A Fast 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@complang.tuwien.ac.at`

Abstract. The Java virtual machine (JVM) is usually implemented with an interpreter or just-in-time (JIT) compiler. JITs provide the best performance, but must be substantially rewritten for each architecture they are ported to. Interpreters are easier to develop and maintain, need less memory and can be ported to new architectures with almost no changes. The weakness of interpreters is that they are much slower than JITs. This paper describes work in progress on faster Java interpreters. Our goal is to bring interpreter performance to a new higher level by developing new optimisations for modern computer architectures, and by adapting recent research on compilers to interpreters.

1 Introduction

The Java virtual machine (JVM) is usually implemented by an interpreter or a just-in-time (JIT) compiler. JITs provide the best performance but much of the JIT must be rewritten for each new architecture it is ported to. Interpreters, on the other hand, have huge software engineering advantages. They are smaller and simpler than JITs, 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 JITs. 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. It would allow those introducing a new architecture to provide reasonable Java performance from day one, rather than spending several months building or modifying a JIT.

This paper describes work in progress on building faster Java interpreters. Why do we think that interpreter performance can be improved? One reason is that interpreters are rarely designed to run well on modern architectures which depend on pipelining, branch prediction and caches. We also believe that many compiler optimisations can be applied to interpreters. Finally, there seems to be a widespread attitude that interpreters are inherantly slow, so there is little

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
JOSES (Java Optimization Strategies for Embedded Systems) Workshop at ETAPS 2001

point in worrying about performance. The goal of our work is to show that this is not true.

This paper is organised as follows. In section 2 we introduce the main techniques for implementing interpreters. Section 3 describes our work in progress on a fast Java interpreter. Section 4 presents our plans for future optimisations. Finally, in section 5 we draw conclusions.

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. Unfortunately, direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels.

Fortunately, there is a widely-available language with first-class labels: GNU C (version 2.x); so direct threading can be implemented portably (see 1). If portability to machines without `gcc` is a concern, it is easy to switch between direct threading and ANSI C conforming methods by using macros and conditional compilation.

```
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"

Implementors who restrict themselves to ANSI C usually use the giant switch approach (2): VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine; in this method the whole interpreter, including the implementations of all instructions, must be in one function.

Figures 3 and 4 show MIPS assembly code for the two techniques. The execution time penalty of the switch method over direct threading is caused by a range check, by a table lookup, and by the branch to the dispatch routine generated by most compilers.

```
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`

```
lw   $2,0($4) #get next inst., $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Fig. 3. Direct threading in MIPS assembly

```
$L2: #for (;;)
 lw    $3,0($6) #$6=instruction pointer
 #nop
 sltu  $2,$8,$3 #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4  #branch delay slot
 sll   $2,$3,2  #multiply by 4
 addu  $2,$2,$7 #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...
$L13: #switch target table
 .word   $L12
 ...
$L12: #add:
 ...
 j   $L2
 #nop
```

Fig. 4. Switch dispatch in assembly

3 A Fast Java Interpreter

We are currently building a fast threaded interpreter for Java. Rather than starting from scratch, we are building the interpreter into an existing JVM. We are currently working with the CACAO [Kra97] JIT-based JVM, but our intention is that it will be possible to plug our interpreter into any existing JVM. It is important to note that we don't interpret Java bytecodes directly. We first translate them to threaded code. In the process, we optimise the code by replacing instructions with complicated run-time behaviour to simpler instructions that can be interpreted more easily. The goal is to put the burden of dealing with complexity in the translator, rather than the inner loop of the interpreter.

The interpreter system consists of three main parts. 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 5 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. 5. 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 first parameter on the stack. This complicates the translation of calls and returns, but speeds up parameter handling.

The second part of our interpreter system is the *interpreter generator*. This 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 pop 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 code can be generated automatically. Optimisations easily be applied to all instructions. We can automatically add tracing and profiling to our interpreter. The interpreter generator can also produce a dissassembler, and some of the routines needed by the translator. Specifying the stack manipulation at a more abstract level also makes it easier to change the implementation of the stack. For example, many interpreters keep one or more stack items in registers. It is nice to be able to vary this without changing each instruction specification.

The third part, the *translator* translates the Java byte-code to threaded instructions. In the process of this translation, we rewrite the byte-codes to remove some ineficiencies and make other optimisations 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 `ILOAD` can be mapped to the same threaded instruction. This reduces the number of VM instructions and makes it easier to find common patterns (for instruction combining).

The 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 replaces itself with the first instruction of the threaded code. An alternative way to deal with these difficulties is to use *quick instructions*. These are individual VM instructions which replace themselves the first time they are invoked. Unfortunately, they make instruction combining difficult.

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. Currently, our interpreter runs a small number of Java programs. Once it is fully stable, we will start work on the optimisations described in the next section.

4 Future Directions

Once our basic interpreter is working we will develop new optimsations to make it faster. We will start with those techniques which we believe will give us the most speedup for the least development effort. As out interpreter becomes more efficient, and we become more familiar with the trade-offs we will apply more complicated techniques, whose payoff may be less.

4.1 Reducing Branch Mispredictions

Modern processors depend on correctly predicting the target of branches. In recent work [EG01] we show that efficient VM interpreters contain many difficult-to-predict indirect branches, and that up to 62%–79% of the running time of many interpreters is spent on branch mispredictions. The indirect branches in switch-based interpreters are correctly predicted only about 10% of the time using branch target buffers (the best prediction technique used in current processors). Unless they contain many other inefficiencies, the run time of switch-based interpreters is dominated by indirect branch mispredictions.

The branches in threaded interpreters are correctly predicted about 40% of the time. The reason is that threaded interpreters have a separate indirect branch for each VM instruction, which maps to a separate entry in the processors branch target buffer. We believe that accuracy can be further improved by further increasing the number of indirect branches. For example, conditional branch VM instructions could have two indirect branches rather than one, corresponding to the different directions of the branch. Replicating commonly used instructions will also increase the number of indirect branches. To avoid maintenence problems from multiple versions of instructions, we will modify the interpreter generator to do this automatically. Reducing branch mispredictions is the single most important optimisation for current interpreters, and we expect that it will reduce running time substantially.

Unfortunately, we know of no way to reduce the number of mispredictions in switch-based interpreters. This suggests that it may not be possible to build a particularly fast switch-based interpreter. Our experiments show that the threaded version of existing interpreters is up to twice as fast as the switch-based version of the same interpreter. As we apply optimisations to reduce branch mispredictions, the gap is likely to widen.

4.2 Automatic Instruction Combining

Interpreter overhead can be reduced by combining several Java instructions into one "super" instruction that behaves identically to the sequence, but has the overhead of a single instruction. Previous researchers have used interpreter generators to automatically combine commonly occurring sequences of instructions [Pro95]. Our interpreter generator will also do this, and we plan to experiment with various heuristics for identifying important sequences, based on static and dynamic frequencies.

Another interesting area is more general instruction combining. There is scope for combining not just sequences, but groups of instructions containing branches. For example, it may be that many basic blocks start with a load instruction, so a "branch and conditionally load" instruction may be valuable. Classic optimisations to increase the length of basic blocks, such as tail duplication, may also help instruction combining.

4.3 Run-time instruction combining

Perhaps the most remarkable research on interpreters in recent years is Piumatra and Ricardi's [PR98] technique which copies fragments of executable code to create longer sequences without jumps. This allows instruction combining to take place at run-time, when the program to be interpreted is already known. The technique can be applied to any threaded interpreter, and makes use of GNU C's label variables. They report that it reduces the running the running time of an already fast threaded interpreter by about 30% on a Pentium processor, with almost no reduction in portability or simplicity.

We are keen to evaluate this technique on more modern processors with longer pipelines. It appears to have the potential to allow interpreters to come within striking distance of the performance of JITs. In effect, it allows run-time code generation, but without loss of portability. Clearly, there are many open research questions about the use and effectiveness of this technique.

4.4 Translation to Register VM

The Java VM is a stack machine, which means that all computations are performed on the evaluation stack. For example, to implement the assignment `a = b + c`, one would use the VM instruction sequence `load b; load c; add; store a;`. Register VMs specify their arguments explicitly, so the same statement might be implemented with the instruction `add a, b, c;`. Note that the "registers" in a register VM are usually implemented as an array of memory locations. Register machines have two major advantages. First, as in the example, register machines may need fewer VM instructions to perform a computation. Although the amount of work may be the same, reducing the number of VM instructions reduces the dispatch overhead.

Secondly, register VMs make it much easier to change the order of VM instructions. It is very difficult to reorder stack machine instructions, since all instructions use the stack, so that every instruction depends on the previous one. Register instructions can be reordered provided provided there are no data dependences. Reordering opens new opportunities for instruction combining. We can change the sequence of instructions to better match our "super" instructions.

Register VMs have two important drawbacks. Stack machines implicitly use the top elements of the stack as their operands, whereas register machines must specify the operands. Decoding these operands adds additional overhead. Secondly, there is clearly a cost in translating the JVM stack code to register code. However, for switch-based interpreters, which suffer a branch misprediction for almost every VM instruction they execute, almost anything that reduces the number of executed VM instructions is likely to be of benefit.

4.5 Software Pipelining

Interpreter software pipelining [HATvdW99] reduces the effect of branches on interpreters by moving some of the dispatch code for the next instruction into the current instruction. It was developed for the Phillips Trimdia VLIW processor, which has no branch prediction. Fetching and decoding the next instruction while waiting for the branch for the current one to resolve allowed them to greatly speed up their interpreter.

Software pipelining is also likely to be useful on other processors as a way to reduce the cost of branch mispredictions. Moving the dispatch code into the previous instruction allows much of the dispatch to take place while waiting for the mispredicted branch to resolve. A problem with interpreter software pipelining, is a taken VM branch causes the pipeline to stall. Delayed VM branches can reduce this problem, but require a register VM rather than a stack one, since some VM instruction needs to be moved into the VM branch delay slot. Given that we already plan to experiment with register machines, we plan to investigate the usefulness of delayed VM branches.

4.6 Cache Optimisations

Cache misses have a large and growing effect on the performance of modern computers. Interpreters need little memory, so they work well with caches. Nonetheless, we will investigate how to reduce cache misses further by placing the code to interpret the most frequently executed Java instructions in the same cache lines. This may be very important if, as a result of instruction combining or other optimisations, we greatly increase the size of our VM instruction set.

5 Conclusion

We believe that these optimisations, especially branch prediction ones, will create a very fast Java interpreter. As our interpreter becomes faster, the proportional benefit of other lesser optimisations will increase. For example, if we can reduce running time of our basic interpreter by 50%, then a lesser optimisation that might not be considered worthwhile, since it might give only a 5% speedup on the original interpreter, would give a 10% speedup on the faster one. As interpreters become faster, anything that can eliminate some part of the running time will give a proportionally bigger benefit. Given their simplicity, portability, low memory requirements and reasonable performance, we expect that they will be an attractive alternative to just-in-time compilers for many applications.

References

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

[EG01] Anton Ertl and David Gregg. The structure and performance of efficient interpreters. To be submitted in February 2001 to the Europar 2001 European Conference on Parallel Computing., 2001.

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

[Kra97] Andreak Krall. Cacao - a 64 bit javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11), 1997.

[PR98] Ian Piumatra and Fabio Ricardi. Optimising direct threaded code by selective inlining. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–299. ACM, 1998.

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