ELSEVIER

Microprocessors and Microsystems 26 (2002) 341–352

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

Design of an optimal folding mechanism for Java processors

Lee-Ren Ton[a,*], Lung-Chung Chang[a,b], Jyh-Jiun Shann[a], Chung-Ping Chung[a]

[a]*Department of Computer Science and Information Engineering, National Chiao Tung University, No. 1001, Dashiue Rd., Hsinchu 300, Taiwan, ROC*
[b]*Computer and Communications Research Laboratories, Industrial Technology Research Institute Bldg. 51, No. 195-11, Sec. 4, Jungshing Rd., Judung Jen, Hsinchu 310, Taiwan, ROC*

Received 16 May 2001; revised 10 February 2002; accepted 10 May 2002

Abstract

Java has become the most important language in the Internet area, but its execution performance is severely limited by the true data dependency inherited from the stack architecture defined by the Sun's Java Virtual Machine (JVM). To enhance the performance of the JVM, a stack operations folding mechanism for the picoJava-II processor was proposed by Sun Microsystems to fold 42.3% stack push/pop instructions. A systematic folding algorithm—Producer, Operator, and Consumer (POC) folding model was proposed in the earlier research to eliminate up to 82.9% of stack push/pop instructions. The remaining push and pop instructions cannot be folded due to the sequential checking characteristic of the POC folding model. A new folding algorithm—enhanced POC (EPOC) folding model is proposed in this paper to further fold the remaining push and pop instructions. In the EPOC folding model, stack push/pop instructions are folded with the proposed Stack Reorder Buffer (SROB) architecture. With a small SROB size of 584 bits, almost all of the stack push/pop instructions can be folded with the precise exception handling capability. Statistical data shows that 98.8% of the stack push/pop instructions can be folded, and the average execution performance speedup of a 4-foldable processor with a 7-byte instruction buffer is 1.74 as compared to a traditional single-pipelined stack machine without folding.

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 with the machine code named bytecode. The execution performance of JVM is limited by true data dependency among bytecodes. A means of avoiding such a limitation, i.e. Stack Operations Folding [4–6], was proposed by Sun Microsystems. In this paper, we give the definition of the Stack Operations Folding as below:

Definition. *Stack Operations Folding*. The ability to detect some instructions with true data dependency in the instruction flow of a stack machine, and execute these instructions collectively in some way like a single, compound instruction.

Assume that we want to perform an arithmetic expression $C = A + B$, the four generated bytecodes and the execution behavior of a Java processor without Stack Operations Folding is shown in Fig. 1.

According to the JVM specification [2], JVM defines various runtime data areas that are used during execution of a program. As shown in Fig. 1, the Local Variable (LV) and the Operand Stack (OS) are used to store data and partial results for each execution method. The six dashed lines are used to represent the accesses of the OS and the three solid lines are used to indicate the accesses of the LV. All four bytecodes require thirteen execution cycles in a single-pipelined Java processor as shown in Fig. 2. The six pipeline stages are the same as the Sun's picoJava-II processor. The abbreviated symbols are F (Instruction Fetch), D (Instruction Decode), R (Operand Read), E (Execution), C (Cache Access), and W (Write Back) [6]. The gray ellipses indicate

* Corresponding author.
E-mail addresses: lrton@csie.nctu.edu.tw (L.R. Ton), lcchang@ccl.itri.org.tw (L.C. Chang).

0141-9331/02/$ - see front matter
PII: S0141-9331(02)00042-X

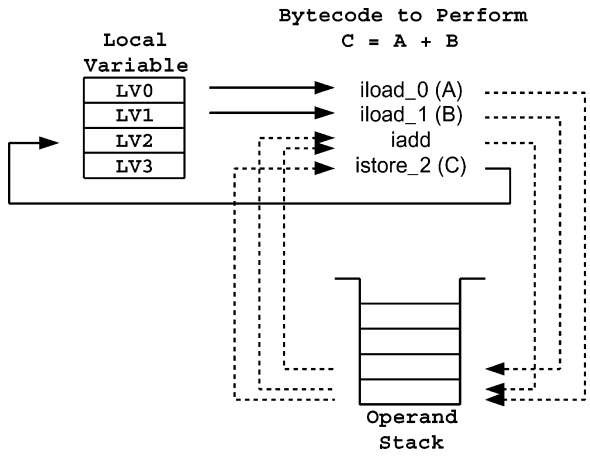

Fig. 1. Execution behavior of a Java Processor without Stack Operations Folding.

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
iload_0	F	D	R	E	C	W							
iload_1		F	D	R	E	C	W						
iadd			F	D			R	E	C	W			
istore_2				F	D					R	E	C	W

Fig. 2. Execution cycles of a six-stage single-pipelined Java processor without Stack Operations Folding.

the pipeline bubbles introduced by true data dependency among OS accesses.

In the above example, we can observe that some bytecodes define some values onto the OS and some bytecodes that are true dependent on them use those values. The define-use chain is linked by role of the OS. Consequently, Stack Operations Folding tries to observe these true dependence relations among original accesses of OS and performs direct data linking among decoded bytecodes instead of multiple accesses of OS. The execution behavior of a Java processor with Stack Operations Folding is shown in Fig. 3.

In order to detect the true data dependency among accesses of the OS, the Stack Operations Folding mechanism requires wider instruction fetcher and decoder to provide multiple decoded bytecodes in each cycle. This seems not a good idea for a low-cost oriented Java processor design. In fact, the JVM itself compensates this by its compact code size of the stack-based zero-address machine architecture. Study shows that the average bytecode size is only 1.8 bytes [4]. Another proof of this is the Sun's picoJava-II processor design with the decoder width of 7-byte to provide the maximum of four bytecodes for folding check in each cycle [6].

In Fig. 3, if we can decode these four bytecodes simultaneously in one cycle, we can know all the required source/destination fields of each bytecode. By comparing all the source/destination fields, the data dependency relations can be determined and thus Stack Operations Folding can be performed. In this example, we can observe that stack push/pop bytecodes (iload_0, iload_1, and istore_2) either provide or consume data to/from OS. They do not need any functional unit to be physically executed. Instead, only data movements are performed for these bytecodes. The only one bytecode to be executed is the integer addition (iadd). As shown in Fig. 4, decoded bytecodes with source/destination fields are folded together to form the so-called Folded Bytecode Instruction (FBI) with all true data dependency among accesses of OS been eliminated. As shown in Fig. 5, the number of required execution cycles is five as compared to thirteen without Stack Operations Folding.

In Fig. 5, we can observe that all four bytecodes are fetched and decoded together. The stack push/pop bytecodes are folded to the iadd bytecode and it will fetch the source operands according to the reassigned source fields. Similarly, when the iadd is about to write the result back, the

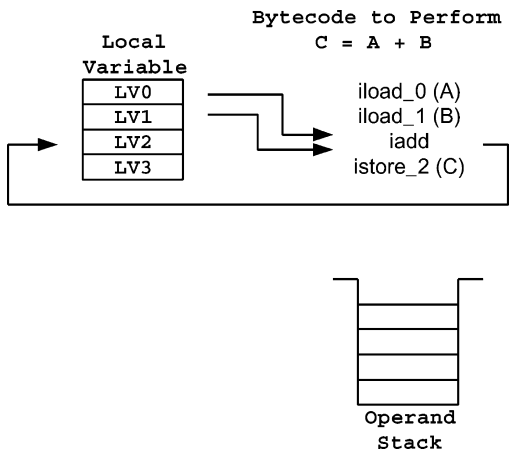

Fig. 3. Execution behavior of a Java Processor with Stack Operations Folding.

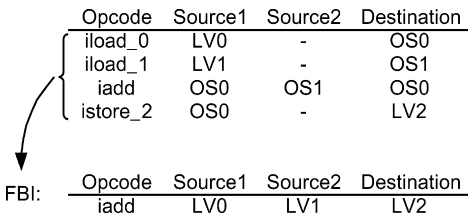

Fig. 4. Performing Stack Operations Folding by re-assigning Source/Destination fields of bytecodes.

reassigned destination field indicates the result to be written to LV instead of OS. By performing Stack Operations Folding, a single-pipelined Java processor can break the theoretically upper bound of the Instructions per Cycle (IPC) with the maximum value of one. This is the major contribution for enhancing the execution performance of Java processors. Various Stack Operations Folding mechanisms are proposed to detect as more foldable bytecodes as possible. In this paper, we will propose an optimal folding mechanism to fold all foldable bytecodes within Java methods and shown its highly applicable design.

This paper is organized as follows. Section 2 describes researches about Stack Operations Folding. Section 3 proposes the EPOC folding model and its corresponding folding procedures. Section 4 describes the hardware implementation of the EPOC folding model. Furthermore, the SROB architecture to keep precise interrupt in the Java processor is also given. Section 5 shows the trace-driven simulation environment, benchmarks, and simulation results. Finally, a conclusion about Stack Operations Folding is given in Section 6.

2. Related work

Performance of the stack-based JVM suffers mainly from the sequential accessing of OS. Sun's solution revealed in their JavaChip family is the folding technique. The first implementation of the JVM in hardware is Sun's picoJava-I and picoJava-II cores design. The folding technique is implemented in both picoJava-I and picoJava-II cores with folding capabilities of up to 2 and 4 bytecodes, respectively [4–6].

In the Stack Operations Folding researches, 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 Sections 2.1–2.4.

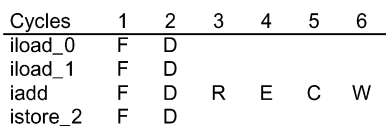

Fig. 5. Execution cycles of a six-stage single-pipelined Java processor with Stack Operations Folding.

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 comparison circuitry. Researches about continuous-folding with patterns are parts of our early projects in 1997. In Refs. [7,8], different folding patterns with different cost/performance issues were proposed. Vijaykrishnan also proposed similar folding method in 1998 [9]. 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, bytecodes are classified into six types as shown in Table 1 [6]. The Instruction Folding Unit (IFU) then examines the top 7 bytes in the instruction buffer to determine how many instructions can be folded (up to a maximum of four) according to the IFU grouping patterns as shown in Table 2 [6].

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

2.2. Continuous-folding without patterns

Further folding benefits can be achieved by applying the Producer, Operator and Consumer (POC) folding model, the previous research results of our team in 1998 [10]. As shown in Table 3, bytecodes in the POC folding model are classified into three types according to the usage of source

Table 1
Instruction types in picoJava-II core (cited from Ref. [6])

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 patterns in picoJava-II core (cited from Ref. [6])

1st Bytecode	2nd Bytecode	3rd Bytecode	4th Bytecode
LV	LV	OP	MEM
LV	LV	OP	
LV	LV	BG2	
LV	OP	MEM	
LV	BG2		
LV	BG1		
LV	OP		
LV	MEM		
OP	MEM		

and destination storage. 'O' type instructions are further divided into four sub-types according to their execution behavior.

In the POC folding model, foldability check is performed by examining each pair of consecutive bytecodes. By applying the POC folding rules, the two bytecodes may be combined into a new POC type, which is used in further foldability check with the following bytecodes. Consequently, the POC folding model is quite different from previous one because there is no fixed folding patterns. The POC folding rules can be represented as a state diagram shown in Fig. 6. 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 bytecodes. If more Ps are provided then the O or C need, the extra Ps will be executed sequentially.

For example, if the bytecode 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 bytecodes will 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 bytecode, and the result of the iadd bytecode is the second. If we can fold the first P with the last two bytecodes, then these six bytecodes can be issued in two cycles with the IIPC of three. This could be done using discontinuous-folding mechanisms in Sections 2.3 and 2.4.

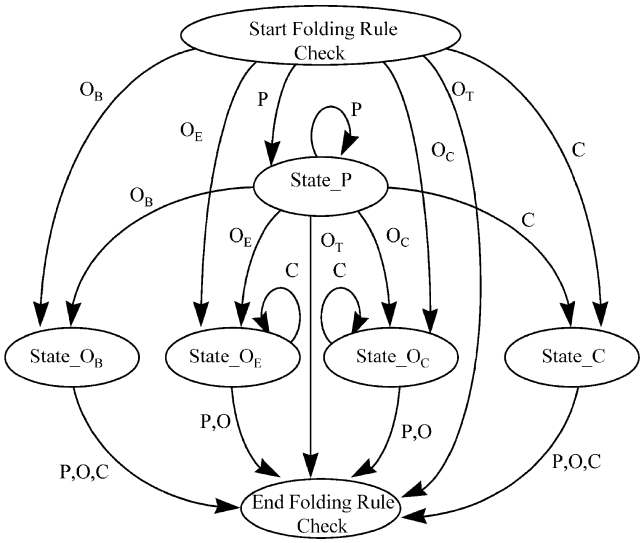

Fig. 6. Folding rules for POC Folding Model.

2.3. Discontinuous-folding with patterns

The researches of discontinuous-folding with patterns were proposed in September 2000 with the name of the so-called advanced POC model [11,12]. Based on the POC folding model, four new folding sequence types are further added to fold the discontinuous bytecodes. 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. The occurrence percentages for each type used in the advanced POC model are shown in Table 4.

As shown in Fig. 7, the execution order is rearranged according to data dependency. Consequently, out-of-order execution occurs except in the sequence type III. Statistical data shows that the percentage of both types II and III is nearly zero [11,12]. The proposed folding patterns with the percentages of the corresponding occurrences are shown in Table 5.

There are some mistakes in the advanced POC model. As shown in Table 4, the occurrence percentage of the O_P type and O_C type bytecodes is 26.1%. Researches show that the dynamic frequency of O type instruction is about 50% [13,

Table 3
POC types

POC	Descriptions	Occurrence (in %)
P	A bytecode that pushes constant or loads variable from LV to OS	41.09
O_E	A bytecode that will be executed in functional units	21.29
O_B	A bytecode that conditionally branches or jumps to target address	12.90
O_C	A bytecode that will be executed in micro-coded ROM or trapped as a sequence of bytecodes	20.22
O_T	A bytecode that will force the folding check to be terminated for the difficulty in performing folding	2.30
C	A bytecode that pops the value from OS and stores it into LV	2.29

Table 4
POC types in the Advanced POC Model (cited from Ref. [12])

Type	Definition	Example	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

14]. 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 data from memory (laload: P), compute the result (ladd: O_P) and store to memory (lastore: C) within an instruction cycle. Furthermore, many frequently occurred patterns such as P–O_P and P–P–O_P are not included in Table 5. This will lead the folding performance degrade dramatically and even worse than the original POC folding model. For the correctness of the paper, we do not show their folding performance in Section 5.

2.4. Discontinuous-folding without patterns

In this paper, we will propose an optimal folding mechanism named as an Enhanced POC (EPOC) folding model [15]. Without defining any folding pattern, the EPOC folding model can fold almost all the possible combinations in Java bytecode sequences using the proposed SROB architecture. Unlike the out-of-order execution manner in the advanced POC model, the bytecodes are issued in-order to the SROB in the EPOC folding model. In the following section, the EPOC folding mechanism is shown with an overview of the proposed Java processor architecture.

3. The EPOC folding model and overview of the proposed Java processor architecture

The POC folding model handles the continuous folding well, and the EPOC folding model is designed to further fold the discontinuous Ps with their corresponding O or C type bytecodes. In the following two subsections, we will introduce the EPOC folding rules and the overview of the proposed Java processor architecture.

Table 5
Folding patterns in the Advanced POC Model (cited from Ref. [12])

Instruction pattern	Percentage	Instruction pattern	Percentage
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

3.1. EPOC folding rules

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 folding rules can be simplified to the following two rules:

- P type bytecode can be folded into the following adjacent C or O type bytecode except O_T type bytecode.
- C type bytecode can be folded into previous adjacent O_E or O_C type bytecode.

In the original POC folding model, if the number of Ps is greater than the requirement of the corresponding O or C type bytecode, the extra Ps are issued sequentially. In the EPOC folding model, the extra Ps are logged in the SROB instead of issuing them sequentially. The processing steps of the EPOC folding model are shown in Fig. 8 with the following symbols as described below:

- SROB.WP #: the number of write ports of the SROB.
- FBI.Source #: the number of source operands required for an O or C type bytecode.
- FBI.Result #: the number of results generated by an O_E or O_C type bytecode.

In Fig. 8, if the number of Ps in the instruction buffer is greater than the number of write ports of the SROB, these Ps will be logged in-order to SROB for further folding check in

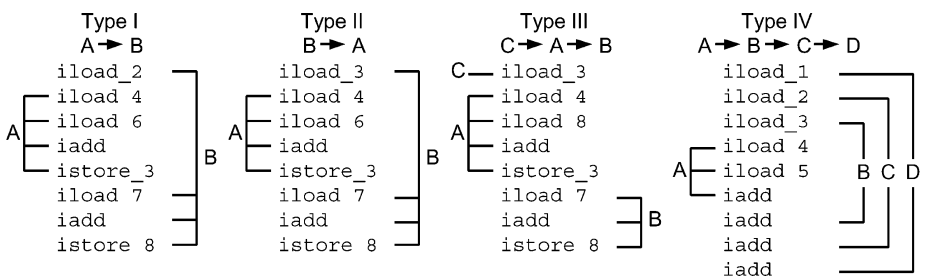

Fig. 7. New instruction sequence types in the Advanced POC Model (cited from Ref. [12]).

FBI.Result #: the number of results generated by an O_E or O_C type *bytecode.*

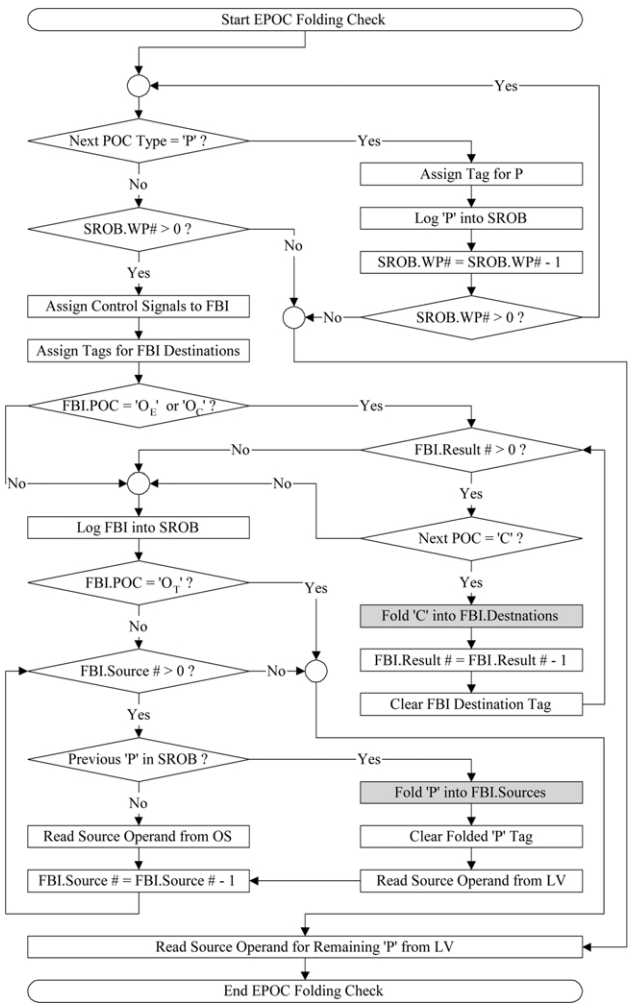

Fig. 8. Flow Chart for the EPOC Folding Rules Check.

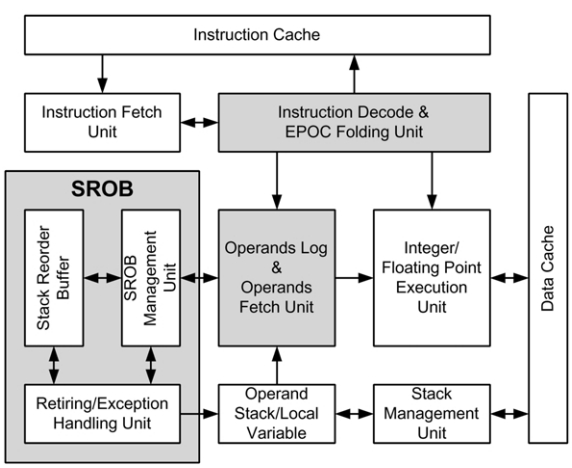

Fig. 9. Block diagram of the proposed Java architecture with EPOC folding.

the following cycles. The source values of these Ps will be fetched in the operand fetch stage and this prevents the anti-dependency from occurring on these Ps. Further description about the SROB will be shown in Section 4. The folding-related units of both P and C type bytecodes are shown in grayscale. If a P/C type bytecode can be folded into an FBI, the EPOC folding model just replaces the source/destination field of the FBI by the P/C type's source/destination tag.

3.2. Overview of the proposed Java processor architecture

In our Java processor architecture as shown in Fig. 9, we use six pipeline stages similar to picoJava-II's design [6]. The six pipeline stages are described below in the sequence in which they occur:

- *Fetch.* The Instruction Fetch Unit fetches bytecodes either from the instruction cache or from external memory.

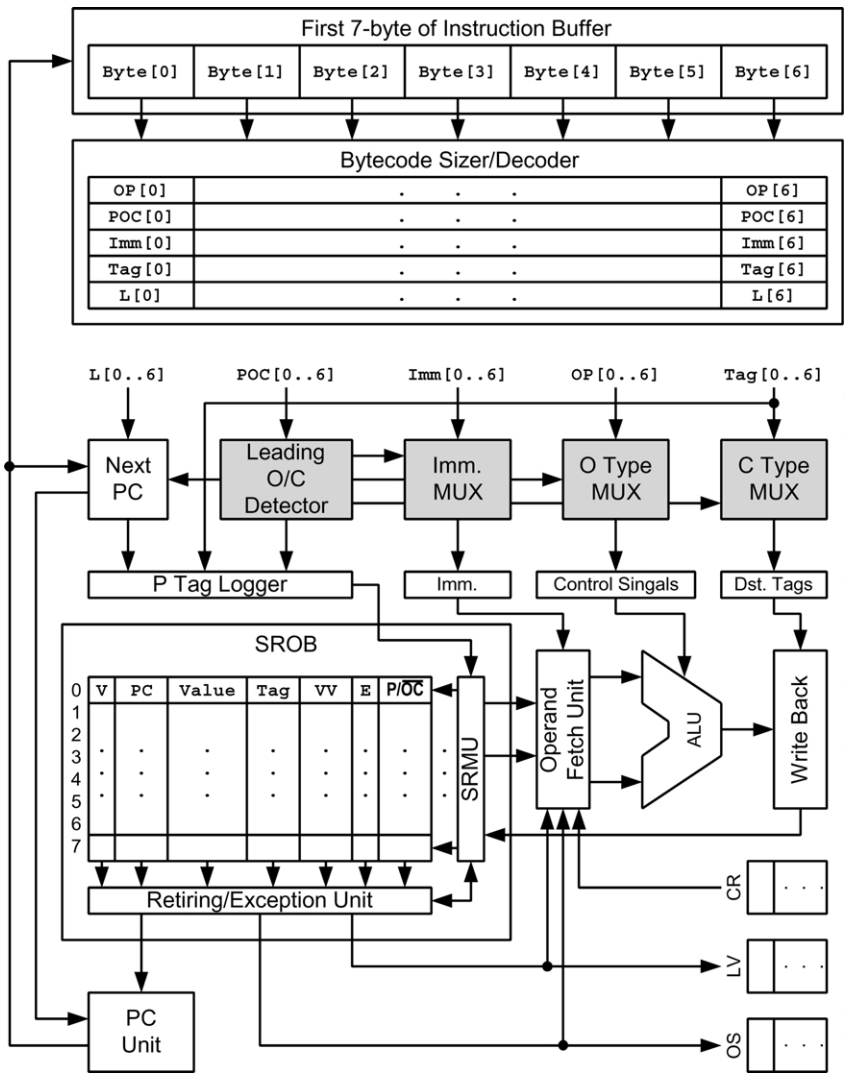

Fig. 10. Java processor architecture with the EPOC Folding.

- *Decode and Fold*. The Instruction Decode Unit decodes the bytecodes to generate information like POC types and tags. POC information is sent to EPOC Folding Unit for folding rules check.
- *Operands Fetch*. The Operands Log and Operands Fetch Unit fetches the operands from SROB first. If the condition of SROB miss occurs, it will fetch the operands from OS or LV.
- *Execution*. The Integer/Floating Point Execution Unit either executes arithmetic or calculates the effective address for memory load/store bytecodes.
- *Memory*. Data cache access for object fields and arrays.
- *Write Back and Retire*. Execution result is written back to SROB. The bottom entry of the SROB is retired in each cycle.

In the SROB, the read/write controls are implemented in the SROB Management Unit. Each bytecode is logged in-order to the SROB in the Operands Fetch stage. If an exception occurs for some bytecodes in the pipeline, the exception bit in the SROB will be set. The Retiring/Exception Handling Unit will examine this bit at the bottom entry of the SROB. The Stack Management Unit is used to maintain the data in the on-chip Operand Stack with the filling/spilling mechanism like what picoJava-II processor does [6].

4. Design of the Java processor architecture with EPOC folding

According to the simulation results, the decoder width, foldability and other parameters for the Java processor architecture with the EPOC folding design are proposed. Similar with the design in picoJava-II processor, the decoder examines first 7 bytes of the instruction buffer in our Java processor architecture [6]. As shown in Fig. 10, the first 7 bytes of the instruction buffer are sent to 7 decoders in parallel to generate the decoded information like control signals (OP), POC type information (POC), and bytecode length information (L). Since the first byte in the instruction

Table 6
An example program slice and the corresponding POC types

Instruction #	Bytecode	POC types
1	aload_1	$\boldsymbol{P}_a$
2	iload_3	$\boldsymbol{P}_b$
3	aload_1	$\boldsymbol{P}_c$
4	iload_3	P
5	iconst_1	P
6	Iadd	${}_2O_1$
7	Iaload	${}_2O_1$
8	Iastore	${}_2O_0$

Source statement: $a[j] = a[j+1]$.

buffer is designed to be a valid opcode field of Java bytecodes, the length information of the first bytecode will indicate the location of the next bytecode. If byte i is a an opcode and the next location of a valid opcode is j, each decoded information from $i+1$ to $j-1$ is cleared and the values from byte $i+1$ to $j-1$ are assigned to the immediate operands (Imm) of the byte i. For each valid opcode, the tag field (Tag) is assigned for each data generated by the opcode.

The four boxes shown in grayscale is the core folding units for the EPOC folding model. According to the POC information generated by the bytecode decoders, the Leading O/C Detector determines where the first O or C type bytecode is and generates the select inputs for the P, O, and C type multiplexers. After the number of folded bytecodes is determined, the value of next program counter address can also be generated. Simple priority encoders are enough to implement the function of Leading O/C Detector. If there are extra Ps in the EPOC folding check procedure, they are sent to the P Tag Logger and updated into the SROB in the original order. The SROB consists of seven fields as described below. The required number of bits for each field is shown in the parentheses after the field name.

- *V field (1)*. This field indicates that whether an entry in the SROB is valid or not.
- *PC field (32)*. This field is to keep the program counter for the precise interrupt consideration.
- *Tag field (5)*. This field is used as an identification number of the data in the dependency chain. For a P type bytecode, the Tag field is used to indicate the source from one of LV, Constant Register (CR), or immediate value. For an O type bytecode, the Tag field is used to indicate the destination of OS. For a C type bytecode, the Tag field is used to indicate the destination of the LV.
- *Value field (32)*. This field is the resulting value of the bytecode.
- *VV field (1)*. This field is used to indicate whether the result value in the Value field is valid or not.
- *E field (1)*. This field is used to indicate the exception status for the corresponding bytecode. If exception occurs for one bytecode, this field is set to 1.

Table 7
Dynamic bytecode counts of SPECjvm98 Benchmark Suite

Trace names	Bytecode counts (million)
compress	1137
db	74
jack	341
javac	63
jess	121
mpegaudio-3	1220
raytracer	160

- *$p\overline{oc}$ field (1)*. This field is used to indicate the POC type for the corresponding bytecode. Upon an exception condition is examined in the Retiring/Exception Handling Unit, the $p\overline{oc}$ field is used to indicate whether the Value field is required to restore to the LV.

A simple program slice is shown in Table 6 to demonstrate the operation of the EPOC folding with SROB architecture. Three Ps who provide source operands but not used immediately by O or C type bytecodes are shown in bold italic font with subscripts from a to c. The O type bytecodes with ${}_mO_n$ notation means that it needs m source operands and produces n results from/to the OS. In the original POC folding model, the $\boldsymbol{P}_a$ through $\boldsymbol{P}_c$ will be issued in-order. Bytecodes 4–6 form a FBI and the number of total execution cycles is six.

In the EPOC folding model, if we assume that the number of write ports of SROB is four, the bytecodes 1–4 will be logged in-order to the SROB in the first cycle. Bytecodes 5 and 6 will be logged into the SROB in the second cycle while the first four Ps are in the Operands Fetch stage. EPOC folding is done by passing the value in SROB to bytecode 6. In the third cycle, the $\boldsymbol{P}_c$ will be folded into bytecode 7. Finally, both $\boldsymbol{P}_a$ and $\boldsymbol{P}_b$ will be folded into bytecode 8 in the fourth cycle.

With the SROB design, both data forwarding and precise exception are done. Due to the simplicity of the SROB, the required storage cost for an n-entry SROB is $(1+32+5+32+1+1+1)\times n$ bits. Simulation data shows that 8 entries (584 bits) are enough to achieve the proposed performance.

Table 8
Occurrence percentages of the POC types

Trace names	P	O_E	O_B	O_C	O_T	O_{ALL}	C
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

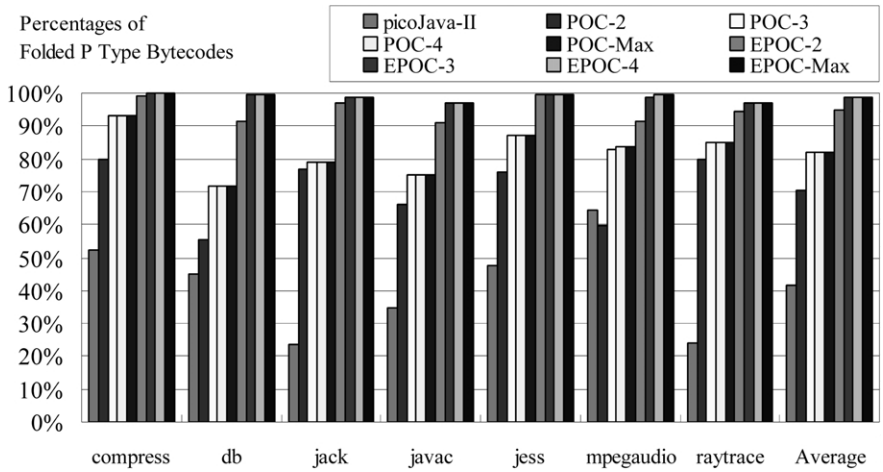

Fig. 11. Percentages of Folded P Type bytecodes.

5. Simulation results

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 three different folding models include picoJava-II, POC and EPOC folding models. We use the SPECjvm98 benchmark as our simulation source data. There are three input data set scales for the SPECjvm98 benchmarks: s1, s10, and s100 [17]. In this paper, we use s10 data set as the simulation basis. The number of bytecodes in the traces for the SPECjvm98 benchmarks is collected by the benchmark profiler as shown in Table 7. The detailed occurrence percentages for each type of bytecodes are shown in Table 8.

In Table 8, the average occurrence percentage for P and C type bytecodes are 43.38%. Theoretically speaking, if all of the P and C type bytecodes are folded, the number of bytecodes to be executed will be reduced to 56.62% of the original traces. Reducing the number of bytecodes to be executed results in the increase of average number of IIPC from 1 to the maximum of about 1.77 with the assumption of one cycle execution for each bytecode. In practice, unlike the P type bytecodes that could be folded completely, not all of the C type bytecodes could be folded. The only three foldable combinations for the C type bytecodes are $P + C$, $O_E + C$ and $O_C + C$. 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 LV. Consequently, only the C type bytecodes in the $O_E + C$ and $O_C + C$ combination are treated as 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 of an optimal folding mechanism can be derived from the following formula. According to this formula, the upper bound of speedup for the SPECjvm98 benchmark is 1.74 if all foldable P and C type bytecodes

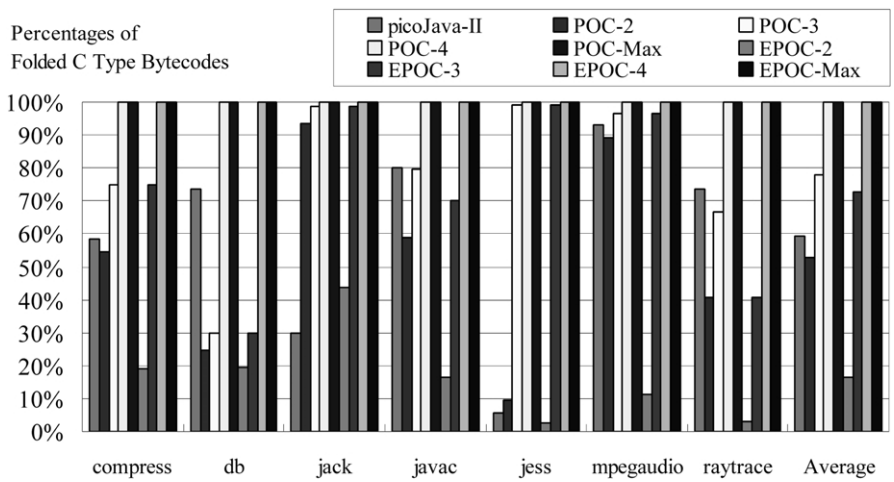

Fig. 12. Percentages of Folded C Type bytecodes.

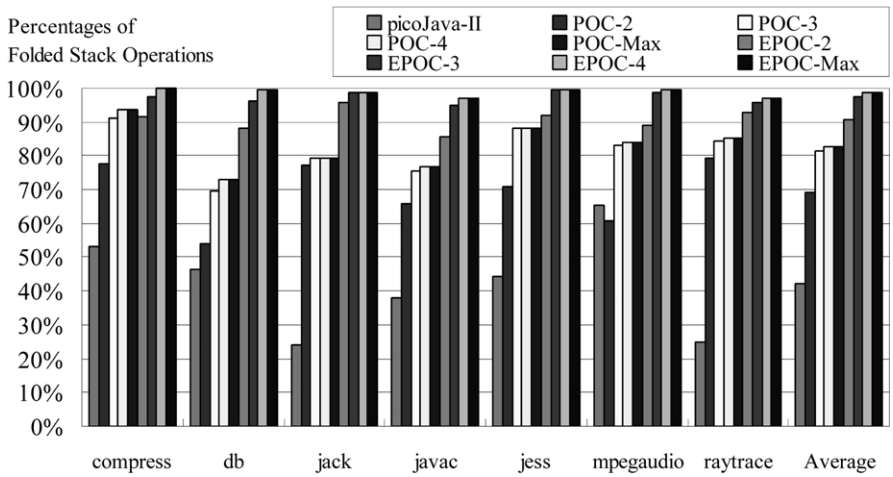

Fig. 13. Percentages of Folded P and C Type bytecodes.

are folded.

$$\text{Speedup} = \frac{\text{Cycles}_P + \text{Cycles}_O + \text{Cycles}_C}{\text{Cycles}_P(1 - \text{FP}_P) + \text{Cycles}_O + \text{Cycles}_C(1 - \text{FR}_C\text{FP}_C)}$$

where

$$\begin{cases} \text{Cycles}_x = \text{Execution Cycles} \\ \text{FP}_x = \text{Folded Percentage} \qquad \text{for } x \text{ type bytecodes} \\ \text{FR}_C = \text{Max Foldable Ratio for C} \end{cases}$$

The following simulation results are gathered using the parameter of a 7-byte instruction buffer and an 8-entry SROB. As shown in Fig. 11, 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 bytecodes that can be folded together are simulated. Results show that 3-foldable is enough for both POC and EPOC folding models if we want to fold P type bytecodes only. With the information from Table 8, the average occurrence 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, 3-foldable POC and 3-foldable EPOC folding mechanisms, respectively.

In Fig. 12, 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 4-foldable POC or EPOC folding mechanism can fold all the foldable C type bytecodes.

Fig. 13 shows the percentages of all folded P and C type bytecodes. 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, 82.90 and 98.83% for the folding mechanism of picoJava-II, POC and EPOC folding model, respectively.

The numbers of IIPC for a single-pipelined Java processor architecture are shown in Fig. 14. 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,

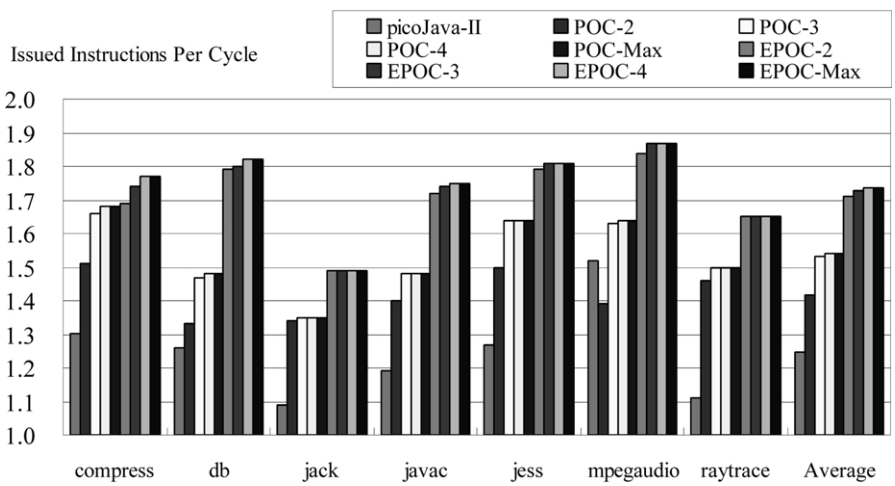

Fig. 14. Issued instructions per cycle for each Folding Model.

respectively. As mentioned at the beginning of this section, the average upper bound of IIPC is 1.74. This reveals that the EPOC folding model achieves the optimal folding speedup as compared to all other folding models.

6. Conclusion

In this paper, we have proposed the EPOC folding model based on the previously proposed POC folding model. Discontinuous-folding has shown more powerful and overrides the continuous-folding used in picoJava-II. Furthermore, the folding mechanism with patterns is shown that is 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 bytecodes. The 4-foldable strategy which folds up to four bytecodes for POC and EPOC folding model can eliminate 82.90 and 98.83% P and C type bytecodes, respectively. For all the bytecodes, the percentages of eliminated bytecodes for POC and EPOC are 36 and 43%, respectively. In other words, less than 60% of bytecodes should be executed by the Java processor with the EPOC folding model.

The hardware implementation of the EPOC folding model is shown easily to be integrated into the decoding stage using parallel priority encoders to generate the tags selection fields of the FBI within constant delay time. The SROB is proposed to meet the precise exception requirement in modern processors. With the SROB size of 584 bits, data forwarding is also done using the tag field to prevent the pipeline from stalling. 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, 1999.

[7] H.-M. Tseng, L.C. Chang, L.R. Ton, C.P. Chung, Performance enhancement by folding strategies of a Java processor, Proceedings of International Conference on Computer Systems Technology for Industrial Applications—Internet and Multimedia, 1997.

[8] L.-R. Ton, L.C. Chang, M.F. Kao, C.P. Chung, Instruction folding in Java processors, Proceedings of the International Conference on Parallel and Distributed Systems, 1997, pp. 138–143.

[9] N. Vijaykrishnan, N. Ranganathan, R. Gadekarla, Object-oriented architectural support for a Java processor, Proceedings of the ECOOP'98, Lecture Notes in Computer Science, vol. 1445, 1998, pp. 330–354.

[10] 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) 333–340.

[11] A. Kim, M. Chang, An advanced instruction folding mechanism for a stackless Java processor, Proceeding of the International Conference on Computer Design (ICCD), 2000, pp. 565–566.

[12] A. Kim, M. Chang, Advanced POC model-based Java instruction folding mechanism, Proceedings of the 26th EUROMICRO Conference, vol. 1, 2000, pp. 332–338.

[13] M.W. El-Kharashi, F. ElGuibaly, K.F. Li, A quantitative study for java microprocessor architectural requirements. Part I. Instruction set design, Microprocessors and Microsystems 24 (2000) 225–236.

[14] M.W. El-Kharashi, F. ElGuibaly, K.F. Li, A Quantitative study for Java microprocessor architectural requirements. Part II. High-level language support, Microprocessors and Microsystems 24 (2000) 237–250.

[15] L.-R. Ton, L.-C. Chang, C.-P. Chung, Exploiting Java bytecode parallelism by dynamic folding model, Proceedings of the Sixth International Euro-Par Parallel Processing Conference, Lecture Notes in Computer Science, vol. 1900, 2000, pp. 994–997.

[16] Sun Microsystems Inc., Java Development Kit 1.x, http://www.javasoft.com/products/jdk/.

[17] Standard Performance Evaluation Corporation, SPECjvm98 Benchmark, http://www.spec.org/osg/jvm98/.

Lee-Ren Ton received the BS and MS 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, and also a hardware design engineer of Hurco Automation Ltd., while working towards the PhD degree. Currently, he is a PhD 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 BS degree in Electrical Engineering from the Chung Yuan Christian College, Taiwan, in 1977, and the MS degree from the University of Southern California in 1987. He received the PhD degree of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, in April 2002. Currently, 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.

Jean Jyh-Jiun Shann received the BS degree in Electronic Engineering of Feng Chia University, Taiwan, in 1981. She attended the University of Texas at Austin from 1982 to 1985, where she received the MS degree in Electrical and Computer Engineering in 1984. She was a lecturer in Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, while working towards the PhD degree. She received the degree in 1994 and is currently an Associate Professor in the department. Her research interests include computer architecture, parallel processing, and information retrieval.

Chung-Ping Chung received the BS degree from the National Cheng Kung University, Taiwan, in 1976, and the MS and PhD 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 PhD 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 returns to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.