The Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, Aug. 27-29, 2003 Chengdu, China. pp.180–187

An Instruction Cache Architecture for Parallel Execution of Java Threads

Wanming Chu
Department of Computer Hardware, University of Aizu
Aizu-Wakamatsu, 965-8580 Japan

Yamin Li
Department of Computer Science, Hosei University
Tokyo, 184-8584 Japan

Abstract

Designing a Java processor supporting horizontal multithreading has been becoming more attractive as network computing gains importance. Different from the traditional superscalar processors that issue multiple instructions from a single instruction stream to exploit the instruction level parallelism (ILP), the horizontal multithreading Java processors issue multiple instructions (bytecodes) from multiple threads in parallel to exploit not only the ILP but the thread level parallelism (TLP). Such processors have multiple dispatch slots and require the instruction fetch unit to supply instructions with much higher bandwidth than superscalar processors. Using a traditional superscalar cache architecture in a horizontal multithreading Java processor results in high cache miss ratio caused by the interference among the threads. This paper investigates a multibank instruction cache architecture for horizontal multithreading Java processor to meet the requirements of the high instruction fetch bandwidth. In order to evaluate the cache performance as well as the horizontal multithreading Java processor performance, we developed a trace driven simulator. The simulator consists of a trace generator which generates the Java bytecode execution traces and an architectural simulator which reads the traces and evaluates the performance of the instruction cache and the overall performance of the Java processor. Our simulation results show that the performance improvements are obtained by the low cache miss ratio and the high instruction fetch bandwidth of the proposed cache architecture. The IPC (instructions per cycle) performance is about 19 when the numbers of slots and banks both are 8, about 5 times better than one bank cache.

keywords: Cache, Java virtual machine, Java processor, instruction level parallelism, thread level parallelism, multithreading, performance evaluation, trace-driven simulation

1. Introduction

The superscalar processors try to exploit instruction level parallelism (ILP) by issuing multiple instructions in every clock cycle. However, the performance improvements obtained by exploitation of ILP from a single thread eventually reached their upper bound due to the data and control dependences. Data dependences between instructions cause the parallel execution of multiple instructions impossible; while control dependences break down the execution pipelines. Previous research results showed that the branch instructions occupy more than 15% of the executed total instructions in general, this means that, on average, the size of a *basic block*[1] is six to seven instructions in length. Many computer researchers have been concentriting on the fetching and scheduling instructions across the basic block.

The *multithreading* is a technique to tolerant the latency caused by time-consuming memory access. When a thread has to access memory due to a cache miss, processor switches to another thread to execute during the memory access of the original thread. Such context switching mechanism is referred to as *coarse-grain multithreading*. Another type of multithreading, called *fine-grain multithreading*, schedules instructions from different threads on every clock cycle. Fine-grain multithreading can also reduce the penalty of the data and control dependences. We call both the multithreadings *vertical multithreading* because, at a given clock cycle, only one thread's instructions are fetched.

Vertical multithreading tolerates memory access latency. On the other hand, *horizontal multithreading* tries to exploit thread level parallelism (TLP). The horizontal multithreading tries to fetch, issue and execute multiple instructions from multiple threads in parallel in every clock cycle. In [8], Tullsen et al proposed a RISC superscalar-based simultaneous multithreading (SMT) architecture. An SMT pro-

[1] A code sequence in which there is no branch instructions except the last instruction

cessor issues instructions of multiple threads from an instruction queue to functional units but fetches instructions from a single thread each cycle. Horizontal multithreading processors require much higher instruction fetch bandwidth than superscalar or SMT processors. This will enlarge the performance gap between processor and memory. In order to make multiple functional units busy, hence to improve the overall performance, an instruction cache with a different architecture from that for superscalar processors is absolutely required. We examine such instruction cache architecture for a horizontal multithreading Java processor and evaluate its performance in this paper.

Java programming language [4] 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) [7]. JVM is a stack-based architecture, it is not dependent on any particular real machine.

Java bytecodes are executed in an execution engine. Currently, the execution engines are implemented mainly in the following four schemes [10]. The simplest kind of execution engine is an software *interpreter* which interprets the bytecode one at a time. Another kind of execution engine is a *just-in-time* (JIT) compiler. In this scheme, the bytecodes of a *method* are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so the code can be reused the next time that method is invoked. This scheme is faster than the first one but requires more memory. A third type of execution engine is an *adaptive optimizer*. In this approach, the virtual machine starts by interpreting bytecodes but monitors the activity of the running program and identifies the most heavily used areas of code. As the program runs, the virtual machine compiles and optimizes only these heavily used areas to native machine code. Lastly, on a Java virtual machine built on top of a chip that executes bytecodes natively, the execution engine is embedded in the chip.

The picoJavaI [12] and picoJavaII [9] are simple pipelined Java processors with no support on ILP and TLP exploitation. Sun Microsystems' Microprocessor Architecture for Java Computing (MAJC) [1] 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 natively, 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 to support TLP exploitation. 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. On the other hand, if there is no sufficient TLP, some processors will be idle.

As network computing gains importance, high performance Java processors are demanded. In order to exploit the ILP and TLP of Java applications efficiently, we proposed a Java processor architecture which has the following three features.

1. Horizontal multithreading – there are multiple instruction dispatch slots to support the parallel execution of Java multiple threads.
2. Vertical multithreading – multiple threads can share each of the slots.
3. Superscalar – the ILP inside a thread can be exploited.

We call a processor holding these three features *three dimensional processor* (3D processor for short). Note that it is a processor, not a multiprocessor. In this paper, we investigate the instruction cache architecture to support the 3D processor. We developed a simulator which investigates Java bytecode instruction/thread level parallelism and evaluates the performance of the instruction cache as well as the performance Java processor. The simulator consists of a Java bytecode tracer and a Java processor architectural simulator. The tracer executes Java class file and records desirable information into a trace file. The architectural simulator reads the trace files and processor configurations, evaluates the processor performance and the utilization of the functional units. The tracer was developed in C++. We used the JNI (Java native interface) [6] and JVMDI (Java virtual machine debug interface) to load JVM and Java applications and gathers behavior data during the execution of the applications. The architectural simulator was developed in Java. By changing a configuration file, we can change the processor configuration, such as the number of slots, the number of banks of the instruction cache, the size of instruction scheduling window, the number of read/write ports of stack cache, the number of functional units etc. This simulator and the simulation results can be helpful for making the design decisions when high-performance Java processors are designed.

In [2], Chu and Li investigated the effects of the number of slots on the performance of a Java processor with the assumption of 100% instruction cache hit ratio. In this paper,

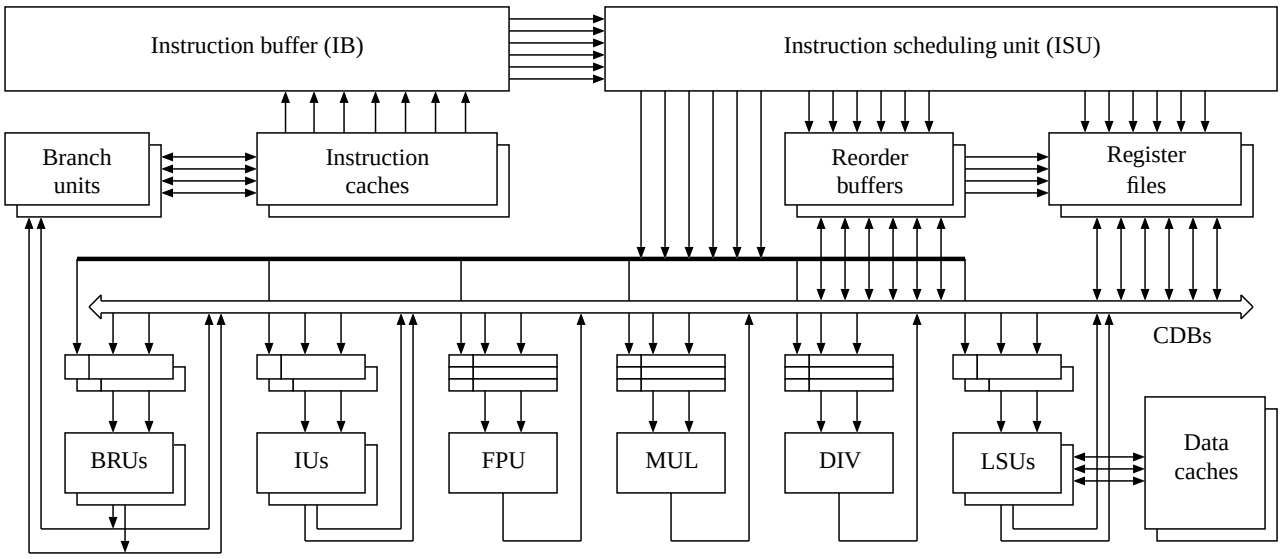

Figure 1. Java processor architecture

we describe an architecture of the instruction cache for a 3D Java processor architecture and give some preliminary simulation results to show the cache and the processor performances. The rest of this paper is organized as follows. Section 2 proposes the instruction cache architecture. Section 3 describes the processor architectural simulator. Section 4 gives simulation results. Section 5 concludes the paper and presents some future research directions.

2. Java Instruction Cache Architecture

Our processor architecture supports both ILP and TLP exploitation. It consists of multiple dispatching slots, each slot has a program counter and a register file which can exploit the ILP by executing multiple bytecodes as well as by executing *push* and *pop* with zero time. The TLP is exploited by the parallel execution of multiple threads simultaneously. The processor contains multiple functional units (FUs). The FUs are shared by all slots. We can consider a slot as a logical processor, and therefore, the processor consists of multiple logical processors. At runtime, each slot dispatches instructions from a thread so that up to n threads can be executed in parallel where n is the number of slots. If there are m threads and $m > n$, a slot switches among m/n threads on average.

JVM is a stack-based architecture. In the design of a 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 which serves as the stack top area with multiple read and write ports so that the constant, local variable, stack management, and array elements and fields reference instructions no need to be executed if these data are located in the cache – they can be accessed directly through the multiple ports of the cache. If the accessed data are not in the cache, the processor loads the data from memory to cache or store the cache date to memory, just like a load/store RISC architecture does. We can consider this cache as a register file in traditional RISC processors. In addition to the multiple read/write ports cache (register file), a data cache with a larger size than the register file is still required. When the processor loads data from main memory to register file, it also puts the data in the data cache. The register file is at the top position of the stack. It can be implemented in a circular structure [12]. When the overflow and underflow are about to happen, the contents of the register file will be saved and restored to and from the data cache, respectively. This can be done in background if an additional read port and a write port are designed in the register file. Each slot has its own register file, but the data cache can be shared by all slots.

Based on the types of Java bytecode instructions, we use the following kinds of FUs in the Java processor design: integer unit (IU), integer multiplier (MUL) and divider (DIV), branch unit (BRU), load/store unit (LSU) and floating point unit (FPU). These FUs are shared by all slots. Except for integer and floating point dividers, all the FUs are capable of accepting a new operation on every clock cycle. The Java processor architecture is shown in Figure 1.

In order to support *out-of-order* execution, we arrange a *reorder buffer* to each of the slots. The execution results are put into reorder buffer first, and then written to register file in order in the *commit* pipeline stage. The instruction scheduling scope is equal to the size of the reorder buffer.

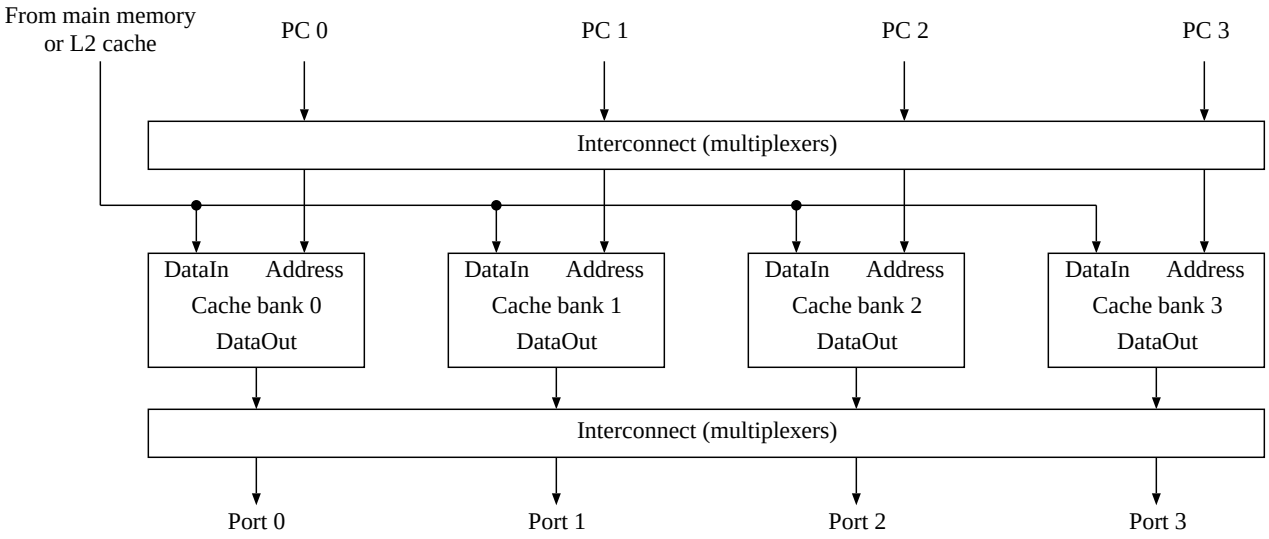

Figure 2. Instruction cache architecture

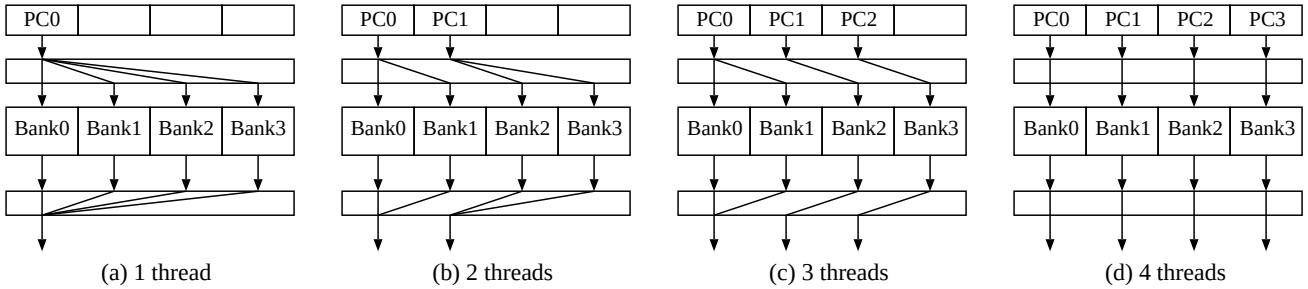

Figure 3. Instruction cache and threads

Each slot is provided with a branch unit containing a *branch target buffer* (BTB) [5], and each entry in BTB has a branch predictor. The BTB can be fully associated or set associated. The number of bits of a branch predictor can be 1 or 2. Instructions are fetched from instruction caches and put into an *instruction buffer* (IB). Then the instructions are scheduled and issued to FUs by an *instruction scheduling unit* (ISU).

The FUs are provided with *reservation stations*. Instructions can be issued to FUs even their source operands are not ready. The source operands are put into reservation stations once they are available. There are three sources the source operands come from: register file, reorder buffer and *common data buses* (CDBs). The CDBs connect the output of FUs, reorder buffers, register files and reservation stations. The instructions which are not issued due to resource restrictions stay in the IB waiting for being issued next clock cycle.

Assume that the number of slots is n. In order to fetch instructions from multiple threads simultaneously, we arrange n separated instruction cache banks. Generally, each cache bank holds instructions of a thread and all the banks can be accessed in parallel with their corresponding slot PCs. The fetched instructions from n banks through their read ports are put in the IB. The question is that, if there are m threads and $m < n$, then $n - m$ slots will not be used. In order to use the instruction banks efficiently, an interconnect is inserted between the cache banks and IB, as shown as in Figure 2.

The usage of the cache banks is described below. On average, each thread uses $k = \lfloor n/m \rfloor_{power2}$ banks, i.e., n/m is rounded to an integer of power of 2. The rest $n - mk$ banks are used by the first $(n-mk)/k$ threads. That is, each of the first $(n-mk)/k$ threads uses two-time banks compared to other threads. Totally, $2k(n-mk)/k + k(m-(n-mk)/k) = n$ banks are used by m threads. For example, if $n = 8$ and $m = 3$, the first thread will use 4 banks and each of the rest two threads uses 2 banks. Figure 3 shows the usage of a 4-bank cache where the cases of $m = 1, 2, 3$ and 4 are il-

lustrated in Figure 3 (a), (b), (c) and (d), respectively. If a thread uses more than one bank, the higher address bit(s) left to the bank index is used to select banks.

In the case of $m > n$, several threads have to share a single bank. Let thread T_i, where $i = 0, 1, \ldots, m-1$, be put in cache bank B_j, where $j = 0, 1, \ldots, n-1$, then we have $j = i \bmod n$. For example, thread T_j and thread T_{j+n} for $j+n < m$ will share cache bank B_j. Mapping multiple threads to a same bank will increase the cache miss ratio due to the interference among the threads if a direct mapping cache bank is used.

There are two methods to solve this problem. One is to use set-associative mapping cache such that a different thread may be put into a different *way* of the cache. The other is to use coarse-grain multithreading to switch threads at a given time interval. The first method makes fine-grain multithreading possible but if the number of threads mapping to a bank is larger than the ways of the set-associative cache, the cache performance still suffers. In addition, cache line replacement is needed for the set-associative cache, LRU (least recently used) strategy for instance, which requires extra gate resources. The second method requires fast context-switching facility and the cache will be refilled when the context-switching happens.

Both the methods implement the vertical multithreading. Note that the threads located in different cache banks can be executed in parallel (horizontal multithreading). Meanwhile, our Java processor architecture also exploit the ILP by fetching, issuing and executing multiple instructions from a single thread (superscalar). We investigate the effects of the instruction cache on the processor performance in Section 4.

3. Java Processor Simulator

This section describes the implementation of the processor simulator. *Trace-driven simulation* is one of the most widely used techniques for computer architecture design and performance evaluation. With this method, a tracer loads a user application, executes the application in *single step* manner, gathers desirable information from each instruction and records the information into a trace file that will be used late by an architectural simulator. This method requires a large storage space to store trace files. A similar method is called *execution-driven simulation* by which the generated trace is fed into the architectural simulator directly. The execution-driven simulation does not store the traces as files but it takes longer time for execution than the trace-driven simulation if an application is used more than one time: each time the application has to run to generate its trace. We used trace-driven simulation in our implementation because a trace file is needed several times for simulating the performance of the processors with different configurations, and also, multiple trace files are needed at same time for the simulation of multithreaded applications.

There are several ways to trace the execution of a Java application. A popular one is to develop a tracer that implements Java platform with the function of information extraction. This can be done by customizing the source code for a JVM implementation, the *Kaffe* virtual machine [11] for instance. The other way is to use the *Tracing JVM* [13], a modified JVM which can gather data on the behavior of Java programs. In our implementation, we use Java Native Interface (JNI) [6] and JVM Debug Interface (JVMDI) provided by JDK/JRE 1.3.0_02 for Linux operating environment. The JNI allows a native application, written in C/C++ for instance, to invoke Java applications and vice versa. The JVMDI is a programming interface that provides a way both to inspect the state and to control the execution of applications running in the JVM. JVMDI provides many functions, such as *set break point*, *enable single step execution* and *enable event handling*.

We use C++ to implement our tracer. We first use JNI invocation interface to load JVM implementation and Java application, then call JVMDI functions to set break point at the beginning of the Java application (`main` method) and to enable event handling, and use JNI invocation interface to invoke the Java application.

Setting break point at the beginning of the Java application lets the control transfer to an event handling procedure when the Java application is invoked first time. The event handling procedure is invokes not only on break point event, but also on other events, single step execution for instance. In the case of break point event, we use JVMDI functions to enable single step execution and clear all the break points (only one break point in our case), that is, the the break point event happens only one time.

Single step execution also lets the control transfer to the same event handling procedure after finishing the execution of each instruction. In the event handling procedure, we extract desirable information for each kind of instructions and put the information to a trace file. Then the control is returned back to the single step execution of the Java application.

In order to get desirable trace information, it is needed to use some structure definitions. For example, through the frame structure definition `javaframe` found in JDK `interpreter.h` and a frame *id* obtained by the JVMDI `GetCurrentFrame` function call, we can access constant pool, program counter of the next instruction, current top of stack, pointer to this frame's variables, previous java frame, program counter of the last executed instruction, method currently executing, monitor, profiler and start of frame's stack.

The following command starts the execution of the tracer and traces a Java application – `javaApp`:

```
$ jtracer -Xnoclassgc -Xdebug -Xnoagent \
-Djava.compiler=NONE -Djava.class.path=. \
javaApp
```

The important options in the command line are `-Xdebug` and `-Djava.compiler=NONE`. The `-Xdebug` option enables with debug and `-Djava.compiler=NONE` disables JIT compiler.

The trace information which is recorded to a file is described follow. For every instruction, the program counter (PC) and the first byte of bytecodes (opcode) are recorded. The stack pointer is also recorded for all instructions except for `nop`, `goto`, `goto_w`, `iinc`, `ret` and `return`. For the instructions that access memory, the memory address is evaluated and recorded. For goto, branch, switch, method invocation and procedure call and corresponding return instructions, the target addresses are recorded.

The second component of our simulator consists of a Java processor architectural simulator written in Java. It reads trace files and processor configuration, issues instructions to functional units cycle by cycle, and finally, outputs the experimental results.

4. Cache Performance Simulation Results

We selected the following Java applications for testing our cache architecture and evaluating the processor performance. The performance evaluated with trace- or execution-driven simulation method depends strongly upon the test applications, so the results presented in this paper are only for reference and comparison with each other to show the relative improvement with different cache architectures.

1. Linpack.java: Linpack benchmark (floating point matrix calculations). 44k instructions are executed.
2. BinTree.java: adds new node with values into a binary tree using recursive calls. (66k)
3. Dhrystone.java: short dhrystone synthetic benchmark. (60k)
4. Fibonacchi.java: calculates Fibonacci numbers iteratively. (56k)
5. Hanoi.java: solves the Towers of Hanoi puzzle recursively. (91k)
6. MatMult.java: floating point matrix multiplication. (58k)
7. QSort.java: sorts the elements of an integer array with a quick sort algorithm. (55k)
8. Sieve.java: generates prime numbers. (70k)

In the current version of our simulator, the processor configuration can be changed with the following parameters: the number of cache banks, cache line size, the number of each kind of functional units, the sizes of instruction buffers, reorder buffers and register files, the numbers of read and write ports of register files, and the number of slots. We assume that:

1. The multipliers has a 3-cycle latency and a 1-cycle throughput (fully-pipelined).
2. Both the throughput and latency of the dividers are assumed 14 clock cycles (non-pipelined).
3. Other FUs are fully pipelined with 1-cycle latency; however, `lookupswitch` instructions executed in BRU require 3 or more cycles and `tableswitch` instructions require 5 cycles.
4. The size of register file is 32 words with 4 read ports and 2 write ports.
5. The reorder buffer has 32 entries.
6. The data cache is a 4-way set-associated cache with a 32-byte block size. The total size of the data cache is 32K bytes.
7. The BTB has 512 entries organized to 4-way set-associated. A 2-bit branch predictor is assigned to each BTB entry.
8. The instruction cache size is 32K bytes and the line size is 32 bytes. The cache miss penalty is 5 cycles.

First, we investigate the potential ILP of bytecodes and the requirements for FUs. Toward to this, we assume that the FUs are always available if required, the hit ratio of the instruction cache is 100%, and instructions are provided as many as large enough to find instruction level parallelism (256 bytes per cycle in the simulation). Tables 1 list the processor performance and the functional unit requirements.

Table 1. Processor performance and average number of functional units required for each program

Programs	IPC	BRU	IU	FPU	MUL	DIV	LSU
Linpack	5.705	0.537	0.485	0.548	0.106	0.391	2.023
BinTree	5.234	0.599	0.422	0.0	0.015	0.015	1.659
Dhrystone	5.470	0.613	0.459	0.0	0.018	0.028	1.664
Fibonacchi	4.775	0.413	0.736	0.0	0.004	0.007	2.249
Hanoi	4.907	0.369	0.365	0.0	0.004	0.007	1.289
MatMult	4.353	0.335	0.272	0.231	0.081	0.007	1.696
QSort	5.278	0.619	0.434	0.011	0.015	0.025	1.643
Sieve	4.517	0.825	0.463	0.0	0.003	4.220	1.587

The column IPC in the table gives the average IPC calculated by dividing the total number of executed instructions by the total number of clock cycles. We can find that the

IPC is about 5. If we do not count the stack manipulation instructions, the IPC is about 4.

The required average number of each functional unit is calculated by dividing the sum of number of a functional unit required each clock cycle by the total number of clock cycles. The LSU is highly required for all the applications, especially for Linpack and Fibonacchi. The Sieve requires modular operation to find primes so the integer divider is used heavily. Notice that the divider is non-pipelined with a latency of 14-cycle. If we use a pipelined divider capable of accepting one operation per cycle, the average required number of dividers will be $4.220/14 = 0.3$.

Next, we test the processor performance with multiple threads. 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 [3]. The Java program can generate multiple threads and let them run with `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 multiple tasks in parallel will improve machine's throughput greatly. In order to simplify the implementation, we assume that a Java application generates 8 threads simultaneously, while each thread executes a distinct program of the 8 applications we used so far.

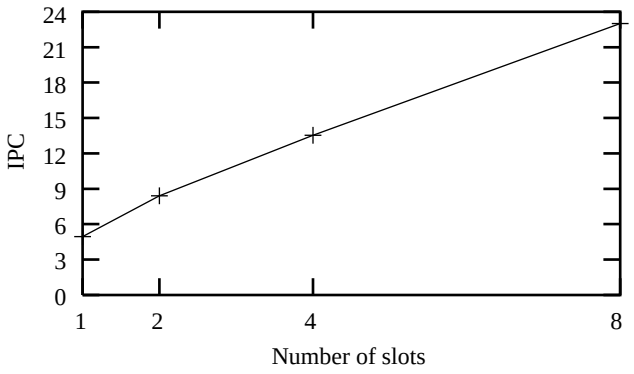

Figure 4. IPC *vs* slots (ideal)

Figure 4 shows the processor IPC performance when the 8 applications are executed concurrently. We change the number of issuing slots to n with $n = 1,\ 2,\ 4,\ 8$. The processor configuration is the same as the one described above. On average, $8/n$ threads share a slot. Each slot uses *round robin* to switch among these $8/n$ threads. The ISU also uses round robin to select instructions in IB among slots. We can find that the performance is improved greatly as the number of slots increases.

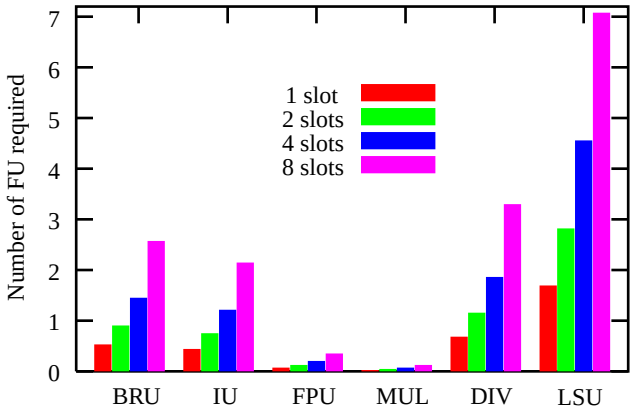

Figure 5. The number of FUs required

Figure 5 shows the average number of each functional unit needed when the 8 applications are executed concurrently. The average number of BRUs required is larger than that of IUs for our test applications. Also, much more LSUs are required. Executing multiple threads in parallel will mitigates the problem of heavy requirement on a particular FU, the integer divider for example. The table shows that the LSUs play the key role on the IPC performance. In general, the utilization of the LSUs is higher than that of other FUs.

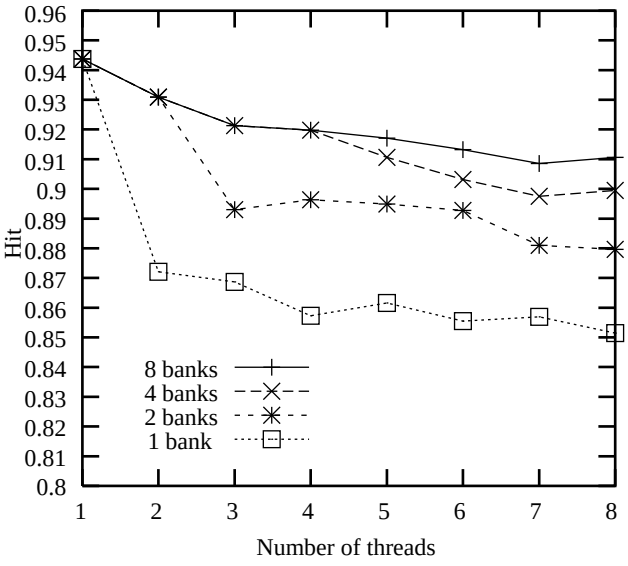

Figure 6. Hit ratio (cache size: 32K bytes)

Now, we consider effects of instruction cache on the processor performance. The cache size is 32K bytes and 32 bytes are read from each port. The number of slots is set to 8. In the design of the Java processor, the number of banks may be the same as the number of slots, but we want

to show here the effect of instruction cache architecture on the processor performance. Let $n = 1, 2, 4, 8$ be the number of instruction cache banks (ports) and $m = 1, \ldots 8$ be the number of threads. When $m < 8$, some slots will be idle. When $m < n$, some threads each use more than one bank, and when $m > n$, some banks are shared by multiple threads and we switch them on every 1024 clock cycles.

The cache performance is shown in Figure 6. As the number of threads increases the hit ratio decreases due to the interference among threads. But importantly, the performance of the multiple banks cache is better than that of single bank cache.

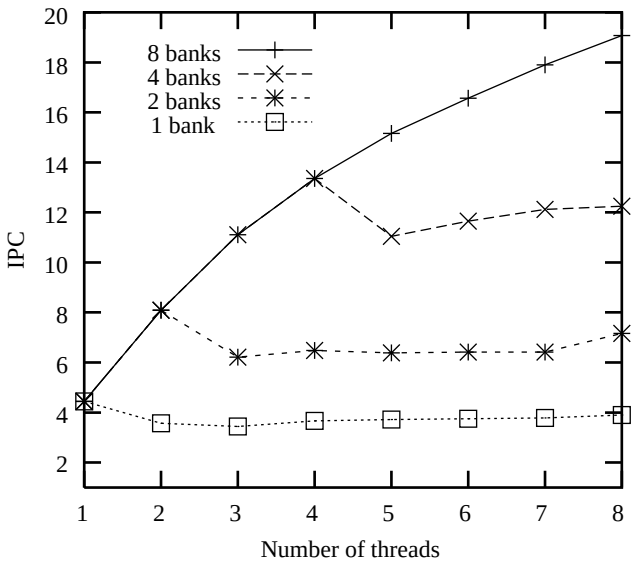

Figure 7. IPC *vs* banks (8 slots)

Figure 7 shows the IPC performance. The processor performance gains were achieved as the number of cache banks increases. On the other hand, when the number of threads reaches the number of banks, further increasing the number of threads will not improve the processor performance since some slots finish their execution earlier than others, that is, some slots will be idle while others are busy. For example, the IPC is 13.35 in the case of $m = n = 4$, but the IPC decreases to 11.04 when $n = 4$ and $m = 5$, because threads 0 and 4 share cache bank 0 and other threads in other banks may finish their execution earlier. The peak IPC performance reaches about 19 in the case of 8 banks, about 5 times better than the case of one bank cache.

5. Conclusion Remarks

A horizontal multithreading Java processor architecture is proposed and the effects of the instruction cache on the processor performance are evaluated in this paper. As Java is accepted by both industry and academia as well as the network computing gains importance, the development on high-performance Java processor to execute bytecodes natively becomes interesting. The simulator and the preliminary simulation results presented in this paper will help us to make the design decisions.

References

[1] MAJC architecture tutorial. see http://developer.java.sun.com/developer/products/majc/docs.html.

[2] W. Chu and Y. Li. Performance evaluation of a multiple-threaded multiple-pipelined java processor. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, volume V, Computer Science I, pages 281–286, July 2002.

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

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[5] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 244–251, January 1984.

[6] S. Liang. *The Java Native Interface Programmer's Guide and Specification*. Addison-Wesley, 1999.

[7] F. Yellin T. Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementaber simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[9] 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.

[10] B. Venners. *Inside the JAVA 2 Virtual Machine*. McGraw-Hill, second edition, 1999.

[11] The Kaffe virtual machine. see http://www.kaffe.org/.

[12] Sun's WhitePaper. Picojava-i microprocessor core architecture. see http://solutions.sun.com/embedded/databook/pdf/whitepapers/wpr-0014-01.pdf.

[13] M. Wolczko. Using a tracing java virtual machine to gather data on the behavior of java applications. see http://research.sun.com/people/mario/tracing-jvm/. 1999.