An instruction folding solution for a Java processor

Tan Yiyu1, Yau Chihang2, Anthony S. Fong2 and Yang Xiaojian1

1 College of Information Science and Engineering, Nanjing University of Technology, No. 5 Xinmofan Road, Nanjing, China
2 Department of Electronic Engineering, City University of Hong Kong, Tat Chee Avenue, Kowloon Tong, Hong Kong

Java is widely applied into embedded devices and network applications. Java programs are compiled into Java bytecodes, which are executed in the Java virtual machine. The Java virtual machine is a stack machine and instruction folding is a technique to reduce the redundant stack operations. In this paper, a simple instruction folding algorithm is proposed for a Java processor named jHISC, where bytecodes are classified into five categories and the operation results of incomplete folding groups are held for further folding. In the benchmark JVM98, with respect to all stack operations, the percentage of the eliminated P and C type instructions varies from 87% to 98% and the average is about 93%. The reduced instructions are between 37% and 50% of all operations and the average is 44%.

Keywords: Instruction folding, Java processor, Java virtual machine, bytecode.

1. INTRODUCTION

With development of Internet, multimedia objects have proliferated on Internet, and web services and other network applications are becoming more and more important [20-22]. Due to its object-oriented features and corresponding advantages of security, robustness and platform independence, Java is widely applied in network applications and embedded devices, such as PDAs, mobile phones, TV set-up boxes and hand-held PCs [17]. Java programs are compiled into Java bytecodes, which are executed into the Java virtual machine. The Java virtual machine is a stack machine, where all operands, such as temporary data, intermediate results, and method arguments, are frequently pushed onto or popped from the operand stack during execution. Thus some redundant load or store operations are performed, which results in low execution efficiency and affects the system performance, especially in embedded devices, where real-time operations and low power consumption are needed in case of limited memory. For example, when the bytecode stream “iload_0, iload_1, iadd, istore_2” are executed in the Java virtual machine, firstly, the load type instructions iload_0 and iload_1 push data from the local variable onto the top of the operand stack. Secondly, the instruction iadd pops data from the top of the operand stack, operates on them and stores the operation result onto the stack. Finally, the store type instruction istore_2 moves the operation result from the top of the operand stack to the local variable. In this execution procedure, the operations to push the operands and operation result onto the operand stack, pop operands from the operand stack will consume extra clock cycles. Moreover, operations are executed one at a time by using the operand stack, thus introducing a virtual data dependency between the successive instructions, which restricts instruction level parallelism and adversely affects the system performance.

To address these shortcomings, Sun Microsystems introduced the notion of instruction folding [1-4], which was a technique to eliminate the unnecessary load or write-back operations to the operand stack by detecting contiguous instructions and executing them collectively as a single, compound instruction. For example, to execute the bytecode stream mentioned above, the generated compound instruction may read data into the ALU from the local variables directly, operate on them and write the operation result back to the local variables. Thus the intermedi-
ate operands and results will not be pushed onto or popped out from the operand stack.

In this paper, a new folding algorithm is presented in jHISC, a Java processor for embedded devices. The rest of this paper is structured as follows. The previous work on instruction folding is summarized in Section 2. The jHISC instruction set is described in Section 3. Section 4 depicts our proposed folding algorithm, including bytecode type definitions and categories, folding rules, system diagram, and foldability detection procedure. In Section 5, the performance estimation results based on JVM98 benchmarks are introduced. Finally, a summary is made in Section 6.

2. RELATED WORK

As described in PicoJava II released by Sun Microsystems, bytecodes were classified into six types: LV, OP, BG1, BG2, MEM, and NF. Nine folding patterns were also predefined [1-4]. The Instruction Folding Unit (IFU) monitored the successive bytecodes to determine how many instructions were folded according to the folding patterns. N. Vijaykrishnan et al and L. R. Ton et al proposed similar folding algorithms by introducing different folding patterns [5][6]. Although these folding algorithms are simple and easily implemented, only the continuous bytecodes that exactly match the predefined folding patterns are folded. If the bytecode stream does not match the folding patterns, these bytecodes will be executed in serial. Thus the folding is inefficient.

L. C. Chang et al proposed the POC folding algorithm to improve folding efficiency in 1998, where bytecodes are classified into P (Producer), O (Operator), and C (Consumer) types according to the bytecode operation characteristics [7]. The O type bytecodes were further divided into four subtypes: O_B, O_C, and O_T. Recursive check was performed for every two consecutive instructions according to the POC folding rules. If the two instructions were foldable, they were marked with a new POC type, which was then checked with the following unfolded bytecode instructions until no folding was possible. The POC folding algorithm has no fixed folding patterns and can be implemented as finite automation through a state machine. Compared with the previous folding algorithms, although its folding efficiency is improved, similar to the previous folding algorithms, it is only efficient to fold the consecutive bytecode instructions. If a folding group is separated, the corresponding bytecodes will be issued sequentially.

Based on the POC folding algorithm, A. Kim and M. Chang introduced the advanced POC folding algorithm by adding additional four discontinuous folding sequence patterns to fold the discontinuous bytecode instructions [8][9]. Different from the original POC folding algorithm, the O type bytecodes were further divided into two subtypes: Oc (Consumable operator) and Op (Producible operator), according to their operation results being written back onto the operand stack or not. Thus the type decoder was simpler and easier to implement than the previous folding algorithms, because the bytecodes were only classified into four types. The advanced POC folding algorithm achieves higher folding efficiency with a relatively simple implementation circuitry. However, improper type definitions for some bytecodes exist [10][11]. For example, the bytecode lastore should be Oc type according to its operation behavior, but it is defined as C type in the advanced POC algorithm.

L. R. Ton et al. presented the Enhanced POC (EPOC) folding algorithm by using a stack reorder buffer to hold the extra P type bytecodes and the incomplete folding groups for further folding [10][11]. The bytecode types and their definitions were the same as those in the POC folding algorithm, and the incomplete folding groups were treated as P type. M.W. El-Kharashi et al developed an operand extraction-based algorithm by tagging the incomplete folding groups as tagged producers and tagged consumers, which were further used as producers or consumers in the following folding groups [12][13]. In the algorithm, bytecodes were classified into twelve types according to the way they handled the stack, and five folding pattern templates were defined. Although it claimed that 97% of stack operations and 50% of all operations were eliminated, the foldability check and bytecode type decoder are complicate to implement in hardware due to so many bytecode categories.

3. JHISC INSTRUCTION SET

In jHISC, the instruction set supports up to three operands. Each instruction is 32 bits in length with 8 bits for the opcode to define the instruction operations. The operands may be registers or 11-bit, 16-bit, and 24-bit immediate data. The current local variable frame is mapped into general-purpose registers and accessed with 5-bit index, therefore addressing up to 32 general-purpose registers. The instruction formats are shown in Fig. 1 to Fig. 6.

Seven groups of instructions are defined in jHISC. They are logical instructions, arithmetic instructions, branching instructions, array manipulation instructions, object-oriented instructions, data manipulation instructions and miscellaneous instructions. Excluding the instructions for floating-point operations, 94% of all bytecodes and 83% of the object-oriented related bytecodes are implemented in hardware directly [14][15]. Moreover, some quick instructions are provided to perform the operations of putting or getting variables after the first execution to speed up their operations.

4. THE PROPOSED INSTRUCTION FOLDING ALGORITHM

4.1 Bytecode categories

The Java virtual machine provides a rich set of instructions to manipulate the operand stack. We may classify bytecodes into several categories, but the more the bytecode categories are, the more complicate the type decoder in the instruction folder is. In the original POC folding algorithm and its extensions (EPOC and the advanced POC), the O (Operator) type was defined as the bytecodes which popped data from the operand stack and performed operations. Thus some bytecodes, such as getstatic, which perform operations without popping data from the operand stack, are not O type. However, in the original POC folding algorithm and its extensions, these bytecodes were defined as O type. To fix this problem, Tp and T type are introduced.
to define these type instructions according to their behaviors of handling the operand stack. The other bytecode types and their definitions are similar to those in the advanced POC algorithm. The type definitions are described as follows [14].

- **Producer (P)**: instructions that get data from constant registers or local variables and push them onto the operand stack, such as `iconst_1`, `iload_3`.

- **Operator (O)**: instructions that pop data from the top of the operand stack and perform operations. This type is further divided into two subtypes, namely **Producible Operator (O_P)**, such as `aload`, which pushes its operation result onto the operand stack, and **Consumable Operator (O_C)**, which does not push the operation result, such as `if_icmpeq`.

- **Consumer (C)**: instructions that remove data from the operand stack and store them back into the local variables, such as `istore`.

- **Termination (T)**: instructions that do not operate on the operand stack and some non-foldable bytecodes, such as `goto` and `return`. Such instructions contain table jump, multidimensional array creation, exception throw, monitor enter and exit. They are more suitable to be emulated by software traps because their execution procedure is complicated and needs the support of operation system.

- **Temporary (Tp)**: instructions that perform operations without popping data from the operand stack, but push the operation results onto it, such as `getstatic`.

The distribution of each category in the benchmark JVM98 is presented in Table 1. The P and C type bytecodes compose 49% of all operations. Thus how to handle the P and C type bytecodes has great impact on the system performance.

### 4.2 Folding algorithm

#### 4.2.1 Folding rules

Different from the other folding algorithms, the folding rules are simply summarized and shown as follows.

1. P type bytecodes are folded into the following adjacent C or O type bytecodes.
2. C type bytecodes are folded into the previous adjacent P, Op or Tp type bytecodes.
3. T type bytecodes cannot be folded.
The Instruction Classifier classifies bytecodes according to their bytecode types and opcodes and type definitions. The bytecode types, opcodes, operand types, the constant values and local variable indices are stored in the different buffers. The Folding Manager Unit checks the foldability and identifies the central instructions according to the bytecode types and folding rules. If some bytecodes can be folded, the Folding Manager Unit will generate the relevant jHISC opcode, a foldable signal and a folding length signal. The Opcode is determined by the central instruction and the related operand type. The foldable signal is used to trigger the jHISC instruction, the central instruction determines the opcode and type definitions. The bytecode types, opcodes, and operand types form a complete folding group. If an Op or Tp type bytecode is not followed by a C type bytecode, the C type bytecode, Op or Tp type bytecode, and the previous P type bytecode(s) form a complete folding group. If an Op or Tp type bytecode is not followed by a C type bytecode, it and its preceding P type bytecode(s) constructs an incomplete folding group, where the corresponding information on the operation result is written into buffers for further folding as a P type bytecode.

In every folding group, there is a central instruction, which operates on the operand stack and modifies its contents. The central instruction, the necessary producer and consumer instructions form a folding group. Typically, each folding group only has a central instruction, which may be an O or T type instruction. A consumer bytecode can also be a central instruction when it follows a producer instruction directly. In the generated jHISC instruction, the central instruction determines the opcode while the related producer and consumer instructions affect the operands.

### 4.2.2 System diagram

The block diagram of instruction folding and translation unit is illustrated in Fig. 7. Bytecodes are fetched from the instruction cache or memory and stored into the Instruction Buffer. The Instruction Classifier classifies bytecodes according to their opcodes and type definitions. The bytecode types, opcodes, operand types, the constant values and local variable indices are stored in the different buffers. The Folding Manager Unit checks the foldability and identifies the central instructions according to the bytecode types and folding rules. If some bytecodes can be folded, the Folding Manager Unit will generate the relevant jHISC opcode, a foldable signal and a folding length signal. The jHISC opcode is determined by the central instruction and the related operand type. The foldable signal is used to trigger the jHISC instruction. The folding length signal is used to update the pointer of the buffers. If a bytecode is not folded, it will be simply translated to a jHISC instruction in sequence.

- **Type_buffer**
  Type_buffer stores the bytecode types and each entry is 3 bits.

- **Ibuffer**

The Ibuffer stores the opcodes of bytecodes. Each entry is 8-bit. For an incomplete folding, the corresponding opcode equals to -1 in the Ibuffer.

- **Flag_buffer**

The constant indicator is stored in the Flag_buffer. If a bytecode pushes constant values onto the operand stack, such as `iconst_0`, the value 1 will be written into the Flag_buffer. If a bytecode operates on the local variable, the value 0 will be written to the Flag_buffer.

- **Data_type_buffer**

The operated data types are stored into the Data_type_buffer. In the Java virtual machine, operand types contain int, byte, signed 16-bit byte, char, floating-point, long, double and reference.

- **Vbuffer**

In jHISC, the constant registers and local variable frames are implemented by the register file. The Vbuffer holds the local variable indices or constant values. When the operated data type is long or double, it will take two continuous registers to hold the datum.

### 4.2.3 Foldability detection

The Folding Manager Unit firstly performs foldability detection according to the bytecode types and folding rules. The detection procedure is presented in Fig. 8.

Once a bytecode is fetched, the type of its previous bytecode is checked to see whether there is an Op or Tp type bytecode. If such bytecode exists and the current fetched bytecode is a C type bytecode, the C type bytecode, its precede Op or Tp type bytecode, and the required P type bytecode(s) or intermediate result(s), will form a folding group, which will be converted to a jHISC instruction. If such bytecode exists and the current fetched bytecode is not a C type bytecode, only the precede Op or Tp type bytecode and its required P type bytecode(s), will form a folding group and be folded. The operation result is treated as a P type bytecode and the related information is held by buffers.

When the Op, Tp and P type bytecodes are fetched, their related information, such as opcodes, operated data types, local variable indices or constant values, and so on, is pushed onto the buffers directly. The Op and Tp type bytecodes need to see whether the next bytecode is a C type bytecode or not. If it is, it

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>Tp</th>
<th>Oc</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>39.68%</td>
<td>9.08%</td>
<td>2.17%</td>
<td>0.27%</td>
<td>13.76%</td>
<td>35.05%</td>
</tr>
<tr>
<td>_202_jess</td>
<td>43.08%</td>
<td>5.66%</td>
<td>3.51%</td>
<td>1.38%</td>
<td>24.22%</td>
<td>22.15%</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>39.17%</td>
<td>1.73%</td>
<td>1.43%</td>
<td>0.38%</td>
<td>31.91%</td>
<td>25.39%</td>
</tr>
<tr>
<td>_209_db</td>
<td>44.36%</td>
<td>11.20%</td>
<td>4.72%</td>
<td>0.08%</td>
<td>15.56%</td>
<td>24.07%</td>
</tr>
<tr>
<td>_222_mpegudio</td>
<td>45.56%</td>
<td>6.27%</td>
<td>2.41%</td>
<td>1.86%</td>
<td>8.33%</td>
<td>35.57%</td>
</tr>
<tr>
<td>228_jack</td>
<td>42.85%</td>
<td>5.43%</td>
<td>5.81%</td>
<td>1.24%</td>
<td>24.91%</td>
<td>19.77%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>42.45%</td>
<td>6.56%</td>
<td>3.34%</td>
<td>0.87%</td>
<td>19.78%</td>
<td>27.00%</td>
</tr>
</tbody>
</table>

Table 1 Distribution of bytecode categories.
will be folded together; otherwise, the OP and TP type bytecodes are folded incompletely. When a C type bytecode is fetched, if its immediately previous bytecode is an OP or TP bytecode, it will be folded into the OP or TP bytecode; otherwise, it will be folded with the directly preceded P type bytecode. The OC type bytecodes will be folded with their required P type bytecodes directly while the T type bytecodes are simply translated into jHISC instructions.

4.2.4 Instruction folding

In jHISC, the constant registers and local variable frames are implemented by the register file. Typically, local variables are mapped into the register file with the same index and all bytecodes can be translated into jHISC instructions by one to one if no folding occurs. For example, if a bytecodes stream is “iload_2, iload_1, iload_3, iadd, istore_1, iload_4, isub, istore_3, return”, their types will be “P, P, P, O P, C, P, O P, C, T”. The one-to-one mapping results are shown in Table 2.

In the table, registers Rb, Rc and Rd are temporary registers allocated by the register file control engine. During translation, the first three P type bytecodes iload_2, iload_1, and iload_3 are pushed onto the instruction buffer. When the bytecode iadd is fetched, it is marked as a central instruction and pushed onto the instruction buffer. Once the bytecode iadd is fetched, the Folding Manager Unit finds a C type bytecode directly follows an OP type bytecode, and then generates the signals: foldable signal and folding length. The bytecodes iload_1, iload_3, iadd, and istore_1 form a folding group and are translated to a jHISC instruction arith.add R1 R1 R3. Then the bytecodes iload 4 and isub are pushed onto the instruction buffer and isub is marked as a central instruction. When the next bytecode istore_3 is fetched, the Folding Manager Unit identifies it as a C type bytecode directly following an OP type bytecode. Thus the bytecode istore_3 forms a folding group together with the three preceding bytecodes in the instruction buffer: iload_2, iload_4 and isub. The folding group is converted into a jHISC instruction arithm.sub R3 R2 R4. Finally, the bytecode return is T type and is translated into the jHISC instruction rvk directly. The folding results are presented in Table 3 and the folding procedure is shown in Fig. 9.

From Table 3, we observe that the instruction length is reduced from 9 to 3 after folding and the temporary registers are not needed. In Fig. 9, the first four bytecodes and their corresponding information are pushed onto the buffers (shown in (a)). The current pointers of buffers are incremented by 4, and the bytecode iadd is marked as a central instruction. When the fifth bytecode istore_1 is fetched, it forms a folding group with
Table 3 Folding results.

<table>
<thead>
<tr>
<th>Original bytecode</th>
<th>Folding group</th>
<th>jHISC instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_2</td>
<td>group 1: iload_1, iload_3, iadd, istore_1</td>
<td>arith.add R1 R1 R3</td>
</tr>
<tr>
<td></td>
<td>group 2: iload_2, iload_4, isub, istore_3</td>
<td>arith.sub R3 R4 R2</td>
</tr>
<tr>
<td></td>
<td>group 3: return</td>
<td>rvk</td>
</tr>
</tbody>
</table>

its previous three bytecodes and the folding group is translated to the jHISC instruction arith.add R1 R1 R3 (shown in (b)). The current pointers of buffers are decremented by 3 along with the folding. Then the sixth and seventh bytecodes and their related information are pushed onto the buffers (shown in (c)). The current pointers of buffers are therefore incremented by 2. When the eighth bytecode istore_3 is fetched, it forms a folding group with its previous three bytecodes and the folding group is converted into the jHISC instruction arith.sub R3 R4 R2 (shown in (d)). The current pointer of buffers is thereby decremented by 3. Finally, the ninth bytecode return is fetched and translated directly because it is T type.

5. PERFORMANCE ESTIMATION

The proposed folding algorithm was evaluated based on the JVM98 benchmark trace analysis, which was a Java benchmark suit released by the Standard Performance Evaluation Corporation (SPEC) [16]. The evaluation platform was a PC system with an Intel Pentium 2.8 GHz processor and 1 GB RAM. The operating system was Windows XP with Service Pack 2 and the Java virtual machine was JDK 1.5.0_05 [18] with interpretation mode. JVMTI profiler [19] was used to implement the proposed fold-
ing algorithm, trace bytecode behavior at run-time, and dump the executed bytecodes, the generated jHISC instructions, and some other results. The analysis in this Section is based on these dumped results.

5.1 Folded P and C type bytecodes

Fig. 10 shows the percentages of the folded P and C type bytecodes relative to all operations and stack operations. With respect
Figure 10 Folded P and C type bytecodes.

Figure 11 Added data move operations.

Figure 12 Generated jHISC instructions.
to all stack operations, the percentage of the eliminated P and C type instructions varies from 87% to 98% and the average is about 93%. However, the Sun’s folding algorithm in PicoJava II folded up to 60% of all stack operations [2-4] [13]. The POC folding algorithm with 4-foldable strategy reduced up to 84% of all stack operations [7]. And the advanced POC folding algorithm claimed to eliminate about 93% of all stack operations in case that the array load and store operations were mistaken to be treated as P and C type operations, respectively. Thus its actual folding ratio is smaller than 93%. With respect to all operations, the percentages of the eliminated P and C type instructions are from 42.2% to 51.8% and the average is 47% in the proposed folding algorithm.

5.2 Added data move operations

During instruction folding or translation, some data move operations are added because the two operated data may be 16-bit immediate values in Java bytecodes and the jHISC instruction is only 32 bits in length. For example, if two 16-bit immediate data precede a bytecode \texttt{iadd}, during folding, one immediate datum needs to firstly move to a register through a data move operation, and then the corresponding bytecodes are translated into the jHISC instruction \texttt{arith.addi}. The percentage of the added operations relative to all operations is shown in Fig. 11, which varies from 0.3% to 7.4%.

5.3 Generated jHISC instructions

Fig. 12 shows the percentage of the generated jHISC instructions relative to all operations. The number of the generated jHISC instructions is much smaller than the original bytecodes. The minimum number of the generated jHISC instructions is about 50% of the original operations in the benchmark program _222_mpegaudio while the maximum occurs in the benchmark program _202_jess, which is about 63.7% of the original operations. This indicates that our proposed algorithm is useful and effective.

5.4 Intermediate operations

As mentioned before, when the instruction folder meets an incomplete folding group, the intermediate operation results need to hold for further folding to improve the folding efficiency. Fig. 13 shows the percentage of the intermediate operation results of incomplete folding groups relative to all operations. The percentage ranges between 14% and 32% with an average around 26%. The minimum occurs in the benchmark program _209_db while the maximum is in the program _222_mpegaudio. The reason is that the operations in the program _222_mpegaudio are more complicated than those in the other benchmark programs. Fig. 13 also shows that plenty of folding groups are issued incompletely. Thus the intermediate operation results have a great impact on the folding efficiency if they are not held by buffers for further folding and issued directly.

5.5 Performance enhancement

The reduced instructions can be calculated through the following equation.

\[ N = N_{total} - N_{jHISC} \]

Where \( N_{total} \) is the number of bytecodes, \( N_{jHISC} \) denotes the number of the generated jHISC instructions.

The percentage of the reduced instructions over all operations is obtained and shown in Fig. 14. The worst folding efficiency is in the benchmark program _202_jess, where the reduced instructions are about 37% of all operations. The best folding efficiency appears in the benchmark program _222_mpegaudio, where more than 50% of all operations are reduced. And averagely, about 44% of all operations can be reduced in the benchmarks, which means that the actually executed instructions are only 56% of the original bytecode instructions before folding.

6. CONCLUSION

More and more complex Java programs are applied in embedded devices, such as complicate games, network application pro-
grams. They require the processors in embedded devices to have good performance to run Java programs. To address this, a new instruction folding algorithm is presented to improve the performance of a Java processor. In this folding algorithm, all bytecodes are classified into five types according to their behaviors of handling the operand stack, and the intermediate operation results are written into buffers and treated as P type bytecodes for further folding.

With respect to all stack operations, the percentage of eliminated P and C type instructions varies from 87% to 98% and the average is about 93%. With respect to all operations, the percentage of eliminated P and C type instructions is from 42.2% to 51.8% and the average is 47%. As for folding efficiency, the worst occurs in the benchmark program _202_jess, where the reduced instructions are about 29% of the original operations. The best is in the benchmark program _222_mpegaudio, where more than 46% of the original operations are reduced. Averagely, about 44% of the original operations can be reduced in the benchmarks. Compared with other folding algorithms, the proposed algorithm has a great improvement on the system performance.

Acknowledgment

This work is partially supported by the City University of Hong Kong under Strategic Research Grant 7001847.

REFERENCES


