Instruction Folding in a Hardware-Translation Based Java Virtual Machine

Hitoshi Oi
Department of Computer Science
The University of Aizu
Aizu-Wakamatsu, JAPAN

hitoshi at u-aizu.ac.jp

ABSTRACT

Bytecode hardware-translation improves the performance of a Java Virtual Machine (JVM) with small hardware resource and complexity overhead. Instruction folding is a technique to further improve the performance of a JVM by reducing the redundancy in the stack-based instruction execution. However, the variable instruction length of the Java bytecode makes the folding logic complex. In this paper, we propose a folding scheme with reduced hardware complexity and evaluate its performance. For seven benchmark cases, the proposed scheme folded 6.6% to 37.1% of the bytecodes which correspond to 84.2% to 102% of the PicoJava-II's performance.

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES; B.8.2 [**Hardware**]: PERFORMANCE AND RELIABILITY—*Performance Analysis and Design Aids*

General Terms

Performance, Design

Keywords

Java Virtual Machine, Hardware-Translation, Performance Evaluation, Instruction Folding.

1. INTRODUCTION

In this section, we present an introduction to the hardware-translation based Java Virtual Machine and the instruction folding.

1.1 Hardware-Translation of Java Bytecode

Hardware-translation is a technique to enhance the performance of the Java Virtual Machine (JVM) [1] by dynamically replacing the bytecodes to native machine instructions [1]. A small translation logic is inserted between the fetch and decode stages of the processor pipeline. When a flag in the processor's status register indicates that the fetched instruction is a Java bytecode, it is converted into native instructions by the translation unit. If the native instruction is fetched, it bypasses the translation logic. In theory, the decode and later stages of the processor pipeline do not see the difference between the native and Java bytecode execution modes which implies that the changes to the processor core is kept minimum.

Table 1 shows an example of the bytecode translation. In this example, two local variables which are assigned local variable indicies 3 and 4, are added and the result is written to the local variable 3. First two bytecodes, ILOAD_3 and ILOAD_4 push the values of two local variables onto the stack. Following the ARM Jazelle's specification, R0 to R3 are used to hold the top four words of the operand stack in this example [2]. Therefore, the first two bytecodes are translated into two load word instructions (LDR) using R7 which holds the address of the local variable 0 and corresponding offsets. Next bytecode, IADD, pops and adds two top of stack words and pushes the result onto the stack. This bytecode is translated into a native instruction which adds two registers R0 and R1. The last bytecode, ISTORE_3 pops the top of stack word and writes it to the local variable 3. This bytecode is replaced with a store word instruction (STR).

As shown in the above example, the translation unit reads a single bytecode at a time and generates a short sequence of native machine instruction(s). The hardware-translation is limited to the simple 140 bytecodes such as load, store, and arithmetic/logical operations on the stack [4]. Complex bytecodes, such as **new** (create a new object), are emulated

CF'06, May 3–5, 2006, Ischia, Italy.

---

[1] In the court order [2] dated September 30, 2003, ARM's Jazelle and Nazomi's U.S. Patent No. 6,332,215 are distinguished as follows. While Nazomi's patent translates Java bytecodes into native instructions before reaching the decode stage of the CPU, ARM's Jazelle translates bytecodes into controls signals. While this difference may be important for the patent issues, it is not essential for the ideas discussed in this paper. Therefore, the readers of this paper can interchangeably read "native instructions" as "the sequence of control signals corresponding to the native instructions", vice versa.

[2] Since the number of registers assigned to hold operand stack entries is fixed to four, they are considered to be used as a circular buffer with a modulo 4 pointer to R0 to R3. If more than four items are pushed onto the stack, spill and restore operations are required. For the sake of simplicity, however, these operations are omitted in this example.

Table 1: An Example of Bytecode Hardware-Translation

Bytecode	ARM Instruction
ILOAD_3	LDR R0, [R7, #12]
ILOAD_4	LDR R1 [R7, #16]
IADD	ADD R0, R1
ISTORE_3	STR R0, [R7, #12]

by the software. By limiting the complexity of the translation mechanism, the hardware resource overhead and the performance gain are balanced: in the case of Jazelle, it is reported that 8x performance gain was achieved by 12K gates, while typical dedicated or co-processors are around 20-25K [4].

1.2 Instruction Folding of Java Bytecode

As shown in the bytecode sequence in Table 1, there is inherent redundancy in the Java bytecode which comes from its stack architecture. In the above example, it took four bytecodes to add two variables and write the result back to one of them. Almost all microprocessors can do an *equivalent* operation with a single instruction, such as `ADD A, B`. This technique of merging multiple bytecodes into a single instruction is called the instruction folding and can be found in Java processors [7]. However, there are two issues when applying the instruction folding scheme to the hardware-translation based JVMs. First, most embedded microprocessors, which are the target platform of the hardware-translation JVM, are RISC architectures. This implies that arithmetic and logic operations cannot take memory locations as operands. If the part of operand stack for the local variables is allocated on the main memory (which is likely as shown in the example in Table 1), the instruction folding is not possible. Previously, we proposed to add a small register file to the datapath of the JVM to reduce the number of memory accesses caused by the local variables [6]. This extra register file (called local variable cache in [6]) also makes instruction folding possible on the hardware-translation based JVMs.

Another issue is the hardware complexity of the logic circuit that detects foldable bytecode sequences. In Sun's PicoJava-II, up to four bytecodes are folded into a single microprocessor operation, meaning four bytecodes are decoded simultaneously. Compared to the single bytecode decoding policy of the hardware translation, decoding four bytecodes for folding detection may be too complex. Moreover, the variable length of Java bytecode adds more complexity to the detection of the foldable sequences. The first bytecode opcode is pointed to by the program counter (PC), but the next bytecode could be either at $PC + 1$, 2 or even at $PC + 3$ depending on the first bytecode (excluding nonfoldable bytecodes). This means that to obtain the i-th opcode, where $i = 2 \cdots 4$, we have to decode $1 \cdots i - 1$ bytecodes beforehand.

In this paper, we present an instruction folding mechanism that provides similar performance to that of PicoJava-II with a reduced hardware complexity. The proposed scheme is evaluated by bytecode level simulations and analysis of bytecode sequence patterns that contribute to instruction folding are presented.

The rest of this paper is organized as follows. In the next section, an overview of PicoJava-II's folding scheme is presented. In Section 3, we propose an instruction folding scheme that alleviates the hardware complexity of PicoJava-II's scheme. In Section 4, the experimental environment including Java Virtual Machine and benchmark programs are first described, and then the proposed schemed is evaluated by comparing it to the performance of the PicoJava-II through simulations. Related work and conclusions are presented in Sections 5 and 6, respectively.

2. PICOJAVA-II'S INSTRUCTION FOLDING SCHEME

In this section, the instruction folding scheme of Sun's PicoJava-II and its source of hardware complexity are described. In PicoJava-II, Java bytecodes are classified into six types [7]:

LV: A local variable load or load from global register or push constant (e. g. ILOAD)

OP: An operation that uses the top two entries of stack and that produces a one-word result (IADD)

BG2: An operation that uses the top two entries of the stack and breaks the group (IF_ICMPEQ)

BG1: An operation that uses only the topmost entry of the stack and breaks the group (IFEQ)

MEM: A local vars store, global register store, and memory load (ISTORE)

NF: A nonfoldable instruction (GOTO)

Based on this classification, the following nine bytecode patterns (groups) are defined:

Group 1 LV LV OP MEM

Group 2 LV LV OP

Group 3 LV LV BG2

Group 4 LV OP MEM

Group 5 LV BG2

Group 6 LV BG1

Group 7 LV OP

Group 8 LV MEM

Group 9 OP MEM

Fig. 1 shows the block diagram of the PicoJava-II's foldable sequence detection circuit. Note that this diagram is reproduced from [7] with the following changes. First, while PicoJava-II has extended two-byte long instructions, they are specific to PicoJava-II's implementation and unrelated to other JVMs in general, including ours. Therefore, the folding type decoders handle only single byte bytecodes. Second, in PicoJava-II, each instruction byte is associated with the length of the bytecode by assuming that the byte is the opcode. This instruction length is decoded in the instruction cache which is in the fetch stage of the pipeline. In this paper, we assume that the hardware translation module

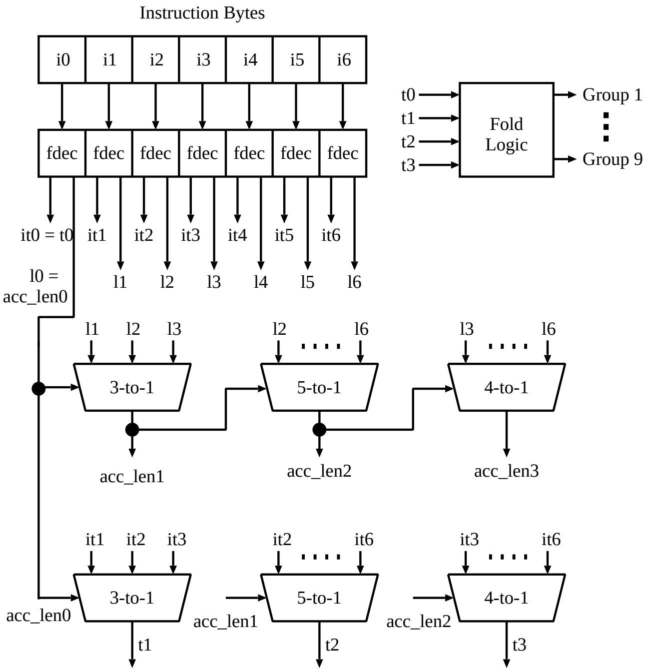

Figure 1: PicoJava-II's Foldable Bytecode Detection Logic

of the Java Bytecode is inserted between the fetch and decode stages and try to minimize the changes to the processor core. Therefore, unlike PicoJava-II, decoding of the instruction length is also performed in the hardware translation module.

In the instruction buffer, there are seven entries and each entry consists of an instruction byte (i0 to i6). All instruction bytes are speculatively decoded by folding type decoders (fdec) and generate instruction types (it0 to it6) and instruction lengths (l0 to l6). The first byte in the buffer (i0) is always the opcode of the first bytecode (b0). Therefore, it0 is actually the folding type of the first bytecode (t0).

Since the length of foldable bytecodes varies from one (e. g. IADD) to three (e. g. SIPUSH), the length of the first bytecode (l0) is fed to a 3-to-1 mux to select the folding type of the second bytecode (t1) from it1 to it3. The length of the first bytecode (l0) is also used to select the accumulated length of the first and the second bytecodes (acc_len1) from l1 to l3, which in turn selects the folding type of the third bytecode (t2) from it2 to it6 as well as to select the accumulated length of the first through third bytecodes (acc_len2).

Similarly, acc_len2 selects the folding type of the fourth bytecode (t3) from it3 to it6 and the accumulated length of the first through fourth bytecodes (acc_len3).

The folding detection logic takes folding types of all four bytecodes (t0 to t3) and enables one of nine output (Group1 to Group9) if any foldable sequences are detected. Needless to say, since a longer sequence has a priority, for example, for an LV LV OP MEM sequence, only Group1 output is enabled (i. e. Group2 is disabled).

Note that, the length of the first bytecode (l0) is propagated through three multiplexers to determine the folding type of the fourth bytecode (t3). t3 is then fed into the folding detection logic and then finally a possible folding pattern is determined.

It has been pointed out that the instruction folding unit (IFU) can be a critical path in the decode stage of the PicoJava-II processor pipeline [8]. In addition, if we remember that the hardware-translation based JVM translates one bytecode at a time, the hardware resource and complexity overhead of PicoJava-II's instruction folding scheme may not be suitable for incorporation without modification.

3. THREE BYTECODE FOLDING SCHEME WITH REDUCED COMPLEXITY

In this section, we propose an instruction folding scheme that takes up to three bytecodes with reduced hardware complexity and still provides a similar performance as PicoJava-II. The primary source of complexity in the PicoJava-II's folding mechanism is the variable length of the bytecode, especially, the length of the LV type bytecodes that varies from one to three bytes. To reduce this complexity, we modified the PicoJava-II's scheme in the following two points. First, we limit the number of folding bytecodes to three (i. e. Group1 is excluded) Next, we exclude SIPUSH, which is the only three byte long LV type bytecode and handle it as an NF bytecode.

As we will see in the next section, in general, the fraction of Group 1 sequence (LV LV OP MEM) is small and the instruction count of SIPUSH is also small compared to other LV bytecodes. The lengths of MEM, BG1 and BG2 are also variable. However, these bytecodes are always at the end of the foldable sequence and hence do not affect the position of the opcodes of other bytecodes in a foldable sequence.

Fig. 2 shows the block diagram of the proposed folding scheme. While the instruction buffer stores seven instruction bytes, it only (speculatively) decodes the folding types (it0 to it4) and lengths (l0 to l4) of the first five bytes (i0 to i4) since the opcode of the third bytecode in foldable sequences does not go beyond i4. The folding type of the second bytecode (t1) is selected from either it1 or it2 based on the length of the first bytecode (acc_len0) which is actually l0. The accumulated instruction lengths up to second and third bytecodes (acc_len1 and acc_len12, respectively) are obtained in the same manner as in the PicoJava-II with fewer candidates.

The hardware complexity of the proposed mechanism is reduced in the following points. First, since we have dropped Group 1, the length of the multiplexer chain to obtain the accumulated instruction lengths has reduced from three to two. Second, the sizes of the MUXes for t1 and t2 as well as acc_len1 and acc_len2 have changed from 3-to-1 and 5-to-1 to 2-to-1 and 3-to-1, respectively. Moreover, the number of folding type decoders (fdec) has been reduced from seven to five.

Block diagrams in Figs. 1 and 2 do not include the circuit which handle the cases where the bytecode sequence pattern is foldable but its length exceeds the size of the instruction buffer. In such a case, the folding type of the corresponding bytecode must be changed to NF regardless of its predecoded folding type. The proposed scheme is also simpler than PicoJava-II in this part because only acc_len0 = 1 or 2 and acc_len1 = 2 to 4 are valid for the folding type signals t1 and t2, respectively.

A complete and precise estimation of the hardware over-

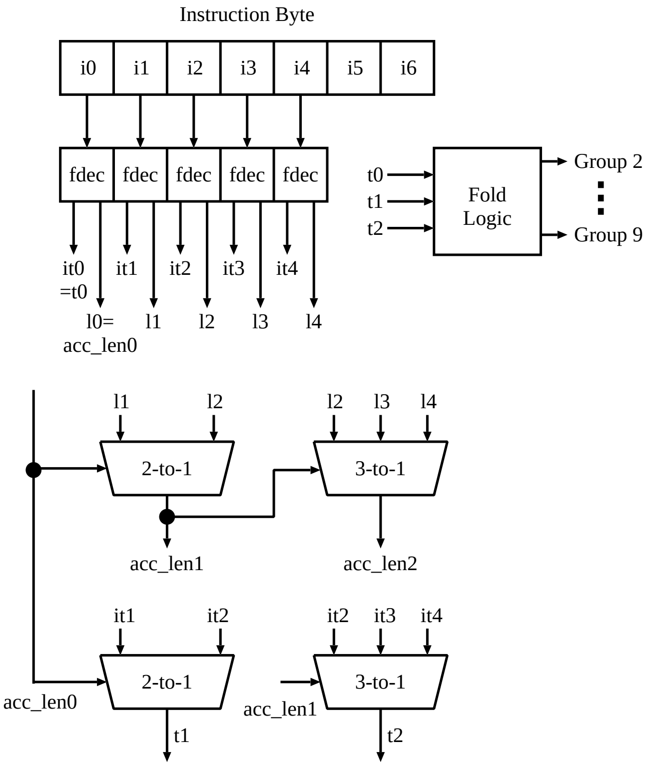

Figure 2: Proposed Foldable Bytecode Detection Logic

head and speed cannot be obtained without (at least) the datapath that is controlled by the folding pattern signals detected in the pattern detection module. At present, we do not have such a complete model of a hardware-translation based JVM and also the emphasis of this paper is placed on the optimization of the folding patterns. However, as a metrics of the hardware complexity and operation speed, we wrote Verilog models for the circuits in Figs. 1 and 2 and synthesized them under the 0.35μ rule. In PicoJava-II's model (Fig. 1), the delay of the longest path (from i0 to Group 1) was 2.82ns. On the other hand, in the proposed scheme (Fig. 2), the delay of the longest path (from i0 to Group 2) was 2.50ns, which is a reduction of 11%. We also compared the logic circuit areas and it was found that the proposed scheme occupied 35% less area than that of PicoJava-II.

4. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed folding mechanism and compare it to that of PicoJava-II. We also use two bytecode version of PicoJava-II's mechanism which only works for groups 5 to 9 as another reference. First, we present the simulation environment including JVM and benchmark programs and the simulation results follow.

4.1 Experimental Environment

For the JVM and Java Runtime Environment, we use Kaffe version 1.0.7 [9]. Kaffe is an open-source implementation of the JVM and we compiled it with "–with-engine=intrp" option so that all bytecodes are interpreted. It is assumed that a 16-entry local variable cache is attached to the JVM which works as described in [6]. This size of the local variable cache can accommodate all local variables of DES, ECM and PNG, or is effectively large enough for SAXON with XSLTMark test case documents.

Note that, the LV type bytecodes can be further divided into two classes. The first class is those actually accessing local variables, such as ILOAD. If the local variable cache does not have a valid copy of the accessed local variable, it must be loaded from the memory. Therefore, for example, a Group7 sequence ILOAD_1, IADD, is effectively not folded if the local variable 1 is not present in the local variable cache. Another class of LV bytecodes is those not actually accessing local variables, such as ICONST_0. When such a bytecode appears in any foldable sequence, it is always folded.

The benchmark programs used for the evaluation are listed in Table 2. The first set of benchmark programs is SAXON Version 6.3, an XSLT processor [10], driven by four test case XML documents from XSLTMark [11]. We chose four test case documents, chart, decoy, encrypt and trend, based on the average number of bytecode executed for a method invocation and the functional categories defined in the XSLTMark.

The Embedded CaffeineMark consists of five tests, Sieve, Loop, Logic, Method and Float [12]. Each of these tests is basic and tries to measure various aspects of JVM. Composite results of all five tests are used.

DES is a DES based encryption and decryption of a text file using the Bouncy Castle Crypto Package [13]. A text file of 3KB is first encrypted and then decrypted using the sample program included in the Bouncy Castle Crypto package (`src/org/bouncycastle/crypto/examples/DESExample.java`).

PNG extracts properties of a PNG image (512×512 from [15]) such as pixel size and bit depth using com.sixlegs.png PNG decoder and its sample program `PropertiesExample.java` [14].

EEMBC's GrinderBench is getting a popularity as a benchmark for embedded Java 2 Micro Edition [5]. At this moment, we do not have access to GrinderBench and could not use it for the performance evaluation of the proposed folding mechanism. However, three out of five benchmark programs in GrinderBench (Crypto, kXML and Png) are of similar types of applications as the benchmarks used in this paper.

The fraction of each bytecode type and the average execution length for each benchmark program are presented in Tables 3 and 4, respectively. The average execution length is the number of contiguously executed bytecodes without interruption by invocation or return. The higher this number, the more chances of folding. The number in parentheses in the LV column indicates the fraction of three-byte LV bytecode (SIPUSH). Since this bytecode is handled as an NF in the proposed scheme, this number indicates the cases where PicoJava-II can fold instructions but the proposed scheme cannot.

4.2 Simulation Results

In this section, we present the results of simulations. Fig. 3 shows the breakdown of the folded bytecodes for SAXON with four XSLTMark test cases. Compared to other benchmark programs, the fractions of folded bytecodes for SAXON are small. Two reasons can be found in Tables 3 and 4. First, they have high fractions of non-foldable bytecode (NF)

Table 2: Benchmark Program Description

Bench-mark	Description
SAXON Version 6.0 with XSLTMark 1.2.0	
chart	Generates an HTML chart of some sales data (select, control).
decoy	Simple template with decoy patterns to distract the matching process (match).
encrypt	Performs a Rot-13 operation on all element names and text nodes (function).
trend	Computes trends in the input data (select, functions).
ECM	Embedded CaffeineMark (Sieve, Loop, Logic, Method and Float).
DES	DES encryption/decryption using the Bouncy Castle Crypto
PNG	Extract PNG image properties using com.sixlegs.png

Table 3: Benchmark Program Bytecode Analysis. The numbers in parentheses in the LV column are the fractions of three byte LV bytecode (i. e. SIPUSH).

Bench-mark	Bytecode Types (%)					
	LV	OP	BG1	BG2	MEM	NF
SAXON with XSLTMark						
chart	44.4 (0.7)	4.3	7.7	12.6	4.1	26.8
decoy	44.4 (0.2)	2.3	8.3	9.9	4.0	31.1
encrypt	42.5 (0.1)	4.0	7.4	14.0	3.3	28.8
trend	39.9 (0.1)	1.6	9.8	5.8	2.6	40.3
ECM	45.3 (0.0)	4.7	9.1	14.9	6.7	19.2
DES	43.8 (0.7)	24.9	1.6	9.8	9.4	10.5
PNG	42.8 (2.5)	11.0	3.8	13.3	2.9	26.3

ranging from 26.8% to 40.3%. Second, the average execution lengths are short, leading to fewer chances of folding. Among these four cases, the highest folding performance was archived in chart. With the proposed scheme, 17.6% of bytecodes were folded, which is 95% of PicoJava-II (18.5%) as shown in Table 5. While the difference is small, Groups 1, 7 and 8 are major contributors to it.

The performance of the proposed scheme is almost the same as PicoJava-II's for other three test cases (99.7, 99.6 and 102%). The proposed scheme cannot fold Group 1, but more Groups 2, 5, 6 and 9 (also Group 4 in trend) are folded in the proposed scheme. While the difference is quite small, the proposed scheme performed better than PicoJava-II on trend (6.6% v.s. 6.5%). Groups 2, 4, 5 and 6 are the sources of this better performance. Two-Bytecode version of PicoJava-II's scheme (PJ2B) performed significantly worse than the other two schemes (71.6% to 89.5% of PicoJava-II). In PJ2B, three and four byte sequences (Group 1 to 4) are partially detected as two byte sequences and folded. The increases resulting from partially folded sequences are mostly in Group 5 to 7. The number of NF bytecodes and the average execution length of decoy are lower than those of encrypt. However, all three folding schemes performed better on decoy than on encrypt.

Table 4: Average Execution Lengths of Benchmark Programs (The number of contiguously executed bytecodes without interruption by invocation or return).

Benchmark	chart	decoy	encrypt	trend
Length	11.6	8.8	10.8	4.9

Benchmark	ECM	DES	PNG
Length	90.6	66.1	24.3

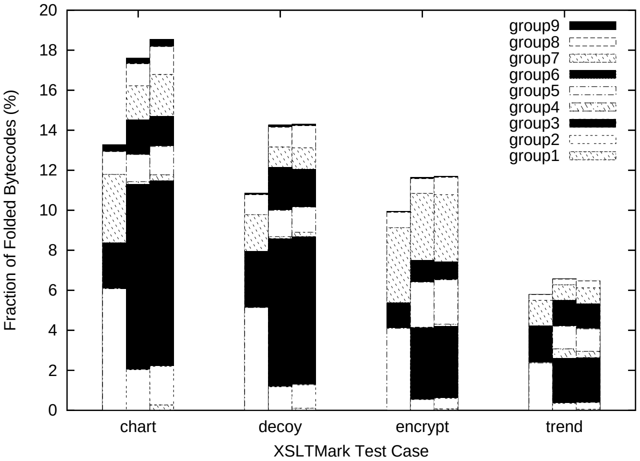

Figure 3: Breakdown of the Folded Bytecodes for SAXON with XSLTMark Test Cases. For each benchmark, three bars indicate Two-Bytecode version of PicoJava-II, proposed scheme, PicoJava-II's scheme (left to right).

Fig. 4 shows the breakdown of the folded bytecodes in the Embedded CaffeineMark, DES and PNG. The average execution length of Embedded CaffeineMark (ECM) is 90.6 bytecodes, which is the longest among the benchmark programs used in this paper. This long execution per method invocation led to high folding ratios (32.3% to 26.8%). One thing that should be noted is that, while 6.7% of bytecodes were of MEM type, we hardly see Group 4 in PicoJava-II and the proposed scheme. Instead, 5.8% of bytecodes were folded in Group 8 in all three schemes. Therefore, most of MEM bytecodes follow LVs: meaning either they copy one LV to another (e. g. ILOAD_1 and ISTORE_2) or initialize an LV by a constant (ICONST_2 and ISTORE_3). 1.6% of bytecodes were folding in Group 1 in PicoJava-II. However, when it is compared to the sums of Groups 1 and 2 in the proposed scheme, the difference shrinks to 0.5%. This implies that the increase of Group 2 in the proposed scheme effectively absorbed most of the Group 1 sequences. The proposed scheme achieved a relative performance of 98.5% of PicoJava-II's scheme. In PJ2B, we see that the fraction of Group 7 is significantly, higher than PicoJava-II and the proposed scheme (3.3% against 0.3%). This is possibly because the LV OP part of Groups 1 and 2 were detected and folded as Group 7 in PJ2B. The relative performance of PJ2B was 83.0%.

Table 5: Relative Performance of Two-Bytecode version of PicoJava-II (PJ2B) and the Proposed Scheme Normalized to PicoJava-II's Four-Bytecode Folding Scheme.

Folding	Benchmark Programs			
Scheme	chart	decoy	encrypt	trend
PJ2B	71.6	75.9	85.0	89.5
Proposed	95.0	99.7	99.6	102

Folding	Benchmark Programs		
Scheme	ECM	DES	PNG
PJ2B	83.0	67.4	81.5
Proposed	98.5	95.4	84.2

DES has a very long average execution length (next to ECM) and the lowest fraction of NF bytecodes (10.5%). It also has a very high fraction of OP bytecodes (24.9%), which leads to large numbers of folded bytecodes in Groups 2, 7 and 9. These two properties confirm computation-intensive nature of DES encryption/decryption algorithms. While the fraction of MEM bytecodes is the highest among the benchmarks used, the fraction of Group 8 foldings is quite low (0.1%). This means, unlike ECM, MEM bytecodes are used to store the results of OP bytecodes rather than initialization or copy of local variables. In DES, the fractions of three-byte LV bytecodes (SIPUSH) is relatively high (0.7%) and the fraction of Group 1 is the highest among the benchmark programs used. Since SIPUSH and Group 1 are excluded in the proposed scheme, these two properties are disadvantages for the proposed scheme. The fraction of Group 2 in the proposed scheme is not high enough to cover excluded Group 1 in PicoJava-II and also the fraction of Group 7 is slightly lower in the proposed scheme. The proposed scheme performed 95.4% of PicoJava-II's folding. In PJ2B, we see that the bar for Group 7 is much longer than in PicoJava-II or in the proposed scheme, but it is not long enough to cover Groups 1 to 4 that are missing in PJ2B. The relative performance of PJ2B is only 67.4% of the original four-byte folding scheme.

Since PNG's runtime behavior is closer to SAXON than ECM and DES, so is its folding performance. Its fraction of NF bytecode is 26.3% and the average execution length is only 24.3 bytecodes. As a results, even with the PicoJava-II's scheme, only 16.8% of bytecodes are folded. Also, 2.9% of bytecodes were folded as Group 1 in PicoJava-II and 2.5% of bytecodes were SIPUSH, both of which are disadvantages for the proposed scheme. The relative performance of the proposed scheme was 84.2% which was worst for all the benchmarks tested in this paper. With PJ2B, the fractions of Groups 5 to 8 increased significantly by partially folding the longer sequences (Groups 1 to 4). Its relative performance, 81.5%, was close to that of the proposed scheme.

In principle, it is possible that opcodes form a foldable sequence but the folding is not performed in the PicoJava-II. This is because, while the opcode of the last bytecode in a foldable sequence is present in the instruction buffer, its parameter is not. For example, the sequence of SIPUSH 0x100, ILOAD 4, IADD, ISTORE 5 is a Group 1 sequence and is eight byte long. While the opcode of the last bytecode ISTORE 5 is in the instruction buffer (and its folding type is decoded as MEM), its parameter (local variable index 5) is not. Therefore, this sequence cannot be folded as a Group 1. However, for the simulations in this paper, we did not see any instance of such "parameter overflow" in PicoJava-II, even for PNG which has the highest fraction of SIPUSH.

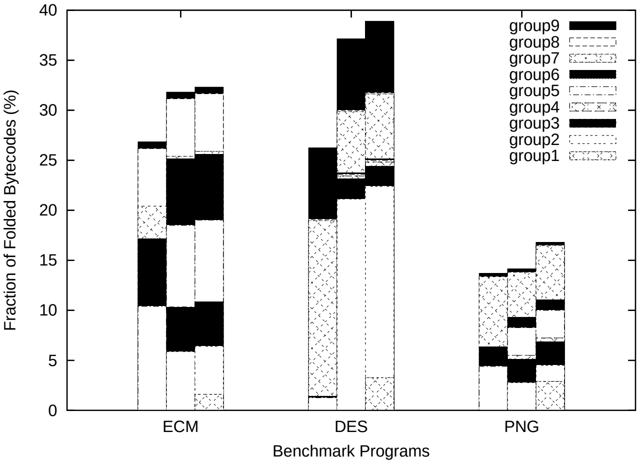

Figure 4: Embedded CaffeineMark, DES, PNG. For each benchmark, results of Two-Bytecode version of PicoJava-II, proposed scheme, PicoJava-II's scheme are shown from left to right

5. RELATED WORK

ARM's Jazelle and Nazomi's Jstar are commercial products that incorporate hardware-translation based JVM. In this paper, the base design of the hardware-translation based JVM assumed the information published in ARM's white paper [3]. However, the ideas presented in this paper do not depend on the features specific to ARM or Nazomi's architectures and should be applicable to most embedded RISC microprocessors.

PicoJava-II [7] directly executes Java bytecodes by the hardware. Since a pure JVM is not sufficient to build a real system, PicoJava's instruction set is extended for running applications written in "legacy" programming languages such as C/C++. Therefore, its design approach takes an opposite direction from the hardware-translation which tries to execute Java bytecodes by adding a small translation logic to the standard RISC type microprocessors. Our proposed instruction folding scheme is based on PicoJava-II.

Radhakrishnan et. al studied the microarchitecture of PicoJava-II and pointed out the instruction folding is the critical path of the processor pipeline [8]. To solve this issue, they proposed to move the instruction folding module from the decode stage of the pipeline to the instruction fill unit in the fetch stage. They also proposed to store the folded bytecodes in a dedicated cache (decoded bytecode cache) so that the folded bytecodes will be executed faster in the future.

Kim and Chang proposed a more aggressive folding mechanism which tried to find two or more foldable instruction sequences in which one breaks the sequence of the others [16]. The emphasis was placed on how to find such multiple sequences in the instruction stream and they did not work on how to fold each basic sequence (such as LV LV OP MEM). Moreover, since their scheme detects the foldable sequences by a state machine, it can be only used for ahead-of-time folding (and used later by storing it in a decoded bytecode cache as in [8]). Otherwise, if it is implemented by a combination circuit, the size of the detection logic and instruction buffer will be larger than that for single sequence folding schemes.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an instruction folding scheme for a hardware-translation based JVM. One of the sources of hardware complexity in the instruction folding in the PicoJava-II is the variable length of the bytecode format. The proposed scheme alleviated this problem by removing the SIPUSH which is the only three-byte long LV bytecode in the folding type classification of PicoJava-II. We also excluded the four-byte code sequence (LV LV OP MEM, Group 1 in PicoJava-II) so that the number of bytecodes decoded simultaneously is reduced from four to three. The proposed scheme achieved 84.2% to 102% of the PicoJava-II's scheme for seven benchmarks. If we exclude PNG, which is the only case where the effects of SIPUSH and Group 1 removal are significant, the worst relative performance of the proposed scheme' jumps to 95.0%.

Currently, a Group 1 sequence (LV LV OP MEM) is partially folded as a Group 2 sequence (LV LV OP) in the proposed scheme. However, if the local variable accessed in the first LV is not present in the local variable cache, we should have more chances of folding by discarding the first LV and handle the (partial) sequence of LV OP MEM as Group 2. Note that, LV is a local variable load bytecode while MEM is a local variable store. Therefore, MEM bytecode does not require the local variable to be present in the local variable cache and hence its access is hit as long as the index of the variable is with in the range of the local variable cache. A possible improvement for the proposed scheme is to look up the status of the local variable cache before folding. With this scheme, we can expect the folding ratio to be increased at the cost of local variable cache look-up.

In this paper, we evaluated the effectiveness of the proposed instruction scheme by the fraction of folded bytecodes. Actually, foldable bytecodes are only one of three types of codes executed on a JVM, other two are non-foldable (that include bytecodes that cannot be hardware-translated and handled by software emulation) and native methods. We plant to develop a more complete model of hardware-translation based JVM so that we can evaluated the performance by total execution time (or, possibly by power consumption) of all three types of codes mentioned above.

Acknowledgment

This work is supported in part by The University of Aizu Competitive Research Funding (Grant Number P25). Yuichi Okuyama, from Computer Education Laboratory, The University of Aizu, contributed the area and delay estimation of the folding logic circuits in Section 3 The tools used in the estimation was provided by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. The author would like to thank the comments from the anonymous reviewers which significantly improved the quality of this paper.

7. REFERENCES

[1] Tim Lindholm and Frank Yellin, “The Java(TM) Virtual Machine Specification (2nd Edition)”, Addison-Wesley Professional, 1999

[2] ORDER GRANTING MOTION FOR PARTIAL SUMMARY JUDGMENT OF NON-INFRINGEMENT , Case No. C 02-02521-JF, UNITED STATES DISTRICT COURT, NORTHERN DISTRICT OF CALIFORNIA, SAN JOSE DIVISION, September 30, 2003.

[3] Steve Steel, “Accelerating to meet the challenges of embedded Java”, Whitepaper, ARM Limited, Nov 2001.

[4] “Jazelle - ARM Architecture Extensions for Java Applications”, Whitepaper, ARM Limited.

[5] GrinderBench, `http://www.grinderbench.com/`.

[6] Hitoshi Oi, “On the design of the local variable cache in a hardware translation-based java virtual machine”, in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp87–94 June 2005.

[7] “PicoJava-II Microarchitecture Guide”, Sun Microsystems, March 1999.

[8] Ramesh Radhakrishnan, Deependra Talla and Lizy K. John, “Allowing for ILP in an embedded Java processor” in *Proceedings of the 27th annual International Symposium on Computer architecture*, Vancouver, British Columbia, Canada pp294–305, 2000.

[9] Kaffe.org, `http://www.kaffe.org`.

[10] “SAXON The XSLT and XQuery Processor”, `http://saxon.sourceforge.net/` .

[11] “DataPower: XSLTMark XSLT Performance Benchmark”, `http://www.datapower.com/xmldev/xsltmark.html` .

[12] “The Embedded CaffeineMark”, Pendragon Software Corporation, 1997.

[13] “The Legion Of The Bouncy Castle”, `http://www.bouncycastle.org/`

[14] “PNG SOFTWARE”, `http://www.sixlegs.com/software/png/` .

[15] “The USC-SIPI Image Database”, `http://sipi.usc.edu/database/` .

[16] Austin Kim and Morris Chang, “Advanced POC Model-Based Java Instruction Folding Mechanism”, in *Proceedings of The 26th EUROMICRO Conference (EUROMICRO'00)*, Vol. I, No. 1 pp1332–1337, Sep. 2000.