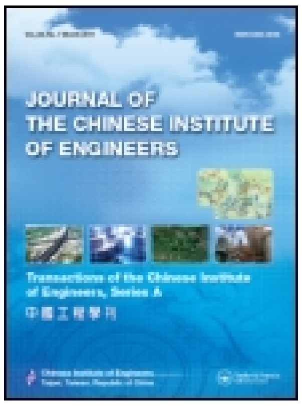

Journal of the Chinese Institute of Engineers
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tcie20

Enhancing java processor performance with smart dynamic folding

Lung-Chung Chang [a b], Lee-Ren Ton [a], Min-Fu Kao [a] & Chung-Ping Chung [a]

[a] Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 300, ROC

[b] Computer & Communications Research Laboratories, Industrial Technology Research Institute, Hsinchu, Taiwan 310, ROC

Published online: 03 Mar 2011.

To cite this article: Lung-Chung Chang, Lee-Ren Ton, Min-Fu Kao & Chung-Ping Chung (2000) Enhancing java processor performance with smart dynamic folding, Journal of the Chinese Institute of Engineers, 23:6, 711-719, DOI: 10.1080/02533839.2000.9670592

To link to this article: http://dx.doi.org/10.1080/02533839.2000.9670592

Journal of the Chinese Institute of Engineers, Vol. 23, No. 6, pp. 711-719 (2000)

ENHANCING JAVA PROCESSOR PERFORMANCE WITH SMART DYNAMIC FOLDING

Lung-Chung Chang[1,2], Lee-Ren Ton[1], Min-Fu Kao[1], and Chung-Ping Chung[1]*
[1]*Department of Computer Science and Information Engineering*
National Chiao Tung University
Hsinchu, Taiwan 300, ROC
[2]*Computer & Communications Research Laboratories*
Industrial Technology Research Institute
Hsinchu, Taiwan 310, ROC

Key Words: stack machine, stack operations folding, true data dependence, java processor.

ABSTRACT

The Java processor is suitable for Internet appliances or embedded controllers due to its speed and low memory requirement. However, its performance is severely limited by true data dependence. In this work, we present a smart and dynamic stack operations folding – POC model-based folding. The stack instructions are classified into P,O, and C three types. The folding algorithm can automatically determine the folding relations among all the instructions based on the type and folding attributes of each instruction. The proposed algorithm has no requirement to match different patterns. A typical folding mechanism design based on this model is then introduced. Also, the performance of various folding methods based on the POC model is evaluated. Simulation data indicate that the 4-foldable method eliminates 84% of all stack operations. Furthermore, the 2-, 3-, and 4-foldable methods accelerate the overall program by 1.22, 1.32 and 1.34, respectively, as compared to a Java processor without folding.

I. INTRODUCTION

The Internet has become the most feasible means of accessing information and performing electronic transactions. Java (Jame *et al.*, 1996) is the most popular language used over the Internet owing to its security, robustness and write-once-run-anywhere characteristics. Java bytecodes can be executed on any platform that provides a Java Virtual Machine (JVM) (Lindholm *et al.*, 1996) environment.

JVM is a stack-based machine (Koopman, 1997) and its performance is limited by true data dependence. A means of avoiding such a limitation, i.e. stack operations folding, was studied by Sun Microelectronics. Microprocessor Report (Case, 1996; Turley, 1996; Lentczner, 1996) and IEEE Micro (O'connor *et al.*, 1997) have also published related information. Specific reports on stack operations folding were made by Tseng *et al.* (1997) and Ton *et al.* (1997). These works all have one thing in common: they need to identify the different folding patterns via comparison with the target foldable instructions sequentially. In this study, we present a systematic folding solution. All bytecode instructions are classified into POC types with number of operands, source and destination identifiers. If an

*Correspondence addressee

instruction has the matched source type and number of operands with the destination type and number of operands of the preceding instruction, then these two contiguous instructions can be folded together. This folding process can be continued recursively.

This paper is organized as follows. Section II presents the proposed intelligent and dynamic stack operations folding model, the POC model. Instructions are scanned and checked sequentially based on the proposed folding algorithm (δ operation). Section III introduces an architecture design based on the proposed POC model. It includes a *Folding Rule Checker & Address Assigner* and *Source/Destination Address Generation Units*. Section IV summarizes the performance measurements of different folding methods. Conclusions are finally made in Section V.

II. POC MODEL IN STACK OPERATIONS FOLDING

In this section, the POC model of stack operations folding, plus a folding example are presented.

Considering the operations related to the operand stack and their characteristics, Java bytecode instructions can be classified into three types: Producer, Operator, and Consumer. Their definitions are as follows:

Definition: *Producer (P)* – An instruction that transfers data from Constant Register or Local Variable (but not Array or Constant Pool) to the operand stack.

Definition: *Operator (O)* – An instruction that retrieves data from the operand stack (may be null), and then performs the different tasks based on the following four operator subtypes:

- O_E – ALU type operator that writes the result back to the operand stack.
- O_B – Branch type operator that unconditionally or conditionally jumps to the target address according to the result of the corresponding branch decision instruction.
- O_C – Complex type operator (e.g. array accesses, constant pool accesses, and method invocations) that is implemented in micro-coded ROM. (It may or may not store the result back to the operand stack.)
- O_T – Termination type operator which is difficult to fold or impossible (e.g. iinc, goto, and athrow). Alternatively, it is a complex type operator that is implemented with trapped software emulation.

Definition: *Consumer (C)* – An instruction that consumes data from the operand stack, and stores data back into the Local Variable (but not Array or Constant Pool).

The Operator (O) is also called a *Primary Instruction* within a folding group. Producer (P) and Consumer (C) are both called Auxiliary Instruction (Ton *et al.*, 1997).

1. POC Model

The basic action of the POC model is that it always checks the foldability of one instruction (folded or not) N with its next instruction N+1. By examining their instruction types, data types, operand sources and number, operand destinations and number, the POC model determines whether they are foldable or not. If they are foldable, the resulting folding instruction then becomes the new instruction N, and it will be checked with its next instruction N+1 for further foldability. Some notations are defined below:

δ : Folding operator of instructions N and N+1.

$P_{Sn,Wn/TOS,Wn'}$: Producer with source Sn with number of operands Wn, and destination TOS with number of operands Wn'.

$O_{Sn,Wn/Dn,Wn'}$: Operator with source Sn with number of operands Wn, and destination Dn with number of operands Wn'.

$C_{TOS,Wn/LV,Wn'}$: Consumer with source TOS with number of operands Wn, and destination LV with number of operands Wn'.

Two possible relations exist between two consecutive instructions N and N+1. They are:

SI: Serial Instructions, indicating N and N+1 are serialized pipelined instructions that are not foldable.

FI: Foldable Instructions, indicating N and N+1 are foldable.

After folding check, the indicating state for further folding check can be either of:

con: Continuing state, meaning the folded instruction (N plus N+1) may be checked for further foldability.

end: Ending state, meaning the folded instruction (N plus N+1) can not be folded any further.

Figure 1 shows the foldability checking rules. The foldability check continues if the current indicating state is 'con'. The process stops if the indicating state is 'end'. Detailed state diagram and algorithm were illustrated by Chang *et al.* (1998)

	δ		Instruction N+1					
			$P_{S2,W2/TOS,W2'}$	$O_{TOS,W2/TOS,W2'}$				$C_{TOS,W2/LV,W2'}$
				$O_{E/TOS,W2/TOS,W2'}$	$O_{B/TOS,W2/-,-}$	$O_{C/TOS,W2/TOS,W2'}$	$O_{T/-,-/-,-}$	
Instruction N	$P_{S1,W1/TOS,W1'}$		$P_{S1+S2,W1+W2/TOS,W1'+W2'}$/SI/con	$O_{E/S1,W2/TOS,W2'}$/FI/con	$O_{B/S1,W2/-,-}$ FI/end	$O_{C/S1,W2/TOS,W2'}$/FI/con	$P_{S1,W1/TOS,W1'}$/SI/end	$C_{S1,W2/LV,W2'}$/FI/end
	$O_{S1,W1/D1,W1'}$	$O_{E/S1,W1/D1,W1'}$	$O_{E/S1,W1/D1,W1'}$/SI/end	$O_{E/S1,W1/D1,W1'}$/SI/end	$O_{E/S1,W1/D1,W1'}$/SI/end	$O_{E/S1,W1/D1,W1'}$/SI/end	$O_{E/S1,W1/D1,W1'}$/SI/end	$O_{E/S1,W1/LV,W2'}$/FI/con
		$O_{B/S1,W1/-,-}$	$O_{B/S1,W1/-,-}$ /end	$O_{B/S1,W1/-,-}$ /end	$O_{B/S1,W1/-,-}$ /end	$O_{B/S1,W1/-,-}$ /end	$O_{B/S1,W1/-,-}$ /end	$O_{B/S1,W1/-,-}$ /end
		$O_{C/S1,W1/D1,W1'}$	$O_{C/S1,W1/D1,W1'}$/SI/end	$O_{C/S1,W1/D1,W1'}$/SI/end	$O_{C/S1,W1/D1,W1'}$/SI/end	$O_{C/S1,W1/D1,W1'}$/SI/end	$O_{C/S1,W1/D1,W1'}$/SI/end	$O_{C/S1,W1/LV,W2'}$/FI/con
		$O_{T/-,-/-,-}$	$O_{T/-,-/-,-}$/SI/end	$O_{T/-,-/-,-}$/SI/end	$O_{T/-,-/-,-}$/SI/end	$O_{T/-,-/-,-}$/SI/end	$O_{T/-,-/-,-}$/SI/end	$O_{T/-,-/-,-}$/SI/end
	$C_{TOS,W1./LV,W1'}$		$C_{TOS,W1./LV,W1'}$/SI/end	$C_{TOS,W1./LV,W1'}$/SI/end	$C_{TOS,W1./LV,W1'}$/SI/end	$C_{TOS,W1./LV,W1'}$/SI/end	$C_{TOS,W1./LV,W1'}$/SI/end	$C_{TOS,W1./LV,W1'}$/SI/end

Note 1: Assume that instructions N and N+1 have matched data types and number of operands. Otherwise, they can not be folded, and instruction N will be assigned "SI/E" state.

Fig. 1 Foldability check for instructions N and N+1

Table 1 Annotated POC types

Instruction No.	Instruction	Annotated POC types
I1	iconst_2	$P_{iconst\_2,1/TOS,1}$
I2	iload index1	$P_{LV(index1),1/TOS,1}$
I3	iadd	$O_{E/TOS,2/TOS,1}$
I4	istore index2	$C_{TOS,1/LV(index2),1}$

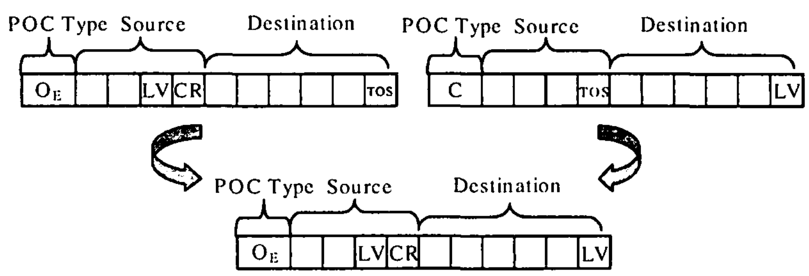

Fig. 2 Folding process of step 3

2. Example of POC Model Folding

Assume a sequence of bytecode instructions I1 ~ I4. Their POC notations are listed in Table 1.

The folding process proceeds as follows, and step 3 is depicted in Fig. 2.

Step 1: $P_{iconst\_2,1/TOS,1}$ folded with $P_{LV(index1),1/TOS,1}$ becomes $P_{iconst\_2+LV(index1),2/TOS,2}$/SI/con.

Step 2: $P_{iconst\_2+LV(index1),2/TOS,2}$ folded with $O_{E/TOS,2/TOS,1}$ becomes $O_{E/iconst\_2+LV(index1),2/TOS,1}$/FI/con.

Step 3: $O_{E/iconst\_2+LV(index1),2/TOS,1}$ folded with $C_{TOS,1/LV(index2),1}$ becomes $O_{E/iconst\_2+LV(index1),2/LV(index2),1}$/FI/end.

III. LOGIC DESIGN OF JAVA PROCESSOR FOLDING

Figure 3 shows the block diagram of the POC folding mechanism. The Bytecode instructions are fetched from *Instruction Cache* into the *Instruction Ring Buffer*. The *OP Code Checker (Sizer)* checks the instructions simultaneously to identify the locations of successive opcodes and operands. In addition to identify the folding group and its primary instruction based on the POC model, the *Folding Rule Checker & Address Assigner* assigns all of its sources and destinations for the primary instruction.

The *Source/Destination Address Generation Units (AGUs)* generate the actual addresses (on-chip or off-chip) based on the type of data unit (LV, STACK, and CR), the base address (VAR, TOS), and index (Operand). The folded instruction is then stored in the *Folded Primary Instruction Buffer*. Next, the *Execution Unit* fetches operands from on chip register or memory, and finally, executes the folded primary instruction and writes the results back to the *Operand Stack or Local Variable*. The *Program Controller* then starts the next folding cycle based on the folding result. In addition, it skips over the exact number of bytes required by the current folding operation. The *Folding Group Bytes Checker* generates this number. Detailed explanations about some of these function units are given below.

1. OP Code Checker (Sizer)

The instruction lengths of Java bytecodes vary

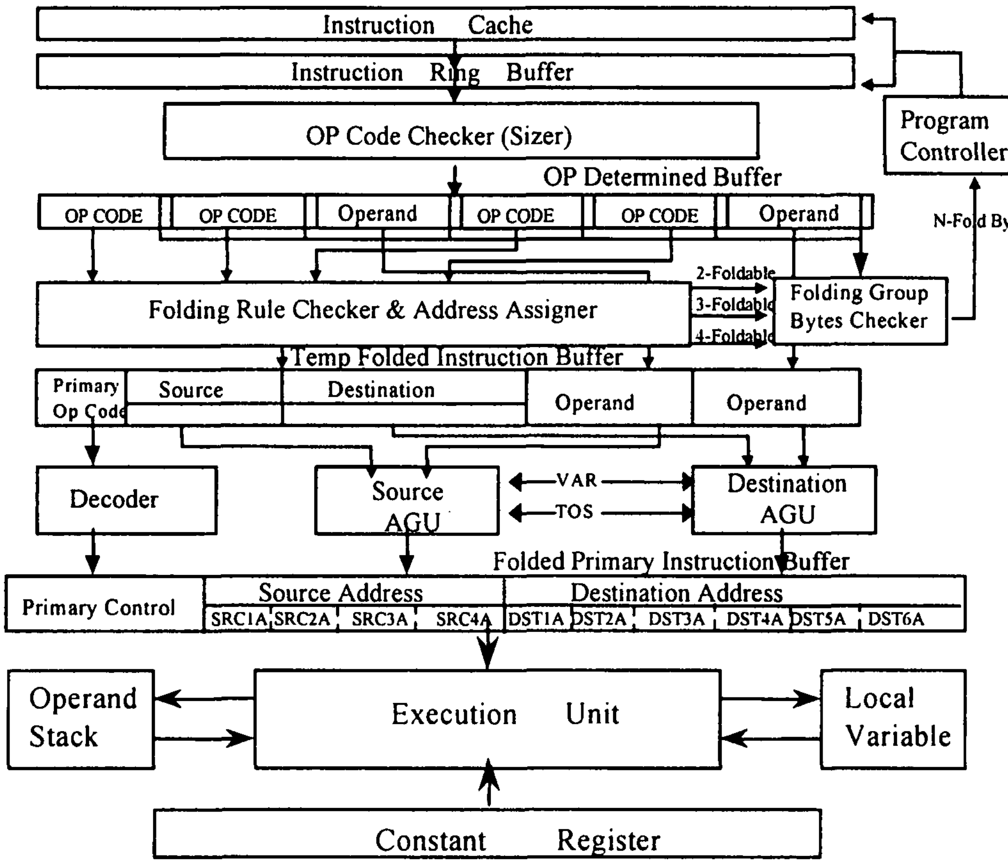

Fig. 3 Block diagram of POC folding mechanism

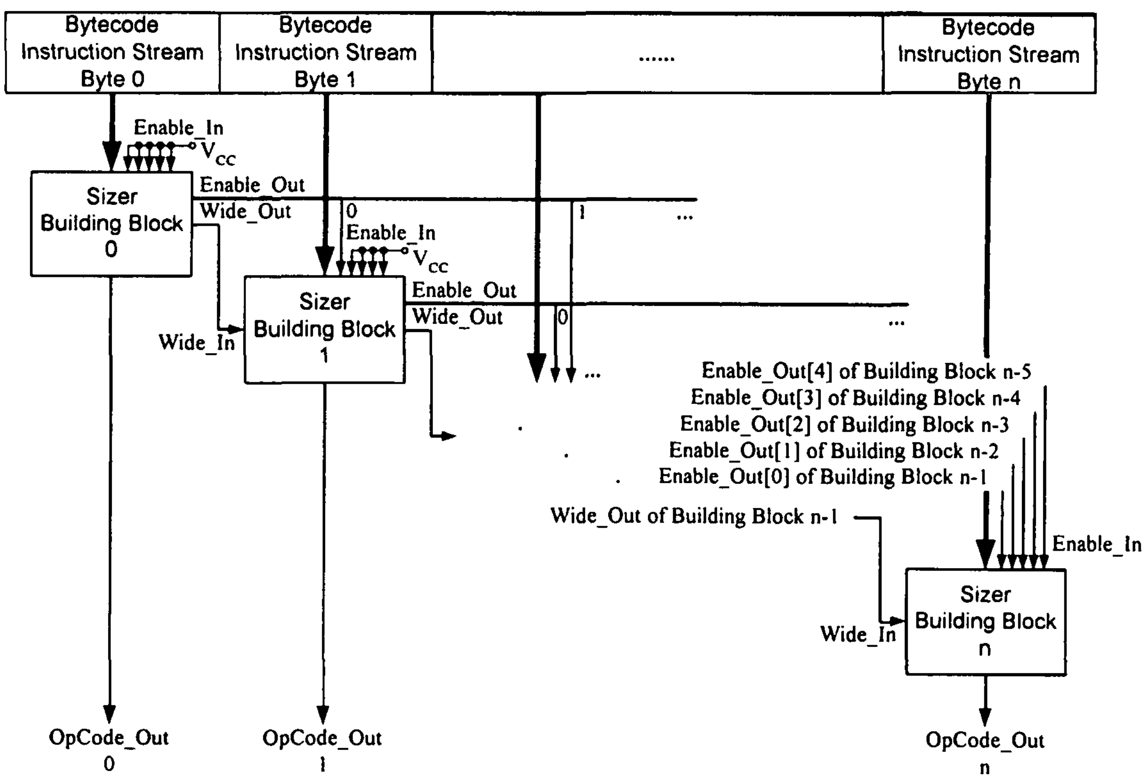

Fig. 4 Block diagram of the sizer

from one byte to five bytes, not including the lengths of *lookupswitch* and *tableswitch* instructions whose lengths are not known until run-time. Furthermore, the n-foldable design requires the simultaneous identification of at least n bytecode instructions.

As shown in Fig. 4, the Sizer is constructed with many identical building blocks, each dedicated to examining one byte in the instruction stream. Note that, the Sizer constantly treats the first byte of the instruction stream as an opcode. This opcode denotes

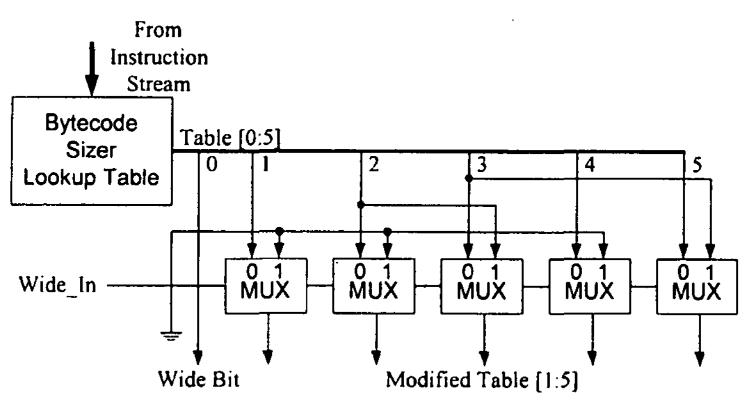

Fig. 5 Modification circuit for WIDE instruction

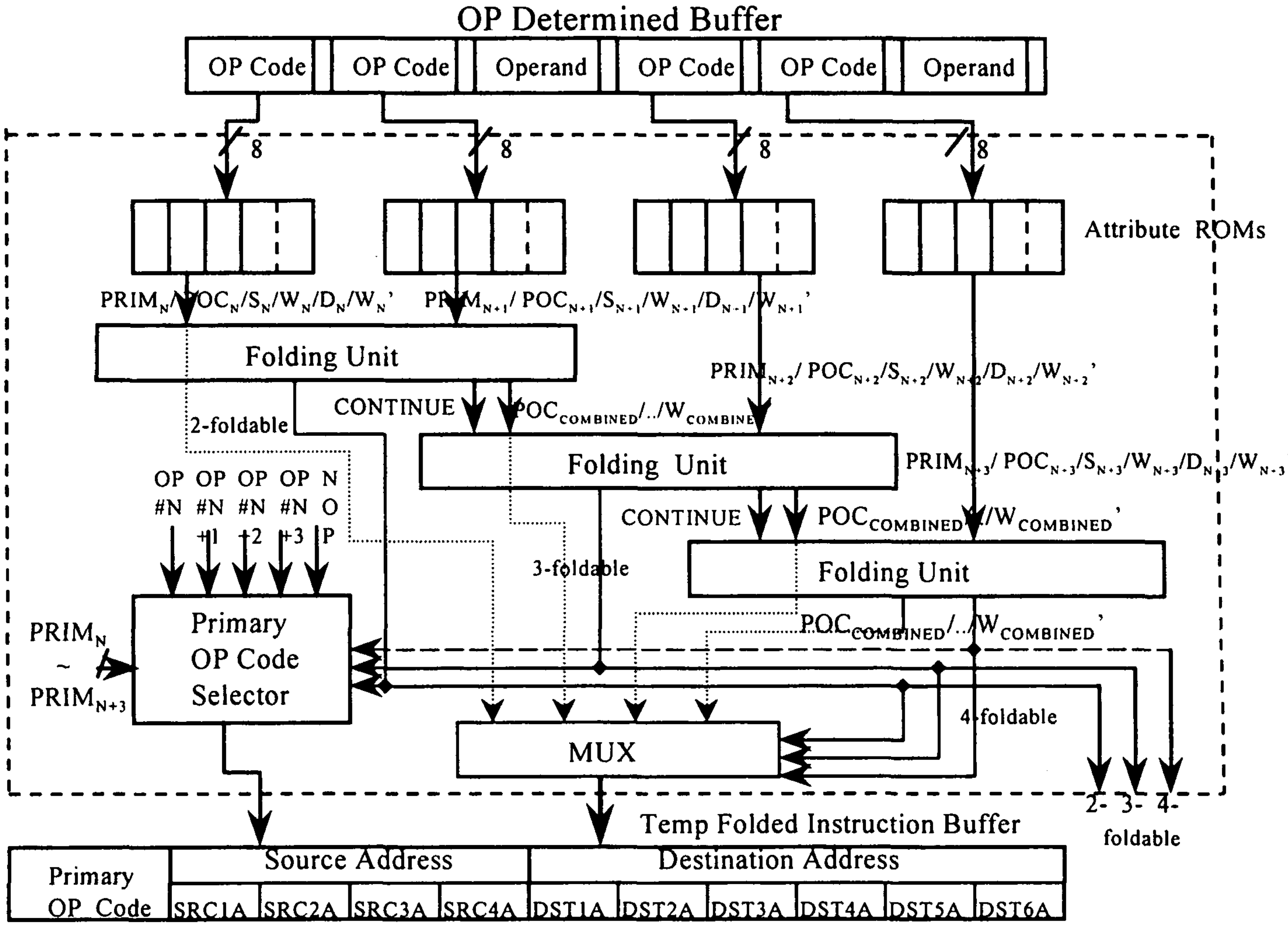

Fig. 6 Folding rule checker & address assigner

the position of the succeeding opcode.

In JVM (Lindholm *et al.*, 1996), the wide instruction is used to double the operand lengths of the next bytecode. In our design, the wide instruction is confirmed as an opcode first. It then notifies the next bytecode to modify its operand lengths information to maintain the correctness of the size check. Fig. 5 shows that only five 2-to-1 multiplexors are required to implement this modification circuit.

The size of the lookup table is 6×256 bits. Bit 0 denotes whether or not this instruction is a wide instruction. In addition, Bits 1~5 are used to denote the position of the next opcode. If a byte is an opcode, then the corresponding *OpCode_Out* signal is set to 1; otherwise, it is 0.

The function of the Sizer unit is as follows:

```
if (Any Enable_In Bit == 1'b1) {
   OpCode_Out = not (Table [0]);
   if (Wide_In == 1'b1)
      Enable_Out [0:4] = (2'b00, Table [2], 1'b0,
      Table [3]);
   else
      Enable_Out [0:4] = Table [1:5];
   Wide_Out = Table [0]; }
else {
   OpCode_Out = 1'b0;
   Enable_Out [0:4] = 5'b00000;
   Wide_Out = 1'b0; }
```

2. Folding Rule Checker & Address Assigner

Figure 6 illustrates the *Folding Rule Checker & Address Assigner* for 4-foldability. The number of opcodes to be checked are selected to generate the primary information ($PRIM_N$), POC types (POC_N), source/destination types (S_N, D_N), and number of operands (W_N, W_N') through the Attribute ROMs. The *Folding Unit* accepts two sets of this information, operates on it (based on the δ operation), and generates a folded instruction with its combined POC type ($POC_{COMBINED}$), source/destination types ($S_{COMBINED}$, $D_{COMBINED}$) and number of operands ($W_{COMBINED}$, $W_{COMBINED}$'). Note that, the next *Folding Unit* is enabled if the current indication state is 'c' (*CONTINUE* =1) and the next opcode is available for checking. The final results are stored in the *Temp Folded Instruction Buffer*.

(i) Attribute ROMs

Several *Attribute ROMs* provide the necessary information for the *folding units*. Each opcode is fed to an *Attribute ROM* and generates the following information:

$PRIM_N$: Primary information that indicates whether or not this opcode is a primary instruction.
POC_N : The POC type of this opcode.
S_N : Source type of this opcode.

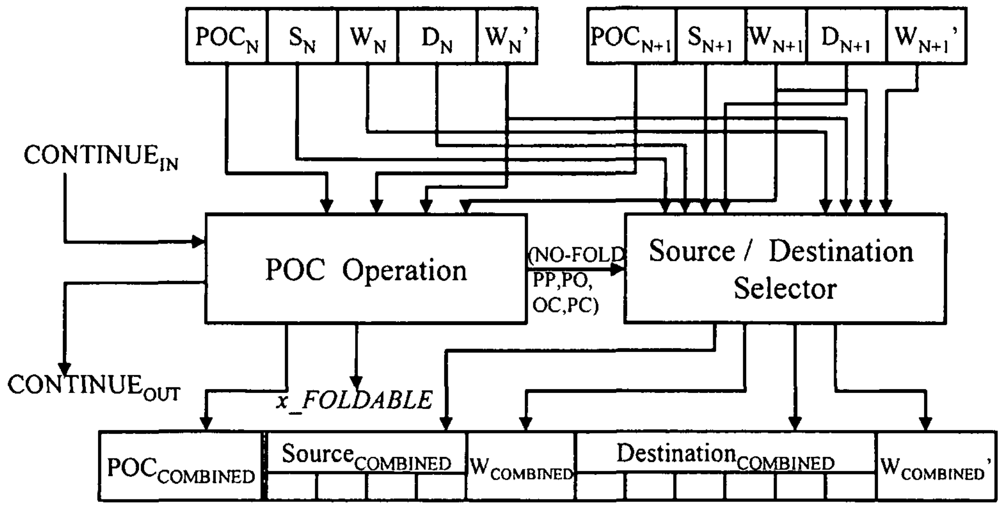

Fig. 7 Folding unit

W_N : Number of source operands of this opcode.
D_N : Destination type of this opcode.
W_N' : Number of destination operands of this opcode.

(ii) Folding Unit

The *Folding Unit* is sketched in Fig. 7. In this figure, the *POC Operation* matches the destination type (D_N) and number of operands (W_N') of instruction N with the source type (S_{N+1}) and number of operands (W_{N+1}) of instruction N+1. If the types and number of operands match, then POC_N and POC_{N+1} can be combined ($POC_{COMBINED}$). Continuation information ($CONTINUE_{OUT}$), $x\_FOLDABLE$ (x=1, 2, 3, ...) and the combination result of *POC* Operation (*NO_FOLD*, *PP*, *PO*, *OC* or *PC*) are also generated if the first instruction (POC_N) can be checked for further folding ($CONTINUE_{IN}$=1). The *Source/Destination Selector* assigns the source and destination types and their number of operands for the folded instruction.

Four bits were used to represent the instruction POC type. Table 2 lists the instruction types for Java stack operations.

The $POC_{COMBINED}$ equals POC_{N+1} if instruction N is of P type and instruction N+1 is not of O_T type. Otherwise, it equals POC_N. In addition, the *FOLDABLE* and *CONTINUE* signals can be generated using the following formula:

$$FOLDABLE=(POC_N[3]\cdot(POC_{N+1}[1]+POC_{N+1}[2]) +POC_{N+1}[0]\cdot(POC_N[3]+POC_N[2])) \cdot CONTINUE_{IN} \quad (1)$$

$$CONTINUE_{OUT}=(POC_N[3]\cdot(POC_{N+1}[3]+POC_{N+1}[2]) +POC_{N+1}[0]\cdot POC_N[2]) \cdot CONTINUE_{IN} \quad (2)$$

Table 2 Bit representation of instruction types for the POC model

Type	Symbol	P	O		C
		Bit 3	Bit 2	Bit 1	Bit 0
Producer	P	1	0	0	0
	O_E	0	1	0	0
Operator	O_B	0	0	1	0
	O_C	0	1	1	0
	O_T	0	0	0	0
Consumer	C	0	0	0	1

Based on the combination result (*NO_FOLD*, *PP*, *PO*, *OC*, and *PC*) and the source/ destination / number of operands information of instructions POC_N and POC_{N+1}, the *Source/Destination Selector* generates the combined source/destination information for the folded instruction ($POC_{COMBINED}$). The combined information including source, destination and number of operands information, are summarized below:

NO_FOLD:

$S_{COMBINED}[1]\sim S_{COMBINED}[W_N]=S_N[1]\sim S_N[W_N]$
$D_{COMBINED}[1]\sim D_{COMBINED}[W_N']=D_N[1]\sim D_N[W_N']$
$W_{COMBINED}=W_N$; $W_{COMBINED}'=W_N'$

PP:

$S_{COMBINED}[1]\sim S_{COMBINED}[W_N]=S_N[1]\sim S_N[W_N]$
$S_{COMBINED}[W_{N+1}]\sim S_{COMBINED}[W_N+W_{N+1}]=S_{N+1}[1]\sim S_{N+1}[W_{N+1}]$
$D_{COMBINED}[1]\sim D_{COMBINED}[W_N'+W_{N+1}'']=STACK[TOS+1]\sim STACK[TOS+W_N'+W_{N+1}']$
$W_{COMBINED}=W_N+W_{N+1}$; $W_{COMBINED}'=W_N'+W_{N+1}'$

PO:

$S_{COMBINED}[1]\sim S_{COMBINED}[W_N]=S_N[1]\sim S_N[W_N]$
$D_{COMBINED}[1]\sim D_{COMBINED}[W_{N+1}']=D_{N+1}[1]\sim D_{N+1}[W_{N+1}']$
$W_{COMBINED}=W_N$; $W_{COMBINED}'=W_{N+1}'$

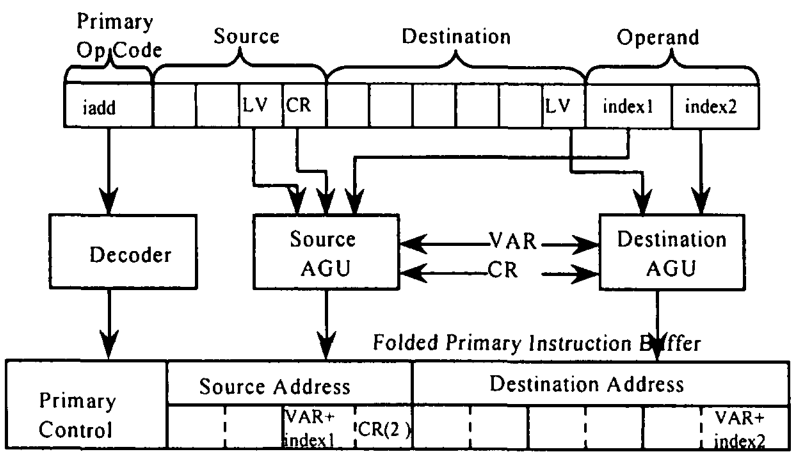

Fig. 8 Source/Destination AGU contents illustrated

OC:

$$S_{COMBINED}[1] \sim S_{COMBINED}[W_N] = S_N[1] \sim S_N[W_N]$$
$$D_{COMBINED}[1] \sim D_{COMBINED}[W_N'] = D_{N+1}[1] \sim D_{N+1}[W_{N+1}']$$
$$W_{COMBINED} = W_N;\ W_{COMBINED}' = W_{N+1}'$$

PC:

$$S_{COMBINED}[1] \sim S_{COMBINED}[W_N] = S_N[1] \sim S_N[W_N]$$
$$D_{COMBINED}[1] \sim D_{COMBINED}[W_{N+1}'] = D_{N+1}[1] \sim D_{N+1}[W_{N+1}']$$
$$W_{COMBINED} = W_N;\ W_{COMBINED}' = W_{N+1}'$$

(iii) Primary OP Code Selector

The opcode of the primary instruction after folding is selected according to the following rules:

Case 1: The first opcode, if there is no folding.
Case 2: NOP, if the folding combination type is PC.
Case 3: The M-th opcode OP#M if the $PRIM_M$=1 exclusively.

3. Source/Destination Address Generation Units (AGUs)

The *Source/Destination AGUs* generate the actual source or destination addresses based on the address type (LV, STACK, CR), initial address (VAR, TOS) and index (Operand). Some addresses are renamed or mapped onto the stack cache registers based on the hardware implementation, the others are physical memory addresses. The generated information is stored in the *Folded Primary Instruction Buffer* and is used by the *Execution Unit*. Fig. 8 illustrates the *Source/Destination AGUs* functions for the example described in Subsection 2 of Section II.

IV. PERFORMANCE OF POC MODEL

The maximum number of bytecode instructions that can be folded is timing critical and is predefined according to the implementation specification. Fig. 9 shows the percentage of eliminated stack operations

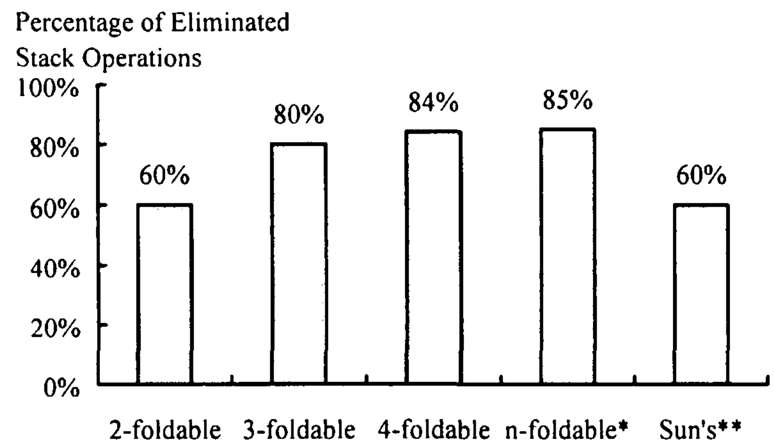

Fig. 9 Percentage of eliminated stack operations with respect to all stack operations

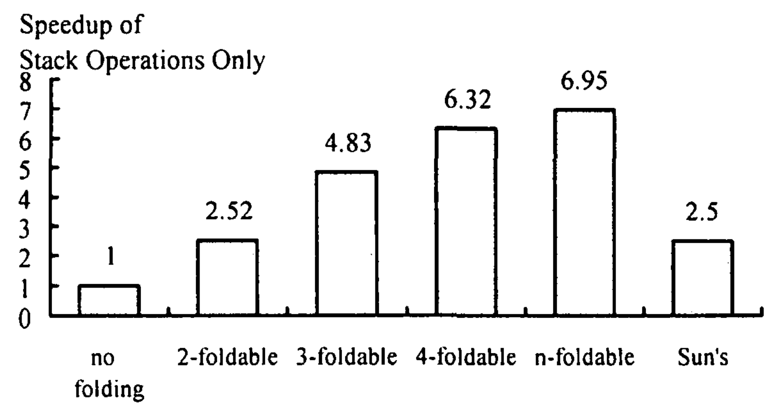

Fig. 10 Speedups of stack operations only due to different scopes of folding

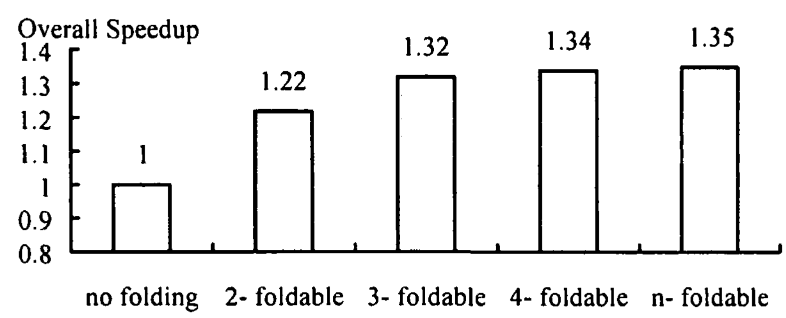

Fig. 11 Overall speedups due to different scopes of folding

for different scopes of folding.

Figures 10 and 11 depict the speedups of different scopes of folding in terms of machine cycles for stack operations only, or all operations.

V. CONCLUSIONS

This study presents the theorem and operations of a stack operations folding POC model. The proposed model can automatically generate folding patterns by classifying the Java bytecodes into POC types. Furthermore, this method is generic for any stack machine.

Various scopes of folding are evaluated. Simulation results indicate that 2-, 3-, 4-, and n-foldable methods can eliminate 31%, 41%, 43%, and 44% of all operations in the Java program trace files, respectively. Since 52.05% of all operations are stack oriented, the 2- to 4-foldable methods can eliminate 60%, 80%, and 84% of all stack operations, respectively. By translating the instruction counts into clock cycles (Ton *et al.*, 1997), the corresponding speedups are 1.22, 1.32 and 1.34, respectively, as compared to a conventional Java stack machine without stack operations folding support.

With proper VLSI implementation, it is not difficult to construct a 200 MHz Java processor using the POC model for folding check, without using much silicon area. For the 4-foldable design, Verilog-XL™ simulation indicates that the POC Operation takes 3.62 ns using 0.6 um SPDM standard cells library. If higher performance is desired for future Java processors, The POC model should be highly promising for performance enhancement via folding check.

ACKNOWLEDGEMENTS

This paper presents partial results of research projects financed by Computer & Communications Research Laboratories of Industrial Technology Research Institute, ROC, under contract no. G3-86040 and I87001.

NOMENCLATURE

AGU	Address Generation Unit
C	Consumer
con	continuing state
CR	Constant Register
D_N	Destination type of instruction *N*
end	ending state
FI	Foldable Instruction
LV	Local Variable
NO_FOLD	No folding combination
O	Operator
O_B	Operator with Branch subtype
OC	Folding combination of Operator with Consumer
O_C	Operator with Complex subtype
O_E	Operator with Execution subtype
O_T	Operator that Terminates the folding check
P	Producer
PC	Folding combination of Producer with Consumer
PO	Folding combination of Producer with Operator
POC_N	*POC* type of instruction *N*
PP	Folding combination of Producer with Producer
$PRIM_N$	Primary information of instruction *N*
SI	Serial Instruction
S_N	Source type of instruction *N*
TOS	Top of Stack
W_N	Number of source operands of instruction *N*
W_N'	Number of destination operands of instruction *N*

Greek symbol

δ	Folding operator of instructions *N* and *N*+1

REFERENCES

1. Case, B., March 1996, "Implementing the Java Virtual Machine," *Microprocessor Report.*
2. Chang, L.C, Ton, L.R., Kao, M.F., and Chung, C.P., September 1998, "Stack operations folding in Java processors" *IEE Proceedings on Computer and Digital Techniques*, Vol. 145, No. 5.
3. Gosling, J., Joy, B., and Steele, G., August 1996, The Java™ Language Specification, *Addison-Wesley Publisher Co., Inc.*
4. Koopman, P., 1997, "Stack Computers: The New Wave" http:// www.cs.cmu.edu/~koopman/stack_computers/.
5. Lentczner, M., March 1996, "Java's Virtual World" *Microprocessor Report.*
6. Lindholm, T., and Yellin, F., September 1996, "The Java™ Virtual Machine Specification," *Addison-Wesley Publishing Co., Inc.*
7. O'Connor, J.M., and Tremblay, M., March/April 1997, "picoJava-I: The Java Virtual Machine in Hardware," *IEEE Micro*, Vol. 17, No. 2.
8. Ton, L.R., Chang, L.C., Kao, M.F., Tseng, H.M., Shang, S.S., Ma, R.L., Wang, D.C., and Chung, C.P., December 1997, "Instruction Folding in Java Processor" *International Conference on Parallel and Distributed Systems.*
9. Tseng, H.M., Chang, L.C., Ton, L.R., Kao, M.F., Shang, S.S., and Chung, C.P., April 1997, "Performance Enhancement by Folding Strategies of a Java Processor" *International Conference on Computer Systems Technology for Industrial Applications – Internet and Multimedia.*
10. Turley, J., October 1996, "Sun Reveals First Java Processor Core" *Microprocessor Report.*

Discussions of this paper may appear in the discussion section of a future issue. All discussions should be submitted to the Editor-in-Chief.

Manuscript Received: Jul. 30, 1999
Revision Received: Mar. 20, 2000
and Accepted: Apr. 12, 2000

利用智慧型動態之指令摺疊方法來提昇爪哇處理機之效能

張隆昌[1,2] 唐立人[1] 高敏富[1] 鍾崇斌[1]

[1] 國立交通大學資訊工程研究所

[2] 工業技術研究院電腦與通訊工業研究所

摘 要

爪哇處理機由於其速度快所需記憶體少，非常適合網路資訊機及嵌入式控制器之應用。但是爪哇處理機之效能還是受到資料相依性之嚴重限制。在本篇研究裡，我們介紹了一個具智慧又可作動態檢察之堆疊運算摺疊方法--POC摺疊法。在此摺疊法下，堆疊指令基本上可分爲 POC 三種類別。依此類別及指令相關屬性即可檢察出指令間可否摺疊，而不需去比較不同之所有允許的固定指令摺疊樣式。除了摺疊方法外，本文亦介紹其摺疊機構的設計。這個設計的模擬結果指出，4- 摺疊可除去 84% 之所有堆疊運算，而 2- ， 3- ， 4- 摺疊與無摺疊之效能比分別爲 1.22 ， 1.32 與 1.34 。

關鍵詞：爪哇處理機，堆疊機器，堆疊運算摺疊，資料相依性。