of superinstructions. The speedups obtained with this approach were much more conservative using this approach. At 512 superinstructions, the maximum speedup was 1.75 (mpeg) and the minimum 1.07 (jack). At 32 superinstructions the maximum speedup was 1.36 (mpeg) and the minimum 1.05 (jack).

Analysis of the superinstructions selected using the strategy above yielded some interesting results. It appeared that some long sequences from a limited number of benchmarks were dominating the statistics. In an attempt to reduce this effect, we decided to bias the statistics in favor of shorter, and therefore more commonly occurring (across benchmarks), sequences. In order to do this, we used precisely the same superinstruction selection strategy as before, but this time each superinstruction's weight(previously its static frequency) was divided by its length-1. This biases the selection strategy heavily in favor of shorter sequences.

Results were generated exactly as before, but this time using the modified weightings to decide which superinstructions to include. The results are presented in figure 11. The speedups were considerably better with this minor modification. At 512 superinstructions, the maximum speedup was 2.06 (mpeg) and the minimum 1.19 (jack). At 32 superinstructions the maximum speedup was 1.39 (mpeg) and the minimum 1.04 (jack). These results show two interesting points. Firstly, the superinstruction selection scheme is critical. Even small changes in the selection algorithm can have dramatic effects. Secondly, there is two opposing goals in that we would like choose long superinstructions (to eliminate as many dispatches as possible) but shorter superinstructions can be applied at more points in the code. More sophisticated selection algorithms will need to be examined to throw more light on this aspect of superinstructions.

Superinstructions, by virtue of eliminating dispatches, reduce the number of indirect branches and the consequently the number of indirect branch mispredictions. This is illustrated in figures 12 and figures 13 respectively. These represent indirect branch measurements and mispredicted indirect branch measurements for the same benchmarks and selection strategy presented in figure 11. Normally one would expect that the misprediction of indirect branches as a proportion of indirect branches stays more or less constant as superinstructions are added to the JVM. If this were the case, both the indirect branch counts and mispredicted indirect branch counts would be decreasing at the same rate as superinstructions are added. However, looking at figures 12 and 13 it can be seen that the number of indirect branch mispredictions decreases more sharply than the number of indirect branches. From an average misprediction rate across all benchmarks with 0 superinstructions of 45%, the rate drops substantially, down to a little over 23% at 512 superinstructions.

The explanation for this is twofold. The addition of each superinstruction adds an extra entry to the Branch Target Buffer (subject to BTB size). Also, by the time a dispatch occurs at the end of a superinstruction, we are guaranteed to have executed a certain sequence of component instructions. This has the effect of adding context to the dispatch at the end of the superinstruction in question, and makes branch prediction at that point more accurate as a result.

One complication in a Java interpreter is that the JVM comes with a large library of classes that are used internally by the JVM and by running programs. Approximately 33% of the executed bytecode instructions in the SPECjvm98 benchmark suite [14] are in library rather than program methods [16]. This library code is available at the time the interpreter is built, so there is potential for choosing superinstructions specifically for commonly used library code.

## 4.1 Parsing

The use of superinstructions is in many respects the same problem as dictionary-based text compression [2]. Dictionary-based compression attempts to find common sequences of symbols in the text, and replaces them with references to a single copy of the sequence. Thus, when designing a superinstruction system, we can draw on a large body of theory and experience on text compression.

Parsing is the process of modifying the original sequence of instructions by replacing some subsequences with superinstructions. The simplest strategy is known as *greedy parsing*, where at each VM instruction we search for the longest superinstruction that will match the code from that point.

To be guaranteed to find the best possible parse, an optimal parsing algorithm must be used. Fortunately, optimal parsing can be solved using dynamic programming [2], so efficient algorithms are available. Our interpreter currently allows for either greedy or optimal parsing to be selected at compile time. Tiger generates parsing tables that remove this requirement, so we are free to add superinstructions to a JVM without ensuring all subsequences are present. This will allow us to exploit more advanced superinstruction selection strategies.

All results in this paper are presented using the optimal parsing algorithm except for those presented in figure 14 where we present a comparison of greedy parsing versus optimal parsing for the same selection strategy as used in figure 11. This comparison shows that there is not a huge difference between optimal parsing and greedy parsing in terms of performance. Indeed sometimes greedy gives a better speedup. The extra computational overhead required for an optimal parse is the most likely reason for this, where it occurs. It can be quite possible that, with superinstruction selection strategies other than that used for the comparison in figure 14, that the difference between an optimal parsing process and a greedy one would be more noticeable.

## 4.2 Quick Instructions

Several Java bytecode instructions must perform various class initialisations the first time that they are executed. On subsequent executions no initialisations are necessary. A common way to implement this functionality is with "quick" instructions. The first time a given instruction of this type is executed, it performs the necessary initialisations, and then

replaces itself in the instruction stream with a corresponding quick instruction, which does not do these initialisations. On subsequent executions of this code, the quick instruction is executed.

Quick instructions are vital to the performance of most Java interpreters, since the check for class initialisation is expensive, and because they are among the most commonly executed instructions. For example, in the SPECjvm98 benchmarks `GETFIELD` and `PUTFIELD` account for about one sixth of all executed instructions, and run very slowly unless converted to quick versions [16]. Eller [3] found that adding quick instructions to the Kaffe interpreter could speed it up by almost a factor of three.

A problem with quick instructions is that they make it difficult to replace sequences of instructions with superinstructions. No instruction that will be replaced with another instruction at run time can be placed in a superinstruction, since that would involve replacing the entire superinstruction. Furthermore, some instructions, such as LDC (load constant from constant pool) and INVOKEVIRTUAL become different superinstructions depending on the value of their inline arguments, or the type of class or method they belong to.

An additional complication when dealing with non-quick instructions is race conditions. Due to the threaded nature of the Java interpreter, during quickening it is quite possible for two threads to almost simultaneously access a non-quick instruction triggering a potential race condition. Such race conditions are avoided in the current implementation of CVM by using mutually exclusive locks, but adding support to allow quickened instructions to become part of a superinstruction after translation could lead to race conditions.

Our current implementation allows for "quick" instructions to be components in superinstructions by utilizing a simple approach. Each time an opcode is quickened in a method, the method is re-parsed in an attempt to incorporate that newly created quick instruction into a superinstruction. This approach does include some overhead but this overhead seems to be relatively insignificant in relation to other inefficiencies in the interpreter such as branch misprediction. If one wanted to reduce the parsing overhead, a possibility is just to reparse the basic block in which the instruction has been quickened. The difficulty with this approach, however, is that we currently have the capability of having superinstructions that span basic blocks (see below). Thus we cannot limit a re-parsing to the basic block in which an instruction was quickened if we want to attempt to use these longer superinstructions.

## 4.3 Across Basic Blocks

Superinstructions are normally only applied to instructions within basic blocks. However, with relatively small modifications, it is possible to extend superinstructions across basic block boundaries in two specific situations. First, we consider control flow joins. A join is a point in the program with incoming control flow from two or more different places. Usually one of those places is simply the proceeding basic block, and control falls through to the join without any branching. In these cases, the falling-though code is simply a straight-line sequence of instructions. However, it is not normally safe to allow a superinstruction to be formed across the join, because it would not then be clear where the other incoming control-flow paths should branch to.

```
        ILOAD                   ILOAD
        4                       4
        ILOAD                   ILOAD-IADD
        5                       5
join:   IADD            join:   IADD
        ISTORE                  ISTORE
        6                       6
```

**Figure 15: Original code (left) and same code with `ILOAD-IADD` superinstruction (right).**

The solution we use is to create superinstructions, but not to remove the gaps that are created by eliminating the original instructions. In fact, we leave the original instructions in these gaps. Figure 15 shows an example of, where we have replaced the sequence `ILOAD`, `IADD` with the superinstruction `ILOAD-IADD`. We actually replace the `ILOAD` instruction with `ILOAD-IADD`, but leave the IADD instruction where it is. When we fall-through from the first basic block to the second, we execute `ILOAD-IADD`, which performs its normal work and then skips over the `IADD` instruction. On the other hand when we branch to the second basic block from elsewhere, we branch to the `IADD` instruction which executes and continues as normal. This scheme allows us to form superinstructions across fall-though joins.

We believe that this scheme is particularly valuable for `while` loops. The standard *javac* code generation strategy appears to be to place the loop test at the end of the loop, and on the first iteration to jump directly to this test. Unfortunately, the result is that there is a control flow join just before the loop test that would normally hinder optimisation. We believe we have successfully overcome this problem.

A second opportunity for cross-basic block superinstructions is with the fall-through direction of VM conditional branches. Superinstructions are permitted to extend across branches due to faclities provided by tiger. Figure 16 shows the instruction definition for a branch instruction. Inside the `if` statement the tiger keyword `TAIL` is used to specify that a copy of the dispatch code that normally appears at the end of the instruction should be placed here.

The TAIL macro in tiger is redefined at the beginning of every component instruction in a superinstruction to allow a branch out of the superinstruction at that point, if required. Thus a single superinstruction can be generated that spans multiple untaken branches. Of course if the branches are

```
IFNULL ( #aTarget aRef -- )  0xc6
{
  if ( aRef == NULL ) {
    SET_IP(aTarget);
    TAIL;
  }
}
```

**Figure 16: Definition of a branch VM instruction**

taken, the TAIL macro will flush items from local variables to the stack and update the stack pointer before a branch out of he superinstruction.

Tiger also redefines a `FLUSH_ALL` macro that carries out a similar task to TAIL (but without the dispatch) before each instruction. This was necessary for certain 'superinstruction-unsafe' instructions such as `ANEWARRAY_QUICK` which may trigger a garbage collection. When a GC occurs, the stack pointer and items on the stack should be up to date. Previously we could not allow such 'superinstruction-unsafe' instructions to be components in a superinstruction because the stack pointer had not been up to date and stack items were still in local variables (ready to be written later in the superinstruction). Now, if the `FLUSH_ALL` macro is used before such instructions, all items will be flushed to the stack before the instruction (and hence before the GC takes place). Elimination of stack pointer updates and accesses using local variables can resume for subsequent component instructions in the superinstruction after the 'superinstruction-unsafe' instruction has completed Using this mechanism, a large number of instructions that previously were unsafe to include in superinstructions can now be used.

Using a similar method to select superinstructions as that in figure 8 we created a set of CVM binaries containing superinstructions tailored to each individual benchmark. The difference is that instead of using the most commonly occurring basic blocks we now use the most commonly occurring regions. The start point for these regions can be any join point and the end point can be any branch or invocation. The results are presented in figure 9. The speedups obtained by using the longer regions instead of shorter basic blocks appear to be reasonable. Most benchmarks had some sort of speedup with the exception of mtrt. 31 of 49 JVM runs registered an improvement over basic block selection (7 benchmarks with 7 runs per benchmark). We feel that a superinstruction selection strategy tailored to exploit this new capability of superinstructions across basic blocks can yield even better results.

# 5. INSTRUCTION REPLICATION

Instruction replication aims to reduce indirect branch mispredictions in a threaded code interpreter by creating multiple versions of the code to implement commonly occuring VM instructions. Each separate version is ended by an indirect branch to dispatch the next VM instruction. By varying the version of the implementation code that is used in different parts of the Java program, some context information is captured which may improve branch prediction accuracy.

For example, consider a loop containing the following sequence of instructions: `ILOAD IADD ILOAD ISUB`. If there is only one version of the `ILOAD` code, then the indirect branch at the end of the code to implement `ILOAD` will constantly switch between the two and indirect branch prediction accuracy will be very poor. If, on the other hand, we have two versions of `ILOAD`, then this could could be rewritten as `ILOAD_rep1 IADD ILOAD_rep2 IADD`. There are separate indirect branches at the end of each replication, both of which are now almost perfectly predictable.

## 5.1 Implementation

Tiger supports instruction replication by (1) generating a profiling version of the interpreter to collect profiles of instruction frequencies, (2) automatically generating copies of the C source code to implement VM instructions and, (3) generating C source code to rewrite the VM code with replicated instructions. When several versions of a VM instruction exist, the versions are used in round-robin order when rewriting the VM code. This usually ensures that the same version of a VM instruction is not used more than once in a basic block.

An important question is which VM instructions should be chosen for replication, and how many copies of each should be created. Based on profiles, we computed the frequency of each VM instruction. We added one replication of the most frequent VM instruction, and reduced its frequency by splitting the frequency between each of the copies. We then applied the same process again, until we had chosen the required number of total replications. Note that this process can result in the same instruction being replicated multiple times, because even its split frequency may be higher than that of other instructions.

## 5.2 Evaluation

The frequency of an instruction might be measured by its static or dynamic frequency, and we might try optimize the interpreter for a given program, or look at several programs to find generally useful sets of replicas. We found that there was almost no additional benefit from customizing the replicated instructions for a particular program. Static measures of frequency perform a little worse than dynamic measures.

Figure 17 shows the effect on running time of using varying numbers of instructions selected based on their dynamic frequency in all SPECjvm98 programs except the one being measured. As with instruction specialization, adding small numbers of replicated instructions actually makes the interpreter slower. As in section 3, we investigated this using the Pentium 4's hardware instruction counters, and found that adding small numbers of replications increases the branch misprediction rate (see figure 18), because of the very frequent local load instruction being replicated. There was also a significant increase in the number of instruction cache (trace cache) misses. As the number of replications increases, the reduction in indirect branch mispredictions outweighs the other costs, and there is a significant net speedup (almost a factor of two for compress).

# 6. RELATED WORK

The Sable VM [8] is an interpreter-based research JVM. This interpreter uses a run-time code generation system [12], not dissimilar from a just-in-time compiler. Sable uses a novel system of *preparation sequences* [9, 7] to deal with bytecode instructions that perform initialisations the first time they are executed, which make code generation difficult. The same process would be an alternative to our solution for implementing superinstructions in the presence of instruction "quickening".

Venugopal et al. [15] proposed optimizing Java interpreters for embedded systems using *semantically enriched code* (sEc). The idea of sEc is to profile the application, and generate specialized superinstructions specifically for that applica-

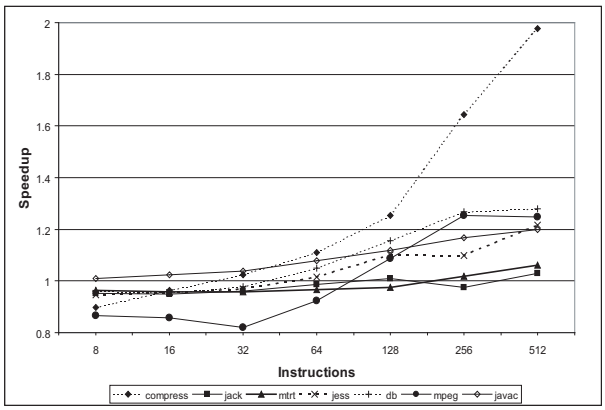


**Figure 17: Speedup from replicated instructions chosen using dynamic frequency in other programs.**

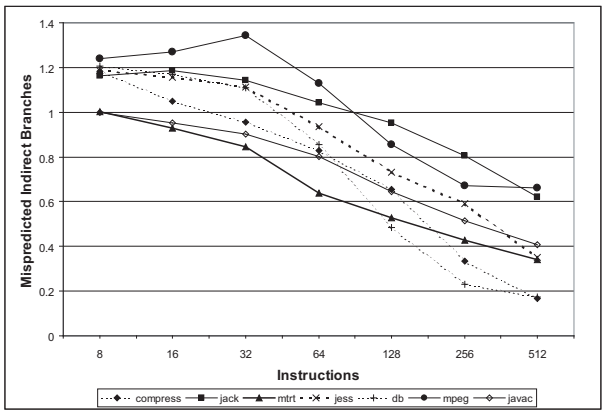


**Figure 18: Reduction in indirect branch mispredictions from replicated instructions chosen using dynamic frequency in other programs.**

tion. The main difference between the sEc work and ours, is that sEc was never implemented. Thus, the implementation details were never worked out, and no real performance results were presented. In contrast, we have a fully-working system, which we have evaluated using large, standard benchmarks.

Combining operations using an interpreter generator system was previously explored in the context of superoperators [13]. A superoperator is pattern of more than one operator in a tree representation of an expression. Superoperators chosen for a particular program allowed speedups of about a factor of two in an interpreter using switch dispatch. Switch dispatch is so expensive that almost anything that reduces the number of dispatches is worthwhile.

`Vmgen` [6] provided the original inspiration for our interpreter generator. `Vmgen` implements superinstructions, as well as other optimisations we do not support such as stack-caching [4]. It does not, however, support instruction specialization or instruction replication. `Vmgen` is written in Forth; we chose to build our own generator from scratch in Java rather than reusing `vmgen` because of the language.

Ertl and Gregg [5] evaluated superinstructions and replication in a Forth interpreter. Although replication did reasonably well, their speedups for superinstructions were very low at less than 10%. This is partly the result of basic blocks in Forth being extremely short (perhaps 2-3 VM instructions on average), and partly because they used a very poor superinstruction selection strategy (dynamic sequences from only a single program). In contrast, we have shown that superinstructions can be extremely effective, but that the effectiveness depends on the selection strategy.

# 7. CONCLUSION

We have described a system of enhanced VM instructions for a portable, efficient Java interpreter. Our interpreter generator automatically creates source code for specialized instructions, superinstructions and replicated instructions from simple instruction definitions. Our system deals with several difficult issues, such as allowing specialized versions of "quick" instructions, superinstructions containing "quick" instructions, and superinstructions that extend across basic blocks.

We have evaluated these VM instruction enhancement techniques using many different strategies for applying them. We found that although instruction replication achieves its goal of reducing operand fetch, its impact on indirect branch prediction has a much greater impact on performance. We have shown how our `tiger` generator can be used to create an optimized version of our interpreter which is customized with superinstructions for a particular program, giving speedups of 1.35 to 3.35. We have also experimented with a large number of strategies for selecting useful superinstructions form a group of representative programs. Perhaps counter-intuitively, we found that it is better to look at the static occurrence of sequences of instructions rather than their dynamic execution frequencies, and that we should favor shorter sequences. Speedups of 1.2 to 2.1 are possible using such generic superinstructions. Finally, we found that instruction replication does not always lead to speedups, and

that the effect on branch mispredictions is only positive for large numbers of replicas.

## 8. REFERENCES


[1] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[2] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, 1990.

[3] H. Eller. Threaded code and quick instructions for kaffe. http://www.complang.tuwien.ac.at/java/kaffe-threaded/.

[4] M. A. Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[5] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 03)*, pages 278–288, San Diego, California, June 2003. ACM.

[6] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — A generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[7] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, Mc Gill University, December 2002.

[8] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *First USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001.

[9] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Proceedings of the 12th International Conference on Compiler Construction*, LNCS 2622, pages 170–184, April 2003.

[10] D. Gregg and J. Waldron. Primitive sequences in general purpose forth programs. In *18th Euroforth Conference*, pages 24–32, Vienna, Austria, September 2002.

[11] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, Sept. 1999.

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

[13] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[14] SPEC. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.specbench.org/osg/jvm98/press.html.

[15] K. S. Venugopal, G. Manjunath, and V. Krishnan. sEc: A portable interpreter optimizing technique for embedded java virtual machine. In *Second USENIX Java Virtual Machine Research and Technology Symposium*, San Francsico, California, August 2002.

[16] J. Waldron. Dynamic bytecode usage by object oriented java programs. In *Proceedings of the Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition*, Nancy, France, June 7-10 1999.