JavaVM Implementation: Compilers versus Hardware

Andreas Krall[1] and Anton Ertl[1] and Michael Gschwind[2]

[1] Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{andi,anton}@complang.tuwien.ac.at
[2] Institut für Technische Informatik, Technische Universität Wien
Treitlstraße 1, A-1040 Wien
mike@vlsivie.tuwien.ac.at

Abstract. The Java Virtual Machine (JavaVM) has contributed greatly to Java's success because it provides a common intermediate format which can be shared across the Internet. Unfortunately, the JavaVM has been optimized for an interpreted model, resulting in inferior performance because its stack-based execution model cannot exploit instruction-level parallelism. The inherent serialization of the stack execution model can be addressed either by using compilation techniques or by hardware. In this article, we review the different JavaVM implementation methods based on our experiences with the implementation of the CACAO just-in-time compiler. For comparison, we have also investigated different hardware architectures for the direct implementation of the JavaVM.

1 Introduction

Java's [Arnold and Gosling, 1996] success as a programming language results from its role as an Internet programming language. The basis for this success is the machine independent distribution format of programs with the Java Virtual Machine (JavaVM) [Lindholm and Yellin, 1996]. The standard interpretive implementation of the JavaVM makes execution of programs slow. This does not matter if small applications are executed in a browser, but becomes intolerable if big applications are executed. There are two solutions to solve this problem:

- specialized JavaVM processors
- compilation of byte code to the native code of a standard processor

Sun took both paths and is developing both JavaVM processors and native code compilers. Our JIT compiler (CACAO) is described in detail in [Krall and Grafl, 1997] and is freely available via the world wide web. In this work, we compare compilation techniques such as just-in-time (JIT) compilation and specialized JavaVM hardware.

1.1 Previous Work

Many attempts at language-specific architectures have been made, for a wide variety of languages including LISP [Steele and Gabriel, 1996], Prolog [Holmer et al., 1996], Smalltalk [Ungar, 1987], Forth [Hayes and Lee, 1989], Algol (Borroughs), Modula-2 [Ohran, 1984], Java [Lindholm and Yellin, 1996], and many

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
John Morris (ed.), Computer Architecture (ACAC '98), Perth, pages 101--110
Australian Computer Science Communications 20(4), Springer

others. Depending on the requirements of the supported language, the proposed architecture has ranged from stack-based architectures to extensions for conventional RISC architectures (such as the SPUR, BAM, SOAR and SPARC architectures).

Stack architectures have a long history in the implementation of computer architectures and in the design of intermediate representation for compilers and interpreters. Originally, stack architectures were seen as an ideal representation for compilation, as code generation was easy and more advance compilation techniques had not yet been developed. Consequently, the appeal of stack architectures was great, affording to use the same simple representation for internal representation and as input to interpreters of computer architectures.

Well-known implementations of stack architectures were manufactured by companies such as Burroughs (now Unisys), Hewlett-Packard, or Tandem. Alas, these architectures were eventually outperformed by RISC architectures and today only a few implementations are still available. Many others have been migrated to RISC architectures, sometimes supported by emulation and binary translation techniques.

More recent stack-based architectures include a number of Forth processors [Koopman, 1989; Hayes and Lee, 1989]) and the JavaVM. While the Forth architectures have never gained any widespread acceptance, the jury is still out on the JavaVM.

The first JavaVM implementations available were interpreter-based, but their performance was disappointing. In order to provide acceptable performance for JavaVM-based applications, a number of solutions have been proposed and implemented:

just-in-time compilation Several JIT compilers have become available over the past year, such as the publicly available *kaffe* [Wilkinson, 1997] and CACAO [Krall and Grafl, 1997] systems, as well as products from Sun (for the SPARC and PowerPC architectures), SGI (MIPS RISC) and Microsoft, Netscape, Symantec (for the Intel x86).

batch compilation Batch translation can be performed in advance to generate directly executable native machine code from the Java class files. This approach has the advantage that sophisticated code generation and optimization techniques can be used. Examples for such systems are the *Caffeine* system [Hsieh et al., 1996] which directly generates object code and the Toba system [Proebsting et al., 1997] which uses C as intermediate representation.

hardware implementation Sun has announced a hardware implementation of the JavaVM, called picoJava-I [O'Connor and Tremblay, 1997].

Both JIT and native compilation techniques can be improved by appropriate hardware primitives designed to support efficient JavaVM execution. An example for this approach is the DAISY (Dynamically Architected Instruction Set from Yorktown), a VLIW architecture developed at IBM for fast execution of x86, PowerPC, S/390 and JavaVM code [Altman and Ebcioğlu, 1997]. Compatibility with different old architectures is achieved by using a JIT compilation technique. The JIT compilation scheme for the JavaVM is described in [Ebcioğlu et al., 1997].

1.2 The Java Virtual Machine

The JavaVM is a typed stack architecture [Lindholm and Yellin, 1996]. A JavaVM implementation has to check the program for type correctness and executes only correct programs. The JavaVM specification also defines the format of Java class files, which include code, data (constant pool) and type information for each Java class.

There are different instructions for integer, long integer, floating point and address types. Byte and character types have only special memory access instructions and are treated as integers for arithmetic operations. The main instruction set consists of arithmetic/logical and load/store/constant instructions. There are special instructions for array and field access (memory access), method invocation, function call, type checking, and complex table branches. The complex instructions provide an abstraction layer by hiding implementation details, thereby offering different implementation choices to JavaVM implementors.

2 Compiler

The architecture of a RISC processor is completely different from the stack architecture of the JavaVM. RISC processors have large sets of registers and execute arithmetic and logic operations only on values which are held in registers. Load and store instructions are provided to move data between memory and registers. Local variables of methods usually reside in registers and are saved in memory only during a method call or if there are too few registers.

2.1 Machine code translation examples

If JavaVM code is translated to machine code, the stack is eliminated and registers replace the stack slots [Ertl, 1992]. A naive translation scheme just would represent each stack location by a register and would generate a lot of copying instructions which just move values between local variable registers and stack registers. A more optimized translation scheme as implemented in the CACAO system does not generate these copying instructions and generates code which is equivalent to code generated by a C compiler.

The example expression `a = b * c + d` has the JavaVM code given on the left and will be translated to the two Alpha instructions on the right (the variables a, b, c and d reside in registers):

`iload b`	load contents of variable b	`MULL b,c,tmp0`	tmp0 = b * c
`iload c`	load contents of variable c	`ADDL tmp0,d,a`	a = tmp0 + d
`imul`	compute b * c		
`iload d`	load contents of variable d		
`iadd`	compute (b * c) + d		
`istore a`	store stack top in variable a		

2.2 Method invocation

A method invocation as described by the JavaVM is a complex instruction. Many implementation issues are left over for the implementor. On a RISC processor a good choice is to store in a stack frame only spilled registers (including the return address register) and nothing else. The parameters are passed via argument registers. Fast register allocation is done by preassigning argument registers before register allocation of local variables and stack registers.

The example method

```
public int add (int a, int b) {
        return a + b;
        }
```

has the JavaVM code given on the left and will be translated to the two Alpha instructions on the right (the parameters `a` and `b` reside in parameter registers):

`iload a`	load contents of variable a	`ADDL a,b,v0`	resultreg = a + b
`iload b`	load contents of variable b	`RET`	
`iadd`	compute a + b		
`ireturn`	return method result to caller		

Invoking a method with the command `o.add(x * y, z - 3)` has the JavaVM code given on the left side and will be translated to the Alpha instructions on the right (registers `a0`, `a1` and `a2` are argument registers):

`aload o`	load reference to o	`MOV  o,a0`	argreg0 = o
`iload x`	load contents of variable x	`MULL x,y,a1`	argreg1 = x * y
`iload y`	load contents of variable y	`SUBL z,3,a2`	argreg2 = z - 3
`imul`	compute x * y	`LDQ  mp,0(a0)`	load class pointer
`iload z`	load contents of variable z	`LDQ  mp,add(mp)`	load address of add
`iconst_3`	push constant 3	`JSR  (mp)`	call o.add
`isub`	compute z - 3		
`invokevirtual add`	call o.add		

2.3 JIT compiler

A JIT compiler like the CACAO system [Krall and Grafl, 1997] translates JavaVM code into native code on demand. Each method is translated when it is invoked for the first time. Because the JIT compiler operates at run time, only medium complexity optimizations can be performed. In effect, only optimizations where the increased compilation time is offset by reduced run time are worthwhile. For example, optimizations like graph coloring register allocators or global instruction scheduling cannot be used because of their high cost. Therefore, fast and simple register allocation techniques and basic block scheduling techniques are applied.

The advantage of a JIT compiler is that information about the complete program (like the class hierarchy) is available. It has information about final classes (without being declared final) and can perform inlining on more methods than a Java to JavaVM compiler.

2.4 Batch Compiler

Batch compilers directly generate native code which can be executed by the processor. Because the batch compiler is not executed at run time, but at compile time, the compiler can use more sophisticated optimization strategies than a JIT compiler, in particular optimizations like loop-invariant code motion, strength reduction, and global register allocation, possibly even interprocedural optimizations like type analysis for determining monomorphic method invocations [Diwan et al., 1996], inlining, and interprocedural register allocation. Interprocedural optimizations are important given the high frequency of calls and returns in Java programs (together 7% of the dynamically executed JavaVM instructions).

3 Java hardware

Hardware support for the JavaVM can be achieved with different approaches. A brute force approach consists of using the JavaVM as the instruction set architecture of a microprocessor and implement all JavaVM functionality in hardware. An alternative, hybrid approach can be to supply additional features in a RISC or VLIW instruction set to support efficient JavaVM execution using just-in-time compilation or interpretation.

3.1 An existing JavaVM processor: the picoJava-I

Currently, the only existing hardware implementation of the JavaVM is Sun's picoJava-I architecture [O'Connor and Tremblay, 1997]. The picoJava-I architecture uses a mixture of hardwired implementation, state machines, microcode and software to implement the functionality of the JavaVM. This approach is necessary because the JavaVM is optimized as intermediate language for interpretation, using a number of high-level instructions to reduce interpretation overhead.

As a result, JavaVM implementation suffers from many of the same problems that CISC instruction sets suffer, such as variable length instructions and instructions which cannot be implemented in hardware without inordinate resource usage.

The picoJava-I architecture uses in-order execution based on a 4 stage pipeline with fetch, decode, combined execution/memory and write back phases. Pipelining of execution and memory access would not result in any further gains because any operation following a memory access would most likely require the result loaded by the memory phase due to the stack execution model. To optimize the implementation of the stack architecture, the picoJava-I implements a register file which is used as a stack cache for the top 64 stack elements.

The instruction decode unit is relatively straight forward, decoding one variable-length instruction per clock cycle. To improve processor utilization and performance, the instruction decoder implements a *folding* mechanism to map multiple JavaVM instructions to a single hardware operation: in the JavaVM, instructions frequently copy values from local variables to the top of the stack immediately before the instruction which consumes that value. These instructions are merged into a single hardware operation which receives the value directly from the local variable, reducing the number of copy operations required for JavaVM execution. This simple folding mechanism reduces the instruction count by 15%.

Since references are resolved at run time in the JavaVM specification, the JavaVM instruction set includes a number of instructions referencing objects symbolically. In the picoJava-I architecture, these instructions cause a trap to the operating system which is responsible for resolving references. The reference instructions are then replaced by quick variations directly executable in JavaVM hardware. This reduces the time to execute the instruction the next time it is encountered.

In addition to the instructions defined in the JavaVM specification, the picoJava-I architecture implements a number of instructions for systems programming (e.g., I/O, cache management, garbage collection).

3.2 High-performance hardware implementation issues

The design of the JavaVM as an interpreted intermediate language poses a number of challenges for the implementation of an aggressive high-performance architecture. Two problem areas can be identified in the specification of the JavaVM.

- Use of the JavaVM specification results in a number of instructions with high complexity.
- The stack architecture limits parallelism.
- Variable-length instructions complicate parallel instruction decoding for superscalar instruction issue.

High complexity instructions (such as method invocation and the `lookupswitch`) are difficult to accommodate in pipelined architectures. Possible solutions to this problem are the use of an instruction decode unit which translates complex JavaVM instructions into a series of pipelined micro-operations (as used in implementations of the Intel x86 architectures) or trapping to a software implementation. These approaches are complementary, and both can be used in a single design to implement instructions of different complexity, as used in the picoJava-I architecture. While a solution based on software traps results in a degradation of performance, hardware based interpretation (based on the generation of micro-ops in the instruction decode unit) increases design complexity.

The decision to use a stack architecture for the JavaVM incurs two problems when trying to design high performance architectures, namely the implicit dependency of instructions on preceding instructions and the presence of move operations of local variables and program constants to the stack before they can be referenced.

The dependencies introduced by the stack model can be resolved by a two step scheme: first, all stack references must be translated to register references and anti-dependencies have to be removed by register renaming. Unfortunately, this still leaves the code with long chains of flow dependencies from the original instruction order. Thus, an out-of-order execution mechanism has to be employed to generate an optimal schedule. Without out-of-order execution, it is not possible to exploit the performance potential of super-scalar and deeply pipelined architectures.

A final problem with the JavaVM architecture is the abundance of move instructions necessary to execute a program which is inherent in stack architectures. O'Connor and Tremblay [O'Connor and Tremblay, 1997] report that 48.3% of all dynamic instructions only move data and constants between different types of registers.

To eliminate these moves, register renaming has to be enhanced: instead of moving a value between two registers, the mapping table can be changed such that two logical registers are mapped to the same physical register and the move instruction can be squashed. This approach is similar to the copy elimination approach used in our JIT compiler. While this approach can be used to eliminate spurious moves, it complicates the management of physical registers because more than one logical register can be mapped to a single physical register. To know whether a physical register can be re-used, either a reference count field has to be added to each physical register or a content-addressed search in the mapping table is necessary.

As discussed above, the problems arising from using the JavaVM as ISA can be solved, but at the price of considerable hardware complexity and possible cycle time penalties. We think that the hardware complexity invested in making the JavaVM ISA feasible would be better invested in additional performance-enhancing features, such as increased instruction-level parallelism.

3.3 Hardware/Software solutions

An alternative solution to using a brute force hardware implementation approach of the JavaVM is to design an architecture with appropriate support for interpretation or JIT compilation.

An example of a VLIW architecture with extensive JIT support for binary compatibility with various (CISC and RISC) instruction set architectures is the DAISY (Dynamically Architected Instruction Set from Yorktown) architecture reported in [Altman and Ebcioğlu, 1997]. Compatibility with different old architectures (such as the x86, PowerPC, S/390 and the JavaVM) is achieved by using a JIT compilation technique. Altman and Ebcioğlu argue that with appropriate support, JIT compilation is an effective way to efficiently support different instruction set architectures while taking advantage of the increased instruction-level parallelism possible in VLIW machines.

When performing just-in-time compilation, the primitive instructions found in the target architecture should facilitate efficient mapping from the emulated architecture. For a JavaVM architecture, we have identified a number of operations crucial for efficient execution:

null pointer check The null pointer check can be implemented using the MMU in most microprocessors. By blocking access to the low memory area, all accesses to a null pointer will generate a trap.

bounds check Access to arrays requires a bounds check to validate the array index. A single cycle implementation of the bound check is possible using an unsigned variant of the subtract and trap on overflow instruction (e.g., SUBV on the Alpha).

array access A register plus shifted register memory addressing mode can be used to optimize array accesses (such as found on the Motorola 88k). While native code compilers can use strength reduction to eliminate many such accesses, JIT compilation can perform only “cheap” optimizations.

method invocation For JavaVM applications register windows can be used to reduce the cost of method invocation. Dynamic opcode distribution shows a call/return to computational instruction ratio of 7.3 to 9.2, so efficient method invocation is necessary to achieve good performance.

constant pool PC-relative addressing allows to cheaply access the constant pool associated with each method.

garbage collection Hardware support for efficient garbage collection could be useful, although our benchmarks suggest that garbage collection times are not a major performance bottleneck.

We feel that a combined hardware/software solution has the best potential to achieve good performance in the execution of JavaVM code. A pure hardware solution is burdened with the complexity of some JavaVM instructions which can be resolved easily in software (such as translating `lookupswitch` to a cascade of conditional branches). On the other hand, a number of operations can be optimized by using an appropriate hardware implementation and support a software-based JIT compilation scheme.

4 Comparison

In this section, we compare the respective advantages of JIT compilation and custom Java hardware for different market segments. The suitability of the possible solutions depends on the differing requirements of price, price/performance or performance, as well as other characteristics.

4.1 Memory requirements

It has previously been argued that one of the advantages of JavaVM-based applications is a small memory footprint, which is important for some market segments. To investigate this claim, we have performed some empirical measurements of key Java programs such as the `javac` compiler. To obtain these measurements about program and data size for different JavaVM implementation approaches, we have instrumented our CACAO JIT compilation system to report statistics about referenced class files, JIT compilation and the generated object code.

	size (kB)	**hardware**	**interpreter**	**JIT**	**batch**
heap	> 2000	•	•	•	•
JavaVM code	149	•	•	•	
native code	447			•	•
constant pool	731	•	•	•	
virt. func. tbl.	103	•	•	•	•
runtime system	248		•	•	
total		2983	3231	3678	2550

The measurements were performed on a DEC Alpha, resulting in increased data requirements to accommodate 64 bit words and pointers. Requirements are reduced for architectures using smaller processor words. The constant pool as given in this table reflects the size of the constant pool when decompressed into memory for access. The original size of the constant pool was 484 kB. The size of the runtime system given here reflects the size of the CACAO compiler, the JavaVM interpreter from Sun requires 455 kB.

4.2 Embedded control

In the embedded control market, JavaVM hardware (e.g., the picoJava-I) will try to occupy a niche in the high end, competing with embedded control implementations of RISC architectures, such as the PPC 403, the StrongARM, or the MIPS R4300.

We can identify a number of problems associated with the use of JavaVM in the embedded control market:

- Compiling directly for a target architecture offers significant advantages (global optimizations).
- JavaVM hardware will suffer in this market from being specific to Java and similar languages.
- JavaVM code is not suitable for stand-alone applications in a ROM, as program operation requires that symbolic instructions be replaced by their quick counterparts.

Most embedded control applications can afford compiling directly for the architecture of the target processor, instead of going through the JavaVM. This results in higher code quality and lower memory requirements than for the JavaVM JIT compilation and simpler hardware than using the picoJava-I processor.

In some applications, software is downloaded relatively frequently to the embedded computer; in these applications a standardized format like Java class files will often be preferred to an application or board-specific download format; this leaves us with JavaVM hardware or JIT compilers, with JIT compilers suffering penalties due to their memory requirements.

In the embedded control market the code size can be very important. Stack machines in general and the JavaVM in particular are claimed to be very space-efficient. The average instruction size for the JavaVM is said to be 1.8 bytes [O'Connor and Tremblay, 1997], but ignores additional resources required for the constant pool. Conventional RISC instructions require 4 bytes, but architectures such as the ARM Thumb and the MIPS16 use a denser encoding. Moreover, as long as we do not know if the programs for the different architectures have the same number of instructions, the average instruction length is meaningless.

We think that it will be difficult for JavaVM based solutions to replace the currently used designs used in the embedded market, because for most applications, JavaVM based solutions do not have any inherent advantages.

4.3 Low-end NCs

For low-end network computers, both direct JavaVM hardware and JIT compilation solutions are feasible. Some issues have been raised about the cost of JIT compilation, but, actually, the required resources are moderate: the CACAO system requires about 250K for the program and compilation for a non-trivial program such as the `javac` takes 80 milliseconds and requires 50KB temporary space and 380K for the generated native Alpha code and constant area.

One concern about a JIT compiler for an NC on a high-speed network may be the increased load time for the applications. However, the loader has to check the class file anyway, so there is already a delay. Moreover, JIT compilers like CACAO, compile only those methods that are executed, when they are executed, minimizing compile time, space usage, and, in particular, the load delay.

The code a JIT compiler produces is not as good as the code of a (slower) batch compiler. In particular, the register allocation is not as good, causing a higher number of moves. Only a simple instruction scheduler can be used. Finally, optimizations like strength reduction for array accesses that are not possible in the Java→JavaVM compilation, are typically not performed by a JIT. The expected end result is similar performance between picoJava-I and a JIT compiler/simple RISC combination.

4.4 High-performance NCs

As previously discussed, high-performance implementations of the JavaVM are certainly possible: by using a register renamer and out of order execution, a superscalar JavaVM processor implementation can be kept busy. Such an implementation could conceivably outperform a JIT/high-performance RISC combination, but at considerable hardware complexity and cost.

Truly high-performance systems can be built at about the same complexity level by using VLIW architectures with appropriate JIT compilation support, as reported in [Altman and Ebcioğlu, 1997].

5 Conclusion

In this article, we have presented two implementation approaches of the JavaVM: the CACAO JIT compilation system and dedicated JavaVM hardware processors. We have reviewed the potential of JavaVM compilers and JavaVM hardware. Because of greater flexibility of compilers, support for other programming languages and similar performance at lower cost, we believe that JavaVM compilers are superior to JavaVM hardware.

References

Altman, Erik and Ebcioğlu, Kemal (1997). DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA'97 - The 24th Annual International Symposium on Computer Architecture*. ACM, IEEE.

Arnold, Ken and Gosling, James (1996). *The Java Programming Language*. Addison-Wesley.

Diwan, Amer, Moss, J. Eliot B., and McKinley, Kathryn S. (1996). Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 292–305.

Ebcioğlu, Kemal, Altman, Erik, and Hokenek, Erdem (1997). A Java ILP machine based on fast dynamic compilation. In *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*.

Ertl, M. Anton (1992). A new approach to Forth native code generation. In *Euro-Forth '92*, pages 73–78, Southampton, England. MicroProcessor Engineering.

Hayes, John and Lee, Susan (1989). The architecture of the SC32 Forth engine. *Journal of Forth Application and Research*, 5(4):493–506.

Holmer, Bruce K., Sano, Barton, Carlton, Michael, van Roy, Peter, and Despain, Alvin M. (1996). Design and analysis of hardware for high-performance Prolog. *Journal of Logic Programming*, 29(1-3):107–139.

Hsieh, Cheng-Hsueh A., Gyllenhaal, John C., and Hwu, Wen-mei W. (1996). Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'29)*.

Koopman, Jr., Philip J. (1989). *Stack Computers*. Ellis Horwood Limited.

Krall, Andreas and Grafl, Reinhard (1997). CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9:to appear.

Lindholm, Tim and Yellin, Frank (1996). *The Java Virtual Machine Specification*. Addison-Wesley.

O'Connor, J. Michael and Tremblay, Marc (1997). picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53.

Ohran, Richard Stanley (1984). *Lilith: a workstation computer for Modula-2*. PhD thesis, ETH Zürich.

Proebsting, Todd A., Townsend, Gregg, Bridges, Patrick, Hartman, John H., Newsham, Tim, and Watterson, Scott A. (1997). Toba: Java for applications. Technical report, University of Arizona, Tucson, AZ.

Steele, Guy L. and Gabriel, Richard P. (1996). The evolution of Lisp. In *History of Programming Languages*, pages 233–309. ACM Press/Addison-Wesley.

Ungar, David (1987). *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press.

Wilkinson, Tim (1997). KAFFE: A free virtual machine to run Java code. `http://www.kaffe.org`.