ACSAC01, Australian Computer Science Communications, Vol.23, No.4, IEEE Computer Society Press, 2001, pp.122–129

Exploiting Java Instruction/Thread Level Parallelism with Horizontal Multithreading

Kenji Watanabe†, Wanming Chu†, and Yamin Li‡

†Department of Computer Hardware
University of Aizu
Aizu-Wakamatsu 965-8580 Japan

‡Department of Computer Science
Hosei University
Tokyo 184-8584 Japan

Abstract

Java bytecodes can be executed with the following three methods: a Java interpretor running on a particular machine interprets bytecodes; a Just-In-Time (JIT) compiler translates bytecodes to the native primitives of the particular machine and the machine executes the translated codes; and a Java processor executes bytecodes directly. The first two methods require no special hardware support for the execution of Java bytecodes and are widely used currently. The last method requires an embedded Java processor, picoJavaI or picoJavaII for instance. The picoJavaI and picoJavaII are simple pipelined processors with no ILP (instruction level parallelism) and TLP (thread level parallelism) supports. A so-called MAJC (microprocessor architecture for Java computing) design can exploit ILP and TLP by using a modified VLIW (very long instruction word) architecture and vertical multithreading technique, but it has its own instruction set and cannot execute Java bytecodes directly. In this paper, we investigate a processor architecture which can directly execute Java bytecodes meanwhile can exploit Java ILP and TLP simultaneously. The proposed processor consists of multiple slots implementing horizontal multithreading and multiple functional units shared by all threads executed in parallel. Our architectural simulation results show that the Java processor could achieve an average 20 IPC (instructions per cycle), or 7.33 EIPC (effective IPC), with 8 slots and a 4-instruction scheduling window for each slot. We also check other configurations and give the utilization of functional units as well as the performance improvement with various kinds of working loads.

1. Introduction

Java programming language [3] has been widely accepted by industry and academia because of its powerful functionality and portability, as well as the popularity of the Internet. Java programs are compiled to classes, containing Java bytecodes and data, based on a virtual architecture – the Java Virtual Machine (JVM) [9]. JVM is a stack-oriented architecture, it is not dependent on any particular real machine. Java bytecodes may be executed on various platforms by interpretation or Just-In-Time (JIT) compiling to the native primitives of the particular machine. The interpretation means that the JVM architecture is emulated on a machine and an interpreter interprets bytecodes one by one. Compared to the direct execution of instructions, interpretation will run at one tenth or even hundredth the speed. Because the JVM is a stack oriented architecture, there are 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 JIT compiler translates the Java bytecodes into native RISC primitives dynamically. Because the Java bytecodes cannot be executed directly, time for the compilation is always needed at run time. Several variants of the JIT concept have been proposed based on when bytecodes are translated [2][11]. 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.

For the purpose of increasing the performance of JVM, Sun Microsystems has announced hardware solutions, picoJavaI [10] [13] and picoJavaII [14]. These are stack based processors, and, do not address the issues of instruction level parallelism (ILP) and thread level parallelism (TLP). PicoJava aims at low cost embedded applications. However, with the increasing requirement of high-performance

network computing with Java, it is necessary to consider the ILP and TLP issues for high-performance Java processor architecture.

Sun's Microprocessor Architecture for Java Computing (MAJC) [12] design features both chip multiprocessing and multithreading in order to beef up performance. MAJC adopts a modified VLIW (very long instruction word) architecture and up to 4 instructions can be packed into a 128-bit long instruction word. This means that the MAJC cannot execute Java bytecodes directly, it still needs a compiler to compile bytecodes to the instructions of the MAJC machine. MAJC adopts a vertical multithreading technique which switches to another thread when one thread results in a cache-miss. The vertical multithreading allows the processor execute only one thread at a given time. MAJC can be implemented with multiple identical processors on a chip die. Each processor supports operations on all data types, such as integer, fixed-point, floating-point, and packed integers. Suppose one thread running on a processor needs only integer operation, then other functional units will not be used, and cannot be used by other threads running on other processors. This will result in low utilization of the functional units. Functional units are cheap, but important thing is that fully use of functional units can improve processor performance.

Similar work can be done with DAISY (Dynamically Architected Instruction Set from Yorktown) [4]. DAISY translates the binary code of a source architecture such as PowerPC, x86, and S/390 to the binary code of a VLIW target architecture at run-time, in a manner transparent to the user.

JVM architecture is a stack based architecture. Most Java arithmetic instructions operate only on the top of the stack. This feature makes the Java bytecodes more efficient for its transmission over internet [10]. But meanwhile, it also makes parallel execution of the Java bytecodes more difficult because the stack top becomes a bottleneck for performance improvement.

In this paper, we investigate the various level parallelism of Java bytecodes and propose a Java processor architecture which can exploit Java ILP and TLP efficiently. Our processor architecture consists of multiple issuing slot, each slot can exploit the ILP by executing multiple computational bytecodes as well as by executing *push* and *pop* with zero time. The TLP is exploited by the parallel execution of multiple threads simultaneously, we call it *horizontal multithreading*. The processor architecture contains multiple functional units shared by all threads. We can expect higher performance with a small amount of functional units. We have developed a Java ILP/TLP architectural simulator which evaluates the performance and functional unit utilization at various processor configurations. The simulator consists of two parts: a Java bytecodes tracer and a performance simulator. The tracer interprets Java class file and record the execution trace into a trace file. The simulator reads the trace file and the processor configurations, simulates the performance, and outputs the simulation results. Our simulation results show that the Java processor could achieve an average 20 IPC, or 7.33 EIPC which counts only the computational instructions, with 8 issuing slots and 4-instruction scheduling window for each slot.

The rest sections are organized as following. Section 2 describes the concept of multithreading. Section 3 introduce the Java ILP/TLP processor architecture. Section 4 describes the Java architectural simulator and gives the simulation results. We conclude the paper in Section 5.

2. Vertical and Horizontal Multithreadings

Java provides the capability of *multithreading*. The programmer specifies that applications contain *threads of execution*, each thread designating a portion of a program that may execute concurrently with other threads [1]. The Java software in an application can generate multiple threads and let them run by *start*() method. These threads should be *synchronized* if there are dependencies. In the other hand, most modern machines support multiple tasks among which there may be no any dependency. Those tasks run on a single thread machine in a context switching manner. Executing multi-tasks in parallel will improve machine's throughput.

Multithreading can efficiently use the microprocessor's computing power. When one task results in a cache-miss, the processor will have an idle period while the system imports data from the main memory. Multithreading allows the CPU to work on a different thread during that down time. The overall throughput is much higher. The first thread will finish at about the same time as it would on a traditional CPU. In this case the same processor can complete several other things at the same time [15]. This capability is called *vertical multithreading*, as shown as in Fig. 1.

Vertical multithreading improves the utilization of the traditional CPU by putting other useful tasks into the idle time of the first task. But if we view the execution of the CPU at a given time, there is only a single task running. This is very similar to the technique of *time slicing* by which each task has a scheduled period for the execution. Sometimes we say the vertical multithreading or time slicing can perform multiple task *concurrently*, but not *simultaneously*.

The *parallel multithreaded architecture* (PMA) [6][7][8] supports the parallel execution of multiple tasks in a single processor environment. In a PMA processor, there are several *instruction issuing slots*. Each slot is equivalent to a *logical processor*. Multiple slots issue multiple instructions simultaneously from multiple threads at a clock cycle as shown as in Fig. 2. We also refer it to as *horizontal multithreading*.

PMA is different from a multiprocessor. In multiprocessor, there are several identical processors connected by

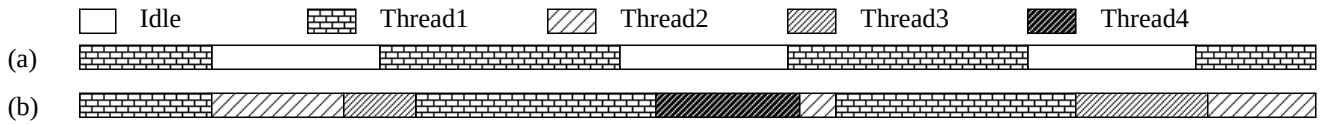

Figure 1. Vertical multithreading (a) Cache miss (b) Vertical multithreading.

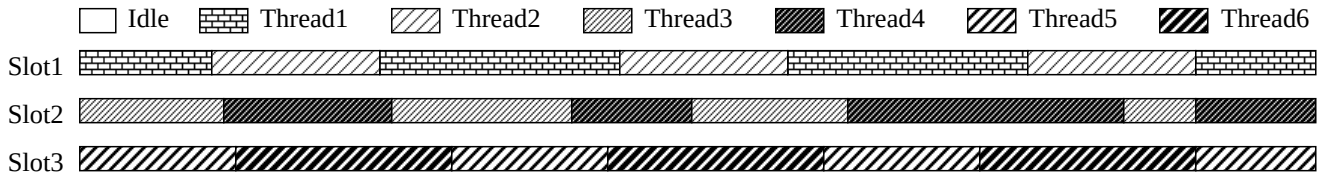

Figure 2. Horizontal multithreading.

interconnection network. Suppose a program running on a processor does not contain any floating point operation, then the floating point units in the processor will not be used and cannot be used by other programs running on other processors. In PMA, multiple functional units are shared by all slots, this will result in efficient use of the functional units. In Fig. 2, there are 3 slots and each slot can also perform the vertical multithreading.

3. Java ILP/TLP Processor Architecture

3.1. Java ILP Processor Architecture

Java ILP processor architecture focuses on the parallel execution of Java computational instructions within a single thread [16]. A typical computational instruction pops two operands from 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. 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.

Fig. 3 shows a possible 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). 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 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 [5], 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.

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

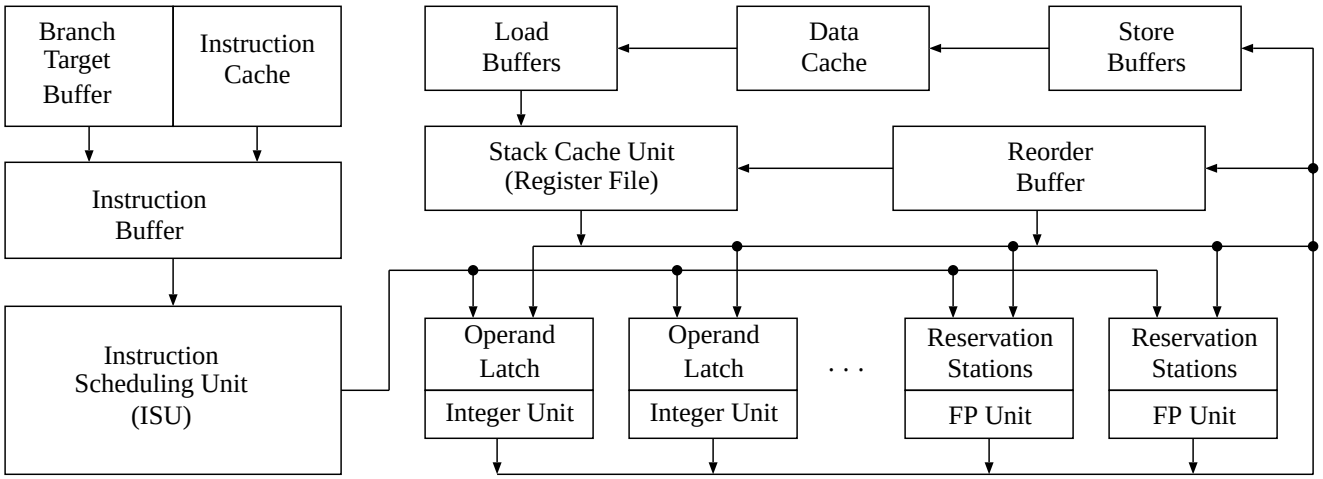

Figure 3. Java ILP Processor

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 bytecodes, but in any networked computing model, it also effectively increases the available bandwidth [10].

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.

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, in RISC architectures, a large number of instructions use the index addressing mode. This results in the instruction scheduling unit encountering 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.

In JVM, many memory reference dependencies can be recognized from the instruction opcodes and the data types before all the memory addresses are calculated, because the across-boundary access and operation on data with different types are not permitted. 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 recognized 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.

3.2. Java TLP Processor Architecture

The Java ILP processor issues multiple instructions from a thread at every clock cycle. We refer to the number of instructions the processor can issue as the *issue width*. The vertical multithreading can be implemented on such proces-

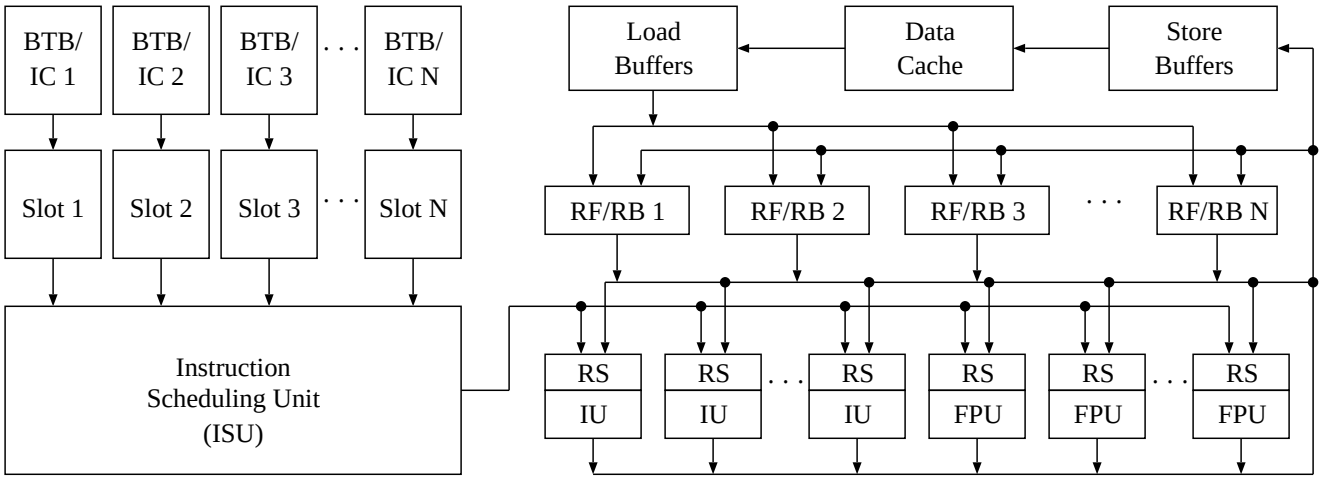

Figure 4. Java TLP processor.

sor by means of *context switching* on cache miss.

We extended the Java ILP processor to Java TLP processor as shown in Fig. 4, based on the Multiple-Instruction-Stream Multiple-Execution-Pipeline (MIS-MEP) architecture [7]. The Java TLP processor has multiple issuing slots which are provided with instruction caches (ICs), branch target buffers (BTBs), register files (RFs) and reorder buffers (RBs). Each slot can issue multiple instructions from one thread and deal with vertical multithreading. Multiple slots issue multiple instructions from multiple threads in parallel. Instructions are scheduled in ISU. If there is neither data dependency nor resource conflict, instructions are sent to functional units for execution. Different from multiprocessor, multiple functional units are shared among all slots.

Java bytecode instructions are executed in integer units (IUs) and floating point units (FPUs). The IUs contain branch units (BRUs), ALUs, multiplication units (MULs), division units (DIVs), and load/store units (LSUs). The BRUs are used for executing conditional branch instructions. The LSUs are used for executing load and store instructions except local accesses. Same as the Java ILP processor, the functional units are provided with reservation stations (RSs).

The multiple slots implement the horizontal multithreading; meanwhile, each slot can also implement the vertical multithreading if there are enough threads. For each thread, it is still possible to issue multiple instructions in every clock cycle.

4. Architectural Simulator and Experimental Results

In order to evaluate the performance of the Java ILP/TLP processor architecture, we developed an architectural simulator. The simulator focuses on the measurement of effective instructions per cycle (EIPC), which just counts computational instructions. We also measure the utilization of the functional units which is the average number of functional units required same time during the execution. We use the trace-driven method: a Java bytecode tracer interprets Java class file and saves required information into a trace file; then an architectural simulator reads the trace file and simulates the performance under various processor configurations.

We make the following basic assumptions. First, the stack cache (like register file) for each slot is 32 words, 2 read ports, 1 write port, and 1 background port. The background port is used to save and restore cache items when overflow and underflow happen. Second, the branch prediction buffer for each slot employs a 4-way associative table with 512 entries and 2-bit history counters for prediction. Third, the data cache is 32K bytes, 32-byte line size, and 4-way associative. Next, Java processor functional units fall into 6 types: BRU, ALU, FPU, LSU, MUL, and DIV. The execution latency of BRU is 1 cycle; however, *lookupswitch* instructions require 3 or more cycles, and *tableswitch* instructions require 5 cycles. The latencies of ALU and FPU are 1 and 3 cycles, respectively. The LSU execution latency is 1 cycle if data cache hits, and miss penalty is 3 cycles if the L2 cache hits. The latencies of MUL and DIV are 3 and 14 cycles respectively.

Our simulation studies the effective instructions per cycle and the number of required functional units. We carried out the the simulation with different number of slots (1, 2, 4, and 8 slots) and different reorder buffer size of each slot (4, 8, 16, and 32 words).

We used the following Java programs for the simulation.

- *array*: stores random values into an array.
- *bbsort*: sorts the elements of an array by a bidirectional

Table 1. Total executed instructions (I) and effective instructions (E)

	array	bbsort	bsort	dstone	fibo	hanoi	linpack	qsort	sieve	tree
I	0.645	0.497	0.539	0.511	0.542	0.792	8.049	0.630	0.748	0.743
E	0.271	0.192	0.219	0.188	0.145	0.236	3.759	0.228	0.278	0.270

($\times 10^6$ instructions)

bubble sort algorithm.

- *bsort*: sorts the elements of an array by a bubble sort algorithm.
- *dstone*: short dhrystone synthetic benchmark.
- *fibo*: calculates Fibonacci numbers iteratively.
- *hanoi*: solves the Towers of Hanoi puzzle recursively.
- *linpack*: Linpack benchmark (matrix calculations).
- *qsort*: sorts the elements of an array by a quick sort algorithm.
- *sieve*: generates prime numbers.
- *tree*: adds new node with values into a tree using recursive calls.

The programs are compiled to Java class files for simulation. In each program, the total executed instructions (I) and effective instructions (E) are shown in Tab. 1. We can see that the effective instructions are generally 30–50% of all instructions.

The trace files are used as instruction streams of the TLP architectural simulator. In order to investigate the potential parallelism, we first simulate the test programs without resource restriction of functional units for Java ILP/TLP processor (Configuration I). The simulation studies the effective instruction execution per cycle and the average number of required functional units.

Fig. 5 shows the measured instructions per cycle (IPC) and effective instructions per cycle (EIPC) with different RB sizes. These represent the potential parallelism of Java bytecodes. EIPC is calculated by dividing the total number of executed effective instructions by the total number of clock cycles, where the effective instructions are the computational instructions. IPC includes all instructions: branch, method invocation and return, stack manipulations and local variable instructions (transfers between local variable and the top of the stack), in addition to the computational instructions. The effective instructions are 30–50% of all instructions, so the IPC results are about two to three times of the EIPC. EIPC improves by increasing the instruction-level scheduling scope but will become saturated, except Java programs have the inherent parallelism. The Java processor could achieve an average 1.47 EIPC with a 32-instruction scheduling window of each slot and one instruction stream, and an average 7.33 EIPC with a 4-instruction scheduling window of each slot and 8 instruction streams.

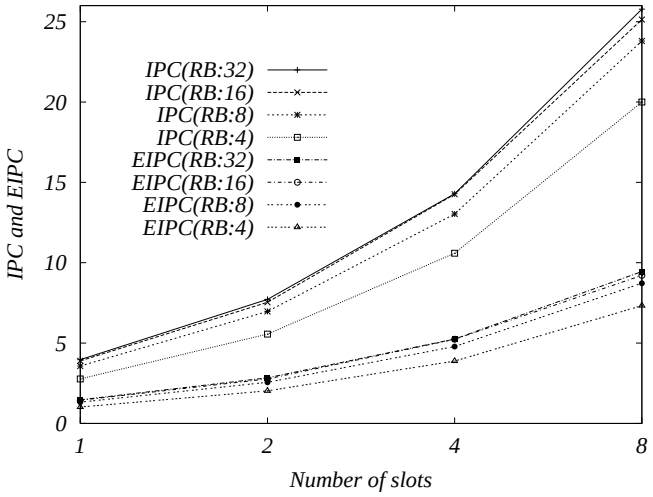

Figure 5. Result of IPC and EIPC, Configuration I.

Fig. 6 shows the total utilization of functional units with 4, 8, 16, 32-instruction scheduling window for each slot. The utilization is the average number of functional units per cycle requested for computation. We found that the load/store units are most used among all kinds of functional units. It means that the number of load/store units and the access ports of data cache are important issues for the performance improvement.

Fig. 7 shows the speedup of overall simulation results over the basic configuration which has 1 slot and 4-word reorder buffer. The speedup becomes higher by the increasing reorder buffer size and number of slots. However, by increasing the scheduling scope of instructions, performance improvement is not obvious. This is because of the limitation of ILP in one thread due to the data dependency and control dependency. Increasing the number of slots improve performance greatly if there are sufficient independent threads. We conclude that exploiting the thread level parallelism is becoming more important than that of instruction level parallelism.

The simulation results shown above assume that there is no restriction on functional units, that is, there are as many functional units as required. However, in a real processor

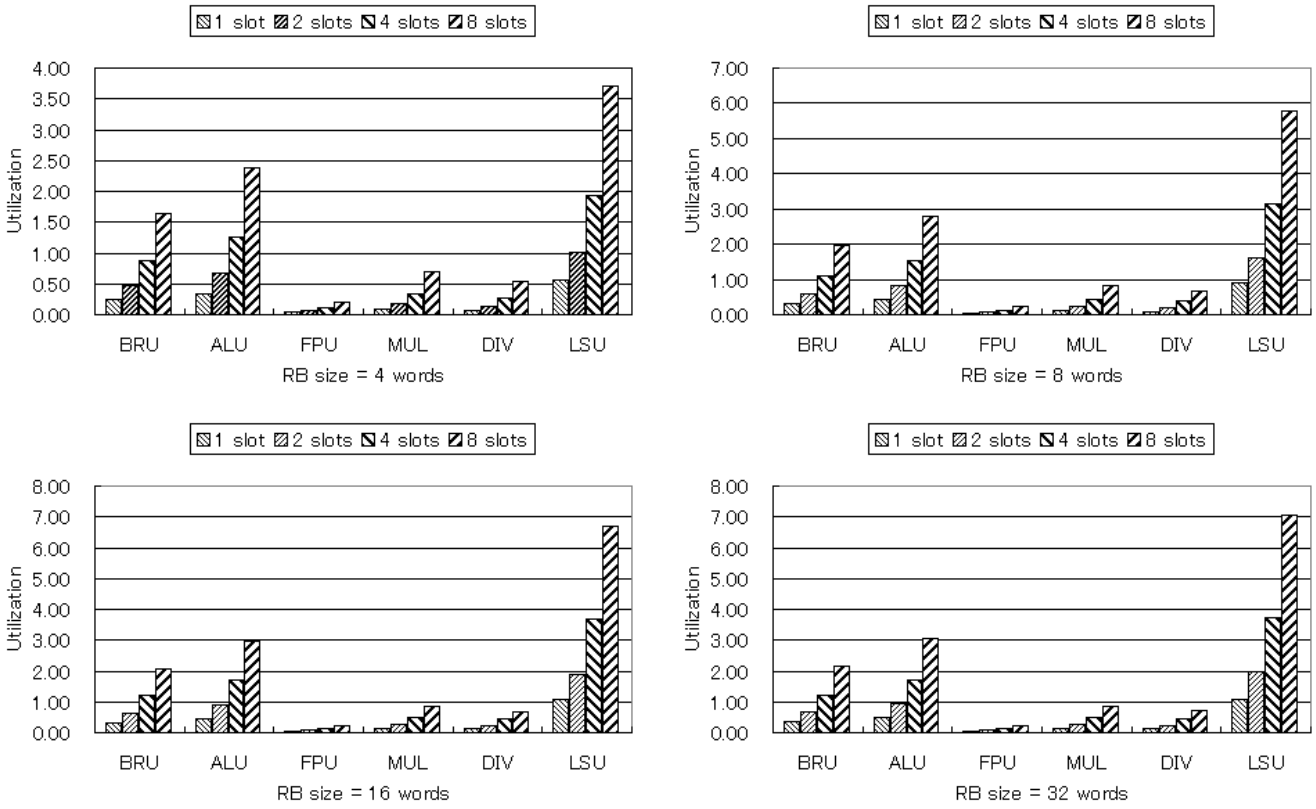

Figure 6. Utilization of functional units, Configuration I.

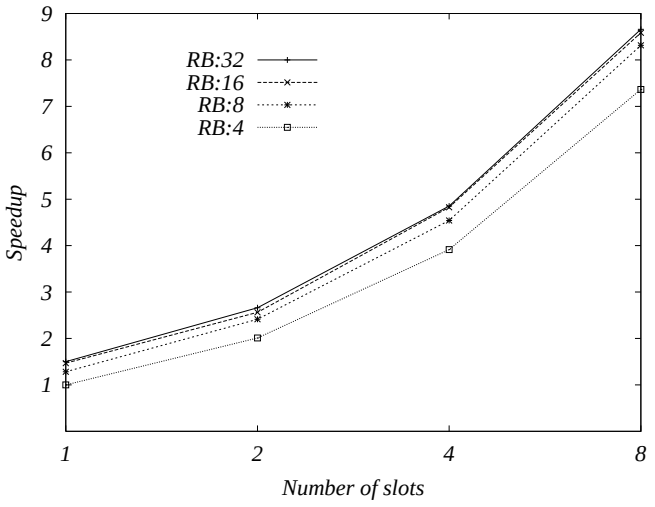

Figure 7. Result of IPC and EIPC, Configuration I.

design, we must fix the number of functional units. Next we will show the results under the following two configurations: Configuration II (CII) and Configuration III (CIII).

FUs:	BRU	ALU	FPU	MUL	DIV	LSU
CII:	1	1	1	1	1	1
CIII:	2	2	1	1	1	4

Tab. 2 shows the performance degradation of the two configurations. In Tab. 2, ET stands for the execution time in 10^3 clock cycles, ratio is the times of ET of Configuration I (CI). As the scale of the processor becomes larger, the ratio also becomes larger, this is because of the lack of the functional unit resource. For example, the ET of CIII with 8-slot and 4-instruction scheduling window is 3.40 times of the ET of CI.

5. Conclusion

Java has been widely accepted by industry and academia, because of its portability and ease-of-use, as well as the popularity of the Internet. Higher performance of the JVM is indispensable in order to develop the applications that demand high performance. As network computing gains importance, high performance Java ILP/TLP processors will be demanded in the near future.

Table 2. Performance comparison of three configurations.

	Slots	1				2				4				8			
	RB	4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32
CI	ET	8971	6995	6107	5978	4462	3717	3500	3369	2292	1975	1859	1849	1218	1079	1045	1036
	IPC	2.77	3.57	3.89	3.97	5.56	6.97	7.54	7.72	10.59	13.03	14.26	14.29	20.00	23.80	25.13	25.78
	EIPC	1.02	1.32	1.44	1.47	2.03	2.56	2.77	2.84	3.88	4.78	5.24	5.25	7.33	8.72	9.21	9.45
CII	ET	9457	8183	7998	8015	5832	6138	6357	6360	5322	6201	6360	6364	5545	6337	6471	6509
	IPC	2.63	3.12	3.26	3.33	4.37	4.32	4.26	4.26	4.61	4.26	4.21	4.16	4.22	4.00	3.91	3.88
	EIPC	0.96	1.15	1.19	1.22	1.61	1.58	1.55	1.56	1.70	1.56	1.54	1.52	1.55	1.47	1.43	1.42
	Ratio	1.05	1.17	1.31	1.34	1.31	1.65	1.82	1.89	2.32	3.14	3.42	3.44	4.55	5.87	6.19	6.28
CIII	ET	9274	7756	6748	7024	5215	4700	4598	4559	4271	4198	4199	4195	4140	4170	4163	4159
	IPC	2.76	3.53	3.80	3.89	5.42	6.63	7.01	7.20	9.13	10.04	10.12	10.18	11.10	10.63	10.78	10.81
	EIPC	1.01	1.30	1.40	1.43	1.97	2.42	2.56	2.63	3.29	3.61	3.64	3.66	4.01	3.85	3.90	3.91
	Ratio	1.03	1.11	1.10	1.17	1.17	1.26	1.31	1.35	1.86	2.13	2.26	2.27	3.40	3.86	3.98	4.01

In order to investigate the potential parallelism of Java bytecodes, we have described a Java processor architecture which aims to exploit ILP and TLP. Our Java processor can directly execute multiple computational bytecode instructions and multiple threads in parallel so that we can expect higher performance than attainable with PicoJava, or from interpretation and JIT compilation. Because the Java bytecodes are stack based, the top of the stack becomes the bottleneck for the performance improvement. High performance is achieved by executing local variable instructions with zero time and providing multiple instruction streams to execute threads simultaneously.

We have also developed a Java ILP/TLP architectural 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 and TLP. The simulation results show that the Java processor could achieve an average 1.47 EIPC with one instruction stream and a 32-instruction scheduling window (ILP), and an average 7.33 EIPC with 8 instruction streams and a 4-instruction scheduling window (ILP and TLP).

References

[1] H. M. Deitel and P. J. Deitel. *Java, How to Program*. Printice-Hall, Inc., 1997.

[2] K. Ebcioglu, E. Altman, and E. Hokenek. A java ilp machine based on fast dynamic compilation. In *Intl. Workshop on Security and Efficiency Aspects of Java*, Jan. 1997.

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley. see also http://www.javasoft.com/docs/books/jls/index.html.

[4] IBM. Daisy: Dynamically architected instruction set from yorktown. http://oss.software.ibm.com/developerworks/opensource/daisy.

[5] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[6] Y. Li and W. Chu. A performance prediction model for a parallel multithreaded risc processor architecture. In *Sixth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 162–166, Oct. 1994.

[7] Y. Li and W. Chu. Design and implementation of a multiple-instruction-stream multiple-execution-pipeline architecture. In *Seventh IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 477–480, Oct. 1995.

[8] Y. Li and W. Chu. The effects of stef in finely parallel multithreaded processors. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 318–325, January 1995.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley. see also http://www.javasoft.com/docs/books/vmspec/.

[10] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *IEEE Computer*, pages 22–30, October 1998.

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

[12] Sun. Majc architecture tutorial. http://developer.java.sun.com/developer/products/majc/docs.html.

[13] Sun'sWhitePaper. Picojava-i microprocessor core architecture. http://solutions.sun.com/embedded/databook/pdf/whitepapers/WPR-0014-01.pdf.

[14] J. Turley. Microjava pushes bytecode performance – sun's microjava 701 based on new generation of picojava core. *Microprocessor Report*, pages 9–12, November 17, 1997.

[15] W. Wade. Majc combines multiprocessing, multithreading features. http://www.edtn.com/story/tech/OEG19990819S0036-R.

[16] K. Watanabe and Y. Li. Parallelism of java bytecode programs and a java ilp processor architecture. *Australian Computer Science Communications*, 21(4):75–84.