# High Performance Java Hardware Engine and Software Kernel for Embedded Systems

Morgan Hirosuke Miki, Motoki Kimura, Takao Onoye*, Isao Shirakawa
*Department of Information Systems Engineering,*
*Graduate School of Engineering, Osaka University*
*2-1 Yamada-Oka, Suita, Osaka, 565-0871 Japan*
**Department of Communications and Computer Engineering,*
*Graduate School of Informatics, Kyoto University*
*Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan*
*e-mail: miki@ise.eng.osaka-u.ac.jp, motoki@ise.eng.osaka-u.ac.jp,*
*onoye@kuee.kyoto-u.ac.jp, sirakawa@ise.eng.osaka-u.ac.jp*

**Abstract**: This paper describes an effective approach to Java execution through the use of embedded processors. A pair of hardware engine and software kernel are devised for existing embedded systems in order to execute Java applications efficiently, in such a way that 39 instructions are added to the original JVM dedicatedly for the software kernel implementation. The whole embedded system including the hardware engine of 6-stage pipeline with 30K gates can be integrated in a single chip. The proposed approach improves the execution speed by a factor of 5.7 in comparison with J2ME software implementation.

**Key words**: Java, embedded system, hardware engine, software kernel

## 1. INTRODUCTION

Since the introduction of Java [1] in 1995, it has been widely used in innumerous applications, from small systems such as electronic cards to high performance data base servers. Java also receives special interest as the network language not only for personal computers and work stations but also for the growing embedded system applications, due to the main features of (i) platform independence provided by Java Virtual Machine (JVM) [2], (ii) instruction level network security, and (iii) object oriented language.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35597-9_40

In general, JVM is implemented by software with the so called *interpreter* or *just-in-time* (JIT) compiler, where it should be pointed out that the software implementation can not suit embedded systems in terms of large memory usage, slow speed, and large power consumption. On the other hand Java specific processors[3-8] intended for efficient execution suffer from enormous overhead and cost of reconstructing operating system as well as intricate interfaces to existing embedded systems.

In order to solve the technical issues stated above, this paper investigates a unified approach to construct an efficient Java execution scheme added to existing embedded systems. This scheme consists mainly of hardware engine, software kernel, and configurable interface to host embedded processor. A part of those Java methods, which are defined in the non-I/O API library, can be invoked from the host embedded processor to execute Java applications.

The proposed architecture employs a 6-stage pipeline including a stack stage customized for the stack-based semantics of JVM. In order to enhance the execution speed of Java applications, the instruction folding and 39 additional instructions to JVM are provided. The architecture has been verified with the use of Altera APEX EP20KE 400 and Tensilica XT1000 emulation system.

As for ASIC implementation, the proposed system has been synthesized with 30K gates and 30K memory bits by using Virtual Silicon $0.18\mu m$ library, which can operate at 96MHz of clock rate. This result admits a single chip implementation of the whole Java execution system, including the proposed engine, host processor, and I/O modules.

As a result, Java execution performance has been improved up to a factor of 5.7 on an average, as compared with J2ME[9] for embedded systems.

Section 2 details the proposed Java System, Section 3 describes the implementation results, and Section 4 discusses the performance evaluation. Finally, Section 5 addresses concluding remarks.

## 2. PROPOSED JAVA SYSTEM

Figure 1 outlines a possible implementation of the overall system of our Java execution scheme. The 'Java System' is constructed of the 32-bit 'Java Engine', 'Java Memory', and 'Host Interface', while the 'Original System' is composed of the embedded 'Host Processor', 'System Memory', and 'I/O Interface'.

JVM defines 32- and 64-bit operation instructions. However, the usage of 64-bit instructions is much less than that of 32-bit ones in embedded systems, and hence in order to reduce the die size, Java Engine is designed for a 32-

bit machine. In addition, Java System can be easily customized for different embedded systems by modifying only Host Interface. For example, Java Memory and System Memory can be either integrated in one memory or separated into two.

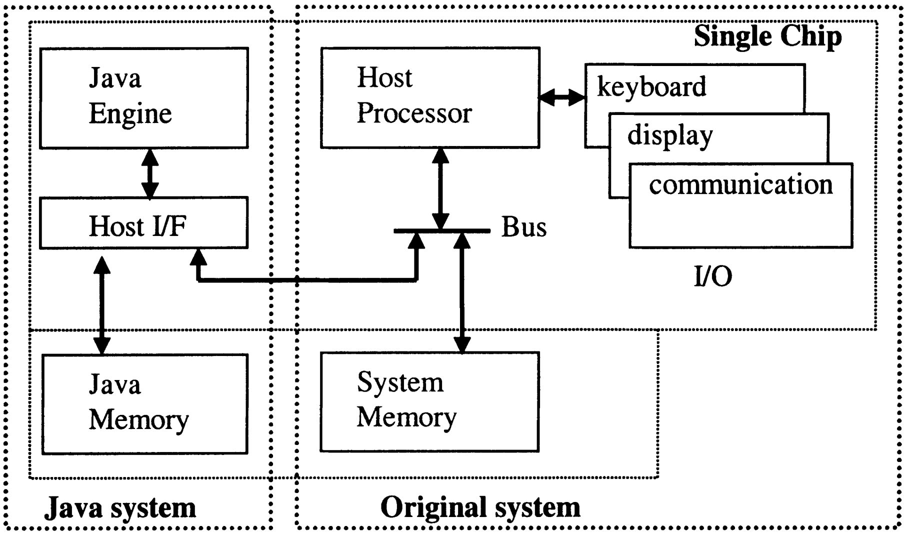

*Figure 1.* Overall system

Figure 2 outlines a Java application flow, which is achieved cooperatively by Host Processor and Java System. The execution starts with Host Processor invoking a Java method. Then, Java System executes Java bytecode for the method until the termination of execution. Meanwhile, if an I/O method is executed, Java System transfers the process to Host Processor which returns the process to Java Systems after the execution. While Java System executes bytecodes, Host Processor is free to execute any other non Java process in parallel.

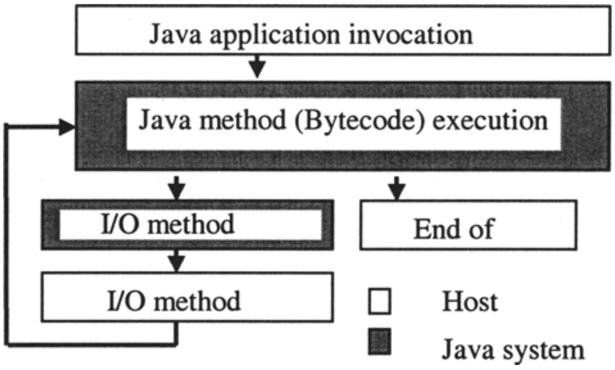

*Figure 2.* Java application flow

## 2.1 Design flow

Figure 3 shows a design flow of our implementation.

The design flow starts with dynamic analysis of JVM98[10] and CaffeineMark[11] benchmarks in order to determine the system specification features such as hardware/software instructions, additional instructions for software kernel implementation, cache configuration, pipeline, and instruction folding for optimized instruction execution.

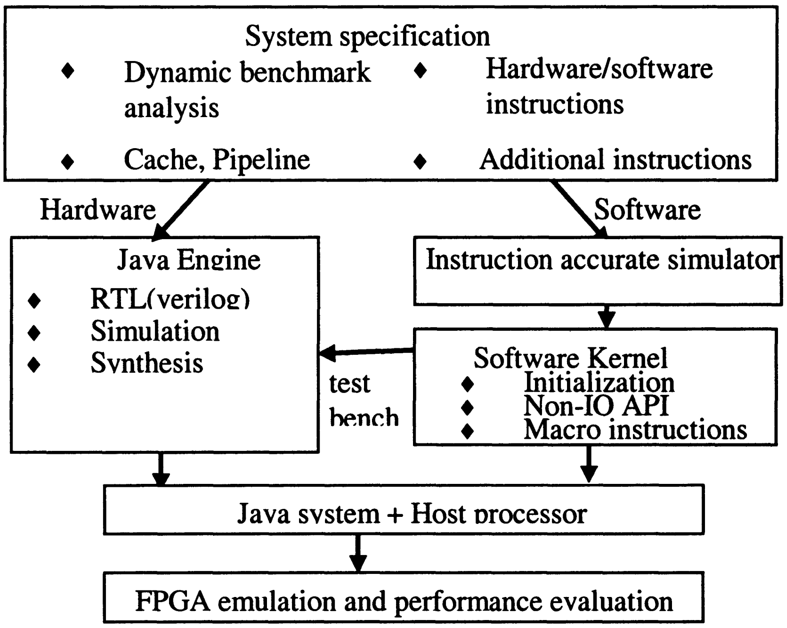

*Figure 3.* Design flow

Then, the design flow is split into software and hardware developments. For the efficient software kernel design, an instruction accurate simulator which models both hardware instructions and cache configuration is newly constructed. The software kernel is composed of codes for initialization sequence, non-I/O API library, and software macro instructions.

On the other hand, the hardware design is performed in the register-transfer-level (RTL) through the use of the verilog-HDL language with configurable options for cache size, number of stack registers, and bus width.

Test benches for software kernel implementations are also used for hardware design simulation.

Details of each step of design flow are described in the following.

## 2.2 Instruction set

JVM specifies totally 203 instructions. Most of the instructions, such as those for 32-bit data transfer (`iload`, `iconst_0`, `fload`), 32-bit integer operation (`iadd`, `ior`, `ishl`), and 32-bit branch (`ifeq`, `goto`), are easily implemented in one-cycle hardware pipeline.

However, highly complex instructions, such as those for method invocation (`invokevirtual`, `return`) and object allocation (`new`, `newarray`), are achieved by software macro.

On the other hand, the implementation of such instructions as swap, dup, 64-bit instructions, and floating point operations must be performed through the careful investigation into the instruction distribution of benchmarks and the complexity of hardware synthesis. As a result, all of 64-bit instructions for data transfer (`lload`, `dload`) and 64-bit integer operation (`ladd`, lor) are implemented by multicycle hardware pipeline; while those for `swap`, `dup`, and floating point operation (`fadd`, `dadd`) by software macro.

In order to implement macro software instructions in our Java Engine, 39 instructions are newly added to the JVM instruction set. These are mainly for (i) direct memory address 8/16/32-bit data transfer between the instruction/data memory and the operand stack, (ii) data transfer between special registers and the operand stack, and (iii) replacing/rewriting time-consuming instructions such as `new`, `invokevirtual`, and `getfield`.

## 2.3 Instruction folding

The instruction folding consists in the execution of a set of two or more instructions in one cycle, enhancing the execution performance of Java bytecode. Since it is not suitable to implement all possible patterns, a dynamic analysis is attempted for the benchmarks. Table 1 shows the result of most frequently used patterns, and Table 2 summarizes the performance improvement of equation (1) with the use of these patterns. It can be seen from this table that 15% improvement is achieved on an average.

$$\textit{Folding performance improvement} = \frac{\textit{Number of executed and folded instructions}}{\textit{Number of executed instructions}} \quad (1)$$

*Table 1.* Instruction folding patterns

| pattern | comment |
|---|---|
| LC LV OP | compute memory address for data read and performs the operation, reducing 2 data write in top of stack. |
| LV LC OP | compute memory address for data read and performs the operation, reducing 2 data write in top of stack. |
| LC LC OP | performs an operation with 2 constants, reducing 2 data write in top of stack. |
| LV OP | get one data from top of stack, compute memory address for data read, and performs the operation reducing one data write in top of stack. |
| LC OP | get one data from top of stack, and performs the operation reducing one data write in top of stack. |
| LC MEM | compute memory address for constant data write, reducing one data write in top of stack. |

LC: load constant in top of stack
LV: load data from local variable in top of stack
OP: operation instruction
MEM: memory write

*Table 2.* Instruction folding performance improvement

| benchmark | execution improvement |
|---|---:|
| JVM98.check | 25.4% |
| JVM98.compress | 20.1% |
| JVM98.jess | 5.8% |
| JVM98.db | 12.0% |
| JVM98.javac | 6.2% |
| JVM98.mpegaudio | 26.2% |
| JVM98.mtrt | 4.4% |
| CaffeineMark | 16.3% |
| average | 14.6% |

## 2.4 Java engine architecture

Figure 4 details an architecture overview of Java Engine which is based on *Harvard Architecture* and 6-stage instruction execution pipeline. Since JVM is a stack-based machine, an *operand stack* (STK) stage is included in the pipeline. A *memory read* (MR) stage precedes the *execution* (EX) stage in order to execute efficiently specific patterns of instruction folding.

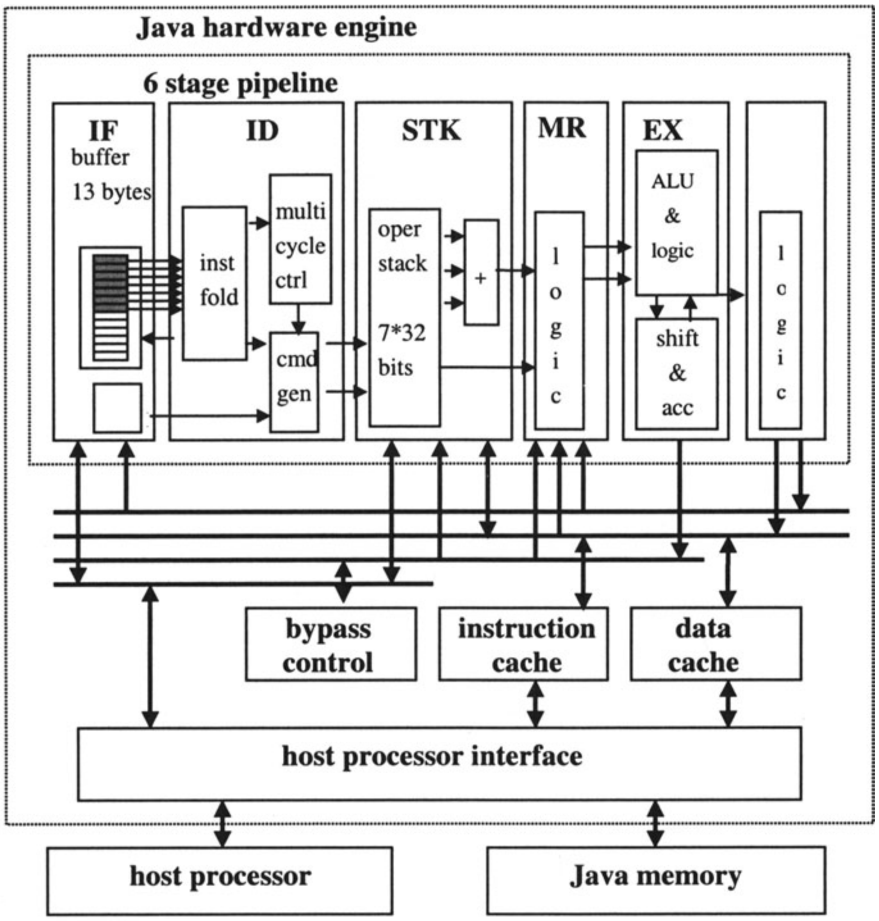

*Figure 4.* Architecture overview

The following are the details of each module of the architecture.

**Instruction Fetch (IF)**

IF is to fetche an instruction code from instruction memory and updates the program counter register. Since Java bytecode length is variable, a 13-byte shift register is used to buffer instructions from the instruction memory. Up to the leading 7 bytes of the shift register can be read by the *instruction decoder* (ID), which returns the number of decoded bytes to update the instruction shift register.

**Instruction Decoder (ID)**

ID is constructed by tree units; instruction folding, multicycle control, and command generator.

The input of the instruction folding unit are the 7 bytes from IF, which are interpreted as a single instruction or a pattern of 2 or 3 instructions of instruction folding.

The multicycle control unit is a state machine which generates control signals for the command generator unit for multicycle instructions. The number of cycles is set to 2 for long instructions, 16 for `imul`, and 32 for `idiv`.

The command generator unit gets the output of instruction folding and multicycle control units, and generates the command and data for the following 4 stages.

**Stack (STK)**

STK consists of an operand stack unit, an adder, and most of special registers of Java Engine.

Top 7x32 bits of the Java operand stack are loaded to the operand stack unit of STK, while the rest is in a segment of data memory. When the operand stack unit underflows or overflows, it transfers data to or from, respectively, the data memory.

The adder is to add the data from ID, operand stack unit, or local variable register, for the memory address calculation or for the comparison of conditional branch instructions, which are used in the next stages of the pipeline.

**Memory Read (MR)**

MR is to read 8/16/32-bit data from the instruction memory or the data memory.

**Execution (EX)**

EX is to execute arithmetic/logic and shift operations, where the shift unit is provided with an accumulator which holds the values of multicycle instructions such as multiplication and division.

**Write Back (WB)**

WB is to write a 8/16/32-bit data into the instruction memory, data memory, operand stack, and program counter.

**Bypass**

To reduce the number of data hazards of the Java stack engine, the bypass mechanism is indispensable. Whenever possible, one or two 32-bit data output from EX and/or MR are bypassed to the input of EX, MR, or STK, reducing the stalls of pipeline.

**Cache**

Instruction Cache is for the direct mapped cache write back, with 16 bytes in each block.

Data Cache is for the 2-way set associative write back with 16 bytes in each block. *Least Recently Used* (LRU) method is used in each set for the cache miss control.

The cache sizes of both Instruction and Data Cache are configurable to 1K, 2K, 4K, 8K, and 16K bytes.

## 2.5 Software kernel

Since the JVM specification does not deal with the implementation details of JVM and its instructions, an optimized software kernel should be constructed to maximize the performance of bytecode execution. The kernel includes the codes for initialization sequences, macro instructions, API library, and all necessary data structures.

During the initialization, the memory is initialized and some classes of API library are loaded.

Macro instructions are the software implemented instructions such as floating operations, method calls, and instructions to deal with objects.

## 3. IMPLEMENTATION RESULTS

The proposed Java System composed of a hardware engine and software kernel implements the total of 203 Java bytecodes plus 39 additional instructions. Table 3 details the implemented instructions.

The proposed hardware architecture described in the Verilog-HDL has been verified on Altera APEX EP20KE400 FPGA with the use of Xtensa XT1000 processor emulation module (Figure 5). The architecture has also been synthesized by Virtual Silicon $0.18\mu m$ library through the use of Synopsis Design Compiler. Table 4 shows the implementation results. In addition, the cost of instruction folding unit, which improves the performance by 15%, is only 700 of 30K gates.

*Table 3.* Implemented instructions

| instructions | implementation (num) | detail |
|---|---|---|
| JVM instructions | hardware(140) | 32 bit data transfer, integer operation, branch |
| | software(63) | method call, object, float operation |
| additional instructions | hardware(24) | direct memory access, register data transfer |
| | software(15) | fast instructions |

*Table 4.* Implementation results

| | |
|---|---|
| technology | 0.18$\mu m$ |
| number of gates | 30k gates |
| max. clock rate | 96MHz |
| cache | 30k bits (1kbyte for each set) |

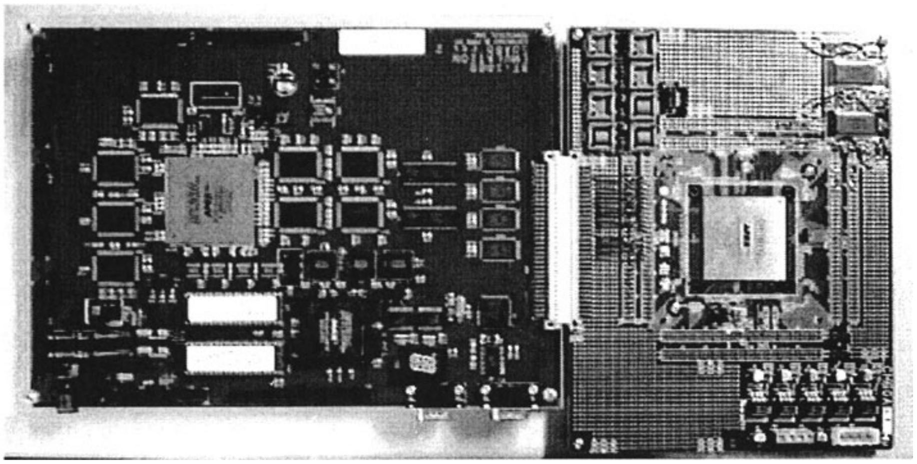

*Figure5.* Verification board: Xtensa XT1000 (left) and proposed hardware (right)

## 4. PERFORMANCE EVALUATION

Expected performance of the proposed system has been evaluated with the use of the CaffeineMark[11] benchmark. Table 5 shows the performance of the proposed hardware engine and J2ME run on Sun Ultra Sparc (450 MHz). The proposed architecture is 5.7 times faster than J2ME on an average.

Since J2ME is originally devised for PDAs, we have also tested CaffeineMark with the use of Palm III, working on MC68EZ328[12] of 20MHz. However, due to the old CISC architecture, the performance is so slow as to be less than 0.05 *cm*/MHz.

*Table 5.* Performance Evaluation on CaffeineMark

| Benchmark | J2ME on Ultra Sparc(*cm*/MHz) | ProposedHardware (*cm*/MHz) |
|---|---|---|
| CaffeineMark.Sieve | 0.52 | 9.65 |
| CaffeineMark.Loop | 0.50 | 14.65 |
| CaffeineMark.Logic | 0.49 | 8.35 |
| CaffeineMark.String | 1.45 | 2.9 |
| CaffeineMark.Float | 0.46 | 1.05 |
| CaffeineMark.Method | 0.53 | 0.45 |
| Overall | 0.60 | 3.4 |

## 5. CONCLUSION

This paper has described a hardware and software codesign approach to high performance Java execution for embedded systems. The system implements totally 203 JVM bytecode instructions plus 39 additional instructions for software kernel optimization. Among these, 78 instructions are implemented by the software kernel, and 164 instructions by the 6-stage pipeline engine. The proposed system executes all non-IO Java methods allowing the parallel operation of both Java System and Host Processor of the embedded system.

The proposed hardware Java Engine has been coded in the verilog-HDL to be synthesized by Virtual Silicon 0.18$\mu m$ library, and has been integrated with the use of 30K gates, allowing a single chip implementation of the whole system including Host Processor and I/O modules.

Future work is continuing on the hardware implementation of floating point instructions and the optimization of method invoke/return instructions in order to enhance the performance of String, Float, and Method CaffeineMark tests.

## 6. REFERENCES

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Second Edition*, Addison Wesley, Los Altos, California, April, 2000.

[2] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification Second Edition*, Addison Wesley, Palo Alto, California, April, 1999.

[3] J.M. O'Connor and M. Tremblay, "picoJava-I: The Vitrual Machine in Hardware", *IEEE MICRO*, Vol. 17, No. 2, Mar./Apr. 1997, pp. 45-53.

[4] Sun Microsystems Inc., *picoJava-II Microarchitecture Guide*, Mar. 1999.

[5] Advancel Logic Corporation Inc., *TinyJ Processor Core Datasheet*, May 1999.

[6] aJile Systems Inc., *Real-time Low-power Java Processor aJ-100 Datasheet*, Sept. 2000.

[7] Patriot Scientific Corporation Inc., *PSC1000 Microprocessor*, July 1997.

[8] S. Kimura, H. Kida, K. Takagi, T. Abematsu, and K. Watanabe, "An application specific Java processor with reconfigurablities", *Proc. Asia and South Pacific Design Automation Conference 2000 (ASP-DAC 2000)*, Jan. 2000.

[9] Sun Microsystems Inc., *Java 2 Platform, Micro Edition*, June 1999.

[10] The Standard Performance Evaluation Corporation, *SpecJVM98 VERSION 1.03*, 1998.

[11] Pentagon Software Evaluation Corporation, *Java CaffeineMark 3.0*, 1999.

[12] Motorola Inc., *MC68EZ328 – DragonBall EZ Product Brief*