Implementing an Efficient Java Interpreter

David Gregg[1], M. Anton Ertl[2] and Andreas Krall[2]

[1] Department of Computer Science,
Trinity College, Dublin 2, Ireland.
David.Gregg@cs.tcd.ie
[2] Institut für Computersprachen, TU Wien,
Argentinierstr. 8, A-1040 Wien,
{anton|andi}@complang.tuwien.ac.at

Abstract. The Java virtual machine (JVM) is usually implemented with an interpreter or just-in-time (JIT) compiler. JIT compilers provide the best performance, but must be substantially rewritten for each architecture they are ported to. Interpreters are easier to develop and maintain, and can be ported to new architectures with almost no changes. The weakness of interpreters is that they are much slower than JIT compilers. This paper describes work in progress on a highly efficient Java interpreter. We describe the main features that make our interpreter efficient. Our initial experimental results show that an interpreter-based JVM may be only 1.9 times slower than a compiler-based JVM for some important applications.

1 Introduction

The Java Virtual Machine (JVM) is usually implemented by an interpreter or a just-in-time (JIT) compiler. JIT compilers provide the best performance but much of the compiler must be rewritten for each new architecture it is ported to. Interpreters, on the other hand, have huge software engineering advantages. They are considerably smaller and simpler than JIT compilers, which makes them faster to develop, cheaper to maintain, and potentially more reliable. Most importantly, interpreters are portable, and can be recompiled for any architecture with almost no changes.

The problem with existing interpreters is that they run most code much slower than compilers. The goal of our work is to narrow that gap, by creating a highly efficient Java interpreter. If interpreters can be made much faster, they will become suitable for a wide range of applications that currently need a JIT compiler. It would allow those introducing a new architecture to provide reasonable Java performance from day one, rather than spending several months building a compiler.

This paper describes work in progress on a fast Java interpreter. In section 2 we introduce the main techniques for implementing interpreters. Section 3 describes the structure of our interpreter, and our main optimizations. Section 4 presents some preliminary experimental results, comparing our interpreter with other JVMs. Finally, in section 5 we draw conclusions.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
High-Performance Computing and Networking (HPCN Europe 2001), Springer LNCS 2110, pp. 613--620

2 Virtual Machine Interpreters

The interpretation of a virtual machine instruction consists of accessing arguments of the instruction, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction. The most efficient method for dispatching the next VM instruction is direct threading [Bel73]. Instructions are represented by the addresses of the routine that implements them, and instruction dispatch consists of fetching that address and branching to the routine (see fig. 1). Direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels, but GNU C provides the necessary features. Implementors who restrict themselves to ANSI C usually use the giant switch approach (see fig. 2): VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine.

```
void engine() {
  static Inst program[] = { &&add /* ... */ };
  Inst *ip; int *sp;

  goto *ip++;
 add:
  sp[1]=sp[0]+sp[1];  sp++;  goto *ip++;
}
```

Fig. 1. Direct threading using GNU C's "labels as values"

```
void engine() {
  static Inst program[] = { add /* ... */ };
  Inst *ip; int *sp;

  for (;;)
    switch (*ip++) {
    case add:
      sp[1]=sp[0]+sp[1];  sp++;  break;
    /* ... */
    }
}
```

Fig. 2. Instruction dispatch using `switch`

When translated to machine language, direct threading typically needs three to four machine instructions to dispatch each VM instruction, whereas the switch method needs nine to ten [Ert95]. The execution time penalty of the switch method is caused by a range check, by a table lookup, and by the branch to the dispatch routine. In addition, indirect branches are more predictable in threaded code interpreters than those in switch based interpreters which leads to far fewer pipeline stalls and shorter running times on current architectures.

3 Implementing a Java Interpreter

We are currently building a fast threaded code interpreter for Java. Rather than starting from scratch, we are building the interpreter into an existing JVM. We started working with the CACAO [KG97,Kra98] JIT compiler based 64-bit JVM. Therefore, we used the infrastructure available from the JIT compiler and implemented a 64-bit interpreter. But it is our intention that it will be possible to plug our interpreter into any existing JVM. For this reason, we are defining a standard interface which describes a set of services that the wider JVM must provide for our interpreter.

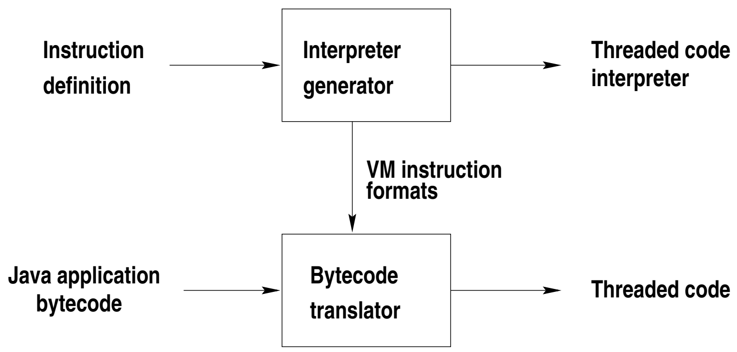

Fig. 3. Structure of the Interpreter System

Figure 3 shows the structure of our interpreter system. It is important to note that we don't interpret Java byte code directly. Instead, the byte code is translated to threaded code. In the process we also apply optimizations to make the threaded code easier to interpret. Also important is that we do not write all the code in the interpreter ourselves. Instead, we are building an interpreter generator, which constructs an efficient interpreter from a specification of the behavior of each instruction. The following subsections describe the major components in more detail.

3.1 The Byte Code Translator

The main goal of the translator is to remove complex and expensive operations from the interpreter, and instead perform these operations once at translation time[1]. The simplest, and most important example of this is the translation from byte code to threaded code. Like a JIT compiler translates byte code from a

[1] This is the RISC principle of dealing with complex and difficult operations in the compiler, to allow a simpler, faster processor implementation. A processor is a hardware implementation of an interpreter, and many principles of processor design can be applied to interpreters.

complete method to machine code, our translator transforms byte code into threaded code. To interpret a byte code the interpreter must look up the address of the routine that implements the VM instruction in a table. When translating to threaded code, we perform that lookup just once, and replace the byte code with the address. Thereafter, we can interpret the threaded code without any table lookups. Note that the threaded code is larger than the original byte code. An important research question is whether the benefits of these optimizations could be outweighed by an increase in data cache misses, due to threaded code that occupies more memory.

The byte code translator also replaces difficult to interpret instructions with simpler ones. For example, we replace instructions that reference the constant pool, such as `LDC`, with more specific instructions and immediate, in-line arguments. We follow a similar strategy with method field access and method invocation instructions. When a method is first loaded, a stub instruction is placed where its threaded code should be. The first time the method is invoked, this stub instruction is executed. The stub invokes the translator to translate the byte code to threaded code, and redirects itself to point to the first instruction of the threaded code.

In the process of translation, we rewrite the instruction stream to remove some inefficiencies and make other optimizations more effective. For example, the JVM defines several different load instructions based on the type of data to be loaded. In practice many of these, such as `ALOAD` and `LLOAD` can be mapped to the same threaded code instruction. These transformations reduce the number of VM instructions and makes it easier to find common patterns (for instruction combining). Another simple optimization is based on the fact that many instructions in the JVM take several immediate bytes as operands. These are shifted and OR-ed together to form a larger integer operand. We perform this computation once at translation time, and use larger integer immediates in the threaded code.

One implication of translating the original byte code is that the design problems we encounter are closer to those in a just-in-time compiler than a traditional interpreter. Translating also requires a small amount of overhead. Translating allows us to speed up and simplify our interpreter enormously, however. Original Java byte code is not easy to interpret efficiently.

3.2 Optimizations

The CACAO JIT compiler has an analysis to compute the necessary information to remove bound checks. We reused this analysis to implement bound check removal in the interpreter. We defined two sets of array access instructions, one with bound checks and one without bound checks. Depending on the results of the analysis the right instruction is chosen.

The CACAO compiler does null pointer checks using the hardware. This feature is hardware dependent and not suitable for interpreters. To evaluate the cost of software null pointer checks we generated a version of the interpreter without null pointer checks.

3.3 Instruction Definition

The *instruction definition* describes the behavior of each VM instruction. The definition for each instruction consists of a specification of the effect on the stack, followed by C code to implement the instruction. Figure 4 shows the definition of `IADD`. The instruction takes two operands from the stack (`iValue1,iValue2`), and places result (`iResult`) on the stack.

```
IADD ( iValue1 iValue2 -- iResult ) 0x60
{
  iResult = iValue1 + iValue2;
}
```

Fig. 4. Definition of `IADD` VM instruction

We have tried to implement the instruction definitions efficiently. For example, in the JVM operands are passed by pushing them onto the stack. These operands become the first local variables in the invoked method. Rather than copy the operands to a new local variable area, we keep local variables and stack in a single common stack, and simply update the frame pointer to point to the first parameter on the stack. To correctly update the stack and frame pointer on calls and returns using this scheme, one needs to compute several pieces of information about stack heights and numbers of local variables. We compute this information once at translation time, and thereafter the handling of parameters during interpretation is much more efficient.

One complication in our instruction specification is that it was originally designed for Forth. In this language, the number of stack items produced and/or consumed by a given VM instruction is always the same. Java has several VM instructions that consume a variable number of stack items, however. For example, the instruction to create a multidimensional array (`MULTIANEWARRAY`) takes a number of items from the stack equal to the number of parameters of the array. Similarly, the various method invocation instructions consume a number of stack items equal to the number of parameters. Currently, we have no way to cleanly express this in our instruction definition. One possibility is to create separate VM instructions for each possible number of stack items consumed. This does not scale however, since in theory there can be almost any number of parameters. Our current solution is to manipulate the stack pointer directly in the instruction specification.

3.4 Interpreter Generator

The *interpreter generator* is a program which takes in an instruction definition, and outputs an interpreter in C which implements the definition. The interpreter generator translates the stack specification into pushes and pops of the stack, and adds code to invoke following instructions.

There are a number of advantages of using an interpreter generator rather than writing all code by hand. The error-prone stack manipulation operations can be generated automatically. Optimizations can easily be applied to all instructions. For example, the fetching of the next instruction can be moved up the code (interpreter pipelining [HA00]). It is easy to have both a threaded code and switch-based version of the interpreter. Specifying the stack manipulation at a more abstract level also makes it simpler to change the implementation of the stack. For example, our interpreter keeps one stack item in a register. It is nice to be able to vary the number of cached stack items without changing each instruction specification. The generator also allows us to add tracing and profiling code trivially, and easily disassemble the threaded code.

The main advantage of the generator is that it allows more complicated optimizations such as automatic instruction combining or instruction specialization. Combining replaces a common sequence of VM instructions with a single "super" instruction [Pro95]. For common operands specialization uses different instructions to eliminate the operand decoding overhead. Combining greatly increases the number of VM instructions, making maintenance more difficult if done manually. The generator can create these combinations automatically. Using the stack specifications, it also optimizes stack operations for combined instructions.

4 Preliminary Experimental Results

Our basic thesis is that interpreter based JVMs can be so fast that at least for some applications they are not very much slower than a JIT compiler. To test the performance of a simple, efficient threaded-code interpreter implementation of the JVM we compared it with the CACAO JIT compiler, with an interpreter and two different JIT compilers from COMPAQ, with an interpreter from OSF and with the KAFFE interpreter on different processors. CACAO is one of the fastest available JVM implementations and for computationally intensive programs provides 42% to 82% of the performance of optimized C code [Kra98].

Our two main benchmarks are *javac* and *db* from the SPECjvm98 benchmark suite. The *javac* benchmark is the Java compiler from the JDK 1.0.2. The *db* benchmark performs a sequence of add, delete, find and sort operations on a memory resident database. We also tested two computationally intensive mini-benchmarks *sieve* and *suml*. The first of these is the well-known prime number computation program and *suml* is the single pathological loop {`long i, l1, l2; for (i = l1 = l2 = 0; i > 0; i--)` {`l1++; l2--;`}} that was designed to maximally stress an interpreter based JVM.

Table 1 shows the run time of our benchmarks relative to the CACAO JIT compiler on Alpha 21064a and Alpha 21164a based workstations. The results for the SPEC benchmarks are surprisingly good, with our interpreter taking not much more than twice the time of the CACAO just-in-time compiler and faster than some other JIT compilers. This result sharply contradicts the widespread belief that interpreter based JVMs are inherently slow and will always perform many times slower than JIT compilers. We examined the proportion of time

21064a	javac	db	sieve	suml
CACAO native	1.00	1.00	1.00	1.00
CACAO int	2.12	2.29	10.16	39.91
COMPAQ JVM 1.1.4 native	13.14	23.65	2.04	3.85
COMPAQ JVM 1.1.4 int	15.80	27.50	22.51	104.88
OSF JVM 1.0.1 int			29.06	156.42
21164a	javac	db	sieve	suml
CACAO native	1.00	1.00	1.00	1.00
CACAO int	2.27	2.10	17.54	25.08
COMPAQ JVM 1.3.1 native		5.01	2.52	3.23
KAFFE 1.0.5 int		19.57	63.47	93.21

Table 1. Relative performance of JIT compilers and interpreters

spent in native functions rather than executing byte codes. We found that both programs spend about 30% of their time in native functions (synchronization, garbage collection). Both the interpreter and JIT compiler have to spend the same amount of time in native functions. Therefore, even though the JIT compiler executes byte codes about six times as fast as the interpreter for these programs, the overall speedup is not much more than a factor of two.

The results for *sieve* are rather less encouraging, showing the interpreter to be ten to seventeen times slower than the JIT compiler. Not only does the JIT compiler avoid the expensive overhead of dispatching instructions, it is also able to optimize array accesses in the inner loop with pointer arithmetic. This suggests that interpreters are not at all suited to computationally intensive code. Finally, the *suml* benchmark shows the worst case performance of interpreter based JVMs.

	null	bound	null & bound
relative run time	0.995	0.976	0.971

Table 2. Effects of optimizations on sieve

Table 2 shows the effects of different optimizations on the *sieve* benchmark. Whereas the elimination of null pointer checks only gives a small speedup of 0.5%, the elimination of array bound checks gives a speedup of 2.5%. The benefit of these optimizations is quite small compared to speedups of more than 30% achievable by JIT compilers using these optimizations.

Figure 5 shows the potentials of instruction combination. The left graph gives the number of all possible used superinstructions for lengths of up to four instructions. As *javac* is a bigger program more different combinations exist. The right graph shows the speedups for *javac* and *db* on different processors and different length of superinstructions. For all programs *db* has been used as training input for the instruction combiner. The Alpha 21064a has a 16 Kbyte direct mapped instruction cache, the Alpha 21264 has a 64 Kbyte two way set associative instruction cache. For small caches exhaustive instruction combining can lead to cache trashing. For the Alpha 21264 the gap between interpreter and JIT compiler can be reduced to a factor of 2.07 for *javac* and 1.88 for *db*.

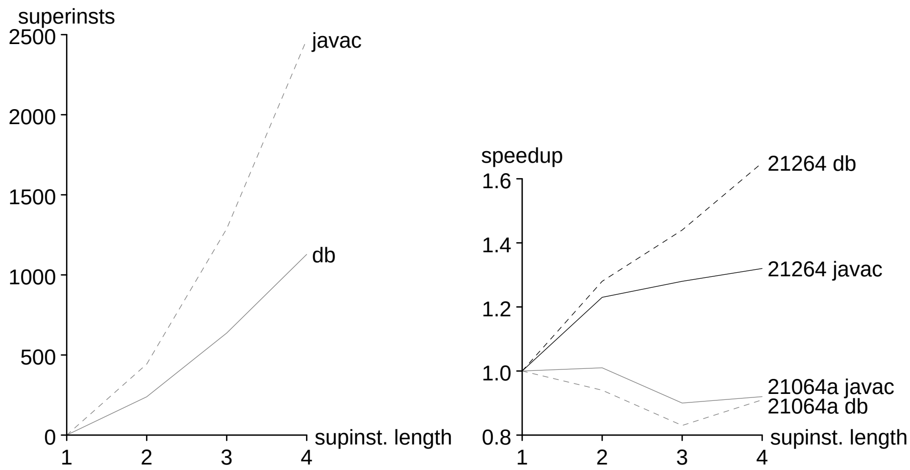

Fig. 5. Left: Number of superinstructions for different maximum sequence lengths. Right: Speedup for different programs and CPUs.

5 Conclusion

We have described an efficient interpreter based implementation of the Java virtual machine. Our interpreter system translates the original Java byte code to threaded code, greatly reducing the interpreter overhead. Our translator also computes constants, targets and offsets at translation time, allowing us to greatly simplify the interpretation of many instructions, such as method invocations. Experimental results show that our interpreter based JVM may be not much more than 1.9 as slow as a good JIT based JVM for some general purpose applications. For scientific code an interpreter cannot replace a compiler.

References

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

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java virtual machine interpreters. In *Proceedings of the 9th International Conference on Compiler Construction (CC' 00)*. Springer LNCS, 2000.

[KG97] Andreas Krall and Reinhard Grafl. CACAO - a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[Kra98] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference of Parallel Architectures and Compilation Techniques*, pages 205–212. IEEE Computer Society, October 1998.

[Pro95] Todd Proebsting. Optimising an ANSI C interpreter with superoperators. In *Proceedings of Principles of Programming Languages (POPL'95)*, pages 322–342, 1995.