Towards Superinstructions for Java Interpreters

Kevin Casey[1], David Gregg[1], M. Anton Ertl[2], and Andrew Nisbet[1]

[1] Department of Computer Science, Trinity College, Dublin 2, Ireland
`{Kevin.Casey,David.Gregg,Andy.Nisbet}@cs.tcd.ie`
[2] Institut für Computersprachen, TU Wien, A-1040 Wien, Austria
`anton@complang.tuwien.ac.at`

Abstract. The Java Virtual Machine (JVM) is usually implemented by an interpreter or just-in-time (JIT) compiler. JITs provide the best performance, but interpreters have a number of advantages that make them attractive, especially for embedded systems. These advantages include simplicity, portability and lower memory requirements. Instruction dispatch is responsible for most of the running time of efficient interpreters, especially on pipelined processors. Superinstructions are an important optimisation to reduce the number of instruction dispatches. A superinstruction is a new Java instruction which performs the work of a common sequence of instructions. In this paper we describe work in progress on the design and implementation of a system of superinstructions for an efficient Java interpreter for connected devices and embedded systems. We describe our basic interpreter, the interpreter generator we use to automatically create optimised source code for superinstructions, and discuss Java specific issues relating to superinstructions. Our initial experimental results show that superinstructions can give large speedups on the SPECjvm98 benchmark suite.

1 Motivation

The Java Virtual Machine (JVM) is usually implemented by an interpreter or just-in-time (JIT) compiler. JITs provide the best performance, but interpreters have a number of advantages that make them attractive, especially for embedded systems. First, interpreters require much less memory than JITs, both for the interpreter itself and the Java bytecode. For example, Hoogerbrugge et al. [13] found that a bytecode representation of a program could be up to five times smaller than the corresponding machine code. Many embedded systems have small memories giving interpreters a decisive advantage.

A second important advantage of interpreters is that they can be constructed to be trivially portable to new architectures, assuming that a C compiler for the new architecture already exists. In contrast, it can take many months to port the back end of a JIT compiler. Portability means that the Java interpreter can be rapidly moved to a new architecture, reducing time to market. There are also significant advantages in different target versions of the interpreter being compiled from the same source code. The various ports are likely to be more reliable, since the same piece of source code is being run and tested on many different architectures. A single version of the source code is also significantly cheaper to maintain. There are other parts of the JVM that are more difficult to port (such as the Java Native Interface for calling machine code functions), but many embedded JVMs, such as Sun's KVM [19] for mobile devices, have limited support for these unportable features.

A third advantage of interpreters is that they are significantly smaller and simpler than JIT compilers. Simplicity makes them more reliable, quicker to construct and easier to maintain. When building a JIT compiler one must not only debug the code for the compiler, but must often also debug the code *generated* by the compiler. This is not an issue for interpreters. A final smaller advantage of interpreters is that they do not necessarily have to compile the bytecode into another format before execution. Sun's Hotspot mixed mode compiler/interpreter JVM takes advantage of this by only compiling code that has been shown to be frequently executed. The compilation overhead for rarely used code is often greater than the time needed to execute that code on an interpreter. A similar strategy is used by Transmeta for their Crusoe processor which emulates the x86 instruction set through a combination of interpreting and binary translation.

A weakness of using interpreters is that they run most code much slower than JITs. Even very efficient interpreters are typically about ten times slower than a JIT compiler [13]. The goal of our work is to narrow that gap, by applying speed optimisations to Java interpreters. One such optimisation is the use of superinstructions. Certain sequences of VM instructions (such as `ALOAD_0 GETFIELD`) occur frequently in Java bytecode. A superinstruction is a new instruction that behaves in the same way as a sequence of simple Java instructions. By replacing such sequences with the corresponding superinstruction, the work of several instructions can be performed, but with the interpreter overhead of only a single VM instruction.

Superinstructions have been used for many years to optimise interpreters. Traditionally, the addition of superinstructions to an interpreter made it much less maintainable, because they increased the size of the source code. We use an interpreter generator to automatically generate source code for superinstructions, based on a specification of the component instructions. Our generator system automatically optimises the source code for superinstructions to avoid unnecessary loads and stores by keeping intermediate values in registers, and by combining stack pointer updates.

This paper describes the design and implementation of a system of superinstructions for an optimised Java interpreter. Preliminary experimental results show that superinstructions can greatly increase the speed of a portable Java interpreter, allowing it to significantly outperform commercial Java interpreters hand-coded in assembly language.

2 Superinstructions

A superinstruction is a new virtual machine instruction that consists of a sequence of several existing VM instructions. There are several advantages in combining instructions in this way. First, it reduces the number of instruction dispatches required to perform a certain sequence of instructions. This is important since instruction dispatch is usually the most time consuming part of executing and instruction[1]. Secondly, it allows us to optimise the interpreter source code. For example, our interpreter generator automatically reuses values across VM instructions without reloading them, eliminates cancelling stack pointer updates, and performs other small stack optimisations when generating C code from the instruction definition. Thirdly, combining the source code for instructions together exposes a larger "window" of code to the C compiler, which allows greater opportunities for optimisation.

We use the interpreter generator `vmgen` [7] to allow us to generate superinstructions using profiling information. `vmgen` 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.

Figure 1 shows the instruction definition for the JVM instruction `ILOAD` (load integer local variable). The # symbol in the definition means that it takes an immediate value from the VM instruction stream. Note that we need to update the instruction pointer by two positions, since the VM instruction consists of the `ILOAD` opcode followed by an immediate operand containing the number of the local variable to load onto the stack.

```
ILOAD ( #iIndex -- iResult ) 0x21
{
  iResult = locals[iIndex];
}
```

Fig. 1. Definition of `ILOAD` VM instruction

By adding `ILOAD-IADD` to the list of superinstructions for our code copying compiler, `vmgen` will produce the source code in figure 2, which is generated automatically from the instruction definitions of `ILOAD` and `IADD`.

There are a number of notable features about this code. First, all used stack items are loaded from memory into local variables at the start of the code. The different VM instructions within the superinstruction communicate by reading from and assigning to these local variables.

Presuming that the C compiler is able to allocate these local variables to registers, this will greatly reduce the amount of memory traffic from accessing the VM stack. `IADD` alone requires two loads and one store to access the stack, and

[1] Instruction dispatch is expensive on modern architectures because it involves a difficult-to-predict indirect branch. In the case of threaded code interpreters, superinstructions not only reduce the number of dispatches, but also make the remaining branches more easily predictable using a branch target buffer (BTB) [6].

ILOAD requires one store. In contrast, the superinstruction ILOAD-IADD requires only one load and one store access to the stack to perform the same work. Thus stack memory traffic is reduced by 50%.

```
START_ILOAD_IADD: /* start label */
{
int sp0;  /* synthetic names */
int sp1;
int ip1;  /* synthetic name for item in VM instruction stream */
ip1 = *(ip+1); /* fetch immediate value */
sp0 = *(sp);
{ /* ILOAD */
  int iIndex; /* declare stack item */
  int iResult;
  /* fetch stack item to local variable */
  iIndex = ip1;
  {               /* user provided C code */
    iResult = locals[iIndex];
  }
  sp1 = iResult;  /* store stack result */
}
{ /* IADD */
  int iValue1;    /* declare stack items */
  int iValue2;
  int iResult;
  iValue1 = sp1;  /* fetch stack items to */
  iValue2 = sp0;  /* ...local variables */
  {               /* user provided C code */
    iResult = iValue1 + iValue2;
  }
  sp0 = iResult;  /* store stack result */
}
*(sp) = sp0;
ip += 3;          /* update VM ip */
}
NEXT;             /* indirect goto */
```

Fig. 2. Simplified Vmgen output for ILOAD-IADD superinstruction

Another notable feature of the code in figure 2 is that there is no stack pointer update. ILOAD increases the size of the stack by one, and IADD reduces its size by one. Vmgen detects that the two stack pointer updates are redundant, and eliminates them. In addition, there is only one instruction pointer update.

3 Design Issues

3.1 Which Sequences?

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 [12] 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. We use this strategy for the main experiments in this paper.

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 [18] are in library rather than program methods [21]. 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.

3.2 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.

For example, consider the basic block in figure 3. Assume that we have two superinstructions available: `ILOAD-ILOAD` and `ILOAD-IADD-ISTORE`. Following a greedy strategy, we would find the longest sequence that matches a superinstruction from the start of the basic block. Thus, we would replace the first two instructions with the superinstruction `ILOAD-ILOAD`, and reduce the number of dispatches needed to execute this code by one. The main advantage of greedy parsing is that it is very fast — an important factor in an optimisation that we apply to a Java method at run time, the first time that it is invoked. Greedy parsing is also simple to implement and requires little memory.

```
ILOAD 4   ; load local 4
ILOAD 5   ; load local 5
IADD      ; integer add
ISTORE 6  ; store TOS to local 6
ILOAD 6   ; load local 6
IFEQ 7    ; branch by 7 if TOS == 0
```

Fig. 3. Example basic block

The weakness of greedy parsing becomes apparent when we consider whether a better parse of the code in figure 3 is possible. Clearly, it would be better to replace the second, third and fourth instructions with the superinstruction `ILOAD-IADD-ISTORE`. This would reduce the number of dispatches by two. To be sure of always finding 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. However, our preliminary experiments show that even fast implementations are measurably slower than greedy parsing. Furthermore, these preliminary experiments show optimal parsing reducing the number of instruction dispatches by less that 5%.

Our current implementation uses a simple version of greedy parsing. In the bytecode translator, we always keep a buffer of the most recently generated threaded code instruction in the basic block. When we generate the next instruction, we check whether it can be combined with the one in the buffer. If it can, then the instruction in the buffer is replaced with the corresponding combined superinstruction. If not, the instruction in the buffer is written to the the code area for that method, and it is replaced in the buffer by the just generated instruction. This strategy is simple to implement, requires little memory, and makes the check for replacement with superinstructions extremely fast.

One weakness of this strategy, however, is that for a long superinstruction to be usable, all prefixes of the instruction must also be valid superinstructions. For example, if we have the superinstruction `ILOAD-ILOAD-IADD-ISTORE`, then we must also have the superinstructions `ILOAD-ILOAD` and `ILOAD-ILOAD-IADD`. In practice, this is not a problem, since we usually select superinstructions based on the frequency of sequences in real programs, and by definition subsequences have a frequency at least equal to that of the longer sequence. However, in future implementations we intend to relax this restriction to allow us to exploit more complicated superinstruction selection strategies.

3.3 Quick Instructions

Several Java bytecode instructions must perform various class initialisations on 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 [21]. 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 simultanuously 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 does not allow any "quickable" instructions to participate in superinstructions. However, we are experimenting with a wide range of strategies to change this. Perhaps our most promising is to simply add an extra routine to the quickening process to reparse the basic block once the original instruction has been replaced. This approach is greatly simplified by leaving gaps for removed instructions in the code, as is outlined in the next subsection.

3.4 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.

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 4 shows an example of, where we

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

Fig. 4. Original code (left) and same code with `ILOAD-IADD` superinstruction (right)

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.

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

Fig. 5. Definition of a branch VM instruction

A second opportunity for cross-basic block superinstructions is with the fall-through direction of VM conditional branches. Currently, superinstructions are not allowed to extend across branches. However, `vmgen` already provides a facility for specifying a taken branch. Figure 5 shows the instruction definition for a branch instruction. Inside the `if` statement the `vmgen` 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. We believe that with some modifications to `vmgen`, the same facility can be used to create superinstructions that extend across untaken branches, with the necessary code for the taken path generated using the `TAIL` mechanism.

4 Experimental Evaluation

The primary purpose of the work presented here was to evaluate the effect of adding superinstructions to the JVM. By adding superinstructions, we reduced the number of stack updates and also eliminated branch target mispredictions for instructions within the superinstruction. As a result, we expected to see significant improvements as more and more superinstructions were added to our JVM (subject to some limitations). It was also strongly suspected that the method used to select which superinstructions to add would have a substantial effect on the superinstructed JVM. The benchmarks selected for evaluating the effect of changes to the JVM were taken from the SPECjvm98 suite.

In order to obtain a JVM with support for superinstructions it was necessary to modify Sun Microsystem's CVM for embedded processors. Apart from converting the JVM to work with dynamically threaded code, the bulk of the work was in porting the main interpreter loop to Vmgen in a satisfactory manner to allow for superinstructions. Once the interpreter loop had been ported to `vmgen`, the selection of candidate superinstructions and the actual inclusion of superinstructions in the JVM became a relatively straightforward process due to the nature of Vmgen.

To select superinstructions to add to the JVM, two contrasting approaches were taken. In the first approach, all benchmarks were run and all sequences of bytecode (and their subsequences) encountered for the first time were recorded. When all benchmarks were completed, a histogram of these sequences was built up. From this histogram the most common statically appearing sequences of bytecodes were selected.

The second approach was more aggressive from an optimization point of view. In this approach we ran each benchmark separately and for each benchmark recorded all sequences of bytecodes encountered during the execution of the benchmark (i.e. not just the first time they are encountered). Thus the same sequence of superinstructions could be recorded several times, for example if they occurred within the body of a loop. Then, for each individual benchmark a histogram of the most commonly encountered superinstructions was generated. Then, to optimize for a particular benchmark, the histogram for that particular benchmark was used to select the most commonly executed (dynamically appearing) sequences.

Generating superinstructions based on static frequency may appear to be over-simplistic, but as an initial method of selecting superinstructions, it does seem more realistic than the dynamic approach. One of the main reasons is that for the static approach we attempted to optimize the JVM for all benchmarks at once. With the dynamic approach, the JVM was optimized separately for each benchmark before running that benchmark. Despite the artificial nature of the dynamic approach, it does give us a standard by which to measure the performance of the static approach.

When selecting superinstructions from the histogram in either approach, some superinstructions are not permitted. For example superinstructions containing "quickable" (see section 3.3) instructions are dispensed with, as there is currently no facility for dealing with them in our modified JVM.

In our modified JVM, translation takes place when a method has been called for the first time, but before any bytecodes in that method have been executed. At this point in time only immutable opcodes can be included in superinstructions, since superinstructions themselves are immutable. One possible workaround would be to try to quicken all instructions in the method and then try to translate the code to superinstructed code. However this approach would inevitably lead to the quickening of code that may never be run, and also may force static initializers to be run before they are supposed to be. Another approach would be to allow superinstructions to be added dynamically as instructions get quickened in the usual way.

The modified version of CVM used for these tests was compiled under GCC 2.96. Optimization flags "-O4 fomit-frame-pointer" were used. The "-fno-gcse" flag was used additionally to compile the file containing the main interpreter loop. This is used to disable global common subexpression elimination, which can interact badly with GNU C labels as values, which are used by our interpreter for efficient instruction dispatch [5]. The hardware used to run the benchmarks was based on a Pentium IV 1.6 Ghz with 1GB of memory.

Each benchmark was run with no superinstructions to establish a reference time. Then the benchmarks were run on versions of CVM compiled with 8, 16, 32, 64, 128, 256, 512 and 1024 superinstructions. The results were then graphed as a speedup over the time it took each benchmark to complete with no superinstructions. All SPEC benchmarks were run using the largest (size 100) input sets.

The static results are shown in figure 6. All results are averages of 5 runs of the benchmark under the same conditions. In this figure we see a small improvement for most benchmarks, even at 8 superinstructions. Two benchmarks with 8 superinstructions perform worse, however. One possible reason for the lack of improvement in *compress* and *mtrt* is that the 8 superinstructions added simply do not occur frequently, if at all, in these benchmarks. One explanation for the reduction in performance (albeit less than 1%) could be the overhead of scanning through code at translation time to see if superinstructions can be formed. Other possible reasons are discussed below.

As superinstructions are added, there is a general trend upwards in performance which is what we would expect. Benchmarks *mpegaudio*, *compress* and to a lesser degree *db*, all spend much of their time in a small number of methods [21]. It seems most likely that some superinstructions are being introduced into these commonly used methods, giving the significant performance boost. The benchmark that gets greatest benefit from superinstructions is *mpegaudio*, with a maximum speedup of about 1.56. It is interesting to note that this benefit is not substantial until 256 superinstructions are introduced.

It is not always the case that the addition of extra superinstructions improves performance. A temporary drop-off in performance can be seen in all benchmarks at some stage, the most spectacular being when moving from 32 superinstructions to 64 superinstructions in both jack and jess. There are a number of possible explanations for these drop-offs. One possibility is that the register allocation mechanism in gcc is breaking down for superinstructions added at these points. Another is that superinstructions are causing conflict misses in the instruction cache or branch predictor. Finally, the process of scanning through a method to find possible superinstructions is slowed by the addition of extra superinstructions to the JVM.

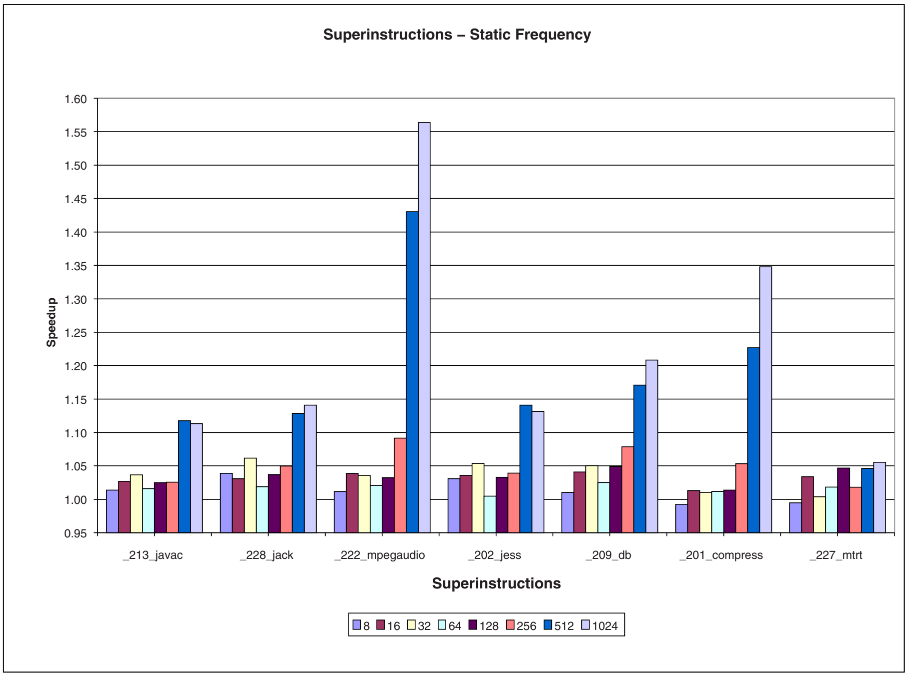

Fig. 6. Running times of the benchmarks with varying numbers of superinstructions. Superinstructions are chosen on the basis of **static** frequency of sequences across all SPECjvm98 programs

In figure 7 the performance of CVM with superinstructions based on dynamic frequency for this particular program can be seen. Performance is much better, but this is expected since CVM is optimized for each benchmark separately. This time the maximum speedup is 1.90 (*mpegaudio*). As before, the benchmarks that register the greatest improvements are those that spend much of their execution time in a limited set of methods. It can be surmised that a substantial number of superinstructions are being created in these methods.

At certain stages, the JVMs based on dynamically selected superinstructions suffer from the same drop-off in performance seen in figure 6. This time *javac*

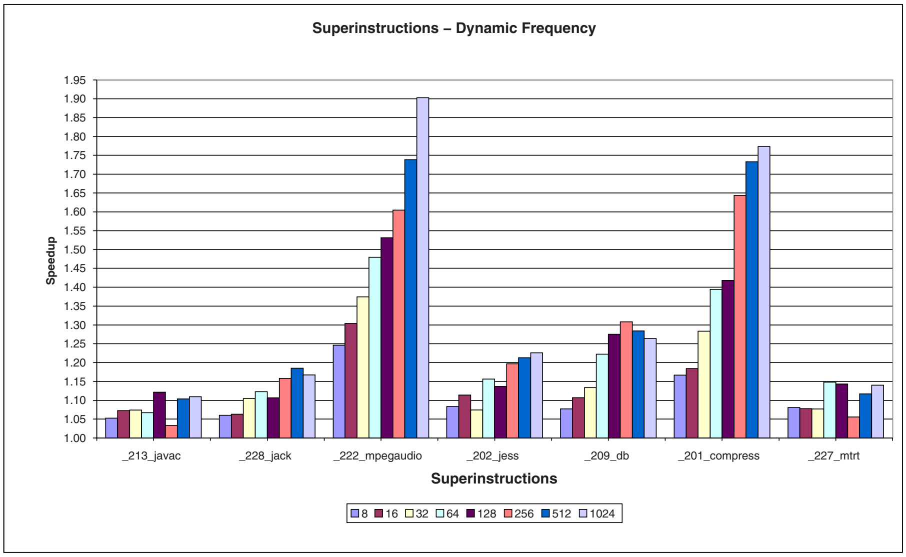

Fig. 7. Running times of the benchmarks with varying numbers of superinstructions. Superinstructions chosen are the most frequent dynamically executed sequences based on a training run of the same program

and *mtrt* both suffer a degradation in performance when moving from the 128 superinstruction JVM to a 256 superinstruction JVM.

Table 1 shows the absolute running times of the SPECjvm98 benchmarks on three different JVMs. The first is our base interpreter with no superinstructions. We also show running times for Sun's HotSpot mixed-mode interpreter and JIT compiler, and for HotSpot using only the interpreter. Overall, the Hotspot interpreter is on average 20.4% faster than the our interpreter.

Table 1. Comparison of running time of our base interpreter (without superinstructions) with the Sun HotSpot Client VM Interpreter, and mixed mode interpreter—JIT compiler on the SPECjvm98 benchmark programs

Benchmark	Our Base Interp.	Hotspot Interp.	Hotspot Mixed-mode
javac	55.79	44.38	10.16
jack	33.48	27.68	5.19
mpeg	150.08	139.65	9.55
jess	48.14	34.38	4.35
db	116.63	86.27	26.6
compress	170.01	153.19	18.9
mtrt	52.41	43.56	6.06

There are two main reasons for this. Firstly, Hotspot has a much faster run time system than CVM. This can be seen especially strongly in the *db* benchmark, which runs 34% faster on Hotspot. The Hotspot run time system is large and sophisticated, and would not be suitable for an embedded system. Furthermore, much effort has been put into tuning the Hotspot run time system as it is more widely used than CVM. The second reason that Hotspot outperforms our version of CVM is that 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. The difference in speeds of the interpreter cores can be seen by examining the benchmarks that spend most of their time in the interpreter core: *compress* is 9.1% faster and *mpeg* is 5.2% faster on the Hotspot interpreter. Finally, the mixed-mode compiler- interpreter is very much faster than either our interpreter or the Hotspot interpreter. Where speed is more important than memory use, portability, and maintainability, a JIT compiler is the correct solution.

5 Related Work

Some recent important developments in interpreters include the following. Stack caching [4] is a general technique for storing the topmost elements of the stack in registers. Ertl and Gregg [5] showed that interpreters (especially those using switch dispatch) spend most of their time in branch mispredictions on modern desktop architectures. Interpreter software pipelining [13] is a valuable technique for architectures with delayed branches (e.g. Philips Trimedia) or prepare to branch instructions (e.g. PowerPC), which makes the target of the dispatch branch available earlier by moving much of the dispatch code into the previous VM instruction. Costa [17] discusses various smaller optimizations.

The Sable VM [9] is an interpreter-based research JVM. This interpreter uses a run-time code generation system [15], not dissimilar from a just-in-time compiler. Sable uses a novel system of *preparation sequences* [10,8] to deal with bytecode instructions that perform initialisations the first time they are executed, which make code generation difficult. We believe that the same procedure could also be used to allow such instructions be part of superinstructions.

Venugopal et al. [20] present an embedded JVM system, which uses *semantically enriched code* (sEc). The sEc technique generates a custom JVM for each application. In addition, aggressive optimizations are applied to the program to allow it to make the best use of the custom JVM features. This tight coupling of the program and the interpreter allows large speedups. The weaknesses of this approach are that the code to be run must be available at the time the JVM is created, and that the JVM is no longer general purpose.

Combining operations using an interpreter generator system was previously explored in the context of superoperators [16]. 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.

Gregg et al. [11] and Ertl et al. [7] presented a prototype interpreter based on the Cacao research JVM [14]. This interpreter was built using Vmgen and used the facility for generating superinstructions. With large numbers of superinstructions, reductions in running time of the order of one third were possible. Unfortunately, the system was rather unstable and could run only a handful of programs. It also did not support a number of language features such as multithreading and correct initialisation of classes. In contrast, the interpreter described in this paper is a full, stable version that fully supports the standard and runs all programs that we have tried.

6 Conclusion

We have described a system of superinstructions for a portable, efficient Java interpreter. Our interpreter generator automatically creates source code for superinstructions from instruction definitions. Stack access code is optimised to reuse the topmost stack items between the component instructions in a superinstruction. This can significantly reduce stack traffic. Furthermore, our interpreter generator optimises stack pointer updates by combining and possibly eliminating them across component instructions. Our interpreter generator also provides a profiling system to identify common sequences of instructions. Experimental results show that significant speedups of up to 90% are possible with large numbers of appropriate superinstructions, due to reduction in dispatches and optimised superinstruction code.

Although our superinstruction system is stable and gives speedups in most configurations, considerable work remains for the future. The most important future development will be a scheme to allow "quickable" instructions to participate in superinstructions. Many of the most frequently executed Java instructions such as field access (16.4% of executed instructions in SPECjvm98 [21]) and method invokes (5.7%) are "quickable". We believe that allowing these instructions to participate in superinstructions will greatly increase the running speed of our interpreter. We also plan work in the area of better heuristics for choosing superinstruction, better parsing algorithms, and superinstructions across basic block boundaries.

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. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

6. 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 2003)*, San Diego, California, June 2003. ACM. to appear.
7. 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.
8. E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode.* PhD thesis, Mc Gill University, December 2002.
9. 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.
10. 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.
11. D. Gregg, A. Ertl, and A. Krall. Implementation of an efficient Java interpreter. In *Proceedings of the 9th High Performance Computing and Networking Conference*, LNCS 2110, pages 613–620, Amsterdam, The Netherlands, June 2001.
12. D. Gregg and J. Waldron. Primitive sequences in general purpose forth programs. In *18th Euroforth Conference*, pages 24–32, Vienna, Austria, September 2002.
13. 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.
14. A. Krall and R. Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. In G. C. Fox and W. Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.
15. 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.
16. T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL'95)*, pages 322–332, 1995.
17. V. Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.
18. 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`.
19. Sun Microsystems Inc. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000.
20. 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.
21. 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.