Exploiting Java Bytecode Parallelism by Enhanced POC Folding Model

Lee-Ren Ton[1], Lung-Chung Chang[2], and Chung-Ping Chung[1]

[1] Department of Computer Science and Information Engineering
National Chiao Tung University
No. 1001, Dashiue Rd., Hsinchu, Taiwan 30056, ROC
{lrton, cpchung}@csie.nctu.edu.tw
[2] Computer & Communications Research Laboratories
Industrial Technology Research Institute
Building 51, No. 195-11, Sec. 4, Jungshing Rd., Judung Jen, Hsinchu, Taiwan 31041, ROC
lcchang@ccl.itri.org.tw

Abstract. Instruction-level parallelism of stack codes like Java is severely limited by accessing the operand stack sequentially. To resolve this problem in Java processor design, our earlier works have presented stack operations folding to reduce the number of push/pop operations in between the operand stack and the local variable. In those studies, Java bytecodes are classified into three major POC types. Statistical data indicates that the 4-foldable strategy of the POC folding model can eliminate 86% of push/pop operations. In this research note, we propose an Enhanced POC (EPOC) folding model to eliminate more than 99% of push/pop operations with an instruction buffer size of 8 bytes and the same 4-foldable strategy. The average issued instructions per cycle for a single pipelined architecture is further enhanced from 1.70 to 1.87.

1 Introduction

Internet has become the most feasible means of accessing information and performing electronic transactions. Java [1] is the most popular language used over the Internet owing to its portability, compact code size, object-oriented, multi-threaded nature, and write-once-run-anywhere characteristics.

The performance of the stack-based Java Virtual Machine (JVM) [2, 3] is limited by true data dependency. A means of avoiding such a limitation, i.e. stack operations folding, was proposed by Sun Microelectronics [4, 5, 6, 7] with folding capabilities of up to 2 and 4 bytecode. While executing, pre-defined and pre-stored folding patterns are compared with bytecodes in instruction stream sequentially. Consequently, we call this kind of folding as fixed-folding-pattern matching. Other researchers [8, 9, 10] have also proposed the folding method of this fixed-matching type.

In our earlier study, we proposed a dynamic-folding-pattern matching named POC folding model [11]. All bytecode instructions are classified into three major POC (Producer, Operator, and Consumer) types. Table 1 lists the POC types and the 'O' type is further divided into four sub-types according to their execution behavior.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 994-997, 2000.

Table 1. POC Instruction Types

POC	Description	Occurrence
P	An operation that pushes constant or loads variable from LV to OS	47.14%
O_E	An operation that will be executed in execution units	10.87%
O_B	An operation that conditionally branches or jumps to target address	11.54%
O_C	An operation that will be executed in micro-ROM or trap	22.19%
O_T	An operation that will force the folding check to be terminated	3.96%
C	An operation that pops the value from OS and stores it into LV	4.29%

In the POC folding model, foldability check is performed by examining each pair of consecutive instructions. By applying the POC folding rules, the two consecutive bytecode instructions may be combined into a new POC type, which is used in further foldability check with the following bytecode instructions. Consequently, the POC folding model is quite different from the fixed-matching one because there is no fixed-instruction-pattern. The POC folding rules can be represented as a state diagram shown in Fig. 1. If the Ps are not consumed immediately by O or C type instructions, they will be issued sequentially.

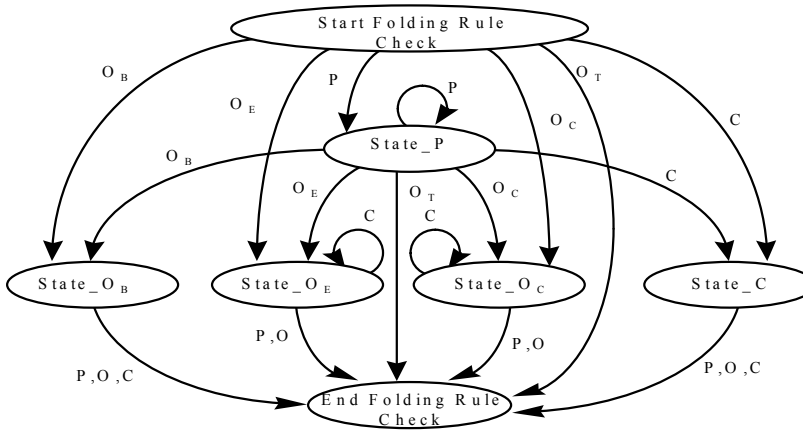

Fig. 1. Folding Rules for POC Model

2 Enhanced POC Folding Model

The main improvement of the EPOC model over the POC model is the capability of folding the discontinuous Ps. As shown in Fig. 2, the *P Counting* state will record how much Ps are there before the O or C type instructions. If there is no O or C type instruction in the instruction buffer, the Ps will be issued sequentially to the execution unit like the POC does. The *C Counting* state will check whether the preceding state is O_E state or *P Counting* state. If the preceding state is O_E, *C Counting* state will fold

the Cs into O_E. If it is *P Counting* state, then Ps are folded into Cs according to the number of Cs. Otherwise, if the C type instruction is the first instruction in instruction buffer, the EPOC issues the C sequentially.

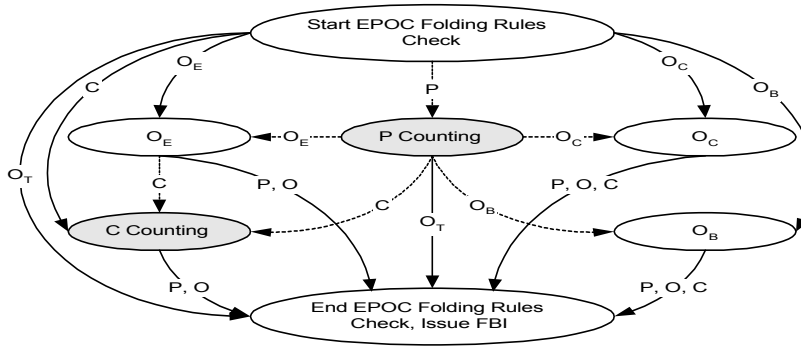

Fig. 2. Folding Rules for EPOC Folding Model

3 Performance Comparison of Various Folding Models

In Fig. 3, the average percentages of eliminated P + C type instructions for different foldability are shown. Note that the picoJava-II has the foldability of four according to the released specification. We duplicate the picoJava-II's performance results to each column for comparison only. The issued instructions per cycle (IIPC) for a single pipelined picoJava-II architecture for each model is shown in Fig. 4.

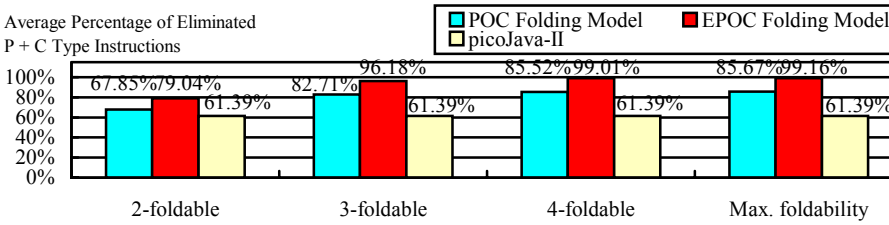

Fig. 3. Average Percentages of Eliminated P + C Type Instructions

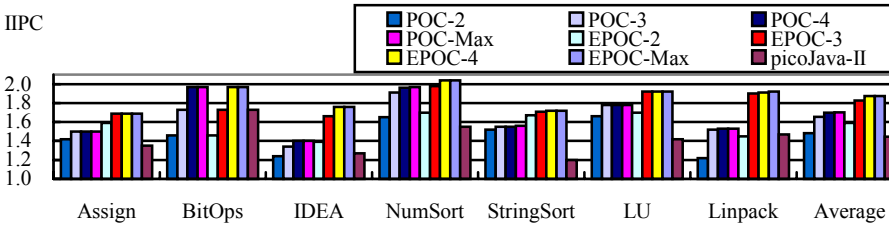

Fig. 4. Issued Instructions per Cycle for Each Folding Model

4 Conclusion

In this research note, we have proposed the EPOC folding model based on the previously proposed POC folding model. The dynamic-folding-pattern matching of POC and EPOC overrides the fixed-folding-pattern matching used in picoJava-II with the folding ratio of 39% and 61%, respectively.

The performance enhancement from POC to EPOC folding model benefits mainly from the foldability of discontinuous P type instructions, which results 85.5% and 99.0%.folding ratio using the 4-foldable strategy, respectively. That is, 44% and 50.9% program codes are folded, respectively.

The hardware implementation of the recursive EPOC folding model is integrated into decoding stage using parallel priority encoders to generate the source and destination fields of the FBI in constant delay time (non-recursive). Extra circuits must be added to maintain the instruction buffer after folding. For a superscalar Java processor of our current research, the EPOC folding model is integrated with the stack reorder buffer. Furthermore, the source-ready FBIs can be issued in parallel to exploit higher ILP.

References

1. James Gosling, Bill Joy and Guy Steele: The Java™ Language Specification, Addison-Wesley, Reading MA (1996)
2. Tim Lindholm and Frank Yellin: The Java™ Virtual Machine Specification, Addison-Wesley, Reading MA (1996)
3. Venners, B.: Inside the Java Virtual Machine,McGraw Hill, New York (1998)
4. M. O'Connor and M. Tremblay: picoJava-I: The Java Virtual Machine in Hardware, IEEE Micro, Vol. 17, No. 2, (1997) 45-53
5. H. McGhan and M. O'Connor,: picoJava: A Direct Execution Engine for Java Bytecode, IEEE Computer, (1998) 22-30
6. Sun Microsystems Inc.: picoJava-II™ Microarchitecture Guide, Sun Microsystems, CA USA (1999)
7. Sun Microelectronics: microJava™-701 Processor," http://www.sun.com/microelectronics/microJava-701/
8. Han-Min Tseng, et. al.: Performance Enhancement by Folding Strategies of a Java Processor, Proceedings of International Conference on Computer Systems Technology for Industrial Applications – Internet and Multimedia, (1997)
9. Lee-Ren Ton, et. al.: Instruction Folding in Java Processor, Proceedings of the International Conference on Parallel and Distributed Systems, (1997)
10. N. Vijaykrishnan, N. Ranganathan and R. Gadekarla: Object-Oriented Architectural Support for a Java Processor, Proceedings of the ECOOP'98, Lecture Notes in Computer Science, Springer Verlag, (1998)
11. L. C. Chang, L. R. Ton, M. F. Kao and C. P. Chung,: Stack Operations Folding in Java Processors, IEE Proceedings on Computer and Digital Techniques, Vol. 145, No. 5, (1998)