Exploiting Java-ILP on a Simultaneous Multi-Trace instruction Issue (SMTI) Processor

R. Achutharaman[1], R. Govindarajan[2], G. Hariprakash[1], Amos R. Omondi[3]

[1]Sun Microsystems, Bangalore. {achutharaman.rangachari, hariprakash.govindarajalu}@sun.com

[2]Supercomputer Education and Research Center, Indian Institute of Science, Bangalore, India. govind@serc.iisc.ernet.in

[3]School of Informatics and Engineering, Flinders University, Australia. amos@infoeng.flinders.edu.au

Abstract

The available Instruction Level Parallelism in Java bytecode (Java-ILP) is not readily exploitable using traditional in-order or out-of-order issue mechanisms due to dependencies involving stack operands. The sequentialization due to stack dependency can be overcome by identifying bytecode traces, which are sequences of bytecode instructions that when executed leave the operand stack in the same state as it was at the beginning of the sequence. Instructions from different bytecode traces have no stack-operand dependency and hence can be executed in parallel on multiple operand stacks. In this paper, we propose a simultaneous multi-trace instruction issue (SMTI) architecture for a processor that can issue instructions from multiple bytecode traces to exploit Java-ILP. The proposed architecture can easily take advantage of nested folding to further increase the ILP. Extraction of bytecode traces and nested bytecode folding are done in software, during the method verification stage, and add little run-time overhead. We carried out our experiments in our SMTI simulation environment with SPECjvm98, Scimark benchmarks and the Linpack workload. Simultaneous multi-trace issue combined with nested folding resulted in an average ILP speedup gain of 54% over the base in-order single-issue Java processor.

1. Introduction

With platform independence as its core feature, Java technology has become extremely popular and is found in a variety of systems, ranging from embedded systems to high performance server systems. The wide spread deployment of Java also necessitates the need for high performance. First we review related works on efficient execution of Java Programs.

1.1. Related Work

General purpose and high performance machines use either interpretation or Just-In-Time (JIT) compilation to convert the Java bytecodes to native instructions which are then run on conventional processors. While interpretation is simple, it is neither efficient nor well-suited to high performance applications. The JIT approach [3, 23], incurs runtime overheads in translating bytecodes to native code, although the efficiency in executing native code more than offsets the translation cost. Recent research has led to adaptive dynamic compilation in which methods that are repeatedly executed are adaptively retranslated and traditional compiler optimizations are then applied to obtain more efficient translated code [1, 6].

Processors to execute bytecode directly are also becoming popular as increasingly larger number of complex server applications are developed in Java and it is important to achieve very high performance for these server applications [8, 9, 13]. The designs of such processors are mainly based on stack architectures. Although such processors have primarily been considered for embedded applications, they are also now slowly being adapted for high-end server applications. A major issue in these architectures is that the extent of ILP is limited by the dependency (introduced by stack operands) between Java bytecodes.

Several techniques to exploit available ILP in Java bytecode have been investigated in recent research [4, 5, 12, 15, 18]. Stack operation folding is one technique that eliminates stack operand dependency by converting a set of bytecodes into a register-based instruction [2, 21]. This also saves pipeline cycles by eliminating redundant stack-copy operations. Sun's picoJava-II processor does simple instruction folding in hardware at the decode stage of its pipeline [14]. It also implements the stack cache as a register file for parallel access of stack operands to eliminate the inefficiencies of stack operations. More complex folding techniques, such as nested folding, are used to further reduce stack operand dependency, but at the expense of increased hardware complexity [2, 4].

Further ILP can be exploited by combining multiple in-order instruction issue and stack operation folding, so that folded instructions operate directly on register operands and in parallel. However, the extent of ILP is limited

when execution is restricted to program order [15].

There have been studies on using *thread level parallelism* (TLP) to extract coarse-grained parallelism from Java bytecodes. Sun's MAJC processor proposes a way to run Java methods as threads in hardware and includes speculative execution of multiple threads on to multiple processors (on-chip) in order to exploit TLP [12]. Another similar concept was implemented in the Java Multi-Threaded Processor (JMTP) architecture, consisting of a single chip with an off-the-shelf general-purpose processor core coupled with an array of Java Thread Processors (JTP) [5]. However, the Java code needs to be JIT compiled for the MAJC processor, and for the latter, compiler intelligence is needed to identify the set of concurrent threads that can be forked as JTP threads.

1.2. Overview of our Work

In this paper, we propose to exploit fine-grained ILP through the use of out-of-order multiple instruction issue and speculative execution. The basic concept is a *bytecode trace*, which enables out-of-order multi-issue of Java bytecodes. A bytecode-trace is a sequence of Java bytecodes that has no stack operand dependency with any other bytecodes in a method. Each bytecode-trace can run independent of, and in parallel with, other traces, on a separate operand stack. To avoid hardware complexity, we extract bytecode-traces in software during the *method verification* stage, where integrity checks are carried out on the bytecodes of a method before the method is executed. We propose to also include folding of bytecodes in the method verification phase. In addition to reducing the hardware complexity, software folding allows us to take advantage of (i) more complex folding patterns, and (ii) nested folding. Our experiments with SPECjvm98 benchmarks reveal that the software overheads in extracting bytecode-traces and in folding bytecodes are no more than 28% of the method verification time, which amounts to less than 5% of the total execution time.

The proposed architecture is evaluated using SPECjvm98, Scimark benchmarks and Linpack workloads. The multi-issue scheduling of bytecode traces within a basic block resulted in an average ILP speedup gain of 13.5%. Bytecode-traces when combined with instruction folding yielded an average ILP speedup gain of 54%.

In the following section we give the motivation for extracting Java ILP using bytecode-traces. In section 3, we discuss Java bytecode-trace extraction. Section 4 describes the proposed architecture. We present performance results in Section 5. The last section summarizes our work.

2. Motivation

Consider, for example, the code snippet shown in Table 1 that illustrates how stack dependencies limit ILP in Java bytecodes.

Table 1. Example code snippet and corresponding bytecodes and folded instructions.

Sample code snippet

```
x = (a*b) + (b*c);
y = (a*(c-(b*d)));
```

Java bytecodes		Operand stack contents	Java Instructions	
			Folded	Nested folded
b1	iload a	a		
b2	iload b	a, b		
b3	mul	t1(=a*b)	mul a, b, t1	mul a, b, t1
b4	iload b	t1, b		
b5	iload c	t1, b, c		
b6	mul	t1,t2(=b*c)	mul b, c, t2	mul b, c, t2
b7	add	x(=t1+t2)	add t1,t2, x	add t1,t2, x
b8	istore			
b9	iload a	a	iload a	
b10	iload c	a, c	iload c	
b11	iload b	a, c, b		
b12	iload d	a, c, b, d		
b13	mul	a,c,t3(=b*d)	mul b,d, t3	mul b, d, t3
b14	sub	a,t4(=c-t3)	sub stack[top],t3,t4	sub c,t3, t4
b15	mul	y (=a*t4)	mul stack[top],t4,y	mul a, t4, y
b16	istore			

The second column in Table 1 shows the contents of the operand stack after the execution of each bytecode instruction. Assume that the operand stack is initially empty. Then, the operand stack contains one or more operands after bytecode instruction b1 and remains non-empty until after instruction b8. Thus the bytecode instructions b1 to b8 have to execute sequentially on a stack machine, as they depend on the contents of the operand stack. Such dependencies are referred to as *stack dependencies*. Bytecode instructions starting from b9 are stack-independent of any of the earlier instructions, but in an in-order issue machine b9 cannot be issued until all earlier instructions (b1 to b8) have been issued. Thus a simple stack machine cannot exploit any parallelism in the above sequence. Assuming each instruction takes one cycle to execute, the Java bytecode sequence shown in Figure 1 will take 16 cycles in a strict stack machine.

In the given bytecode sequence, the operand stack becomes empty after executing instruction b8 and also after instruction b16. More precisely, if the stack pointer is pointing to some position *p* at the beginning of a Java method or a basic block, then after executing the sequence of bytecode instructions b1 to b8, the stack pointer regains its old value *p*. We shall use the term *clean-stack-point* to refer to a point at which the stack-pointer valued is restored in such a manner. The sequence of Java bytecodes between any two consecutive clean-stack-points form a *bytecode-trace*. Since each bytecode-trace is stack independent of every other bytecode-trace, multiple bytecode-traces can be executed in parallel. In the example of Table 1, there are two bytecode traces: one from b1 to b8 and another from b9 to b16. By taking instructions from different bytecode traces and issuing them in parallel to multiple functional units, each of which has its own private operand stack, instruction-level-parallelism can be exploited. We refer to this instruction-issue approach as *simultaneous multi-trace instruction*

issue (SMTI). SMTI is similar to the approaches followed in Simultaneous Multithreading architectures and Trace Processor [22, 16]. If the bytecodes in Table 1 are issued in this manner, execution of the entire sequence will require only 8 cycles, in contrast with the 16 needed with in-order single issue stack machine.

Table 2. Per-cycle parallel execution of Table-1 instructions on in-order multi-issue and multi-trace-issue processors.

Cycle #	in-order multi-issue + folding	in-order multi-issue + nested folding	multi-trace-issue + nested folding	
			issue slot #1	issue slot #2

1	`mul a,b,t1`
	`mul b,c,t2`
	`add t1,t2,x`
		`mul a,b,t1`
		`mul b,c,t2`
		`add t1,t2,x`
			`mul a,b,t1`	`mul b,d,t3`

2	`iload a`	`mul b,d,t3`	`mul b,c,t2`	`sub c,t3,t4`
3	`iload c`	`sub c,t3,t4`	`add t1,t2,x`	`mul a,t4,y`
4	`mul b,d,t3`	`mul a,t4,y`		
5	`sub stack[top],t3,t4`			
6	`mul stack[top] t4,y`			

With instruction folding, the bytecode sequence shown in Table 1 can be executed in 8 cycles in an in-order single issue machine. For example, in Table 1, bytecode instructions b1 to b3 may be replaced with a single folded instruction: *`mul a,b,t1`*. In this example, operands a and b are local variable and can be accessed in parallel from a local variable register file. Further reduction in redundant stack operations can be achieved by nested folding. Table 1 illustrates that the bytecode instructions b9 and b10, which could not be folded in simple folding, can be eliminated in nested folding. With nested folding, the bytecode sequence can now be executed in 6 cycles on an in-order single issue machine. Combining bytecode traces and nested folding, the example sequence can be executed in just 3 cycles as shown in Table 2. Software folding has two main advantages: (i) it is more flexible, as the algorithm can be changed to cover more complex folding patterns; (ii) nested folding takes little time (2% to 6% of method verification time) and eliminates the need for the complex hardware.

3. Bytecode Trace and Instruction Folding

In this section we describe the extraction of bytecode traces and the nested folding of bytecode instructions in software, during the method verification stage. The method verification process is mandatory for server JVMs to check the integrity of the Java bytecodes of a method for security purposes [10]. Hence, performing bytecode trace extraction, local variable data dependence analysis and nested folding during the verification process simplifies the hardware implementation at the cost of a very small penalty in time.

3.1. Bytecode Trace Extraction

The method verification procedure is invoked in the trap handler, and a trap signal is generated when a method-invocation bytecode (`invokevirtual, invokestatic, invokespecial, or invokeinterface`) is encountered in the decode stage and the target method is called for the first time.

We shall use the code snippet in Figure 1 to explain the bytecode-trace extraction procedure; the code is taken from the *SPECjvm98/201_compress* benchmark. In bytecode-trace extraction, the operand stack is simulated and the stack depth is tracked, starting with a stack depth of 0, at the beginning of each Java method. As each bytecode instruction is processed, the stack depth is calculated. When the stack depth reaches 0, the current bytecode-trace is ended and a new bytecode-trace is started. In Figure 1, there are 6 bytecode-traces within the basic block.

```
public void compress() {
                int i = 0, c;
                int fcode, ent, disp;
                int hsize_reg, hshift;
  <skip>
  while ( (c = Input.getbyte()) != -1) {
                in_count++;
                fcode = (((int) c << maxbits) + ent);
                  /* xor hashing */
                i = ((c << hshift) ^ ent);
                int temphtab = htab.of [i];
                if ( htab.of [i] == fcode ) {
                ....
  }
```

PC: Java bytecode	Operand Stack snapshot (stack depth)	Java code equivalent
72:aload_0	sp (1) start trace	
73:dup	sp+=1 (2)	
74:getfield #31	sp-=1, SP+=1 (2)	in_count++
77:iconst_1	sp+=1 (3)	
78:iadd	sp-=2, sp+=1 (2)	
79:putfield #31	sp-=2 (0) end trace	
82:iload_3	sp (1) start trace	
83:aload_0	sp+=1 (2)	
84:getfield #34	sp-=1,sp+=1 (2)	fcode =
87:ishl	sp-=2,sp+=1 (1)	(((int)c <<
88:iload #4	sp+=1 (2)	maxbits)+
90:iadd	sp-=2,sp+=1 (1)	ent)
91:istore_1	sp-=1 (0) end trace	
92:iload_3	sp (1) start trace	
93:iload #7	sp+=1 (2)	
95:ishl	sp-=2,sp+=1 (1)	i=((c<<hshift)^
96:iload #4	sp+=1 (2)	ent)
98:ixor	sp-=2,sp+=1 (1)	
99:istore_2	sp-=1 (0) end trace	
100:aload_0	sp (1) start trace	
101:getfield #30	sp-=1,sp+=1 (1)	htab
104:astore #8	sp-=1 (0) end trace	
106:aload #8	sp (1) start trace	
108:getfield #47	sp-=1,sp+=1	temphtab =
111:iload_2	sp+=1 (2)	htab.of(i)
112:iaload	sp-=2,sp+=1 (1)	
113:istore #8	sp-=1 (0) end trace	
115:iload #8	sp (1) start trace	
117:iload_1	sp+=1 (2)	if(htab.of[i]==
118:if_icmpne 134	sp-=2 (0) end trace	fcode)

Figure 1. Bytecode-trace extraction process; code snippet from *SPECjvm98/ 201_compress*

There can be one or more bytecode-traces within a basic block, and in a few rare cases a trace can extend beyond a basic block. A bytecode-trace that ends within a basic block is called a *complete bytecode-trace*, and that which extends beyond a basic block is an *incomplete bytecode-trace.* An *incomplete bytecode-trace* is stack

dependent on the bytecode-trace of a following basic block, which is again an incomplete bytecode-trace that depends on the former. If the end of the trace is a control transfer instruction, then the bytecode-trace may end in any one of the next basic blocks, primary or alternate basic block, depending on the path the execution control takes. Incomplete bytecode-traces, each of which typically spans two or more basic-blocks, are run sequentially on the same stack. Our study reveals that more than 95% of the bytecode-traces are complete.

Table 3. An Incomplete trace spanning multiple basic blocks `z = x / ((y != 0)? y: 1).`

Basic block#: trace#	*Bytecode sequence*	*Comments*
bb1: t1	b1: iload x b2: iload y b3: ifeq b8	bb1 ends; t1 ends incompletly
bb2: t2	b4: iload y b5: idiv b6: istore z b7: goto b11	next primary BB (of bb1) starts end of trace t1 (complete)
bb3: t3	b8: iconst_1 b9: idiv b10: istore z	next secondary BB (of bb1) starts end of trace t1 (complete)

In the example of Table 3, if a control transfer takes place from instruction b3 to b8, then the incomplete trace t1 of basic block 1 and the incomplete trace t3 of the basic block 3 should be scheduled to execute on the same stack. The bytecode-trace scheduling logic should take care of scheduling such incomplete trace-pairs on the same stack.

Once the bytecodes are extracted, information on the bytecode-traces, including the trace-id, trace-start PC, trace-end PC, and the basic-block id, as well as additional information such as, inter-trace dependency constraints, folding information, etc., is stored in a *basic block trace table* (BBTT) in the proposed architecture. The BBTT is subsequently accessed by the trace fetch logic and trace scheduler.

3.2. Local Variable Dependence Analysis

In addition to stack operands, data dependencies on local variables can exist within and across bytecode traces. Dependencies within a trace are resolved through sequential execution of bytecode instructions and dependencies across a pair of bytecode- traces are resolved in the bytecode-trace extraction procedure, during the method verification stage. Local variables are allocated and accessed with constant indices in Java. Hence static analysis of local variable data dependence across bytecode is possible and sufficient. Therefore, the appropriate information for local variables dependencies across bytecode-traces is extracted statically and is stored in BBTT. For each bytecode-trace, the ids of other bytecode-traces on which it is dependent on are stored in the BBTT entry.

Memory dependency arising from object reference, such as class and arrays, are speculated and detected only during run-time. Since only bytecode-traces that are wholly contained within a basic block are scheduled for execution, c*ontrol dependence* analysis between bytecode-traces is not necessary.

3.3. Folding and Nested Folding

In our approach we use the same set of folding groups as implemented in picoJava II [14]. The folded instructions are similar in format to 3-address instructions. Nested folding is performed by recursively applying folding on bytecode sequences that have folded instructions. A folded bytecode instruction replaces a sequence of bytecodes and it may alter the subsequent instructions and their target addresses. Hence, the instruction and target address are recalculated for the entire method. The folded instructions along with the bytecodes of a method are stored in a *method instruction buffer* and also in a *method cache* during the execution of the method.

4. SMTI Java Processor Architecture

We propose a five stage Java processor pipeline with instruction-fetch, decode, issue, execute and store/commit stages. The schematic block diagram of the SMTI Java processor architecture is shown in Figure 2.

4.1. Trace Fetch and Decode

The instructions are fetched from a method cache. The bytecode-trace fetch logic uses the trace information in the BBTT to select traces that are free from local variable dependency. The traces are selected from the same basic block. The program-counter values of selected traces are passed to Instruction Fetch (IF) controller, which then fetches the bytecode instructions from the method cache. After fetching all the traces of a basic block, the trace fetch logic starts selecting the traces from the next basic block as predicted by the branch predictor. We assume a simple 2-bit branch prediction using a Branch Prediction Table (BPT).

We propose a decoder similar to that of picoJava-II decoder to handle both bytecode instructions and folded instructions. The decoded instructions are placed in the *decoded trace buffer (DTB)*. The DTB is implemented as multiple small buffers to separately hold decoded instructions from each trace. The number of such buffers is twice the issue width, which is the maximum number of instructions that the trace scheduler can issue in a cycle, to keep the scheduler busy.

4.2. Trace Scheduler

Bytecode-traces have no stack dependence and hence can be executed in parallel. Further the fetched traces do not have any local variable dependence among themselves. Among the traces in DTB, the trace scheduler selects some based on a priority function that depends on the available resources; such a function could be simply

based on the trace-ids. The instruction issue order within a bytecode-trace is in-order, whereas instruction issue between bytecode-traces is out-of-order. Memory dependencies between traces are speculated with the initial assumption that there is no memory dependency across traces. Each selected trace is assigned a dedicated operand stack (OPstack) and a Reservation Station (RS). From the selected bytecode-traces, one instruction from each trace is issued to a dedicated RS. The trace scheduler maintains sufficient trace program-counters to keep track of the issued traces.

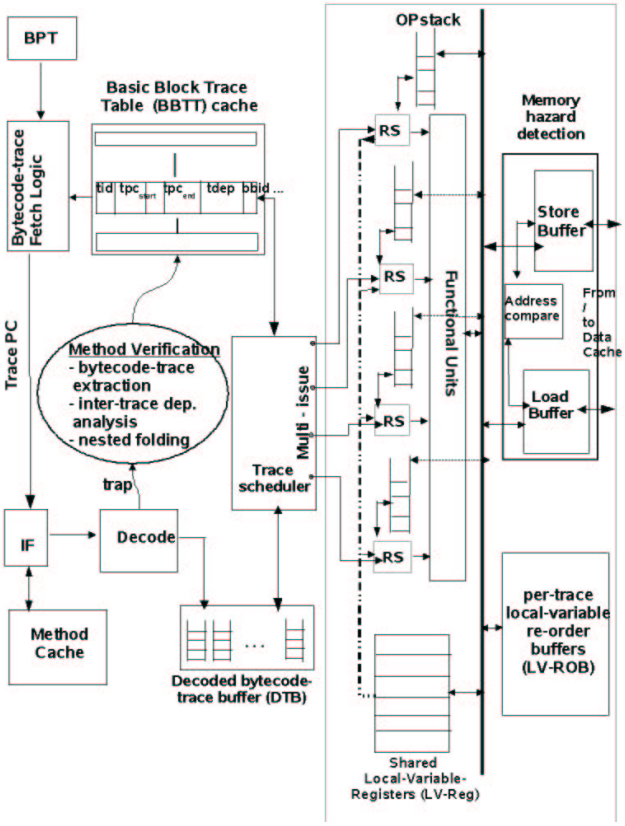

Figure 2. Multi-trace instruction issue (SMTI) Java processor architecture.

4.3. Bytecode-Trace Execution

The RS controls the operand movement among a dedicated OPstack, the shared local variable registers (LV-Reg), and a pool of functional units. The operands for the instructions may be loaded from or stored to any one of several locations -- the OPstack, LV-Regs, and the data cache. The RS gets the required operands from any of these source locations, and dispatches them to the functional unit or to the OPstack (in the case of instructions which push onto or pop from OPstack). Each OPstack is implemented as a register stack cache, as in the picoJava processor [14], in order to enable parallel access of stack operands for folded Java instructions. Each bytecode-trace does run-time local variable Writes with the per-trace local variable reorder buffer (LV-ROB). Reads of the shared local variable register are passed through the LV-ROB to ensure that any WAR hazards are avoided. The organization of LV-ROB is similar to that of a reorder buffer in a superscalar processor [19].

4.4. Memory Mis-Speculation Handling

The out-of-order memory dependency between bytecode-traces is detected at run-time by the memory hazard-detection logic. The memory-hazard unit basically consists of a load buffer, a store buffer and comparator circuits [19]. The trace id is stored along with the data in each entry of the load and store buffers. Further, we assume that the trace-ids uniquely specify the program order, i.e., the larger the trace id, the later the trace appears in the basic block. The speculative stores (from traces that are executing out of program order) are always written into the store buffer. A speculatively executed load instruction may take the value from the store buffer, if there is an entry in the buffer corresponding to a store from this or an earlier trace to this address location; otherwise, the value is loaded from the D-Cache.

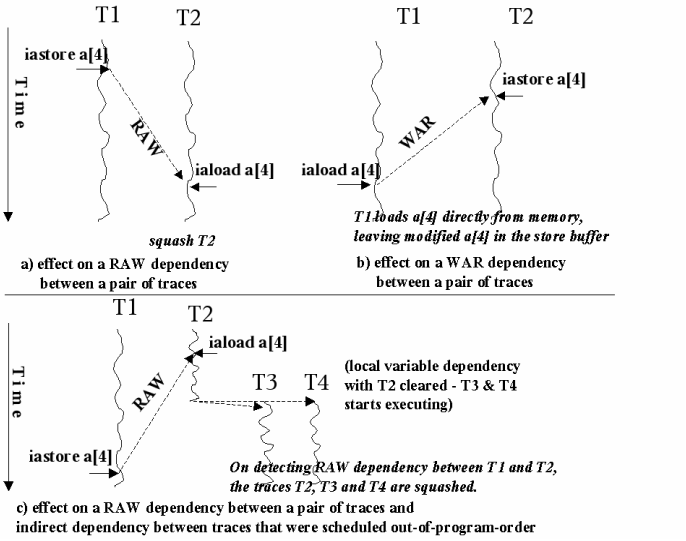

Figure 3. Memory dependency detection at run time across multiple speculated traces T2,T3,etc., with the program order trace (non-speculative) T1.

The conditions in which the memory dependencies are detected are shown in Figure 3. When a load or store is executed, the following actions are carried out by the memory hazard-detection logic: (i) A search is done on the load/store buffer to check whether some other trace running out-of-order (with a greater trace id) has issued a memory write. (ii) If so, the dependency type (RAW, WAR, or WAW) is determined by comparing the trace-ids and checking the operation type. (iii) If the dependency is a WAR, the current trace fetches the data from the data cache, ignoring the value in the load/store buffer that has

been modified by the elder traces (as shown in Figure 3b). (iv) If the dependency is a WAW, the current trace makes a separate entry in the store buffer. Multiple store-buffer entries with the same address, by different trace ids, will not introduce an inconsistency, since at trace commit only the entry store made by a highest trace id will be flushed in to the D-cache. (v) If the dependency is a RAW, which is a true dependency, the elder traces and their dependent traces are squashed and rescheduled (as shown in Figures 3a and 3c).

The following actions are carried out to squash a trace: (i) all the entries of the victim trace in the LV-ROB are deleted; (ii) all the load/store entries made by the victim trace in the load/store buffer are invalidated; and (iii) the OPstack assigned for the trace is cleared.

4.5. Trace Commit

When the last instruction of a trace that is executing in the program order (non-speculative trace) completes, the local variables modified by the trace from LV-ROB are written to LV-Regs and the trace is said to be committed. At the same time, the trace's committed value in the load /store buffers is flushed to the data cache.

5. Performance Results

We simulate the proposed architecture by modifying the open source Java VM interpreter Kaffe [7]. Nested folding is implemented using folding groups similar to that of picoJava-II and extended to multiple nesting levels. We used SPECjvm98 [20], Scimark [17], and Linpack [11] benchmarks. The data set used for SPECjvm98 applications is s1. The Scimark and Linpack benchmarks were included in order to study the benefits of byte-code tracing on scientific workloads.

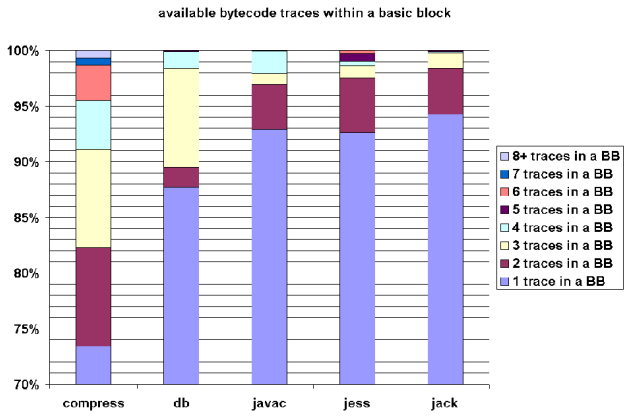

Figure 4. Available bytecode traces within a basic block in SPECjvm98 applications.

First, we determine the amount of available bytecode-trace level parallelism within a basic block. In measuring the available parallelism, we have considered inter-trace local-variable data dependency, but not memory dependency. Figure 4 shows the available parallelism, a dynamic performance measure, in SPECjvm98 applications. From Figure 4, we observe that in the various benchmark programs that we used, 75% - 95% of the basic blocks have only one trace. The instance when two traces occur in a basic block range from 2% to 9%; and for the cases where three or four traces in a basic block ranges from 1% to 9%. In some of the applications there are very few basic blocks that contain more than four traces. Hence, we chose to implement a 4-way trace issue bytecode-trace scheduler in our experiments.

We also report the software overhead involved in performing trace extraction, inter-trace dependency analysis and folding during method verification stage in Table 4. On an average, the software overhead is about 28% of the method verification time. The method verification procedure itself accounts for less than 5% of the total execution time.

To study the gain in ILP and speedup with SMTI architecture, we ran two types of simulation: one in which every bytecode instruction takes a single cycle, and the other in which the number of cycles taken by different bytecodes vary according to the picoJava specification. The former is useful in determining ILP gain or ILP speedup purely from the multiple instruction issue viewpoint. The latter is helpful in determining the actual speedup, which is indicative of the actual performance gain in an SMTI architecture. In our simulation, we have assumed an ideal instruction cache.

Table 4. Software overhead in bytecode-trace extraction, inter-trace dependency analysis, folding and nested folding.

Application	Time taken to perform (in % of method verification time)			
	inter-trace extraction and dependency analysis	folding	nested folding	Total
compress	22.00%	0.50%	2.60%	**25.10%**
jess	18.00%	0.40%	2.20%	**20.60%**
db	25.00%	0.50%	3.00%	**28.50%**
javac	27.00%	0.60%	3.00%	**30.60%**
jack	31.00%	0.70%	4.00%	**35.70%**
mtrt	27.00%	0.70%	3.20%	**30.90%**
mpegaudio	20.00%	1.00%	4.50%	**25.50%**
scimark	23.00%	0.50%	6.00%	**29.50%**
linpack	24.00%	0.50%	3.00%	**27.50%**

The ILP gain on a SMTI architecture with bytecode-trace issue for the different benchmarks is depicted in Figure 5. The effects of simple and nested folding on bytecode-traces are also shown (the set of bars denoted by SMTI in the figure). The results are compared with a single-issue Java processor (like picoJava) with simple and nested folding (the set of bars shown as SI in the figure). The instruction window for nested folding is the entire basic block in the case of single-issue; for the multi-trace issue, the window is limited to the length of a bytecode-trace.

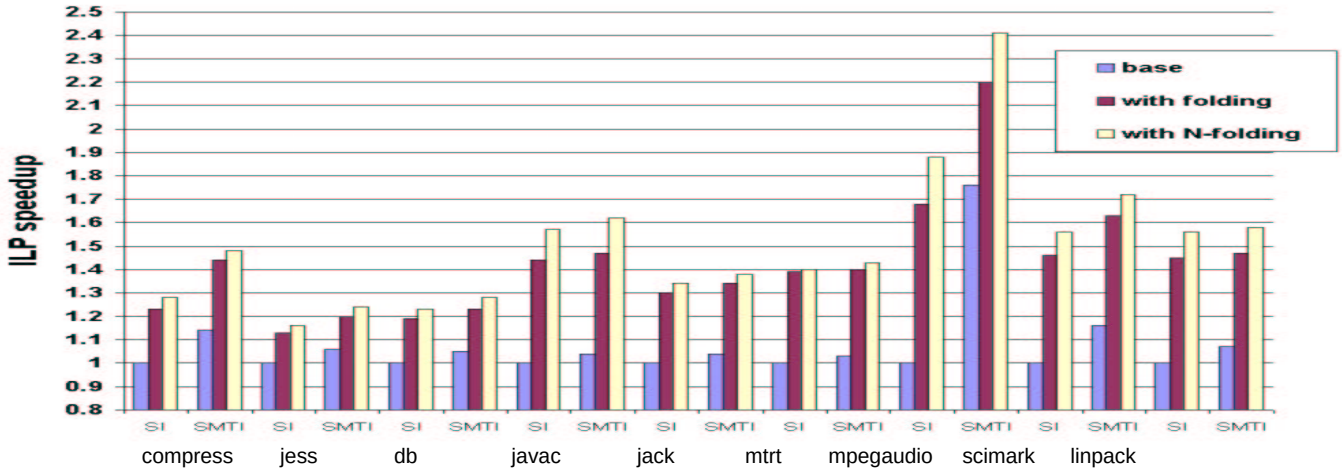

Figure 5. ILP speedup gain: SMTI machine vs. the in-order single-issue (SI) stack machine.

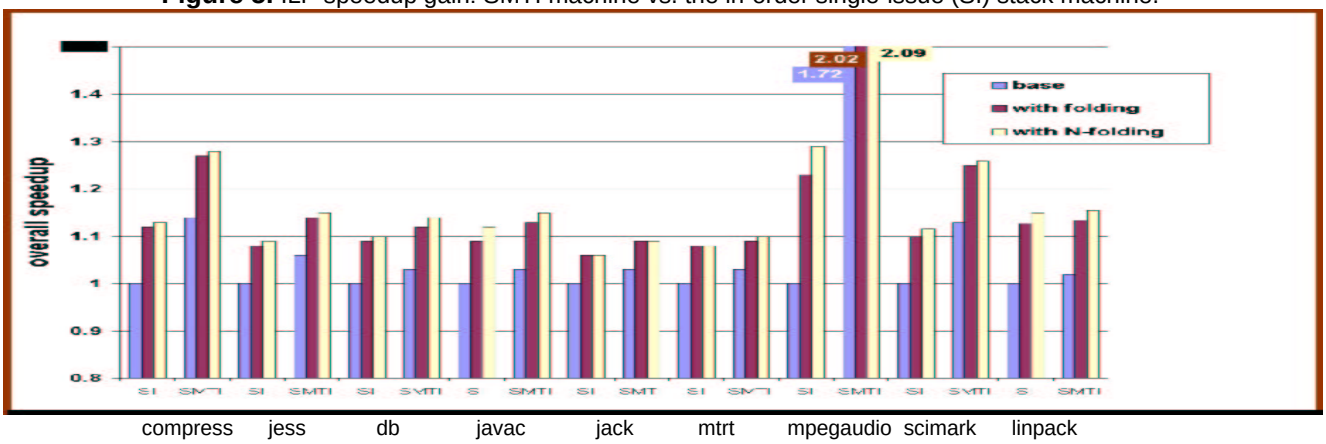

Figure 6. Overall speedup: single-issue (SI) stack machine vs. SMTI machine.

The ILP gain with multi-trace issue over single-instruction issue is observed (refer to Figure 5) to range from 3% to 16% for all applications, except mpegaudio, where the gain is 76%. We can see the ILP gain increases with nested folding, even on a single-issue architecture, from 16% to 88%. When nested folding is enabled on SMTI, the ILP increases further, ranging from 16% to 141%.

Table 5. Percentage of instructions executed in parallel on a SMTI architecture.

Applications	Number of instructions executed in parallel					
	2		3		4	
	MI	MI+NF	MI	MI+NF	MI	MI+NF
compress	14.40%	17.00%	3.80%	4.40%	3.30%	3.20%
jess	6.70%	6.90%	2.95	2.40%	1.10%	1.20%
db	5.40%	5.20%	2.40%	1.80%	0.40%	0.30%
javac	4.20%	4.90%	1.60%	1.00%	0.30%	0.30%
jack	4.10%	3.10%	2.70%	1.60%	0.10%	0.10%
mtrt	1.80%	2.00%	2.60%	1.30%	0.10%	0.10%
mpegaudio	19.30%	22.90%	17.80%	4.60%	11.50%	10.00%

MI: *with multi-trace instruction issue on the bytecodes*
MI + NF: *with multi-trace instruction issue on the folded instructions*

The actual speedup obtained using non-unit latencies for instructions, is shown in Figure 6. With just bytecode-trace issue on a SMTI architecture, an improvement of 2% to 14% is observed, and with nested folding included the speedup ranges from 9% to 28% for all applications, except mpegaudio. Compared to this, in-order single issue (SI) achieved an actual speedup of only XX% to YY%. The actual speedup gain for mpegaudio is 72% with SMTI and increases to 109% for SMTI with nested folding.

The reasons for the much higher performance observed in mpegaudio is that the code contains more traces within a basic block, and these are executable in parallel. Table 5 shows the percentage of instructions executed in parallel with multi-trace-issue and multi-trace-issue with nested folding. The percentage of two, three and four instructions executed in parallel is much higher for mpegaudio than for any other applications.

Let us quickly compare the performance of SMTI with the previous research work of in-order multi-issue of the folded Java instructions [15]. As per the results declared in [15], the proportion of instructions that was executed in parallel ranged from 11% to 21% for two instructions, whereas that for three instructions remained at less than

3%; and there were no sets of four instructions executed in parallel. Table 5 shows that with SMTI, where simultaneous multi-trace-issue is combined with nested folding, higher ILP can be exploited.

Table 6. Average (geometric mean) speedup with bytecode-folding, multi-trace issue and the combination of both for the taken benchmark set.

	With SI + NF	With SMTI	With SMTI + NF
ILP gain	43.00%	13.50%	**54.00%**
speedup gain	12.40%	12.00%	**24.50%**

Lastly, we report certain statistics on local-variable memory dependence and trace squashing due to memory dependence mis-speculation. On an average, over the benchmarks used in our study, 15% of the traces have a local-variable dependence with another trace in the same basic block. The number of traces squashed due to mis-speculation account for less than 2% of the total (dynamic) traces for all applications. Thus we find the squashing overhead due to memory dependence mis-speculation is very small, possibly because the traces running speculatively in parallel are limited to a single basic block.

To summarize, the geometric mean of ILP gain and that of actual gain in speedup over all the applications are shown in Table 6. With SMTI and nested folding, the ILP gain is 54% and the actual speedup gain is 25%. We anticipate that speculative simultaneous multi-trace instruction issue, which issues instructions from traces beyond basic block, in a control speculative manner, will yield even better performance.

6. Conclusions

A new technique called bytecode-traces has been proposed to exploit Java-ILP. We have also proposed an architecture, the Simultaneous Multi-Trace Issue (SMTI) architecture, to schedule multiple traces on different operand stacks in parallel. Further the effect of folding and nested folding with bytecode-trace execution on the SMTI Java processor has been studied, and it has been shown that combining multi-trace issue with nested folding has significantly increased the average ILP over a single issue bytecode processor. The bytecode-trace extraction and nested folding are done in software, during bytecode verification phase, eliminating the need for additional hardware.

Our immediate future work will be to extend parallel bytecode-trace execution beyond single basic blocks to exploit greater Java-ILP.

Reference

[1] 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 ACM Java Grande Conference, June 1999, pp. 129-141.

[2] L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung, *Stack operations folding in Java Processor*, IEEE proceedings on Computers and Digital Techniques, vol. 145, pp. 333-340, September 1998.

[3] T. Cramer, R. Friedman, T. Miller, D. Seberget, R. Wilson, and M. Wolczko, *Compiling Java just in time*, IEEE Micro, Vol.17, pp. 36-43, May-June 1997.

[4] M.W. El-Kharashi, F. Elguibaly, and K. Li, *An Operand Extraction-Based Stack Folding Algorithm*, In 2nd annual workshop on Hardware support for Objects and micro architecture for Java, September 2000, pp. 22-26.

[5] R. Helaihel, and K. Olukotun, *JMTP: An Architecture for Exploiting Concurrency in Embedded Java Applications with Real-time Considerations*, In the international Conference on Computer-Aided Design, November 1999, pp .551-557.

[6] *HotSpot JVM*, http://java.sun.com/products/hotspot

[7] *Kaffe Virtual Machine*, http://www.kaffe.org

[8] *LavaCORE: Configurable Java Processor Core*, http://www.derivation.com

[9] *Lightfoot Java Processor Core*, http://ww.dctl.com/lightfoot.html.

[10] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd edition, Addision-Wesley, 1999.

[11] Linpack, http://www.netlib.org/linpack

[12] *MAJC architectural tutorial*, Sun Microsystems, http://developer.java.sun.com/developer/products/majc/

[13] J.M. O'Connor and M. Tremblay, *picoJava-I: The Java virtual machine in hardware*, IEEE Micro, March-April 1997, Vol.17, pp 45-53.

[14] *picoJava-II programmer's reference manual*, Sun Microsystems, 1997.

[15] R. Radhakrishnan, D. Talla, and L.K. John, *Allowing for ILP in an Embedded Java Processor*, In the international Symposium on Computer Architecture, June 2000, pp. 294-305.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, J. Smith, *Trace Processors*, In the international symposium of microarchitecture, December 1997, pp. 138-148.

[17] *Scimark*, http://math.nist.gov/scimark2

[18] K. Scott and K. Skadron, *BLP: Applying ILP techniques to Bytecode Execution*, In the 2nd annual workshop on hardware support for objects and micro architecture for Java, September 2000.

[19] J.E. Smith, and G.S. Sohi, *The micro architecture of Superscaler Processors*, In proceedings of the IEEE, vol. 83, pp. 1609-1624, December 1995.

[20] *SPEC JVM98 Benchmarks*, http://www.spec.org/osg/jvm98.

[21] L.-R. Ton, et al., *Instruction folding in Java Processor*, In the international conference on Parallel and Distributed Systems, 1997.

[22] D. Tullsen, S.J. Eggers, H.M. Levy, *Simultaneous Multithreading: Maximizing on-chip Parallelism*, In the international Symposium on Computer Architecture, 1995, pp. 392-403.

[23] 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.