PICOJAVA-I: THE JAVA VIRTUAL MACHINE IN HARDWARE

J. Michael O'Connor

Marc Tremblay

Sun Microelectronics

This small, flexible microprocessor core provides performance five to 20 times better than other means of Java execution.

Simple, secure, and small, Java's object-oriented code promotes clean interfaces and software reuse, while its dynamic, distributed nature makes it a natural choice for network applications. Applications written in Java offer the advantage of being more robust, more portable, and more secure than applications written in conventional languages. (Gosling, Joy, and Steele give a thorough description of the Java language.[1])

Applications in Java are compiled to target the Java Virtual Machine.[2] Until now, however, such applications compiled to this virtual machine's instruction set (called Java byte codes) had to run on a processor either through an interpreter or through just-in-time (JIT) compilation. Often, these mechanisms are embedded in an operating system or an Internet browser.

Each of these methods offers advantages and disadvantages. Interpretation is simple, does not require much memory, and is relatively easy to implement on an existing processor. On the other hand, the nature of interpretation involves a time-consuming loop which, combined with the emulation of the Java byte codes, affects performance significantly.

A JIT compiler, while offering significant speedups over an interpreter, consumes much more memory, a precious resource in the embedded market. The compiler itself along with the memory footprint for the compilation may require a few megabytes of storage.

By directly executing the byte codes, a processor can combine the advantages of interpretation and JIT compilation while eliminating their disadvantages. First, because we can tailor the processor to the Java environment, it can deliver much better performance than a processor designed to run C or Fortran applications. For instance, by providing hardware support for garbage collection and thread synchronization not typically found on conventional chips, the processor can further enhance the performance of Java applications. Second, the elimination of a JIT compiler drastically reduces the amount of memory required.

The possibility of designing a processor that could offer these advantages while maintaining other characteristics important in the embedded market is what led Sun Microelectronics to develop picoJava-I.

Our primary goal here is to describe the picoJava-I architecture. To do so, we first describe characteristics of the Java Virtual Machine that are of interest to a processor designer. To illustrate the microarchitecture trade-offs we made for picoJava-I, we also present statistics on the dynamic distribution of byte codes for various Java applications as well as the impact of the Java runtime. Finally, we present the microarchitecture itself and discuss its performance.

Java Virtual Machine: A hardware perspective

Applications written in Java are compiled to an intermediate representation before being sent to a client over the Internet or another network. Any processor and operating system combination that has an implementation of the Java Virtual Machine, either embedded in the operating system or in a browser, can execute these Java applications and produce correct results.

The Java compiler produces a class file that contains the code and static data for the application being compiled. It generates the code based on a complete definition of a hypothetical target machine, the Java Virtual Machine. This virtual machine comprises a specification of a file format for the executable (called a class file), an instruction set (Java byte codes), and other features such

Table 1. Dynamic opcode distributions.

Instruction class	Dynamic frequency (percentage)
Local-variable loads	34.5
Local-variable stores	7.0
Loads from memory	20.2
Stores to memory	4.0
Compute (integer/floating-point)	9.2
Branches	7.9
Calls/returns	7.3
Push constant	6.8
Miscellaneous stack operations	2.1
New objects	0.4
All others	0.6

as threads and garbage collection. The main factors that influenced this design are portability, security, code size, and the ease of writing an interpreter or a JIT compiler for a target processor and operating system.

Unfortunately, features that contribute to making the virtual machine portable, secure, and so on often presented challenges to designing an effective processor implementation. In some cases, we had to design novel structures, while in others, we applied techniques used in typical RISC processors.

Instruction set. The Java Virtual Machine's instruction set defines instructions that operate on several data types. These types include byte, short, integer, long, float, double, char, object, and return address. All opcodes have 8 bits, but are followed by 0 to 4 operand bytes. The specification defines 200 standard opcodes that can appear in valid class files. In addition, the specification describes 25 "quick variations" and three reserved opcodes.

Quick variations of some opcodes exist to support efficient dynamic binding. The specification requires all references to an object's fields or methods to be resolved during runtime at the time of the initial reference. The first time a method or field is referenced, the original instruction resolves which method or field is correct for the current runtime environment. The original instruction also replaces itself with its quick variant. The next time the program encounters this particular instruction, the quick variation skips the time-consuming resolution process, and simply performs the desired operation directly on the already resolved method of field.

Since most of the instructions implicitly operate on the top of the stack, there are no register specifiers. Therefore, Java-based byte codes are relatively small; the dynamic average instruction size is 1.8 bytes.

The instructions fall into several broad categories, Table 1 shows their dynamic frequencies. (We obtained these frequencies from two benchmark programs, discussed later.)

The most common instructions are loads from the stack's local-variables area to the top of the stack; more than one-third of the instructions belong to this category. Loads of data from memory are the next most common group of instructions, with another one out of every five instructions belonging to this class. The other types of instructions include stores to local variables and stores to memory. These operations are much less frequent than their corresponding loads. Compute instructions such as integer ALU operations and floating-point operations constitute 9.2% of the dynamic instruction count, while branches are 7.9% of the instruction mix. Method calls and returns make up 7.3% of the instructions. Finally, the last category of instructions that has a significant dynamic frequency are pushes of simple constants onto the stack; these operations are 6.8% of all instructions.

Miscellaneous stack operations such as popping data off the stack and duplicating stack values only make up around 2% of the total instructions. Other instructions exist, such as those to create new objects or synchronize with other threads; however these occur less than 0.5% of the time.

Many instructions are associated with data type information. In fact, many instructions that nominally operate on different data types have identical functions. For instance, iload and fload both copy a 32-bit value from a local variable onto the top of the stack. The only difference is that subsequent instructions treat the data moved by iload as an integer, but treat the data moved by fload as a single-precision floating-point value. The reason different instructions with the same function exist is to facilitate static type checking of Java code by the byte code verifier software prior to execution. Internal to an implementation of the virtual machine, these two instructions can use the same execution mechanism. Thus, hardware must support a smaller total number of distinct operations than the number of different opcodes suggests.

Stack architecture. The Java Virtual Machine architecture is stack based; all operations on data occur through the stack. The machine pushes data from the constant pool and from local variables onto the stack, where instructions implicitly get their operands. Both integer and floating-point instructions take their operands from the same stack.

The stack functions as a repository of information for method calls as well as for working data for expression evaluation. Each method invocation creates a call frame on the stack at execution time. The frame contains the parameters for the method and local variables. The frame also includes the frame state, which documents pertinent information needed when returning after the method is completed, such as the program counter and any monitor entered.

Java programs typically contain a high percentage of method calls, so optimizing method invocation can substantially improve the performance of Java code. The stack structure optimizes parameter passing from a caller to a method. We designed the method calls to allow overlap between the methods, enabling direct parameter passing without requiring copying of the parameters. The machine pushes parameter values onto the top of the operand stack where they become part of the local-variables area of the called method. By passing parameters through the operand stack, the virtual machine avoids the explicit register-spilling and -filling operations found in conventional architectures.

As mentioned earlier, the stack contains the parameters and local variables for each method. Access to this area is available through the local-variable load and store instructions defined by the Java Virtual Machine. Recall that, according to dynamic instruction counts, local-variable accesses are the most common type of instruction. These instructions do one of two things: They copy a value at some offset from the current method's local-variables area on the stack to the top of the stack (for example, the iload instruction). Or they write the value at the top of the stack into some offset from the current method's local-variables area (for example, the istore instruction). These instructions require any Java Virutal Machine implementation to offer good support for random access into the stack.

Fortunately, on average the start of the local-variables region for the current method is less than 15 elements down from the current top of the stack. This fact allows some optimizations in the implementation.

Multithreading. The Java Virtual Machine provides support for running multiple concurrent threads of execution. Furthermore, thread primitives are an integral part of the Java language, which suggests applications will use threads widely. As a result, Java programs may frequently wish to enter "monitors" associated with objects. A monitor ensures that only one thread of execution can access the associated object at a time. No other thread can access the object until the first thread exits the monitor.

This function allows threads to perform atomic updates to a group of fields in an object without the worry that another thread may try to use the object in an inconsistent state (see Figure 1). For example, if object X were being inserted into the middle of a doubly linked list between objects J and K (Figure 1a), the insertion method would attempt to enter the monitors for all three objects. Once the thread had acquired exclusive access to J, K, and X, it could update J's next pointer, K's previous pointer, and X's next and previous pointers. No other thread using this doubly linked list would ever see a situation with X partially inserted (Figure 1b). Note, however, that methods that traverse the doubly linked list must also enter monitors associated with each object in the list as they move over the list to make sure no updates are taking place.

This example illustrates the fact that for almost any shared object, a thread will have to enter a monitor associated with that object for either reading or updating the object. This is particularly true for some of the standard Java libraries. Since multiple threads could use the standard libraries, the libraries must be "paranoid"—guarding objects with monitor entries even from applications with only one thread.

Java monitors also allow a single execution thread to enter a monitor associated with an object multiple times without releasing the monitor. Because various standard library functions may call other methods in the same library, the same monitor for the same shared resource may be entered in multiple methods. Consequently, each monitor must also keep track of the number of times the current thread has entered it, as well as efficiently allowing multiple entries into the monitors.

Memory management. The virtual machine provides substantial leeway for the memory management implementation. Essentially, it specifies only that any space allocated for a new object is preinitialized to zeros, and that there must be some form of automatic garbage collection.

The Java Virtual Machine does not rely on a programmer to explicitly indicate when the program will no longer use an object (thus allowing the program to reclaim the memory the object occupies). Instead, once an object is no longer referenced by any objects in any thread of execution, a mechanism "collects" this "garbage" object's memory and returns it to the pool of available memory.

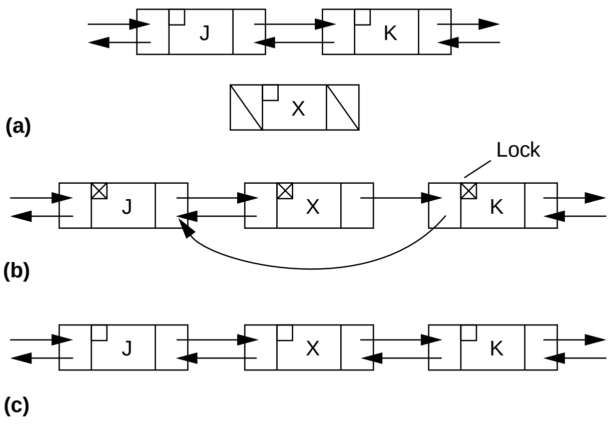

Figure 1. Using "monitors" to maintain consistency among threads: consistent state before insertion—no objects are locked (a); inconsistent state during insertion—all affected objects are locked (b); and consistent state after insertion—no objects are locked (c).

An abundance of literature exists on various garbage collection techniques; Wilson surveys most of these.[3] Many of these algorithms are based on the following model: Periodically a process runs that examines the objects in memory, determines which objects are reachable, and thus determines that all unreached objects have become garbage. Simple algorithms scan all the objects in memory; however, this may require a significant amount of time, introducing unacceptable performance.

Wilson discusses one class of algorithms that aim to improve garbage collection performance: generational collectors.[3] These collectors divide memory into several regions called generations. Generational collectors exploit the observation that recently allocated objects are the most likely to become garbage within a short period of time. Those objects that survive for some period of time progress into successively older generations. To minimize the number of objects examined during each period, these algorithms typically collect only the youngest generation. Since young objects often become garbage at the highest rate, restricting a collection to this area is still very effective at keeping plenty of memory available for new object allocation.

For simplicity, consider an example with only two generations, young and old. The young generation is typically only a small fraction of the total memory, while the old generation occupies the remainder. Since the young generation can contain only a small subset of all the objects in the system, the garbage collector must only scan a fraction of all the objects in the system to determine which objects in the young generation are garbage. Only references from old-generation objects can serve as starting points in the algorithm's examination of the young generation for still-active objects.

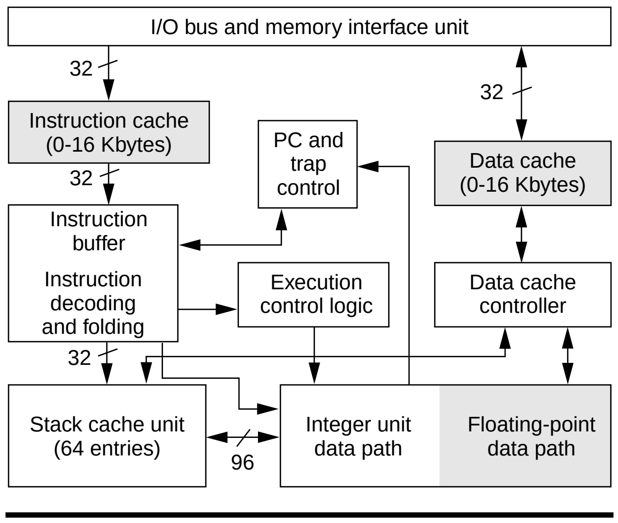

Figure 2. Block diagram of picoJava-I core. Shading indicates configurability.

Thus, between collections, the algorithm must actively update a list of all the pointers from old-generation to young-generation objects. Otherwise, it would have to scan all the old-generation objects to find the current pointers into the young generation.

Tracking these intergenerational pointers can be difficult, however. One technique for logging updates to these pointers uses "write barriers." This technique is based on the fact that any store of a pointer into the old generation might be an intergenerational pointer. Thus, a write barrier intercepts any pointer stores into the old generation and examines them to determine if they must be logged as intergenerational pointers.

Many software-based systems accomplish this write barrier with a few extra instructions around every store to perform the necessary checks. This scheme, however, reduces performance by introducing overhead near every pointer store. Alternatively, some memory management units can be configured to cause an exception on stores to certain regions of memory. The exception handler could then perform the appropriate checks. Unfortunately, since conventional MMUs cannot distinguish between stores of pointer values and stores of other data, they perform many more write barrier checks than are actually needed. Because the Java Virtual Machine instruction set contains type information, a processor implementing the virtual machine can avoid write barrier checks in many cases of stores of nonpointer data. This would improve the performance of a system implementing a generational garbage collection scheme.

picoJava-I microarchitecture

picoJava-I is a small, configurable core designed to support the Java Virtual Machine specification with excellent price-performance (see Figure 2). This core architecture gives designers maximum flexibility when building a Java-based application. Depending on cost and performance goals, the designer can choose the appropriate I/O and functional blocks for the target environment. Moreover, picoJava-I allows variable-size instruction and data caches and the option of including or excluding a floating-point unit. This means that developers of cost-sensitive applications can save die area by taking advantage of configurable features.

Extended instruction set. The picoJava-I core includes a RISC-style pipeline and a straightforward instruction set. We implemented in hardware only those instructions that directly improve Java execution. Most instructions execute in one to three cycles. For example, integer addition and quick loads of object fields fall into this category. Of the instructions not implemented directly in hardware, those deemed critical for system performance execute through microcode or state machines. The core traps and emulates the few remaining instructions.

For instance, invoking a method is a common, performance-critical operation, but it is fairly complex. picoJava-I handles this instruction category with microcode. Creating a new object is less common, and far more complex than invoking a method. Thus, picoJava-I traps and emulates the new instruction. This instruction execution hierarchy leaves complex and infrequent operations to software to keep the complexity and size of the core manageable.

In addition to the instruction set defined by the Java Virtual Machine, picoJava-I implements some extended instructions in the reserved opcode space set aside in the specification. These instructions all have 2-byte opcodes, the first byte being one of the reserved virtual machine opcode bytes. We included these extended instructions to allow programmers to write system-level code. The Java Virtual Machine typically relies on library calls to the underlying operating system to perform certain functions; picoJava-I must provide instructions to allow those functions. The extended byte codes fall into four major classes: arbitrary load/store, cache management, internal register access, and miscellaneous.

The arbitrary load and store instructions allow access to arbitrary memory locations and permit noncacheable, little-endian, and/or sign-extended loads and stores of 1-, 2-, or 4-byte values. Applications typically use these load and store instructions to communicate with memory-mapped I/O devices. The cache management instructions support cache flushing for coherency reasons. The internal register access instructions allow the internal state of the chip to be saved, restored, or modified to permit such services as context switching. Finally, the miscellaneous instructions include a power-down instruction, diagnostic accesses to the caches, and various others.

The virtual machine specification generally specifies the input and output data types for each instruction. Some of the quick variations, however, only specify the size of the input or output data items. For instance, putfield_quick only specifies that the data item to be stored in an object field is one word in size. It may be an integer, a single-precision floating-point instruction, or a pointer. Therefore, we added a couple of instructions to the picoJava-I instruction set to cover this case, in which type information is ambiguous. The picoJava-I core uses one of these new instructions, aputfield_quick, when writing a pointer into an object field. As a result of these new instructions, the picoJava-I core can always detect when a pointer is written to memory. This means that garbage collection checks for intergenerational pointers (described earlier) occur only when strictly necessary.

These extended instructions are part of the picoJava-I native instruction set and cannot be contained in Java class files sent over the Net. The Java byte code verifier enforces this requirement.

Front end. An instruction cache that can range from 0 to 16 Kbytes in size stores the Java byte codes. This direct-mapped cache has a line size of 8 bytes, which is small compared to other RISC processors. But since the virtual machine's instructions are shorter on average, the end result is approximately the same.

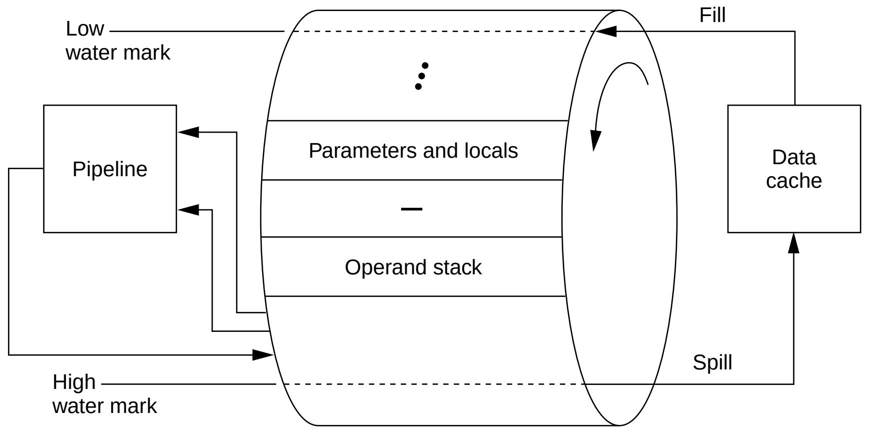

Figure 3. picoJava-I stack cache.

A 12-byte instruction buffer decouples the instruction cache from the rest of the pipeline. The processor can write a maximum of 4 bytes into the buffer at one time, whereas it can read out 5 bytes at once. Since most instructions are 1.8 bytes, the processor can read more than one instruction in a single cycle. All instructions consist of an 8-bit opcode and 0 or more operand bytes as defined in the Java Virtual Machine specification. The processor can decode up to 5 bytes at the head of the instruction buffer and send them to the next pipeline stage for execution in a single cycle. The 5 bytes that the processor can read out correspond to the largest single instruction or to a pair of smaller instructions that can be folded together. (We discuss folding later.)

The picoJava-I core does not have branch prediction logic; it simply predicts every branch as not taken. Because picoJava-I core's pipeline is short (four stages, discussed later), there is a penalty of only two cycles when a branch is taken. The additional complexity of supporting a branch prediction scheme would improve performance by only a few percentage points. Since picoJava-I's goals are good price-performance and low power consumption, Sun deemed the area and power impact of branch prediction logic too expensive.

Furthermore, in picoJava-I, many of the control transfers in Java programs are method invocations implemented in microcode. The implementation of the microcode exposes the details of the picoJava-I pipeline and allows the core to hide the taken-branch penalty completely. It does this by identifying the target PC to the instruction cache well before the microcode finishes updating the call frame on the stack. Thus, many microcode operations can take place in the branch's delay slot. This branch delay slot is visible only to the microcode—not to the application programs.

Stack cache. The picoJava-I core implements a hardware stack directly supporting the Java Virtual Machine's stack-based architecture. The core caches the stack's top entries in its 64-entry on-chip stack cache (see Figure 3). The stack cache is implemented as a register file and managed as a circular buffer. As elements are pushed onto the stack, the pointer to the top of the stack in the stack cache is decremented (the stack grows downwards). As elements pop off the stack, the top-of-stack pointer increments. If too many pushes or pops take place, the top-of-stack pointer wraps around.

If the stack cache were to wrap around when too many pushes take place, it might overwrite valid data. Similarly, as popping takes place, it is possible that the stack cache might contain no valid data. Therefore, the core uses a technique called dribbling. When the stack cache is almost full, it writes the oldest entries to the data cache, thus making available room for further growth. Similarly, when the number of valid entries gets too low, the stack cache reads back in the entries it has previously scrubbed out until there are enough valid entries. The points at which the dribbler decides to spill or fill entries depends on high and low water marks set in a control register.

An example is the best way to illustrate this mechanism's effectiveness. A certain Java application could call several methods in a row and thereby exceed the stack cache capacity. In the background, the dribbling mechanism would create space on the stack and store the oldest entries in the data cache. As the most recently executed methods completed, the mechanism would preload older stack entries from previous call frames onto the stack from the data cache well in advance of their use. The very predictable behavior of the stack when expanding and contracting in size allows this dribbling implementation to minimize pipeline stalls due to overflows and underflows.

Folding. picoJava-I also accelerates Java byte code execution with a folding operation (see Figure 4, next page).

In most stack implementations, stack operations require several steps, adversely affecting the throughput of instruction execution. These implementations typically access operands from the stack and put the result back on the stack. Furthermore, conventional stack architectures limit access to the top portion of the stack. As a result, when variables are not available on the top of the stack but are needed for an operation, they must be copied from the stack's current local-

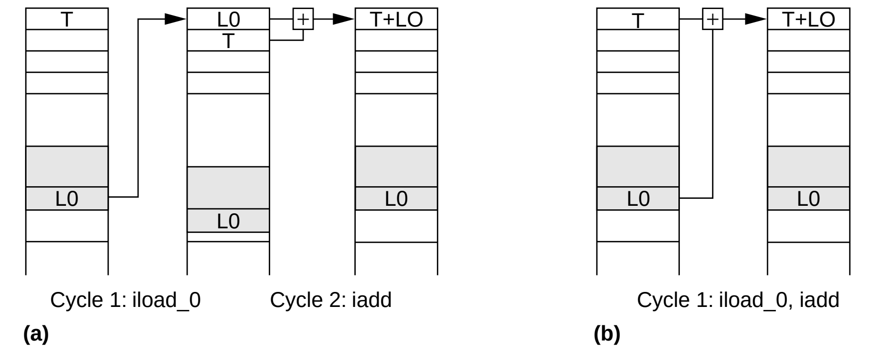

Figure 4. Example of folding: Without folding, the processor executes iload_0 during the first cycle and iadd during the second cycle (a). With folding, iload_0 and iadd execute in the same cycle (b).

Table 2. Opcode frequencies with folding.

Instruction class	Dynamic frequency before folding (percentage)	Dynamic frequency after folding (percentage)	Instructions folded (percentage)
Local-variable loads	34.5	24.4	10.1
Local-variable stores	7.0	7.0	0
Loads from memory	20.2	20.2	0
Stores to memory	4.0	4.0	0
Compute (integer/ floating-point)	9.2	9.2	0
Branches	7.9	7.9	0
Calls/returns	7.3	7.3	0
Push constant	6.8	2.0	4.8
Miscellaneous stack operations	2.1	2.1	0
New objects	0.4	0.4	0
All others	0.6	0.6	0
Total	100.0	85.1	14.9

variables area to the top of the stack. Eliminating this extra step could dramatically improve the performance of the typical stack architecture.

To boost performance, picoJava-I relies on a folding operation that takes advantage of random, single-cycle access to the stack cache. Frequently, an instruction that copies data from a local variable to the top of the stack immediately precedes an instruction that consumes that data. The instruction decoder detects this situation and folds these two instructions together. This compound instruction performs the operation as if the local variable were already located at the top of the stack. Since, on average, the variables area is within 15 entries of the top of the stack and the stack cache is designed to contain nearly 64 valid entries, the local variable requested is almost always in the stack cache. In the unlikely event that the local variable is not contained on the stack cache, folding cannot occur, and picoJava-I suppresses it. picoJava-I also suppresses folding for single-step debugging purposes.

A further enhancement of the folding technique allows the core to fold pushes of simple constants, which constitute almost 7% of the dynamic instruction frequency. Because the processor can determine the constant early in the decoding stage, the constant can go directly to the subsequent instruction without requiring a stack update.

Simulations of the same benchmarks used for the data in Table 1 show that folding eliminates approximately 15% of the total dynamic instruction count. Once we enabled folding, we made a composite of instruction distribution for several key benchmark programs (Table 2). The resulting instruction distribution shows reduced overhead for local-variable access and constant pushing.

Data cache. The data cache, like the instruction cache, can range in size from 0 to 16 Kbytes. To improve the hit rate and performance, it is a two-way, set-associative, write-back cache. The data path between the picoJava-I data cache and the pipeline is 32 bits wide.

The data cache also supports a line_zero instruction, which sets a line as valid and dirty and sets all the data in the line to zero. This instruction is very helpful in reducing the bus traffic required for initializing new objects.[4] The virtual machine requires all newly allocated objects to be initialized to zero. Without the line_zero instruction, the code must waste bandwidth writing out zeroes one word at a time, creating a data cache line that will be completely overwritten with zeroes to be read into the cache. The line_zero instruction simply creates a fully zeroed cache line in a few cycles without causing an unnecessary cache line fill from memory.

Floating-point unit. The Java Virtual Machine supports both single- and double-precision floating-point operations. It dictates that all floating-point arithmetic comply with IEEE Std 754, including full support for denormalized values and gradual underflow. The only rounding mode supported is the round-to-nearest mode for floating-point results and round-towards-zero for floating-point conversions to integer results.

To enable lower cost designs, we made the floating-point unit easily removable from the picoJava-I core. If the floating-point unit is not present, each floating-point instruction traps to a software routine that may emulate the instruction using available integer instructions. When present, the floating-point unit supports the virtual machine's specification for floating-point arithmetic. In particular, the floating-point unit directly supports both single- and double-precision computations, denormalized values, and gradual underflow. It also performs round-to-nearest and round-towards-zero when appropriate.

The floating-point unit executes only one instruction at a time. Since a stack-based architecture typically results in code in which every instruction depends on the result of the previous instruction, there is little benefit to overlapping different stages of floating-point execution from several different instructions. This feature allows a fairly compact floating-point unit that still has good latency characteristics. For instance, most single-precision addition and multiplication operations require only three cycles.

Memory and I/O controller interface. The picoJava-I core can easily interfaced with various memory controllers (SDRAM, EDO, SRAM, DRAM, flash, and so on) and various I/O controllers (PCI, USB, PCMCIA, and so on). To achieve this flexibility without loss in memory latency, we designed an interface from the core to a virtual memory controller.

The picoJava-I memory bus is a 32 bits wide, optimized for 32-bit transfers. Designers can apply simple logic and buffering outside the core to support narrower or wider bus widths. One exception is that during boot up, an external signal can control the instruction fetch width. This signal allows the system designer to interface the picoJava-I core to an 8-bit boot PROM interface. Asserting this signal at boot time causes the instruction fetches to be in sizes of single bytes.

Simple four-stage pipeline. The pipeline in picoJava-I has four stages based on the fundamental paths needed for execution, similar to other RISC pipelines (see Figure 5). The core fetches instructions from the instruction cache into an instruction buffer in the fetch stage. It decodes the instructions at the head of the instruction buffer, folds them, and accesses the stack cache in the decode stage. The instruction takes one or more cycles to execute (accessing the data cache as necessary) in the execute stage. Finally, the core writes the result back to the stack in the write-back stage. There is full bypassing, so the execute stage does not need to wait for the write-back stage to complete before using the result of the previous computation.

In a stack-based architecture, an instruction almost always depends on the result of the previous instruction. Consequently, one operation at a time occupies the execute stage. Unlike most RISC processors, picoJava-I does not overlap access to the data cache by one instruction (or a folded pair of instructions) with the execution of the next instruction (or folded pair). This is because in all likelihood the result of the data cache access is required as an input for the execution of the next instruction.

Monitor support. The speed of monitor entry operations plays an important role in overall system performance. This motivation resulted in a simple mechanism that speeds the common case, while efficiently supporting the full range of possible situations for monitor entry.

One key element of the monitor support is simply reserving the low-order 2 bits in each object header. These are called LOCK and WANT. All accesses to the object header except by the monitor-enter and monitor-exit operations have these bits masked off by hardware, allowing the core to treat the object header as a word-aligned pointer. The LOCK and WANT bit scheme allows a monitor entry operation to merely check the state of the LOCK bit, setting it if it is not set. When another thread holds that object's monitor,

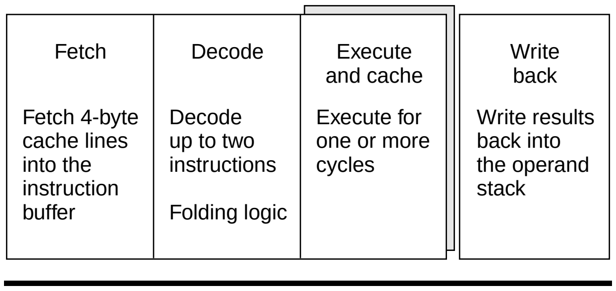
Fetch
Fetch 4-byte cache lines into the instruction buffer
Decode
Decode up to two instructions
Folding logic
Execute and cache
Execute for one or more cycles
Write back
Write results back into the operand stack

Figure 5. picoJava-I pipeline.

a trap is signaled, and the operating system sets the WANT bit, tells the thread to wait, and marks the thread as waiting on that monitor.

When a thread exits a monitor, the operating system clears the LOCK bit and checks the WANT bit. If the WANT bit is set, that indicates that another thread is waiting for that object, and a trap to the operating system is generated. The operating system identifies and informs the waiting thread that it can acquire the monitor.

In addition to efficiently supporting the initial entry into a monitor, we have accelerated a thread's reentry of a monitor before exit as well. The operating system maintains a list of the monitors each thread has entered, and the picoJava-I core contains a two-entry cache of the two most recent monitors that the current thread has entered. Associated with each of these cache entries is a counter that indicates how many times the thread has entered the monitor. On each monitor entry, this cache is associatively examined. If the requested monitor is in the cache and, thus, has already been entered, the entry count for that monitor is simply incremented in the hardware. A monitor exit decrements the counter in hardware. If the counter reaches zero, the monitor is exited completely. Maintaining a small cache listing the monitors that a thread currently holds greatly accelerates monitor reentry over a pure software solution. A software mechanism would have to search some data structure for the held monitor before it could adjust the entry count of the monitor.

To reduce complexity, the core manages the monitor cache completely in software. Traps occur whenever a monitor cache miss is detected. The trap code determines the placement policy in the cache and handles evicting and adding new entries. The resulting hardware structures are very small and simple, yet can greatly speed Java execution.

Performance

Clearly, this approach of implementing the Java Virtual Machine in hardware has merit only if the resulting processor has performance advantages over alternative methods of executing Java code. This section describes the resulting performance advantage for this approach.

Selecting Java benchmarks. We specifically designed the picoJava-I architecture to run complex object-oriented Java code. Most existing Java benchmark suites do not offer these characteristics. Therefore, the challenge was to find applications that would give a good representation of real-world Java code.

Table 3. Scaled times (to 100 MHz).

		Benchmark performance (s)	
Method	System	Javac	Raytracer
Native	picoJava-I	1.8	13.0
JIT	Pentium	9.3	64.5
	486	10.7	109.5
Interpreter	Pentium	20.4	174.3
	486	27.3	254.8

Table 4. Resulting speedups compared to the slowest system.

		Speedup factor	
Method	System	Javac	Raytracer
Native	picoJava-I	15.2	19.6
JIT	Pentium	2.9	3.9
	486	2.6	2.3
Interpreter	Pentium	1.3	1.5
	486	1.0	1.0

We considered several programs for suitability but did not select them. For example, we chose not to use a benchmark called Pentominos[5] because it is far too simplistic. This tiny 16-line C program, when translated into Java, does not accurately reflect the sophisticated applications that Java can create. For similar reasons, we did not include other microbenchmarks like CaffeineMarks,[6] since they do not reflect the characteristics of actual Java code.

The two Java programs we did choose as benchmarks for the analysis provide the advanced programming activity likely to be found in a typical Java application. The programs include a high percentage of method calls and returns—common occurrences in object-oriented code—and they create objects at a reasonable rate, similar to real-life applications. The two benchmarks are

- *Javac*—Compilers typically represent a large block of complex code. This Java compiler, from Sun Microsystems JDK 1.0.2, is an object-oriented program, with over 25,000 lines of Java source code in 170 different classes. It has a Java byte code size of approximately 422 Kbytes.
- *Raytracer*—This program, based on work resulting from a Stanford University class project, represents a more traditional, scientific benchmark with substantial floating-point activity. The Raytracer program generates a 100×100-pixel image of a 1,400-triangle dinosaur standing on a glossy table. This is a heavily object-oriented, 3,500-line Java program containing 32 different classes. Its byte code size is 36 Kbytes.

Test setup and methodology. To prove that picoJava-I delivers the expected performance gains, we simulated these benchmarks on the picoJava-I core. Specifically, we compared Java code running in native state on the picoJava-I core with the same code executed on Intel's 486 and Pentium processors, using both a JIT compiler and an interpreter. We used a simulator to emulate the performance of the picoJava-I core. We scaled all the measurements to the same clock rate—100 MHz—and normalized the results to the slowest running processor.

We chose the Intel processors as competitive benchmark systems primarily because both the platforms themselves and tuned Java implementations on those platforms were readily available.

Our choice of clock frequency was arbitrary. We intend the picoJava-I core for use across a broad range of applications, from thin-client network computers to low-cost, embedded applications. We expect that picoJava-I will support clock frequencies comparable to most other microprocessors implemented in similar semiconductor process technology. We chose 100 MHz for the sake of comparison; it does not reflect the maximum clock frequency expected for chips based on the picoJava-I core.

The systems we used for the benchmarks were configured as follows. The first was a 33-MHz Dell 433/ME System 80486, with 16 Mbytes of RAM, a 256-Kbyte external cache, and the Windows 95 operating system. The interpreter was Sun Microsystems' JDK 1.0.2 for Windows 95/NT, and the JIT compiler was the Symantec Cafe 1.5 for Windows 95/NT.

The second system was a Hewlett-Packard Vectra VL 5/166 Series 4 with a 166-MHz Pentium processor, 32 Mbytes of RAM, a 256-Kbyte external cache, and Windows 95.

The picoJava-I environment consisted of a 100-MHz picoJava-I functional simulator that included a 4-Kbyte direct-mapped instruction cache, an 8-Kbyte, two-way set-associative data cache, no external cache, a floating-point unit, and 120-ns latency to DRAM.

We ran the benchmark programs on the three systems with the times scaled to 100 MHz. In other words, we multiplied the execution times for the 486 by 0.33 and the times for the Pentium system by 1.66. The picoJava-I simulator output was already configured for a 100-MHz system. Since the picoJava-I simulator does not accurately simulate I/O, we added a 0.8-second penalty for the Javac benchmark for I/O. For the Raytracer benchmark, we added 0.4 seconds to the results for I/O. We based these values on experiments that measured the I/O time for these benchmarks on a Sun Sparc system running the Java interpreter.

We also minimized the effects of garbage collection by sizing the amount of memory allocated by Java for the program. If we allocate a large amount of memory for the benchmarks, the core never invokes garbage collection. However, because the 486 system had only 16 Mbytes of RAM, sufficient RAM could not be allocated for the program, so a small amount of garbage collection was required.

Results. Table 3 lists the scaled runtime results from the benchmarks. Table 4 illustrates how much faster in orders of magnitude the picoJava-I executes the benchmark code compared with the Java code running on the other processors. For example, picoJava-I is 15 to 20 times faster than a 486 with an interpreter at an equal clock rate, and still five times faster than a Pentium with a JIT compiler at an equal clock rate.

With significant performance gains, the benchmark results clearly demonstrate the advantage of directly executing Java applications on the picoJava-I core. This large performance advantage gives an embedded design team a substantial degree of flexibility when designing an embedded application around this core.

For example, if minimal power consumption is paramount, the designers might run the core 10 times slower than a competitive solution to reduce power requirements. Another possibility is that picoJava-I could be implemented in a slower, less-expensive technology to minimize manufacturing costs. Also, its small footprint and low power requirements make it easy to integrate a full solution on a single chip. The performance gains will vary depending on the application characteristics, the size of the caches, and whether or not the floating-point unit is included.

OVERALL, WE DESIGNED this architecture to be simple and flexible and to support a broad range of possible applications—from low cost to high performance. With the configurable instruction and data caches and the option to include or exclude the floating-point unit, designers can customize the core processor to meet their specific area, performance, and power requirements. At the same time, they can maintain cost, performance, and/or power advantages over competitive Java Virtual Machine execution solutions. Sun Microelectronics is also working on several future Java core designs. These will target a wide range of price and performance points—from the very low end to the very high end.

Acknowledgments

We acknowledge the work and helpful influence of Bill Joy. In addition, we thank members of the management team—specifically Shrenik Mehta and Robert Garner—and the people at JavaSoft for their help. Key members of the picoJava-I design team at Sun are R. Kunchitham, S. Vishin, S. Koppala, R. Buchamwandla, H. Donahue, F. Chao, T. Chen, F. Younas, B. Ho, G. Chu, C. Lee, and C. Chen.

References

1. J. Gosling, B. Joy, and G. Steele, *The Java^{TM} Language Specification,* Addison-Wesley, Reading, Mass., 1996.
2. T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification,* Addison-Wesley, 1996.
3. P.R. Wilson, "Garbage Collection," (In revision, to appear) *Computing Surveys,* 1996. Available via anonymous FTP from ftp.cs.utexas.edu as pub/garbage/bigsurv.ps.
4. C.J. Peng and G.S. Sohi, "Cache Memory Design Considerations to Support Languages with Dynamic Heap Allocation," Tech. Report 860, Computer Sciences Dept., Univ. of Wisconsin, Madison, July 1989.
5. B. Case, "Java Performance Advancing Rapidly," *Microprocessor Report,* Vol. 10, No. 7, May 27, 1996.
6. Pendragon Software, CaffeineMark home page, http://www.webfayre.com/cm.html.

J. Michael O'Connor is a microprocessor architect in Sun Microelectronics' Volume Products Group, where he participated in the conception and design of the picoJava-I core and is involved in the development of future Java processors. His research interests include computer architecture, hardware-software codesign, and high-performance multimedia.

O'Connor holds a BSEE from Rice University and an MSEE from the University of Texas at Austin. He is a member of the IEEE Computer Society.

Marc Tremblay is a distinguished engineer involved in the architecture of high-performance processors at Sun Microelectronics. His current work involves designing a new generation of microprocessors tailored to the Java computing environment and to processing new-media applications. Prior to working on the architecture for picoJava and UltraJava, he was coarchitect for Sun's UltraSparc I and II microprocessors.

Tremblay holds an MS and a PhD in computer science from UCLA, and a BS in physics engineering from Laval University in Canada. He is a member of the IEEE Computer Society.

Address questions concerning this article to J. Michael O'Connor, Sun Microsystems, M/S USUN02-301, 2550 Garcia Ave., Mountain View, CA 94043; mike.oconnor@eng.sun.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 165 Medium 166 High 167