A Java Processor IP Design for Embedded SoC

CHUN-JEN TSAI, National Chiao Tung University
HAN-WEN KUO, National Chiao Tung University
ZIGANG LIN, National Chiao Tung University
ZI-JING GUO, National Chiao Tung University
JUN-FU WANG, National Chiao Tung University

In this paper, we present a reusable Java processor IP for application processors of embedded systems. For the Java microarchitecture, we propose a low-cost stack memory design that supports a two-fold instruction folding pipeline and a low-complexity Java exception handling hardware. We also propose a mapping between the Java dynamic class-loading model and the SoC platform-based design principle so that the Java core can be encapsulated as a reusable IP. To achieve this goal, a two-level method area with two on-chip circular buffers is proposed as an interface between the RISC core and the Java core. The proposed architecture is implemented on a Xilinx Virtex-5 FPGA device. Experimental results show that its performance has some advantages over other Java processors and a Java VM with JIT acceleration on a PowerPC platform.

Categories and Subject Descriptors: **C.3 [Computer Systems Organization]**: Special-Purpose and Application-Based Systems --- Real-time and embedded systems; D.3.4 [Programming Languages]: Processors—Run-time environments; B.7.1 [Integrated Circuits]: Types and Design Styles—Microprocessors and microcomputers

General Terms: Design, Experimentation, Performance.

Additional Key Words and Phrases: Java accelerator, application processor SoC, dynamic class loading, embedded systems.

1. INTRODUCTION

Platform-independence makes the Java programming model attractive to embedded system programmers. In the past, embedded systems are usually designed using proprietary architecture that makes it very difficult to create portable applications across different devices. As a result, in order to subscribe to the value-add telecomm and broadcast services, users are forced to purchase customized client-provided equipment (CPE). This industry practice has hindered the promotion of new multimedia services significantly. In order to enable fast adoptions of new user-centric multimedia services, new generations of smart consumers' electronics, such as the DVB set-top boxes, Blu-ray players, and mobile phones, often adopt the Java runtime environment (JRE) as their application platforms.

There are many variations of embedded JRE, including Sun's CDC/PBP, CLDC/MIDP and Google's Android platform. Most existing implementations are software-centric, which means they require a sophisticated operating system (OS) to support the JRE (e.g., the reference implementations of the Java ME and the Android platforms heavily depends on Linux or OS's with similar capabilities). In general, operating systems handle thread management, memory management, I/O interfaces, and dynamic class loading from local and/or remote file systems for the JRE. However, most Java middleware stacks of JREs have already included main functions of a typical OS kernel. Therefore, adopting a complete OS underneath a JRE is a duplication of system functions, which is not a good design philosophy for embedded devices with resource constraints.

This research is funded by National Science Council, Taiwan under grant number NSC 100-2221-E-009-052-MY3. Author's address: Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. C.-J. Tsai (corresponding author); email: cjtsai@cs.nctu.edu.tw.

1.1 The Java OS Model for Java Processor IP Design

A different design approach, the JavaOS model [Ritchie 1997; Montague 1997], was proposed to implement the Java application platform. In this approach, the OS itself is written in the Java language as part of the JRE. The JavaOS system model uses a small microkernel written in native codes to support low-level physical resource management. The majority of the OS itself and all the applications are written in Java. The concept of the JavaOS model was not adopted widely for JRE on high performance computers since resource limitation is not a major concern for such platforms. However, with the emerging of new embedded applications and application processor SoCs, it is time to revisit the concept again. Although the JavaOS idea of using minimal native C codes to support the complete JRE was originally proposed for software-based virtual machines (VM), the concept is nevertheless very suitable for the encapsulation of a hardware Java core as a reusable IP for application processor SoCs.

The SoC software stack shown in Figure 1 illustrates this concept. For the JRE, the RISC-side software only handles the low-level I/O-related tasks. The Java core handles the execution of the rest of the JRE software stacks, which can be encapsulated inside a reusable IP. Typical service tasks include I/O controller management (i.e., management of device drivers), file system accesses, network communication tasks, and management of application accelerators (e.g. audio/video codecs). In short, the RISC-side of system software provides a hardware abstraction layer (HAL) for the Java platform. Some researchers may feel that adopting a RISC core for the sole reason of implementing a HAL for the Java core might not be cost-effective and the RISC core should be replaced by a dedicated hardware block. Nevertheless, the inclusion of a RISC core does facilitate the support of the legacy I/O controllers. We will provide some other rationales on why adopting a RISC core in a Java-based application processor can be appealing in section 1.2.

Note that in the proposed model in Figure 1, the RISC core does not execute any bytecode instructions. Although one can choose to implement some bytecodes on the RISC side, the inter-processor communication (IPC) overhead will significantly hinder the performance of the system. It is also important to point out that the JRE system model proposed in Figure 1 does not prohibit the adoption of a sophisticated OS kernel on the RISC-side. On the contrary, because this Java system model has little dependency on any particular OS kernels, it makes it easy to integrate such model into existing embedded systems. We have presented a Java SoC that does not

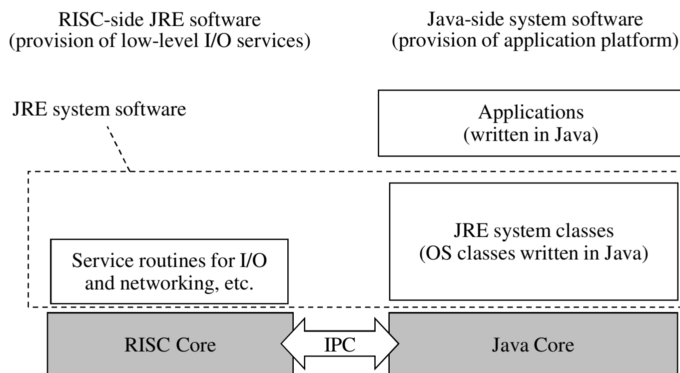


Fig. 1. The adoption of the 'JavaOS' system model for a JRE encapsulated inside a reusable IP. The RISC-side system software is minimized to facilitate the integration of the Java core into the existing application processors.

require any operating systems to support the execution of Java GUI programs in Hwang et al. [2010]. The SoC adopts a heterogeneous dual-core design that is composed of a generic RISC core and a double-issue Java bytecode execution engine [Ko and Tsai 2007]. However, the Java SoC in Hwang et al. [2010] does not support dynamic class loading and method area management. All the Java classes of an application must be converted to a proprietary runtime image format offline and loaded into the on-chip memory before the execution of an application begins. Runtime support of the Java class file format for dynamic linking and loading is a crucial part of the Java VM model and contributes to its claim of platform independence [Venner 2000]. Binary portability based on the Java class file specification is one of the reasons why Java ME was adopted by organizations such as the 3GPP and DVB. Although some new platforms such as the Android JRE do not follow this policy, the concept of a single portable binary execution format should be honored to promote an open third-party application market.

1.2 Why Using a RISC Core in a Hardwired Java SoC

In the proposed SoC architecture, a RISC core is used to provide some low-level services. If the RISC core is only used for the implementation of a HAL for the Java core, it may not seem cost-effective. However, this is not true when the support of dynamic loading of the standard Java class files is required. As we have mentioned in section 1.1, supporting the Java class files at runtime is an essential key to Java's platform independence. However, a Java class file is not suitable for efficient direct execution by a hardwired Java processor. To support the standard Java class files at runtime, a Java processor must convert a class file to a runtime format on-the-fly. The key question now becomes whether it would be more efficient to use a dedicated circuit to support runtime file conversion and HAL or to use a RISC core for the same tasks.

Class file conversion and HAL support can be done very efficiently using a small register-based RISC core. Take the RISC core we use in the proposed Java SoC for example. It only takes 1927 FPGA LUT6s [1] to implement the 5-stage pipeline Microblaze core. According to our experiences, a dedicated circuit that performs similar tasks can easily use more FPGA resources than the RISC core does. Although the dedicated circuit will definitely be much faster in class file conversion and possibly more efficient in I/O management, such gain will not improve the overall Java execution performance much. Since each Java class file is only converted once throughout the life cycle of an application program, the performance of conversion is not crucial. In addition, standard system classes of the Java OS can be converted in advance so the performance of the converter is even less important. In fact, we have tested two different RISC cores with 4X performance differences in the proposed Java SoC and seen only negligible differences in the benchmark results.

The second reason for using a RISC core for dynamic class loading and I/O devices management for the Java core is that it is more flexible in adopting new class file formats and new I/O devices since the tasks are implemented in C code. Furthermore, it would be more cost-effective to use a single RISC core to provide services to multiple Java cores for multicore systems. If a dedicated circuit for class file conversion and I/O management were integrated into the Java core, a multicore Java SoC would waste circuit resources on duplicated file conversion hardware for limited performance gain.

[1] LUT6 means a lookup table with 6 inputs in a Xilinx FPGA logic cell.

Finally, evidences show that if we perform some bytecode-level optimization at class loading time, the performance of a Java processor can be increased significantly [Tyystjaervi et al. 2010]. Such design explorations can achieved with a RISC core-based class loader without any extra hardware cost.

1.3 Contribution and Organization of the Paper

In this paper, we propose an architecture design that encapsulates a Java core as a reusable IP. The Java core is referred to as the Java application IP (JAIP). The key idea is to design a flexible two-level method area module as the Java core interface. The method area manager connects to the on-chip system bus so that the RISC core can access the Java IP using a memory-mapped interface similar to those used in the multimedia codec accelerators. Such interface is different from the ARM Jazelle approach [Porthouse 2005] where the co-processor interface exclusive to the ARM architecture is used. In the proposed design, the dynamic class loading-and-linking model of the Java language is adopted as the abstraction scheme for the RISC core to request accelerated execution of Java classes from the Java core. To reduce the software effort of integrating JAIP into the existing application processor SoCs, the proposed system minimizes the software dependencies on the RISC side. That is, the only system software support required from the RISC core is the installation of some interrupt service routines (ISRs) for operations related to dynamic class loading and I/O tasks. The main contributions of this work can be summarized as follows.

— We propose to apply the dynamic class loading and linking model of the Java language to encapsulate the Java core as a reusable IP such that the integration of the Java IP into an application processor follows the platform-based design principle of SoCs. The key contribution is the architecture of the two-level method area design. In addition, the proposed SoC can be used as an ideal platform for the original JavaOS model proposed in Ritchie [1997].

— We propose a Java stack architecture that is constructed using interleaving on-chip dual-port memory and seven 32-bit registers to facilitate two-fold instruction folding of frequent bytecode instruction pairs. The proposed design makes hazard detection quite easy and significantly reduces the logic complexity of the two-fold instruction folding controller.

— We present the design of a Java exception handling circuit which supports call-stack unwinding and exception object inheritance. To our knowledge, there is no previous publication that provides detail information on the design of Java exception handling hardware support to this degree. Some descriptions on Java trap implementation can be found in Puffitsch and Schoeberl [2007], but it did not provide enough details or overhead evaluations. In this paper, we conduct some experiments to show that hardware-based exception handling can be done very efficiently (compared to a software-based VM) using little hardware resources.

— We conduct some subtle investigations on the performance differences between a Java VM with the just-in-time (JIT) acceleration scheme and the proposed Java processor. In particular, we present the impact of JIT compilation overhead on real-word performance of Java applications that are often overlooked in previous studies. Based on the experiments, one can see that the proposed JAIP has some advantages over JIT for embedded applications.

The organization of the paper is as follows. Section 2 presents some previous work on Java processor designs. Section 3 is an overview to the proposed JAIP. Section 4 describes the details of the microarchitecture of JAIP, including the two-level stack

architecture, the two-fold instruction folding policy, and the exception handling mechanism. Section 5 presents the dynamic class loading module and the related architecture, including the design of the two-level method area and the IPC mechanism. Section 6 conducts the performance evaluation of JAIP. Finally, some conclusions and future work are given in section 7.

2. RELATED WORK

There are many publications on the design of stand-alone Java processors [McGhan and O'Connor 1998; Vijaykrishnan et al. 1998; El-Kharashi et al. 2003; Pitter and Schoeberl 2010]. Most of the researches focus on how to implement object-oriented features of the Java language efficiently. Less attention has been dedicated to the study of interface design between a Java processor core and a general-purpose processor core. The most well studied objected-oriented features in the Java language include the schemes for method and interface invocations [Alpern et al. 2001; Preusser et al. 2007], the dynamic optimization of object accesses [Duesterwald 2005], and the garbage collectors [Schoeberl and Puffitsch 2010]. In addition, due to the objected-oriented programming practices, the locality behavior of the method bytecode execution is quite different from that of executing imperative programs on a register-based processor. Instruction cache and object data cache customization for object-oriented behaviors are also crucial to the performance of a Java platform [Vijaykrishnan et al. 1998].

One interesting work that has some similarity to JAIP is the REALJava VM [Tyystjaervi et al. 2010; Saentti 2008]. It is similar to JAIP in a way that it is also intended to be used in an SoC where many Java IPs can form a multicore JRE for better performance. However, there is a fundamental design difference between JAIP and REALJava. REALJava adopts a hardware-software co-design approach where the host processor takes more responsibilities in executing Java tasks. For example, the memory management and thread management is handled by the host CPU in REALJava. For JAIP, everything related to the execution of the Java programs except for dynamic class loading and I/O are handled by the Java core alone, without any help from the host processor. In our previous work, we have tried the co-design approach; but eventually gave it up due to high IPC overhead. For the rest of this section, we provide an overview to related work on the design of the host processor interface and the Java stack that supports instruction-level parallelism.

2.1 Method Area and Dynamic Class Loading Designs

The method area is a crucial component of the Java virtual machine (JVM) model [Venner 2000]. A JVM accesses the method area for fetching method bytecodes, retrieving symbol information, and reading constant values. For a Java VM, the design of the method area is not specified in the Java VM specification. For a Java processor IP, since the host processor can trigger the loading of class files into the method area. It is not only internal to the Java core but should also be taken into account as part of the host-processor interface.

Although the method area is a key component for a Java processor IP to be efficiently used in SoCs, there are hardly any publications providing details on the method area design for scalable, reusable IPs. There are systems that combine a Java core with a host processor core for accelerated Java execution [Kimura et al. 2002; Kent and Serra 2002; Yen and Liang 2009], but these systems still focus on the design of the high performance bytecode execution engine. There is no detail discussion on either the host interface design or the dynamic class loading

mechanism. In Kimura et al. [2002], it describes the host interface as a logic connected to the system bus and controls the transfer of data between the Java internal memory and the shared memory. The host interface also sends interrupts to the host processor whenever there are I/O requests from the Java core. In Kent and Serra [2002], the host interface was described as being involved in retrieving instructions and data (from external memory) as well as handshaking with the software partition in performing context switching. However, according to our experiences, if the Java core requires a host processor to handle context switching, the overhead would be quite high. In Yen and Liang [2009], the CPU interface is described as having the same interface as an asynchronous SRAM. The Java accelerator attached behind the CPU interface is architected as a filter between the host processor and the external memory. However, there is neither detail on the class-loading subsystem nor its interaction with the method area controller.

For new generations of smart embedded devices (e.g. smart phones and smart TVs), the dynamic class loading capability is certainly a desirable feature of a JRE. However, it poses some issues to a Java processor and has not been treated thoroughly in previous publications. There are fundamentally two different class loading/resolution strategies for JREs: eager resolution and late resolution [Venner 2000]. For eager resolution, all classes that may potentially be referenced by a target class file will be loaded before the execution of the target class. However, some of the referenced classes may not be physically used during runtime. For late resolution, a class is only loaded when it is used at runtime. Since late resolution requires smaller memory footprint and lower (and smoother) overall class loading overhead, it is preferable to adopt this approach for embedded systems. Ideally, one may want to implement the dynamic class loading and symbol resolution circuitry completely inside the Java core. However, a class loading operation involves searching the class files among file systems, parsing the class file, and constructing the lookup tables for the use of dynamic resolutions. As mentioned in section 1.2, a register-based general-purpose processor can execute these tasks with less circuit resources, and would be a better choice than a dedicated circuit as long as the efficiency of the class loading operations does not hinder the overall performance.

2.2 Java Stack Architecture Design

The Java VM model adopts a stack machine model where all the intermediate results of computations, as well as the local variables, are stored on the stack. Therefore, the limitation in the number of read/write ports of the stack memory device often becomes a bottleneck. This is particularly true for the bytecode execution techniques that explore ILP [Radhakrishnan et al. 2000b]. To resolve such bottleneck, some Java processors [Sun 1999; Yen and Liang 2009] uses a register file of 64 entries as a stack cache. The JEM core from aJile [Hardin 2001] contains a stack cache with 24 register entries, where only six of them are used to cache the top elements of the stack. For stack cache-based architecture, concurrent background spill and fill operations are often used to keep the stack cache consistent with the top entries of the stack.

Similarly, Komodo [Brinkschulte 1999] and FemtoJava [Ito 2001] use a three-port (two reads, one write) on-chip memory as a large stack cache. However, for low cost, embedded applications, large register file or general purposes multi-port memory are often too expensive. The Java optimized processor (JOP) [Schoberl 2005] proposes an interesting two-level stack design. It uses two registers to store the top two elements of the stack, and a dual-port SRAM to store the rest of the stack elements. This design is less expensive than the stack cache approach and uses only one dual-port

SRAM. However, such Java stack design would be difficult to explore ILP. To enable straightforward double-issue of Java bytecodes without resorting to complex instruction-folding techniques, we have proposed a three-register variant of the two-level stack cache architecture [Ko and Tsai 2007]. However, when on-chip memory is used as the second level stack memory, accesses to local variables could cause structure hazard and prohibit concurrent execution of some frequent instruction pairs. An improved version of the two-level stack design will be presented in section 4.2 to facilitate two-fold instruction folding.

3. OVERVIEW OF THE PROPOSED ARCHITECTURE

Figure 2 illustrates an application processor SoC with the proposed Java core, JAIP. The SoC is composed of two processor cores: a generic RISC-core that handles the dynamic class loading and I/O requests for the JRE, and a Java core that is responsible for the execution of Java applications. The IPC between JAIP and the RISC core is achieved using two mechanisms. The first one is an interrupt-driven mailbox device for low-bandwidth control data exchange and the second one is a special interface module, the method area manager unit (MAMU). The mailbox device allows the Java core to trigger the execution of an ISR routine on the RISC side, and uses a dedicated set of mailbox registers to pass parameters to the ISR. Due to high overhead of interrupt-based communication between the two processor cores, this IPC mechanism is only used by JAIP to request the I/O or dynamic class loading services from the RISC core.

The Java core microarchitecture proposed in this paper is composed of six circuit components (see Figure 2), including a two-fold instruction folding bytecode execution engine, a mailbox device, a two-level Java stack, a dynamic resolution controller, a method area manager, and an object cache controller for caching of the Java heap. The actual heap space is located in the external DDR2-DRAM. The method area manager is similar to the instruction cache controller for other general-purpose

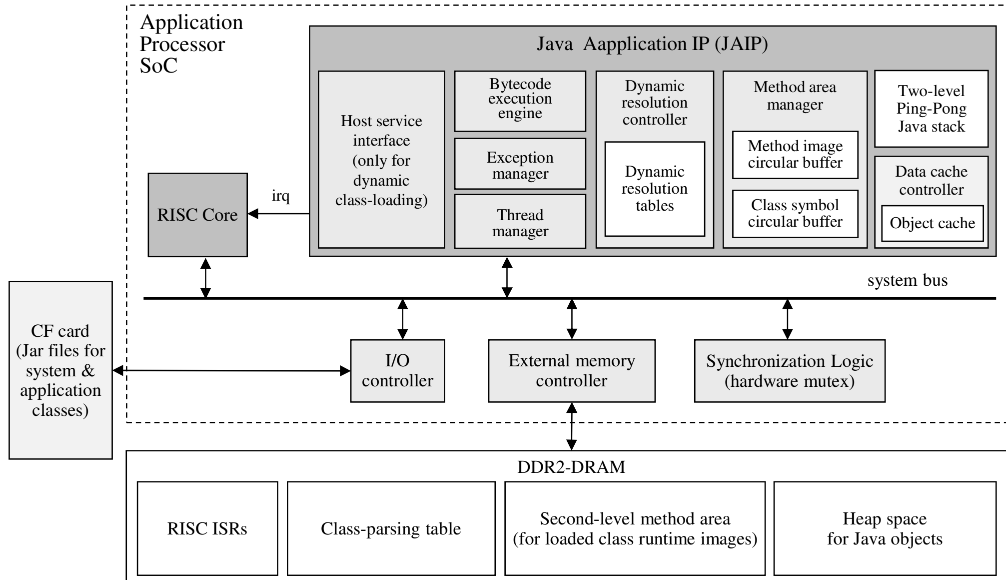


Fig. 2. An application processor SoC with the proposed JAIP.

processors. It caches the method images (and related symbol information) in the second-level method area. The object cache controller is essentially a two-way set associative cache that allows efficient accesses to the runtime Java objects stored in the heap space. Each cache line in the object cache is 32 bytes. The allocation of a heap object is triggered by the execution of the Java bytecode 'new,' 'newarray,' 'anewarray,' or 'multianewarray.' Note that arrays of object references are also stored in the heap space and managed by the object cache controller in our current implementation. Currently, JAIP does not perform runtime garbage collection during the life cycle of a Java program. On our implementation platform, the heap space is set to 32 MB while the total DRAM size is 256MB.

In the proposed JRE model, the method area is the key interface component between the RISC core and the Java core. A class is located and loaded into the method area before it can be executed. We adopt a two-level memory hierarchy of the method area and the late resolution policy of class loading for efficient usage of the on-chip method area. The detail of the method area design will be discussed in section 5.1.

4. BYTECODE EXECUTION ENGINE OF THE PROPOSED JAVA CORE

4.1 Overview of the Pipeline Architecture

Figure 3 shows the microarchitecture of the Java core. The Java core has four pipeline stages: translate, fetch, decode, and execute. Since the Java VM model is a stack machine, it must perform frequent accesses to the operand stack for the intermediate values of computations. One way to eliminate excessive stack operations is to apply the instruction-folding mechanism [McGhan and O'Connor 1998] to combine several instructions into one operation. However, a general instruction-folding controller that allows folding of variable number of instructions is quite complex [Chang et al. 1998] and requires possibly extra pipeline stages [El-Kharashi et al. 2001]. The proposed bytecode execution engine in this paper adopts a simplified two-fold instruction folding architecture. The instruction folding policy and the hazard detection mechanism will be presented in section 4.3. The design goal for this simplified two-fold JAIP engine is to explore instruction-level parallelism with a small increase in control logic complexity.

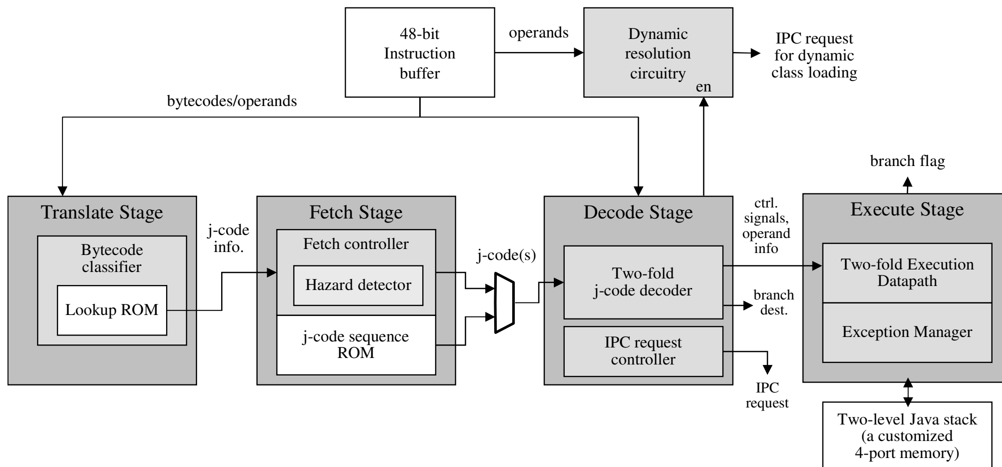


Fig. 3. The architecture of the bytecode execution engine.

The two-fold instruction folding scheme is not performed at the Java bytecode level. The translate stage of JAIP will fetch the Java bytecodes and translate them to native instruction codes (referred to as the j-codes in this paper). Unlike Sun's picoJava [McGhan and O'Connor 1998] where some of the native instructions are implemented using a microcode sequencer, JAIP is hard-wired to execute all j-code instructions. However, a single Java bytecode instruction may be translated into a single j-code or a short j-code sequence, similar to the designs in picoJava and JOP [Schoberl 2008]. The ISA of the j-code contains the following types of instructions: stack manipulations, ALU operations, heap access instructions, control operations, and instructions for accessing bytecode operands in the 48-bit instruction buffer. The design of the j-code ISA is still based on the stack machine model. Nevertheless, all j-code instructions are single-byte instructions without any operands.

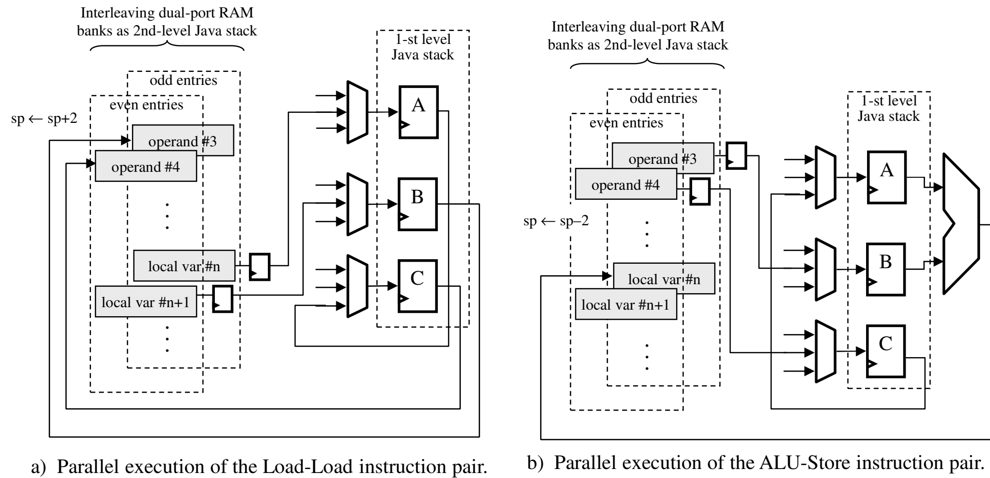
The 48-bit instruction buffer shown in Figure 3 is mainly used to cope with the variable-length nature of the Java ISA and used as a temporary store of the Java bytecode operands for implicit accesses by the j-code (since all j-codes are single-byte instructions without operand fields). The control logic of JAIP will fill the instruction buffer with data fetched from the on-chip method area for the translate logic to determine how they will be translated into j-codes. In addition, the control logic may perform a special flush-and-fill operation for the two bytecode instructions 'table-switch' and 'lookup-switch'. After the decoding of a switch bytecode, the pipeline will be stalled until the flush-and-fill operation of the instruction buffer scans through the 'jump table' following the switch bytecode and locates the target offset of the local jump. After that, the pipeline will be flushed and the jump offset will be loaded into the Java bytecode program counter 'jpc'.

In the translate stage, the controller will check the incoming three bytes in the instruction buffer and classify each incoming byte into one of the three cases: a simple Java bytecode, a complex Java bytecode, or an operand byte. A simple bytecode means that there is a matching j-code in the proposed ISA that can perform the operation. A complex bytecode means that its operation is implemented using a sequence of j-codes stored in the on-chip ROM. By checking the first three incoming bytes in the instruction buffer, the translate stage will prepare a j-code information signal that contains the information of up to two j-codes to the fetch stage.

The fetch stage is in charge of sending two j-code instructions per clock cycle to the decode stage. The two j-codes can come directly from the j-code information signal of the translate stage, or fetched from the j-code sequence ROM if the j-code information signal specifies a complex bytecode. Before issuing two j-codes to the decode stage, the hazard detector will check whether the two j-codes can be executed simultaneously or not. If not, the two j-codes will be separated and each j-code will be paired up with a 'nop' instruction for the decode-and-execute of the next two cycles. Note that in Figure 3, the hazard detector does not check the j-code pairs fetched from the j-code sequence ROM. Therefore, the designer of the j-code sequence ROM is responsible for making sure all j-code pairs in the ROM can be executed concurrently by properly inserting 'nop' instructions manually. The decode stage will then decode the two j-code instructions and setup the datapath control signals accordingly. Finally, the execute stage performs the intended operations.

4.2 The Proposed Java Stack Architecture

The proposed JAIP performs instruction folding at the j-code level. In order to fold two stack operations into one, simultaneous stack accesses are necessary. Therefore, large stack caches are often used for high performance Java processors [McGhan and



a) Parallel execution of the Load-Load instruction pair. b) Parallel execution of the ALU-Store instruction pair.

Fig. 4. Instruction folding for the execution of a Load-Load-ALU-Store bytecode sequence in two cycles. Registers A, B, and C store the operands #0 (top of stack), #1, and #2, respectively. The stack pointer (sp) points to operand #3.

O'Connor 1998; Yan and Liang 2009]. To reduce the implementation cost for embedded applications, we have extended the two-level stack idea proposed in Schoeberl [2005] and presented a new architecture in Ko and Tsai [2007] such that two-fold instruction folding of frequent bytecode pairs can be achieved without an expensive stack cache. However, the design in Ko and Tsai [2007] are constrained by the adoption of a simple two-port memory for the second-level stack such that certain frequent load/store operations will cause structure hazards. In addition, the hazard detection logic is complex. In this paper, we adopt a new design of the second-level stack memory to improve the folding rate and simplify the hazard detection logic.

In Vijaykrishnan et al. [1998], they analyze frequent bytecode sequences based on a large set of programs. The study shows that the bytecode sequences of Load-Load-Arithmetic, Load-Arithmetic-Store, and Arithmetic-Store are among the most frequent code sequences in Java applications. The design in Ko and Tsai [2007] uses three registers to store the top-three stack elements exclusively (shown as A, B, and C in Figure 4). The rest of the stack elements are stored in a pair of interleaved on-chip memory banks. Such architecture enables folding of some stack operations mentioned above. For example, the bytecode sequence Load-Load-Arithmetic-Store can be executed in two cycles. The bytecode execution pipeline can execute the first two load instructions in parallel using the datapath in Figure 4 (a) if the two local variables are stored in different banks. The succeeding Arithmetic and Store instructions can be executed in parallel using the datapath in Figure 4 (b).

The two-level stack design in Figure 4 encounters structure hazard whenever the j-code instruction pairs try to transfer two local variables stored in the same memory bank to the operand stack (or vice versa). This structure hazard can be removed by using a general-purpose four-port memory for the second-level stack. However, since a general-purpose four-port memory is often expensive [Wang 2010], we proposed a special-purpose 4-port memory customized to the Java ISA to reduce the occurrence of structure hazards while maintaining low implementation cost [Lin et al. 2012].

According to the Java VM specification [Lindholm and Yelling 1999], the first four local variables should be the most frequently used ones (which can be arranged by an optimized Java compiler). Hence, some Java instructions (with no operands) are designed specifically for accessing these variables. The proposed second-level Java

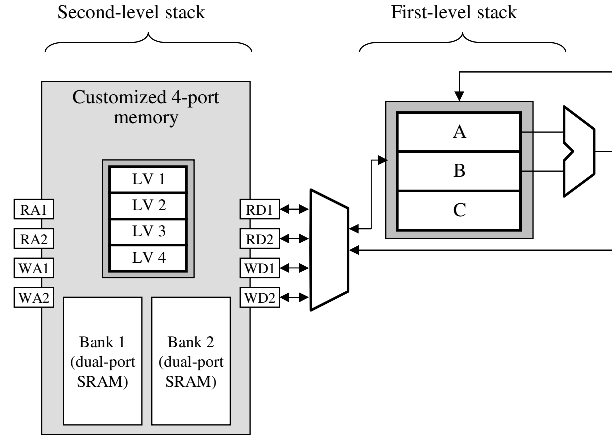


Fig. 5. The proposed two-level stack architecture. LV1 ~ LV4 are registers that cache the first four local variables of the current method. A, B, and C are registers which contain exclusively the top-three stack elements. The synchronization between the LV registers and local variables happens upon the method invocation and return.

stack memory is constructed by using two on-chip memory blocks organized in an interleaving structure. In addition, four 32-bit local variable (LV) registers are used as a small cache for the first four local variables as shown in Figure 5. Upon a method invocation, the first four local variables will be copied from the Java stack to the LV registers. Before the method returns, the LV registers will be copied back to the Java stack. The initialization/restoration of the LV registers only takes one cycle (since each bank has two ports) and is performed in parallel with the dynamic resolution process of method invocation and return such that they do not incur extra overhead. With this new design, the folding of two stack operations often does not cause structure hazard, as will be discussed in section 4.3.

4.3 The Proposed Two-Fold Instruction Folding Mechanism and Hazard Detection

The proposed microarchitecture of JAIP performs two-fold instruction folding of stack-related Java operations using a simple decision policy. In short, JAIP only supports the folding of the eleven stack operation pairs illustrated in Figure 6. Note that in Figure 6, 'Load' means loading a data item on to the operand stack. The source of the data can be from the local variable area of the Java stack or a constant value. 'Store' means removing a data item from the operand stack. The destination of the removed data can be the local variable area or a null space (as in the 'pop' operation). Finally, 'ALU' means an arithmetic and logic operation. The fetch stage of the pipeline will guarantee that, at any given cycle, the j-code information passed to the decode stage belongs to one of the following three cases. First, they can be a stack operation j-code pair shown in Figure 6. Secondly, it can be a single control instruction (such as a conditional branch). Finally, it can be a special data-processing j-code (such as the 'swap' operation).

As mentioned in section 4.1, the fetch stage receives the j-code information signal from the translate stage about the next few instructions in the instruction buffer. If the j-code information represents two simple bytecodes, the proposed folding mechanism will send a j-code pair to the decode stage for concurrent execution if the two simple bytecodes belong to one of the following cases:

— The two simple bytecodes match the j-code pair that belongs to one of the cases from Figure 6(a) to 6(i). There will not be any structure hazard since, in the worst

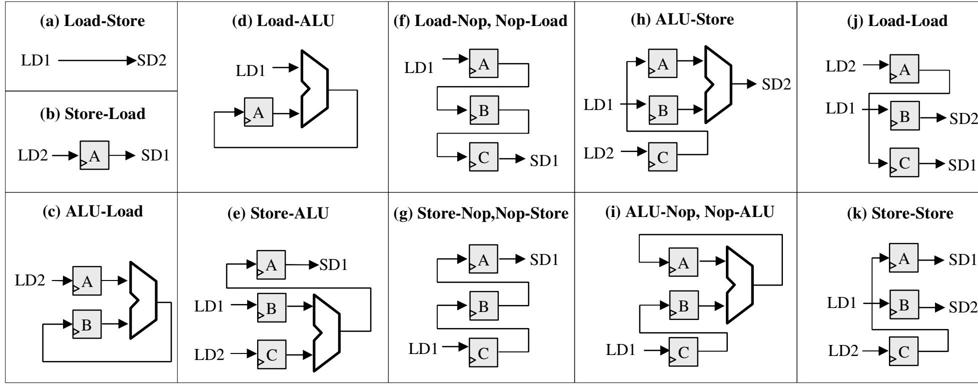


Fig. 6. The datapaths of all supported two-fold instruction pairs. Note that A, B, and C are the registers that store the top-three stack elements. LD/SD represents the load/store data from/to the second-level stack (either the operand stack items or the local variables) or constant values.

case, these datapaths access two consecutive data items in the operand stack and a local variable. The two operand stack items must belong to different memory banks because their addresses are consecutive. The local variable can be accessed together with one of the operand stack item because each memory bank has two ports. Therefore, the datapath can be executed in one cycle without hazard.

— The two simple bytecodes match the j-code pair that belongs to the case in Figure 6(j), and there is at most one of the LD1/LD2 signals representing a local variable beyond the first four. This case will not cause structure hazard since SD1/SD2 are from consecutive addresses of the operand stack.

— The two simple bytecodes match the j-code pair that belongs to the case in Figure 6 (k), and there is at most one of the SD1/SD2 signals representing a local variable beyond the first four. This case will not cause structure hazard since LD1/LD2 are from consecutive addresses of the operand stack.

For any other cases of j-code information, the fetch stage will either generate a j-code pair of a non-foldable instruction plus a 'nop' instruction, or fetches a pair of j-code instructions from the complex bytecode sequence ROM and send them to the decode stage for execution. In the current design of JAIP, we do not allow parallel execution of some simple bytecode pairs such as ALU-ALU, or combining a simple j-code with the first j-code of a complex bytecode sequence to form a j-code pair. These constraints are design decisions to keep the controller logic simple. However, in the future, we may look into ways to optimize the design further without increasing the cost significantly.

4.4 The Exception Handling Mechanism

In the Java programming language, exception handling is a crucial technique for runtime error recovery. Upon a runtime error, the VM or the Java program can throw an exception object which can be caught by a user-defined exception handling routine. The routine will be compiled by the Java compiler as an isolated code segment appended to the end of the method that contains the 'catch' statement of the exception object. The process of finding the correct routine to process the exception object is not a trivial task since the handling routine of the exception may be defined in the caller of the method where the exception event happens. Therefore, a "call stack unwinding" procedure that searches through the call stack sequentially must

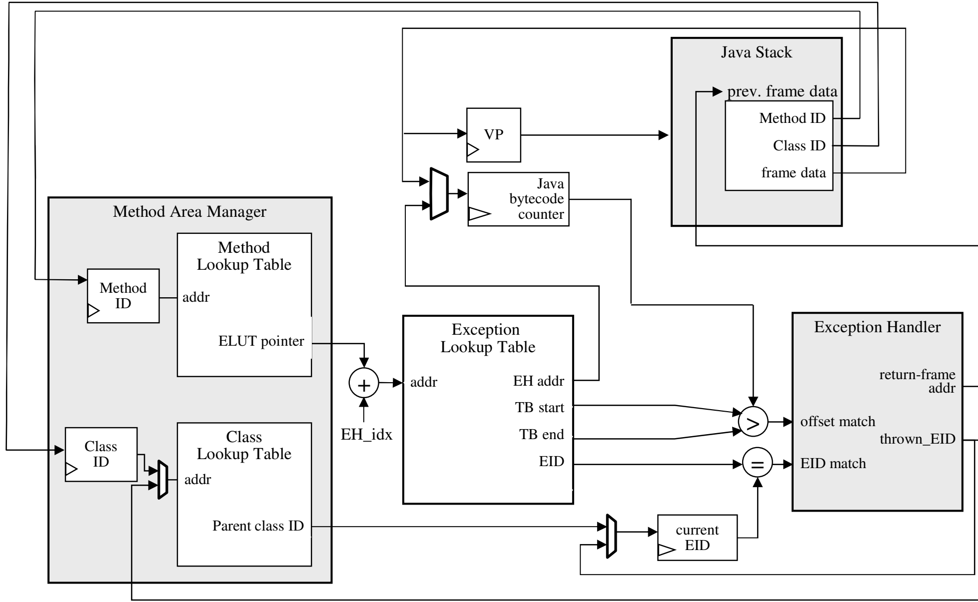


Fig. 7. Architecture of the exception manager.

be performed to locate the target exception routine. In addition, an exception object that was thrown may be processed by a handling routine of the exception object's parent class. Therefore, the class ID of the exception object and all its parent classes' ID must be used for matching the target exception handling routine.

In this section, we describe the exception handling mechanism of the proposed JAIP. The exception manager of JAIP is implemented completely in hardware as shown in Figure 7. For each Java application, the system class loader maintains an exception lookup table (ELUT) in the on-chip memory. Every time a class is loaded, the class loader will parse the exception routine information from the class file and append the information to the ELUT. Each exception handling routine of the application has an entry in the ELUT. Each entry contains four 16-bit values, namely, the starting bytecode offset of the try-block (TB_start), the ending bytecode offset of the try-block (TB_end), the bytecode offset to the exception handler (EH_addr), and the class ID of the exception object (EID). All offsets in the ELUT are defined with respect to the base address of the method runtime image that contains the try-blocks and the exception handlers. In our implementation, the hardware of the exception manager costs about 370 FPGA LUT6s and the ELUT is 2K bytes (supporting up to 256 exception handlers for each Java application).

The exception manager controller can access the entries in the ELUT through the method lookup table (MLUT). Figure 8 shows an example of how the class loader sets up the entries in the ELUT and the MLUT for a Java code segment with two try-blocks in a method. For each method in a loaded class, there is an ELUT pointer in the MLUT. The pointer points to the location of the first exception handler of the method in the ELUT. Another entry, the EH count, in the MLUT tells the controller how many exception handlers does the method contain.

Once an exception occurs, the exception manager will use the class IDs of the exception object and its parent objects to search the ELUT. If the Java bytecode program counter (the 'jpc' register) value falls within a try-block where the exception

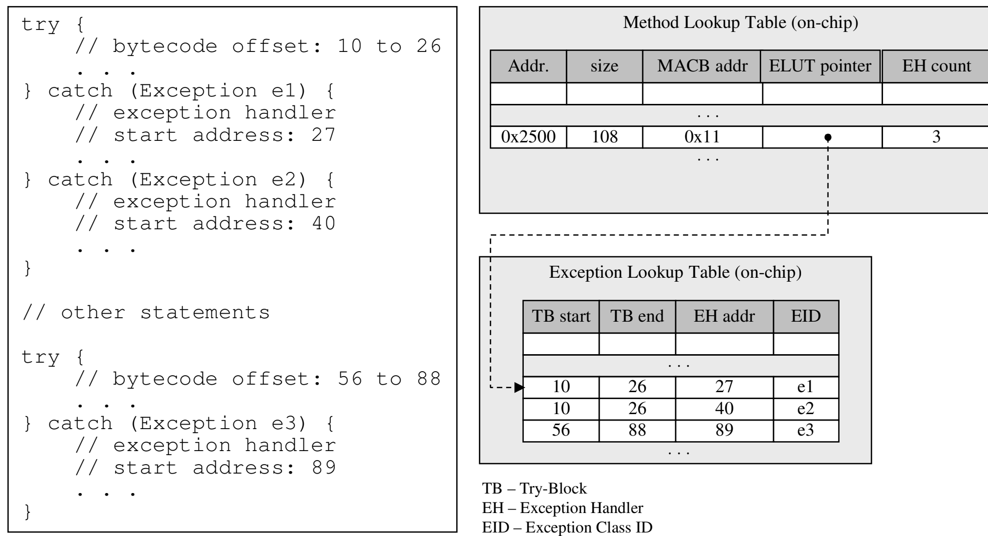


Fig. 8. An example of exceptions in a method and the associated tables.

handler's EID matches either the class ID of the exception object or one of its parents' ID, the EH_addr entry will be used to update the Java bytecode counter. Thus, the control flow will be transferred to the exception handling routine. At the end of the handling routine, bytecodes generated by the Java compiler will transfer the control back to the Java statement after the catch-block. If, however, there is no matching exception handler in the current method, the controller will pop the current stack frame and look for a matching exception handler in the caller method. The controller will keep searching through the methods in the call stack until it finds a matching exception handler, or it will flag an error and abort the Java application.

5. DYNAMIC CLASS LOADING AND RESOLUTION MECHANISM

5.1 Method Area Manager

To encapsulate a Java core as a reusable IP, the key architecture component is the method area manager unit (MAMU) shown in Figure 9. In the proposed framework, the Java method area has a two-level memory hierarchy. The first-level method area contains two on-chip circular buffers called the method image circular buffer (MICB) and the class symbol circular buffer (CSCB). Both circular buffers are composed of 32 256-byte memory blocks that store the dynamically loaded method runtime images and the class symbol tables in a FIFO manner. MICB and CSCB are used as a cache of the second-level method area located in the external DDR-SDRAM memory. MICB caches the most recently executed method images while CSCB caches the most recently referenced class symbol tables. Complete copies of all the loaded class runtime images of a Java application are stored in the second-level method area. The Java core is designed to only fetch the method bytecodes, access the non-string constant values, and lookup the dynamic resolution information from the on-chip MICB and CSCB. If a bytecode tries to reference a method or a class symbol that has not been loaded into MICB or CSCB yet, the method area manager will trigger a loading operation from the class runtime image in the second-level method area to the on-chip circular buffers. Furthermore, if the target class has not been loaded into the second-level method area yet, a dynamic class loading request will be issued to

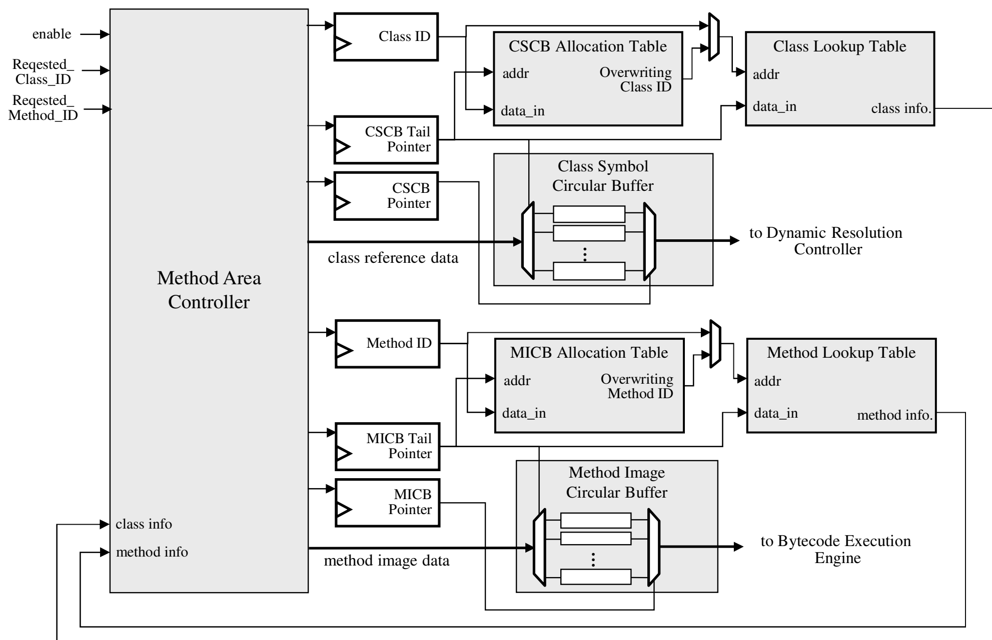


Fig. 9. Architecture of the method area manager.

the RISC core to parse-and-load the new class file into the second-level method area before the cache update operation of MICB or CSCB can proceed.

A method image loaded into the first-level method area occupies one or several contiguous MICB blocks. According to the investigation conducted in Vijaykrishnan et al. [1998], the size of each block in MICB should be small since most of the typical Java methods are quite small. In our implementation, we set it to 256 bytes. A tail-block pointer points to the last block occupied by the last loaded image in the circular buffer. If all the circular buffer blocks are occupied, the next method image to be loaded will be written into the blocks after the tail-block pointer position (with possible wrap-around). This behavior will overwrite the method images currently in these blocks. Although other more sophisticated method replacement techniques (e.g., LRU) can be used, we choose to implement FIFO policy for its simplicity and reasonable performance for embedded applications. The behavior for caching the class symbol tables in CSCB is similar to the behavior of MICB.

For dynamic class loading, the class loader C code of the RISC core will search the class path (set to the jar files stored in a CompactFlash card in our implementation) for the target class file. Once the target class file is located, the loader would parse-and-translate it into a proprietarily defined class runtime image (CRI). The CRI contains only the method bytecodes, partially resolved symbol information, and non-string constant literals of the class (string constants are stored in a class-parsing table in SDRAM). Therefore, the CRI image is usually smaller than the original class file. The newly loaded CRI will then be registered and stored in the second-level method area. The flow chart of dynamic resolution and class loading process is shown in Figure 10. The pipeline of the Java bytecode execution engine is stalled during the dynamic class loading process.

The method area controller relies on an on-chip class information table (which is maintained by the class loader C code) to locate a loaded class or to request the

loading of a previously unreferenced class. In addition, each circular buffer requires two tables: a circular buffer allocation table (a tiny table constructed using registers) and a lookup table (stored in on-chip memory), for normal operations. A list of all the on-chip tables and memory blocks used by JAIP is shown in Table I.

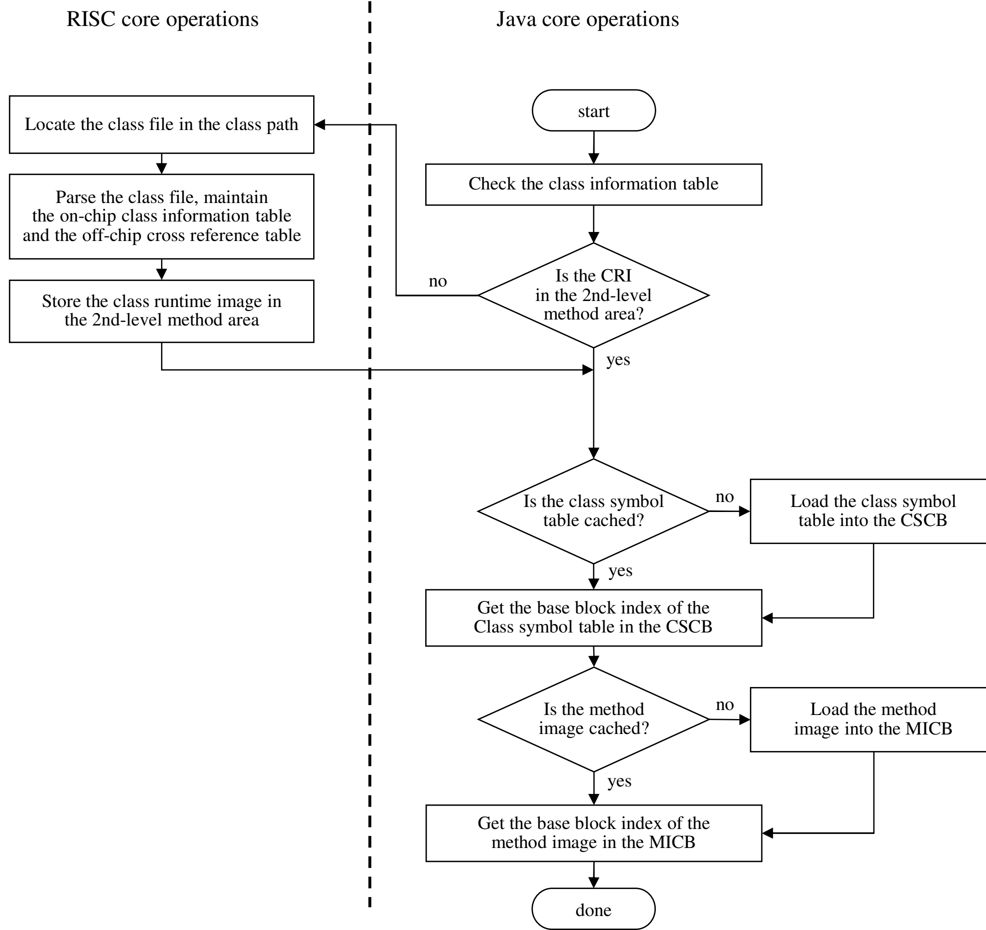


Fig. 10. The dynamic resolution and class-loading process.

Table I. List of the on-chip memory used by JAIP

Name of the table	Size (KB)
Class information table	2
Cross reference table	16
Class symbol circular buffer	8 (32×256 bytes)
Class lookup table	2
Method image circular buffer	8 (32×256 bytes)
Method lookup table	8
Exception lookup table	2
Thread control block table	0.5
j-code sequence ROM	2
Bytecode translation ROM	2
Java stack	4
Object cache	16

5.2 RISC Service Invocation through the IPC mechanism

One of the key goals of JAIP is to ensure easy integration into the existing application processor SoC. The proposed system tries to simplify the integration effort from both the hardware aspect and the software aspect. On the hardware side, the Java core pipeline is encapsulated in a reusable IP and connected to the system bus via standard on-chip bus protocols. Only the MAMU logic is exposed to the RISC core as a slave device. Therefore, at boot time, the RISC core can load the Java boot class into the second-level method area in SDRAM and set up control registers (e.g. the Java program counter and stack registers, etc.), and trigger the Java core for the execution of the boot class. The Java core can execute the loaded class completely by itself, except when it wants to load a new class into the second-level method area or perform I/O operations. As a result, the software aspect of integrating the JAIP into existing systems is to install ISRs under the RISC core to provide such services. Since most operating systems today allow user-installable ISRs, the effort of JAIP integration is quite modularized. There is no need for the RISC-side operating system to provide any other services (such as memory management or scheduling).

Whenever JAIP wants to request dynamic class loading or I/O services from the RISC core, it will use a mailbox device for IPC. In JAIP, there are two ways to invocate RISC services using the mailbox. The first one is through normal invocation of a 'native method' defined in the Java programming specification. The second one is through hardwired invocation from the dynamic resolution module of JAIP.

Invocations of RISC services through a Java-defined native method is compatible the Java specification and are used for services such as terminal I/O's. A Java native method invocation is just like other method invocation except that we have internally maintained a native routine table that tells JAIP to trigger a mailbox device to inform the RISC core to start the service. The mapping of the Java method parameters on the stack to the mailbox IPC parameter registers is shown in Figure 11. The register file of the mailbox is composed of three argument registers and a service ID register. The argument registers are copies of the top-three operand stack elements (stored in registers 'A', 'B', and 'C' in Figure 11). The service ID register tells the RISC which ISR services it should provide. The service ID register will be initialized by the method invocation j-code sequence using the ID information retrieved from the native routine table. In addition to I/O services, this type of native method invocation can also be used for invocations of hardware-accelerated multimedia APIs such as JSR-135 and JSR-184 [JCP 2005; JCP 2006].

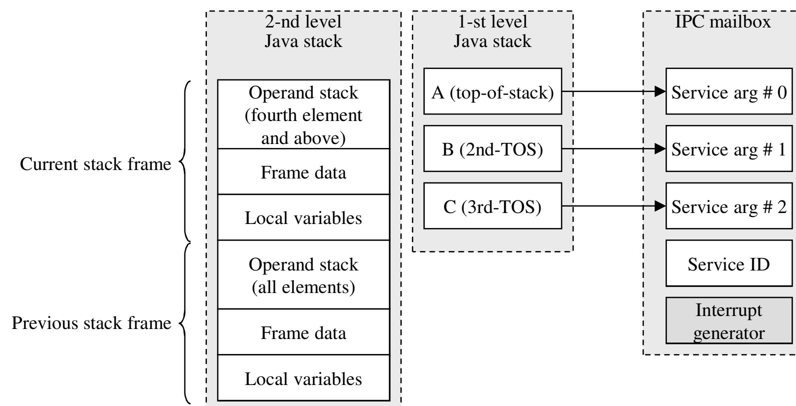


Fig. 11. IPC Mailbox for Java-to-RISC service invocation. The top three stack elements A, B, C contain the Java native method parameters and are aliased to the service argument registers.

Another usage of the mailbox IPC to request the RISC service is for the dynamic class loading module to perform the parsing-loading of a new class file into the second-level method area. Such invocation is not triggered by a native method invocation in the Java source code, but a signal from the dynamic resolution unit, as described in section 5.1.

Note that the overhead of IPC is quite high. For example, on Xilinx ML-507, the turn-around overhead of an empty ISR is around 500 clock cycles. Such overhead is not significant for complex services, such as the dynamic class loading or I/O operations. However, the overhead is not negligible for small services implemented on the RISC side. In our previous designs [Lin et al. 2012], we have implement memory management (i.e. 'new' bytecode) and string manipulations using RISC services. However, due to high overhead of the IPC requests, we have implement both operations within JAIP in the current design.

6. EXPERIMENTAL RESULTS

The proposed JAIP has been implemented on a Xilinx ML507 SoC emulation platform. The ML507 board contains a Virtex 5 FPGA device where the JAIP core is synthesized using Xilinx XST 13.4 along with a Microblaze core to form a heterogeneous dual-core application processor. The system frequency is set to 83.3 MHz. The RTL model of JAIP is written in VHDL. The synthesis report on the resource usages of the major IPs (JAIP, Microblaze, and DDR2 controller) and the full system are shown in Table II. The FPGA of ML507 is a Virtex 5 device (XC5VFX70TFFG1136). In addition to the circuit components shown in Figure 2, the JAIP core also contains a dedicated string acceleration module that costs around 1900 LUT6s. If the string manipulation performance is not crucial for the target applications, the logic size can be reduced further (JAIP can still perform string operations with proper system classes support).

In section 6.1, we compare the performance of the proposed JAIP against the JOP Java processor using the JemBench benchmark suite [Schoeberl et al. 2010]. In section 6.2, we will present a detail analysis between Sun's CVM (with JIT technology) and JAIP to highlight the pros and cons of each Java acceleration approach. Finally, in section 6.3, we will present some analyses on the performance of the proposed exception handling mechanism of JAIP, again using CVM-JIT as a comparison point.

Table II. Logic usage of the Java SoC on a Virtex-5 FPGA device

FPGA Logic Units	LUT6s	Flip-Flops	BRAMs
JAIP core	13,677	7,343	37
Microblaze core	1,927	2,155	4
DDR2 Controller	2,541	3,694	5
Total System Usage	21,814 (48%)	18,442 (41%)	73 (49%)

Note: The logic usage of the JAIP core includes all components shown in Figure 2 plus a string accelerator. JAIP has an object heap cache. The cache size is set to 16KB to match the data cache size of the PowerPC platform for CVM-JIT. Each block RAM (BRAM) of a Virtex-5 device is 2KB in size (18-kbit with parity bits). The percentages shown in the "Total System Usage" entries represent the percentage of LUT6s, flip-flops, and BRAMs used in the FPGA device.

6.1 Performance Comparison against other Java Processors

In this section, the performance of JAIP is first compared against another hardwired Java processor, JOP, using the JemBench benchmark suite. JOP is a Java processor designed for real-time, embedded Java applications. The processor has a very small

footprint but it does not support dynamic class loading. The design goal of JOP focuses on real-time applications where the behavior of the processor should be deterministic. The author of JOP also develops the JemBench test suite for the benchmarking of Java VMs targeted at embedded applications (e.g., the CLDC profile). There are two major categories of benchmark programs in JemBench, namely serial benchmarks and multithreaded benchmarks. Since the focus of this paper is on the efficiency of the bytecode execution pipeline with dynamic resolution capabilities, we have only used the serial benchmark of JemBench for investigation in this paper. For the temporal multi-threading performance of JAIP, please refer to Su et al. [2014]. Table III lists the benchmark scores reported by the JemBench application for JAIP running at 83.3 MHz and for JOP running at 100 MHz. Each score roughly represents the number of iterations the benchmark program can execute in one second. The scores for JOP are retrieved from Jopwiki [2010].

From Table III, one can see that even if JAIP is running at a lower clock rate, it still has a higher performance than JOP does. On the other hand, JAIP uses a lot more FPGA logic resources than JOP does. The performance of the RISC core used in the JAIP SoC does not play a crucial role in its benchmark performance. The RISC core is only used to perform dynamic class loading. Since each class is only loaded once into the DDR SDRAM, the performance of the RISC core does not matter much here.

Another famous Java processor is the picoJava II processor. According to the Sieve and Kfl scores published in Puffitsch and Schoeberl [2007], the performance of JAIP@83.3Mhz is slightly faster than a picoJava-II@40MHz. However, the picoJava II on FPGA does not support dynamic class loading and its FPGA logic usage is much larger than that of JAIP's. On the other hand, picoJava II has a hardware FPU while JAIP does not.

Table III. JemBench Performance of JOP and JAIP.

	Sieve	Bubble	Kfl	Lift	UdpIp
JOP @ 100 MHz	6913	3878	20686	20557	9174
JAIP @ 83.3 MHz	7742	5004	28248	22850	13596
picoJava-II@40 MHz	7797	N.A.	23290	N.A.	N.A.

Note: The numbers are JemBench scores. Higher numbers means better performance.

6.2 Performance Comparison against CVM-JIT

Just-In-Time (JIT) compilation is a popular technique for Java acceleration. In this section, we conduct some experiments to show the pros and cons between JAIP and Sun's CVM-JIT. The version of CVM we use is phoneme_advanced_mr2-b167, revision 20547 from the PhoneME project, available from Oracle [2013]. The PowerPC version of CVM is compiled for a Xilinx ML-403 platform running Linux 2.6.38. The PowerPC on the ML403 is a hard-core IP with 16KB of instruction cache and 16KB of data cache. The system clock is set to 83.3MHz. Although we could have simply quoted the JemBench scores of CVM-JIT for performance comparisons, this approach does not reveal the complete runtime behaviors of Java acceleration with JIT technology. The design of JemBench does not take into account the JIT compilation overhead.

Theoretically, JIT technique compiles a Java program into the native codes of a register-based processor, which should be more efficient than a stack machine for general-purpose computing. However, in practice, the overhead of runtime compilation is not negligible. In general, a JIT compiler performs adaptive compilation to reduce the memory requirement for native code cache and possibly the

overhead of compilation. In short, a Java VM with JIT may not compile a complete Java method into native codes upon its first invocation. The decision on when to compile a method into native codes is not trivial. If a method is intended to be invoked only for a few times, the overhead of compilation may outweigh the gain. Such overhead of JIT compilation does not show up in most synthetic benchmarks. For example, before JemBench performs measurement of the score for each test, it first conducts some test runs of no less than 1024 iterations to estimate the number of iterations required for the actual test. As a result, the JIT algorithm will most likely compile the key methods into native code during the test run period. When the actual benchmarking process begins, the measured time does not include the test run time and hence excludes the JIT overhead. The situation is similar in other benchmark programs such as the Embedded Caffeine Mark. Whether excluding the JIT compilation overhead for performance evaluation is reasonable or not depends on the actual use cases. In this paper, we merely try to present the pros and cons of each Java acceleration technologies, including the impact of JIT overhead.

To conduct a more thorough investigation of the performance differences between a hardwired Java processor and the JIT acceleration technique, we extract the application program Kfl, Lift, and UdpIp from the JemBench suite so that we can control the number of repetitions to reveal the impact of JIT overhead. In addition, we have included two other applications, Logic and Pi for the investigation[2]. The Logic program is extracted from the Embedded Caffeine Mark. It performs sophisticated conditional branches in a loop which is difficult for a compiler to optimize. The Pi program uses Euler's formula [Beeler et al. 1976] to compute the number π to 500 digits. The program only uses integer arithmetic to approximate the value of π.

Table IV and Table V show the execution time of each program running for 1, 10, 50, 100, and 500 iterations using JAIP and CVM-JIT, respectively. The execution time is measured in milliseconds and rounded to an integer because the system timer of CVM-JIT has only millisecond resolution. Since both platforms have instruction (method) and data caches, the time measurements do not always grow linearly with respect to the number of iterations. Note that both platforms have a 16KB two-way set associative data caches with 32-byte cache line. Although Table IV and V show that JAIP outperforms CVM-JIT in almost all cases, for the final JemBench scores shown in Table VI (where the JIT overhead is not accounted for), CVM-JIT does outperform JAIP. On the other hand, if we use the extracted Kfl, Lift, and UdpIp programs to estimate the JemBench scores while taking into account the JIT overhead, the scores drops to 27907, 36397, and 13135, respectively. In this case, JAIP still has higher scores than CVM-JIT for two out of three applications.

For the BubbleSort and Sieve programs of JemBench, the JIT overhead only reduces the JemBench scores slightly (to 12437 and 6267, respectively) because these two programs are quite small. The reason why JIT performs better on BubbleSort and Sieve than JAIP does is probably because the bytecodes generated by a Java compiler are often un-optimized. Evidences show that if a Java processor performs some simple bytecode-level optimizations at class-loading time, the performance can be increased by up to 29% [Tyystjaervi et al. 2010]. We will look into this approach in the future.

It would be interesting to plot out the long-term performance curves between JAIP and CVM-JIT. In Figure 12, we use a log-scale to plot the performance curve of

[2] The source of these test programs are available at: http://www.cs.nctu.edu.tw/~cjtsai/research/jaip.

JAIP and CVM-JIT for Kfl, Lift, and UdpIP when the execution iterations are 10, 100, 1000, and 10,000. Note that the plot of the Lift program in Figure 12 shows two cross-points between the JAIP and the CVM-JIT curves. There are several possible reasons for this behavior, such as the mismatch between the time-varying working sets of the Lift program and the data cache, the kick-in of the adaptive JIT algorithm somewhere between 10 and 50 iterations of execution, and the impact of the JAIP two-level method area caching architecture. In short, for the three application benchmark programs of JemBench, if the executions are repeated for only a few thousand times, JAIP will run faster than CVM-JIT. However, if the number of executions becomes large, eventually CVM-JIT will out-perform JAIP. Of course, there are cases that CVM-JIT runs slower than JAIP regardless of the number of repetitions. For example, for the 'Logic' and 'String' test programs from the Embedded Caffeine Mark test suite, JAIP always out-performs CVM-JIT. Nevertheless, these two test programs are quite synthetic that they may not represent the workloads in real usage cases [Isen et al. 2008]. One practical application where JAIP always outperforms CVM-JIT is the Pi program used in our test. When the number of repetitions approaches a larger number, say 50000, and the overhead of JIT diminishes, it takes 167.58 msec on average for CVM-JIT to execute one round of Pi calculation while it only takes JAIP 166.60 msec to do so.

Other cases where a hardwired Java processor can outperform CVM-JIT are the situations when exception handling or context switching for multi-threading are involved. We will conduct some experiments in section 6.3 to show that JAIP has a much lower overhead on Java exception handling than CVM-JIT does. For multi-threading comparisons, please refer to Su et al. [2014]. Based on our experiments, for single-core temporal multi-threading, JAIP can outperform CVM-JIT significantly when the thread number increases (above or equals four).

Table IV. Application execution time (in milliseconds) of JAIP.

Test \ # Iter.	1	10	50	100	500
Logic	2	16	78	156	780
Pi	167	1666	8330	16660	83301
Kfl	0	1	2	4	18
Lift	18	19	21	23	40
UdpIp	0	1	4	8	37

Note: In column one, the rounded time of Kfl and UdpIp are zero. The original unrounded numbers are 0.15 and 0.22 msec, respectively.

Table V. Application execution time (in milliseconds) of CVM-JIT.

Test \ # Iter.	1	10	50	100	500
Logic	55	71	144	235	962
Pi	213	1884	8561	16943	84118
Kfl	1	4	19	46	172
Lift	11	13	25	41	120
UdpIp	2	8	39	58	218

Table VI. JemBench performance of Sun's CVM on an 83.3MHz PowerPC platform.

	Sieve	Bubble	Kfl	Lift	UdpIp
CVM (interpreter)	955	508	3141	3305	1605
CVM-JIT	12720	7104	36632	40504	16458

Note: Each JemBench score roughly represents the iterations per second a test can execute. However, the scores for CVM-JIT exclude the JIT compilation overhead (see the text for explanation). When the JIT overhead is included, the Kfl, Lift, and UdpIp scores drop to 27907, 36397, and 13135, respectively.

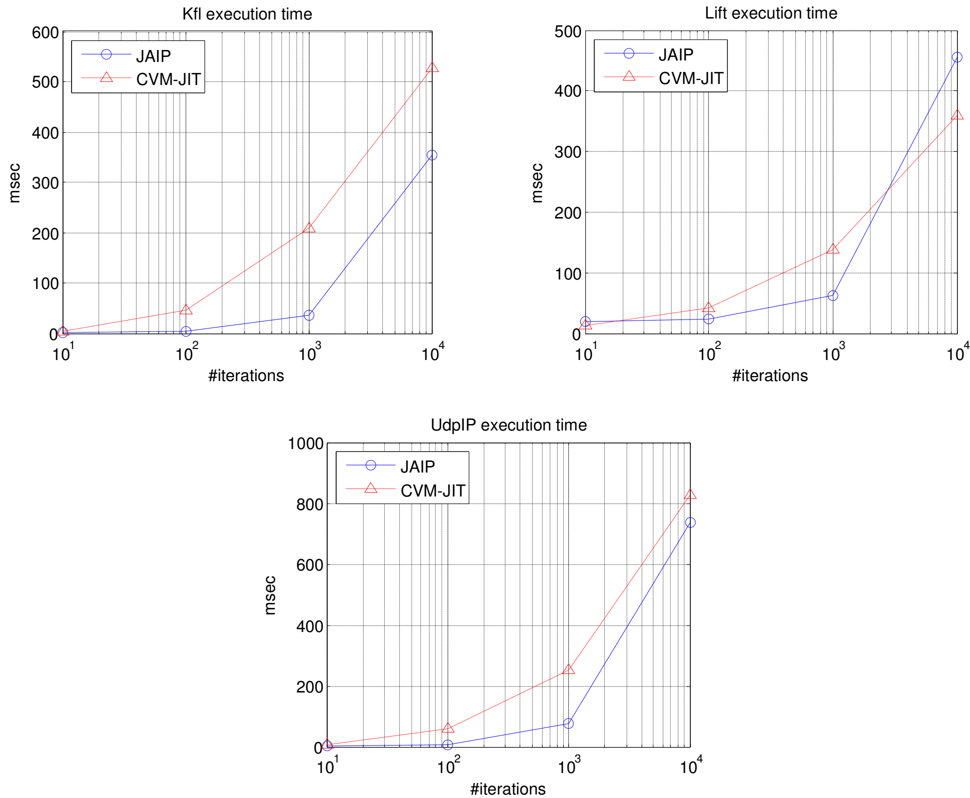


Fig. 12. The execution time of Kfl, Lift, UdpIp using JAIP and CVM-JIT. Lower numbers mean better performance. The horizontal axis is the number of repeated executions of each test.

6.3 Performance of Java Exception Handling

The proposed JAIP architecture contains full hardware support for Java exception handling. For most Java programs, the efficiency of exception handling is not crucial since error recovery usually restores the program to a safe, idling state. Nevertheless, as mentioned in section 4.4, the hardware complexity required to implement the proposed Java exception handling mechanism is quite low, it would be interesting to quantify the runtime overhead of the proposed architecture. In this section, we conduct some experiments to estimate the machine cycles required for exception handling. Again, we use CVM as a comparison point. However, for the case of exception handling, CVM without JIT actually has lower overhead than CVM with JIT.

We have designed two test cases to measure the overhead of exception handling. The first test case focuses on the overhead of searching the target exception routine in the same method where the exception occurs. The test program is a small Java program with a try-catch block in a for-loop. The loop runs through the try-catch block for 10000 iterations. In the try block, there are only two statements. The first statement instantiates an ArithmeticException object; and the second statement throws the exception object. The number of catch statements varies from 1 to 15. All the catch statements are empty exception handling routines. To estimate the

overhead of the exception handling mechanism, we use a base program that is exactly the same as the test program except that the 'throw' statement is commented out. Thus, the difference in the execution times between the test program and the base program divided by 10000 represents the overhead of a single exception handling operation. We convert the time differences from milliseconds to machine cycles and the results are shown in Table VII. As one can see, searching the exception routines using JAIP is extremely efficient. Since all offsets of the exception routines are parsed and stored in an on-chip exception lookup table when the class is loaded, once JAIP determines that the Java bytecode counter is located within the try-block, it can use the exception object class ID (and the IDs of all its parent classes) to search the on-chip table to locate the handler routine quickly. On the other hand, both CVM (interpreter) and CVM-JIT requires more than an order of magnitude of time to find the correct handler. It is interesting to see that CVM-JIT actually has much higher overhead than the CVM interpreter does. The sluggish of CVM-JIT in exception handling should have nothing to do with the overhead of JIT compilation since the number of cycles shown in Table VII is the difference between the test program and the base program. Therefore, the JIT overhead should have been excluded already.

Test case two focuses on the overhead of searching the target exception routine in a sequence of calling methods, which involves the processes of unwinding the Java stack. Each method (except the last one) in the call sequence contains an exception routine that does not match the target exception object. The results are shown in Table VIII. We have used a similar scheme as in test case one to measure the runtime overhead of exception handling. Again, JAIP has much lower overhead than both the CVM interpreter and CVM-JIT in performing stack unwinding and searching for the target handler along the caller methods sequence.

Table VII. The overhead (in clock cycles) of catching an exception in test case one.

# searched routines	JAIP	CVM	CVM-JIT
1	11	2017	3920
5	73	2300	4487
10	158	2773	4818
15	243	3260	5082

Table VIII. The overhead (in clock cycles) of catching an exception in test case two.

# searched caller methods	JAIP	CVM	CVM-JIT
5	197	3741	7548
10	435	6224	11837
15	675	8481	15567

7. CONCLUSIONS

In this paper, we have proposed several architecture designs to facilitate the encapsulation of a Java core as a reusable IP. With the proposed architecture, the Java processor IP can be integrated into an application processor SoC easily. The key architecture that enables host processor invocation of the Java accelerator is the method area manager unit and its associated two-level method area memory hierarchy. For system software integration with the RISC core, mailbox-driven ISRs on the RISC side are used to modularize the integration effort. The design of the software stack of the JRE follows the concept of the JavaOS model.

For the bytecode execution engine, we have also proposed a new variant of two-level stack design with local variable cache. With this stack architecture, we can implement a two-fold instruction folding pipeline with low hardware cost. We have implemented a complete Java application processor on a Xilinx ML-507 FPGA

development board and conduct performance verification to show the advantage of the proposed architecture over other Java processors and a VM based on the JIT technology.

Currently, we are working on the implementation of a multicore Java SoC. Based on our initial results on a four-core JAIP SoC, the performance tested using the three JemBench parallel benchmark programs can scale up by more than three times. We are also looking into hardware-friendly techniques for improving the performance of the dynamic resolution controller.

REFERENCES

B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. *Proc. of 16th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications* (2001).

M. Beeler, R. W. Gosper, and R. Schroeppel. 1972. *HAKMEM: MIT Artificial Intelligence Laboratory, Memo AIM-239*, Item 120, p. 55. Cambridge, MA.

U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer. 1999. A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling. *Proc. of 1999 Int. Conf. on Parallel Architectures and Compilation Techniques*, Newport Beach, USA. pp. 34-39.

L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung. 1998. Stack Operations Folding in Java Processors. *IEE Proc. on Computers and Digital Techniques*, vol. 145, pp. 333-340.

M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. 2004. A Dynamic Compiler for Embedded Java Virtual Machines. *Proc. of 3rd Int. Symp. on Principles and Practices od Programming in Java*. Dublin.

E. Duesterwald. 2005. Design and Engineering of a Dynamic Binary Optimizer, Proceedings of the IEEE, Vol. 93, No. 2, pp. 436-448.

M. W. El-Kharashi, F. Elguibaly, and K. F. Li. 2001. Adapting Tomasulo's Algorithm for Bytecode Folding based Java Processors. *ACM SIFARCH Computer Architecture News*, Vol. 29 Issue 5.

M. W. El-Kharashi, J. Pfrimmer, K. F. Li, and F. Gebali. 2003. A Design Space Analysis of Java Processors. *Proc. IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, 159-163.

D. S. Hardin. 2001. Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the JavaTM Virtual Machine. *Proc. of the 4th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing*, pp.53-59.

C.-F. Hwang, K.-N. Su, and C.-J. Tsai. 2010. Low-Cost Class Caching Mechanism for Java SoC. *In Proc. of IEEE Int. Symposium on Circuit and Systems*, Paris. 3753-3756.

C. Isen, L. John, J.P. Choi, and H. J. Song. 2008. On the Representativeness of Embedded Java Benchmarks. *Proc. of IEEE Int. Symp. on Workload Characterization*. Seattle, USA. 153-162.

K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, Proc. of the 15th ACM SIGPLAN conf. on Objected-Oriented Programming Systems, Languages, and Applications. New York, USA.

S. A. Ito, L. Carro, and R. P. Jacobi. 2001. Making Java Work for Microcontroller Applications, *IEEE Design & Test of Computers*, Vol. 18, Issue 5, pp. 100–110.

JCP 2005. *Mobile 3D Graphics API, JSR-184*, Java Community Process Program.

JCP 2006. *The J2ME Mobile Media API, JSR-135*, Java Community Process Program.

W. Ji, F. Shi, B. Qiao, and H. Song. 2007. Multi-Port Memory Design Methodology Based on Block Read and Write. Proceedings of IEEE Int. Conference on Control and Automation, Guangzhou, China. 256-259.

Jopwiki. 2010. Jopwiki project webpage. Retrieved July 27, 2010 from http://www.jopwiki.com/JemBench.

K. B. Kent, and M. Serra, 2002. Hardware Architecture for Java in a Hardware/Software Co-design of the Virtual Machine. *Proc. of Euromicro Symposium on Digital System Design*.

M. Kimura, M. H. Miki, T. Onoye, and I. Shirakawa. 2002. A Java Accelerator for High Performance Embedded Systems, *Proc. 4th Int. Conf. of Massively Parallel Computing Systems*. Italy.

H.-J. Ko, and C.-J. Tsai. 2007. A Double-Issue Java Processor Design for Embedded Applications. *Proc. of IEEE Int. Symposium on Circuit and Systems*, New Orleans. 3502-3505.

A. Krall. 1998. Efficient Java Just-in-Time Compilation. *Proc. of the IEEE Int. Conf. on Parallel Architectures and Compilation Techniques*, Paris. 205-212.

Z.-G. Lin, H.-W. Kuo, Z.-J. Guo, and C.-J. Tsai. 2012. Stack Memory Design for a Low-Cost Instruction Folding Java Processor. *In Proc. of IEEE Int. Symposium on Circuit and Systems*, Soeul, Korea. May 20-23, 3326-3229.

T. Lindholm and F. Yelling. 1999. *The Java Virtual Machine Specification, 2nd. ed.*, Addison-Wesley, Longman Publishing Co., Inc. Boston, MA, USA.

H. McGhan and M. O'Connor. 1998. PicoJava: A Direct Execution Engine for Java Bytecode. *IEEE Computer*, 31, 10, 22-30.

Oracle. 2013. Phoneme project webpage. Retrieved Sep. 27, 2011 from https://java.net/projects/phoneme.

B. R. Montague. 1997. JN: OS for an Embedded Java Network Computer. *IEEE Micro*, 17, 3, 54-60.

C. Pitter and M. Schoeberl. 2010. A Real-Time Java Chip-Multiprocessor. *ACM Trans. on Embedded Computing Systems*, Vol. 10, No. 1.

C. Porthouse. 2005. High Performance Java on Embedded Devices, *Jazelle DBX technology: ARM acceleration technology for the Java Platform*, White paper of ARM Ltd.

T. B. Preusser, M. Zabel, R. G. Spallek. 2007. Enabling Constant-Time Interface Method Dispatch in Embedded java Processors. *Proc. of 5th Java Tech. for Real-time and Embedded Systems (JTRES'07)*, Sep. 26-28, Vienna, Austria.

W. Puffitsch and M. Schoeberl. 2007. picoJava-II in an FPGA. *Proc. of 5th Java Tech. for Real-time and Embedded Systems (JTRES'07)*, Sep. 26-28, Vienna, Austria.

R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. 2000a. Architectural Issues in Java Runtime Systems. *Proc. 6th Int. Symp. on High-Performance Computer Architecture*, 387-398.

R. Radhakrishnan, D. Talla, and L. K. John. 2000b. Allowing for ILP in an Embedded Java Processor. *ACM SIGARCH Computer Architecture News*, pp. 294-305.

R. Radhakrishnan, R. Bhargava, L. K. John. 2001. Improving Java Performance Using hardware Translation. *Proc. of the Int. conf. on Supercomputing*, Italy, 427-439.

S. Ritchie. 1997. Systems Programming in Java. *IEEE Micro*, 17, 3 (Mar.). 30-35.

M. Schoeberl. 2005. Design and Implementation of an Efficient Stack Machine. *Proc.of 19th IEEE Int. Parallel and Distributed Processing Symp.*Apr. 04-08.

M. Schoeberl. 2008. A Java Processor Architecture for Embedded Real-Time Systems. *The EUROMICRO Journal of System Architecture*, 54, 1-2, 265-286.

M. Schoeberl and W. Puffitsch. 2010. Non-blocking Real-time Garbage Collection. *ACM Trans. on Embedded Computing Systems*, Vol. 10, No. 1.

M. Schoeberl, T. B. Preusser, and S. Uhrig. 2010. The Embedded Java Benchmark Suite JemBench. *JTRES'10*, Aug. 19-21, Prague, Czech Republic.

Sun Microsystems. 1999, *picoJava-II Microarchitecture Guide, Sun Microsystems,* Sun Microsystems.

T. Saentti. 2008. *A Co-Processor Approach for Efficient Java Execution in Embedded Systems*. Ph.D. dissertation, University of Turku, 2008.

H.-C. Su, T.-H. Wu, and C.-J. Tsai. 2014. Temporal Multithreading Architecture Design for a Java Processor. *Proc. of IEEE Int. Symposium on Circuit and Systems*, June 1-5, Melbornue, Austria.

J. Tyystjaervi, T. Saentti, and J. Plosila. 2010. Efficient Bytecode Optimizations for a Multicore Java Co-Processor System. *The Proc. of 12th Biennial Baltic Electronics Conf.*, Tallinn, Estonia, Oct. 4-6.

B. Venner. 2000. *Inside the Java 2 Virtual Machine*, 2nd ed., McGraw-Hill Companies.

N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. 1998. Object-Oriented Architectural Support for a Java Processor. Brussels, Belgium, *The Proc. of 12th European Conference on Object-Oriented Programming*. 330-354.

Z. Wang. 2010. An Intelligent Multi-Port Memory. *Journal of Computers,* 5, 3, 471-478.

L. Yan and Z. Liang. 2009. An Accelerator Design for Speedup of Java Execution in Consumer Mobile Devices. *Computers and Electrical Engineering*. Vol. Issue 6. 904-919.