Exploiting Dataflow to Extract Java Instruction Level Parallelism on a Tag-based Multi-Issue Semi In-Order (TMSI) Processor

Hai-Chen Wang, Chung-Kwong Yuen

Dept. of Computer Science
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543
{wanghaic, yuenck}@ comp.nus.edu.sg

Abstract

To design a Java processor with traditional modern processor architecture, the Instruction Level Parallelism (ILP) is not readily exploitable due to stack operands dependencies. This paper presents a dataflow-based instruction tagging scheme. With instruction tagging, the independent bytecode instruction groups with stack dependences are identified. The different bytecode instruction group can be executed in parallel because there are no stack dependences among them. With the instruction tagging scheme, we propose a tag-based multi-issue semi-in-order (TMSI) Java processor. The processor takes advantage of instruction-tagging and stack-folding to generate the tagged register-based instructions. When the tagged instructions are ready, they are bundled out-of-order depending on data availability to form VLIW-like instruction words and issued in-order. To achieve high performance, a VLIW engine is employed. We have conducted some experiments in our TMSI simulation environment using SPECjvm98 and Linpack workload. The results indicate that the proposed processor has good performance gain.

1. Introduction

The feature of the machine independence distribution format of programs with the Java Virtual Machine (JVM) [5] makes Java technology extremely popular, which is found in a variety of systems, ranging from embedded systems to high performance server systems.

Java bytecodes may be executed on various platforms by interpretation or Just-In-Time (JIT) compiling. The first Java virtual machine (VM) available was interpreter-based, but it was neither efficient nor well-suited to high performance applications. The JIT compiler translates bytecodes to the native code of the host machine dynamically. Several variants of the JIT concept [22, 23] have been proposed. However, the JIT approach incurs runtime overhead in translating bytecodes to native code, although acceptable performance for Java applications can be provided. To further improve the performance of JIT, adaptive dynamic compilation [24] is proposed. In this scheme, methods that are most heavily used are compiled and optimized in traditional compiler technique in order to obtain more efficient native machine code.

Hardware processors to execute bytecode directly are also becoming popular. The designs of Java processors, such as picoJava [10], are mainly based on stack architectures, and the Java VM is used as their instruction set architecture. A major issue in this architecture is the existent limitation of ILP by the stack dependence.

Several techniques to extract ILP in Java bytecode have been investigated [11, 13, 17, 18]. Stack operation folding is one technique to reduce the limitation by converting a set of bytecodes into a register-based instruction [8, 9, 10, 11, 12, 14]. In Sun's picoJava-II processor, simple instruction folding in hardware is done by using pattern matching at decode stage of its pipeline [10, 11], and the stack folding is supported by the stack cache as a register file for parallel access of stack operands to eliminate redundant stack operations. More sophisticated folding techniques, such as nested folding [8, 9, 12, 14], may further reduce stack operand dependence but need complex hardware to support. Combining multiple in-order issue with stack folding can further exploit ILP [11], but ILP is limited when execution is restricted in program order. To

1-4244-0054-6/06/$20.00

support out-of-order execution, SMTI [17] is proposed with software involved to extract independent bytecode trace and implement bytecode folding, but special fetch logic is needed to identify independent traces from instruction cache.

To meet the requirement of high-performance network application with Java, thread level parallelism (TLP) is exploited to extract coarse-grained parallelism. Sun's MAJC processor adopts a vertical multithreading technique, in which Java methods are treated as a thread in hardware and speculative execution of multiple threads is included to exploit TLP [18]. But MAJC needs a JIT compiler to convert bytecodes to the native codes. The Java Multi-Threaded Processor (JMTP) architecture is a similar hardware implementation, which is a single-chip CPU containing an off-the-shelf general purpose processor core coupled with an array of Java Thread Processors (JTPs) [13]. But an intelligent compiler is needed to identify the set of concurrent threads that can be forked as JTP threads. DAISY [19] is another VLIW architecture, which combines JIT with native compilation techniques by appropriate hardware primitives designed to execute Java efficiently. It dynamically translates Java bytecodes with JIT into VLIW instructions and exploits a VLIW engine. This approach can take advantage of the increased ILP possible in VLIW machines to achieve high performance [6].

There are another two dynamic-translation-supported Java processors: Femtojava [3] and Delft-Java [4]. The FemtoJava processor is a stack-based architecture with replicated functional units and instruction decoders, and employs a VLIW as its execution engine. In FemtoJava [3], the bytecodes in the entire Java program are divided into the instruction groups, and the instructions within the same group are translated into VLIW word to be executed. The grouping algorithm is to find those instructions that depend on the result of the previous one, and group them in one instruction block. The Delft-Java [4] processor provides hardware assisted dynamic translation, and the bytecodes are translated on-the-fly into the Delft-Java instruction set. Hardware support for Java language constructs is incorporated into the processor's ISA. This allows application level parallelism inherent in the Java language to be utilized as instruction level parallelism.

1.1 Overview of Our Work

Dataflow [1, 2] is an alternative architecture that directly contrasts with the traditional von Neumann architecture. Although dataflow has been studied for more than two decades, actual implementations of the model have failed to deliver the promised performance. Most modern processors utilize complex hardware techniques to detect data and control hazards and dynamic parallelism in order to bring the execution engine closer to an idealized dataflow engine [21]. In this paper, we propose the architecture of Java ILP processor, in which dataflow is exploited to identify stack dependence, reduce redundant stack operations and extract ILP in Java programs. In the proposed processor, stack instruction tagging may enable out-of-order instruction execution and extract most of ILP in Java programs. After bytecode tagging, stack dependences are identified by tags, and the availability of operands (or tags) is used to help issue instructions similar as in dataflow machines. Within an instruction window, the instructions without data dependences may be issued simultaneously and executed in parallel. By combining the EPOC folding [12] method with the tagging scheme, we implemented a new tag-based POC scheme. By means of bytecode folding, stack instructions are converted to tagged register-based instruction formats (two source operands, one destination operand), then bundled and formed as VLIW wide words to be executed on a VLIW engine.

We executed the Benchmark suite of Spec JVM 98 [16] and Linpack [20] to conduct the performance evaluation and achieved the performance gain in an average ILP speedup of 59% over the base in-order single-issue Java processor.

1.2 The organization of the Paper

The remainder of this paper is organized as follows. Section 2 introduces our tag-based dataflow scheme (TDS) with one example. Section 3 discusses the bytecode folding scheme implemented with instruction tagging in the processor. Section 4 illustrates the architecture of our proposed Java ILP processor and some related issues. The performance evaluation and results are presented in Section 5. Section 6 summarizes our work.

2. Motivation -- Exploiting Dataflow in Stack Architecture

In some ways it is easier to parallelize execution of stack programs on stack machines, because those operands on execution stacks are erased once they are used by an operator. Hence, an operand only needs to be supplied to one operator which can be uniquely identified by a tag. Once a tag is used, its new result is immediately discarded without being actually stored into the stack; in contrast to GPR machines, new register contents must be written back to physical registers from the reorder buffer even if they may already have been superseded by later writes. To

exploit the feature of the tag in stack machines, we propose a tag-based dataflow scheme to extract ILP and implement out-of-order bytecode instruction execution in stack programs. This scheme exploits a stack of tags rather than a stack of values.

We explain our scheme as follows. If we want to compute an expression: g = a*b+(c+d), its corresponding stack program can be:

- LD A, LD B, MUL, LD C, LD D, ADD, ADD, ST G

In Table 1, we will describe how the parallelism can be achieved after renaming stack location with tags, using Operand Tag Stack (OTS) to identify source operands with operators that consume them. Here, the renaming unit uses a new tag for every instruction that leaves a result on the stack instead of an operand value. The tags on OTS are used for attachment to a later instruction that consumes the operand. As shown in Table 1, the tagged instructions are dispatched after an instruction is executed in a load/store or ALU unit and the result is delivered to the later instruction that carries its result. For example, the first two *loads* deliver their operands to the *mul* operator, and the last two *loads* to the *add* operator, then they will execute and deliver their results to the second *add* operation, in the same manner as in superscalar machines. From this example, we can see that using a stack of tags makes it easy to attach operand tags to an operator.

Table 1: A sample of stack renaming scheme

Instruction	Naming unit	Operand Tag stack (OTS)
1 load a	T1 load a	T1
2 load b	T2 load b	T1 T2
3 mul	T3 mul T1 T2	T3
4 load c	T4 load c	T3 T4
5 load d	T5 load d	T3 T4 T5
6 add	T6 add T4 T5	T3 T6
7 add	T7 add T3 T6	T7
8 store e	T8 store T7 e	

The proposed tagging method is data-driven. The tags are organized as a reorder buffer in order that can be reused and dynamically assigned to the later coming instructions after they are retired. The single tag entry is composed of the instruction op-code, status bits and a register number which points to the "destined" physical register in the register file. Once an operator instruction is tagged, it will identify its operands by tags. The tag can be seen as a data token in tagged dataflow machines [1] where the flow of data token activates instructions' execution. In the process of instruction tagging, a data dependence graph (DDG) is generated dynamically and instruction execution may follow the graph. As in dataflow machines, the availability of tagged operands of an instruction triggers its execution and the tagged result is passed directly as data tokens between instructions. The tagging execution scheme supports explicit out-of-order instruction execution.

Unlike the dataflow execution paradigm, instruction execution is controlled by the scheduler in our implementation. When an instruction is ready for execution, it is first inserted to the ready-instruction queue and then scheduled to execution. The process of instruction issue is controlled by a scheduler. Hence the availability of operands of the ready instructions did not immediately "trigger" them to be executed. This execution paradigm is similar to the scheduled dataflow [21]. With instruction scheduler, instructions are easy to be synchronized and managed. Designed in this way, the complexity of instruction issue logic [21] is reduced, out-of-order execution can be implemented and fine-grained parallelism is achieved as in dataflow without a need of the dataflow programming paradigm.

3. Stack Instruction Folding

Stack operation folding is one technique that eliminates stack operand dependency by converting a set of bytecodes into a register-based instruction [8, 9, 12, 14]. It can save pipeline cycles by eliminating redundant stack operations. We exploit the tagging technique and EPOC scheme [12] to implement a tag-based POC folding scheme. In this scheme, all bytecode instructions are categorized into three kinds of role in instruction folding: *Producer, Consumer and Operator* (POC) as in [12]. The *Producer* instructions push data form local variable or constant onto operand stack in a single cycle. The *Consumer* instructions pop data from operand stack and store the data into local variables. The *Operator* instructions pop data from operand stack, execute on corresponding functional units, and then push the result back to operand stack. We add another foldable instruction type -- *IP* (intermediate producer), which indicates that the tagging instruction will produce a result and the result will be used by its consumer instruction. The IP instruction type may help to implement nested instruction folding. To identify four bytecode types, two extra bits are needed in pre-decoding stage to help instruction decoding.

Table 2 shows a tagged register-based instruction format we proposed, which is similar to the RISC instruction, and easy to implement in modern high performance processors. Here, *opcode* is the bytecode operation code, *Src1,Src2* are the fields of source operand tags, and *Dest Tag* is the field of destination tag. The Java bytecodes has less than 255 instructions and the length of

bytecode is 8 bits in Java specification. If tagging unit is set in 64 entries, it is enough to use 8 binary bits to address the tag numbers in the field of source operand and destination operand in Table 2.

Table 2: Tagged register-based instruction format

Opcode	Src1 Tag	Src2 Tag	Dest Tag

The instruction tagging has analyzed and identified data dependences among bytecode instructions. The proposed instruction folding scheme will exploit the results of instruction tagging to simplify the implementation. With instruction tagging, all *Operator* instructions have found their dependent operand tags and their dependent information in tags are stored in Naming Unit illustrated in Table 3. In Naming Unit, an *Operator* bytecode instruction entry will store the tag number of its left operand and right operand. The tag numbers can be directly used to generate register-based instruction by folding logic. According to the POC type and instruction dependent information, the folding logic can generate tagged register-based instructions and stores them in folded instruction buffer (FIB). Only those *Operator* instructions can generate corresponding register-based instructions. For *Consumer* instructions, folding logic will check the POC type of its dependent instruction. If the instruction type of the dependent instruction is *Producer or IP*, only a register-copying is needed; if it is *Operator,* the previous generated tag-based instruction's destination tag will be modified to the tag of the *Consumer* instruction. Because bytecode instructions are tagged in sequential and the tagging process follows a stack machine's behavior, the correctness of our scheme could be guaranteed. The detailed instruction folding method can be referred to [12].

Table 3: Sample of Java instruction folding

Java Bytecodes	Naming Unit	Operand Tag stack	Folded Instructions
1 iload a	T1 load a	T1	
2 iload b	T2 load b	T1 T2	
3 imul	T3 mul T1 T2	T3	i1: imul, T1,T2, T3
4 iload c	T4 load c	T3 T4	
5 iload d	T5 load d	T3 T4 T5	
6 iadd	T6 add T4 T5	T3 T6	i2:iadd, T4,T5, T6
7 iadd	T7 add T3 T6	T7	i3:iadd, T3,T6, T8
8 istore	T8 store T7 e		

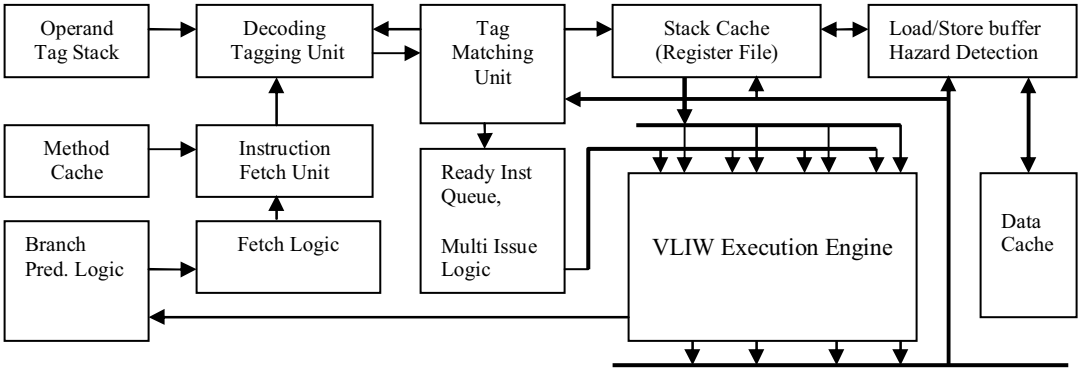
Table 3 illustrates a basic instruction folding process for a nested bytecode instruction sequence. Column 4 demonstrates the newly generated tagged register-based instructions after instruction folding. They are folded into three instructions, i1, i2, and i3 (i1, i2, i3 are only for explanation, not be included in new instructions). The tag number T3 and T6 are reused and treated as the *IP* type in instruction i3. In line 7, the *destined tag* (underlined) is changed from original Tag 7 to Tag 8 (the tag number of the *istore*). From Table 3, we can see that without instruction folding, 8 cycles are needed to sequential execute the listed bytecodes. With instruction folding, only 3 cycles are needed. If multiple-issue is supported, only 2 cycles are needed (because tag-dependence exists in the three new generated instructions).

4. Tag-Based Multi-Issue Semi In-Order (TMSI) Processor

A Java processor is able to execute Java bytecode instructions directly in hardware, which makes it possible to entirely bypass the need for dynamic translation and reestablish a simple, direct execution model for Java code. We propose a Java ILP processor with a six-stage pipeline, including instruction-fetch, decode, stack folding, issue, execute and commit stages. Because instruction folding is on the critical path of the pipeline [11], one individually decoding stage for instruction folding is needed in the design of our pipeline. With instruction folding, the bytecode instructions are converted to the tagged register-based instructions. And the data dependences among the tagged instructions are identified by the tags. In the processor, Tagging Unit (TU) and Tag Matching Unit (TMU) are responsible for controlling the instruction tagging, matching tags and updating the status of tags. When a bytecode instruction enters the decoding unit, TU assigns a tag to it. After the instruction execution is completed, the related tags are released to the free tag pool by TU, where the tags can be reused.

When the operands of a tagged instruction are ready, the tagged instruction is added to the ready instruction queue for scheduling. The multiple tagged instructions are bundled out-of-order depending on data availability to form VLIW-like instruction words, the instruction bundles are put in the issue buffer and issued by the Scheduler in-order. Although the instruction bundles are issued in-order, at the time they are bundled, they may be not in program order. Hence, our processor worked in *multiple-issue semi in-order style*. The Stack Cache as a register file is provided [9] in the processor to eliminate inefficiencies typically associated with stack-based instruction processing, and the temporary results in the instruction execution are stored in it. The schematic block diagram is shown in Figure 1.

Figure 1: The proposed TMSI processor architecture



4.1 Instruction Fetch and Decode

The Java bytecode instructions are fetched from a method cache. The bytecode fetch logic controls the instruction fetching from the same bytecode method according to the program counter values. After fetching all the instructions of a basic block, fetch logic selects the next basic block as predicted by the branch predictor. With Operand Tag Stack (OTS) support, the Decoding / Tagging Unit (DTU) and TMU are together to handle both instruction tagging and folding. When one bytecode is decoded, an entry in TMU is allocated, and the tag number of the entry is assigned to the bytecode. The policy of tag allocation is first come first service (FCFS).

The entry in TMU is to hold the control-related information, which contains the left / right operand tag number, the valid bit, the status bit and the address of a physical register in Stack Cache. A mapping table is used to manage the mapping from tags to registers. The Register File (RF) is a global temporary storage, responsible for storing stack operands and local variables to speed up memory access. The organization of TMU is similar to that of a reorder buffer in superscalar processor [26], but TMU holds more functions than a reorder buffer does. Whenever a result is produced in RF, the corresponding tag number is simultaneously sent to TMU to update the instruction status and wake up the waiting consumer instructions.

The operands for the instructions may be loaded from stack cache (register file) or the data cache. LV variables and intermediate results are both allocated on the register file. The stack intermediate results generated by ALU instructions can be directly written to the register file in parallel. The memory load operation, if it does not exist in load buffer, must load from the data cache.

4.2 Instruction Issue and Schedule

In Java JVM, some temporary data and intermediate values are stored in LV area to speedup their access. In our processor, LV variables are resident on the stack cache. When a folded instruction finishes execution, its result is first written back to the physical register the “destined” tag points to. Within a basic block, only the last write, e.g. istore_x, updates the corresponding LV variable if there are multiple writes to the same LV. This is similar to the register renaming and resolves data conflicts in case of multiple-writes to the same LV variable. With instruction tagging, WAR (write after read) and WAW (write after write) data dependences are removed, because both operations will access different registers. Thus only the real data dependence – read after write (RAW) needs to be considered. When a RAW conflict occurs, e.g. the later read and previous write access the same LV variable, the issue logic may guarantee the later read instruction cannot be issued until the previous write instruction completes.

The memory access instructions, such as iastore, iaload etc, may issue out-of-order. Memory dependences between instructions are detected at run-time by the memory-hazard-detection logic, which consists of a load buffer, a store buffer and address comparator circuits [26]. Store addresses are buffered in an address queue (FIFO). The hardware will check each issued load to determine if an earlier in-flight store instruction was issued to the same physical address, and if so, use the value produced by the store. While each issued store will check to see if a later load to the same physical address was previously issued, and if so, take corrective action.

4.3 Instruction Execution and Commit

In order to achieve high performance with reduced hardware complexity, a VLIW execution engine is employed in our processor. The ready tagged instructions are first dynamically packed into VLIW-like wide words, put into an instruction issue buffer, then issued to the functional units on VLIW engine through a Quasi-crossbar [25]. The bundled instructions are issued in strict locked-step as in VLIW machines.

When a bytecode instruction completes, the result will be written back to register file or load/store-buffer if it is a memory access instruction, and the status of the related tags are needed to be updated. When the status updating of a tag is completed, and if it is no-longer used in the nearly future we say the tag is "committed" and can be returned to the free tag pool for later uses. A tag is alive from the time it is assigned to an instruction until the instruction is "committed". If a bytecode instruction will produce an intermediate result after the instruction's execution and the result will be used by its later instruction, the related tag will be reused. In this case, the reused tag is often used as a producer to provide data for later consumer instruction. The reused tag will be retained until its result is consumed by its later consumer instruction. When a tag is no longer used, it will be removed and released for later retrieval.

5. Performance Evaluation

5.1 Experimental Methodology

We have done a simulation study on our proposed architecture. A trace-driven simulator was developed to model our tag-based processor architecture. The simulator accepts bytecode traces extracted from the execution of the benchmarks programs on the modified open source Java VM interpreter Kaffe [15]. The bytecodes are scheduled and run based on pipeline stages cycle-by-cycle. The behavior of tags follows our model. In the simulator, we assume that TMU has 64 tag entries. The size of physical register file is larger than 64, because register file not only provides tag-mapping registers but also contains the LV storage area.

We used SPECjvm98 [16] and Linpack [20] benchmarks. The benchmarks in SPECjvm98 were run with the s1 data set and the *mtrt* benchmark program is single-thread version. In the experiments, instruction schedule was limited within a basic block, only when all the instructions within a basic-block were issued can the instructions in the next basic-block be scheduled, but instruction prefetch is supported. The branch predictor used is a static predictor as in picoJava-II [11], and has a penalty of 3 cycles for mis-predicted branches. An ideal instruction cache was assumed.

To study the gain in ILP and performance speedup with TMSI processor, we ran two types of simulation: one in which every bytecode instruction assumes at a single cycle latency, and the other in which the different bytecodes take different latency according to the picoJava specification. ILP gain is useful for determining ILP speedup from the viewpoint of multiple instruction issue, and the latter simulation is helpful to demonstrate the actual speedup compared with the existing architecture, which indicates the actual performance gain in TMSI processor. Data caches with 100% hit rate are assumed in our experiments.

5.2 Improvement in Exploitable Parallel Execution

To detect the proportion of parallel execution instructions in our processor, we relax the resources constraints on the number of execution units and set the issue rate at four. When the execute stage is fed all the instructions within the instruction issue window, the processor could potentially execute at most four of them in parallel if there are no dependencies and resource constraints. If there are stack dependences or LV dependences, the following instructions will be executed in the next cycle. We execute different benchmarks on the simulator with above constraints and assume the issue-window holds 64 entries.

Table 4: Percentage of instructions executed in parallel in our scheme

BenchMarks	Instructions executed in parallel (percentage)			
	1	2	3	4
compress	67.37	15.43	10.78	6.42
Db	79.97	14.98	3.78	1.27
Jack	79.54	14.22	3.89	2.35
javac	72.85	21.87	4.24	1.04
jess	81.51	13.47	3.26	1.76
mpegaudio	43.26	16.53	6.78	33.43
mtrt	87.92	9.67	1.55	0.86
Linpack	69.18	16.10	0.38	14.34

Table 4 shows the proportion of instruction execution in parallel. Let us compare it with that reported in the previous research work of in-order multi-issue of the folded Java instruction execution [11]. In [11], by using stack disambiguation technique, only a small number of three-instruction-groups are issued in parallel and no four-instruction-groups are issued in parallel. However, the results of our experiments show that the percentage of issued three-instruction-group is from 0.3% to 10%, and the percentage of issued four-instruction-group is from 0.8% to 14%, except the mpegaudio. The percentage of mpegaudio is around 33%. That is because the basic block of mpegaudio is bigger, and within a basic block there are more instructions which can be run in parallel. These results demonstrate that our proposed dataflow instruction tagging scheme can expose more ILP.

5.3 ILP Speedup Gain

To compute the ILP gain, we assume all instructions with unit latency. Figure 2 presents the ILP speedup results for three different configurations: base in-order single-issue stack machine, stack folding only in-order single-issue stack machine and our multi-issue in-order TMSI machine. The stack folding used in the experiments also supports nested folding. With our tag-based stack folding scheme, the ILP gain can be seen from 20% to 90%. This result demonstrates that our stack folding scheme is effective, particularly for computing-extensive cases, such as Linpack and mpegaudio. The ILP gain with TMSI multi-issue over stack folding only single-issue in-order case is also observed to range from 3% to 27% for all applications except mpegaudio, for which the gain is 49%. The result also demonstrates that with tag-based scheme TMSI processor can improve the performance than stack folding single-issue in-order architecture does. The ILP gain with TMSI processor over base in-order single-issue stack machine can be seen from 21% to 115%, except for mpegaudio case in which the gain is 173%. This shows that the ILP speedup can be obtained through both stack folding and multi-issue instruction in Java processors.

5.4 Performance Enhancement of TMSI Processor Architecture

Figure 3 demonstrates the actual speedup obtained using the varied latency according to the picoJava-II specification. With the configuration of stack folding single-issue in-order architecture, an improvement of 2% to 19% is observed. With multiple-issue TMSI architecture, the speedup ranges from 9% to 34% for all applications, except mpegaudio and compress. The actual speedup of compress with TMSI is 49% while the actual speedup gain of mpegaudio is 86%. The reason for the much higher performance speedup observed in mpegaudio is that more bytecode instructions are executed in parallel than in other benchmark programs. Compared with SMTI [17], the results obtained are as good as or even better than those in SMTI, except *mpegaudio*. This demonstrates that our tag-based dataflow method can exploit more ILP. For mpegaudio benchmark, software-implemented multi-trace [17] may schedule instructions within a bigger instruction window than our scheme when bigger basic blocks exist. In contrast, the instruction schedule window in our processor is constrained by the size of Tag Matching Unit (TMU). However, our architecture does not need complex fetch logic to support.

Figure 2: ILP speedup gain: TMSI machine vs. the in-order single-issue stack machine

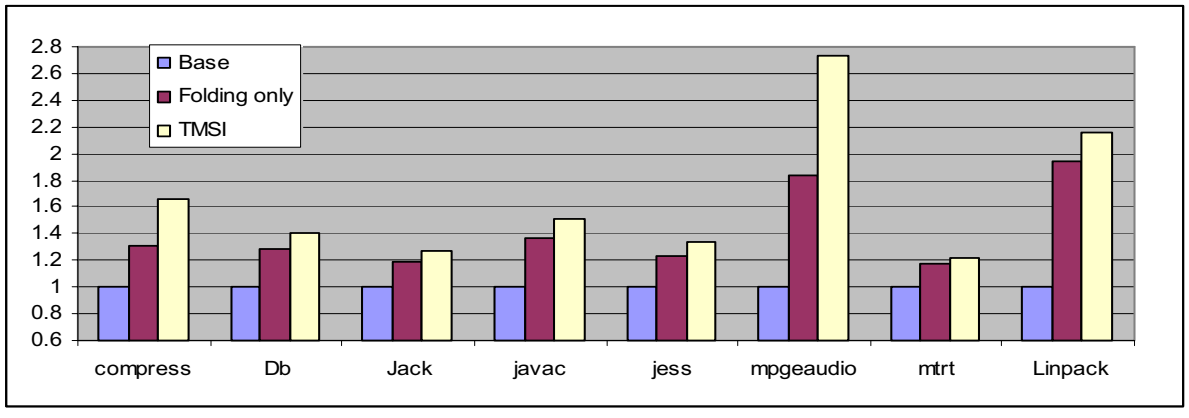
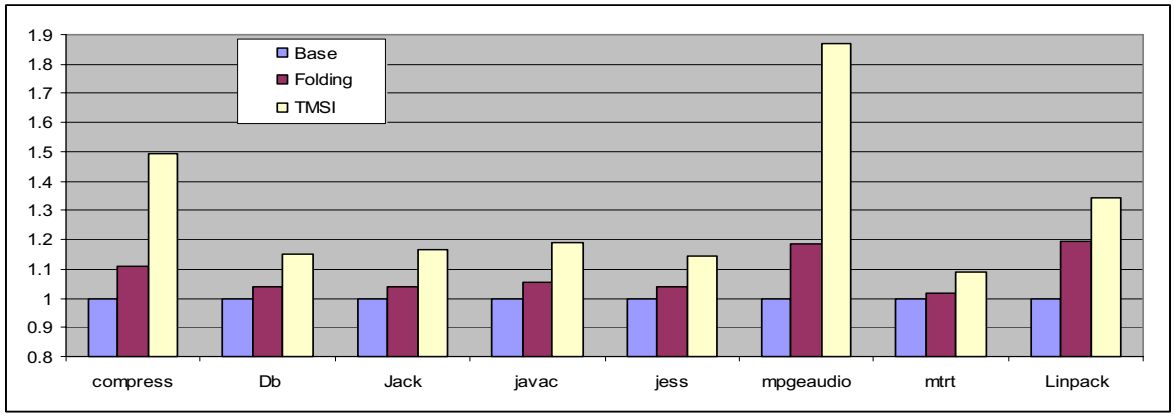


Figure 3: Overall speedup of TMSI vs. in-order single-issue stack machine



6. Conclusions

A new approach to exploiting dataflow to extract Java-ILP has been proposed. We presented the method of instruction tagging with an example. With instruction tagging, the independent bytecode instruction groups with stack dependences are identified. The different bytecode instruction groups can be executed in parallel based on different set of register segments for multiple operands access because there is no stack dependence among them. Based on the instruction tagging scheme, we proposed the architecture of Tag-based Multi-Issue Semi In-Order Architecture (TMSI) which employs a Tagging Unit and a Tag Matching Unit to tag bytecode instructions and execute the tagged instructions on a VLIW engine to achieve high performance. In addition, the tagging-based stack folding scheme has been discussed. The simulation experiments demonstrate that the proposed TMSI processor architecture is able to significantly increase the average ILP over a single-issue Java processor. We calculated the geometric mean of the ILP gain and that of actual gain in speedup over all the applications, the results showed that the ILP gain is 59% and the actual speedup gain is 28%. Based on current implementation, our future work will focus on scheduling and executing instructions beyond single basic blocks.

The proposed tag-based Java processor architecture can be compatible with multi-threading technique if multiple fetching unit and tagging unit are provided. Bytecodes from different threads are tagged by different tagging units, and then bundled to the VLIW instruction to be executed in parallel, and the thread-level parallelism is achieved. To guarantee the correctness of the program execution and to respect the Java Memory Model (JMM) [27], we will implement a memory consistency mechanism, such as sequential consistency or release consistency, in our future work in order to make our proposed dataflow instruction tagging scheme suitable for multi-threading computing environment and achieve better performance.

References

[1]. Arthur H. Veen. Dataflow machine architecture. ACM Computing Surveys, Vol. 18, Issue 4, December 1986.

[2]. Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-Driven and Demand-Driven Computer Architecture. ACM Computing Surveys, Vol. 14, Issue 1, March 1982.

[3]. Antonio C.S.Beck, Luigi Carro. A VLIW Low Power Java Processor for Embedded Applications. In 17th Brazilian Symp. Integrated Circuit Design (SBCCI 2004), Sep.2003.

[4]. John Glossner, et. al. Delft-Java Link Translation Buffer. In Proceedings of the 24th EUROMICRO conference, volume 1, pages 221–228, Vasteras, Sweden, August. 25-27 1998.

[5]. T.Lindholm, F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading MA, 1996

[6]. K. Ebcio˘glu, E. Altman, and E. Hokenek. A Java ILP machine based on fast dynamic compilation. In MASCOTS'97, - International Workshop on Security and Efficiency Aspects of Java, 1997.
[7]. Harlan McGhan and Mike O'Connor. PicoJava: A Direct Execution Engine for Java Bytecode. Sun Microsystems, IEEE Computer Magazine, 1998.
[8]. Lee-Ren Ton, Lung-Chung Chang, Min-Fu Kao, Han-Min Tseng. Instruction Folding in Java Processor. The International Conference on Parallel and Distributed Systems, 1997
[9]. El-Kharashi, W., F. ElGuibaly, and K.F. Li. A robust stack folding approach for Java processors: an operand extraction-based algorithm. Journal of Systems Architecture, Vol 47 (2001), pp.697-726
[10]. J. Michael O'Connor, Marc Tremblay. PicoJava-I: The Java Virtual Machine in Hardware. IEEE Micro, Vol. 17, Issue 2, pp 45-53, March 1997
[11]. Ramesh Radhakrishnan, Deependra Talla and Lizy Kurian John. Allowing for ILP in an Embedded Java Processor. In Proceedings of the 27th International Symposium on Computer Architecture, pages 294--305, June 2000.
[12]. Lee-Ren Ton, Lung-Chung Chang, Chung-Ping Chung. An analytical POC stack operations folding for continuous and discontinuous Java bytecodes. Journal of Systems Architecture, Vol. 48(2002), pp 1-16
[13]. R. Helaihel, and K. Olukotun. JMTP: An Architecture for Exploiting Concurrency in Embedded Java Applicatins with Real-time Considerations. In the international conference on Computer-Aided Design, Nov. 1999, pp. 551-557
[14]. Austin Kin, Morris Chang. Advanced POC Model-Based Java Instruction Folding Mechanism. In Proceedings of the 26th EUROMICRO Conference (EUROMICRO'00) Volume I-Volume1, p.1332
[15]. The Kaffe Virtual Machine, http://www.kaffe.org
[16]. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/.
[17]. R. Achutharaman, R. Govindarajan, G. Hariprakash, Amos R. Omondi. Exploiting Java-ILP on a Simultaneous Multi-Trace Instruction Issue (SMTI) Processor. In International Parallel and Distributed Processing Symposium, pp.76a, 2003.
[18]. M. Tremblay, J. Chan, S. Chaudhry, Andrew W. Conigliaro, S.S.Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. IEEE Micro Vol. 20, (6), Nov. 2000, pp. 12 -25.
[19]. Kemal Ebcioğlu , Erik R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. ACM SIGARCH Computer Architecture News, v.25 n.2, p.26-37, May 1997
[20]. Linpack, http://www.netlib.org/linpack
[21]. Krishna M. Kavi, Roberto Giorgi and Joseph Arul. Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation. IEEE Trans. On Computers, VOL 50, No. 8, August 2001
[22]. B.S. Yang, S.M. Moon, S. Park, J.Lee. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In the International Conference on Parallel Architectures and Compilation Techniques. October 1999.
[23]. Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp205, 1998
[24]. M.G. Burke, J.D.Choi, S.Fink, D.Grove, M. Hind, V. Sarkar, M.J. Serrano, V.Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In Proceedings ACM 1999 Java Grande Conference, 1999, pp.129-141.
[25]. M.Gschwind, E.R.Altman, S.Sathaye, P.Ledak, D.Appenzeller. Dynamic and Transparent Binary Translation. IEEE Computer, Vol 33(3), pp 54—59, March 2000.
[26]. J.E. Smith, and G.S. Sohi. The micro architecture of Superscalar Processors. In proceedings of the IEEE, vol. 83, pp1609-1624, December 1995.
[27]. Jeremy Manson, William Pugh and Sarita V.Adve. The Java Memory Model. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), California, USA, January 12 -14, 2005.