Why Java is Best Suited for the Intel Architecture

Kevin J. Smith
Microcomputer Lab
Intel Corp.
February 9, 1998

Abstract

Existing performance results show that Intel processors are exceptionally efficient at running Java programs.[1] Intel has conducted a number of studies to determine what else could be done to deliver even greater Java performance. The studies conclude that Java runs exceptionally well on the Intel architecture because the architecture and instruction set were designed to be general purpose--so it easily adapts to various computational scenarios. The Intel Architecture (IA) performs well on integer programs as well as on scientific calculations. Thus, it comes as no surprise that Intel architecture is so well suited to the dynamic nature of Java.

Introduction

Programmers can write programs in the Java language. After the program is written, it is compiled into Java byte codes. The byte codes are executed either as a standalone application, or as an applet running under the control of a browser. The byte codes need to be either interpreted or translated into the instruction set of the processor in the system. Typically, the byte codes are translated into the native instruction set by a Just-In-Time Compiler (JIT). It is “Just in Time” because the translation is delayed until just before the code needs to be executed.

Ever since the first compilers were written in the late 1950’s, compiler writers have devised ways to make programs run faster. The set of transformations are known as optimizations. These optimizations make the program run faster without changing the results of the program. In Java, the translation is done during program execution, so the time it takes to translate needs to be minimized, which implies that only limited optimizations can be done.

Instead of translating the byte code to the Intel Instruction set, let’s compare that strategy with a chip that could execute the byte codes directly; let's refer to this as a byte code chip. A byte code chip would not be able to eliminate redundant instructions—it must execute the code presented to it; whereas a JIT can further optimize the code as the JIT selects the native instructions for the processor. The byte code chip also has a continuous overhead of having to interpret each byte code instruction every time it sees it, whereas the JIT only translates and optimizes it once. A byte code chip may have a niche when there are objectives other than performance, such as in the embedded market. Even Sun agrees, “Java does not need heavy architectural support to run efficiently.”[2]

This paper will detail how Intel’s current architecture is well suited for Java. The key performance factors that distinguish Intel processors with regard to Java include:

1) Intel’s low call/return overhead relative to most RISC-based processors
2) Small code size that packs more instructions into the instruction cache

[1] www.webfayre.com/pendragon/jpr. *October 1997.*
[2] Hoelzle, Bak, Grarup, Griesemer, Mitrovic. “Java On Steroids: Sun’s High-Performance Java Implementation”. Hot Chips IX, Stanford, CA. Aug 25-26, 1997. p. 23.

3) Rich addressing modes, wide range of immediate constants, and compound instructions which characterize a CISC architecture and are quite advantageous to Java.

The combination of Intel's existing processors and a high quality JIT provides the best performance for JIT compiled code.

Intel Architecture Wins out

The Intel Instruction Set Architecture (ISA) is known for its Complex Instruction Set (CISC) . Typically, one CISC instruction can take the place of several RISC instructions. Byte codes do not contain CISC - like constructs, but typically several adjacent byte code instructions can be combined into a single CISC instruction by the JIT. Furthermore, this optimization is quite fast and requires minimal analysis to trigger. However, the JIT shouldn't spend too much time optimizing, since JIT time is part of the total execution time of the program. Fortunately, the optimizations with the highest payoff on the Intel Architecture are quick and easy to do.

"The gap between JIT and off-line compilers is greater for the SPARC than for the Pentium® [Processor]. This is due to the fact that binary code for modern RISC processors is complex to optimize and requires analyses that are hard to run in the short time allocated to on the fly compilation."[3]

The Intel architecture contains some of the richest formats to embed immediate constants into the instructions. This eliminates the need for extra data memory traffic because the data cache doesn't need to be accessed.

Sun stated that in the PicoJava implementation, constants in adjacent instructions can be folded into the instruction. Sun's data shows that this eliminated about 4.8% of the total dynamic instruction count.[4] With the Intel architecture, the constant can also be pushed a few instructions before the use (a JIT doesn't require adjacent instructions in order to combine); whereas with PicoJava, the constant push must be the immediately preceding instruction. In the Intel instruction set, immediate constants can be up to 32 bits long, so there will be more opportunities to put constants directly into the instructions on IA because it isn't limited to small 8 or 16 bit constants.

Furthermore, Sun stated that they can reduce total instruction counts by 15% when stack operations (loads) are subsumed into the adjacent instruction that uses the result of the load.[5] The Intel ISA is rich in compound instructions such that loads can generally be subsumed into other operations; furthermore, non stack loads can also be subsumed, thereby reducing total instruction count and code size even more. And to take this one step further, storage of the result can also be incorporated into the same instruction!

On top of that, the Intel ISA has multiple addressing modes that are part of the memory reference instructions. This can often avoid the equivalent of two instructions on typical RISC processors.

Call Return Overhead

Since Java is an object oriented language, there is a much higher density of call and return instructions than with traditional languages like C and Fortran. One of the major performance factors when invoking methods is the overhead of call/return sequences, which typically include saving and storing of some registers. On the Intel Architecture side, there are only eight registers; traditional RISC architectures typically have 32 integer registers. Since fewer registers need to be saved/restored, there is less call/return overhead relative to most RISC processors.

---

[3] Muller, Moura, Bellard, Consel. "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code". (To appear) 3rd Conference on Object-Oriented Technologies and Systems (COOTS). p. 18.

[4] O'Connor, Taremblay. "PicoJava-1: The Java Virtual Machine in Hardware". IEEE, Mar/April 1997. p. 50.

[5] Ibid.

One could change the calling conventions on a RISC processor to use only eight registers so the call/return overhead would be equivalent; however, that would impact performance in other ways. For example, RISC systems need more registers because they have more values to keep track of—in CISC, intermediate computations (e.g., Address computations) often don't need to be materialized into a register.

In general, to make a Java virtual call to a method, the JIT will generate three instructions: Load, Load, Call. When executing on the Intel ISA, the second load and the call can be combined into a single instruction to transfer control to the address from memory, without putting the address in a register. This not only gets more value from the small register set, but also minimizes code size.

The Intel Architecture also has instructions that efficiently put values on the stack. Intel's "Push" instruction not only stores a value on the stack, but in parallel, it adjusts the stack pointer. The Push instruction is typically 1 byte long, minimizing code size. (More on this later!) It is more efficient for JIT compilers to use the native Intel Architecture stack rather than the Java stack for passing parameters and once the native stack is used, the Push and Pop instructions are quite efficient for passing parameters.

Small Code Size

At the Hot Chips IX Symposium at Stanford in 1997, Sun presented a paper discussing what they would like to see added to the hardware to improve performance. The number one performance booster was large caches, especially the Instruction Cache.[6]

Intel optimized the instruction encoding to minimize code size. (Variable length instructions, 2 address instructions, etc.) This means that more Intel Architecture Instructions can fit into the same amount of instruction cache than RISC instructions. Processors have lengthy stalls (no work gets done) when trying to fetch an instruction that is not in the instruction cache. According to Sun, "on average, each byte of bytecode translates into five bytes of SPARC machine code and four bytes of Intel machine code."[7] Hence, Intel has at least a 25% advantage on code size over their SPARC Architecture:

Code Size Comparison

Bytecode	1	
Sun IA code	4	
Sun SPARC	5	(largest)
Intel IA Code	3.1	

This is analogous to seeing a page by page comparison of something written in English and German. The English text tends to contain fewer and shorter words than the German equivalent. Likewise, the Intel instruction set needs fewer bytes to convey the semantics than a RISC processor.

Branches

Java code has characteristics that are quite different from other languages. In analysis of Integer Spec tests, a branch occurs about every 5 instructions.[8] In Java , it is about every 4 instructions. The Intel Architecture is well suited to take advantage of "branchy" code because setting the condition codes that determine if the branch is to be taken can often be done as a side effect of another instruction. Intel's condition codes are set on many arithmetic instructions and sometimes this eliminates the need for a separate compare instruction. This is at no cost because setting the condition code happens as a side effect

[6] Hoelzle, Bak, Grarup, Griesemer, Mitrovic. p. 50.
[7] Cramer, Friedman, Miller, Seberger, Wilson, Wolczko. "Compiling Java Just in Time". IEEE Micro, May/June 1997. p. 42.
[8] Cmelik, Kong, Ditzel, Kelly. "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks". ASPLOS IV, April, 1991.

in parallel with the instruction's execution. Since Java has more branches, there are more opportunities to eliminate redundant settings of the branch condition.

Consider this Java source code:
```
     While(i != 0)
          {
             //do something
             i--;
          }
```
Just in Time compilers might generate code that decrements I, then compares it with zero, then branches if the result is not zero. With the Intel ISA, the compare with zero can be avoided because the decrement instruction sets the same condition code as the compare.

Multithreading
Java makes it easy for the programmer to create a multithreaded application through the use of the Thread Class. As previously mentioned, the Intel Architecture contains a relative small register set so there are fewer registers to save and restore on context switches. Sun has estimated that when running HotJava, there are typically 30 active threads and they also concluded that more registers is not a benefit in this environment.[9]

Conclusion.
The general purpose nature of the Intel instruction set architecture is a good match for Java. JIT compiled Java code can take advantage of the special characteristics of Intel's chip: small code size, compound instructions, embedded immediate constants, small register set. Bottom line: the Intel family of processors is ready today for outstanding Java performance.

Acknowledgment.
Thanks Julie Wang for all your comments and help, Millind Mittal for architectural insights and Jesse Fang for the JIT work, and numerous others who provided thoughtful commentary.

[9] Splain. "Hardware Acceleration Strategies for Java. Sun Microelectronics Group. Nov 11, 1997.

Bibliography

ASPLOS III.

Cmelik, Kong, Ditzel, Kelly. “An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks”. ASPLOS IV, April, 1991.

Cramer, Friedman, Miller, Seberger, Wilson, Wolczko. “Compiling Java Just in Time”. IEEE Micro, May/June 1997.

Hoelzle, Bak, Grarup, Griesemer, Mitrovic. “Java On Steroids: Sun’s High-Performance Java Implementation”. Hot Chips IX, Stanford, CA. Aug 25-26, 1997.

Muller, Moura, Bellard, Consel. “Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and
Compiled Code”. (To appear) 3rd Conference on Object-Oriented Technologies and Systems (COOTS).

O’Connor, Taremblay. “PicoJava-1: The Java Virtual Machine in Hardware”. IEEE, Mar/April 1997.

Splain. “Hardware Acceleration Strategies for Java. Sun Microelectronics Group. Nov 11, 1997.

www.webfayre.com/pendragon/jpr. Oct, 1997.