Australian Computer Science Communications, Vol.21, No.4, 1999, Springer-Verlag Singapore

Parallelism of Java Bytecode Programs and a Java ILP Processor Architecture

Kenji Watanabe and Yamin Li

Graduate School of Computer Science and Engineering
University of Aizu
Aizu-Wakamatsu, 965-8580, Japan
{m5021131, yamin}@u-aizu.ac.jp

Abstract. The Java programming language has been widely used to develop dynamic content in Web pages. The Java Virtual Machine (JVM) executes Java bytecode. For efficient transmission over the Internet, the Java bytecode is a stack oriented architecture: instructions need not contain source and destination specifiers in their bytecodes. The Java bytecodes may be executed on various platforms by interpretation or just in time (JIT) compiling to the native primitives of the particular machine. However, with a few exceptions, Java has not been used for developing network computing applications that demand high performance. In this paper, we investigate the potential parallelism of Java bytecodes and describe a Java processor architecture which aims to exploit instruction level parallelism (ILP). A Java processor can execute bytecodes directly so that we can expect higher performance than from interpretation or JIT compilation. In contrast to other Java processors, our processor exploits ILP: it can execute multiple computational instructions in parallel. Because the Java Virtual Machine stack based, *push* and *pop* instructions are frequent. Usually, the top of the stack will become the bottleneck. We investigate the possibilities of parallel access to the stack and the execution of push/pop with zero time. We have developed an architectural simulator which evaluates the performance of various configurations. The simulation results show that our Java ILP processor could achieve an average 2.96 EIPC (effective instructions per cycle) with a 16-instruction scheduling window and 3.36 EIPC with a 32-instruction window.

1 Introduction

Java[B. and G.,] has been widely accepted by industry and academia because of its portability and ease-of-use, as well as the popularity of the Internet. The Java Virtual Machine (JVM)[Lindholm and Yellin, 1997] is a platform for executing Java bytecode programs. The JVM is “virtual” because it is generally implemented by emulation in software on a “real” machine.

There are three approaches to executing Java bytecodes. The simplest approach is to interpret them: the JVM architecture is realized on a machine and an interpreter interprets bytecodes one by one. Compared to the direct execution of instructions, interpretation will will run at one tenth or even hundredth the speed. Because the JVM is a stack oriented architecture, there many *push* and *pop* instructions which do not perform any computation. The interpreter has to interpret these instructions also. As a result, we cannot expect high performance from this approach.

The second approach translates the Java bytecodes into native RISC primitives. This can be done dynamically by just in time (JIT) compilers. Because the Java bytecodes cannot be executed directly, time for compilation is always needed at run time. Several variants of the JIT concept have been proposed based on when bytecodes are translated[Shieh and Hwu, 1996] [Ebcioglu et al., 1997]. These methods also try to improve performance by removing the push and pop instructions by assigning the locations of variables to suitable RISC registers. In these cases, the stack and its related variables are not needed. We can expect higher performance compared to the first approach. However, because of the differences between the JVM architecture and native architectures, the code generated by the compilers is still not as good as optimized RISC code generated by a C or C++ compiler.

The third approach is to design a Java stack processor to execute bytecodes directly. For example, the Sun PicoJava and MicroJava[WhitePaper,][Turley, 1997] processors can execute one Java bytecode instruction per cycle for most simple instructions. In order to improve performance, these Java processors were designed to recognize certain code sequences that occur frequently and replace them with faster operations (instruction folding). An example computing C=A+B is shown below.

```
iload A  // load A from frame and push it on to stack
iload B  // load B from frame and push it on to stack
iadd     // add the two data at the top of the stack
         // and replace them with the sum
istore C // pop the result from the stack and
         // store it to the destination C in frame
```

Instead of pushing A and B onto the stack, PicoJava 2 (the core of MicroJava) fetches A and B from the method frame and feeds them directly to ALU for computation; similarly, it stores the result generated by ALU directly to the destination. That is, PicoJava 2 processes the four Java instructions just like a RISC instruction: ADD A, B, C.

To keeps costs low, both Java processors did not deal with instruction level parallelism (ILP) issues: they cannot execute multiple computational instructions in parallel in a clock cycle. As network computing gains importance, high performance Java ILP processors will be demanded in the near future.

In this paper, we investigate the potential parallelism of Java bytecode programs and describe a Java processor architecture considering ILP issues. Our Java processor can execute multiple computational bytecode instructions in parallel directly so that we can expect higher performance than attainable with the PicoJava and MicroJava processors, or from interpretation or JIT compilation. Because the Java bytecodes are stack based, the top of the stack will become the bottleneck for performance improvement. We investigate the possibilities of parallel access to the stack and the execution of push and pop in zero time.

We developed an architectural simulator which can evaluate the performance of various processor configurations. The simulation results from Java benchmark programs show that the Java ILP processor could achieve an average 2.96 EIPC (effective instructions per cycle) with a 16-instruction scheduling window or 3.36 EIPC with a 32-instruction window.

The EIPC is calculated by dividing the total number of effective instructions executed by the total number of clock cycles, where the effective instructions are the computational instructions. Effective instructions do not include: unconditional branch, method invocation and return, stack manipulations and local variable instructions that transfer values between local variables and the top of the stack. Since the effective instructions are 30-50% of all instructions, the IPC based on total instructions will be two to three times the EIPC.

Study of the parallelism of Java bytecode programs is helpful not only for tuning the design parameters of a Java ILP processor, but also for developing high performance JIT compilers.

We describe the Java ILP processor architecture next and then give our preliminary measurements of the processor performance.

2 Java ILP Processor Architecture

2.1 Parallel Access to Operand Stack

Our goal is to develop a Java ILP processor architecture which can execute Java computational instructions in parallel. A typical computational instruction pops two operands from the the stack, operates on them, and pushes the result back onto the stack. The push and pop instructions (e.g. iload, istore) do not perform any computation. These can be coalesced with computational instructions resulting in zero cost push and pop.

The Java bytecodes fall into the following categories: (1) *Computational instructions*: The common feature of these instructions is that the operands are popped from the stack. After operation, the results are pushed back onto the stack. (2) *Stack management instructions*: These duplicate, remove and swap data in the stack. (3) *Local variable instructions*: These transfer values between local variables and the operand stack. (4) *Memory reference instructions*: These access fields of objects and array elements which are located in memory. They load fields or array elements and push them onto the stack or store the values from the top of the stack to memory locations. (5) *Branch instructions*: There are two kinds of branch instructions: unconditional and conditional ones. (6) *Method invocation and return instructions*: These are used in method invocations and returns from method. (7) *Others*: The remaining instructions include monitor instructions and exception handling instructions; they can be considered special kinds of branch instructions.

Among the instructions, only the computational ones do useful work. The local variable and stack management instructions appear in the Java instruction set only to support the stack architecture. The memory reference instructions are the equivalent of the load/store instructions in RISC architectures. The method invocation instructions and return instructions are the equivalent of the subroutine call and return instructions in RISC architectures.

In the design of the stack based processor, the key point for performance improvement is to enable the processor to fetch as many operands as possible. This can be done by designing a fast stack cache with multiple read and write ports. We can consider the stack cache as a RISC register file. An example is illustrated in Fig. 1.

```
  Java bytecodes
==================
iload A  // push A
iload B  // push B
iadd     // C=A+B
istore C // pop C
iload D  // push D
iload E  // push E
isub     // F=D+E
istore F // pop F


  ILP Execution
===================
ADD A, B, C
SUB D, E, F
```

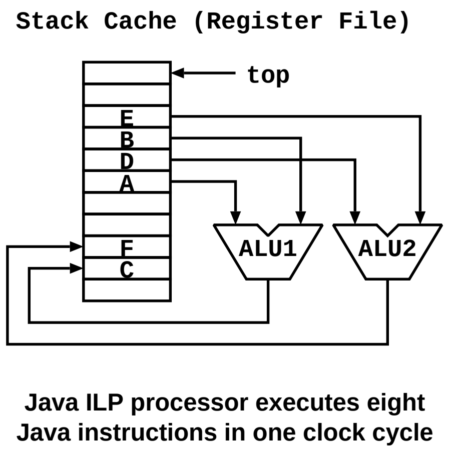
Stack Cache (Register File)
top
E
B
D
A
F
C
ALU1
ALU2
Java ILP processor executes eight
Java instructions in one clock cycle

Fig. 1. Instruction folding

2.2 Java ILP Processor Architecture

Java bytecodes are scheduled in an *instruction scheduling window* which consists of the instruction buffer and the instruction scheduling unit (ISU), as shown as in Fig. 2. An operand stack cache unit (register file) is provided with multiple independent read and write ports.

Instructions are executed in the integer and floating point units. Multiple functional units are provided with reservation stations (RSs) as well as data latches. The source operands can be fetched in parallel from the register file and fed to the RSs/latches. Some source operands may not be available due to data dependencies. In this case, after data are produced by functional units, they will be passed to the RSs/latches where the instructions are waiting for the results. The instructions in the instruction buffer can be issued out-of-order and a reorder buffer is used for instruction graduation. Some

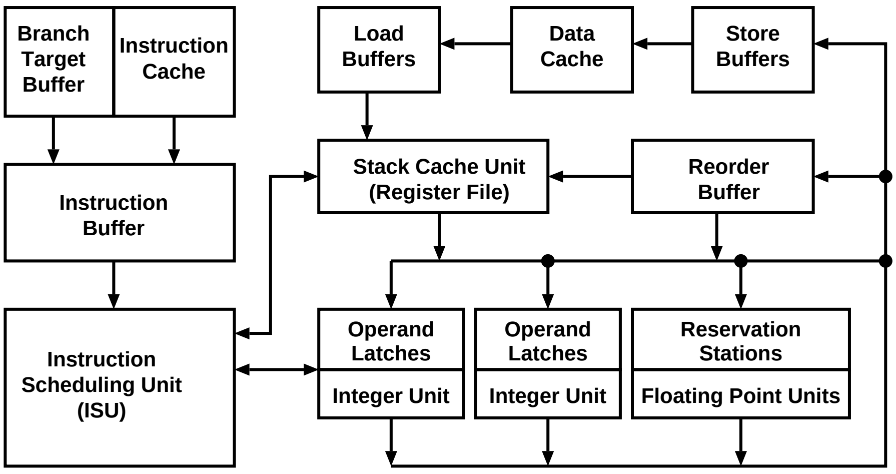

Fig. 2. Java ILP Processor

temporary variables will not be used again: there is no need to store them in the register file.

When the ISU encounters a branch instruction whose branch target address can be determined (such as unconditional branch, method invocation or return), it can transfer to the correct branch target and continue fetching instructions; if it encounters a conditional branch, it predicts the branch target by using Branch Target Buffer[Lee and Smith, 1984], fetches instructions from the predicted target, and instructs the instruction scheduling unit to execute them speculatively. If the branch is mispredicted, it instructs the instruction scheduling unit to cancel the speculatively executed results and fetches instructions from the correct branch target.

Memory access dependencies can be more easily checked than in a RISC processor, because in JVM, different types of objects are accessed by different types of instructions and there is no data dependency between the different types of objects. A load buffer and a store buffer are provided with tags indicating the type of the data in the buffer. Data dependences are only checked within the same data type and data can be bypassed from the store buffer to the load buffer if their addresses are the same.

2.3 Local Variable Accesses

Local variable accesses are coalesced with computational instructions. In the JVM, the *local variables* and *temporary variables* are located in the operand stack. Most Java instructions are temporary variable oriented. These instructions get operands from the stack, operate on them and then push results onto the stack.

Java is an object-oriented language: method invocations (equivalent of function, procedure, or subroutine calls in structured languages) occur very frequently. For each method, the JVM creates at runtime a method frame of variable size, which contains the method parameters and local variables. In order to eliminate unnecessary parameter passing between methods, the stack is constructed so as to allow overlap between methods, enabling direct parameter passing with no parameter copying. This structure overcomes the disadvantages of register reuse in global register file architectures (e.g. Power PC, Alpha etc). Furthermore, it avoids the shortcomings of a rigid structure in circular overlapping register windows architectures with fixed window sizes (e.g. SPARC).

When operations require access to the local variables, *iload*-like instructions will be used to load values of the local variables onto the operand stack (temporary variables) and *istore*-like instructions store values of the temporary variables from the operand stack to local variables. Consequently, Java computational instructions can use zero-address format. This not only reduces the cost of storing bytecode, but in any networked computing model, it also effectively increases the available bandwidth[McGhan and O'Connor, 1998].

By including a register-file-like operand stack cache in our design, variables can be addressed directly so that data movement between the variables and the top of the stack becomes unnecessary.

2.4 Memory Accesses

The JVM architecture reflects object-oriented features of Java language. Similar to traditional RISC architectures, JVM memory reference instructions also include *load* and *store* instructions, but with significant differences in accessing modes.

Usually, RISC architectures have two kinds of addressing mode: direct addressing and indexed addressing (we don't distinguish between an index register and a base register). However, in reality, in most application programs, a large number of instructions use the index addressing mode because the direct addressing mode is unable to access the whole address space. Using the index addressing mode means that the instruction scheduling unit encounters many unknown memory addresses during program execution, hampering efficient instruction scheduling and ILP exploitation.

There are three memory addressing modes in JVM: (1) *Array element access*: An element of an array is accessed by an array access reference address followed by an element index (*array, index*). Both the array access reference address and the element index are variables. (2) *Object field access*: A field of an object is accessed by an object access reference address followed by a field offset (*object, member*). The object access reference address is a variable, but the field offset is a constant which can be put in the instruction code. (3) *Static field access*: A static field is accessed directly by the static field reference address (*address*). The address is in the constant pool and instructions provide an index to the constant pool.

Cross-boundary access and operation on data with different types are not permitted. Therefore, many memory reference dependencies can be recognised from the instruction opcodes and the data types before all the memory addresses are calculated. We have the following rules for dependency detection: (1) There is no dependency between instructions with different addressing modes. (2) There is no dependency for instructions with different data types. (3) For object field accesses, if the object reference addresses or field offsets are different, no dependency exists. (4) For static field accesses, if the *addresses* are different, then there is no dependency. (5) For array object accesses, if the array reference addresses or element indices are different, then no dependency exists.

With these rules, a lot of non-dependencies can be recognised in advance. This not only improves the effect of instruction scheduling, but also simplifies the design of a dynamic scheduling unit for memory reference instructions.

3 Architecture Simulator and Test Programs

In order to evaluate the performance of the Java ILP processor architecture, we developed an architectural simulator in C++. The simulator focuses on the measurement of effective instructions per cycle (EIPC) (which just counts computational instructions) and the number of read/write ports of the operand stack cache. It consists of a Java bytecode tracer and a performance simulator. The tracer accepts Java bytecode programs (class files) and executes them by interpretation. During interpretation, each Java bytecode instruction (trace) is sent to the simulator. The performance simulator accepts the trace and measures the performance with various processor configurations.

We make the following basic assumptions. First, the data cache is 32Kb, 32-byte line size, and 4-way set associative. Second, the branch prediction buffer employs a fully associative table with 512 entries and 2-bit counters for prediction. Third, Java processor functional units have 3 kinds of execution pipelines: integer, branch and float-point, used for executing integer, conditional branch, and floating point instructions, with execution latencies of 1, 1, and 3 cycles, respectively. All are fully pipelined: floating point division may need more cycles for execution, but here we assume its latency is also 3 cycles.

Our simulation studies the effective instruction execution per cycle and the number of required read/write ports of the operand stack cache. We carried out the simulation with different instruction scheduling windows (16, 32 and 64 instructions).

In this paper, we used the following Java programs for testing. Programs were compiled to Java class files for simulation.

linpack: linpack Java version for measuring JVM performance; uses double precision data (2 words in size) for matrix calculations; total instructions 8×10^6.

qsort: sorts the elements of an integer array; total instructions 217.2×10^6.

sieve: generates prime numbers; total instructions 165.5×10^6.

hanoi: solves the Towers of Hanoi puzzle recursively; total instructions 178.3×10^6.

dhry: short dhrystone synthetic benchmark; total instructions 85.3×10^6.

fibo: calculates Fibonacci numbers of long words (2 words) iteratively; total instructions 288.2×10^6.

array: stores random integer values into an array; total instructions 42.5×10^6.

tree: adds new nodes with integer values into a tree using recursive calls; total instructions 77.0×10^6.

4 Preliminary Results

Table 1 shows the measured EIPC values with no resource restrictions: they represent the potential parallelism of the Java bytecodes. The table also lists the number of ports of the stack cache required to obtain this performance.

SR	Items		linpack	qsort	sieve	hanoi	dhry	fibo	array	tree	overall
			Program								
16	EIPC		2.11	2.74	2.89	3.85	4.22	3.99	1.94	1.95	2.96
	IPC		4.57	7.67	7.79	13.08	11.69	15.96	5.31	6.01	9.01
	AvgPorts	Read	5.09	4.83	4.66	6.77	5.85	12.97	4.69	3.90	6.10
		Write	2.28	1.01	0.67	2.46	1.86	4.99	2.19	1.84	2.16
	MaxPorts	Read	20	22	14	15	16	20	14	16	22
		Write	8	9	9	9	9	11	9	9	11
32	EIPC		3.40	2.82	2.89	4.55	5.22	3.99	1.94	2.10	3.36
	IPC		7.39	7.91	7.79	15.45	14.45	15.97	5.31	6.47	10.09
	AvgPorts	Read	8.22	4.98	4.66	8.00	7.24	12.97	4.69	4.21	6.87
		Write	3.68	1.04	0.67	2.91	2.30	4.99	2.19	1.98	2.47
	MaxPorts	Read	36	27	27	27	27	27	27	27	36
		Write	14	14	14	14	14	14	14	14	14
64	EIPC		5.44	2.83	2.89	6.20	6.29	3.99	1.94	2.10	3.96
	IPC		11.81	7.92	7.79	21.08	17.41	15.97	5.31	6.47	11.72
	AvgPorts	Read	13.14	4.99	4.66	10.91	8.71	12.97	4.69	4.21	8.04
		Write	5.88	1.05	0.67	3.97	2.78	4.99	2.19	1.98	2.94
	MaxPorts	Read	49	56	30	30	30	30	30	30	56
		Write	23	21	20	20	20	20	20	20	23

SR: Scheduling range (instructions)

Table 1. EIPC and number of ports required

AvgPorts represents the average number of read/write ports for the stack cache and MaxPorts represents the maximum number of read/write ports which are required.

EIPC is calculated by dividing the total number of executed computational instructions by the total number of clock cycles, where the effective instructions are the computational instructions; while IPC includes all instructions: unconditional branch, method invocation and return, stack manipulations and local variable instructions (transfers between local variable and the top of the stack), in addition to computational instructions.

The IPC results are two to three times the EIPC ones, because effective instructions are 30–50% of all instructions. EIPC improves as the instruction scheduling scope increases but will become saturated, except for *linpack*, *hanoi*, and *dhry*. This is, in part, due to the inherent parallelism of the programs. The other factor which influences the EIPC is correctness of branch prediction. We list the branch prediction success in Table 2.

From Table 1, we observe that the required number of the operand stack cache ports becomes larger as the size of instruction scheduling window increases. MaxPorts is the maximum number of ports which would allow the processor to access as many data as

Program	linpack	qsort	sieve	hanoi	dhry	fibo	array	tree
Hit ratio	0.967	0.795	0.878	0.667	0.867	0.999	0.750	0.817

Table 2. Branch prediction success

desired in parallel without performance decreasing.

Although MaxPorts is large, AvgPorts is quite small. In a real implementation, we can fix the number of the operand stack cache ports at reasonable values. Next, we measured EIPC with fixed numbers of operand stack cache ports. In general, a computational instruction reads two operands from the stack, operates on them, and writes the result onto the stack. So we assumed that the number of read ports is twice the number of write ports for the operand stack cache. All results are shown in Table 3. EIPC improves as the number of ports increases. If the number of ports is small, EIPC is not enhanced by increasing the size of instruction scheduling window. The results listed in Table 1 and 3 are helpful for tuning the architectural parameters when we design a Java ILP processor.

SR	Read/Write ports				
	2/1	4/2	6/3	8/4	10/5
16	0.96	1.79	2.25	2.63	2.77
32	0.97	1.82	2.39	2.86	3.05
64	0.97	1.82	2.39	2.93	3.25

SR: Scheduling range (instructions)

Table 3. EIPC results with fixed number of ports in all.

We also measured the hit ratio and overflow/underflow for the operand stack cache. The results are listed in Table 4.

SR	Cache size (words) & Hit Ratio				SR	Cache size & Overflow/Underflow (10^3)			
	64	128	256	512		64	128	256	512
16	0.990	0.996	0.999	1	16	245.8/245.8	70.3/70.3	12.5/12.5	0/0
32	0.989	0.995	0.998	1	32	231.3/231.2	70.3/70.3	12.1/12.1	0/0
64	0.988	0.994	0.998	1	64	159.7/159.6	60.4/60.3	12.0/11.9	0/0

SR: Scheduling range (instructions)

Table 4. Operand stack cache hit ratio and overflow/underflow in all.

Because the JVM is a stack oriented architecture (most operations take place at the top of the stack), the hit ratios are quite high. The hit ratio becomes a little higher as the size of the cache increases. An interesting fact is that the hit ratio becomes worse as the scheduling scope increases. This is because, in the case of instruction scheduling with a large scope, more data needs to reside in the cache simultaneously.

Overflow or underflow occurs when the program invokes new methods resulting in access to locations out of the range of the stack cache. The cache can be organized as a circular buffer. A procedure for dealing with overflow and underflow is needed. It usually saves old data to the data cache when overflow occurs and restores data from the data cache on underflow. In a real processor design, the save and restore operations can be performed in the background so that we can use a small operand stack cache. This also makes it easy to design a stack cache with multiple read/write ports.

5 Conclusion

We have described a Java ILP processor design which can perform parallel execution of the computational instructions. High performance is achieved by providing parallel access to the stack cache and executing local variable instructions with zero time.

We have also developed a Java ILP processor architecture simulator which can evaluate the performance at various processor configurations. Simulations on various Java benchmark programs were performed in order to predict the performance improvement compared to the traditional stack based processor architecture which does not exploit ILP. Our results show that the new processor with a 16-instruction scheduling window could achieve an average 2.96 EIPC or 3.36 EIPC with a 32-instruction window.

References

B., Gosling J.and Joy and G., Steele. The java language specification. Addison-Wesley. see also http://www.javasoft.com/docs/books/jls/index.html.

Ebcioglu, K., Altman, E., and Hokenek, E. (1997). A java ilp machine based on fast dynamic compilation. In *International Workshop on Security and Efficiency Aspects of Java*.

Lee, J. and Smith, A. J. (1984). Branch prediction strategies and branch target buffer design. In *IEEE Computer*, pages 6–22.

Lindholm, T. and Yellin, F. (1997). The java virtual machine specification. Addison-Wesley. see also http://www.javasoft.com/docs/books/vmspec/index.html.

McGhan, H. and O'Connor, M. (1998). Picojava: A direct execution engine for java bytecode. In *IEEE Computer*, volume 31, pages 22–30.

Shieh, C. A.and Gyllenhaal, J. C. and Hwu, W. W. (1996). Java bytecode to native code translation: The caffeine prototype and preliminary results. In *Proc. MICRO-29*. IEEE Press.

Turley, J. (1997). Microjava pushes bytecode performance – sun's microjava 701 based on new generation of picojava core. In *Microprocessor Report*, pages 9–12.

WhitePaper, Sun's. Picojava-i microprocessor core architecture. http://www.sun.com/microelectronics/whitepapers/wpr-0014-01/.