The Second European IASTED International Conference on Parallel and Distributed Systems, July 1–3, 1998, Vienna, Austria, pp.80–86

JAViR – Exploiting Instruction Level Parallelism for JAVA Machine by Using Virtual Registers

Yamin Li[⋆], Sanli Li [⋆†], Xianzhu Wang[†], and Wanming Chu[⋆]

[⋆]Department of Computer Hardware,
University of Aizu,
Aizu-Wakamatsu, 965-8580 Japan

[†]Department of Computer Science,
Tsinghua University,
Beijing, 100084 China

Abstract

Java Virtual Machine architecture is a stack based architecture. Because most Java instructions can operate only on the top of the stack, it is difficult to exploit instruction level parallelism (ILP). In this paper, we introduce a new kind of storage, named virtual register (VR), working together with the stack, to provide a simultaneous access mechanism for a wide-issue high-performance JAViR (Java processor Architecture with Virtual Register) design. The VR is transparent to programmers and compilers. At runtime, the dependences of Java bytecode sequence are checked and VR is used for presenting the dependences. Instead of accessing the top of the stack, most arithmetic instructions can get source operands from VR, and write results to VR. Because of the wide accessibility of the VR, it is possible for JAViR to execute more instructions in parallel in a clock cycle than that of stack based Java processors. Simulations show that the new JAViR processor could achieve the performance of average 2.89 EIPC (effective instructions per cycle) with a 16-instruction scheduling window or 4.01 EIPC with a 64-instruction window.

1. Introduction

Java language [1] has been widely accepted as a network computing oriented language with rapid development of Internet and Intranet. Java Virtual Machine (JVM) [2] is a platform for running Java bytecode. In order to implement the standard JVM in the existed computers with different architectures, the computers have to interpret or recompile the Java bytecode with their own instruction sets.

However, because of the differences between the JVM and the existed computer architectures, the running efficiency of Java program is rather low. For the purpose of increasing the performance of JVM, Sun Microsystems has announced hardware solutions, PicoJava 1 and 2 (the core of MicroJava 701) [3] [4]. They are stack based processors, and, they do not address the issue of instruction level parallelism (ILP).

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 issue for high-performance Java processor architecture. Actually, modern high-performance processors all employ the dynamic instruction scheduling for exploiting the ILP, such as out-of-order execution, register renaming, branch prediction, speculative execution and etc.

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

We have investigated the use of virtual register (VR) architecture in Java processor design. The original virtual register architecture [8] is different from the conventional register-oriented RISC architecture. In the VR architecture, the data flow is not represented by register numbers (the register numbers do not appear in instruction set). Instead, the data flow is represented by the dependences of instructions, i.e., the specifiers of source operands refer to the locations (denoted by offsets to the current instruction) where the instructions generate results. Thus this original virtual register architecture does not keep the RISC binary compatibility. When the program is being executed, the required locations for saving instructions' results with VR are dynamically allocated (generation) and released (death) with the control of *reference counters*. A reference counter records the *life-time* of a result. The VR technique can eliminate false dependences caused by static register allocation and reuse of registers.

In this paper, we introduce a Java processor architecture using the VR technique – JAViR (Java processor Architecture with Virtual Register), that keeps the Java bytecode's compatibility. Different from the traditional RISC processors, Java bytecode does not use the register numbers in its

instruction set, this is why the VR technique can be easily applied to Java processor design while keeping the compatibility with the Java bytecode. Simulation results show that the JAViR could achieve the performance of average 2.89 *EIPC (effective instructions per cycle)* when the size of instruction scheduling window is 16 instructions. The EIPC is calculated by dividing the total number of executed effective instructions by the total number of required clock cycles, where the effective instructions are the instructions that enter into instruction scheduling unit (ISU), and are executed by integer units or floating point units. The following instructions do not enter the ISU, and therefore are not included in the effective instructions: unconditional branches, method invocations and returns, stack instructions, and local variable instructions. The JAViR could furthermore achieve the performance of average 4.01 EIPC when the size of instruction scheduling window is 64 instructions.

In section 2, we discuss some JVM features that can be used in the development of ILP. In section 3, we explain how to implement the VR technique in Java processor design. In section 4, we show the simulation results, and in section 5 we draw our conclusions.

2. The Useful Features of JVM Architectures for Exploiting ILP

Although JVM architecture is a stack based architecture and the stack top of Java processor becomes bottleneck for exploiting parallel execution of Java instructions, we found that many other features of JVM specification are actually useful for exploiting the ILP. Such features can be used with VR technique to alleviate the performance restrictions of the stack top.

These features come mainly from the operands' storage methods and operand reference modes. In the following, we investigate the differences of the operand types and reference modes between the conventional processor architecture and the JVM architecture, then we compare their influences on use of VR technique.

In the JVM, the *local variables* and *temporary variables* are located in operand stack. Most Java instructions are temporary variable oriented. This kind of instructions get operands from the stack, operate on them, and then push the results onto the stack.

When the operations require accessing the local variables, *lload*-alike instructions will be used to load the values of the local variables onto the operand stack (temporary variables); and *lstore*-alike instructions store values of the temporary variables from the operand stack into local variables. The JVM employs this method to designate operands with zero-bit field in the arithmetic instructions, thus it reduces the instruction average length, as well as it reduces the transmission time for Java bytecode on network.

The temporary variable access method identifies the life-time period of a temporary variable. When it is pushed onto the stack, the life-time begins; when it is popped from the stack, the life-time ends. According to the situation of the temporary variable generation and usage, it is easy to construct a dataflow relationship among instructions, i.e. the dependency relationship. With the VR technique, it is also easy to map the temporary variable onto a virtual register. However, if we use the traditional superscalar structures to explore the ILP for high-performance Java processors, the pointer of the stack will become a bottleneck for performance improvement. Meanwhile the use of explicitly assigned register numbers in their instruction sets make the determination of a variable's life-time very difficult.

Java is an object-oriented language, *the method invocations* (equivalent to function, procedure, or subroutine calls in structured languages) occur very frequently. The JVM creates a method frame with variable size for each method at runtime. The frame contains the parameters for the method and local variables. In order to eliminate the unnecessary parameter passing between methods, the stack is constructed to allow overlap between the methods, to enable direct parameter passing without requiring any coping of the parameters. This structure overcomes the disadvantages of the register reuse in the global register file architecture (e.g. Power PC, Alpha and etc). Furthermore, it avoids from the shortcomings of rigid structure in cyclic overlapping register windows architecture in which the size of each window is fixed (e.g. SPARC).

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

Usually, RISC architectures have two kinds of addressing modes: direct addressing and index addressing (we don't distinguish between an index register and a base register). However, the real situation is that, in most application programs, there is a large amount of instructions using index addressing mode because of the inability of direct addressing mode for accessing the whole address space. Using index addressing mode makes the instruction scheduling unit encountering a lot of unknown memory addresses during program executions, thus it hampers efficient instruction scheduling for exploiting ILP. There are three kinds of memory addressing modes in JVM:

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.

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.

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 the index to the constant pool.

Cross-boundary access and operation on data with different types are not permitted. Therefore, many dependences of memory reference can be detected according to the instruction opcodes and their data types before all the memory addresses are calculated. The following are the rules for dependency detection: (1) There is no dependency for instructions with different addressing modes. (2) There is no dependency for instructions with different data types; (3) For the object field accesses, if the object reference addresses or field offsets are different, then dependency doesn't exist. (4) For the static field accesses, if the *addresses* are different, then dependency doesn't exist. (5) For the array object accesses, if the array reference addresses or element indices are different, then dependency doesn't exist.

Having these dependency detection rules, a lot of non-dependences can be discovered in advance. It not only improves the effect of instruction scheduling, but also simplifies the design of dynamic scheduling unit for memory reference instructions.

By applying these rules with the virtual register technique, the new Java processor, JAViR, can achieve high-performance while keeping the Java bytecode compatibility.

3. Exploring the ILP by Using VR Technique

For the wide-issue high-performance Java processor design, we introduce the *virtual register* (VR) technique which can provide a wide-access mechanism for parallel execution of multiple instructions. The VRs are not *architectural* registers, that is, the VRs are transparent to programmers and compilers, thus the Java bytecode's compatibility will be kept. At runtime, the dependences of Java instruction sequence are checked and the VRs are used for presenting the dependences. Instead of heavily accessing the top of the stack, most arithmetic instructions can get source operands from the VRs, and can write results to the VRs. Because of the wide accessibility of the VR, it is possible for the new processor to execute more instructions in parallel in a clock cycle than that of solely stack based Java processors. We refer to the Java processor architecture that adopts the VR technique as JAViR (Java processor Architecture with Virtual Register).

Before describing the hardware architecture of JAViR, let us introduce some basic concepts used in JAViR. Instructions are scheduled in an *Instruction Scheduling Window* (ISW). The virtual registers are just labels used to denote the dependency among instructions. Functional units in JAViR are provided with some *reservation stations* (RSs). The VRs are mainly used for broadcasting execution result to the RSs. In some cases, after a result is forwarded to RSs, it will never be used later. So we can eliminate this VR. When a result will also be used later, we have to store it in a location of real storage. First, it is tried to allocate a *physical register* (PR) to store the result. If the instruction which has generated the result is going to leave the instruction scheduling window, the result will be pushed onto the stack. In order to know whether a result will be used later or not, we need a *reference counter* to record the life-time of the result. The initial value of the counter will be set to 1. The *dup*-alike instructions increase the counter, it means that one result will be referred twice. Once a reference operation happens, the counter is decreased. If a counter reaches zero, then the result will never be used later (dead). In this case, the VR or PR where the result stays can be released. So the reference counters are mainly used to maintain the VRs and PRs.

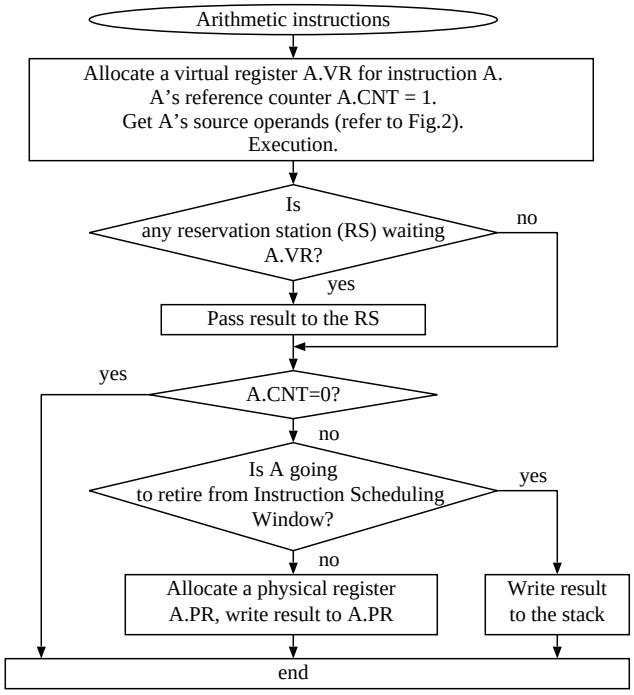

Figure 1. Execution of arithmetic instructions

Fig. 1 illustrates the execution procedure of an arithmetic instruction. A typical arithmetic instruction pops two source operands from the top of the stack, operates on them, and pushes the result onto the top of the stack. By using the VR mechanism, instead of pushing result onto the top of the stack, first, a VR is allocated for the instruction. The

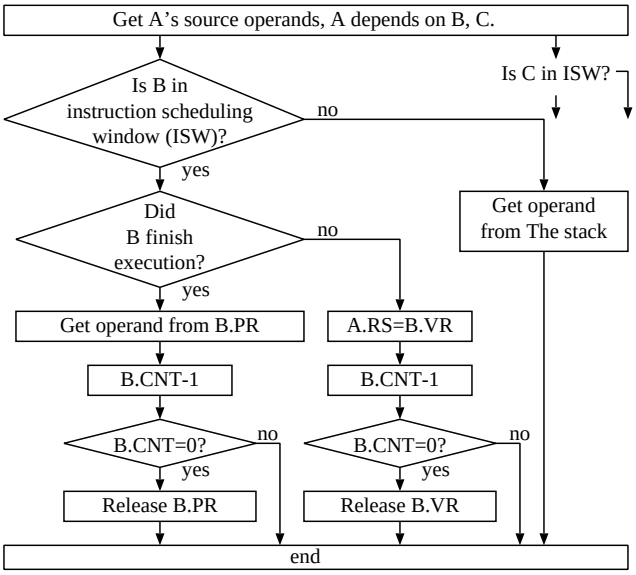

Figure 2. Getting source operands

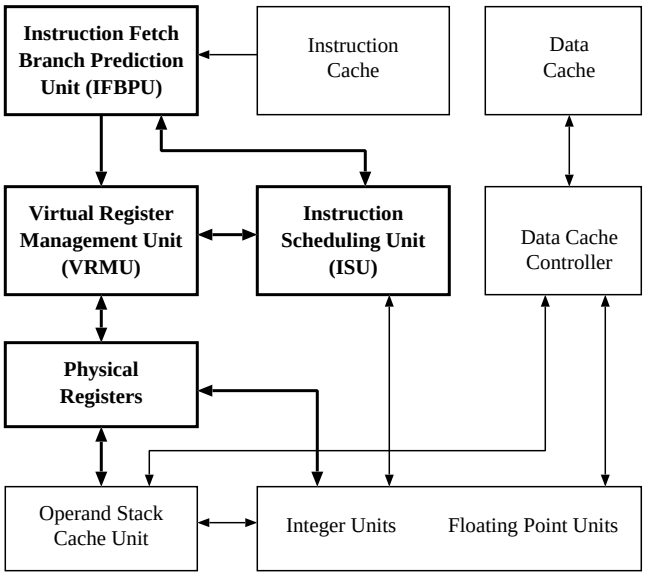

Figure 3. Java processor with VRs

instruction also tries to get source operands from VRs or from PRs (see Fig. 2). Once the VR or PR is accessed, its reference counter will be decreased. The VR or PR will be released when the counter reaches zero. If the operands are in neither VRs nor PRs, the instruction can get them from the stack, as the same as the JVM does.

Once the instruction has produced the result, it will broadcast the result to the RSs where the corresponding functional units are waiting the result. If the reference counter for the VR is not equal to zero, a PR should be allocated and the result should be stored onto the PR, or the result should be pushed onto the stack, based on whether the instruction will leave the instruction scheduling window.

Fig. 3 shows the basic block diagram of the JAViR. It is very similar to PicoJava, but it differs primarily in the following new units:

Instruction Fetch and Branch Prediction Unit (IFBPU). It provides a continuous instruction stream for the subsequent related units. When it encounters a branch instruction whose branch target address can be determined (such as unconditional branch, method invocation and return), it can transfer to the correct branch target timely to continue fetching instructions; if it encounters a conditional branch, it predicts the branch target by using Branch Target Buffer [6], fetches instructions from the predicted target, and informs the instruction scheduling unit to execute them speculatively. If the branch is mispredicted, it informs the instruction scheduling unit to cancel the speculatively executed results, and fetches instructions from the correct branch target.

Virtual Register Management Unit (VRMU): It receives instructions from the IFBPU, and analyzes instruction dependences according to the instruction types and their operand references. It may allocate or release virtual and physical registers based on the "generation" or "death" of variable values. Meanwhile, it dispatches the information of the dependency relationship to the instruction scheduling unit.

Instruction Scheduling Unit (ISU): It performs instruction issuing, completion and committing according to the dependences among instructions.

By analyzing each instruction type and operand references, the relationship diagram among the instructions can be constructed. The VRMU accomplishes this task on the basis of the relationship diagram analysis. For different types of instructions, the VRMU adopts the different processing approaches. The JVM instructions fall into the following categories:

Arithmetic Instructions: The common feature of this kind of instructions is that the source operands are popped from the top of the stack. After operation, the results are pushed onto the stack.

Stack Management Instructions: These are specially used in operations such as duplication, removal, and swap of data in the stack.

Local Variable Instructions: These are used to transfer values between local variables and the operand stack.

Memory Reference Instructions: These instructions are used in accessing fields of objects and array elements which are located in memory. They load fields or the array elements and push them onto the stack, or store the values in the top of the stack to memory locations.

Branch Instructions: There are two kinds of branch instructions: unconditional branch instructions and conditional branch instructions.

Method Invocation Instructions and Return Instructions: They are used in method invocations and returns from method.

Others: The other instructions include monitor instructions and exception handling instructions, these can be considered as special kinds of branch instructions.

Among the instructions mentioned above, the unconditional branch instructions and the method invocation and return instructions are processed inside the IFBPU; they will not be sent to the VRMU. We'll discuss how the VRMU handles the other type of instructions.

In order to analyze the dependences among these instructions according to the situation of accessing operands for each instruction, the VRMU needs an *operands tracking stack* (OTS) to record the information about the source operands are produced by which instructions and where they are located currently.

For arithmetic, memory reference, and conditional branch instructions, according to their source operands and information in the OTS, we can know how to find out which instructions they depend on. These kind of instructions can get their source operands from the virtual/physical registers instead of that from the operand stack because the dependent instructions may write their results into the virtual registers (reservation stations) or physical registers.

With regards to the local variable instructions and stack instructions, what they operate are to move, copy, pop, and swap the data of the operand stack. Therefore, these instructions need not enter to the ISU; the only actions for them are to update the contents of the OTS, to increase/decrease reference counters, and to allocate or release virtual/physical registers.

Notice that it is not necessary for the OTS to record the dependency information for all the locations in the operand stack. Because the dependences only refer to those instructions which are staying in the instruction scheduling window; it is not necessary to indicate the dependency of an instruction with the others which already committed from the instruction scheduling unit. When an instruction quits from the instruction scheduling unit, those items in the OTS related to this instruction will be deleted. The size of the OTS is proportional to the size of instruction scheduling window.

For the purpose of exploring ILP and eliminating the bottleneck caused by the operations on the top of the stack, in our JAViR, the local variables and temporary variables are not stored in the stack, instead, they are stored in the virtual/physical registers which can be allocated and released dynamically.

Each virtual register is identified by a VR number. It is different from the physical register, the VR number does not represent a storage location, it only indicates a specified data (functional units are provided with reservation stations). The dependences among instructions are represented by the VR number. The meaning of allocating and releasing a virtual register just lies on the allocation and release for a VR number, it is not related to the physical register. When an instruction execution is finished while its reference counter does not reach zero, a physical register is allocated to save the result.

4. Simulations Results

For the purpose of checking and measuring the scheduling effects of the JAViR with various possible configurations, our simulator focuses on the issues of the usage of the virtual/physical registers and the performance of effective instructions per cycle.

We make the following basic assumptions. First, the data cache is non-blocked, 32KB, 4-way set associate, 32-byte line size, and 5 cycles for swapping a line with outside cache/memory in cache miss. Second, the branch prediction buffer employs a full associate table with 512 entries and two-bit counters for prediction. Third, the JAViR functional units have three kinds of execution pipelines: integer, branch, and float-point, used for executing integer, conditional branch, and floating point instructions, with the execution latencies of 1, 1, and 3 cycles respectively, and all are fully pipelined (the floating point division may need more cycles for execution, but here we assume the latency for it is also 3 cycles).

Our simulation studies the effective instruction execution per cycle and the ratio of the required physical registers to the virtual registers. We carried out the simulation with different size of instruction scheduling windows (16, 64, and 256 instructions respectively).

Since most of the current benchmark programs are written by structured languages, which are not proper for our simulations, we rewrite/select the following test programs.

simul: This is the background part of our experiment system which is used for this simulation work. It was originally written in C++, we rewrite it in Java for the purpose of testing. In our experiment, we use it to simulate more than 10K Java bytecode's execution on the JAViR. The total executed instructions are around 35.2 million instructions.

javac: The Java compiler in SUN JDK 1.1: sun.tools.javac.Main. It is a very typical object-oriented program. We use it to compile the Java.lang.string source code with optimization option. The total executed instructions are around 48.9 million instructions.

javacc: A program for generating compiler or interpreter for Java source program on the basis of semantic formal description (similar to yacc). It is also a typical object-oriented program. We use it to process the formal descrip-

tion of HTML language. A total of 50 million instructions was sampled and simulated.

li: A Lisp interpreter. It is the 130.li test program in SPEC95, written in C language. We rewrite this program in Java by using object-oriented technique. In our experiment, we use it to run 6-queen problem (test.lsp in 130.li of SPEC95). The total executed instructions are around 24.5 million instructions.

grep: A text pattern matching program in UNIX system. It was originally written in C, we rewrite it in Java, but we don't employ the object-oriented technique. In our experiment, we use it to search a complex pattern from a text file. A total of 50 million instructions was sampled and simulated.

linpack: It is a linpack Java version for measuring the JVM performance. It is not an object-oriented program. The total executed instructions are around 8.1 million instructions.

Table 1 shows the results of measured EIPC (effective instructions per cycle) and the use of virtual/physical registers. The 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 instructions that enter into the instruction scheduling unit (ISU); the following instructions need not enter to ISU and therefore are not included in the effective instructions: unconditional branch, method invocation and return, stack instructions and local variable instructions (transmissions between local variable and the top of the stack). The effective instructions occupy about a dynamic portion of 30-50% of all the instructions.

In Table 1, *PR/VR* represents the ratio of the required physical registers to the generated virtual register. We allocate a virtual register for each effective instruction and assign a reference counter for it. When the execution of this instruction is finished but the counter does not reach zero, we assign the virtual register to a real physical register. For those instructions whose VR reference counters contain zero, the physical registers are not necessary to be assigned. *UseVR* represents the proportion of the virtual register reference (only those virtual registers having no their corresponding physical registers) to the total reference of the source operands. *AvgPR* represents the average number of physical registers which are used simultaneously. And the *MaxPR* is the maximal number of physical registers which are simultaneously required.

As shown in Table 1, the EIPC becomes better as the scope of instruction scheduling increases. But *li* and *grep* have lower performance than others. This is because the low correctness of branch prediction for these two programs (72% and 74% respectively). If we adopt the loop unrolling [5] statically and/or the two-level prediction mechanism [7] dynamically, the performances are expected better

		Test Programs					
SR	Items	simu	javac	javacc	li	grep	linpack
16	EIPC	3.27	3.31	3.44	2.83	2.82	2.11
	PR/VR	20%	46%	46%	53%	51%	45%
	UseVR	62%	60%	57%	67%	50%	45%
	AvgPR	1.32	2.69	2.07	1.55	1.11	4.05
	MaxPR	10	11	10	10	9	8
64	EIPC	5.12	4.09	4.60	3.23	2.91	5.44
	PR/VR	16%	36%	39%	49%	48%	8%
	UseVR	73%	70%	70%	74%	52%	60%
	AvgPR	2.24	4.85	3.26	1.55	1.11	4.05
	MaxPR	16	39	34	10	19	17
256	EIPC	5.49	4.21	4.78	3.23	2.91	6.86
	PR/VR	16%	33%	39%	49%	48%	4%
	UseVR	76%	76%	72%	74%	52%	63%
	AvgPR	3.11	7.44	4.17	1.86	1.11	8.69
	MaxPR	51	100	77	71	38	43

SR: Scheduling range (instructions)

Table 1. EIPC results and VRs/PRs usages

(we haven't done yet such simulations). Consider together with that the effective instructions occupy about a dynamic ratio of 30-50% of all the instructions, the IPC including the total instructions will be two or three times compared with the EIPC. This IPC can be regarded roughly as the speedup comparing with a stack based Java processor.

More than 50% virtual registers don't need its corresponding physical registers, and the UseVR also has high percentage. This means that most operand references need not be stored in the stack, nor in the physical registers. The number of the simultaneously required physical registers becomes larger as the size of instruction scheduling window increases. But it is less than the size of instruction scheduling window. In a real implementation, we can fix the number of physical registers. For the executions of various programs, when the physical registers are not enough, we can use the traditional operand stack cache, or the data cache by means of saving/restoring operations in background.

In addition, we evaluate the benefits of the memory reference features of Java Virtual Machine. We compare the JAViR performance by using the following three methods for the memory reference dependency detection.

RISC method: the memory reference dependences are detected only after the physical addresses are calculated. It is equivalent to the memory references in the conventional RISC architectures, where most memory instructions require calculations for their effective addresses.

JVM method: the memory reference dependences are detected by the method described in this paper.

Ideal method: the memory reference dependences can be detected in advance in any situation. This is just used as a reference base for performance comparison.

Table 2 lists the EIPC simulation results by using the

different methods for the memory reference dependency detection. JVM method offers slighter improvement comparing with the traditional RISC method, but in the RISC architecture, for implementing the dynamic scheduling of the memory instructions, it requires much more complex hardware than that of JVM method. The results show that the JAViR could achieve the performance of average 2.89 and 4.01 EIPC when the sizes of instruction scheduling window are 16 and 64 respectively. Further increasing the window size to 256 get few performance improvement (4.20 EIPC on average).

		Test Programs					
SR	Items	simu	javac	javacc	li	grep	linpack
16	RISC	3.25	3.30	3.43	2.83	2.82	2.11
	JVM	3.27	3.31	3.44	2.84	2.82	2.11
	Ideal	3.27	3.31	3.44	2.86	2.82	2.11
64	RISC	5.03	4.05	4.52	3.21	2.90	5.44
	JVM	5.12	4.09	4.60	3.23	2.90	5.44
	Ideal	5.14	4.10	4.61	3.25	2.90	5.44
256	RISC	5.33	4.12	4.66	3.21	2.90	6.86
	JVM	5.49	4.21	4.78	3.23	2.91	6.86
	Ideal	5.51	4.22	4.79	3.25	2.91	6.86

SR: Scheduling range (instructions)

Table 2. EIPC results with different detections

5. Conclusion Remarks

Generally, it is difficult for a stack based machine to exploit instruction level parallelism due to the frequent accesses to the top of the stack. But the Java Virtual Machine Specification brings part semantics of high-level language into architecture level. By using this feature, we introduced a new Java processor architecture, JAViR, with a new kind of storage, named virtual register, to make the ILP exploitation more effective.

To delay the merging procedure between the instruction set and the hardware resources into execution stage can reduce the false dependences owing to the earlier merger, and can reduce the complexity of hardware for detecting the dependences so that processor can more effectively exploit the ILP, and consequently improve the performance. Meanwhile it also provides the flexibility of processor design.

In the JAViR design, the most complicated part of hardware implementation is the Virtual Register Management Unit (VRMU). The complexity can be expressed by $O(n^2)$, where n is the number of instructions that the VRMU can process simultaneously.

The virtual and physical registers are transparent to programmers and compilers. It means that the Java bytecode's compatibility can be kept. At runtime, the Java bytecode is transformed to the operations on virtual/physical registers. Because the life-time of variables in Java bytecode is quite short, most operation results need not be stored; they are forwarded to the instructions, i.e., the only place where the results are required.

Various simulations are performed in order to predict the performance improvement compared to the traditional stack based processor architecture which does not deal with the issue of exploiting the ILP. Our results show that the new processor with a 16-instruction scheduling window could achieve the performance of average 2.89 EIPC or 4.01 EIPC with a 64-instruction window.

The proposed JAViR in this paper is only one of the possible processor architectures that can exploit Java bytecode ILP with the combination of the semantics of high-level language, instruction set, and processor architecture. We believe that many favorite high-performance Java processor architectures will be investigated in order to satisfy the increased performance requirements for the rapid development of network computing.

References

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

[2] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification", Addison-Wesley, 1997, also see http://www.javasoft.com/docs/books/vmspec/index.html

[3] Sun's WhitePaper, "PicoJava I Microprocessor Core Architecture", http://www.sun.com/microelectronics/whitepapers/wpr-0014-01/

[4] Jim Turley, "MicroJava Pushes Bytecode Performance – Sun's MicroJava 701 Based on New Generation of PicoJava Core", *Microprocessor Report*, November 17, 1997. pp9-12.

[5] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[6] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer*, January 1984, pp6-22.

[7] T. Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-level Adaptive Branch Prediction", *Proc. 19th Intl. Sym. on Computer Architecture*, May 1992. pp124-134.

[8] H. Liao and S. Li, "Virtual Register Structure", *Chinese Journal of Computers*, Vol.19, No. 11, Nov. 1996. pp801-809.