An Instruction Folding Solution to a Java Processor

Tan Yiyu [1], Anthony S. Fong [2], Yang Xiaojian[1]

[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

Abstract. Java is widely applied into embedded devices. Java programs are compiled into Java bytecodes, which are executed into 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 hold 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

As a result of its object-oriented feature 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 Palm PCs [1]. 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 out from the stack during execution. Thus some redundant load or store operations are performed, which results in the low execution efficiency and affects system performance, especially in embedded devices, where real-time operations and low power consumption are needed. 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 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. This execution procedure needs extra clock cycles to push or pop data from the operand stack. 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 system performance.

To address these shortcomings, Sun Microsystems introduced the notion of instruction folding [2][3], which was a technique to eliminate the unnecessary load or write-back operations to the stack by detecting some contiguous instructions and executing them collectively like a single, compound instruction. For example, to execute the bytecode stream mentioned above, the generated compound instruction may read data into ALU from the local variable directly, operate on them and write the operation result back to the local variable. Thus the intermediate operands and data do not need to push onto or pop 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, and system diagram. In Section 5, the performance estimation results based on JVM98 benchmarks are introduced. Finally, a summary is made in Section 6.

2. Related work

In Sun Microsystems solution, the bytecodes were classified into six types and nine folding patterns were predefined. 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 also proposed the similar folding algorithm to Sun Microsystems by introducing different folding patterns [4][5]. 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, the bytecodes will be executed in serial. Thus the folding is inefficient.

L. C. Chang proposed the POC folding algorithm [6] to improve folding efficiency, where bytecodes were classified into P (Producer), O (Operator), and C (Consumer) types according to the bytecode operation characteristics. The O type bytecodes were further divided into four subtypes: O_E, 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 checked 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. But like the previous folding algorithms, it is only used to fold the consecutive instructions.

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 [7]. Different with 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 were written back onto the operand stack or not. This algorithm achieves higher folding efficiency with a relatively simple implementation circuitry. However, improper type definitions for each bytecode exist [8]. For example, the bytecode *lastore* should be *Oc* type according to its operation behavior, but it is defined to be *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 [8]. The incomplete folding groups were treated as P types. M. W. El-Kharashi et al introduced 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 [9][10]. 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 by 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 operation. T The operands may be registers or 11-bit, 16-bit, and 24-bit immediate data. The current local variable frame is accessed with 5-bit index, therefore addressing up to 32 general-purpose registers.

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 [13]. Moreover, some quick instructions are provided to perform the operations of putting or getting variables after the first execution.

4. Proposed instruction folding algorithm

4.1 Bytecode categories

The Java virtual machine provides a rich set of instructions to manipulate the operand stack. 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 perform 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 later extensions, these bytecodes were defined as O type. In our proposed folding algorithm, they are defined as Tp or T type 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 presented as follows [11].

- **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 *iaload*, 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 local variables, such as *istore*.
- **Termination (T)**: instructions that do not operate on the stack and some non-foldable bytecodes, such as *goto* and *return*. Such instructions contain table jump, multidimensional array creation, exception throw, and monitor enter and exit. They are more suitable to be emulated by software traps.
- **Temporary (Tp)**: instructions that perform operations without popping data from the operand stack, but push the operation results onto it, such as *getstatic*.

4.2 Folding algorithm

4.2.1 Folding rules

The folding rules can be 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.
(4) The intermediate results of incomplete folding groups are written into buffers and treated as P type bytecodes for further folding. When an Op or Tp type bytecode is directly 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, the operation result is treated as a P type bytecode and the corresponding information is written into buffers for further folding.

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(s) and consumer instructions form a folding group. Typically, each folding group only has one central instruction, which may be an O or T_P type instruction. When a consumer bytecode follows a producer instruction directly, it can also be a central instruction. 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. 1. 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 the type definitions. The bytecode types, opcodes, operand types, the constant values and local variable indices are stored in different buffers, respectively. 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, foldable signal and 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 Operand Generator to generate the operands of jHISC instruction. The folding length signal is applied to update the pointer of the buffers. If a bytecode is not folded, it will be simply translated into a jHISC instruction in sequence.

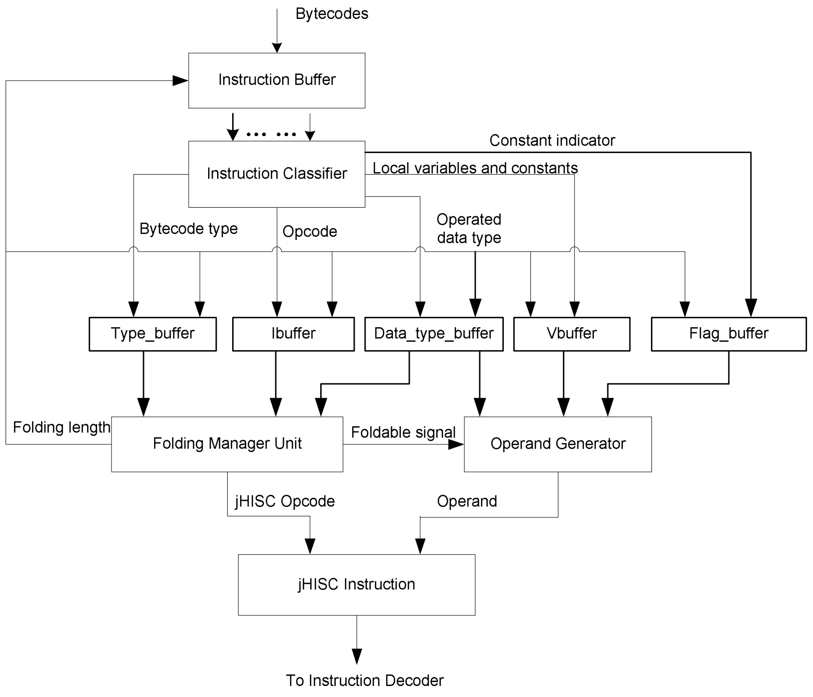

Fig. 1 Block diagram of instruction folder

4.2.3 Instruction folding

In jHISC, the constant registers and local variable frames are implemented by register files. Typically, local variables are mapped into register files with the same index and all bytecodes are 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 corresponding types will be "*P, P, P, O_P, C, P, O_P, C, T*". The one-to-one mapping results are shown in Table 1.

Table 1 Mapping results by one to one

Bytecode	**jHISC instruction**
iload_2	data.move Rb R2
iload_1	data.move Rc R1
iload_3	data.move Rd R3
iadd	arith.add Rc, Rc Rd
istore_1	data.move R1 Rc
iload 4	data.move Rc R4
isub	arith.sub Rb Rb Rc
istore_3	data.move R3 Rb
return	rvk

In the table, registers *Rb, Rc* and *Rd* are temporary registers allocated by the register file control engine. When bytecodes are fetched, the Folding Manager Unit will detect the first *O, T_P* or *C* type bytecode. Since it is *O_P* type (*iadd)*, the Folding Manager Unit will check whether the next bytecode to the *iadd* is *C* type or not. The two previous *P* type bytecodes adjacent to the *iadd* and the *C* type bytecodes will then form a folding group and are folded into the jHISC instruction *arith.add R1 R1 R3*. In the same way, bytecodes in group 2 can also be folded into the jHISC instruction (*arith.sub R3 R4 R2*). The bytecode *return* is *T* type and is translated into the jHISC instruction *rvk* directly. The folding results are presented in Table 2. We observe that the instruction length is reduced from 9 to 3 after folding and the temporary registers are not needed.

Table 2 Folding results

Original bytecodes	Folding group	jHISC instruction
iload_2	group 1: iload_1, iload_3, iadd, istore_1	arith.add R1 R1 R3
iload_1	group 2: iload_2, iload 4, isub, istore_3	arith.sub R3 R4 R2
iload_3	group 3: return	rvk
iadd		
istore_1		
iload 4		
isub		
istore_3		
return		

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) [14]. JVMTI profiler [12] was used to implement the proposed folding algorithm, trace bytecodes at run-time, and dump the executed bytecodes, the folding 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. 2 shows the percentages of the folded P and C type bytecodes relative to all operations and stack operations. 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%. However, the Sun's folding algorithm in PicoJava II folded up to 60% of all stack operations [2-3] [10]. The POC folding algorithm with 4-foldable strategy reduced up to 84% of all stack operations [6]. And the advanced POC folding algorithm claimed to eliminate about 93% of all stack operations in case load and store operations on array were mistaken to be treated as P and C type operations, respectively. Thus the actual folding ratio is smaller than 93%. With respect to all operations, the percentage of the eliminated P and C type instructions is from 42.2% to 51.8% and the average is 47% in the proposed folding algorithm.

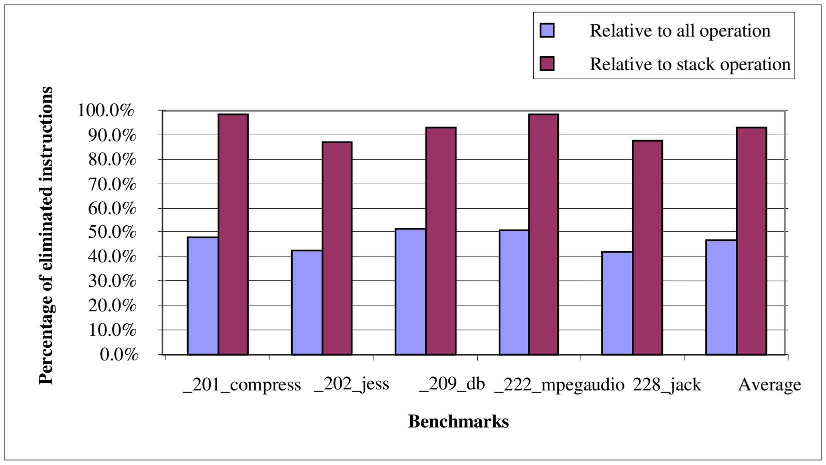

Fig. 2 Folded P and C type bytecodes

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 value in Java bytecodes and the jHISC instruction is only 32 bits in length. For example, if two 16-bit immediate data precede a bytecode *iadd*, during instruction folding and translation, 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 *arith.addi*. The percentage of the added operations relative to all operations is shown in Fig. 3, which varies from 0.3% to 7.4%.

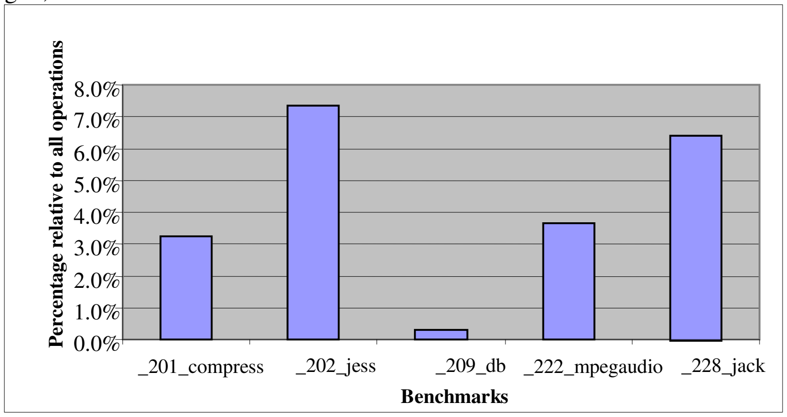

Fig. 3 Added data move operations

5.3 Generated jHISC instructions

Fig. 4 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.

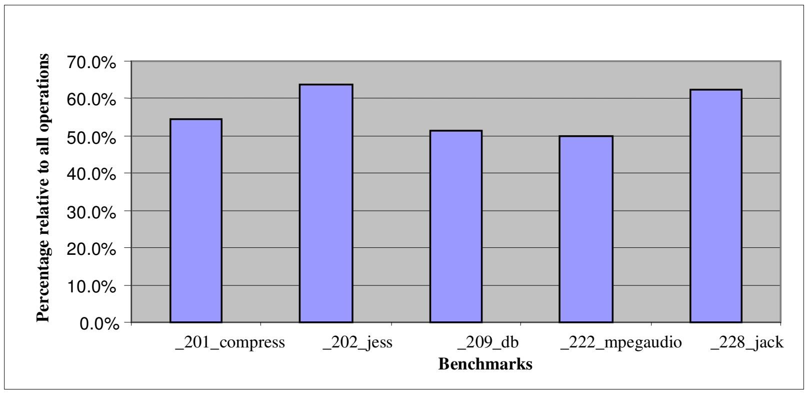

Fig. 4 Generated jHISC instructions

5.4 Performance enhancement

The overall reduced instructions can be calculated through the following equation.

$$\text{The overall reduced instructions } N = N_{total} - N_{jHISC}$$

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

Thus the percentage of the reduced instructions over all operations is obtained and shown in Fig. 5. 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.

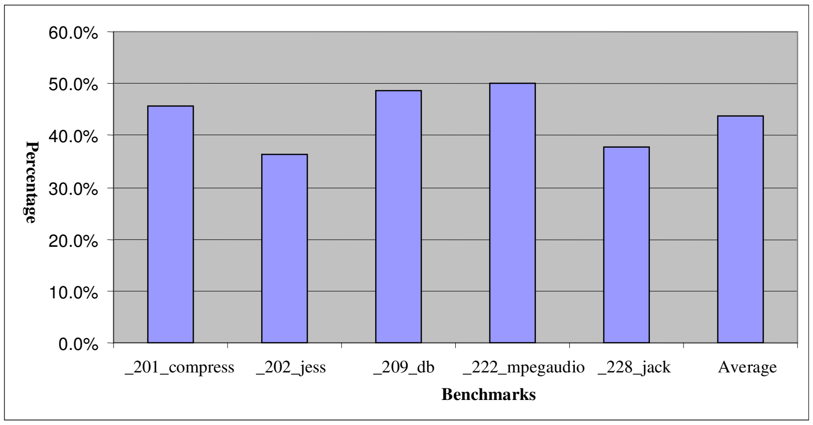

Fig. 5 Performance enhancement

6. Conclusion

More and more complex Java programs are applied in embedded devices, such as complicate games, network application programs. These require 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 results of incomplete folding groups are written into buffers and treated as P type bytecodes for further folding.

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% in the proposed folding algorithm. The overall reduced instructions are from 37% to 50% of all operations, and averagely, about 44% of all operations can be reduced in the benchmarks. Compared with other folding algorithms, the proposed algorithm has a great improvement on folding efficiency and system performance.

Acknowledgment:

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

References:

[1] Y. M. Lee, B. C. Tak, H. S. Maeny, S. D. Kim, “Real-time Java Virtual Machine for Information Appliances,” IEEE Transactions on Consumer Electronics, Vol. 46, No. 4, November, 2000, pp. 949-957.

[2] J. M. O’Connor, M. Tremblay, “PicoJava-I: The Java Virtual Machine in Hardware”, *IEEE MICRO*, March 1997, pp. 45-53.

[3] Sun Microsystems: PicoJava-II: Java Processor Core, Sun Microsystems Data Sheet, April 1998.

[4] N. Vijaykrishnan, and N. Ranganathan, “Object-oriented Architecture Support for a Java Processor”, *The 12th European Conference on Object-Oriented Programming*, 1998, pp.330-354.

[5] L. R. Ton, L.C. Chang, M.F. Kao, C.P. Chung, “Instruction folding in Java processors”, *Proceedings of the International Conference on Parallel and Distributed Systems*, 1997, pp. 138–143.

[6] L. C. Chang, L. R. Ton, M. F. Kao, and C. P. Chung, “Stack Operations Folding in Java Processors”, *IEE Proc.-Comput. Digit. Tech.*, Vol. 145, No. 5, September 1998, pp. 333-340.

[7] A. Kim, M. Chang, “Java Bytecode Optimization with Advanced Instruction Folding Mechanism”, *Lecture Notes in Computer Science*, Vol. 1940, 2000, pp. 268-275.

[8] L. R. Ton, L. C. Chang, J. J. Shann, and C. P. Chung, “Design of an Optimal Folding Mechanism for Java Processors’, *Microprocessors and Microsystems*, Vol.26, 2002, pp. 341–352.

[9] M. W. El-Kharashi: *The JAFARDD Processor: A Java Architecture Based on Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing*. Phd. Dissertation, University of Victoria.

[10] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, “A Robust Stack Folding Approach for Java Processor: an Operand Extraction-based Algorithm”, *Journal of Systems Architecture*, Vol. 47, 2001, pp. 697-726.

[11] Tan Yiyu, Yau Chi Hang, et al, “Design and Implementation of a Java Processor”, IEE Proceedings on Computer and Digital Techniques, Vol. 153, No. 1, Jan., 2006, pp. 20-30.

[12] Sun Microsystems: *JVM^{TM} Tool Interface (JVMTI) Version 1.0*. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[13] Tan Yiyu, Lo Wan Yiu, Yau Chi Hang, et al, “A JAVA Processor with Hardware-Support Object-Oriented Instructions”, Microprocessors and Microsystems, Vol. 30, No. 8, 2006, pp. 469-479.

[14] SPEC: JVM98 benchmark suits. http://www.spec.org/jvm98/