An Optimized Java Interpreter for Connected Devices and Embedded Systems[*]

Andrew Beatty, Kevin Casey, David Gregg and Andrew Nisbet
Department of Computer Science
Trinity College
Dublin 2, Ireland
{Andrew.Beatty, Kevin.Casey, David.Gregg, Andy.Nisbet}@cs.tcd.ie

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 low memory requirements. In this paper we describe a new interpreter core for CVM, Sun Microsystem's JVM for connected devices and embedded systems. The interpreter core is portable and programmed in C. An interpreter generator is used to apply a number of optimisations automatically to the source code. Experimental results show that on benchmarks that spend almost all their time in the interpreter (rather than the run time system) it is 28% to 58% faster than the original CVM interpreter, and is only 5% to 9% slower than the highly-sophisticated, hand-tuned, assembly language interpreter in Sun's desktop JVM.

Keywords

Java, Interpreter, Embedded System

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. [8] 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.

*This work was supported by Enterprise Ireland Research Innovation Fund, Grant IF/2001/350.

A second important advantage of interpreters is that they can be constructed to be trivially portable to new architectures. 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 [14] 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.

Given these advantages, it is hardly surprising that Sun's JVMs for connected devices and embedded systems (CVM), for mobile devices (KVM), and for smart cards (JavaCard) are based on interpreters. Interpreters give the low memory requirements and rapid portability that are important for embedded systems. A weakness of using 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.

In this paper we describe a new interpreter for the CVM. The interpreter uses a number of optimizations, such as direct threading and optimization of constant pool accesses. The interpreter is complete and robust, in the sense that it runs all programs that run within the reduced feature

```
typedef enum {
  add /* ... */
} Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  int *sp;

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

Figure 1: Instruction dispatch using `switch`

set of the original CVM. Experimental results are presented for the standard SPECjvm98 and Java Grande benchmarks, which show the new interpreter is substantially faster than the original CVM.

The rest of this paper is organized as follows. Section 2 describes the basic functioning of a virtual machine interpreter, and the most important types of instruction dispatch. In section 3 we describe the design and implementation of our optimized interpreter core. Section 4 presents our experimental evaluation of our work with respect to other small JVMs. In section 5 we place our work in the context of existing published research on optimized Java interpreters. Finally, section 6 draws conclusions from our results.

2. VIRTUAL MACHINE INTERPRETERS

The Java Virtual Machine uses a stack-based bytecode to represent the program. Interpreting a bytecode instruction consists of accessing arguments, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction.

Instruction dispatch typically consumes most of the execution time in virtual machine interpreters. The reason is that most VM instructions require only a small amount of computation, such as adding two numbers or loading a number onto the stack, and can be implemented in a few machine code instructions. In contrast, instruction dispatch can require up to 10-12 machine code instructions, and involves a time consuming indirect branch. For this reason, dispatch consumes a large proportion of the running time of most efficient interpreters [4].

Switch dispatch is the simplest and most widely used approach. The main loop of the interpreter consists of a large `switch` statement with one `case` for each opcode in the JVM instruction set. Figure 1 shows how this approach is implemented in C.

Switch dispatch is simple to implement, but rather inefficient for a number of reasons. First, most compilers produce a range check to ensure that the opcode is within the range of valid values. In the JVM this is unnecessary, since the bytecode verifier already checks that bytecodes are valid. Secondly, the `break` is translated into an unconditional jump

```
typedef void *Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  Inst dispatch_table = { &&nop, &&aload_null, .... };
  int *sp;

  goto dispatch_table[*ip++];

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

Figure 2: Instruction dispatch using token threading in GNU C

```
typedef void *Inst;

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

  goto *ip++;

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

Figure 3: Instruction dispatch using direct threading in GNU C

back to the start of the loop. Given that the loop already contains a jump, it would be better to structure the loop as a set of routines that jump to one another. A final source of inefficiency results from there being only a single indirect branch for dispatching instructions. On machines with programmer visible pipelines, such as the Philips Trimedia processor for embedded systems, it is difficult to overlap this branch with other instructions [8]. On processors with branch predictors, this branch is very unpredictable [4]

An alternative to using a `switch` statement is threaded dispatch. Threaded dispatch is based on making explicit the sequence of steps generated by a compiler to implement a `switch` statement. Once these steps appear at the source level, the programmer can optimize the code by removing unnecessary work. Unfortunately, it is not possible to break a `switch` statement into its component parts in ANSI C, because there is no facility for `goto` statements that can jump to multiple different locations. To implement threaded dispatch, one requires a language with labels as first class value, such as GNU C, the language accepted by the GCC compiler.

Figure 2 shows how *token threaded* dispatch can be implemented using GNU C. The range check has been eliminated, as has the jump back to the dispatch routine at the end of the code for each VM instruction. Instead, the dispatch

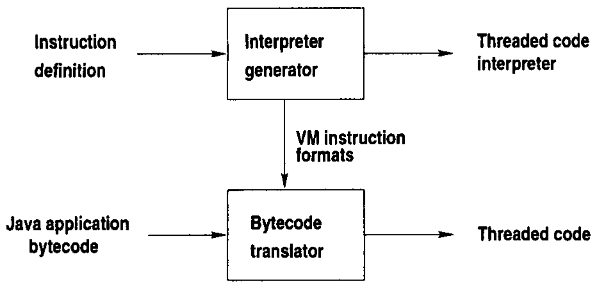

Figure 4: Structure of the Interpreter System

code is appended to the end of the code for each virtual machine instruction. This increases the size of the interpreter slightly, although it is usually faster. Another effect of replicating the dispatch code is that it allows the dispatch branch to be scheduled more efficiently with the code to implement the bytecode instruction, and it also greatly increases the prediction accuracy of the indirect branch on processors with branch target buffers (45% versus 2%–20% for switch dispatch) [4].

Much of the time in a token threaded interpreter is spent in getting from the value of the opcode (the "token") to the address of the routine to execute for this opcode. Thus, the opcode is loaded. This opcode is then used as an index into an array of addresses, so it must be scaled (shifted left) and added to the address of the base of the array. Finally, the address at that location is loaded.

A more efficient alternative is to translate the bytecode into a new format and replace the opcodes with the addresses of the routines that implement them. This scheme is known as *direct threading* [1] (see figure 3). It reduces the code to implement instruction dispatch to just three machine instructions on most architectures. Direct threading is usually the fastest instruction dispatch scheme. It is the most commonly used scheme where interpreter speed is important.

3. A JAVA INTERPRETER CORE

This section describes the core part of the interpreter, which is responsible for executing the Java bytecodes. The interpreter has been built into the CVM. CVM is 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.

Figure 4 shows the structure of our interpreter system. It is important to note that we do not interpret Java bytecode directly. Instead, the bytecode is translated into direct 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 of the interpreter ourselves. Instead, we use the **vmgen** [5] interpreter generator system, which produces an efficient interpreter from a specification of the behavior of each instruction. The following subsections describe the major components in more detail.

3.1 Instruction Definition

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

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

Figure 5: Definition of IADD VM instruction

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

Figure 6: Definition of a branch VM instruction

to implement the instruction. Figure 5 shows the definition of IADD. The instruction takes two operands from the stack (**iValue1,iValue2**), and places the result (**iResult**) on the stack.

We have made every effort to implement the instruction definitions efficiently. For example, operands in the JVM 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 array, 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 more efficient.

Figure 6 shows the specification for a branch VM instruction. Normally, the interpreter generator adds the necessary operations to dispatch the next VM instruction to the end of the specification code. In this case, however, we have added the keyword TAIL to the end of one direction of the branch. Wherever the keyword TAIL appears, a separate copy of the dispatch code is placed. In effect, this means that if the condition is **true**, one copy of the dispatch code will be executed, whereas if it is **false**, a different copy will execute. This is important because the dispatch code contains a computed **goto**, which will become an indirect branch when the interpreter is compiled. Two separate indirect branches are easier to schedule efficiently on processors with programmer-visible pipelines, and result in fewer branch mispredictions on machines with a branch target buffer (BTB).

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 dimensions of the array. Similarly, the various method invocation instructions consume a number of stack items equal to the number of parameters. We have overcome this problem by adding features to manipulate the stack pointer directly.

3.2 Interpreter Generator

The interpreter generator **vmgen** [5] 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, 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.

There are a number of advantages in 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 could easily be moved up the code. It is also easy to have both a threaded code and switch-based version of the interpreter. One need only add an option to the generator.

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 generator also allows us to add tracing and profiling code trivially, by defining macros. Furthermore, **vmgen** also automatically generates a disassembler for threaded code. Finally, the generator produces functions for the bytecode translator to write threaded code to memory in the correct format. These functions can be used to automatically apply peephole optimizations to the threaded code as it is generated (although we do not currently take advantage of this feature).

3.3 The Bytecode 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. The simplest, and most important example of this is the translation from bytecode to direct threaded code. The bytecode translator also replaces difficult to interpret instructions with simpler ones. For example, we replace instructions that access 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 bytecode to threaded code, and replaces itself with 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 precise garbage collector used by CVM requires that certain values normally kept in variables by the interpreter, such as the stack pointer, are spilled to memory from time to time. To ensure that the interpreter is never indefinitely in an unsafe state for the garbage collector due to loops or recursion, these values are spilled at method calls and at taken backward branches. In original Java bytecode, checks for this spilling must be made in the code for all branches, but in the threaded code version we use separate forward and backward branch instructions.

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. A similar strategy is used for constant pool accesses.

4. EXPERIMENTAL EVALUATION

Our basic thesis is that interpreters can be made very much faster by applying modern compiler optimization techniques to them, without the need to resort to assembly language. To test this thesis, we compared the performance of our interpreter to the performance of other small JVMs, using the SPECjvm98 [13] and Java Grande [2] benchmarks. These benchmarks consists of several large programs with real data, which are intended to be representative of a wide range of Java applications.

Our interpreter was built into CVM, a small implementation of the Java 2 Micro Edition (J2ME) standard. It supports the full JVM instruction set, as well as full system-level threads. We made a number of small additions to CVM to enable it to run the SPEC benchmarks and to allow us to safely compile it at a higher level of optimization than the standard distribution. We compiled CVM with GCC 2.96 using the optimization flags "`-O4 -fomit-frame-pointer`". In addition, we used the flag "`-fno-gcse` (disable global common subexpression elimination) on the file containing the core of the interpreter, as GCC sometimes moves global subexpressions to more frequently executed parts of the program when control flow is complicated. The running times are for a Pentium 4 based system running Red Hat Linux version 7.3.

In addition to our interpreter, we also tested the original interpreter that comes with CVM. This is a token threaded interpreter, which uses GCC's labels as values to implement threaded dispatch (see section 2). It is also possible to configure the original CVM to use switch dispatch. However, this makes it 17% to 52% slower according to our measurements. For this reason, no results for CVM with switch dispatch are presented. CVM offers a choice between a semispace garbage collector, and a generational one. We used the former for all our experiments, since it is slightly (1%–3%) faster.

For comparison, we also measured the speed of the widely used Kaffe JVM. Kaffe is a freely available, robust, highly portable JVM which is available under the GNU General Public License. A commercial version of Kaffe is sold for use in embedded systems. It is important to note that the results we present are for the public version not the commercial one, although the two versions share much code. With Kaffe, we tested both the interpreter and the JIT compiler. The results show what can be expected from a simple, unoptimized interpreter, and a small, portable JIT compiler.

Finally, we measured the performance of Sun Microsystem's desktop implementation of Java 2 Standard Edition (J2SE) interpreter from the Hotspot Client VM. Hotspot uses a sophisticated interpreter, coded in hand-tuned assembly language. In addition to careful assembly language programming, it uses a number of optimizations, such as combining common sequences of bytecode instructions into superinstructions [11], and processor (x86) specific optimizations for floating point operations. It also stores the topmost elements of the stack in registers and uses a complicated stack-caching [3] system for managing the various states of the stack. To avoid code explosion due to stack

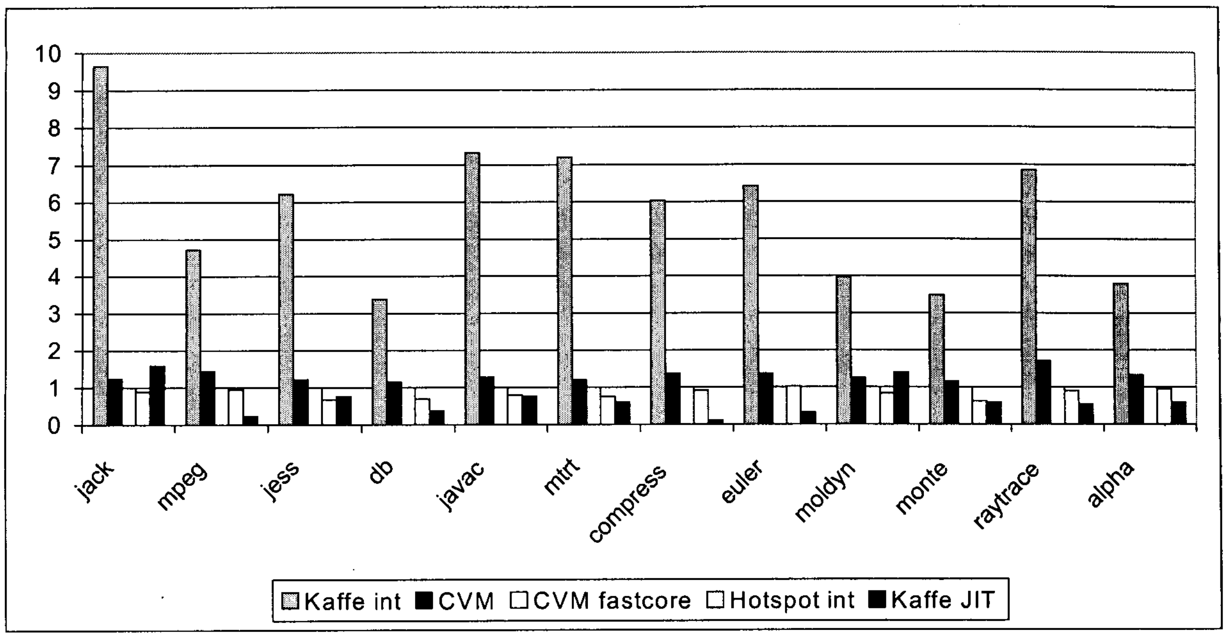

Figure 7: Running times of the benchmarks on various JVMs relative to our interpreter (CVM fastcore).

caching states, the machine code of the interpreter is generated in memory at run time. The result is that although the Hotspot interpreter is fast, it is complicated and completely unportable.

Figure 7 shows the running times of the benchmarks running on each of the implementations of the JVM, relative to our interpreter (CVM fastcore, whose speed is always represented as 1). The most striking result is for the Kaffe interpreter, which is on average 5.76 times slower than our interpreter. The Kaffe interpreter is not at all optimized. In particular, it resolves method names and constant pool references every time they are used, rather than once, the first time they are used. The Kaffe interpreter demonstrates very well that it is easy to write a very inefficient interpreter.

The original CVM interpreter is an average of 31% slower than our optimized interpreter. It does particularly well on *db*, where it is only 16% slower. We investigated the reason for this variance by profiling the code. We found that in the original CVM, only 87% of the time for the *db* benchmark is spent in the interpreter. The rest of the time is spent in the run time system, on garbage collection, synchronization, and native methods. In contrast 98.74% of the time for *mpeg* and 99.89% of the time for *compress* is spent in the interpreter. So although the speedup in the interpreter core is similar across all programs, the overall speedup for *db* is lower, since there is no change in the running time of the run time system. For this reason, programs such as *compress* (38% faster), *mpeg* (44% faster) and *raytrace* (71% faster) give a better indication of the relative speeds of the interpreter cores.

The Hotspot interpreter is on average 20.4% faster than the version of CVM with our interpreter. 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. Our interpreter is actually a little (1.9%) faster on *euler*.

Finally, the Kaffe just-in-time (JIT) compiler is on average more than twice as fast as our version of CVM. In fact, looking at the results for *mpeg* and *compress* shows that it is four to eight times faster at executing bytecodes than our interpreter. On other benchmarks, its poor run time system slows it down to the extent that it is actually substantially slower than CVM on the *jack* benchmark. This nicely demonstrates that a JIT compiler does not *always* guarantee better performance than an interpreter. The run time system is also important. Something that should also be noted is that the Kaffe JIT compiler does not produce especially fast code. In particular, the mixed-mode Hotspot compiler/interpreter for desktop machines is usually more than twice as fast. However, the Kaffe JIT compiler is simple, and similar to the commercial version, which is used in embedded systems.

5. RELATED WORK

Recent important developments in interpreters include the following. Interpreter generators simplify construction and maintanance of interpreters and can allow automatic VM instruction combining [11] and stack optimizations [5]. Stack caching [3] is a general technique for storing the topmost elements of the stack in registers. Ertl and Gregg [4] showed that interpreters (especially those using switch dispatch) spend most of their time in branch mispredictions on modern desktop architectures. Costa [12] discusses various smaller optimizations.

The Sable VM [6] is an interpreter-based research JVM. This interpreter is intended for desktop systems and uses a run-time code generation system [10], not dissimilar from a just-in-time compiler. Gregg et al. [7] presented a prototype interpreter based on the Cacao research JVM [9]. The interpreter used an earlier version of vmgen, but was not designed for embedded systems, could run only a handful of programs, and did not support many language features, such as multithreading. In contrast, the interpreter described in this paper is a full implementation of the J2ME standard, designed specifically for embedded systems, and runs all programs that we have tried.

Venugopal et al. [15] 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. In contrast the JVM described in this paper is fully general purpose, and suitable for connected devices and embedded systems that download and execute Java programs from other sites.

6. CONCLUSION

We have described an efficient interpreter core for connected devices and embedded systems. Our interpreter system uses a generator to produce efficient code from a virtual machine instruction specification. It translates the original Java bytecode to threaded code, reducing the interpreter overhead. The translator also computes constants, targets and offsets at translation time, allowing us to simplify the interpretation of many instructions, such as method invocations.

Experimental results show that the original CVM virtual machine is about 31% slower than CVM our interpreter core. However, a substantial amount of that running time is spent in the run-time system. On the *mpeg* benchmark, where almost all time is spent executing bytecodes rather than in the run time system, the difference is 44%. Our interpreter for embedded systems is competitive with the Hotspot interpreter, a highly sophisticated, hand-tuned, assembly language interpreter, with a much faster (and larger) run time system. Hotspot is on average about 20.4% faster, but on bytecode intensive benchmarks such as *mpeg* and *compress* it is only 5%-9% faster. On average our version of CVM is a factor of 5.76 times faster than the Kaffe interpreter, and can even be competitive with the Kaffe JIT compiler on some benchmarks.

7. REFERENCES

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

[2] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, April 2000.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] V. Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[13] 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.

[14] Sun Microsystems Inc. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000.

[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.