ELSEVIER

Journal of Systems Architecture 48 (2002) 1–16

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

An analytical POC stack operations folding for continuous and discontinuous Java bytecodes

Lee-Ren Ton *, Lung-Chung Chang, Chung-Ping Chung

Department of Computer Science and Information Engineering, National Chiao Tung University, No. 1001, Dashiue Rd., 300 Hsinchu, Taiwan, ROC

Abstract

The execution performance of a stack-based Java virtual machine (JVM) is limited by the true data dependency. To enhance the performance of the JVM, a stack operations folding mechanism for the picoJava-I/II processor was proposed by Sun Microsystems to fold 42.3% stack operations. By comparing the continuous bytecodes with pre-defined folding patterns in instruction decoder, the number of push/pop operations in between the operand stack and the local variable could be reduced. In this study, an enhanced POC (EPOC) folding model is proposed to further fold the discontinuous bytecodes that cannot be folded in continuous bytecodes folding mechanisms. By proposing a stack re-order buffer (SROB) to help the folding check processes, the EPOC folding model can fold the stack operations perfectly with a small size of SROB implementation. Statistical data shows that the four-foldable strategy of the EPOC folding model can eliminate 98.8% of push/pop operations with an instruction buffer size of 7 bytes and the SROB size of eight entries.

Keywords: Java virtual machine; Stack operations folding; POC folding model; EPOC folding model; Java processor

1. Introduction

Internet has become the most feasible means of accessing information and performing electronic transactions. Java [1] is the most popular language used over the Internet owing to its portability, compact code size, object-oriented, multi-threaded nature, and write-once-run-anywhere characteristics. With these, Java is suitable for smart phones, PDAs, Internet TVs, or other consumer and embedded products.

The specification for a virtual machine that executes Java bytecodes is called Java virtual machine (JVM) [2,3]. JVM is a stack-based machine and most operations must push or pop data to or from the top of stack. This will cause serious data hazard due to true data dependence. A means of avoiding such a limitation, i.e. stack operations folding, was proposed by Sun Microelectronics [4–6]. While executing, pre-defined and pre-stored folding patterns are compared with bytecodes in instruction stream sequentially. In our earlier study, we proposed a systematic folding solution

* Corresponding author.
E-mail address: lrton@csie.nctu.edu.tw (L.-R. Ton).

1383-7621/02/$ - see front matter
PII: S1383-7621(02)00053-X

named producer, operator, and consumer (POC) folding model [7]. All bytecode instructions are classified into three major POC types and typical stack operations before folding are listed below:

Step 1: The producer writes data accessed from the constant register or local variable to the top of the operand stack.
Step 2: The operator gets data from the top of the operand stack.
Step 3: The operator (ALU/logic type instructions, branch type instructions or complex type instructions) operates on the accessed operand stack data.
Step 4: The operator writes the result back to the operand stack as needed.
Step 5: The consumer gets the data from the operand stack and writes it back to the local variable.

This procedure is also shown on the left-hand side of Fig. 1, with the numbers showing the execution flow.

If true data dependency occurs among stack instructions, we can fold them together by redirecting the data provided by the producer to the corresponding instruction, as depicted by step 1′ on the right-hand side (after folding) in Fig. 1. The execution flow will be changed to the following after folding:

Step 1′: The operator gets data directly from the source of producer.
Step 3: The operator (ALU/logic type instructions, branch type instructions or complex type instructions) operates on these data.
Step 5′: The operator writes the execution result back to the destination of the consumer directly as needed.

In this case, the number of stack accesses is reduced from five to three. Hence, the system performance can be increased greatly after folding.

In the POC folding model, two POC types of continuous bytecode instructions are sent into the POC folding check unit to determine whether they are foldable or not. Bytecodes that can be folded together form a so-called folded bytecode instruction (FBI). The further foldability check is also performed to determine whether these two bytecode instructions have the possibility to become a larger FBI with the next adjacent bytecode instruction. The generated information is sent back to the POC folding model recursively to find

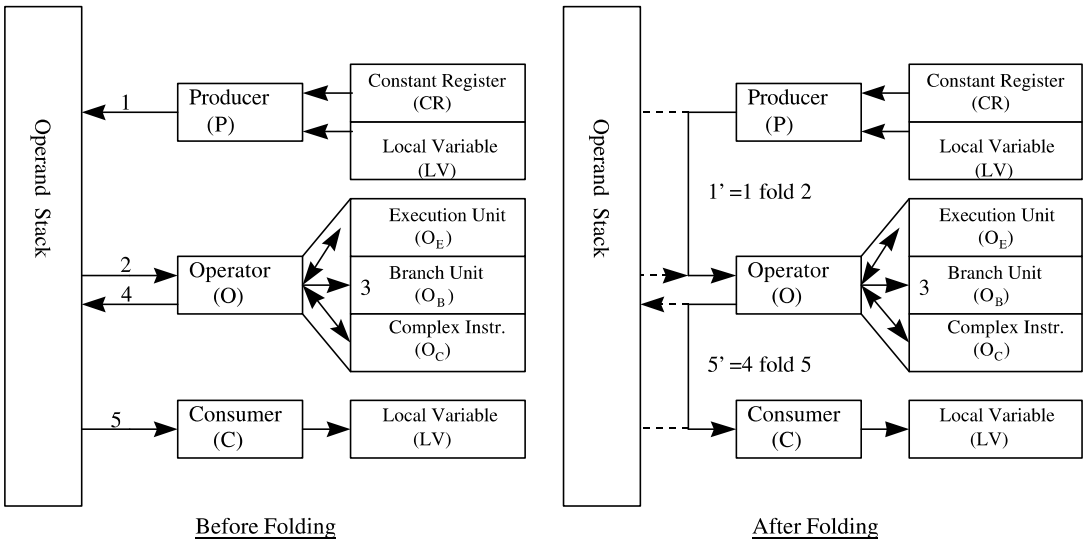

Fig. 1. Concept of stack operations folding.

out the maximum number of continuous bytecode instructions that form a FBI.

The limitation in both picoJava-I/II processor and the POC folding model is that they fold continuous bytecode instructions only. A new advanced POC model based-on the POC model was proposed by other researchers [8,9] to provide partial solutions. In those papers, additional folding patterns are pre-defined to fold the discontinuous bytecode instructions. With the newly added folding patterns, the bytecodes may be executed in an out-of-order manner. However, this may require additional exception handling circuitry for precise interrupt, which is not concerned in those papers [8,9]. Furthermore, some mistakes are found in those papers such that the simulation results may be incorrect.

In this paper, an enhanced POC (EPOC) folding model is proposed with proper cost-effective implementation to fold almost all foldable bytecodes. Unlike the pre-defined patterns in the advanced POC model, a stack re-order buffer (SROB) that acts like a general stack is proposed to assist the folding check process. Besides providing the sequential access nature of general stacks, the SROB provides out-of-order and multiple accesses capabilities by its register-like implementation. The EPOC folding model can fold all foldable bytecodes if the operands are kept in the scope of the SROB. In other words, if the size of SROB is large enough, all foldable bytecodes in the whole program can be folded completely. In this paper, we will show how to calculate the folding efficiency and how the proposed EPOC folding model achieving 100% folding efficiency with unlimited size of SROB.

This paper is organized as follows. Section 2 describes researches about stack operations folding. The POC-related folding models are presented according to their folding styles. Section 3 proposes the EPOC folding model and its corresponding folding procedures. Descriptions of both folding efficiency and how the EPOC folding model achieving 100% folding efficiency are shown. Section 4 shows the trace-driven simulation environment, benchmarks, and simulation results. Section 5 describes one possible hardware implementation of the EPOC folding model. Furthermore, the SROB architecture to keep precise interrupt in the Java processor is also given. Finally, a conclusion about stack operations folding is given in Section 5.

2. Background

Performance of the stack-based JVM suffers mainly from the sequential accessing of operands stack. Sun's solution revealed in their JavaChip family is the folding technique. The first implementation of the JVM in hardware is the Sun's picoJava-I/picoJava-II core design [4–6]. The folding technique is implemented in both picoJava-I and picoJava-II cores with folding capabilities of up to two and four bytecode instructions, respectively.

In the stack operations folding research, we classify different folding approaches into four categories—continuous-folding with patterns, continuous-folding without patterns, discontinuous-folding with patterns and discontinuous-folding without patterns, as described in the following four sub-sections.

2.1. Continuous-folding with patterns

By defining various opcode or instruction type combinations, the continuous-folding with patterns can be implemented using quite simple matching or table lookup circuitry. Researches about continuous-folding with patterns are parts of our early projects in 1997. In [10,11], different folding patterns with different cost/performance issues were proposed. Vijaykrishnan also proposed similar folding method in 1998 [12]. These researches proposed different sets of grouping rules like what Sun's picoJava-I and picoJava-II do with limited folding performance. In this paper, we use the picoJava-II as a representation of the continuous-folding with patterns.

As described in Sun's picoJava-II microarchitecture guide [6], bytecode instructions are classified into six types as shown in Table 1. The instruction folding unit (IFU) then examines the top seven bytes in the instruction buffer to

Table 1
Instruction types in picoJava-II core

Types	Descriptions
LV	A local variable load or load from global register or push constant
OP	An operation that uses the top two entries of stack and that produces a one-word result
BG2	An operation that uses the top two entries of stack and breaks the group
BG1	An operation that uses only the topmost entry of stack and breaks the group
MEM	A local variable store, global register store, and memory load
NF	A non-foldable instruction

Table 2
Grouping rules defined in picoJava-II core

First Bytecode	Second Bytecode	Third Bytecode	Fourth Bytecode
LV	LV	OP	MEM
LV	LV	OP	
LV	LV	BG2	
LV	OP	MEM	
LV	BG2		
LV	BG1		
LV	OP		
LV	MEM		
OP	MEM		

determine how many instructions can be folded (up to a maximum of four) according to the IFU grouping rules as shown in Table 2.

The main drawback of the continuous-folding with patterns is that only continuous bytecode instructions that exactly match the grouping rules can be folded. If the sequence of bytecode instructions matches no grouping rules, the bytecode will be executed in serial.

2.2. Continuous-folding without patterns

Further folding benefits can be achieved by applying the POC folding model, the previous research results of our team in 1998 [7]. As shown in Table 3, bytecode instructions in the POC folding model are classified into three types according to the usage of source and destination storage. 'O' type bytecodes are further divided into four subtypes according to their execution behavior.

Table 3
POC instruction types

POC	Descriptions	Occurrence (%)
P	An operation that pushes constant or loads variable from local variable to operand stack	41.09
O_E	An operation that will be executed in execution units	21.29
O_B	An operation that conditionally branches or jumps to target address	12.90
O_C	An operation that will be executed in microcoded ROM or trapped as a sequence of instructions	20.22
O_T	An operation that will force the folding check to be terminated for the difficulty in performing folding	2.30
C	An operation that pops the value from operand stack and stores it into local variable	2.29

In the POC folding model, foldability check is performed by examining each pair of consecutive instructions. By applying the POC folding rules, the two bytecode instructions may be combined into a new POC type, which is used in further foldability check with the following bytecode instructions. Consequently, the POC folding model is quite different from previous one because there is no fixed folding patterns. In the POC folding model, the valid folding combinations can be expressed as a regular expression as follows:

$$(PC + P^{+}O_E + P^{+}O_B + P^{+}O_C + P^{*}O_EC^{+} + P^{*}O_CC^{+})$$

According to the regular expression, the POC folding model can be implemented as a finite automaton to fold continuous bytecodes efficiently. The state diagram for the POC folding rules is shown in Fig. 2. In some states the P and C type instructions can be repeatedly added according to the source or destination operands of the following O or C type instructions. If more P's are provided then the O or C need, the extra P's will be executed sequentially.

For example, if the instruction sequence is `ILOAD_2`, `ICONST_2`, `ILOAD 5`, `IADD`, `IMUL`, `ISTORE 6`, their corresponding POC types are P, P, P, O_E, O_E, and C. By applying the POC folding rules, the first P must be executed lonely. The following three and the last two instructions will

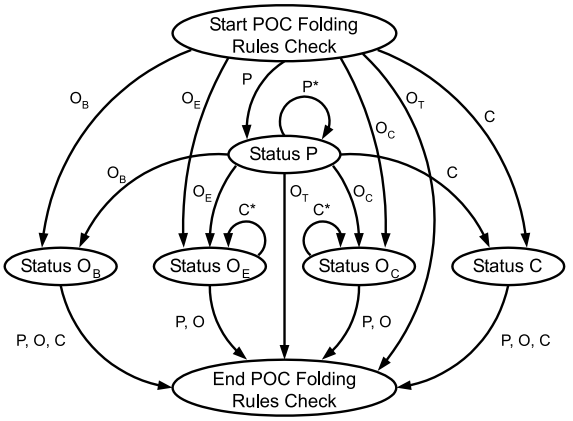

Fig. 2. Folding rules for POC model. (Remarks: Stars in the above figure are defined as kleene's star. The maximum number of repeating depends on the corresponding O or C type instruction.)

become two FBIs, which results the issued instructions per cycle (IIPC) of two for a single pipelined architecture. In this case, it is obvious that the first P is the first operand prepared for the `IMUL` instruction, and the result of the `IADD` instruction is the second. If we can fold the first P with the last two instructions, then these six instructions can be issued in two cycles with the IIPC increasing to three. This could be done using discontinuous-folding mechanisms in the following two sub-sections.

2.3. Discontinuous-folding with patterns

The researches of discontinuous-folding with patterns were proposed in September 2000 with the name of so-called advanced POC folding model [8,9]. Based on the POC folding model, four new folding sequence types are further added to fold the discontinuous bytecode instructions. As opposed to the original POC folding model, the number of sub-types for the O type bytecodes is reduced from four to two. The newly defined subtypes are O_P and O_C that represents the producible and consumable operators, respectively. With this definition type reduction, one of the main contributions in these researches is that the advanced POC folding model achieves higher folding ratio with a simpler folding circuitry (Table 4).

Table 4
POC Types in the advanced POC model (cited from [9])

Types	Definitions	Examples	Percentage (%)
P	Producers	`iconst_1`, `dload_3`	59.5
O_P	Producible operators	`iadd`, `fcmpl`	22.0
O_C	Consumable operators	`if_icmpeq`, `if_acmpne`	4.1
C	Consumers	`lastore`, `istore_0`	14.4

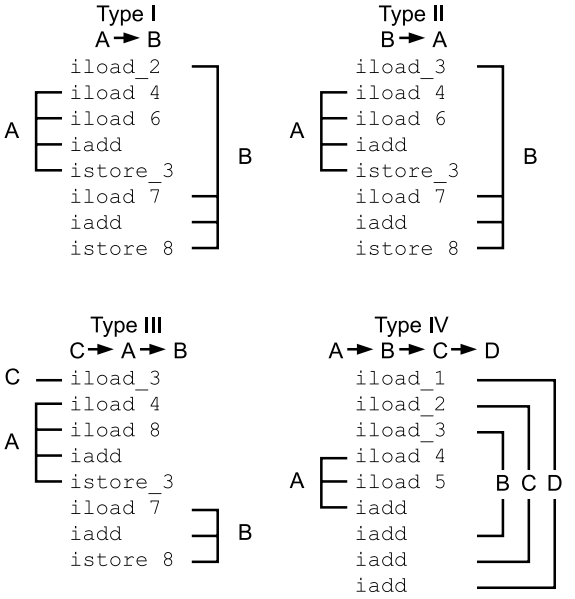

Fig. 3. New instruction sequence types (cited from [9]).

As shown in Fig. 3, discontinuous bytecodes that match one of the sequence types can be folded easily. In each sequence type, the execution order is slightly different with the consideration of data dependency. Consequently, out-of-order execution occurs except in the sequence type III. Statistical data [8,9] shows that the percentage of both type II and III is nearly zero. The proposed folding patterns with the percentages of the corresponding occurrence are shown in Table 5.

There are some mistakes in the advanced POC folding model. As shown in Table 4, the occurrence percentage of the O_P type and O_C type bytecodes is 26.1%. Researches [13] show that the

Table 5
Instruction folding patterns in the advanced POC model (cited from [9])

Instruction patterns	Percentages (%)	Instruction patterns	Percentages (%)
P–C	31.7	P–P–O_P–O_C	0.6
P–O_P–C	1.0	P–P–P–C	10.7
P–P–C	3.6	P–P–P–O_P–C	8.4
P–P–O_C	18.9	P–P–P–O_P–O_C	2.6
P–O_P–O_P–C	0.6	P–O_P–P–O_P–C	0.1
P–P–O_P–C	21.2	P–P–P–P–O_P–C	0.5

dynamic frequency of O type instruction is more than 50% except the 47.5% of mpegaudio benchmark. The difference may be caused by wrong definition of the POC types for each bytecode. As an example of the C type bytecodes in Table 4, the `lastore` bytecode is a memory array store instruction with a long data width of 64-bit [2]. Three source operands should be popped from the top of stack first and the array address is generated to perform the memory store operation. It is not applicable to load the array data from memory (`laload`: P), compute the result (`ladd`: O_P) and store to the array structure in memory (`lastore`: C) within an instruction cycle. Furthermore, enhancing the folding capability without considering the extra complexity for handling the new out-of-order problem may cause errors while executing the bytecodes. For example, if the bytecode sequence matches the type I in Fig. 3, the group A is executed prior to the group B. If the execution of group B causes an exception, how to decide the location of the restarting program counter (PC) may be a problem that is not resolved in the advanced POC folding model.

2.4. Discontinuous-folding without patterns

In this paper, we will propose a folding mechanism named as an EPOC folding model [14]. By defining no instruction folding patterns, the EPOC folding model can fold almost all the possible combinations in any Java bytecode sequences with limited hardware. Unlike the out-of-order execution manner in the advanced POC folding model, the bytecodes are issued in-order with a special designed SROB. In the following section, the EPOC folding mechanism is shown with a simple example to illustrate its folding effectiveness.

3. The EPOC folding model

The POC folding model handles the continuous folding well, and the EPOC folding model is designed to further fold the discontinuous P's with their corresponding O or C type instructions. In the following three subsections, we will introduce the key idea of EPOC folding model and an example to compare the POC, advanced POC and the EPOC folding model. How the EPOC folding model achieves 100% folding efficiency with unlimited SROB size is also shown.

3.1. EPOC folding model: folding by observing stack access behavior

The core concept of the EPOC folding model can be observed by the nature behavior of the stack accesses. Consequently, we will first introduce how to fold all foldable bytecodes by observing the stack access behavior. Here we use a simplified version of POC types similar to previous defined POC folding model. The P type bytecode pushes one item from local variable onto operand stack. The O type bytecode pops two items from top of operand stack and pushes one result item back. The C type bytecode pops one item from operand stack to local variable. With these three simplified POC types, an abstract stack machine (ASM) can be proposed to show the stack access behavior of the JVM. In the ASM, there are some features as listed below:

- From the bytecode's point of view, all bytecodes are consisted of these POC types.
- From stack item's point of view, all stack items are produced by P or O type bytecodes and only O or C type bytecodes can consume the items from stack.

In the POC folding model, only POC types of the bytecode stream are considered. The enhancement of the EPOC folding model is that the produced items are considered altogether. Here we use a simple example to show the behavior of the POC folding model and observe the chance for the EPOC folding model to fold more stack operations. If the instruction sequence is `ILOAD_2`, `ICONST_2`, `ILOAD 5`, `IADD`, `IMUL`, `ISTORE 6`, their corresponding POC types in the ASM are P, P, P, O, O, and C. The execution process in the POC folding model is shown in Fig. 4. In cycle 1, the first P is pushed onto the stack. The following P, P, and O are folded and the result shown as P′ is pushed onto stack in cycle 2. In cycle 3, C is folded with O and the source operands P and P′ are popped from the stack.

In Fig. 4, the key concept of EPOC folding model can be observed in the snapshot of cycle 2. The stack items in conjunction with the remaining bytecode sequence can be treated as a FBI consisting of four (P, P′, O, and C) types by using the same POC folding concept. In other words, if the stack items are considered for folding, the first P can be folded also. The only one problem for doing this is that the first P is pushed onto the operand stack already and further cycles are required to pop the P from the operand stack. Consequently, the P and P′ are not possible to be written onto operand stack and popped again for folding check. What we need is a new structure to temporary keep the information of P and P′ with multiple read/write capability. With this structure, the EPOC can fold discontinuous P's than the POC folding model. In summary, the design considerations of the EPOC folding model include the following to out-perform the original POC folding model:

- Besides the original bytecode sequence, the stack items are also included as the input data for EPOC folding check.
- A data structure must emulate the behavior of the original stack items and further provide the ability like a content-addressable memory with multiple read/write ports for EPOC folding check.
- Instead of issuing the discontinuous P's, the EPOC folding model must find out the first O or C type bytecode followed by discontinuous P's and log the discontinuous P's into the data structure.

In the EPOC folding model, a structure called SROB is provided for both continuous and discontinuous P's folding capabilities. With the SROB structure, discontinuous P's are logged into the SROB instead of issuing them sequentially. The processing steps of the EPOC folding model are shown in Fig. 5.

According to the EPOC folding steps, the same bytecode sequence in Fig. 4 can be executed in the manner shown in Fig. 6. Note that the operand stack is replaced by the newly proposed SROB structure.

3.2. A folding example for comparison

A simple trace that is commonly used in sorting is shown in Table 6. There are 27 bytecodes in this trace and their corresponding POC types are shown in the table. Five discontinuous P's who provide source operands but not used immediately by O or C type instructions are shown in bold italic style with subscripts from 'a' to 'e'. The O type instruction with ${}_{m}O_{n}$ notation means that it needs

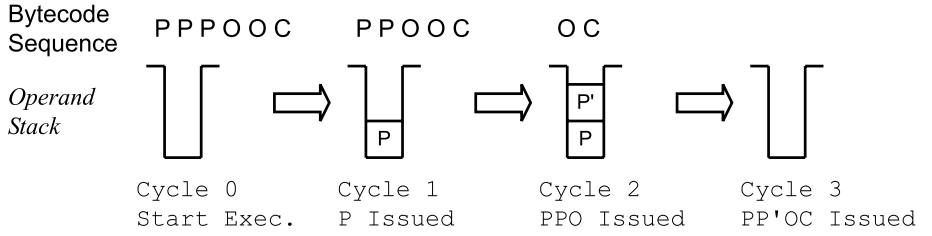

Fig. 4. Stack accesses of the sample bytecodes in the POC folding model.

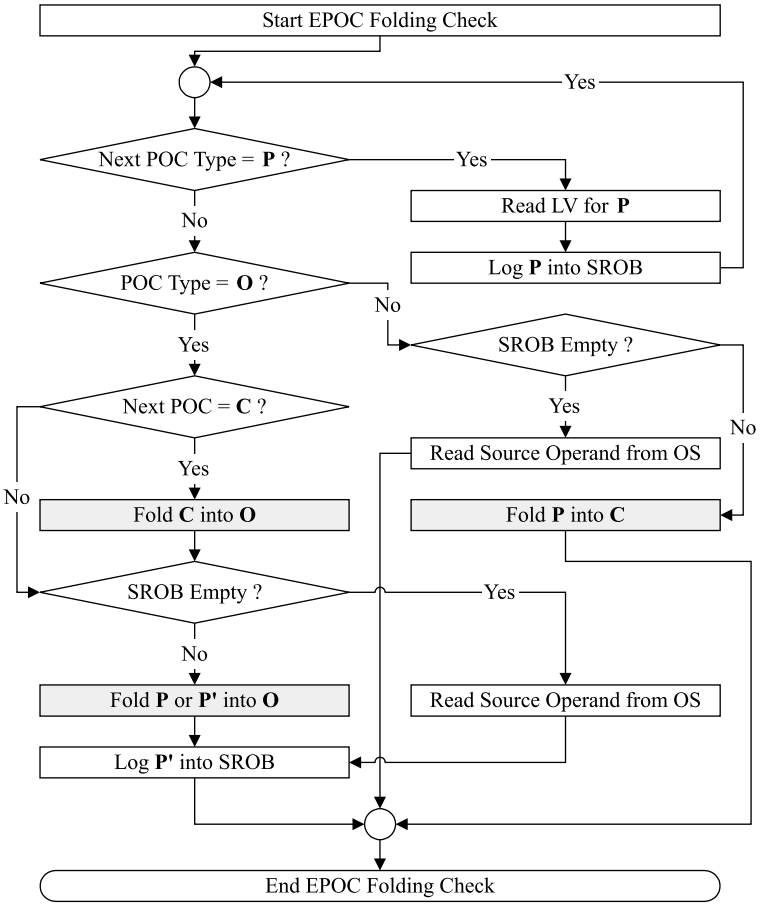

Fig. 5. Flow chart for the EPOC folding rules check.

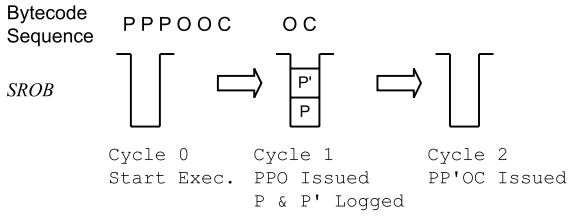

Fig. 6. Execution of the sample bytecodes in the EPOC folding model with SROB.

m source operands and produces n results from/to operand stack. In Table 7, the issued FBIs in each cycle are shown for POC, advanced POC and EPOC folding model. The P_i' symbols with parentheses are used to represent the result of O type bytecode stored in SROB.

In the above example, we can observe that the POC folding model tries to fold continuous bytecodes from the first bytecode in the instruction buffer. If the first bytecode cannot be folded with later bytecodes, the POC folding model issues the first bytecode to the execution unit. In contrarily, the EPOC folding model tries to fold bytecodes from the first O or C type instruction. This ensures that the bytecodes before the first O or C type instruction are all of P type. Consequently, P type bytecodes folding is done by selecting the nearest P's according to the number of source operands that the first O or C needs. At the same time, if the O type instruction does have to produce result values to operand stack, the EPOC folding will examine whether the bytecodes followed are of

Table 6
An example program slice and the corresponding POC types

Instructions #	Memory address	Bytecode trace	Types in POC	Types in advanced POC	Source statement
1	719	aload_1	P	P	if ($a[j] > a[j+1]$)
2	720	iload_3	P	P	
3	721	iaload	$_2O_1$	P	
4	722	aload_1	$\boldsymbol{P}_a$	P	
5	723	iload_3	P	P	
6	724	iconst_1	P	P	
7	725	iadd	$_2O_1$	O_P	
8	726	iaload	$_2O_1$	P	
9	727	if_icmple 750	$_2O_0$	O_C	
10	730	aload_1	P	P	$T = a[j]$
11	731	iload_3	P	P	
12	732	iaload	$_2O_1$	P	
13	733	istore 4	C	C	
14	735	aload_1	$\boldsymbol{P}_b$	P	$a[j] = a[j+1]$
15	736	iload_3	$\boldsymbol{P}_c$	P	
16	737	aload_1	$\boldsymbol{P}_d$	P	
17	738	iload_3	P	P	
18	739	iconst_1	P	P	
19	740	iadd	$_2O_1$	O_P	
20	741	iaload	$_2O_1$	P	
21	742	iastore	$_3O_0$	C	
22	743	aload_1	$\boldsymbol{P}_e$	P	$a[j+1] = T$
23	744	ilaod_3	P	P	
24	745	iconst_1	P	P	
25	746	iadd	$_2O_1$	O_P	
26	747	iload 4	P	P	
27	749	iastore	$_3O_0$	C	

C type instructions. As a result, C type instructions are folded to the O type instruction if the matching is found.

The folding behavior of the advanced POC folding model here shows that the folding patterns are not properly defined. The unsatisfied folding performance is not comparable to the other two models. The main advantage of the EPOC folding over the POC folding model is the foldability of discontinuous P type instructions. In the above example, the IIPC is enhanced from 1.8 to 2.7, or 50% enhancement with the SROB size of only four entries.

3.3. Achieving 100% EPOC folding efficiency

In this subsection, we will first show how to calculate the folding efficiency for folding performance evaluation. Next, the way to achieve 100% folding efficiency for the EPOC folding model is given as an implementation guideline.

The folding efficiency is calculated as follows:

$$\text{Folding efficiency} = \frac{\text{Number of folded stack operations}}{\text{Number of foldable stack operations}} \times 100\%$$

All P and C type bytecodes are of stack operations. The O type bytecodes require an ALU or other kinds of functional unit to execute and cannot be folded intrinsically. All the P type bytecodes are foldable. For C type bytecodes, there are two cases. First, if one C type bytecode is folded with one O type bytecode, we say that this kind of C type bytecode is foldable. Second, if one C type bytecode is folded with one P type bytecode, we say that P is folded into C and thus the C one is

Table 7
Issued FBIs in each folding model

Execution cycle	POC	Advanced POC	EPOC
1	P P $_2$O$_1$ (P′$_1$)	P	P P $_2$O$_1$ (P′$_1$)
2	***P***$_a$	P	P P $_2$O$_1$ (P′$_2$)
3	P P $_2$O$_1$ (P′$_2$)	P	***P***$_a$ P′$_2$ $_2$O$_1$ (P′$_3$)
4	***P***$_a$ P′$_2$ $_2$O$_1$ (P′$_3$)	P	P′$_1$ P′$_3$ $_2$O$_0$
5	P′$_1$ P′$_3$ $_2$O$_0$	P	P P $_3$O$_1$ C
6	P P $_2$O$_1$ C	P	P P$_2$O$_1$ (P′$_4$)
7	***P***$_b$	O$_P$	***P***$_d$ P′$_4$ $_2$O$_1$ (P′$_5$)
8	***P***$_c$	P	***P***$_b$ ***P***$_c$ P′$_5$ $_3$O$_0$
9	***P***$_d$	O$_C$	P P $_2$O$_1$ (P′$_6$)
10	P P $_2$O$_1$ (P′$_4$)	P	***P***$_e$ P′$_6$ P$_3$ O$_0$
11	***P***$_d$ P′$_4$ $_2$O$_1$ (P′$_5$)	P	–
12	***P***$_b$ ***P***$_c$ P′$_5$ $_3$O$_0$	P C	–
13	***P***$_e$	P	–
14	P P $_2$O$_1$ (P′$_6$)	P	–
15	***P***$_e$ P′$_6$ P $_3$O$_0$	P	–
16	–	P	–
17	–	P	–
18	–	O$_P$	–
19	–	P C	–
20	–	P	–
21	–	P	–
22	–	P	–
23	–	O$_P$	–
24	–	P C	–

non-foldable. With 100% folding efficiency and the assumption of a single-pipelined stack machine with the execution latency of one cycle for each bytecode, the total required execution cycles could be reduced to the number of intrinsically non-foldable bytecodes.

Here we start to discuss how to achieve 100% folding efficiency for the EPOC folding model. In a stack-based architecture, any high level language can be compiled into the postfix expression. While executing the postfix expression, a sequence of machine codes forms an instruction trace that can be expressed as a directed acyclic graph (DAG) with an inforest structure. A DAG is called an inforest if the out degree of every equals either zero or one. In an inforest, a vertex with an out degree equals to zero is called a root. An inforest is an intree if only one root is present. Any valid Java bytecode trace is either an inforest or an intree because the life cycle of a stack variable begins at stack pushing and ends by stack popping. This property holds for stack machines and is quite

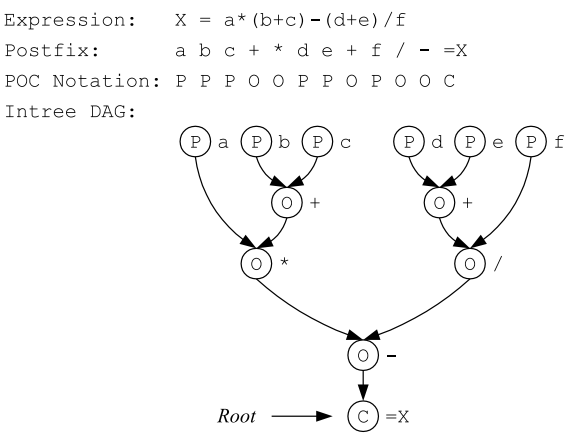

Fig. 7. An example of bytecode trace to Intree DAG expression.

different from register-based machines. As shown in Fig. 7, all bytecodes in the postfix expression are converted to P, O, and C types. The example is an intree and by using the post-order traversal of the DAG, the output sequence is exactly the same as the original bytecode trace. After execution, a valid trace will not leave any unused stack variables. If any unused stack variable is left, there must be some P or O type instruction sequences that should be treated as dead-codes. Here we assume that any valid bytecode trace will not leave any unused stack variables.

Here we conclude two cases of bytecode sequence as shown in Fig. 8 below.

Case I. The trace is a postfix expression that two sub-sequence of bytecodes provide source

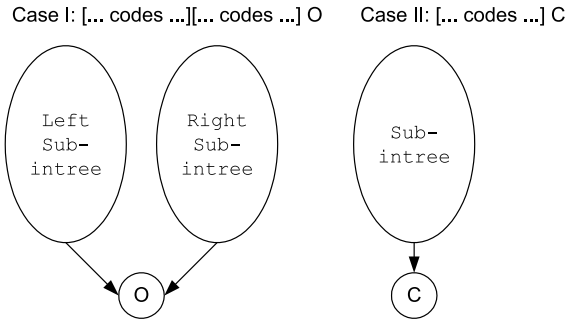

Fig. 8. Bytecode sequence analysis.

operands for the last O type bytecode. These two sub-sequences of bytecodes can be shown in two sub-intrees and the last O type bytecode is the root node of the DAG. In the EPOC folding model with the SROB, the O will get both source operands from SROB instead of the operand stack in the original stack architecture. Consequently, the left sub-intree and right sub-intree provides two source operands of P or P′ type to the O type bytecode. According to the EPOC folding model, this folding sequence (P or P′) (P or P′) O makes it to be folded into a single FBI. That is, if the discontinuous P's are written into SROB with large enough size, the O would fold the required P's from the SROB instead of popping it from the operand stack. Note that if the content provided by the SROB is P′, we say that the P′ is forwarded from the SROB instead of folding.

Case II. If the last bytecode in the trace is C, there must be some bytecodes providing the source for C, or the C will cause the stack underflow error. The bytecodes prior to C are expressed as a sub-intree in the right of Fig. 8. If the size of SROB is large enough, the source of C will be kept in the SROB and the C can read it out. According to the content types of top of the SROB, two sub-cases are considered. With the type of P this P will be folded into C and the P is folded into C. With the other type of P′, the C is said to be folded to the corresponding O type bytecode that generates this P′. Clearly, the C is folded earlier while performing the folding check for the corresponding O type bytecode and only one FBI is generated to include both O and C type bytecodes. Consequently, the C is folded by previous O one already.

Note that if the last node is of P type, one entry in the SROB is allocated because P is not dependent on any previous bytecode. Consequently, no extra cycles are required to push the discontinuous P's onto operand stack if the size of SROB is large enough. With the above analysis, we know that if the SROB size is large enough, all the discontinuous P's will be kept in the SROB for later folding. With the SROB, the exception can be easily handled because the bytecodes are logged into SROB in-order. That is, if we keep the program counter field for each bytecode logged into SROB, the precise interrupt can be handled like what is done in general superscalar processors [15].

4. Performance comparison of various folding models

In this section, the simulation environment and the benchmark suit are introduced. A formula for calculating folding performance gain is given. The folding ratio and issued instructions per cycle are given to compare the effectiveness of various folding models including picoJava-II, POC, and EPOC folding model.

4.1. Simulation environment

By modifying the Sun's JDK virtual machine [16], runtime bytecode traces are generated when the benchmark program is running. In this research, we developed a benchmark profiler and a trace-driven simulator with three different folding models for our performance study. The benchmark profiler scans the input traces and collects the information about instruction count, occurrence percentages of for three POC types, and occurrence frequency for each bytecode, etc. Three different folding models including picoJava-II, POC and EPOC folding models are built in the trace-driven simulator. We use the run-time traces collected from the SPECjvm98 benchmark [17] as our simulation source data. There are three input data set scales for the SPECjvm98 benchmarks: *s1*, *s10*, and *s100*. In this paper, we use *s10* data set as the simulation basis. As shown in Table 8, the instruction counts of the traces for the SPECjvm98 benchmarks are collected by the benchmark profiler. The detailed occurrence percentages for three POC types are shown in Table 9.

In order to analyze the folding performance gain associated with the eliminated stack operations or

Table 8
Dynamic instruction counts of SPECjvm98 benchmark suite

Trace names	Instruction counts
compress	1137M
db	74M
jack	341M
javac	63M
jess	121M
mpegaudio-3	1220M
raytracer	160M

execution cycles, we need to calculate the theoretical performance upper bound that stack operations folding can achieve. The theoretical performance upper bound is calculated by first finding the theoretical foldable instruction groups, then eliminating all foldable stack operations, and counting the resulted execution cycles. Finally, the speedup upper bound is calculated accordingly. The following equation calculates the overall speedup:

$$\begin{aligned}\text{Speedup} = &\{(\text{Cycles}_P + \text{Cycles}_O + \text{Cycles}_C)/(\text{Cycles}_P * \\ &(1 - \text{FP}_P) + \text{Cycles}_O + \text{Cycles}_C * \\ &(1 - \text{FR}_C * \text{FP}_C))\}\end{aligned}$$

where

Cycles_x = Execution cycles
FP_x = Folded percentage
FR_C = Max. Foldable Ratio for C

for x type bytecodes.

In Table 9, the average occurrence percentage for P and C type bytecodes are 43.38%. Unlike the P type bytecodes that could be folded completely, not all of the C type bytecodes could be folded. The only two foldable combinations for the C type bytecodes are $P + C$ and $O_E + C$. In the case of $P + C$ folding, the P type bytecode is folded into the C type bytecode to write the result back to the local variable. Consequently, only the C type bytecodes in the $O_E + C$ combination could be folded. In the case of C type bytecodes, only 40% (FR_C) of them could be folded. This also indicates that the maximum performance speedup with the folding efficiency of 100% can be derived from the above formula. According to the formula, the ideal speedup for the SPECjvm98 benchmark is 1.74 if all foldable P and C type bytecodes are folded.

4.2. Simulation results for various folding models

The following simulation results are gathered using the parameter of a 7-byte instruction buffer and an 8-entry SROB. As shown in Fig. 9, the percentages of folded P type operations for each folding model are compared.

For POC and EPOC folding model, the foldability with 2, 3, 4 and unlimited number of instructions that can be folded together are simulated. Results show that three-foldable is enough for both POC and EPOC folding models if we want to fold P type bytecodes only. With the information from Table 9, the average occurrence

Table 9
Occurrence percentages of the POC types

Trace names	P (%)	O_{ALL}				O_{ALL} (%)	C (%)
		O_E (%)	O_B (%)	O_C (%)	O_T (%)		
compress	40.02	26.24	8.54	19.61	1.39	55.77	4.21
db	44.14	20.48	13.51	14.13	5.63	53.76	2.10
jack	32.67	25.04	13.10	27.87	0.46	66.48	0.85
javac	41.82	15.00	14.76	21.94	3.31	55.01	3.16
jess	44.00	9.31	19.78	20.72	2.43	52.23	3.77
mpegaudio-3	45.61	38.00	4.20	8.99	1.85	53.04	1.35
raytracer	39.35	14.28	16.44	28.27	1.03	60.03	0.62
Average	41.09	21.19	12.90	20.22	2.30	56.62	2.29

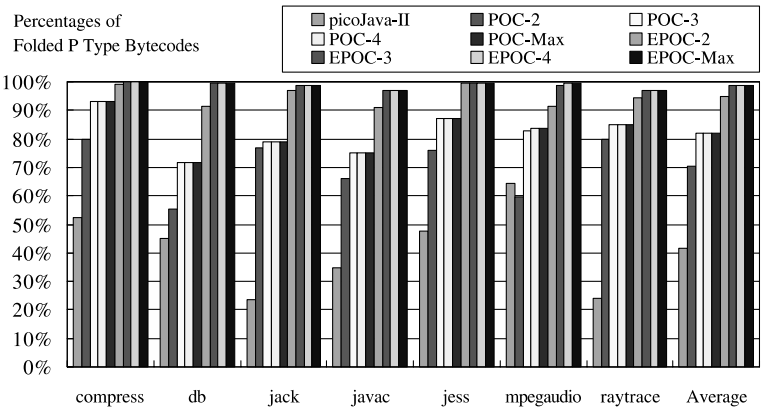

Fig. 9. Percentages of folded P type bytecodes.

percentage of P type bytecodes is 41.09% in the whole program. Consequently, the Java processor would execute 82.8%, 66.4% and 59.5% of the original bytecodes with the picoJava-II, three-foldable POC and three-foldable EPOC folding mechanisms, respectively.

In Fig. 10, the percentages of folded C type bytecodes are shown. The average occurrence percentage of C type bytecodes is far less than the occurrence percentage of P type bytecodes. Architecture designers of Java processors may treat the folding circuitry for C type bytecodes as a design option. Simulation shows that using a four-foldable POC or EPOC folding mechanism can fold all the foldable C type bytecodes.

Fig. 11 shows the folding efficiency for each folding model. If the POC and the EPOC folding models can fold up to four bytecodes like picoJava-II, the average percentages of folded stack operations are 42.32%, 82.9% and 98.83% for the folding mechanism of picoJava-II, POC and EPOC folding model, respectively. Note that the SROB size for the EPOC folding model is only eight entries.

The numbers of IIPC for a single pipelined picoJava-II architecture are shown in Fig. 12. The average numbers of IIPC are 1.25, 1.54 and 1.74 for the folding mechanism of picoJava-II, POC and EPOC folding model, respectively. As mentioned in Section 4.1, the average upper bound of

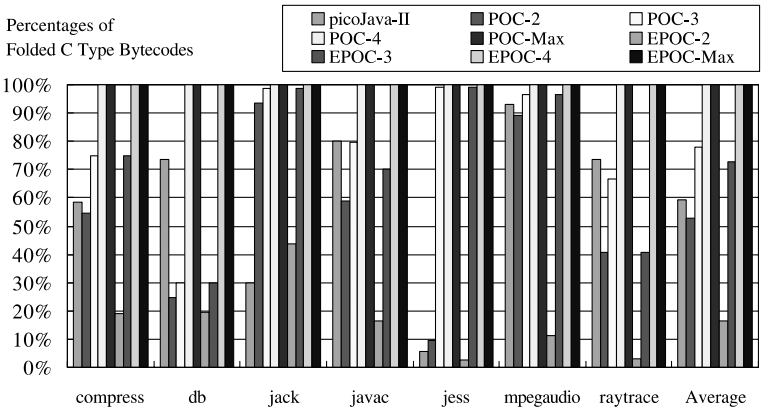

Fig. 10. Percentages of folded C type bytecodes.

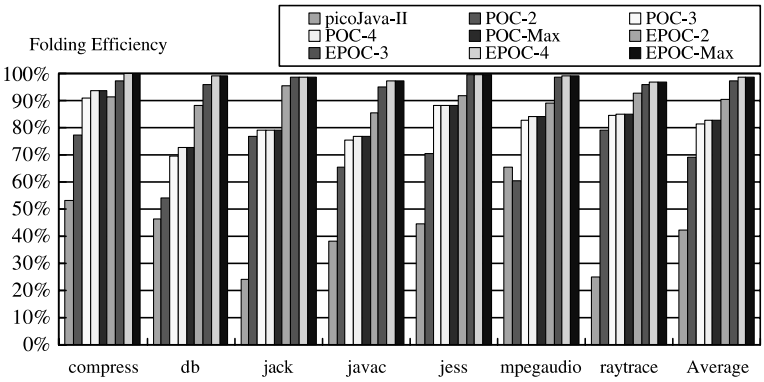

Fig. 11. Folding efficiency for each folding model.

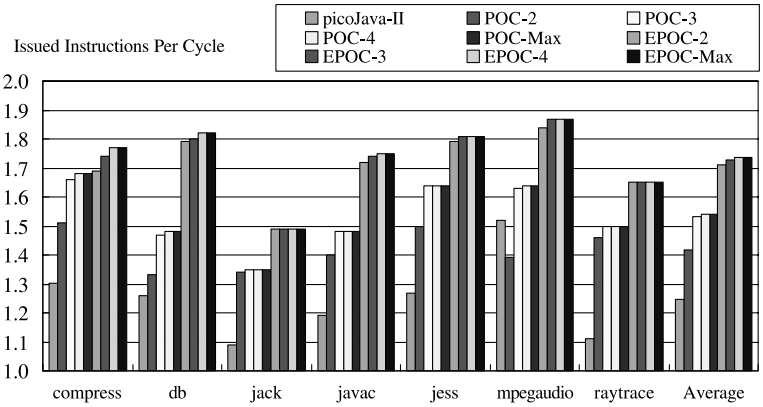

Fig. 12. Issued instructions per cycle for each folding model.

IIPC is 1.74. This reveals that with the SROB size of only eight entries, the EPOC folding model almost achieves the highest folding efficiency as compared to all other folding models.

5. Conclusion

5.1. Summary and discussion

In this paper, we have proposed the EPOC folding model based on the previously proposed POC folding model. Discontinuous-folding is shown more powerful and overrides the continuous-folding used in picoJava-II. Furthermore, the folding mechanism with patterns is shown more restricted than the one without patterns. With the EPOC folding model, the folding ratio is higher than the picoJava-II for 133%.

The performance enhancement from POC to EPOC folding model benefits mainly from the foldability of discontinuous P type instructions by the proposed SROB structure of only eight entries. Precise exception and data forwarding are also provided for both execution correctness and performance enhancement requirements. According to the simulation results, the four-foldable strategy which folds up to four bytecodes for POC and EPOC folding model eliminates 82.9% and 98.8% of foldable bytecodes, respectively. For all the

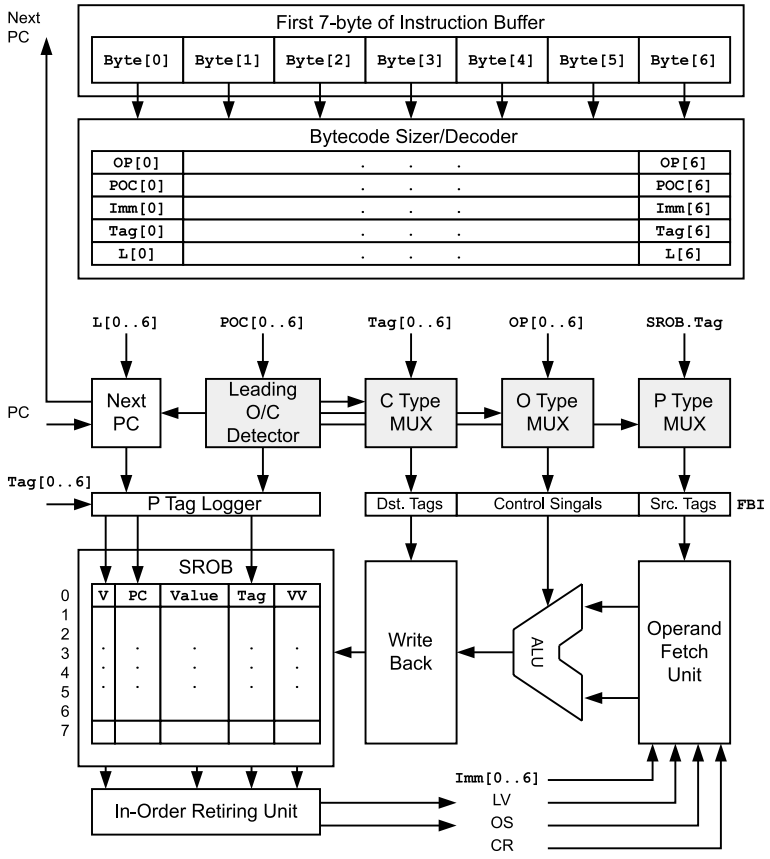

Fig. 13. Overview of an EPOC java processor architecture.

bytecodes, the percentages of eliminated instructions for POC and EPOC are 36% and 43%, respectively. In other words, execution performance is greatly enhanced because less than 60% of bytecodes are executed by the single-pipelined Java processor with the EPOC folding model.

5.2. Sample application: An EPOC Java processor

Here we show one possible hardware implementation of the EPOC Java processor. As shown in Fig. 13, the design parameters are similar with the design of the Sun's picoJava-II processor. The EPOC folding design shown in gray scale blocksand the SROB with corresponding fields are given. Detailed descriptions would probably exceed the scope of the paper and thus not shown here.

In our future research, the SROB will play an important role in a superscalar Java processor. By using the EPOC folding model with SROB, multiple FBIs might be issued in parallel to exploit higher ILP with lower hardware cost as compared to traditional superscalar processors.

References

[1] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, Reading MA, 1996.
[2] T. Lindholm, F. Yellin, The Java™ Virtual Machine Specification, Addison-Wesley, Reading MA, 1996.
[3] B. Venners, Inside the Java Virtual Machine, McGraw Hill, New York, 1998.
[4] M. O'Connor, M. Tremblay, picoJava-I: The Java virtual machine in hardware, IEEE Micro 17 (2) (1997) 45–53.

[5] H. McGhan, M. O'Connor, picoJava: a direct execution engine for Java bytecode, IEEE Computer (1998) 22–30.
[6] Sun Microsystems Inc., picoJava-II Microarchitecture Guide, Sun Microsystems, CA, USA, March 1999.
[7] L.C. Chang, L.R. Ton, M.F. Kao, C.P. Chung, Stack operations folding in Java processors, IEE Proceedings on Computer and Digital Techniques 145 (5) (1998).
[8] A. Kim, M. Chang, An advanced instruction folding mechanism for a stackless Java processor, in: Proceeding of the International Conference on Computer Design (ICCD), September 2000, pp. 565–566.
[9] A. Kim, M. Chang, Advanced POC model-based Java instruction folding mechanism, in: Proceedings of 26th EUROMICRO Conference, vol. 1, September 2000, pp. 332–338.
[10] H.-M. Tseng et al., Performance enhancement by folding strategies of a Java processor, in: Proceedings of International Conference on Computer Systems Technology for Industrial Applications—Internet and Multimedia, 1997.
[11] L.-R. Ton et al., Instruction folding in Java processors, in: the International Conference on Parallel and Distributed Systems, 1997.
[12] N. Vijaykrishnan, N. Ranganathan, R. Gadekarla, Object-oriented architectural support for a Java processor, in: Proceedings of the ECOOP'98, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1998.
[13] R. Radhakrishnan, J. Rubio, L.K. John, Characterization of Java applications at bytecode and ultra-SPARC machine code levels, in: Proceedings of IEEE International Conference on Computer Design (ICCD), 1999, pp. 281–284.
[14] L.-R. Ton, L.-C. Chang, C.-P. Chung, Exploiting Java bytecode parallelism by dynamic folding model, in: Proceedings of the 6th International Euro-Par Parallel Processing Conference Lecture Notes in Computer Science, vol. 1900, August 2000, pp. 994–997.
[15] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
[16] Sun Microsystems Inc., Java Development Kit 1.x. Available from <http://www.javasoft.com/products/jdk/>.
[17] Standard Performance Evaluation Corporation, SPECjvm98 Benchmark. Available from <http://www.spec.org/osg/jvm98/>.

Lee-Ren Ton received the B.S. and M.S. degrees in Information Engineering and Computer Science from the Feng Chia University, Taiwan, in 1993 and 1995, respectively. He was a lecturer of the Department of Computer Science and Information Engineering at the National Chiao Tung University, Taiwan, while working towards the Ph.D. degree. Currently, he is a Ph.D. candidate of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. His research interests include computer architecture, microprocessor system design, and VLSI/IP design.

Lung-Chung Chang received the B.S. degree in Electrical Engineering from the Chung Yuan Christian College, Taiwan, in 1977, and the M.S. degree from the University of Southern California in 1987. Currently, he is a Ph.D. candidate of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. Moreover, he is with the Computer & Communications Research Laboratories (CCL) of Industrial Technology Research Institute (ITRI) as a Manager. His research interests include computer architecture and parallel processing.

Chung-Ping Chung received the B.S. degree from the National Cheng Kung University, Taiwan, in 1976, and the M.S. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in electrical engineering. He was a lecturer in electrical engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Taiwan, where he is a professor. From 1991 to 1992, he was a visiting associate professor of computer science at the Michigan State University. From 1998, he joins the Computer and Communications Laboratories (CCL), Industrial Technology Research Institute (ITRI), Taiwan, as the Director of the Advanced Technology Center (ATC), and then the Consultant to the General Director. He is expected to return to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.