PicoJava: A Direct Execution Engine For Java Bytecode

A small, flexible microprocessor core, picoJava directly executes Java bytecode instructions and provides hardware support for other essential functions of the Java virtual machine. To this end, it introduces a new instruction set architecture based on Java bytecodes, a stack cache to eliminate inefficiencies typically associated with stack-based instruction processing, and hardware support for sophisticated garbage collection algorithms.

Key to the central promise inherent in Java technology—"write once, run anywhere"—is the fact that Java programs run on the Java virtual machine, insulating them from any contact with the underlying hardware. Consequently, Java programs must execute indirectly through a translation layer built into the Java virtual machine. This translator can take different forms—from the simplest interpreter to the most sophisticated just-in-time (JIT) compiler—but its essential function remains the same: It converts Java virtual machine instructions (called bytecodes) into corresponding machine-specific binary instructions intelligible to the underlying CPU.

The primary advantage of bytecode is the ability to create a single image of a program that will execute identically (in principle) on any system equipped with a Java virtual machine. However, this new instruction set has two important secondary benefits as well.

First, a bytecode image of a program, unlike compiled machine binaries, is secure. Indeed, bytecode programs undergo a formal verification process by the Java virtual machine. Verification establishes that bytecode programs are legal and well-behaved before they begin execution. This is a vital step in the overall Java security model, making it practical to download and run programs across a network without fear that the code will cause the system to crash, lock up, or that it will destroy, corrupt, or steal data. Thus Java technology supports the fat-server/thin-client model of network computing, in which code is created and maintained at one location, centralized for purposes of storage and administration at another location, and distributed on demand to still other locations for local execution.

The other noteworthy property of a bytecode program image is its density or the number of bytes needed to encode a program. Measurements of the same program written in C++ and the Java language, with the former compiled to machine code and the latter to bytecode, typically show the bytecode image to be less than half the size of the machine code image. (In some cases, the size difference can be much greater.) This not only reduces the cost of storing bytecode, but in any networked computing model, it also effectively increases the available bandwidth. Especially in wireless and other low-bandwidth environments, high code density is a considerable advantage.
TRANSLATING BETWEEN BYTECODE AND MACHINE CODE

Severing the intimate link between the executable image of a program and the platform it runs on, however, complicates the execution profile of bytecode programs. Performance depends not only on the speed of the underlying platform, but also on the effectiveness of the dynamic translation technology—whether interpreter or compiler—used to convert bytecode instructions into native machine code instructions. For example, the first Java virtual machine supported a very simple (and correspondingly ineffective) interpreter that occupied only about 45 kilobytes of memory (ROM), and that required very little in the way of working space (RAM) to operate. Today, a highly sophisticated (and very effective) dynamic compiler will occupy, at a minimum, hundreds of kilobytes of ROM and require many more megabytes of RAM to operate. Running such a large and sophisticated program requires a system of corresponding complexity and power. Java processors were developed specifically to address the limitations inherent in this performance gradient. That is, they are intended to provide the same high performance bytecode execution possible with sophisticated dynamic compilers, but in small-footprint devices that otherwise are addressable only by simple interpreters.

WHAT IS A JAVA PROCESSOR?

Java processors are CPUs that have been designed to execute Java bytecode instructions directly in hardware. Thus, they entirely bypass the need for dynamic translation and reestablish a simple, direct execution model for Java code, in which code performance scales strictly with the performance of the underlying platform. The result is to make Java code more usable in a wide variety of embedded applications, including special-function Web browsers, set-top boxes, smart phones, mobile phones, PDAs and other handheld devices, automotive systems, smart controllers, smart cards, and so on.

The only other way to restore a direct execution model for Java programs is to directly compile programs written in the Java language all the way down to platform-specific machine binaries (rather than to bytecode). This option, however, succeeds in eliminating the overhead of indirect execution by also eliminating all the advantages of the bytecode instruction set: platform-independence, security, and high code density.

In general, Java processors serve to complement rather than supplant dynamic translation. Taken together, the two technologies—hardware direct execution engines and dynamic translators—can be configured to support Java programming all the way from costly, high-performance systems down to inexpensive, embedded devices.

PICOJAVA DESIGN ISSUES

The first step toward the development of a new class of Java processors was the creation of the bytecode execution engine itself, called the picoJava core. The essential block in this core is the integer execution unit. Optional blocks include a compact floating-point unit that supports the Java virtual machine floating-point specification, and separate instruction and data caches that can be sized independently from 0 to 16 Kbytes (in the obvious binary steps). The target for this execution engine is the class file generated by the Java compiler. The class file contains the code and static data for the application, based on a complete definition of the execution environment provided by the Java virtual machine.

Most of the information in this section applies to any implementation of picoJava, although some details may apply specifically to the picoJava II core and not other generations of the architecture.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Total number</th>
<th>1 byte</th>
<th>2 bytes</th>
<th>3 bytes</th>
<th>More than 3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push constant</td>
<td>13</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Load/store local variable</td>
<td>82</td>
<td>40</td>
<td>41</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stack management</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift and logical</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch and compare</td>
<td>27</td>
<td>5</td>
<td>1</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>Exception handling</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array management</td>
<td>20</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Function return</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch-table branches</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Conversion operations</td>
<td>15</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitors</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object field manipulation</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method invocation</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Miscellaneous object handling</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Total number</td>
<td>226</td>
<td>141</td>
<td>45</td>
<td>34</td>
<td>6</td>
</tr>
<tr>
<td>Portion (percentage)</td>
<td>100</td>
<td>62</td>
<td>20</td>
<td>15</td>
<td>3</td>
</tr>
</tbody>
</table>
The Java security model requires memory to look like a black box. Java programmers can create objects and can start and stop using objects, but they never have any clue as to where the virtual machine is keeping objects.

Basic bytecode

The 226 instructions defined for the Java virtual machine can be divided into 15 different functional categories, as listed in Table 1. To promote code density, the instructions have a variable length, with the more frequent instructions encoded at the shorter lengths. Indeed, a majority are just one byte long (62 percent). Instructions that are not one byte long are most often just two bytes (20 percent) or three bytes (15 percent) long. Only six instructions exceed three bytes in length (3 percent). Given the dominance of the shorter instructions, it is not surprising that traces across a variety of representative bytecode programs show the average length of executed instructions is just 1.8 bytes (14 to 15 bits).

Instructions may operate on several data types, including byte, short, integer, long, float, double, char, object and return address. The first byte of an instruction is typically the opcode, while following bytes, if any, are generally operands. Since the instruction set presumes most operands are popped off the top of a stack, with results pushed back on top of the stack, many instructions do not require any following operand bytes (hence the preponderance of single-byte instructions). For example, the single-byte opcode for `iadd` completely defines the integer add operation, without any explicit mention of what is to be added or where to put the result. It is implicit that the adder will pop the top two elements off the stack, add them together, and then push the sum back on top of the stack.

For the designers of the Java virtual machine, a stack-based instruction set offers a second critical advantage. Besides promoting code density by limiting instruction lengths, operating stack-to-stack avoids the need to make any assumptions about CPU registers. This enhances portability of the virtual machine, since register files vary widely among CPUs.

Contrast with RISC. The bytecode instruction set devised for the Java virtual machine contrasts sharply with an instruction set devised for a standard RISC processor. A RISC processor would support many fewer instructions—indeed, RISC processors are sometimes defined as CPUs that have fewer than 100 instructions—with fixed widths of four bytes (32 bits) each. Assuming this typical RISC processor had 32 registers, compute instructions like `iadd` would consist of an opcode several bits long, followed by three five-bit fields specifying two source registers for the operands and a destination register for the result. (The remaining bits in a 32-bit word often are reserved to hold small constants, in the event one of the operands does not come from a register.)

This difference in ISAs between the Java bytecode instruction set and a RISC instruction set reflects an equally sharp divergence in goals. The architects of RISC developed instruction sets from a hardware standpoint to simplify (improve) the process of CPU design. The architects of the Java virtual machine developed the bytecode instruction set from a software standpoint to create small, secure, platform-independent programs.

From the standpoint of building a Java processor, it’s unimportant how RISC-like (or not) the bytecode instruction set is. By definition, bytecodes, whatever their nature, form the starting point for thinking about the design of Java processors. Clearly, a Java processor must execute all 226 bytecode instructions defined for the Java virtual machine.

Intentionally incomplete. Viewed from a hardware perspective, the bytecode instruction set is not merely deliberately different from RISC instruction sets, it also is intentionally incomplete. For example, there are no bytecodes to test or diagnose hardware, to read or write CPU status and control registers, or to manage on-chip caches. Since these sorts of functions depend on the implementation of the underlying hardware, they have no place in an instruction set designed for a portable virtual machine.

Still other instructions are missing, not because they are inappropriate for any portable virtual machine, but because they are inconsistent with design goals set for the Java virtual machine. The Java security model requires memory to look like a black box. Java programmers can create objects and can start and stop using objects, but they never have any clue as to where the virtual machine is keeping objects. Hence, it is impossible to write virus programs that depend on information about the layout of memory in the Java language.

In terms of the bytecode instruction set, this means there are no bytecodes to directly access arbitrary memory locations. Java virtual machine instructions only operate upon object references, which have a deliberately opaque relationship to the object’s physical storage location.

Of course, a Java program can be spared the burden of tracking the location of objects in memory precisely because this burden is assumed by the Java virtual machine. A Java program is able to refer to objects entirely by object reference because, for every object reference, the Java virtual machine is able to determine the location of the referenced object.

Note that this means that the Java virtual machine cannot itself be a Java program. The careful limits of Java programs are possible only because the Java virtual machine is able to determine the location of the referenced object.

Of course, a Java program can be spared the burden of tracking the location of objects in memory precisely because this burden is assumed by the Java virtual machine. A Java program is able to refer to objects entirely by object reference because, for every object reference, the Java virtual machine is able to determine the location of the referenced object.

Note that this means that the Java virtual machine cannot itself be a Java program. The careful limits of Java programs are possible only because the Java virtual machine is able to determine the location of the referenced object.
virtual machine, but all the programs the virtual machine itself relies on: boot code, kernel services, device drivers, interrupt handlers, system libraries, and so on.

Extended bytecode

While the architects of a Java processor cannot delete or modify any instruction defined for the Java virtual machine, they can and must augment these instructions with new bytecodes designed to repair the omissions mentioned earlier. A Java processor is not a portable virtual machine but a particular real machine. It therefore requires hardware diagnostics together with all the other instructions necessary for low-level hardware management, including the ability to load and store to arbitrary memory locations. And, in a real world that contains much code not written in the Java language, even a real machine targeted to process Java programs had best run the non-Java code efficiently, too.

Consequently, the instruction set defined for Java processors includes a new group of extended bytecodes, original with and exclusive to these CPUs. Table 2 describes these 115 additions to the 226 bytecodes defined for the Java virtual machine. Together, the 341 instructions described in Tables 1 and 2 make up the complete picoJava ISA.

The extended bytecodes defined to complete the picoJava ISA leverage two bytecodes (out of the set of 256 possible initial bytecodes) reserved by the designers of the Java virtual machine specifically for implementation-specific extensions to the Java virtual machine instruction set. Most of the extended bytecodes begin with one of these two prefixes. In effect, whenever the picoJava decode logic encounters either of these two bytecodes at the beginning of an instruction, it knows it is dealing with one of the extended bytecodes, and then looks at the second byte of the instruction to figure out what it does.

The picoJava ISA, composed of all the bytecodes defined for the Java virtual machine as well as all the extended bytecodes intentionally omitted by its designers, not only does everything the Java virtual machine can do, but everything the Java virtual machine can't do (but any real machine can do). It can control real hardware. It can access real memory. It can efficiently execute code written in other high-level languages. In short, the picoJava ISA is no longer a Java virtual machine instruction set. Instead, it is simply a complete real machine instruction set.

By definition, no program written in the Java language—and legally compiled down to bytecodes—will contain a single extended bytecode. Conversely, no program that contains even a single extended bytecode can be a legally compiled Java program. The extended bytecodes included in the picoJava ISA exist entirely for the benefit of programs not written in the Java language.

These programs include all the foundation code for Java programs (from the Java virtual machine on down) as well as code for any “legacy” applications written in other languages that must be ported to Java CPUs.

Since only platforms based on picoJava will understand extended bytecodes, no program making use of these instructions will run on any other kind of platform. Since some of the extended bytecodes provide programs with direct access to memory—contravening the Java security model—no program containing extended bytecodes can be regarded as safe. In other words, non-Java code running on a Java processor is just like non-Java code running on any other type of processor, namely both platform specific and not verifiably secure.

Managing complexity

With an ISA of over 300 instructions, the next critical design issue was how to manage the resulting complexity. Remember that the rationale underlying the development of Java CPUs is to facilitate the adoption of Java technology in a variety of embedded and personal Java environments where, because of resource constraints, dynamic translation may be either too slow or too expensive. In keeping with these target markets, it is important that Java CPUs be small and inexpensive. Thus, the need to support a large and complex instruction set posed an immediate challenge. The architects of picoJava solved this problem by dividing the instructions into three categories, according to how difficult they were to implement: simple, moderately complicated, or very complicated.

Simple instructions. The simple instructions are RISC-like in the sense that they are the sort of basic instructions that readily lend themselves to hardware implementation. This group contains the majority of instructions defined for the Java virtual machine, as well as most of the extended bytecodes. In the picoJava core, all the instructions in this group are hardwired and execute in a single clock cycle. The majority of instructions executed by a typical program would fall into this category (for example, all the integer arithmetic operations and quick loads of object fields).

Moderately complicated instructions. The group of moderately complicated instructions, while not executed as frequently as the simple instructions, are by no means rare. This group contains about 30 of the

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Total number</th>
<th>Number of instructions of the given length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostic</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Register reads and writes</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>Arbitrary loads and stores</td>
<td>35</td>
<td>26 9</td>
</tr>
<tr>
<td>Other language support</td>
<td>6</td>
<td>5 1</td>
</tr>
<tr>
<td>System software support</td>
<td>17</td>
<td>2 10 5</td>
</tr>
<tr>
<td>Total number</td>
<td>115</td>
<td>2 98 15</td>
</tr>
<tr>
<td>Portion (percentage)</td>
<td>100</td>
<td>2 85 13</td>
</tr>
</tbody>
</table>
Figure 1. Once values have been loaded into the register file, a register-based RISC machine can perform an add (a) in a single instruction that specifies the source registers for the two values to be added and the destination register for the result. In contrast, a stack-based add (b) can take four instructions. Even after the values are loaded, they still must be moved to the top of the stack by two local variable load instructions before the add instruction issues. Another local variable store instruction stores the result.

Java virtual machine bytecodes and the remaining handful of extended bytecodes. These more CISC-like instructions are implemented in the same way CISC-style processors generally implement complex instructions; that is, using microcode. From the standpoint of hardware, the cost of microcoded instructions is relatively low. A small microcode ROM can contain the sequence of control signals required to execute these instructions. The picoJava core uses two approximately 2-Kbyte ROMs, one in the integer unit and the other in the optional floating-point unit. Microcode offers a good balance between the need to keep the hardware implementation simple and the need for good performance. Some microcoded instructions execute in just a few clock cycles (iload, for example, completes in three cycles), while others require a larger number of cycles to finish (invokestatic_quick takes 11 cycles while invokesuper_quick needs 21 cycles).

Very complicated instructions. The last group of about 30 instructions consists of all the remaining bytecodes defined for the Java virtual machine. These instructions are either very complicated or require services from the underlying operating system, or both. For example, the new instruction, used to create a new object, meets both criteria. It is quite complicated, requiring the class of the new object to be looked up dynamically from the list of classes loaded in the system, in addition to allocating memory space for the object. If the new object’s class file (describing the object being created) has not been loaded yet, the virtual machine must go out to the network or local file system to obtain the class file. The allocation of memory space for the new object also must be coordinated with the underlying operating system, requiring some flexibility in the way the task is implemented.

Based on the fact that instructions in this final category involve significant complexity and/or require some implementation flexibility (due to dependencies on the underlying operating system), they are implemented in software. More specifically, when a program invokes one of these bytecodes, the CPU takes an instruction emulation trap. Depending on which bytecode caused the trap, the exception handler then calls a particular software routine that emulates the trapped instruction by executing an appropriate sequence of hardwired and microcoded instructions.

Obviously this last method for executing instructions is slow, possibly taking hundreds or even thousands of cycles. There is, nonetheless, no competitive disadvantage in this approach. Like picoJava, any interpreter or dynamic compiler must synthesize these very complicated instructions in software by executing a long sequence of more basic operations. The slowdown picoJava experiences when it encounters one of these (happily infrequent) bytecodes is offset by a corresponding slowdown in the operation of any dynamic translator when it encounters the same bytecode. Indeed, if anything, picoJava has the execution advantage here, since its emulation routine for any bytecode in this final category is not only preloaded, but is reached very quickly via a hardware trap.

More efficient stack processing

The next hurdle in the design of picoJava was to deal with the inherent inefficiency of a stack machine. For CPU designers, the fact that a stack-based instruction set promotes small, secure, portable programs is less important than the fact that stack-based processing is slow. A stack machine must spend cycles moving operands to the top of stack—where the compute operations can get at them—and moving results off the top of stack for storage. In contrast, a register machine spends no cycles juggling values into or out of operative position, since a register file provides random access to many different values simultaneously. The advantage is significant. Studies indicate that, to complete the same number of computations as a register machine, stack machines pay an overhead burden of up to 30 percent more operations spent manipulating the stack. Figure 1 shows a simple but stark example of the difference.

Stack register file. Since adopting a register-based instruction set was not an option for picoJava, the problem was to combine a bytecode instruction set with register-like execution efficiency. Not surprisingly, the answer proved to be a register file, but one organized to support stack-based processing. Figure 2 shows the picoJava 64-entry register file, which is used to cache the top 64 entries of the stack. The picoJava core treats the file as a circular buffer, with a pointer to the top of stack. As execution units push new entries on top, the stack grows and the pointer is incremented. As execution units pop entries off the top, the stack shrinks and the pointer is incremented. (Based on long tradition, the picoJava stack is "upside down" in memory, and grows from higher numbered addresses toward lower numbered addresses.)
The circular organization means the bottom of the stack cache is immediately adjacent to the top of the stack. Thus 65 consecutive pushes will overwrite the first element pushed on top of stack (now the last entry in the stack cache).

The register file has three read and two write ports. Compute operations can simultaneously read out two operands and write back one result. Concurrent background spill and fill operations, intended to keep the stack cache consistent with the top entries of the stack in memory, account for the remaining read and write port. (Spill writes entries in a register file out to the data cache, making room for new values, while fill reads entries into a register file from the data cache, providing values for new computations.)

Data from the constant pool, from local variables, or loaded from objects, are first pushed onto the stack; all compute instructions then access their operands from the stack, and push their results back onto the stack. Procedurally, this stack-to-stack method of operation is equivalent to the register-to-register operations typical of RISC machines, which take all compute operands out of and return all results back to their register file(s).

**High- and low-water marks.** Although the stack cache is implemented as a register file, it enjoys a predictability impossible for a random-access register file since stacks admit of only two basic possibilities. Either new entries are being pushed on top, in which case a stack is growing; or current entries are being popped off, in which case it is shrinking. Whether the stack is growing or shrinking, it is possible to put knowledge about its motion to good use.

If the stack continues to grow, the number of valid dirty entries pushed onto the stack (good results returned to the stack cache but not yet saved to memory) eventually will exceed the value of a programmable high-water mark. This will trigger the spill mechanism, which in the background will begin scrubbing out (saving) the valid dirty entries (to the data cache), starting with the oldest. The consequence is that, although the stack cache is limited to a total of 64 entries, the stack may continue to grow indefinitely by continuing to overwrite clean (already saved) values at the bottom of the stack cache. In effect, the background spill mechanism creates the illusion of infinite register space for caching new stack entries.

When the stack is shrinking, a similar fill mechanism works to prevent an underflow condition which, like overflow, would force a stall in normal processing while the stack cache is reloaded with new values. As entries are popped off the top of the stack, the number of valid entries in the stack cache eventually falls below a programmable low-water mark. This triggers a background fill mechanism, which promptly begins to copy entries from the memory stack into the stack cache. Just as the spill mechanism creates the illusion that the stack cache contains an infinite number of free registers, available for writing new values, so the fill mechanism creates the illusion that the stack cache contains an infinite number of full registers, available for reading valid data.

**Instruction folding.** The most important value of the picoJava stack cache, however, is that it provides a powerful solution to the classic problem of access inefficiency in stack machines. Since the stack cache is, in fact, a full random access register file, the picoJava pipeline has immediate access not merely to the top two entries on the stack, but to all 64 entries held in the stack cache. This opens the way to an execution technique called instruction folding.

Look again at the sequence of instructions shown in Figure 1 for a stack add. The two values to be added are likely both already in the stack cache (in the parameters and local variables area of the current method). The problem is that neither happens to be at top of stack. Consequently, the execution unit must spend a cycle moving each to the top of stack. Likewise, the local store instruction does not move the returned sum out of the stack, but merely relocates it from the top back into the local variables section of the current method frame. As long as all of these local movements happen within the top 64 elements of the stack—true in an overwhelming majority of the cases for local motion in the stack—they will occur inside the stack cache. It is thus possible to combine or fold these several serial operations together into a single RISC-style add operation.

Given random access to the top 64 elements of the stack, the local moves no longer serve any useful purpose. The integer add unit can directly access both operands in the local variable area of the current method (using the two read ports into the stack cache). Similarly, it can return the result directly to the local variables area (using the write port into the stack cache).
In general, then, the picoJava core operates on bytecode instructions in the following way. Based on a set of grouping rules, it scans the incoming stream of bytecodes looking for sequences of instructions that can be folded together (combined into a single operation). These sequences can consist of up to four bytecode instructions, in which

- moves of local data to top of stack are immediately followed by compute instructions that consume the data just moved, and/or
- compute operations are immediately followed by local stores of the result just computed.

When the core finds such a sequence of instructions, it synthesizes a register-based RISC-style operation, by taking the operation to be performed from the compute instruction, the source of the operands from the local variable loads, and the destination of the result from the local variable store.

In other words, picoJava combines the “extra” stack manipulation instructions typical in stack architectures—needed to set up and dispose of calculations—with the calculations themselves. This eliminates essentially all of the computational overhead of stack machines, achieving the same sort of single-cycle execution efficiency found in RISC architectures because (under the hood) picoJava is a comparable RISC architecture, complete with a large 64-entry register file and three-operand register-based computations.

**Benchmark results.** Table 3 provides a side-by-side comparison of a generic RISC processor versus picoJava on a code sequence from the Dhrystone 2.1 benchmark. Using a standard three-operand, register-based instruction set, the generic RISC processor is able to execute nine fixed-width four-byte instructions in 10 clock cycles. Using the bytecode instruction set, picoJava requires 18 variable-width instructions to perform exactly the same work. However, since the average bytecode instruction is less than half the length of the fixed-width RISC instructions, the bytecode version of the code sequence is noticeably smaller (28 bytes versus 36). Further, in this particular instance, picoJava’s ability to fold an average of two bytecode instructions together every clock cycle enables it to get through the 18 instructions in only nine cycles—one cycle earlier than the generic RISC CPU.

Measurements on a number of Java applications indicate that between 23 and 37 percent of all instructions executed get folded into other instructions, with an average of around 28 percent. These measurements confirm that picoJava’s folding capabilities eliminate nearly all of the stack inefficiency found by earlier researchers. Folding significantly improves the performance of all types of code that execute on the picoJava core, regardless of its source—whether written directly in assembler, or compiled from C or the Java language.
Hardware for runtime support

The final set of unusual design challenges faced by the picoJava design team was to provide several new hardware services to support the runtime requirements of Java bytecode programs. One of the most important issues in this area is the need to support the Java virtual machine’s memory management function effectively.

As mentioned earlier, although Java programs may create and use (and stop using) objects, by design they cannot directly manage memory. Instead, the Java virtual machine provides automatic memory management services by dynamically allocating memory as needed and freeing up memory no longer in use by running programs. Eliminating the burden of memory management greatly increases programmer productivity and improves program reliability, since errors here of either omission (memory leaks) or commission (dangling pointers) are by far the largest fraction of all programming errors.

Simple garbage collection. To free up memory that is no longer in use by any running program and return it to the system, the virtual machine uses a technology accurately (if inelegantly) termed garbage collection. In one of its simplest forms, called mark-sweep, the garbage collector works by periodically scanning all memory looking for objects that can be reached by any running program. When a reachable object is identified, it is marked. Once all reachable objects have been marked, all of the remaining memory—probably containing a fair number of unreachable objects—is free. The garbage collector then sweeps through memory, collecting all of the free memory and returning it to the free pool. To avoid the problem of memory fragmentation, the scan often ends by compacting memory, packing all the objects still in use together, and organizing all the free space into a single large, contiguous block.

Until garbage collection completes—while dead objects are eliminated, live objects are relocated, and references to objects are updated—the system is in an unstable state. Consequently, garbage collection must run atomically, that is, once it starts, it must run to completion without interruption.

Especially in systems that support large memories, this sort of simple garbage collection scheme can profoundly limit performance. The system periodically will shut down for human-noticeable intervals while the garbage collector runs. Not only does this slow down processing in general, but it also imposes severe restrictions on any sort of real-time processing (since any code that happens to run during a garbage collection interval cannot rely on accessing or updating any objects in the garbage-collected Java heap).

Generational garbage collection. At least a partial solution to these performance issues lies in moving away from this simple garbage collection model to a more sophisticated generational scheme. The idea behind generational garbage collection is to limit memory scans to a small area. The result is that the process takes much less time to complete, improving both overall performance and the system’s real-time capabilities.

The key observation behind this sort of scheme is that most objects are relatively short-lived. Studies show that, in the time it takes to allocate 32 Kbytes of memory off the heap, over 90 percent of the newly created objects already will be dead.

Generational garbage collection leverages this observation by assigning a relatively small part of memory to be used as a nursery, containing all newly created objects. By regularly scanning just this fraction of memory, the garbage collector can eliminate most dead objects from the system. Objects that survive some incubation period, however, tend to be long-lived. To prevent the nursery region from slowly filling up with these more durable objects, they are moved into the rest of memory for long-term storage. A very sophisticated generational scheme might define several, progressively older generations of objects. It then gradually would move objects that survive the aging process from the nursery, through the regions reserved for successively older generations, and finally into the rest of memory.

This sort of scheme does not entirely eliminate the need for an occasional top-to-bottom memory scan, since the number of durable objects still grows with time, and eventually will threaten to consume all memory. When this threat becomes apparent, a scan must be run over the rest of memory holding the durable objects, too. But the need for this more time-consuming scan of the full memory system now would be the exception rather than the rule. Scanning the small nursery region will suffice to eliminate most dead objects, keeping ample memory space free for the creation of new objects. Even more advanced garbage collection strategies, like the train algorithm, allow partitioned, interruptible collection of the oldest generation, thus permitting guaranteed maximum pause times in a garbage-collected environment.

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>RISC instruction stream</th>
<th>picoJava II instruction stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MOV R3,#0</td>
<td>iconst_0 + istore_3</td>
</tr>
<tr>
<td>2</td>
<td>ADD R2,R1,#10</td>
<td>iload_1 + bipush 10 + istore_2</td>
</tr>
<tr>
<td>3</td>
<td>LDR R5,[R0+#d1]</td>
<td>aload_0 + getfield CharGlob</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CMP R5,#65</td>
<td>bishp 65 + if _icmpne L28</td>
</tr>
<tr>
<td>6</td>
<td>BNE L28</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LDR R5,[R0+#d2]</td>
<td>inc 2,255</td>
</tr>
<tr>
<td>8</td>
<td>SUB R2,R2,#-1</td>
<td>iload_2 + isub + istore_1</td>
</tr>
<tr>
<td>9</td>
<td>SUB R1,R6,R2</td>
<td>iconst_1 + istore_3</td>
</tr>
<tr>
<td>10</td>
<td>MOV R3,#1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Comparison of RISC and picoJava II instruction streams that implement the inner loop of the Dhrystone 2.1 benchmark.
The ability of the picoJava core to check all pointer stores in hardware with no software overhead enables programmers to design a very sophisticated garbage collector that supports very short maximum pause times.

**Segmentation.** The key to implementing a generational scheme (or any other incremental scheme, like the train algorithm) is to segment the memory into smaller pieces for quicker scanning and collection. These segments would be the generations in a generational collector or the “cars” in a train algorithm collector.

To quickly scan a portion of the memory to find reachable objects, the garbage collector must maintain a list of all objects in the segment to be scanned that are reachable from outside the segment. Otherwise, the collector would have to scan the rest of memory to find out which outside objects might reference objects inside the segment of interest, preventing them from being deleted as garbage.

The primary mechanism used to maintain this list is a write barrier. The write barrier mechanism allows stores of pointers into objects held in regions outside of a given segment to be efficiently intercepted and examined to see if they refer to objects within the segment of interest (thus, keeping those objects alive). In a generational system, these are often referred to as intergenerational pointers.

The picoJava core provides a flexible method for defining numerous segment boundaries. Once segments are defined, the hardware checks all stores of any pointer to determine if the stored pointer references an object in a different segment. If so, a trap is generated, and the garbage collector can add the reference to the appropriate list (or take other actions if desired).

Other CPU architectures have to make extra software checks on every pointer store to determine if a pointer is intergenerational. This can impose a significant overhead for garbage collectors. By comparison, the ability of the picoJava core to check all pointer stores in hardware with no software overhead enables programmers to design a very sophisticated garbage collector that supports very short maximum pause times.

The picoJava core also provides specialized hardware runtime support to accelerate other vital functions of the Java virtual machine, like thread management and object handling. The bottom line is a substantial performance boost for bytecode programs achieved with a minimal impact on die area, and so with very little increase in either the cost or power consumption of picoJava relative to other CPU cores.

By directly executing bytecode instructions in hardware, which eliminates the need for dynamic translation, the picoJava core technology extends the useful range of Java bytecode programs to embedded environments where the demand for performance is not in keeping with the scarcity of available resources. By the end of 1998, Java processors like Sun’s microJava 701, which will combine high-performance bytecode execution with minimal resource consumption, should be available for evaluation from several licensees of the picoJava core technology.

**Acknowledgments**

The authors acknowledge the work and helpful influences of members of the picoJava and microJava 701 management and engineering teams.

**References**


Harlan McGhan is group manager of technical marketing for the microelectronics division of Sun Microsystems. His background includes extensive experience with high-performance microprocessors going back to the beginning of the 32-bit era, as software engineering manager for National Semiconductor’s Series 32000 line of microprocessors, and later as engineering manager of software applications and analysis for the Intergraph (originally Fairchild) Clipper microprocessor. McGhan has a BA in philosophy from Michigan State University, and an MA and PhD (pending) in the history and logic of science from Princeton University.

Mike O’Connor is the principal architect of Sun’s second-generation picoJava II core and a key member of the team that designed the original picoJava I core. His research interests include computer architecture, hardware-software codesign, and high-performance multimedia. O’Connor has a BSEE from Rice University and an MSEE from the University of Texas, Austin. He is a member of the IEEE Computer Society.

Contact the authors at Sun Microsystems Inc., SJC02-202, 901 San Antonio Road, Palo Alto, CA 94303-4900; harlan.mcghan, mike.oconnor@eng.sun.com.