Journal of Circuits, Systems, and Computers
Vol. 19, No. 6 (2010) 1235–1244
© World Scientific Publishing Company
DOI: 10.1142/S0218126610006840

ACCELERATING JAVA INTERPRETATION IN LOW-COST EMBEDDED PROCESSORS*

TAEK-KYU KIM
The 3rd R&D Institute, Agency for Defense Development,
Yuseong, Daejeon, Korea
teddykim@add.re.kr

JONG-SUNG LEE
R&D Center, ADChips, #1009-5 Dae Chi Dong,
Seoul 135-280, Korea
jslee@adc.co.kr

HYEONG-CHEOL OH
Dept. of Elec. & Info. Eng., Korea University,
Chungnam 339-700, Korea
ohyeong@korea.ac.kr

Received 27 April 2009
Accepted 19 February 2010

Hardware interpretation is an attractive choice for implementing the Java virtual machine (JVM) in low-end embedded systems. However, the hardware interpretation of complex bytecodes is so expensive that most low-end embedded systems rely on slow software interpreters in processing complex bytecodes. This paper proposes a low-cost hardware approach for accelerating the interpretation of eight complex bytecodes: four object manipulation and four method invocation bytecodes. The proposed approach occupies 204 LUTs in a Xilinx FPGA and reduces by up to 61.5% the number of instructions executed in running the benchmarks considered in this paper.

Keywords: Java virtual machine (JVM); interpreters; embedded systems; object manipulation; method invocation.

1. Introduction

Nowadays a long list of small embedded systems are equipped with Java Micro Edition (JME) compatibility.[1,2] As the number and the diversity of embedded systems are explosively increasing, we can expect that not only more high-end systems, but also even more low-end systems will soon be added to the list. These low-end

*This paper was recommended by Regional Editor Krishna Shenai.

systems, including low-cost toys and smart tools, should not require much of silicon area for implementation. This paper focuses on low-end embedded systems with JME compatibility.

In low-end embedded systems, hardware interpretation is popularly used for implementing the Java Virtual Machine (JVM).[3] A hardware interpreter directly translates each bytecode into a *code fragment* that is a sequence of the instructions, called *native instructions*, of the *host processor* that executes the Java program. The code fragments are usually stored in, and fetched from, a memory for the hardware interpreter, as in Ref. 4.

Since the direct interpretation process is very complex for some bytecode instructions (called *complex bytecodes*), the hardware interpreter often resorts to a software interpreter for processing those complex bytecodes. The software interpretation of the complex bytecodes constitutes a large portion of the native instructions executed in running typical Java programs.[5] In order to process the bytecode *getfield*, for example, the interpreter needs to execute a large number of native instructions in performing symbolic resolution to obtain the pointer to the field offset. A typical acceleration approach for processing the complex bytecodes is to use the *quick* variants.[6] The *quick* variant approach replaces the bytecode with its variant after its symbolic resolution has been carried out. As depicted in Fig. 1, the *quick* variant of *getfield*, of which the opcode is denoted as *getfield_quick*, directly supplies the resolved offset.

Even though the process of finding the object pointer is less time-consuming than that of resolving the field offset, it still costs the processor several tens of native instructions and can exert a significant effect on the overall performance. Whereas high-end Java processors adopt powerful approaches for accelerating this process of finding the object pointer, the low-end embedded systems often repeat the required symbolic resolution process whenever they execute the object-dependent bytecodes.

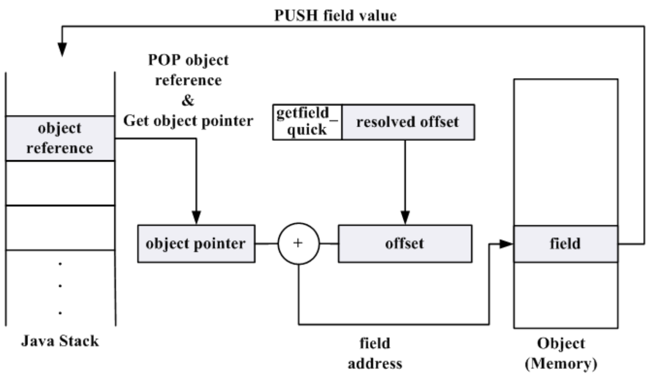

Fig. 1. Processing the bytecode *getfield_quick*.[6]

In this paper, we propose a cost-effective architectural approach for accelerating the interpretation of eight complex bytecodes: four object manipulation bytecodes, *getstatic*, *putstatic*, *getfield*, and *putfield*; and four method invocation bytecodes, *invokevirtual, invokeinterface, invokestatic,* and *invokespecial.* These are eight bytecodes of the most time-consuming ten object and array manipulation bytecodes when they are processed in their original format.[2] Even though the object-oriented operations are known to be relatively infrequent (about 10% of all operations) in some benchmarks including CaffeinMark,[2] Sec. 3 shows that the number of native instructions executed in running some game programs can be reduced by up to 61.5% when we efficiently handle these operations.

Architectural supports for complex bytecodes have been proposed on various machines (see Refs. 2 and 7–8, and the references therein). However, most existing approaches aim at high-end application domains. Whereas many of the existing works provide support for Java processors, the approach proposed in this paper is devised to be used for general-purpose embedded processors that rely on small Java interpreters. We do not assume availability of strong compilation support. Our experimental results show that the proposed approach can reduce the number of native instructions executed in running the benchmarks considered in this paper by up to 61.5%, and costs the interpreter only 204 LUTs in a Xilinx FPGA.

2. The Hardware Interpreter

2.1. *The base hardware interpreter*

The *base* hardware interpreter for which we develop the proposed acceleration approach is able to process seventy simple bytecodes only. All the other bytecodes are processed by a small software interpreter, Wabasoft's WabaVM.[9] As the host processor for the interpreters, we adopt ADChips AE32000C processor.[10] The host processor has sixteen 32-bit general-purpose registers (GPRs), some of which are used in the Java mode as Java stack pointer, local variable pointer, Java program counter, and so forth.

The hardware interpreter is placed between the fetch and decode stages of the host processor. In the Java hardware mode, the hardware interpreter fetches and decodes each bytecode. The hardware interpreter then selects a suitable code fragment stored in ROM and passes it to the host processor. The base hardware interpreter uses a ROM of 866 bytes (or two Block RAMs) and occupies 374 LUTs when it is implemented in a Xilinx FPGA.

2.2. *Supporting object manipulation*

An object in Java is a variable-sized contiguous list of words that can be considered as a set of fields.[3] The object manipulation bytecodes access and manipulate the fields. As mentioned in Sec. 1, even though various powerful approaches have been proposed

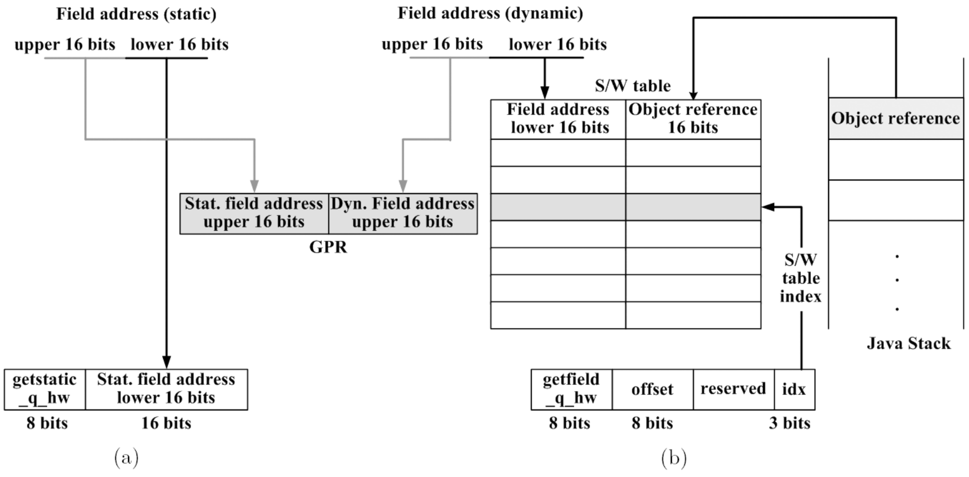

Fig. 2. Accelerating object manipulation for (a) static and (b) dynamic bytecodes.

for object manipulation in Java processors, they are not implementable at low cost in the low-end embedded systems. As a result, many low-end JVMs that adopt the *quick* variant approach repeat the time-consuming process of finding an object pointer whenever they execute an object manipulation bytecode. We propose to avoid these repetitions at low cost by adopting a new variant, denoted as *q_hw*, as shown in Fig. 2.

Using the approach shown in Fig. 2, the bytecode is replaced with its *q_hw* variant after symbolic resolution when it is processed for the first time. Figure 2(a) shows how we use the proposed variants *getstatic_q_hw* and *putstatic_q_hw*, of which the object pointers do not vary at runtime. The lower 16 bits of the field address are stored in the variant, whereas the upper 16 bits are stored in a predefined GPR of the host processor. If some bytecode has a field address that differs in the upper 16 bits from the one stored in the GPR, which happens rarely in small applications, we use the *quick* variants instead of the new *q_hw* variants.

For those *dynamic* bytecodes, *getfield* and *putfield*, of which the object pointers can vary at runtime, we store the object reference as well, to check if the object pointer has been changed. When the JVM processes a bytecode *getfield* or *putfield* for the first time, it resolves the constant pool and stores the object reference and the lower 16 bits of the resolved field address in a software table, as depicted in Fig. 2(b). The index (denoted as *idx* in Fig. 2) to the software table and the resolved (field) offset are stored in the variant. The GPR used by the static bytecodes is shared with the dynamic bytecodes for storing the upper 16 bits of the field addresses. In Fig. 2, we use an eight-entry software table. We use the *quick* variants when we need more storage than allowed.

When the JVM fetches the *q_hw* variant of a dynamic bytecode, it compares the object reference in the Java stack with the corresponding entry of the software table.

If it is a hit, then the JVM can save time and energy by avoiding the whole resolution process including the process of finding the object pointer. In case of a miss, the JVM looks up the object pointer, in the same way as it does in processing the *quick* variant, and it updates the corresponding table entry.

2.3. *Supporting method invocation*

As an object-oriented language, Java allows the same method in an inheritance hierarchy to have different implementations.[3] The implementation that is to be executed depends on the type of the object. As a time-consuming process, this dynamic dispatching (thus the execution of the method invocation instruction) also constitutes a large number of native instructions executed in running many Java applications.[2,5]

A static call, for example *invokestatic*, can be processed straightforwardly: the JVM obtains from the constant pool the class and the symbolic reference to the method, using the index given by the bytecodes, and it then performs symbolic resolution to find the method. The *quick* variant approach[6] avoids this resolution overhead by storing the result of the resolution, i.e., the pointer to the method structure, in the constant pool.

The proposed *q_hw* variant approach further reduces processing time by storing the method and class addresses as shown in Fig. 3. We store the lower parts of the addresses in a software table that has sixteen entries in Fig. 3. The table index (denoted as *Index1* in Fig. 3) is stored in the variant. We allow the *q_hw* variant to replace only the bytecode of which the method and class addresses share the same upper 16 bits. We store those 16 address bits in one predefined GPR of the host processor. If they are different from the stored ones, then we apply the *quick* variant approach.

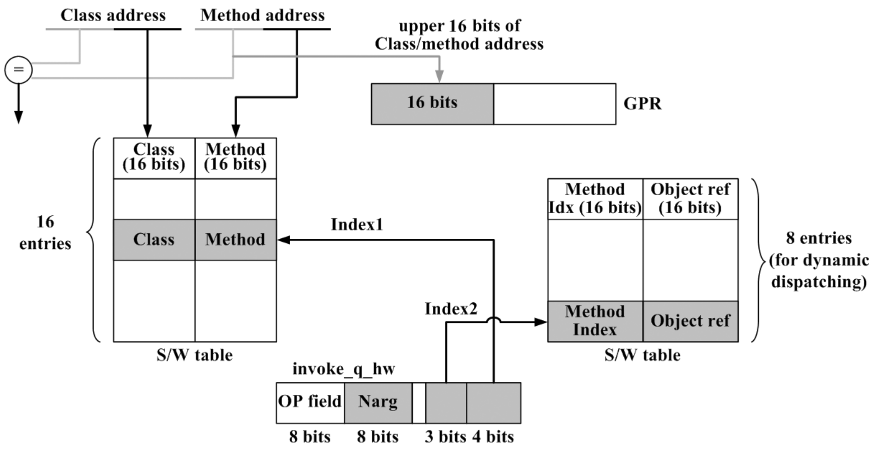

Fig. 3. Accelerating method invocation.

For those bytecodes which are dynamically dispatched, the *quick* variant approach stores and uses the resolved method index. In picoJava,[6] the object index is used as an offset to fetch from a memory table the pointer to the method address, which is not a proper approach for low-end systems since the approach requires the constant pool to be precompiled. In jHISC,[2] the data can be encapsulated into objects and described by the operand descriptors, which cannot be easily implemented at low cost on a general-purpose processor.

WabaVM[9] reuses the resolved method index but repeats symbolic resolution using the object reference whenever it meets the instruction. Our proposed approach works as follows: when the JVM processes a dynamic method invocation bytecode for the first time, it stores the resolved class and method addresses into a software table and the predefined GPR, as it does for the static calls. In addition, the JVM stores the method index in the current class and the object reference into an additional software table that has eight entries in Fig. 3. The index of the entry (denoted as *Index2* in Fig. 3) is stored in the variant. Then the opcode is changed into its *q_hw* variant.

3. Implementation and Evaluation

We have designed and modeled in C and Verilog the hardware interpreter equipped with the proposed acceleration approach. The design has been verified using the AE32000C model[10] ported on SimpleScalar[11] and implemented in a Xilinx FPGA. The proposed approach consumes 204 LUTs whereas the base hardware interpreter consumes 374 LUTs and 866B of ROM (or two Block RAMs).

Tables 1 and 2 compare the implementation results of the proposed approach with those of three Java processors, picoJava II,[8] JOP,[7] and jHISC.[2] For the *quick* and *q_hw* variants, Table 2 shows only the cycles taken by the hardware interpreter. Several hundreds of cycles may be taken when the software interpreter takes over the control. It takes a non-constant time ($156 + \alpha$ cycles) for WabaVM to process *invokevirtual* or *invokeinterface* using the *quick* variant approach. This is because WabaVM has to associatively find out the class of the reference by comparing the descriptors.

Table 1. Comparison of implementation costs. The data for Java processors are adopted from Ref. 2.

picoJava II (microcode unit) (xc2v4000bf957-6)	JOP (xcv800bg432-6)	jHISC (xcv800bg432-6)	Proposed H/W interpreter (xcv600bg432-6)
RAM128X1S: 42	ROM32X1: 21	Block RAMs: 28	Block RAMs: 2
ROM128X1: 6	Block RAMs: 13	—	—
ROM256X1: 122	—	—	—
3053 LUTs	2271 LUTs	15803 LUTs	374 + 204 LUTs

Table 2. Clock cycles taken for processing the bytecodes considered. The data for Java processors are adopted from Ref. [2]. For jHISC, the related instructions are shown.

Bytecode	picoJava II		JOP	jHISC	Proposed interpreter	
	original	*quick* variant			*quick* variant (Waba VM)	*q_hw* variant
putstatic	103	3	7	6 (psfld)	43	4
getstatic	103	3	6	6 (gsfld)	43	4
putfield	130	4	15	6 (pfld) 2 (pifld)	79	31
getfield	114	4	12	6 (gfld) 2 (gifld)	77	26
invokestatic	86	11	67	9 (ivkclass)	142	22
invokevirtual	195	15	88	9 (ivkintance) 5 (ivkinternal)	$156 + \alpha$	22
invokespecial	208	17	67	9 (ivkintance)	151	46
invokeinterface	203	184	96		$156 + \alpha$	46

In order to evaluate the proposed acceleration approach, we have performed simulations using two of three non-trivial programs in JavaBenchEmbedded V1.0,[12] Sieve and Kfl. (UdpIp was excluded since WabaVM[9] does not support threads). We also use Dhrystone[13] and Bboyes,[14] which are synthetic benchmarks aiming at integer arithmetic operations and communication applications, respectively. In Bboyes, the part of the code that uses double and long data types is excluded since WabaVM[9] does not support those data types. We also use two game programs, Connect4[15], of which only the game part is used, and Maze[16] which is a program that searches the best path among 24 mazes. (The benchmark programs can be accessed at http://atlas.korea.ac.kr/java.)

We use jikes-1.22[17] to compile the programs. For Sieve and Kfl, loops were executed $1024 * 8$ times. For Dhrystone, 1000 iterations were performed. The parameters in Bboyes, *ChunkSize* and *count* were set to 2^{15} and 2^{18}, respectively. The number of iterations was set to 200,000 for non-array functions.

Figure 4 summarizes the performance gain obtained by the proposed approach. For Connect4 and Maze, the number of native instructions depends on the game

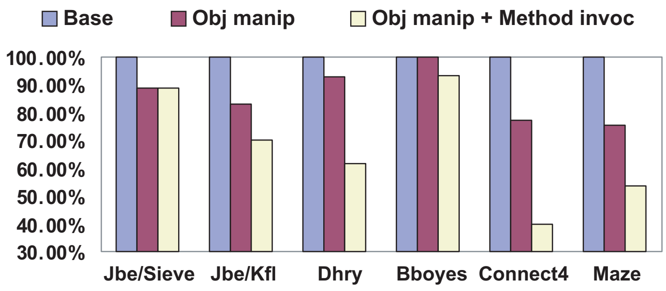

Fig. 4. Comparison of the number of native instructions executed in benchmark runs.

Table 3. Number of native instructions executed and its % reductions achieved by the proposed approach, in running Connect4 for four selected conditions.

Number of stones	Base	Obj. manip.		Obj. manip. + Method invoc.	
7	54,776,167	42,647,820	22.1%	22,035,884	59.8%
15	110,856,428	85,984,264	22.4%	42,723,360	61.5%
22	160,884,724	124,096,876	22.9%	63,420,092	60.6%
28	201,852,378	155,870,929	22.8%	79,074,551	60.8%

condition, so the harmonic averages over twenty game conditions of the numbers are presented for Connect4 and Maze in Fig. 4. The number of native instructions executed in running Connect4 significantly varies depending on the game condition (the number of stones) as it is presented for four selected conditions in Table 3.

Table 4 presents the number of native instructions executed in running the remaining five benchmarks. The number of native instructions executed in running Maze depends insignificantly (standard deviation $< 2\%$) on the game conditions considered (the coordinates of the start and end points), so the arithmetic average over twenty game conditions of the numbers are presented for Maze in Table 4.

Compared to the base hardware interpreter with WabaVM, the object manipulation part of the proposed approach significantly reduces the number of native instructions executed in running the benchmarks except for Bboyes which includes few object manipulation bytecodes. We observed similar reductions of execution time when we assume an ideal memory system. With more practical memory systems, we would observe greater performance improvements. For Bboyes, as shown in Table 4, we observe a slight increase in the number of native instructions due to the overhead of preparing the table for the proposed approach.

The method invocation part of the proposed approach also significantly reduces the number of native instructions executed in running the benchmarks, except for Sieve. Sieve includes few method invocation bytecodes which again present a slight overhead as is shown in Table 4. For Connect4, we have observed up to 61.5% reduction in the number of native instructions.

When the JVM implemented in this paper operates at 40 MHz, it returns 230 and 847 iterations per second for Sieve and Kfl, respectively. An FPGA implementation of picoJava-II, operated at 40 MHz, was reported to return 7797 and 23290 iterations

Table 4. Number of native instructions executed and its % reductions achieved by the proposed approach, in running benchmarks except for Connect4.

Program	Base	Obj. manip.		Obj. manip. + Method invoc.	
Jbe/Sieve	1,301,930,103	1,159,546,840	10.9%	1,159,565,872	10.9%
Jbe/Kfl	468,365,797	389,736,618	16.8%	327,669,071	30.0%
Dhrystone	71,217,207	66,018,432	7.3%	43,957,779	38.3%
Bboyes	770,933,263	771,937,342	−0.1%	720,739,657	6.5%
Maze	105,458,885	79,716,229	24.4%	56,414,222	46.5%

per second for Sieve and Kfl, respectively, in Ref. 8 but requires much more hardware as is shown in Table 1. Additional data including the number of memory accesses can be found in Ref. 18.

4. Conclusion

Low-end embedded systems often cannot afford Java processors, but they rely on Java interpreters. This paper presented a hardware acceleration approach for processing eight complex bytecodes in a low-end embedded system that relies on a small hardware interpreter supported by a small software interpreter. Our simulation and implementation results showed that the proposed approach could be implemented at low cost and that it could significantly improve interpretation performance.

Acknowledgments

We would like to thank the reviewers for their valuable comments and suggestions. This work was supported in part by a research grant from Korea University, Korea and a tool support from IDEC, Korea.

References

1. K. Glahn and E. Ryså, JSR 248: Taking Java platform, micro edition (JavaME) to the next level, *JavaOne Conference* (2007), TS-5608.
2. T. Yiyu, L. W. Yiu, Y. C. Hang, R. Li and A. S. Fong, A Java processor with hardware-support object-oriented instructions, *Micropro. Microsys.* **30** (2006) 469–479.
3. T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification* (Addison-Wesley, 2003).
4. D. J. Seal and E. C. Nevill, Data processing using multiple instruction sets. US Patent 6,965,984, 15 November 2005.
5. V. Narayanan, Issues in the design of a Java processor architecture, PhD Dissertation, Dept. of CSE, Univ. of South Florida, U.S.A. (1998).
6. J. M. O'Connor and M. Trembly, picoJava-I: The Java virtual machine in hardware, *IEEE Micro* (1997), pp. 45–53.
7. M. Schoeberl, A Java processor architecture for embedded real-time systems, *J. Syst. Archit.* **54** (2008) 265–286.
8. W. Puffitsch and M. Schoeberl, picoJava-II in an FPGA, *Proc. JTRES*, Vienna, Austria (2007), pp. 213–221.
9. Wabasoft Corp., WabaVM, http://www.wabasoft.com/download4.shtml.
10. H.-G. Kim and H.-C. Oh, A low-power DSP-enhanced 32-bit EISC processor, *Proc. HiPEAC*, Barcelona, Spain (2005), pp. 302–316.
11. T. Austin, E. Larson and D. Ernst, SimpleScalar: An infrastructure for computer system modeling, *IEEE Computer* **35** (2002) 59–67.
12. JavaBenchEmbedded, http://www.jopdesign.com.
13. A. R. Weiss, Dhrystone benchmark white paper, http://www.Synchro-meshcomputing.com/pdf/dhrystoneWhitePaper.pdf.
14. Bboyes, http://www.practicalembeddedjava.com/index.html.

15. Connect4, http://homepages.cwi.nl/∼tromp/c4/fhour.html.
16. Maze, http://en.wikipedia.org/wiki/Maze_generation_algorithm.
17. IBM jikes compiler for Java language, http://jikes.sourceforge.net/.
18. T.-K. Kim, Schemes for processing method invocations in a Java hardware interpreter, MS thesis, Dept. of EIE, Korea Univ., Korea (2007).

Copyright of Journal of Circuits, Systems & Computers is the property of World Scientific Publishing Company and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.