A Java-to-ARM Hardware Code Translator

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

[1] Department of Computer Science and Information Engineering, National Chiao Tung Universit
No. 1001, Dashiue Rd., Hsinchu 30056, Taiwan
{lrton@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 31041, Taiwan
{lcchang@ccl.itri.org.tw}

Abstract—

Java is one of the most popular programming languages. Various methods to run Java instructions (called Java bytecodes) have been developed, but each has its drawback. Therefore, we propose the concept of a Java/ARM dual-mode processor, which offers a better alternation to run Java and is still compatible with ARM for existing C applications. When executing the Java applets/applications, the bytecode instructions can be executed by the dual-mode processor on-the-fly. This is achieved by the proposed Java-to-ARM hardware code translator which is built in between the instruction fetcher and the instruction decoder. In the Java execution mode, the translator converts each bytecode into one or more ARM instructions on-the-fly. Besides, an ARM instructions folding technique is proposed to enhance the performance by combining the translated but not decode ARM instructions. Simulation results show that up to 15% performance gain can be achieved comparing to the original translator without folding.

Keywords— **Java bytecode, ARM processor, dual-mode execution, binary translation**

I. Introduction

The widespread use of Internet and network computing makes Java a popular language in developing network-related applications. With its portability, compact code size, object-oriented and multi-threaded natures, Java is suitable for network computer processors, PDAs, Internet TVs, or other consumer and embedded products [RICK 98].

Although Java has many advantages for developing embedded applications, current execution environment for Java makes embedded system developers hang back. This situation can be observed clearly by analyzing th following four major methods for executing Java:

- Interpreter: Interpreter is the most common and is traditionally used to execute Java. The interpreter itself is composed of a big time-consuming loop to map each Java bytecode instruction into native code sequences, which affects run-time performance significantly.
- JIT (Just-In-Time) compiler: JIT compiler compiles the Java bytecodes when the bytecodes are neve run before. JIT compiler stores the compiled codes because that the codes may be run again directl instead of compiling them redundantly. JIT compiler improves a large amount of performance as opposed to the interpreter. However, JIT compiler requires a large amount of memory to store the JIT itself and the compiled codes. Stud shows that compiled native code size is 25 times larger than the original Java bytecodes [JIMT 97]. This may not be acceptable in the embedded systems.
- Off-line compiler: Off-line compiler compiles the Java applications or applets before running as opposed to the run-time JIT compiler. Code optimizations are heavily used to exploit more parallelism in Java bytecodes. However, off-line compiler suffers from the same code explosion problem like the JIT compiler.
- Java processor: This is the first concept unveiled b Sun Microsystems to execute Java bytecodes directly by hardware [O'CO 97][SUN 99]. Java processor has no performance and memory size problems, but still suffers from the complete supporting of system software. Moreover, the requirements to execute existing applications based on C or C++ are not what a Java processor can offer [JIMD 98].

Therefore, we propose the concept of a Java/ARM dual-mode processor, which offers a better alternation to run Java and still compatible with ARM for existing C or C++ applications. Unlike the Java ILP machine [KEM 97] or the Caffeine prototype [HSI 96], the proposed Java-to-ARM translator is a hardware code translator built in between the ARM instruction fetcher and the instruction decoder. In Java execution mode, the fetched instructions are of bytecodes. The translator is enabled to translate each of the fetched bytecodes to one or more ARM instructions. The translated ARM instructions are then sent to the ARM instruction decoder to finish their instruction cycles. In ARM execution mode, the fetched instructions are of ARM instructions and thus do not need to run through the translator but to the ARM instruction decoder directly.

The rest of the paper is organized as follows. In section II, we will survey the current status for dual-mode

execution. In section III, we will introduce the simulation environment and benchmarks used in this study. The proposed Java stack to ARM register mapping mechanism to generate operand fields among the ARM instructions is given. In section IV, the translator architecture is described. Simulation results are analyzed for the translator design. In section V, the translator architecture is further enhanced to provide better performance by adopting ARM instruction folding. Related architecture design and simulation result are given also. Finally, section VI summarizes and discusses the future works.

II. Introduction to Dual-Mode Execution

A. *x86/IA-64 Merced Processor*

In approximately five years ago, Intel and HP announce their next generation 64-bit core architecture named IA-64 [PET 96][LIN 97] [CHR 97][RON 98]. This novel architecture adopts some parallel processing and compiling techniques but still faces the problem of lacking supporting software applications. Intel conquers this by combining both x86 and IA-64 execution core into the same chip named Merced [PET 96][LIN 97]. In the US patent [GRA 97] issued by Intel, the possible architecture for Merced is consisted of one IA-64 execution unit for both x86 and IA-64. The register file can be designed as a shared one or separated two files for IA-64 and x86 respectively. Th common instruction fetcher fetches instructions from the instruction cache and checks the internal mode flag to decide that the instructions are either sent to the x86 or to IA-64 translator. Then the common instruction decode decodes the IA-64 instructions sent from the instruction fetcher or translator. The execution mode switching is accomplished by adding eight new instructions for IA-64 and one new and one modified instruction for x86. When interrupt occurs, Merced will switch to x86 mode to activate the interrupt handler regardless of the current mode of IA-64 or x86.

B. *Proposed Java/ARM Dual-Mode Processor*

When designing the Java/ARM dual-mode processor, we target the design goal as direct hardware execution fo both Java and ARM instructions to enhance the execution performance of Java with the compatibility of executing the original ARM applications. This design is based on an ARM7 processor core [ARM 95], including an instruction decoder, an execution unit and a register file. As shown in Fig. 1, the instruction fetcher is modified slightly to matc the unaligned access behavior of Java execution mode. Furthermore, a sizer unit and a Java-to-ARM translator are added in between the instruction fetcher and the instruction decoder. The demultiplexer sends instructions from the instruction fetcher to the sizer or the ARM instruction decoder according to the mode selection bit in CPS (Current Program Status Register).

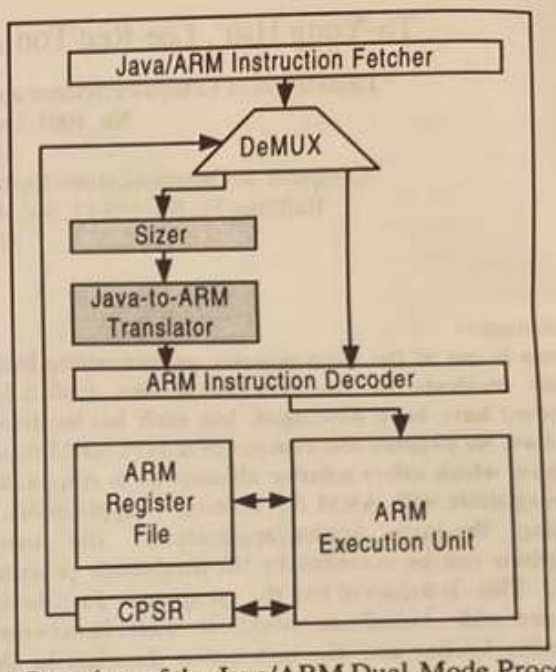

Fig. 1 Overview of the Java/ARM Dual-Mode Processor

C. *Mode Switching for Java/ARM Dual-Mode Processor*

Hardware support for switching between Java and ARM execution modes is accomplished by modifying one reserved bit in ARM's CPSR as the Java/ARM mode bit. As shown in Fig. 2, the bit 8 of the CPSR is one of twenty reserved bits in the ARM processor. We define this bit as the ***J*** bit (Java mode bit). If the ***J*** bit is 1, it means that th Java/ARM dual-mode processor is in the Java execution mode. The fetched instructions are of Java bytecodes and they are sent to the translator prior to the ARM instruction decoder. Otherwise, the Java/ARM dual-mode processor acts like a traditional ARM processor.

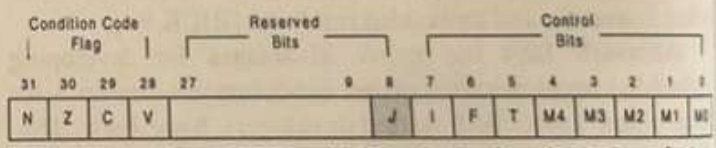

Fig. 2 Modified CPSR with Newly Defined Bit 8 -- ***J*** Bit

In the Java execution mode, mode switching can be achieved by translating complex Java bytecode instructions to the SWI (Software Interrupt) instructions for ARM decoder to force the ARM core entering the supervisc operating mode. In this mode, the interrupt handler can modify the ***J*** bit to 0 to enter the ARM execution mode. On the other hand, if the current execution mode is in ARM mode and the operating mode for ARM is not of user mode, the mode switching can be accomplished by modifying the CPSR directly. If the ARM's operating mode is of the user

mode, the instructions for switching to the non-user mode must be run first to get enough priority to modify the CPSR.

III. Simulation Environment and Preliminary Results Analysis

A. Simulation Environment and Benchmarks

In this study, we developed a trace-driven simulator to simulate the proposed translator. The traces are generated by the trace generator. By modifying the "interpreter.c" of Sun JDK 1.0.2, the trace generator provides the opcode, operands, program counter and branch information, etc. fo the simulator.

The benchmark programs consist of seven single-threaded Java application programs. The first benchmark i the famous Linpack benchmark written in Java. The other six benchmarks, designed to expose the capabilities of a system's CPU, FPU, and memory system, are part of BYTE Magazine's BYTEmark benchmark suite. The properties of these benchmarks are summarized in Table 1.

TABLE I
Properties of Selected Java Benchmarks

Benchmark Name	Bytecode Instructions Count
Linpack	8,019,462
Assignment (Assign)	10,002,219
Bit Operations (BitOps)	6,395,978
IDEA	2,976,197
LU	11,757,031
Numeric Sort (NS)	8,036,966
String Sort (SS)	4,297,097

B. Stack to Register Mapping Mechanism (SRMM)

In the Java/ARM dual-mode processor, the available number of registers is only sixteen. Only the control registers (CR), operand stack (OS) and local variables (LV) in the activated method frame will be mapped during the translation process. For simplicity, the SRMM uses fixed mapping for both OS and LV as opposed to the dynamic allocating.

If the number of mapped LV register is N, then the mapped register indices are arranged from 0 to N - 1. If the Java bytecode instruction's LV index is smaller than N, the translator will translate the ARM instruction which is of register data processing group. If the LV index is greater o equal to N, memory data transfer instructions are translated for accessing the LV in memory. Consequently, more LV entries are mapped into ARM's register file, highe performance can be achieved.

To access the OS, the TOSR (Top of Stack Register) is used as an index to the first element in OS. If the number of mapped OS is M, then the mapped register indices are arranged from N to N + M - 1. Every time when a bytecode instruction is executed, the TOSR is updated. If the required stack entry is not in between N to N + M - 1, OS overflo or underflow occurs. Related control registers are defined below for overflow and underflow processing.

Definition: Operand Page (OP) -- When overflow/underflow occurs, operand stack entries are swapped out/in to/from memory. These swapped operand stack entries form an operand page. The number of operand stack entries, annotated as *sizeof (OP)*, is fixed for design simplicity.

Definition: Operand Page Pointer (OPP) -- This register contains the memory address to store the new operand page.

Definition: Operand Page Counter (OPC) -- This register contains the number of operand pages stored in memory.

When the TOSR is greater than N + M - 1, OS overflo occurs. The overflow handling mechanism is described below:

1. Swap the OP starting from register N to memory address indicated by OPP.
2. Move the remaining registers indexed from N *sizeof (OP)* - 1 to N + M - 1 to the registers indexed from N to N + M - *sizeof (OP)* - 1.
3. OPP = OPP + *sizeof (OP)* * 4.
4. OPC = OPC + 1.
5. TOSR = TOSR - *sizeof (OP)*.

When OPC is greater then 0 and TOSR is less than N + M - *sizeof (OP)* - 1, OS underflow occurs. The underflow handling mechanism is described below:

1. Move the registers indexed from N to N + M - *sizeof (OP)* - 1 to the registers indexed from N + *sizeof (OP)* - 1 to N + M - 1.
2. OPP = OPP - *sizeof (OP)* * 4.
3. Swap the OP addressed by OPP to the registers indexed from N to N + *sizeof (OP)* - 1.
4. OPC = OPC - 1.
5. TOSR = TOSR + *sizeof (OP)*.

Except that parts of the LV and OS are mapped into ARM registers, five frequently used control registers (CR are also mapped into ARM registers for accelerating Java execution mode, as described below:

- Program Counter (PC): This register indicates the address of the next bytecode to be executed.
- Operand Page Pointer (OPP): As described above.
- Local Variable Base Pointer (LVBP): This register contains the base memory address of the LV of the currently activating method frame.

- Frame State Base Pointer (FSBP): This register contains the base memory address of the frame states (e.g., return PC, constant pool address, etc.) of the currently activating method frame.
- Tunnel Register (TR): This register is used to store/restore the TOSR and OPC when exception occurs to/from memory.

The number of ARM registers except that five of them are reserved for CR is only 11 for both LV and OS mapping. We simulate various combinations for LV and OS and the results show that the mapping of 6 OS and 5 LV combination leads to least memory reference overhead cycles (Fig. 3).

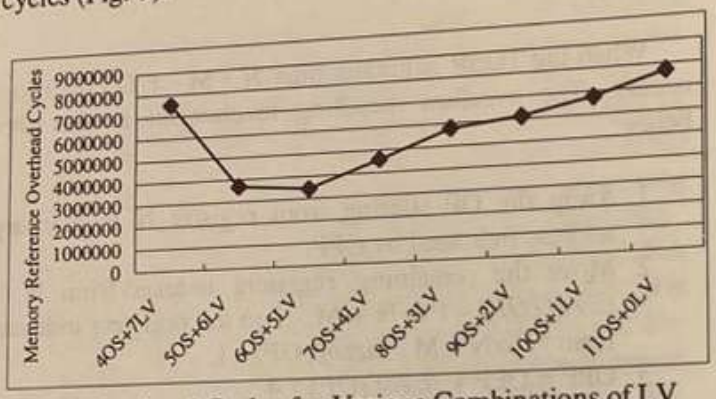

Fig. 3 Overhead Cycles for Various Combinations of LV and OS Mapping

IV. JAVA-TO-ARM TRANSLATOR

The ARM core architecture is a pipelined RISC machine. At most one instruction can be executed each cycle. In contrast, Java bytecode instructions are of CISC type. In other words, a simple Java bytecode instruction can be mapped into one ARM instruction but a complex Java bytecode instruction requires more than one AR instructions. Consequently, the basic Java-to-ARM translator model can translate one Java bytecode instruction to one or more ARM instructions, and send each translated ARM instruction to the ARM decoder sequentially.

A. *Bytecode to ARM Code Mapping*

When designing the translator, we need to know how much ARM instructions are required for translating eac bytecode instruction. Simulation shows that most bytecode instructions can be mapped into four or less ARM instructions. Complex and floating point bytecode instructions like `invokevirtua` and `ddi` need a long sequence of ARM instructions to emulate the function. These bytecode instructions will be translated to the software interrupt (SWI) instructions and trap to the corresponding interrupt handler. Consequently, the translator can translate one bytecode instruction to at most four ARM instructions. As shown in Fig. 4, 83% bytecode instructions can be translated directly to ARM instructions in average. 17% bytecode instructions are translated to the ARM's SWI instruction to switch to the ARM mode with 3 cycles overhead. Furthermore, another SWI instruction is required in the interrupt handler to switch back to the Java mode. This leads to a total overhead of 6 cycles for using the SWI instruction for each untranslated bytecode. The required number of SWI instructions in the LU and Linpack benchmarks is higher than others because both of them are floating-point oriented but ARM's ISA supports fixed-point only.

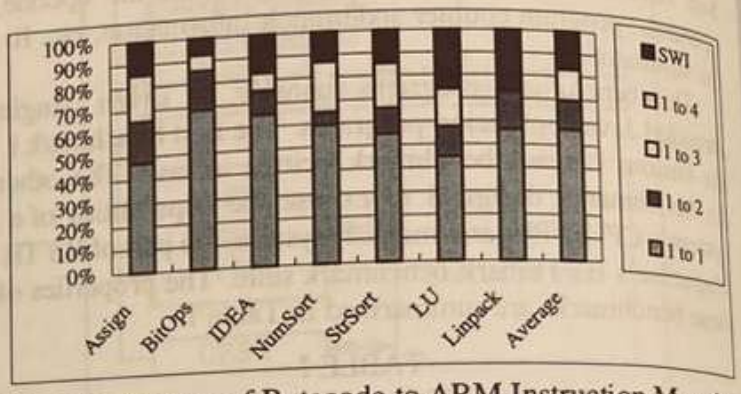

Fig. 4 Percentage of Bytecode to ARM Instruction Mapping

B. *Translator Core Design*

As shown in Fig. 5, the basic translator is consisted of two units -- lookup table and ARM instruction generator. The lookup table contains the required translation information for each bytecode. The number of required entries of the lookup table is $2^8 * 4$, where 2^8 is because that the size of opcode for each bytecode is 8 bits, and 4 mean that at most 4 ARM instructions can be translated for each bytecode. The size for each entry is 38 bits, including 5-bit for ARM opcode index, 2-bit for TOSR maintenance, 30-bit for generating ARM operand fields, and 1-bit fo indicating the end of bytecode translation.

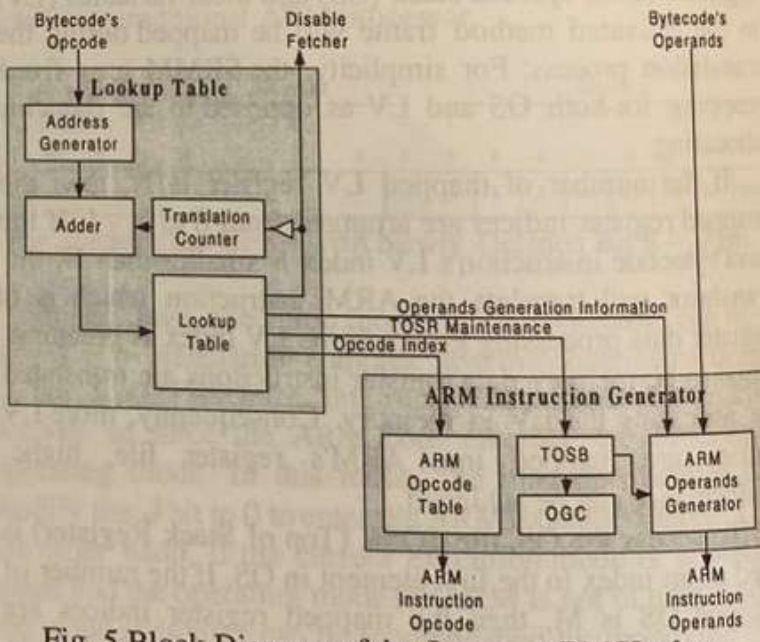

Fig. 5 Block Diagram of the Java-to-ARM Hardwar Translator

There are two special conditions for bytecode to ARM code translation as described below

- Wide condition: In bytecode definition of the JVM, the `wide` instruction is used to double the width of the operand fields of the succeeding bytecode which is one out of `iload`, `floa` , `aload`, `lload`, `dloa` , `istor` , `fstore`, `astore`, `lstore`, `dstor` , `re` , and `iin` . In our translator design, the sizer unit will generate a WIDE bit to indicate whether the wide condition is met or not. If the WIDE bit is 1, the translation information is generated using a special designed table indexed by the succeeding byte of the `wide` instruction instead of the original lookup table.
- Selective translation condition: The translation for some bytecode instructions will depend not only the opcode but also the operand value. For example, the succeeding byte of th `iload` instruction indicates that which LV entry will be loaded. If thi entry is already mapped into one ARM's register, the translated ARM code is `MOV` instead of accessing the memory using `LDR`. This condition can be resolved by adding one small table like the wide condition. In facts, both conditions are considered together and designed to share the same table for simplicity.

The major works for the ARM instruction generator are described below:

- Opcode generation: In our translator design, onl 28 different ARM instructions are used during translation. Consequently, we do not store the translated ARM instructions for each bytecode in lookup table. Instead, these ARM instructions are stored in a small ARM opcode table and the lookup table contains the indices only. This design reduces the required die size and hardware cost.
- Operands generation: In ARM instructions format, bit 0 to bit 19 are of operands. The operands can be further divided into different fields and the definition for each field varies from instruction to instructions. The smallest width of each field is 4 bits. Therefore, we divide the 20-bit operands into five 4-bit groups. According to the ARM's opcode, the operands generation unit resembles these groups and fits them into proper locations.
- Java operand stack pointer maintenance: This unit is used to maintain the TOSR and operand stack overflow/underflow as described in section III.B.

V. ARM Instructions Folding

The stack-based architecture of the JVM uses a large amount of data move instructions between operand stack and local variables. This also indicates that many register move instructions will be translated and executed by the ARM core. In this section, we will propose an ARM instructions folding technique with a translated queue to detect whether the translated register move instructions can be combined into instructions that truly operate in execution unit.

A. *An Example for ARM Instructions Folding*

As shown in Fig. 6, three data move bytecode instructions (`iload_` , `iload_0` and `istore_2`) are translated into ARM's register move instructions. The ARM instructions folding mechanism checks the true data dependency among the translated ARM instructions. If the dependency is found between register move instructions and ALU type instruction (`add`), the ARM instructions folding is applied to remove the register move instructions and adjusts the source/destination register for the ALU type instruction.

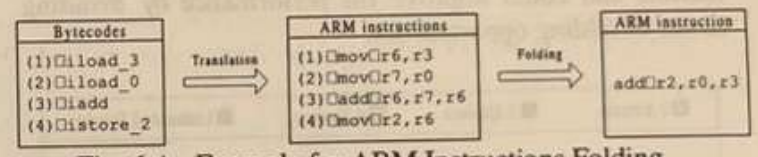

Fig. 6 An Example for ARM Instructions Folding

B. *ARM Instructions Folding Mechanism*

As listed in table 2, the translated ARM instructions are classified into three types for folding check. The folding occurs when one or two continuous P instructions are followed by an E instruction, or an E instruction is followed by a P instruction.

TABLE II
Classification of Translated ARM Instructions for Folding Check

Types	Description
Primary (P)	ALU operations
Erasable (E)	register data move
Independent (I)	memory load/store, branch and software interrupts

To implement the mechanism, we use a translated queue (TQ) to store the translated instructions. The folding check is performed before the newly translated instructions are written into the TQ. As described in section IV.A, one bytecode instruction may be translated into one or more ARM instructions. If more than one ARM instructions are translated, only the first translated ARM instruction will be

checked with the latest instructions among TQ, as depicted in Fig. 7.

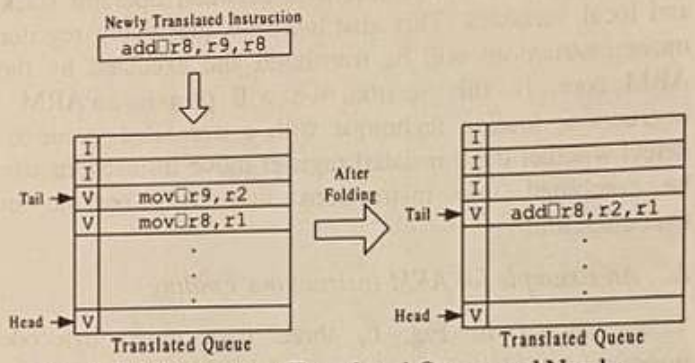

Fig. 7 Folding among Translated Queue and Newly Translated Instructions

C. Simulation Results

Fig. 8 shows the relationship between the number of TQ entries and the folding speedup. Simulation result show that only 4 entries are enough for TQ design. Consequently, using a fully associative orginazation for the TQ design i feasible and could improve the performance by avoiding losses of folding opportunities.

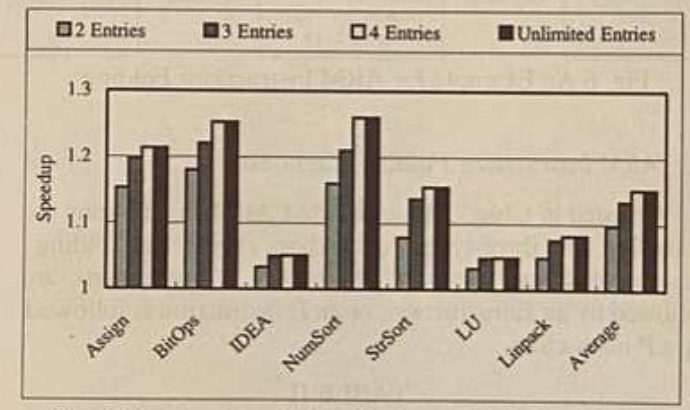

Fig. 8 Comparison of TQ Entries and Folding Speedup

Fig. 9 shows the performance comparison based on the translator with 16 registers. The ARM instructions folding can enhance the execution performance by 15%. If the number of ARM register is unlimited and ARM instructions folding is not used, the performance gain is 37% higher than the 16 registers configuration. If the ARM instructions folding is enabled with unlimited number of registers, the performance gain is 75% higher than the 16 register configuration.

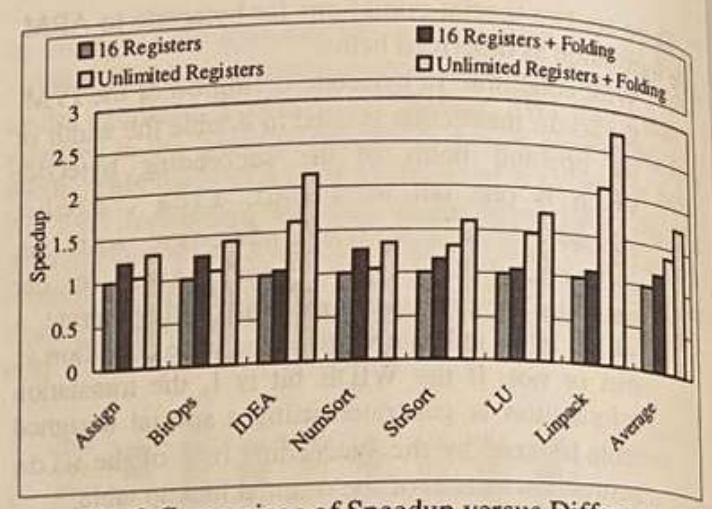

Fig. 9 Comparison of Speedup versus Different Architectural Design Parameters

VI. Conclusions

The proposed Java/ARM dual-mode processor is used to accelerate the ARM's execution performance of Java programs while compatible with existing C coded applications. When executing the Java applets/applications, the bytecode instructions can be executed by the dual-mode processor on-the-fly. A simple Java-to-ARM hardware code translator is designed and built in between the ARM instruction fetcher and decoder. In the Java execution mode, the Java-to-ARM translator can translate one ARM instruction per cycle. 83% of Java bytecode instructions are translated by the Java-to-ARM translator to execute directl on the ARM core. The remaining 17% complex bytecode instructions are implemented as ARM's software interrupt that require ARM's operating system supporting.

The Stack to Register Mapping Mechanism (SRMM) i introduced for Java operand stack to ARM register mapping. Simulation shows that the mapping combination for 6 entries of operand stack and 5 entries of local variable gets least memory swapping overhead. The execution performance of the translated ARM codes can be further enhanced by the ARM instructions folding techniques. Simulation shows that 15% performance gain is achieved for the 16 registers ARM core. If the number of registers is unlimited, 38% or 75% performance gain can be achieved without or with ARM instructions folding, respectively

The concept of the Java-to-ARM binary translator can be applied to other core processor architectures. Higher performance superscalar or VLIW core using DIF or dynamic scheduling techniques is now studying in our research lab. Furthermore, a dynamic folding technique with simple hardware implementation that can eliminate 99.16% data move instructions is recently developed [TON 00]. We believe that more and more dual-mode even multi-mode execution architectures will be proposed with sophisticated binary translation techniques in the future.

VII. References

[ARM 95] ARM Corporation, "ARM7TDMI Data Sheet," August 1995.

[CHR 97] CHRISTIAN Ludloff. IA64 Speculations. http://www.sandpile.org/IA64.html, November 1997.

[GRA 97] GRAY N. Hammond, et al. Processor Capable of Executing Programs That Contain RISC and CISC Instructions. *US Patent US5638525*, June 1997.

[HSI 96] C.-H. A. Hsieh, J. C. Gyllenhaal and W. Mei. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. *Proceedings of the 29 th Annual International Symposium on Microarchitecture (MICRO-29)*, December 1996, pp. 90-97.

[JIMD 98] JIM Davis. IBM Eyes Java Chips. http://www.news.com/, March 1998.

[JIMT 97] JIM Turley. Java Not to Everyone's Taste. *Microprocessor Report*, vol. 11, no. 6, May 1997.

[KEM 97] E. Kemal, A. Erik and H. Erdem. A Java ILP Machine Based on Fast Dynamic Compilation. *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[LIN 97] Linley Gwennap. First Merced Patent Surfaces. *Microprocessor Report*, March 1997.

[LIN 97] LINLEY Gwennap. Intel HP Make EPIC Disclosure. *Microprocessor Report*, vol. 11, no. 14, October 1997.

[O'CO 97] J. M. O'Connor and M. Tremblay. picoJava-I: The Java Virtual Machine in Hardware. *IEEE Micro*, March/April 1997, pp. 45-53.

[PET 96] PETER Christy. IA-64 and Merced -- What and Why. *Microprocessor Report*, vol. 10, no. 17, December 1996.

[RICK 98] RICK COOK. Java Embeds Itself in the Control Market. http://www.javaworld.com/, January 1998.

[RON 98] Ron Curry. IA-64 Architecture. http://developer.intel.com/.

[SUN 99] SUN Microsystems Inc. picoJava-II ™ Microarchitecture Guide. March 1999.

[TON 00] Lee-Ren Ton, Lung-Chung Chang and Chung-Ping Chung. Exploiting Java Bytecode Parallelism b Enhanced POC Folding Model. Proceedings of the Euro-Par 2000, August 2000, Murnich Germany