Stack operations folding in Java processors

L.-C. Chang
L.-R. Ton
M.-F. Kao
C.-P. Chung

Indexing terms: Java processors, Stack machines

Abstract: Traditionally, the performance of a stack machine has been limited by the true data dependency. A performance enhancement mechanism, stack operations folding, was used in Sun Microelectronics' picoJava-I design, and it can fold up to 60% of all stack operations. The authors use the Java bytecode language as the target machine language, and study Java instruction folding on a proposed folding model, the POC model, which is used to illustrate the theoretical folding operations. Various practical folding strategies based on the POC model are introduced and evaluated. Statistical data show that the 4-foldable strategy eliminates 84% of all stack operations, and the 2-, 3-, and 4-foldable strategies result in overall program speedups of 1.22, 1.32 and 1.34, respectively, as compared to a stack machine without folding. Furthermore, the 4-foldable strategy is the most practical and cost effective of a Java stack machine design with a decoder width of 8 bytes. Circuit simulation results show that a 100MHz 4-foldable folding mechanism can be realized with 0.6μm CMOS standard cells, or 240MHz with 0.25μm CMOS technology.

1 Introduction

The Internet has been widely used and network computers [1] are being promoted to be the key component in this application paradigm due to their simplicity, reduced management effort, and low cost.

A Java stack machine has the advantage of small code size, 1.8 bytes per instruction on average [2] as compared to other CISC or RISC machines. No source or destination register identifiers need to be assigned for the instructions, making the instruction size small [3]. However, all of the succeeding ALU or other stack-related operations must be dependent on the previous load or written back data. This inherent true dependence severely limits the instruction level parallelism. Sun Microelectronics proposed the folding technique [2, 4, 5] as a method to avoid the unnecessary loads or writes back to the stack. Before that, studies into stack machine [3] folding were lacking, and the design results revealed by Sun Microelectronics were not clearly elaborated. Stack operations folding (particularly for Java bytecode [6]) still requires extensive study and the purpose of this paper is to present both a theoretical study and practical implementation issues.

In this research, we use trace driven simulation in our performance study. Although Java is a popular language [7], it is still too immature to have typical benchmarks like the SPEC benchmarks. So we gathered many Java programs to use as our benchmarks, and we hope that these benchmarks will serve as a representative sampling of typical Java programs.

Most of our Java benchmarks are applets obtained from Sun Microelectronics' JDK (Java Development Kit) [7] samples. These Java benchmarks can be run in browsers such as Netscape Navigator or JDK appletviewer. Only one benchmark, the Java compiler (javac), is an application which can be run in the command line. We categorise these benchmarks into three types: the first is *animation*, which makes web pages look more attractive; the second is *interaction*, such as web games; and the last is *performance benchmark*, which tests the performance of a computer, such as `CaffeineMark` and `Jstones`. The summary of these benchmarks is shown in Table 1.

To analyse the performance gain associated with the eliminated stack operations or execution cycles, we need to calculate the theoretical performance upper bound that stack operations folding can achieve. Then, a practical folding strategy is suggested, based on the simulation results of how closely the performance of each strategy can approximate the upper bound. The theoretical performance upper bound is calculated by first finding the theoretical foldable instruction groups, then eliminating all foldable stack operations and counting the resulting execution cycles. Finally, the speedup upper bound is calculated accordingly. The following equation calculates the speedup upper bound for stack operations only:

$$\text{speedup upper bound}_{\text{StackOpsOnly}} = \frac{ExecutionTime_{AllStackOps}}{ExecutionTime_{AfterPerfectFolding}}$$

where $ExecutionTime_{AllStackOps}$ is the execution cycle counts of all stack operations, and $ExecutionTime_{AfterPerfectFolding}$ is the execution cycle counts after perfect folding of all stack operations.

© IEE, 1998

IEE Proceedings online no. 19982200
Paper first received 17th November 1997 and in revised form 2nd April 1998

The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 30050, Republic of China

Table 1: Java benchmarks summary

Types	Benchmarks	Instruction counts (millions)
Animation	Animator	4.5
	BarChart	0.5
	Blink	3.0
	Clock	4.2
	Fractal	2.5
	Led	0.5
	NervousText	1.7
	ROC_Flag	4.9
	SimpleGraph	0.4
	SortDemo	7.1
Interaction	ArcTest	1.0
	Ataxxlet	80.0
	CardTest	0.8
	DitherTest	33.8
	DrawTest	20.3
	Dugout	31.1
	GraphicsTest	1.2
	GraphLayout	16.1
	Lceblox	17.0
	ImageMap	6.6
	JumpingBox	3.2
	MoleculeViewer	1.2
	SpreadSheet	3.0
	TicTacToe	1.6
	WireFrame	2.0
CPU performance benchmark	Javac TicTacToe.java	20.3
	Caffeinemark	30.9
	Jstone	5.2
	LinpackJava	7.8
	Plasma	3.5

The following equation calculates the overall speedup:

$$\text{speedup upper bound}_{\text{Overall}} = \frac{ExecutionTime_{AllOps}}{ExecutionTime_{AllAfterPerfectFolding}}$$

where $ExecutionTime_{AllOps}$ is the execution cycle counts of all operations, and $ExecutionTime_{AllAfterPerfectFolding}$ is the execution cycle counts of all operations after perfect stack operations folding.

2 Stack operations folding

In this Section, some terminology is defined, and the basic folding operations are introduced.

2.1 Definitions

Before we present the details of the POC model of stack operations folding we will introduce some folding related definitions:

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.

Stack operations folding group: A collection of contiguous stack instructions that can be folded together.

Primary instruction: The instruction in a folding group that consumes and produces data (i.e. ALU instructions), transfers control (i.e. branch instructions) or invokes a microprogram. If none of the above exists in a folding group, a null primary instruction (NOP) will be assigned.

Auxiliary instruction: An instruction in a folding group that is not a primary instruction (i.e. instruction that provides the source address or destination address to the primary instruction).

Considering the operations related to the operand stack and their characteristics, the Java bytecode instructions can be classified into three types: producer, operator and consumer. Their property and percentage of occurrences in the benchmarks are listed in Table 2, and their definitions are as follows:

Table 2: Instruction types for Java stack operations folding

Type	Symbol	Description	Percentage (%)
Producer	P	push constant/load from LV	47.14
Operator	O_E	execution unit instructions	10.87
	O_B	branch type instructions	11.54
	O_C	complex type instructions	22.19
	O_T	termination type instructions	3.97
Consumer	C	store into LV	4.29

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

Operator (O): An instruction that gets data from the operand stack (may be dummy) and then performs the different tasks based on the following three operator subtypes:

O_E – ALU type operator that writes the result back to the operand stack.

O_B – Branch type operator that may jump to the target address based on the branch decision.

O_C – Complex type operator (including array access, constant pool access, invokevirtual, ...) which is implemented by micro-coded ROM. (It may or may not write back the result to the operand stack.)

O_T – Termination type operator which is unable or finds it hard to join the folding operation (like iinc, goto, athrow, ...), or it is a complex type operator but with software emulation.

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). Within a folding group, the operator O is treated as the primary instruction. Both the producer P and consumer C are treated as auxiliary instructions.

2.2 Stack operations folding procedure

Most operations of a stack machine must push or pop data to or from the top of its stack (TOS). This will

cause a serious data hazard due to true data dependence. 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 type instructions, branch type instructions or complex type instructions) operates on the accessed 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 the stack instructions are of true data dependency to form a folding group, then we can fold them together by redirecting the data provided by the producer to the corresponding primary 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 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 execution steps is reduced from five to three. Hence, the system performance can be increased greatly by folding.

3 POC model of stack operations folding

In this Section, the POC model of stack operations folding, the state diagram of folding rules checking, and the folding algorithm are presented.

3.1 POC model

To give a clear overview of stack operations folding, a generic POC model is constructed. The basic concept of the POC model is that it checks the instructions N and $N + 1$ to see whether they can be folded together (based on the instruction type, operand source, operand destination, data type and width). If they are foldable, the folded result instruction will become the new instruction N, and will be checked with the new following instruction $N + 1$, repetitively, until the end of folding.

The definitions of P, O, and C have been presented in Section 2.1. The other notations are as follows:

δ: Folding operator of instruction N and $N + 1$.

$P_{Sn,Wn/TOS,Wn'}$: Producer with source Sn, data width Wn and destination TOS, data width Wn'.

$O_{Sn,Wn/Dn,Wn'}$: Operator with source Sn, data width Wn, and destination Dn, data width Wn'.

$C_{TOS,Wn/LV,Wn'}$: Consumer with source TOS, data width Wn, and destination LV, data width Wn'.

One of two possible relations exists between two consecutive stack instructions. These two possible relations are:

SI: Instructions N and $N + 1$ are general pipelined serial instructions that are not foldable.

FI: Instructions N and $N + 1$ are foldable stack instructions.

The possible next state after the folding operation may be either of the following:

C: The result of folding instructions N and $N + 1$ may be checked for further foldability with the next instruction.

E: The result of folding instructions N and $N + 1$ cannot be folded any further, and the folding group checking can be terminated.

An example of folding using these notations is given below:

$O_{E/S1,W2/D2,W2'}/FI/C$: The folding result of instructions N and $N + 1$ is O_E type with source $S1$, data width $W2$ and destination $D2$, data width $W2'$. These two instructions are foldable, and they can be checked for further folding with the next instruction in the program.

Fig. 2 shows the foldability check for contiguous instructions N and $N + 1$. The foldability check will continue if the current checked result is in state 'C'. Otherwise, the process stops if the resulting state is 'E'. For example, if the sequence of bytecode instructions is $I1$~$I4$, then their type notations are as follows:

$$\left.\begin{array}{l} I1\text{:const_2} \\ I2\text{:iload index1} \\ I3\text{:iadd} \\ I4\text{:istore index2} \end{array}\right\} \begin{array}{c} I1\sim I4 \text{ will be} \\ \text{annotated into} \end{array} \left\{\begin{array}{l} I1{:}P_{iconst_2,1/TOS,1} \\ I2{:}P_{LV(index1),1/TOS,1} \\ I3{:}O_{E/TOS,2/TOS,1} \\ I4{:}C_{TOS,1/LV(index2),1} \end{array}\right.$$

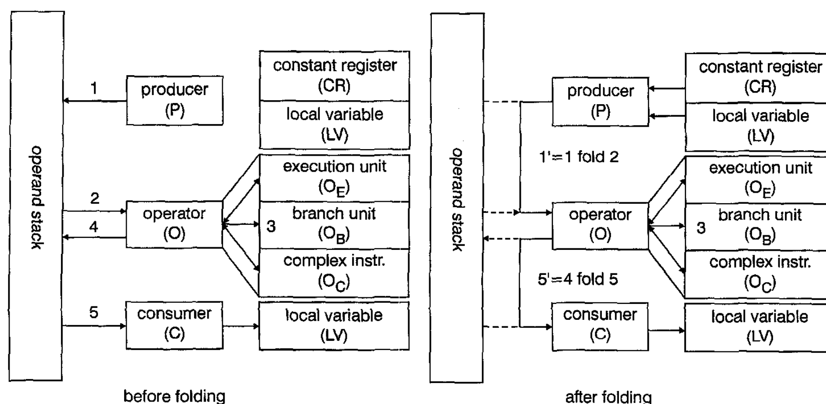


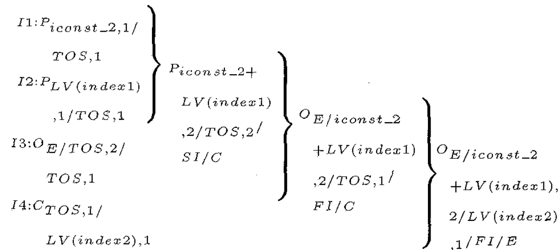
Fig. 1 *Stack operations folding*

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/C	$O_{E/S1,W2/TOS,W2'}$/FI/C	$O_{B/S1,W2/-,-}$/FI/E	$O_{C/S1,W2/TOS,W2'}$/FI/C	$P_{S1,W1/TOS,W1'}$/SI/E	$C_{S1,W2/LV,W2'}$/FI/E
$O_{S1,W1/D1,W1'}$	$O_{E/S1,W1/D1,W1'}$	$O_{E/S1,W1/D1,W1'}$/SI/E	$O_{E/S1,W1/D1,W1'}$/SI/E	$O_{E/S1,W1/D1,W1'}$/SI/E	$O_{E/S1,W1/D1,W1'}$/SI/E	$O_{E/S1,W1/D1,W1'}$/SI/E	$O_{E/S1,W1/LV,W2'}$/FI/C
	$O_{B/S1,W1/-,-}$	$O_{B/S1,W1/-,-}$/SI/E	$O_{B/S1,W1/-,-}$/SI/E	$O_{B/S1,W1/-,-}$/SI/E	$O_{B/S1,W1/-,-}$/SI/E	$O_{B/S1,W1/-,-}$/SI/E	$O_{B/S1,W1/-,-}$/SI/E
	$O_{C/S1,W1/D1,W1'}$	$O_{C/S1,W1/D1,W1'}$/SI/E	$O_{C/S1,W1/D1,W1'}$/SI/E	$O_{C/S1,W1/D1,W1'}$/SI/E	$O_{C/S1,W1/D1,W1'}$/SI/E	$O_{C/S1,W1/D1,W1'}$/SI/E	$O_{C/S1,W1/LV,W2'}$/FI/C
	$O_{T/-,-/-,-}$	$O_{T/-,-/-,-}$/SI/E	$O_{T/-,-/-,-}$/SI/E	$O_{T/-,-/-,-}$/SI/E	$O_{T/-,-/-,-}$/SI/E	$O_{T/-,-/-,-}$/SI/E	$O_{T/-,-/-,-}$/SI/E
	$C_{TOS,W1/LV,W1'}$	$C_{TOS,W1/LV,W1'}$/SI/E	$C_{TOS,W1/LV,W1'}$/SI/E	$C_{TOS,W1/LV,W1'}$/SI/E	$C_{TOS,W1/LV,W1'}$/SI/E	$C_{TOS,W1/LV,W1'}$/SI/E	$C_{TOS,W1/LV,W1'}$/SI/E

Fig. 2 *Foldability check for contiguous instructions N and N + 1*

Note 1: Assume that the two contiguous instructions have the matched data type. Otherwise they are in SI/E state.
Note 2: Assume that the contiguous instructions have the matched data width.
Note 3: Assume that the machine is a stsack machine, and the true data dependency is required in stack operations folding as defined in Section 2.1. P-P type combination is treated as serial instructions because of the lack of true data dependency.

The folding process proceeds as follows:



In Step 1, the two providers $P_{iconst_2,1/TOS,1}$ and $P_{LV(index1),1/TOS,1}$ are combined to become a single, larger provider $P_{iconst_2+LV(index1),2\ /TOS,2}$. The new provider is a general pipelined serial relation (*SI*) but with a continually foldable state *C*. In Step 2, $_{iconst_2+LV(index1),2\ /TOS,2}$ is folded with $O_{E/TOS,2/TOS,1}$ to form a new instruction $O_{E/iconst_2+LV(index1),2/TOS,1}$ with foldable relation (*FI*) and continually foldable state *C*. In that step, the sources of iadd have been changed to iconst_2 and LV (index1). In the final step, the folding operation combines $O_{E/iconst_2+LV(index1),2/TOS,1}$ and $C_{TOS,1/LV(index2),1}$ into $O_{E/iconst_2+LV(index1),2/LV(index2),1}$ which is foldable (*FI*) but should not be further checked for foldability (as indicated by state *E*). As a result, the four instructions are combined into a single instruction iadd with the two source operands iconst_2 and LV (index1), and the destination LV (index2).

3.2 State diagram and algorithm

The state diagram of stack operations folding as presented with the POC model is shown in Fig. 3. In this state diagram, the new notations, State_P, State_O_B, State_O_E, State_O_C, State_C, are used for the different intermediate states during folding. This model allows as many contiguous providers (*P*) to provide sources to the operator as possible, and then enter State_O_E, State_O_B, State_O_C, or State_C upon encountering an O_E, O_B, O_C or C, respectively. In the State_O_E and State_O_C states, if the upcoming input is a consumer (*C*), then the state will not change, but only the folded instruction will have a different destination address. That is because the real operation is performed by the primary instruction (*O*), and all other auxiliary instructions (*P* or *C*) will redirect the data to this primary instruction. All other inputs for State_O_B, State_O_E, State_O_C and State_C will stop the folding rule check, because the stack operations folding group has already ended, and cannot be folded with other instructions anymore. The input O_T will terminate the folding operation in any state of Fig. 3.

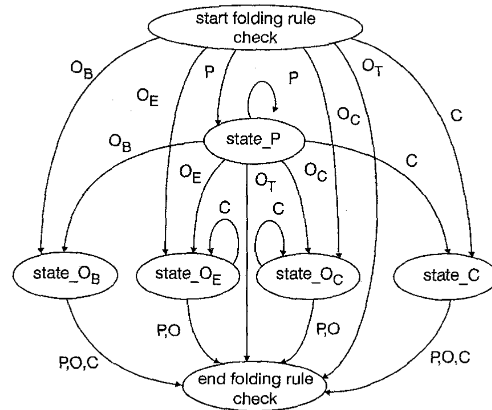


Fig. 3 *N-foldable folding rule check*

In Fig. 3, if the number of examined bytecode instructions in State_O_B, State_O_E, State_O_C, and State_C is greater than or equal to two, then those instructions are foldable and they form a folding group. The primary instruction in a folding group must be one of O_B, O_E, O_C or NULL. Its input is provided by *P*(*s*), and its result is consumed by *C*(*s*).

The algorithm to determine how many bytecode instructions can be folded together is listed below. The complexity of this algorithm is O(N).

Algorithm Folding_Check (I, N)

Input: *I* (an instruction array in the range *M* >= *N*),

N (an integer representing N-foldable strategy).

Output: *foldable#* (foldable stack instructions number under N-foldable strategy).

begin

foldable# := 1; {*default is one instruction to be issued*}

I_{first} := *I*[1]; {*initialise the first instruction for folding check*}

I_{second} := *I*[2]; {*initialise the second instruction for folding check*}

k := 2;

while k <= N do

$I_{result}/x/y := I_{first}\ \delta\ I_{second}$; {*folding operation*}

if x = 'FI' then foldable# := *k*;

if y = 'C' then

$I_{first} := I_{result}$;

$I_{second} := I[k + 1]$;

$k := k + 1$

else break

Issue I[1], *I*[2], ..., *I*[*foldable#*] *instructions as a folding group*

end

4 Folding strategies and performance

In this Section, folding strategies of different degrees of folding are proposed and examined based on the benchmark trace analysis. Performance in terms of reduced stack operations and speedup of the folding strategies are described. Finally, the cost and complexity issues of the decoder are discussed.

4.1 Proposed folding strategies

Based on the POC model of stack operations folding in Section 3, the number of combined *P*, *O*, and *C* operations in a single folding group can range up to thousands. Considering the cost/performance ratio and the limited time budget in folding, it may be necessary to fold only up to a small number of instructions. Based on the benchmark program traces, it is found that most of the foldable patterns consist of only 2 to 4 bytecode instructions (see data in Section 4.2). So we propose to examine in particular three folding strategies in which the foldable bytecode instructions are 2, 3, or 4, respectively. These 2-foldable, 3-foldable, and 4-foldable strategies are described below:

2-foldable: Folds two bytecode instructions. The folding group can be any one of the proposed foldable patterns as shown in Table 3.

Table 3: Proposed 2-foldable patterns

I1	I2
P	O_E
P	O_B
P	O_C
P	C
O_E	C
O_C	C

3-foldable: Folds up to three bytecode instructions. Besides the folding capability of the 2-foldable strategy, the folding group may be any of the proposed 3-foldable patterns as shown in Table 4.

Table 4: Proposed 3-foldable patterns

I1	I2	I3
P	P	O_E
P	P	O_B
P	P	O_C
P	O_E	C
P	O_C	C
O_E	C	C
O_C	C	C

4-foldable: Folds up to four bytecode instructions. Besides the folding capability of the 3-foldable strategy, the folding group may be any of the proposed 4-foldable patterns as shown in Table 5.

Table 5: Proposed 4-foldable patterns

I1	I2	I3	I4
P	P	P	O_E
P	P	P	O_B
P	P	P	O_C
P	P	O_E	C
P	P	O_C	C
P	O_E	C	C
P	O_C	C	C
O_E	C	C	C
O_C	C	C	C

In addition to the 2-, 3- and 4-foldable patterns, 5-, 6-, ..., *n*-foldable patterns (*n* may be any positive number, but in our benchmark traces, the maximum *n* found is 11) are also possible. We do not include those foldable patterns in our study because of the need for a very complex decoder. In Section 4.2, we present the projected performance bounds of the different folding strategies, including the theoretical *n*-foldable folding.

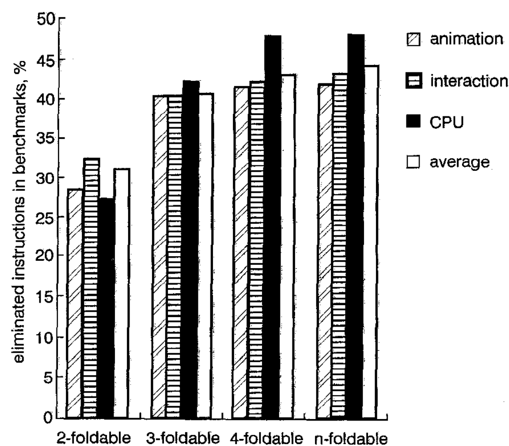


Fig. 4 *Percentage of eliminated stack operations in our benchmarks*
n-foldable means theoretical perfect folding

4.2 Performance of folding strategies

Fig. 4 shows the percentage of eliminated (folded) stack operations in our benchmark programs [8]. As shown in Table 2, the percentage of all stack operations (*P*

type and C type as indicated by Sun Microelectronics [4]) is about 51% of the instruction count. The average percentage of stack operations eliminated by the folding strategies are 31%, 41%, 43% and 44% for 2-, 3-, 4- and n-foldable, respectively, of all the instructions. As a result, if 4-foldable is adopted, the instruction mix percentage of stack operations will be reduced from the original 51% to 14% ((51–43)/(100–43)) in our benchmarks.

Fig. 5 shows the eliminated stack operation ratios with respect to all stack operations only. In this Figure, the folding ratio for the picoJava-I architecture [2, 4, 9, 10], as announced in October 1996, is also shown. Note that the Sun Microelectronics benchmark suite is different from ours.

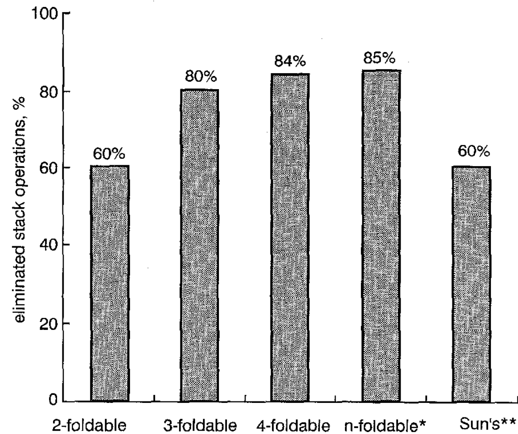


Fig. 5 *Percentage of eliminated stack operations with respect to all stack operations*
* n-foldable means theoretical perfect folding
** Sun's picoJava-I with benchmark suite which is different from ours

4.3 Speedup projection of folding

The Java bytecode instructions are typically executed on a Java Virtual Machine [6]. To estimate the program execution speedup due to folding, the instruction execution cycles for the 17 instruction types [8] must be assumed. The other necessary assumptions are that there is no cache miss and the pipeline never stalls. Figs. 6 and 7 show the speedup of executing stack operations only and the overall speedup that each folding strategy can contribute, respectively.

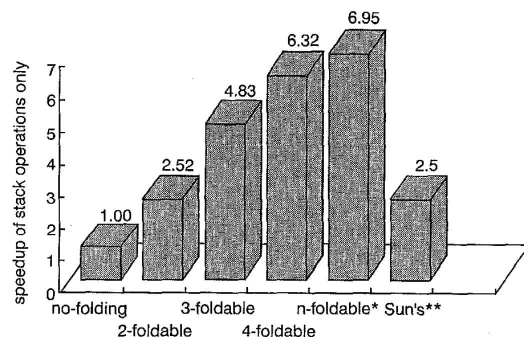


Fig. 6 *Speedup of executing stack operations only for each strategy*
* n-foldable means theoretical perfect folding
** Sun's picoJava-I with benchmark suite which is different from ours

4.4 Design issues

The decoder width has a great impact on the efficiency of the folding strategies. Fig. 8 shows the percentage of eliminated instructions against the decoder width for each folding strategy. It is obvious from this Figure that a bytecode decoder width of up to eight bytes, a moderate amount, is sufficient for any folding strategy. As the number of foldable stack operations decreases, the required decoder width may also be decreased without hurting the performance too much.

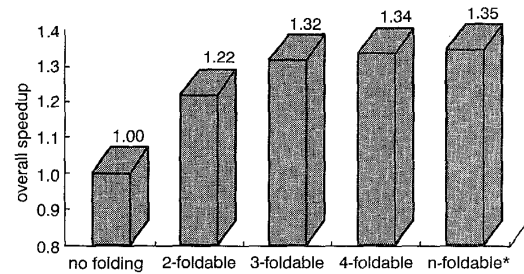


Fig. 7 *Overall speedup for each strategy*
* n-foldable means theoretical perfect folding

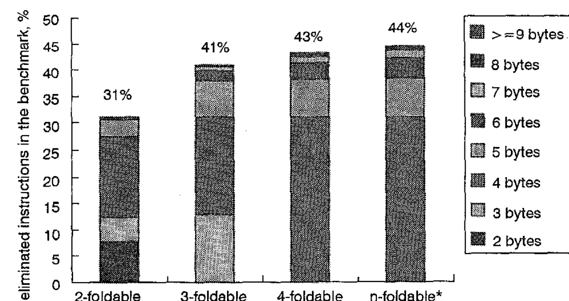


Fig. 8 *Percentage of eliminated instructions against decoder width*
* n-foldable means theoretical perfect folding

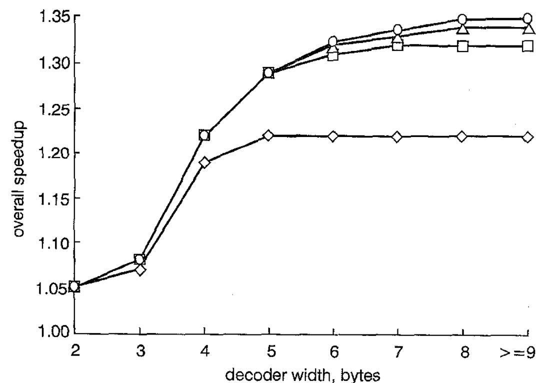


Fig. 9 *Overall program speedup against both decoder width and folding strategy*
—◇— 2-foldable
—□— 3-foldable
—△— 4-foldable
—○— n-foldable (i.e. theoretical perfect folding)

We next focus on the achievable speedup of programs, the most persuasive performance index. Fig. 9 shows the overall program speedup against both the decoder width and folding strategy. These curves show that the 3-foldable strategy with a decoder width of 6 bytes is at the knee point, a performance/cost design choice. This may not hold when designing a real Java processor, however. Because an instruction fetch width of 8 bytes is not too wide and most instruction caches have a line size of a power of 2 bytes, a decoder width of 8 bytes is the natural choice without much extra cost

for multiple instruction cache accesses, buffering, and byte extraction. Furthermore, the incurred extra decoder hardware cost in going from 6-byte decoding to 8-byte is insignificant compared with the whole Java processor design. Hence, the suggested decoder width is 8 bytes, and this width will be used in the subsequent discussion. For an 8-byte decoder width, the performance gained from 3-foldable to 4-foldable is that additional (43.12% – 40.77%) = 2.35% instructions can be eliminated. In our preliminary designs, the decoder circuit complexities for 3-foldable to 4-foldable strategies are comparable to a fixed decoder width, since the Java bytecode instruction length are variable in reality. Hence, we suggest that the 4-foldable strategy with a decoder width of 8 bytes may be the best choice for practical designs. The richer-foldable strategies with any decoder width are not recommended, because the statistical data show that only 44.18% of instructions can be eliminated.

5 Design of folding mechanism

In this Section, the POC model of stack operations folding is implemented using the VLSI standard cell library. This implementation is necessary in evaluating the delay time of the actual folding circuit, which is a part of the instruction decoder.

5.1 Logic design

According to the POC model, one can see that the same folding procedure for two instructions can be applied repeatedly to fold more than two instructions. For the same reason, if we implement the folding function into a basic 2-fold folding unit, then the higher degree of folding can be realised by simply cascading such folding units. In this implementation, we use four bits to represent the POC type of each instruction. Table 6 lists the instruction types for Java stack operations folding and their bit representations.

Table 6: Bit representation of instruction types for POC model

Type	Symbol	P		O	C
		bit 3	bit 2	bit 1	bit 0
Producer	P	1	0	0	0
Operator	OE	0	1	0	0
	OB	0	0	1	0
	OC	0	1	1	0
	OT	0	0	0	0
Consumer	C	0	0	0	1

As shown in Fig. 10, an n-foldable folding logic can be constructed using only basic folding units. An obvious advantage of this is its excellent scalability. Each folding unit has three inputs and three outputs. They are:

Inputs:

(1) 4-bit POC_N bus from instruction decoder or previous folding unit for instruction$_N$ type bits.

(2) 4-bit POC_{N+1} bus from instruction decoder for instruction$_{N+1}$ type bits.

(3) One *continue* line to indicate the current folding status *C/E* as shown in Fig. 2.

Outputs:

(1) *Foldable* line to indicate whether the input instructions are foldable or not.

(2) 4-bit $POC_{combined}$ bus representing the type of the instruction resulting from folding, which is to be checked for further foldability.

(3) One *continue* line to indicate if the resulting instruction can be checked for further foldability with the following instruction.

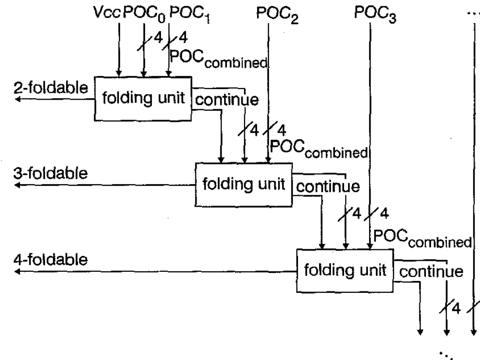


Fig. 10 *Scalable folding logic architecture*

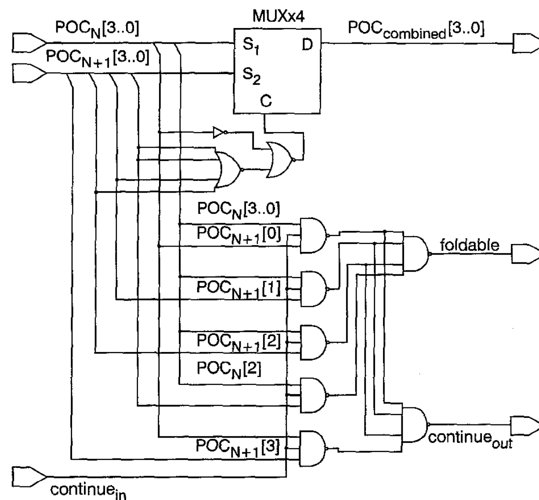


Fig. 11 *Schematic view of the folding unit*

Fig. 11 shows the gate-level implementation of the folding unit. As shown in Fig. 11, the $POC_{combined}$ will equal POC_{N+1} if the first input instruction is of P type and the second input instruction is not of O_T type. Otherwise, the POC_N will be selected. And the *foldable* and *continue* signal can be generated using the following formula:

$$foldable = (POC_N[3] \cdot (POC_{N+1}[1] + POC_{N+1}[2]) + POC_{N+1}[0] \cdot (POC_N[3] + POC_N[2])) \cdot continue_{in}$$

$$continue_{out} = (POC_N[3] \cdot (POC_{N+1}[3] + POC_{N+1}[2] + POC_{N+1}[1]) + POC_{N+1}[0] \cdot POC_N[2]) \cdot continue_{in}$$

We use negative logic devices to implement the above formula. This is necessary if a very high clock frequency is required.

5.2 Delay calculation

To calculate the timing overhead of introducing the folding circuit, we used the Cadence Verilog-XL v2.2.1

and the Verilog Delay Calculator v4.12 to calculate its delay times. The standard cell library used is the COMPASS 0.6μm cell library with a SPTM (single poly triple metal) CMOS process. In this library, there are two types of core cells, high performance (HP) and high density (HD). To predict the delay more accurately, the delay calculation uses the more accurate Verilog ISM (input slope model) model with both types of cells instead of the linear model. The delay calculation environment is shown in Table 7. The results for both HP and HD core cells are shown in Fig. 12. If we are to implement the 4-foldable strategy, the corresponding delay is 3.62ns or 6.74ns for HP or HD core cells, respectively. Assuming that the other delay time for wiring, gates and latches is less than 6ns, then the 4-foldable strategy can be implemented to run at a clock speed of 100MHz.

Table 7: Environment for Verilog delay calculation

Options	HP core cells	HD core cells
Calculation mode	estimate	estimate
Library name	cb60hp231d	cb60hd231d
Placement sites	1000	1000
Process	typical	typical
voltage	5.000	5.000
Temperature	25.000	25.000
Iteration slew precision	0.010	0.010
Rise slew time	0.500	0.500
Fall slew time	0.500	0.500

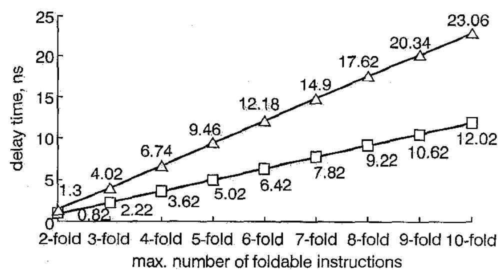


Fig. 12 *Delay time for folding circuits with various degree of foldability*
—◇— HP core cells
—△— HD core cells

6 Conclusion

In this paper, we have focused on solving an inherent problem of the stack machine that handles instruction level parallelism; the true data dependency. A method to deal with this problem, stack operations folding, was presented. A generalised stack operations folding model, the POC model, was also introduced. Various folding strategies based on this POC model were proposed and evaluated. Simulation results show that 2-, 3-, 4-, and *n*-foldable strategies can eliminate 31%, 41%, 43%, and 44% of stack operations in the entire Java program trace files, respectively. Compared with the theoretically perfect folding that can eliminate 44% of such stack operations, the 2- to 4-foldable strategies can achieve 70%, 94% and 98% efficiencies with much less hardware cost. If we translate the instruction counts into clock cycles, the corresponding speedups are 1.22, 1.32 and 1.34, respectively, as compared to a traditional Java stack machine without stack operations folding support.

The proper decoder width is also studied based on the many folding strategies of various degrees. Simulation data shows that the 4-foldable strategy is a good choice if an 8-byte decoder width is used. A sample folding unit design based on the POC model is then presented, and delay calculation shows that this folding mechanism can viably be run at 100MHz when designed with 0.6μm CMOS standard cells.

In this study, we have presented the performance analysis of stack operations folding, the POC model, the POC based folding algorithm, and the folding circuit implementation. Simulation results show that the POC based stack operations folding design is very cost effective due to its simplicity.

7 Acknowledgments

This paper presents partial results of a research project financed by the Computer and Communications Laboratories of Industrial Technology Research Institute of ROC under contract no. G3-86040.

8 References

1 LENTCZNER, M., and TECHNOLOGY, G.: 'Java's virtual world.' Microprocessor Report, March 1996
2 SUN MICROELECTRONICS: 'picoJava™ I microprocessor core architecture'. http://www.sun.com/sparc/whitepapers/wpr–0014–01
3 KOOPMAN, P.: 'Stack computer'. http://www.cs.cmu.edu/
4 O'CONNOR, J.M., and TREMBLAY, M.: 'PICOJAVA-1: the Java virtual machine in hardware', *IEEE Micro*, 1997, **17**, (2), pp. 45–53
5 SUN MICROELECTRONICS: 'Sun blazes another trail – introducing the microJava™ 701 microprocessor.' http://www.sun.-com/sparc/hottopics/microJava.html
6 LINDHOLM, T., and YELLIN, F.: 'The Java™ virtual machine specification' (Addison–Wesley, 1996)
7 GOSLING, J., JOY, G., and STEELE, G.: 'The Java™ language specification' (Addison–Wesley, 1996)
8 TSENG, H.M., CHANG, L.C., TON, L.R., KAO, M.F., SHANG, S.S., and CHUNG, C.P.: 'Performance enhancement by folding strategies of a Java processor.' International conference on *Computer systems technology for industrial applications – Internet and multimedia*, CSIA'97, Hsinchu, Republic of China, April 1997, pp. 286–293
9 TURLEY, J.: 'Sun reveals first Java processor core.' Microprocessor Report, October 1996
10 CASE, B.: 'Implementing the Java virtual machine.' Microprocessor Report, March 1996