A software/hardware cooperated stack operations folding model for Java processors

Lee-Ren Ton *, Lung-Chung Chang, Jyh-Jiun Shann, Chung-Ping Chung

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 30050, Taiwan, ROC

Received 22 October 2002; accepted 28 March 2003

Available online 5 December 2003

Abstract

Java has become the most important language in the Internet area, but the execution performance of Java processors is severely limited by the true data dependency inherited from the stack architecture defined by Sun’s Java Virtual Machine. A sequential hardware-based folding algorithm—POC folding model was proposed in the earlier research to eliminate up to 80.1% of stack push and pop bytecodes. The remaining stack push and pop bytecodes cannot be folded due to the sequential checking characteristic of the POC folding model. In this paper, a new software/hardware cooperated folding algorithm—T-POC (Tagged-POC) folding model is proposed to enhance the folding ability of the POC-based Java processors to fold the remaining stack push and pop bytecodes. While executing the bytecodes, bytecode grouping and rescheduling are done by a T-POC bytecode rescheduler to generate the new binary class images in memory. With the cooperation of the hardware-based POC folding model, higher execution performance can be achieved by executing the newly generated class images. Statistical data show that 94.8% of stack push and pop bytecodes can be folded, and the overall execution speedups of 2-, 3-, and 4-foldable strategies are 1.72, 1.73 and 1.74, respectively, as compared to a single-pipelined stack machine without folding.

Keywords: Java processor; Stack operations folding; POC folding model; T-POC folding model; T-POC bytecode rescheduler

1. Introduction

Internet technologies have ushered in a new era for computer-related products. That includes the Internet-enabled PCs, NC clients, NetPCs, smart hand-held devices (PDA, smart phone), NetTVs (Internet TV, Internet set-top box), game consoles, etc. Java (Gosling et al., 1996) is the most popular language over the Internet owing to its security, robustness and write-once-run-anywhere characteristics. Java computing becomes the fourth computing wave after terminal-host computing, PC-computing and client–server computing. It is an emerging and open technology, suitable for new applications and new developers.

Java Virtual Machine (JVM) (Lindholm and Yellin, 1996; Venners, 1998) is a stack-based machine and its performance is limited by true data dependency. A means of avoiding such a limitation, i.e. stack operations folding, was proposed by Sun Microsystems. In Sun’s picoJava-II processor (Sun, 1999a,b), the stack operations folding is implemented in the Instruction Folding Unit (IFU). While executing, pre-defined and pre-stored folding patterns are compared with bytecodes in instruction stream sequentially. In our earlier study, we proposed a systematic folding solution named the Producer, Operator, and Consumer (POC) folding model (Chang et al., 1998, 2000). In this folding model, dynamic folding rules are used to check the foldability instead of the pre-defined and pre-stored folding patterns. The POC folding model can be implemented into the instruction decoder unit of a Java processor while maintaining low hardware cost (Chang et al., 1998, 2000).

Java classes (Lindholm and Yellin, 1996; Venners, 1998) that are compiled from Java sources can be run on any platforms that provide the JVM. While executing the classes, they are loaded by the Dynamic Class Loader and sent to the Class File Verifier for verification.
Finally they are sent to the execution engine for execution. Java processor is one kind of the hardware-based execution engines for bytecode execution. The Interpreters and Just-In-Time (JIT) compilers (Cramer et al., 1997) are of the software-based execution engines. In general, hardware-based approaches can achieve the highest execution performance. On the other hand, the software-based approaches are more popular and lower in cost. In this paper, a software/hardware cooperated scheme based on the POC folding model is proposed. With the new Java bytecode execution scheme, almost all the stack push and pop operations can be folded to achieve highest performance without introducing extra hardware cost.

In the original POC folding model, bytecodes are classified into three POC types. The JVM runtime data structures like Constant Register, Local Variable, and Operand Stack are used to classify the bytecode execution behavior. The Producer moves the data from Constant Register or Local Variable to Operand Stack. The Operator reads its sources from Operand Stack, executes, and writes the result back to Operand Stack. The Consumer moves the data from Operand Stack to Local Variable. The concept of the POC folding model is that P sends the data to the O or C instead of the Operand Stack. Similarly, the executing result of O is sent to Local Variable instead of Operand Stack. Consequently, bytecodes matching this concept can be folded together explicitly and the number of times in accessing to the top of stack pointer is reduced by the POC folding concept.

To keep the simplicity of the POC folding hardware, folding check is limited among continuous bytecodes. In other words, if one P type bytecode cannot be folded with its adjacent O or C type bytecode, it is executed sequentially. If we can scan the Java bytecodes and find out those discontinuous P type bytecodes and reschedule the sequence of bytecodes, the Java processor with the POC folding unit can execute the rescheduled bytecodes more efficiently. By combining the Tagged-POC (T-POC) bytecode rescheduler and the POC folding hardware on the Java processor, a hybrid folding approach is depicted in Fig. 1. The gray blocks represent the work done in the research to enhance the execution performance of the JVM. Other blocks remain the same as general JVM execution environment. Java classes are loaded by the Dynamic Class Loader and then rescheduled by the proposed T-POC bytecode rescheduler to improve the execution performance on POC-based Java processors.

The paper is organized as follows. Section 2 describes the simulation environment and the benchmarks used in this paper. The basic properties of the benchmarks and

Fig. 1. Execution environment of the proposed Java virtual machine.
the POC types distribution are also given. In Section 3, the detailed descriptions of the POC folding model are described. These include the folding check rules and a folding example based on the POC folding model. The T-POC folding model and the T-POC bytecode rescheduler algorithm are presented in Section 4. Simulation results for the proposed T-POC folding model is given in Section 5. Finally, Section 6 concludes the benefits and applications for the proposed T-POC folding model.

2. Simulation environments and POC types distribution

In this section, basic simulation environment is introduced and bytecode distribution properties are analyzed according to the POC types. SPECjvm98 (SPECjvm98, n.d.) benchmark programs are sent to the bytecode tracer to generate the runtime bytecode traces. By analyzing the traces, the basic properties of the benchmarks for the POC and T-POC folding models are shown.

2.1. Simulation environments

By modifying the Sun’s JDK Virtual Machine (JDK, n.d.), runtime bytecode traces (NMI, n.d.) 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 T-POC folding models. We use the SPECjvm98 benchmarks as our simulation source data. There are three input data set scales for the SPECjvm98 benchmarks: s1, s10, and s100. According to the suggestion of the SPECjvm98, the s1 data set is small and is used for prototype evaluation. For research papers, either s10 or s100 can be chosen. We had tried to gather the s100 traces. The result is that the traces generated by the s100 data set are too large to fit into the 20GB hard disk. 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 and is shown in Table 1.

2.2. POC types distribution

There are 203 bytecodes defined in the JVM. In the POC folding model, these bytecodes are classified into three types according to the usage of source and destination storage. In this research, the O type bytecodes are further divided into four subtypes according to their execution behavior. The first subtype is ‘O_E’, which is a collection of bytecodes that will be executed in execution units. The second subtype is ‘O_B’, which is a collection of bytecodes that conditionally branch or jump to target address. The third subtype is ‘O_C’, which is a collection of bytecodes that will be executed in micro-ROM or trapped as a sequence of instructions. The last subtype is ‘O_T’, which will force the folding check to be terminated for the difficulty in performing folding. The ‘O_ALL’ notation means the summation of the four sub-types of O bytecode instructions. The occurrence percentages for each type of bytecodes are shown in Table 2.

Table 1
Property of dynamic bytecode counts in SPECjvm98 benchmark suite

<table>
<thead>
<tr>
<th>Program name</th>
<th>Bytecode counts (million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1137</td>
</tr>
<tr>
<td>db</td>
<td>74</td>
</tr>
<tr>
<td>jack</td>
<td>341</td>
</tr>
<tr>
<td>javac</td>
<td>63</td>
</tr>
<tr>
<td>jess</td>
<td>121</td>
</tr>
<tr>
<td>mpegaudio-3</td>
<td>1220</td>
</tr>
<tr>
<td>raytracer</td>
<td>160</td>
</tr>
</tbody>
</table>

Table 2
Property of occurrence percentages for each POC type

<table>
<thead>
<tr>
<th>Program name</th>
<th>P</th>
<th>O_E (%)</th>
<th>O_B (%)</th>
<th>O_C (%)</th>
<th>O_T (%)</th>
<th>O_ALL (%)</th>
<th>C (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>40.02</td>
<td>26.24</td>
<td>8.54</td>
<td>19.61</td>
<td>1.39</td>
<td>55.77</td>
<td>4.21</td>
</tr>
<tr>
<td>db</td>
<td>44.14</td>
<td>20.48</td>
<td>13.51</td>
<td>14.13</td>
<td>5.63</td>
<td>53.76</td>
<td>2.10</td>
</tr>
<tr>
<td>jack</td>
<td>32.67</td>
<td>25.04</td>
<td>13.10</td>
<td>27.87</td>
<td>0.46</td>
<td>66.48</td>
<td>0.85</td>
</tr>
<tr>
<td>javac</td>
<td>41.82</td>
<td>15.00</td>
<td>14.76</td>
<td>21.94</td>
<td>3.31</td>
<td>55.01</td>
<td>3.16</td>
</tr>
<tr>
<td>jess</td>
<td>44.00</td>
<td>9.31</td>
<td>19.78</td>
<td>20.72</td>
<td>2.43</td>
<td>52.23</td>
<td>3.77</td>
</tr>
<tr>
<td>mpegaudio-3</td>
<td>45.61</td>
<td>38.00</td>
<td>4.20</td>
<td>8.99</td>
<td>1.85</td>
<td>53.04</td>
<td>1.35</td>
</tr>
<tr>
<td>raytracer</td>
<td>39.35</td>
<td>14.28</td>
<td>16.44</td>
<td>28.27</td>
<td>1.03</td>
<td>60.03</td>
<td>0.62</td>
</tr>
<tr>
<td>Average</td>
<td>41.09</td>
<td>21.19</td>
<td>12.90</td>
<td>20.22</td>
<td>2.30</td>
<td>56.62</td>
<td>2.29</td>
</tr>
</tbody>
</table>
bytecodes per cycle from 1 to the maximum of about 1.77 with the assumption of one cycle execution for each bytecode. This also indicates that the maximum performance speedup of a Java processor with folding is about 1.77 as compared to a stack machine without folding.

3. The POC folding model

The basic concept about stack operations folding was proposed by the Sun Microsystems. Some fixed patterns are selected and built in the instruction decoder of the Java processor. As in the implementation of the Sun’s picoJava-II processor, nine folding patterns are defined in the IFU to enable the maximum foldability of four. In contrast to the approach of defining the fixed folding patterns, the POC folding model based on the dynamic folding checking is presented. A simple example for the POC folding model is given to illustrate the operations of how the POC folding is done.

3.1. Folding check rules

The folding check procedure of the POC folding model is done by checking one bytecode \( N \) (been folded or not) and its next bytecode \( N + 1 \). By examining their POC types, operand sources, operand destinations, data types and widths, the POC folding model determines whether they are foldable or not. If they are foldable, the resulting folded instruction will become the new bytecode \( N \), and it will be checked with its next bytecode \( N + 1 \) by applying the same POC checking procedure. This procedure will be repeated until an ending case is encountered in the POC folding check. The related notations are defined as below.

\[\delta\] folding of bytecodes \( N \) and \( N + 1 \)

\( P_{Sn,Wn} \) producer with source ‘Sn’ of width ‘Wn’, and destination Top of Operand Stack (TOS) of width ‘Wn’

\( O_{Sn,Wn/Dn,W'n} \) operator with source Sn of width Wn, and destination Dn of width W’n

\( C_{TOS,Wn/LV,Wn'} \) consumer with source TOS of width Wn, and destination Local Variable (LV) of width W’n

Two possible relations, as listed below, exist between two consecutive bytecodes:

SI Serial Instructions, referring to a situation in which bytecodes \( N \) and \( N + 1 \) are serialized bytecodes that are not foldable

FI Foldable Instructions, referring to a situation in which bytecodes \( N \) and \( N + 1 \) are foldable

After the folding check, the indication status for further folding check is:

C Continuing status, referring to a situation in which the folded bytecode (\( N \) plus \( N + 1 \)) may be verified for further foldability

E Ending status, referring to a situation in which the folded bytecode (\( N \) plus \( N + 1 \)) cannot be folded anymore. Therefore, the folding check procedure must be terminated

Fig. 2 shows the foldability checking rules for bytecodes \( N \) and \( N + 1 \) in the POC folding model. The foldability check continues if the current indication status is C. The procedure stops if the indicating status is E. An illustrative example of the POC folding model is given in the following subsection to show how to use the foldability checking rules shown in Fig. 2.

<table>
<thead>
<tr>
<th>Instruction ( N+I )</th>
<th>Instruction ( N+I )</th>
<th>Instruction ( N+I )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( C_{TOS,Wn} )</td>
</tr>
<tr>
<td>( P_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( C_{TOS,Wn} )</td>
</tr>
<tr>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
</tr>
<tr>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
</tr>
<tr>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
</tr>
<tr>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
</tr>
<tr>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
<td>( O_{Sn,Wn} )</td>
</tr>
</tbody>
</table>

Note 1: Assume that bytecodes \( N \) and \( N + 1 \) have matched data types and widths. Otherwise, they cannot be folded together, with the operating result of new bytecode \( N \) with “SI/E” state.

Fig. 2. Foldability check for bytecodes \( N \) and \( N + 1 \).
the bytecode are folding into a single one, the execution procedure can be done by using simple comparison
are now only one cycle for execution. From the micro-
four bytecodes that requires four cycles to be executed
4. The tagged POC folding model
As depicted in Section 3, the POC folding model uses
the folding rules to check whether bytecode N and
bytecode \( N + 1 \) is foldable or not. This checking procedure limits the discontinuous P type bytecodes to be
folded by the hardware-based POC folding unit. In this
section, the discontinuous P type bytecodes are rear-
ranged nearby the corresponding O or C type bytecodes
by a T-POC bytecode rescheduler. With the T-POC
bytecode rescheduler, we propose a T-POC folding model
which combines the software-based T-POC bytecode
rescheduler and the original POC-based Java processor
for Java bytecode execution. An algorithm for the T-
POC bytecode rescheduler in cooperating with the
hardware-based POC folding is given and a folding
example is shown to help readers to understand how the
folding procedures are done in both software and
hardware side.

4.1. Extended folding groups
The bytecode sequence is rescheduled by grouping the
folded bytecodes into a hierarchical structure in the T-
POC folding model. Each folding group is expressed as
an Extended Folding Group (EFG) with the notation of
EFG\((i, j)\). The EFG related notations and properties are
explained below.

1. An EFG tree may either be empty or consist of one or
more EFGs. An EFG tree is one kind of Dynamic
Dependency Graph with folding for the Java Instruction
Set Architecture (ISA).
2. An EFG tree is not cyclic.
3. If the height of the EFG tree is \( h \), then the lower
bound for execution cycles is \( h \).
4. If there are \( k \) EFGs with the same level \( i \), EFG\((i, j)\)
means that the EFG is located at the \( i \)th level of the
EFG tree and the \( j \) is the ordering number of these
EFGs, for all \( i \) and \( j \) meet the conditions of \( 1 \leq i \leq h \)
and \( 1 \leq j \leq k \).
5. The number \( k \) in previous description indicates the
maximum number of issued EFGs for an in-order
superscalar machine at level \( i \).
6. If EFG\((i, j)\) depends on the EFG\((i_1, j_1)\), then
\( 1 \leq i_1 < i_1 < h \) holds.

As shown in Fig. 3, the outmost extended folding group
EFG\((5, 1)\) consists of \( P_1 \), an inner extended folding
group EFG\((4, 1)\) and \( O_{E1} \). The EFG\((4, 1)\) consists of
EFG\((2, 1)\), EFG\((3, 1)\) and \( O_{E6} \). Again, the EFG\((2, 1)\)
consists of \( P_3 \), EFG\((1, 1)\) and \( O_{E2} \). Finally, EFG\((1, 1)\)
consists of \( P_4 \), \( P_5 \) and \( O_{E1} \). In this example, only
EFG\((1, 1)\) and EFG\((1, 2)\) can be folded directly by se-
quential hardware of the POC folding model. EFGs with
higher-level numbers (EFG\((2, 1)\), EFG\((2, 2)\), EFG\((3, 1)\),
EFG\((4, 1)\) and EFG\((5, 1)\)) cannot be folded using the
POC folding model, unless their corresponding inner
folding groups have been executed. In this example, \( P_1 \)
cannot be folded with its adjacent code and it must be
issued first. \( P_3 \), \( P_6 \) and \( P_2 \) are treated as serial and non-
foldable instructions in the previous POC folding model.
With the T-POC folding model, they can be folded into

### Table 3
Annotated POC types

<table>
<thead>
<tr>
<th>Instruction no.</th>
<th>Bytecode name</th>
<th>Annotated POC types</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>1const_2</td>
<td>P_{const_2}/TOS.1</td>
</tr>
<tr>
<td>I2</td>
<td>iload index1</td>
<td>P_{L}/index1/TOS.1</td>
</tr>
<tr>
<td>I3</td>
<td>iadd</td>
<td>O_{E}/TOS.2/TOS.1</td>
</tr>
<tr>
<td>I4</td>
<td>istore index2</td>
<td>C_{TOS.1}/LV/index2</td>
</tr>
</tbody>
</table>

3.2. An illustrative example of the POC folding model
Assume that a sequence of bytecodes is I1–I4. Their
POC type notations are listed in Table 3.

The POC folding procedures are explained as the following steps:

1. \( P_{const_2,1}/TOS.1 \) operating with \( P_{L}/index1,1/TOS.1 \)

2. \( P_{const_2,1}/LV(index1,1)/TOS.1 \) operating with \( O_{E}/TOS.2/TOS.1 \)

3. \( O_{E}/const_2,1/LV(index1,1)/TOS.1 \) operating with \( C_{TOS.1}/LV(index2,1) \)

In the first step, bytecodes I1 and I2 are checked using
the folding rules shown in Table 2. The relation between
I1 and I2 is SI, and the folding check can be continued
because of the indication status is C. Consequently,
the I3 is checked with the new combined POC type of I1 and
I2 in the second step. The folding relation FI means that
the I1, I2 and I3 are foldable. The folding procedure
does not end here because the indication status is C. In
the third step, the I4 is checked with the combined POC
type of I1, I2 and I3. The folding relation of FI is still
hold, resulting in the foldability of four bytecodes. The
folding procedure ends because of the indication status
is E. The final combined instruction is sent to the
functional unit for execution. As a result, the original
four bytecodes that requires four cycles to be executed
are now only one cycle for execution. From the micro-
processor hardware design point of view, this folding
procedure can be done by using simple comparison
circuity in the instruction decoding stage. If four
bytecodes are folding into a single one, the execution
unit executes the folded one only. This greatly increases
the Issued Instructions Per Cycle (IIPC) for a single
pipeline Java processor.
distinct folding groups with their corresponding bytecodes to gain higher execution performance.

4.2. Algorithm for the T-POC bytecode rescheduler

In order to fold those discontinuous P type bytecodes, we reschedule these discontinuous bytecodes nearby the corresponding O or C type bytecodes before feeding them into Java processor. The T-POC bytecode rescheduler does: (1) find out the EFGs by analyzing the original bytecode sequence; (2) decide the output sequence of the newly generated EFGs.

In the example of Fig. 3, the execution result of $EFG(1, 1)$ is one of the source operands of the $O_{E2}$ instruction in $EFG(2, 1)$. The execution of both $EFG(1, 1)$ and $EFG(2, 1)$ requires three clock cycles using the POC folding model (the first is $P_3$, followed by $EFG(1, 1)$, and the last is $O_{E2}$). In the T-POC folding model, the execution takes only two clock cycles. Instead of issuing the $P_3$ bytecode first in the POC folding model, the T-POC folding model issues the $EFG(1, 1)$ first. The result of the $EFG(1, 1)$ is then marked as a tag and combined with the $P_3$ bytecode to be folded into the $O_{E2}$ bytecode. The new tag is named as a $P'$ type bytecode. In order to perform further folding capability with the new $P'$ bytecode, we choose one free reserved opcode in JVM as the $P'$ tag. The $P'$ tag represents the result of the $EFG$ that writes data back to the operand stack. The POC type for the $P'$ tag is $P$. That is, no extra POC types are added in the original folding check rules. The attributes of $P'$ tag used in the POC folding model are listed below

$P' : POC = P, \quad SN = DF_{TOS}, \quad WN = 1, \quad DN = TOS, \quad WN_1 = 1$

The $P'$ tag is treated as a $P$ type bytecode whose source comes from the pipeline latches (e.g. data forwarding) or top of the operand stack. The notation of $DF_{TOS}$ is an abbreviation of data forwarding and the top of stack. The destination for $P'$ tag is the top of the operand stack. The numbers for both source and destination operands are one. The POC folding model is modified by adding the ability to recognize the $P'$ tag. As the example shown in Fig. 3, the tagged $POC$ types for $EFG(2, 1)$ are listed in Table 4. The $P'$ tag is used to represent the result of $EFG(1, 1)$.

With the $P'$ tag, the POC folding model is transformed to the T-POC folding model. The T-POC folding check procedures are depicted in Fig. 4. The $I1$ and $I2$ bytecodes are combined to a new $P'$ type for further checking. In the second step, the $O_E$ type of $I3$ is combined

![Fig. 3. An example of extended folding groups.](image)

![Fig. 4. An example of the T-POC folding check procedures.](image)
with the previously generated P and the folding check relation is foldable.

In Figs. 3 and 4, the results of $EFG(2,1)$ and $EFG(3,1)$ are the source operands of $O_{E6}$ bytecode. Consequently, two $P'$ tags are generated to represent the execution results of $EFG(2,1)$ and $EFG(3,1)$, respectively. Both $P'$ tags will be assigned before the $O_{E6}$ bytecode to form a folding group while executing. Again, the result of the $O_{E6}$ bytecode will be represented as another $P'$ tag consumed by the $O_{B1}$ bytecode. By applying the $T$-$POC$ folding model repeatedly, the EFGs among the bytecode sequence are generated. These EFGs can be executed on the $POC$ folding hardware efficiently. The folding algorithm for the $T$-$POC$ folding model is shown in Fig. 5. In this algorithm, folding check is done within the basic block of size $M$, which is implemented as a do-while loop. The $POC$ folding model with the foldability $N$ is used as the kernel of folding check rule. Folded or non-folded bytecodes are stored in the array of FG with the corresponding folding group information like folding relations (FI or SI) and folded number. In inner while loop, source operands from stack are assigned by $P$ bytecode or $P'$ tag. If there are enough $P$ or $P'$ type results in previous folding groups, these $P$ or $P'$ types are folded as source operands of the current folding group and their Available flags are cleared. If the number of $P$ or $P'$ type results is insufficient, extra $P'$ bytecodes are added according to the required number of sources. As defined in the JVM, the maximum required number of source operands from stack is four. As a result, the time complexity for the do-while loop is $O(M)$. The $EFG$s are gathered from the FG in last for loop according to the availability of result and whether the FG contains non-P type bytecode. Again, the time complexity for the for loop is $O(M)$ because $M$ bytecodes can be generated to at most $M$ EFGs.

4.3. Java processor execution model for the software/hardware cooperated $T$-$POC$ folding

In order to further recognize and execute the $P'$ tags generated by the $T$-$POC$ bytecode rescheduler, the decoder unit for a Java processor using $POC$ folding model should be slightly modified to decode the newly defined opcode for the $P'$ tag. As the same example in Fig. 3, the original bytecode sequence is listed below

$P_1P_2P_3P_4P_5O_{E1}O_{E2}P_6P_7P_8O_{E3}P_9O_{E4}C_1O_{E5}O_{E6}O_{B1}$

After the $T$-$POC$ folding, the newly generated bytecode sequence is:

$P_1P_2P_3O_{E1}P_4P_5O_{E2}P_6O_{E3}P_7O_{E4}P_8O_{E5}C_1P_9O_{E6}P_9P'O_{B1}$

The source operand field of the $P'$ tags comes from the nearest pipeline latches in bypassing circuitry, or from top of stack. In order to explain how $P'$ tag gets its source operand correctly, the detailed operations of the newly generated bytecode sequence are executed using the pipeline stages defined in Sun’s picoJava-II processor. There are six pipeline stages and their occurring sequence is $F$ (Fetch), $D$ (Decode), $R$ (Register), $E$ (Execution), $C$ (Cache) and $W$ (Write).

While decoding the O type bytecodes, an identification number for the execution result is assigned and recorded in the decoder pipeline latch. The $P'$ or C type bytecodes with sources from top of stack will check the identification numbers in the pipeline latches. If there are some valid identification numbers in the $E$ and $C$ pipeline stages, the $P'$ tag or C type bytecode will get the nearest one through forwarding. In other words, the data in $E$ stage owns higher priority than that in $C$. If both $E$ and $C$ pipeline latches contain no valid identification number, the operand should be read from the top of stack. Consequently, the identification number assigning circuit and the forwarding mechanism described above are the same as processors without $P'$ tag. The only overhead is to add a new entry for the $P'$ tag in decoder unit, which maintains the simplicity of the low-cost Java processor design.
In Fig. 6, the execution flow with picoJava-II pipeline for the newly generated bytecode sequence is shown. Most O type bytecodes except the O_E5 require two source operands. In cycle 7, both of O_E6 operands come from P tags. It is obvious that P_E is in the previous E stage and P_2 has already been written to top of stack. Consequently, P_4 can be forwarded directly from the E stage and P_2 is read from the top of stack. With the same 4-foldable POC folding unit, the newly generated sequence takes eight cycles to execute, as compared to the 11 cycles using the original sequence.

5. Folding performance and considerations

In this section, folding performance of the T-POC folding model and performance comparisons based on various folding models are shown. For the fairness of performance comparisons, the hardware parameters for simulation are configured the same as defined in the picoJava-II processor. That is, instruction queue size is seven bytes and the maximum foldability is four. Furthermore, extra bytecode memory requirement for T-POC bytecode rescheduler and EFGs are given.

5.1. Folding performance comparisons

The percentages of folded stack push/pop bytecodes for the T-POC bytecode rescheduler are shown in Fig. 7. Various foldabilities of 2-, 3-, 4- and max-foldable are shown for comparison. It is obvious that 4-fold performs almost the same as what max-fold does. In average, the percentages of folded stack operations for 2-, 3-, 4- and max-fold are 86.3%, 94.0%, 94.8% and 94.8%, respectively. The IIPC for the T-POC folding model are shown in Fig. 8. In average, the IIPC for 2-, 3-, 4- and max-fold are 1.72, 1.73, 1.74 and 1.74, respectively. With the T-POC folding model, a single-pipelined Java processor can achieve the competitive performance as compared to a two-issue superscalar processor.

In Figs. 9 and 10, the performance comparisons for the picoJava-II, POC folding model and T-POC folding model are shown. The foldability in both POC and T-POC folding models is four, which is the same as picoJava-II. The average percentages of folded stack operations for the picoJava-II, POC folding model and T-POC folding model are 39.6%, 80.1% and 94.8%, respectively. Furthermore, the average IIPC for each folding model are 1.25, 1.54 and 1.74, respectively.
5.2. Memory space overhead for T-POC bytecode rescheduler and EFGs

In this subsection, memory space overhead of the T-POC bytecode rescheduler is shown. Bytecodes in the same basic block with \( M \) instructions are fed into the T-POC folding algorithm shown in Fig. 5. \( M \) folding groups are allocated as internal used variables. Later, the folding groups will be transformed to EFGs to be executed in the Java processor. As a result, memory space is required for two areas. One is the \( M \) folding groups inside the T-POC bytecode rescheduler, and another one is the EFGs outside the T-POC bytecode rescheduler. At first, the size of data structure for the \( M \) folding groups inside the T-POC bytecode rescheduler is analyzed. As shown in Fig. 11, over 90% basic block contains less than 16 bytecodes for most of the benchmark programs, excluding the computation-oriented
compress and the mpegaudio-3 programs. Consequently, if the $M$ equals to 16, most basic blocks will be covered except those two heavy computation programs. If the $M$ is further added to 64, over 90% basic blocks for the SPECjvm98 using s10 data set will be covered. For most bytecodes except the two variable length bytecodes of lookupswitch and tableswitch, five bytes (one for opcode, four for operands) are enough to store the required information. The indication flags like Available, O_Flag and POC information consume at most two bytes for each folding group. Consequently, total required size for the folding groups equals to $M \times (N \times 5 + 2)$ bytes. If the $M$ is varied from 16 to 64, the required size for $N$ equals to 4 (4-foldable) is varied from 352 bytes to 1408 bytes. We think that this is acceptable while comparing with other software-based JVM implementation of JITs or Interpreters.

Second, we will analyze the output codes of the T-POC bytecode rescheduler. As mentioned before, the T-POC bytecode rescheduler rearranges the P type bytecodes by adding $P'$ tags in the EFGs. Consequently, the percentages of added $P'$ tags are collected in our simulation. As shown in Fig. 12, 37.85% of $P'$ bytecodes are added in the run-time memory in average. This seems too terrible to use the T-POC folding approach. In fact, we still think this acceptable because the actual bytecode size for a Java program is small as compared to constant pool and runtime area as defined in the Java’s class file. For example, the bytecode size for the javac benchmark program is about one-fifth of the corresponding class file size. Consequently, the T-POC bytecode rescheduler, while introducing less than 10% of memory space overhead, delivers 13% performance gain over the POC folding model without adding extra cost for the Java processor design.

6. Conclusions

In this paper, the software/hardware cooperated T-POC folding model is proposed. The output of the T-POC bytecode rescheduler consists of EFGs, which is sent to the Java processor with the POC folding unit for execution. The only modification required for the Java processor is to add a new $P'$ tag information in instruction decoder. With a 4-foldable T-POC folding
model, over 90% of stack operations are folded for every benchmark programs in SPECjvm98 using s10 data set. The resulting folding speedup is 1.74, and this achieves 98.3% of the theoretical upper bound of 1.77. The time complexity for the T-POC bytecode rescheduler is \( O(M \times N) \), where \( M \) is the basic block size and \( N \) is foldability that is usually less than 4. The required space for runtime data structure of the T-POC bytecode rescheduler is less than 2 Kbytes. The increased size of software folded bytecode sequence with extra \( P' \) tag is less than 10% as compared to the original class loaded into memory. The performance enhancement of the T-POC folding over the POC folding is 13%.

In the future research, the algorithm used in T-POC bytecode rescheduler may be implemented directly by hardware if the speed limitation can be solved. If higher folding capability can be implemented by a simple folding circuitry, the T-POC folding can help the superscalar processors to exploit higher instruction-level parallelism while greatly reduce the complexity of the hardware dependency detection. With the future requirements of smart phones and other multimedia applications, the computation power is more important for a Java processor. The potential of the T-POC folding model contains both performance enhancement and complexity cost down. This may be one of the best solutions for a high performance Java processor using the T-POC folding model.

References


Lee-Ren Ton received the B.S. and M.S. degrees in Information Engineering and Computer Science from the Feng Chia University, Taiwan, in 1993 and 1995, respectively. He was a lecturer of the Department of Computer Science and Information Engineering at the National Chiao Tung University, Taiwan, and a hardware design engineer of Hurco Automation Ltd., while working towards the Ph.D. degree. He received the Ph.D. degree of Computer Science and Information Engineering, National Chiao Tung University, Taiwan in July 2002. Currently, he is with the Core Technology Division of Faraday Technology Corporation as a technology manager. His research interests include computer architecture, microprocessor system design, and VLSI/ISP design.

Lung-Chung Chang received the B.S. degree in Electrical Engineering from the Chung Yuan Christian College, Taiwan, in 1977, and the M.S. degree from the University of Southern California in 1987. He received the Ph.D. degree of Computer Science and Information Engineering, National Chiao Tung University, Taiwan in April 2002. Currently, he is with the Computer and 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 B.S. 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 M.S. 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 Ph.D. 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 B.S. degree from the National Cheng Kung University, Taiwan, in 1976, and the M.S. and Ph.D. degrees from the Texas A & M University in 1981 and 1986, respectively, all in electrical engineering. He was a lecturer in electrical engineering at the Texas A & M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Taiwan, where he is a professor. From 1991 to 1992, he was a visiting associate professor of computer science at the Michigan State University. From 1998, he joins the Computer and Communications Laboratories (CCL), Industrial Technology Research Institute (ITRI), Taiwan, as the Director of the Advanced Technology Center (ATC), and then the Consultant to the General Director. He returns to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.