JAIP-MP: A Four-Core Java Application Processor for Embedded Systems

Chun-Jen Tsai(✉), Tsung-Han Wu, Hung-Cheng Su, and Cheng-Yang Chen

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
cjtsai@cs.nctu.edu.tw

Abstract. In this chapter, we present a four-core Java application processor, JAIP-MP. In addition to supporting multi-core coherent data accesses to shared memory, each processor core in JAIP-MP is a hardwired Java core that is capable of dynamic class loading, two-fold bytecode execution, object-oriented dynamic resolution, method/object caching, Java exception handling, preemptive multi-threading, and memory management. Most of the essential OS kernel functions are implemented in hardware. In particular, the preemptive multi-threading performance is much higher than that of a software-based VM running on a traditional OS kernel such as Linux. Currently, single-cycle context switching with a time quantum as small as 20 μs can be achieved by each core. More importantly, the Java language model itself makes it possible to maintain binary portability of application software regardless of the hardwired OS kernel component. In summary, JAIP-MP could be used to study the potential benefits of OS kernel implementation in hardware.

Keywords: Java processors · Multi-core processors · Embedded SoC · Hardwired operating system kernel

1 Introduction

The Java programming language has been one of the most popular programming languages for over a decade. There are many reasons for its popularity. For example, it is a clean language designed with object-orientated paradigm from scratch, without unnecessary features such as multiple inheritance or pointer arithmetic that can be easily abused by programmers. Memory management in Java is implied by the object-oriented model and requires no special treatment from the programmers. It maintains a great Job on backward compatibility that application binaries written for very old version of Java can usually be executed under the latest versions of Java Runtime Environment (JRE) regardless of the underneath operating systems. One of the reasons that a Java program can be portable across versions and platforms is that the Java language model defines some interfaces, such as multi-thread programming and memory management, which are usually defined by the operating systems.

There are many variations of JRE for embedded systems, including Sun's CDC/PBP, CLDC/MIDP and Google's Android platform. Most existing implementations are

Y. Shin et al. (Eds.): VLSI-SoC 2015, IFIP AICT 483, pp. 170–192, 2016.
DOI: 10.1007/978-3-319-46097-0_9

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. A different design approach, the JavaOS model [1, 2], 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.

In this chapter, we will present the design of a multi-core Java System-On-a-Chip (SoC) that implements most of the OS kernel functions, as well as the bytecode execution pipeline using hardware circuits. The key component of the Java SoC is the Java Application IP (JAIP). JAIP is a reusable processor IP that has hardwired support of the following Java language features [3, 4]:

- Two-fold instruction folding
- Multithreading, synchronization, and exception handling
- Heap management and garbage collection
- Class inheritance and interface invocations
- Method and heap object caching
- Dynamic class loading and dynamic symbol resolution
- String operation acceleration for matching and copying

The remainder of this chapter is organized as follows. For the rest of Sect. 1, we will first introduce some previous work on multi-core Java processor designs and discuss the possible benefits of implementing a multi-core Java SoC with the major OS kernel functions crafted in hardware. Section 2 presents the architecture of the JAIP core, including the microarchitecture of the instruction execution pipeline, the stack architecture, the preemptive thread manger, and the memory manager and garbage collector. We also have a brief discussion on how the I/O subsystem interface can be merged into the dynamic resolution module of a Java VM. Section 3 discusses the required glue logics to integrate four JAIP cores into a single SoC, mainly, the inter-core communication unit, the multi-core thread manager, and the data coherence controller. The experimental results are presented in Sect. 4. Finally, some discussions are given in Sect. 5.

1.1 Multi-core Java Processors

Although there are many hardware Java core designs [5], very few of them have been synthesized in a real multi-core application processor [6]. One of the reasons may be due to the fact that previous work shows that a Just-in-time (JIT) based VM running on a high performance general-purpose processor can often outperform a hardwired Java processor [7]. Therefore, it seems that there is no need to further pursuit the development of hardwired Java processors. However, most of the JIT vs. hardwired VM comparisons are conducted using benchmarks where the application class files are not optimized for Java processors. For example, it has been shown that some popular benchmark class files can run much faster on a Java processor if bytecode optimizations in the class files are conducted [8]. Please note that an optimized Java class file still conforms to the Java specification and is portable across different Java platforms. In addition, many benchmarking processes discard the impact of the JIT compilation overhead [3]. Although ignoring the JIT overhead is reasonable for some applications, it is not valid for remote invocations that are common for object-oriented distributed computing. Other reasons why a hardwired VM could be useful for practical applications will be discussed in Sect. 1.2.

Most Java processors support thread synchronization using software modules [9–11]. However, the execution time of a software-based synchronization operation, such as a mutex lock, can take more than a few hundred clock cycles since the lock objects are often accessed in conventional trap routines. PicoJava [10] uses a few special-purpose registers for the speedup of synchronization operations, but it still needs to use the main memory to maintain the information of all waiting threads and lock objects. Therefore, a high number of concurrent synchronized read/write operations can have significant synchronization overheads. JOP-CMP [6] supports at most 8 processor cores with a software-based thread scheduler and a hardwired synchronization unit [6, 12]. There is only one global lock register in the synchronization unit, which means that any threads trying to acquire the lock must wait until the lock is released. In addition, JOP-CMP does not have a coherent data cache. All data accesses will be directly issued to the external memory, which can hinder the multi-thread performance significantly.

1.2 Potentials of Hardwired Virtual Machines

Traditionally, Java programs are executed using software-based virtual machines. To improve the performance of bytecode execution, JIT or ahead-of-time (AOT) compilation techniques are often adopted in modern virtual machines. Previous work shows that JIT or AOT techniques can arguably achieve better performance than a hardwired Java processor. We have already presented some reasons that may lead to the bias of such conclusions in Sect. 1.1. Another reason is that existing Java processors mainly focuses on the architecture design of the bytecode execution pipeline. Things may be different when the full JRE is considered as the target of hardware design.

For example, one of the most intriguing features of the Java programming model is that all data accesses and code invocations must go through the dynamic resolution mechanism. Although dynamic resolution is usually considered as a language feature that hinders efficiency significantly, there are some benefits of dynamic resolution that has not been investigated thoroughly, especially when the Java VM is implemented in hardware. First of all, with dynamic resolution, there is no need to assume a large "flat" memory model for a Java VM implementation. A Java VM may manage many concurrently accessible memory blocks seamlessly to improve the performance of data processing without the programmer knowing the physical layout of the memory subsystem. Securities issues related to malicious pointer-based indirect data accesses can be handled more rigorously with the Java model since all data accesses must be approved by the dynamic symbol resolution unit (DSRU). Finally, a method call can be re-directed to hardware logics without going through memory-mapping process and shared bus transactions, which may improve throughput significantly. The last point will be explained further in Sect. 2.5.

In [4], we have presented the preemptive multi-threading efficiency of the JAIP core. It is capable of single-cycle multi-thread context switching with a time quantum as small as 20 μs. For a traditional OS kernel such as Linux, the time quantum for a thread is usually around 10 ms. As a result, for single-core multithread applications, hardware-based thread manager can achieve much smoother concurrent executions of all threads of equal priorities than a software-based thread manager. This is a very strong reason for the development of an efficient hardwired Java processor core. The Java language specification defines standard programming interfaces for OS kernel services such as process management and memory management. Other popular languages such as C and C++ do not standardize these functions. For example, thread creation API's are OS-dependent for C programs. Therefore, it would be beneficial to investigate the potentials of a fully hardwired OS kernel based on the Java programming model.

In short, a Java processor can be designed to implement the entire OS kernel system services in hardware while maintaining application portability. This is particularly important for embedded real-time applications where context-switching efficiency and dynamic memory management overhead are the key performance factors of a system. On the other hand, it is not so easy to "harden" the OS kernel for a traditional RISC processor due to lack of "standard" system calls for C/C++ applications.

2 The Architecture of the JAIP Core

In this section, we present some design details of the JAIP core that is used as the key component of the hardwired multi-core Java runtime environment (JRE). The design target of the JAIP core is for FPGAs and thus dual-port SRAM blocks that are common in FPGAs are used extensively to optimize the architecture for the object-oriented language model that makes Java one of the most popular programming languages.

2.1 The Overview of JAIP Core

Figure 1 shows the overall block diagram of a single-core SoC based on the JAIP core. The complete SoC is composed of a RISC core and a JAIP core. For the execution of a Java program, the RISC core is only responsible for reading and parsing of the class files stored in a JAR file on the Compact Flash (CF) card. The RISC parser will convert the standard Java class files into runtime class images on-the-fly for direct execution by the JAIP core. The converted class file images are stored in the second-level method area in the main memory. The class file parser will maintain a symbol cross reference table stored in the main memory for all loaded classes. The Java core is completely responsible for the two-fold execution of bytecodes, dynamic loading of the class images into the method area, dynamic resolution, memory management, and preemptive multi-thread scheduling. In the future, we will remove the dependency of the JAIP core on the RISC core for class file reading and parsing.

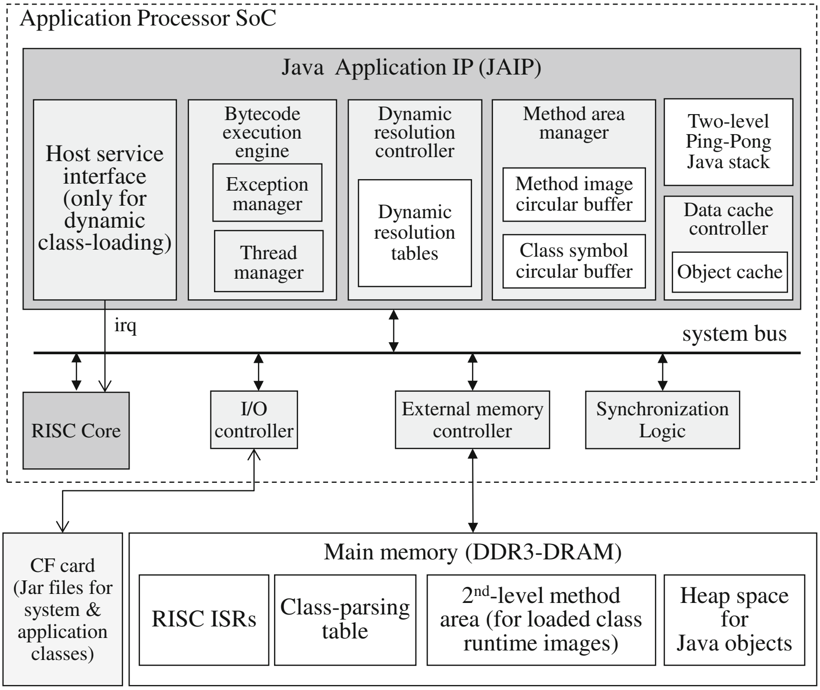

Application Processor SoC
Java Application IP (JAIP)
Host service interface (only for dynamic class-loading)
Bytecode execution engine
Exception manager
Thread manager
Dynamic resolution controller
Dynamic resolution tables
Method area manager
Method image circular buffer
Class symbol circular buffer
Two-level Ping-Pong Java stack
Data cache controller
Object cache
irq
system bus
RISC Core
I/O controller
External memory controller
Synchronization Logic
CF card (Jar files for system & application classes)
Main memory (DDR3-DRAM)
RISC ISRs
Class-parsing table
2nd-level method area (for loaded class runtime images)
Heap space for Java objects

Fig. 1. The architecture of a Java application processor based on the JAIP core.

JAIP adopts a two-level method area design. All classes loaded at runtime will be stored in the main memory (i.e., the second-level method area) using the late-resolution policy. A Java method (and its related symbol information) must be loaded into the on-chip method cache (the first-level method area) before it can be executed by the bytecode execution engine. In short, the complete class images of the Java applications are stored in the main memory while the most recently used methods and symbol information are stored in the on-chip method area in a FIFO manner.

Since the Java VM is basically a stack machine, i.e., all the local variables and the intermediate values of operations are stored in the runtime stack, fast accesses to the most recent stack frames are essential to the performance of a Java processor. JAIP uses a special-purpose on-chip memory and three top-of-stack registers to form a two-level Java runtime stack. The special-purpose on-chip memory is a customized four-port memory device custom-designed for the Java bytecode instruction set architecture. It is composed of a pair of interleaving two-port memory blocks and four registers. The design is a good tradeoff between performance and implementation cost as compared to the Java processors with a large stack cache [10, 13, 14].

The two-level Java stack allows JAIP to perform two-fold instruction folding for frequent bytecode pairs [15] such as load-load, ALU-store, etc. However, to simplify the microarchitecture, some folding patterns, e.g. ALU-ALU bytecode pairs, are not allowed. According to our empirical studies, the instruction folding rate of JAIP ranges from 10 % to 40 % for different benchmark applications.

2.2 The Bytecode Execution Engine and the Stack Memory

The bytecode execution pipeline of the JAIP core is shown in Fig. 2. The Java bytecodes are translated into native JAIP instructions called j-codes before instruction decoding and folding. The JAIP core performs two-fold instruction folding of stack-related Java operations using a simple decision policy. In short, JAIP only supports the folding of the following stack operation pairs: Load-Store, Store-Load, ALU-Load, Load-ALU, Store-ALU, ALU-Store, Load-Load, and Store-Store. Note that in these stack operations, '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: a foldable j-code instruction pair, a single control instruction (such as a conditional branch), or a special data-processing j-code (such as the 'swap' operation). The two-level JAIP stack can encounter structure hazard whenever the j-code instruction pairs try to transfer two local variables stored in the same SRAM bank to the operand stack (or vice versa). This 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, we use a special-purpose 4-port memory customized to the Java ISA to reduce the occurrence of structure hazards while maintaining low implementation cost [16].

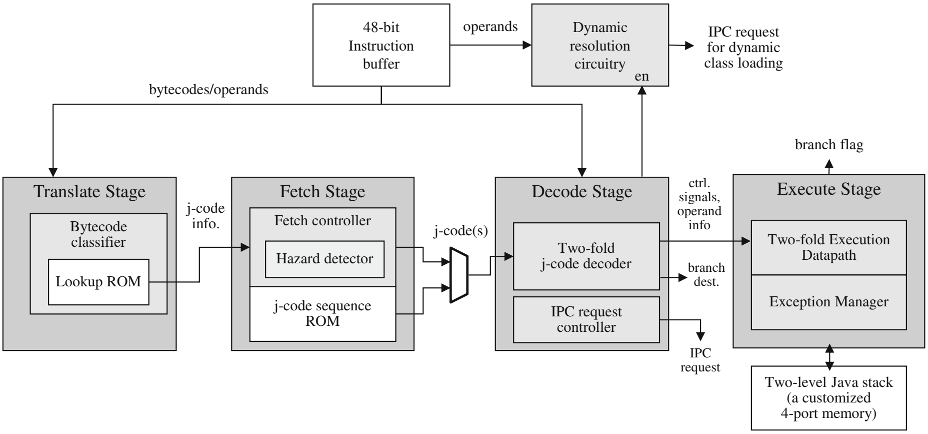


Fig. 2. The bytecode execution engine pipeline of JAIP.

According to the Java VM specification [17], 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 second-level Java stack memory is constructed by using two on-chip memory blocks organized in an interleaving structure to form a Java stack. In addition, four 32-bit local variable (LV) registers are used as a small cache for the first four local variables as shown in Fig. 3.

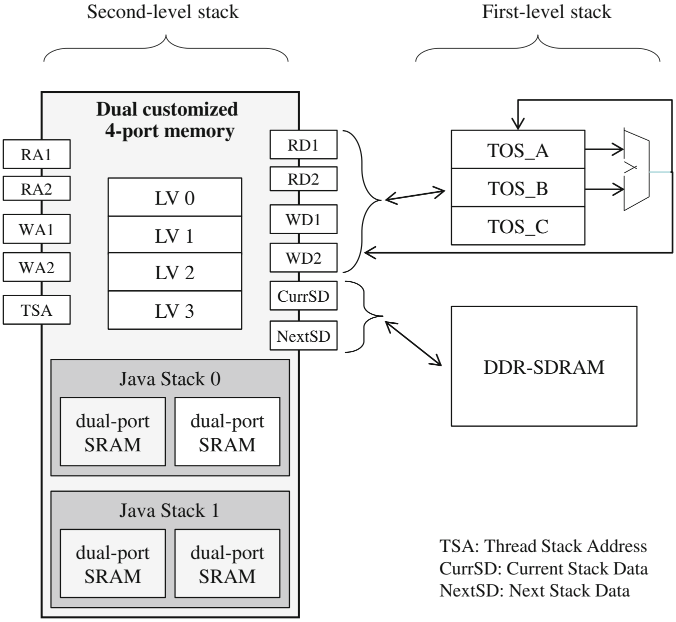


Fig. 3. The architecture of the stacks in a JAIP core.

In Fig. 3, there are two Java stacks instead of one. These two stacks form a ping-pong buffer to support fast context-switching operations for preemptive multi-threading. At any given time, only one of the stacks will be used as the active Java stack. The other stack will be used to load the stack frame of the next selected thread for the execution in the next time quantum. The details of context switching will be discussed in Sect. 2.3. 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/return such that they do not incur extra overhead. With this design, the folding of two stack operations of 'Load' and 'Store' of the first four local variables do not cause structure hazard. However, accesses to the local variables beyond the first four will not be folded by JAIP. This is a design choice to simplify the control logic.

2.3 Single-Core Preemptive Thread Management

For the execution of multi-thread Java programs, each thread must maintain its own registers and runtime stack. Typically, the register file of a Java processor is only composed of few special-purpose registers and can be swapped out to main memory quite efficiently. On the other hand, the Java runtime stack is much larger than the register file. If the runtime stack is stored in the main memory (e.g., DRAM), there is no need to save the runtime stack. However, most high-performance Java processors, including JAIP, use stack cache or on-chip memory to support instruction folding and to reduce the access delay of operands. In either case, the time it takes to swap out the on-chip stack would be non-negligible.

Saving/restoring the context of a JAIP thread involves transferring the stack frames (each ranging from a few bytes to a few hundreds bytes) to/from the main memory. In order for JAIP to support hardware-based multi-threading, we have designed a low-cost thread manager unit to reduce the context-switching overhead. As a result, in most cases, switching from the current thread to the next active thread only takes a single cycle. This is much faster than any software-based preemptive multi-tasking operations where a context-switching operation can take anywhere from a few hundreds to over a thousand cycles.

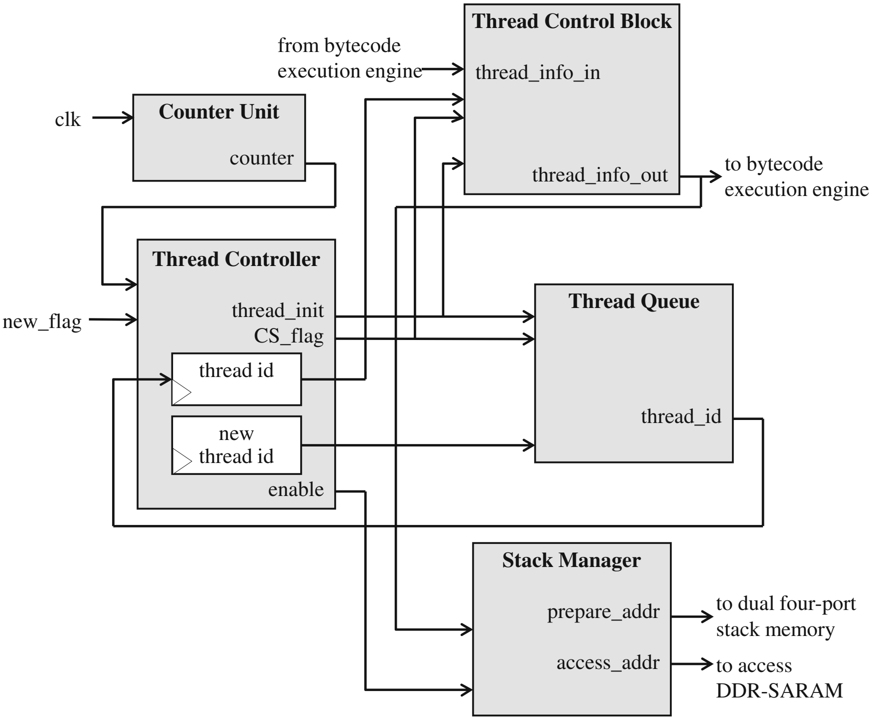


Fig. 4. The thread manager unit.

The architecture of the thread manager unit is shown in Fig. 4. When a Java program executes the start() method of an object derived from the Thread class, the JAIP execution pipeline will send a signal to the thread manager unit, informing the controller to initialize a new task in the on-chip thread control block (TCB) and enters the thread ID into the thread queue. Note that the execution of the start() method via the 'inovkevirtual' bytecode goes through the dynamic resolution unit of JAIP, which trigger the controller circuit of the thread management unit directly. More discussion on the direct invocation of hardware logic or I/O devices through a standard Java method invocation mechanism will be discussed in Sect. 2.5.

The structure of the TCB is shown in Fig. 5. In the current design, a fair round-robin algorithm is used in the controller to select the next ready thread. The state of a thread is stored in a TCB entry, which is composed of the following information:

1. The ID of the thread.
2. The Java class and method IDs of the thread.
3. The local variable pointer and the operand stack pointer.
4. The program counter and the number of local variables of the thread.
5. The first-level operand stack (the top-of-stack A, B, and C registers) of the thread.
6. The object reference (pointer) to the thread object in the Java heap.

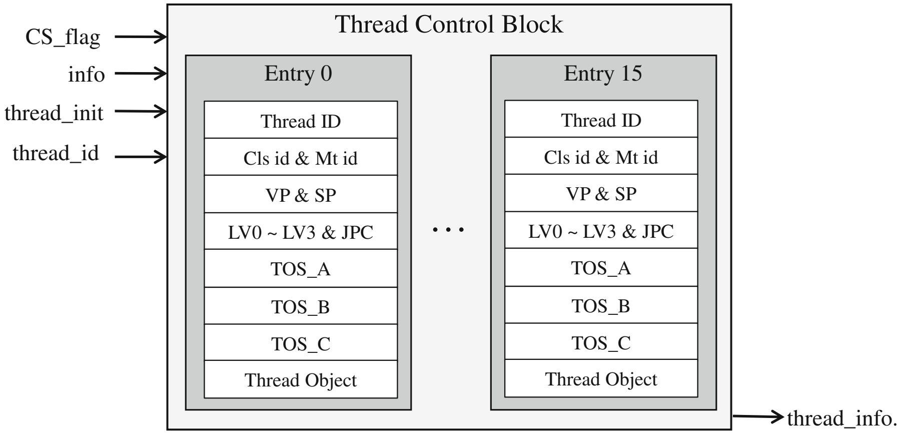


Fig. 5. Structure of the on-chip thread control block.

Each TCB entry is composed of eight 32-bit values. In the current design, the thread control block is implemented using an on-chip memory. We have set the maximal number of threads to 16 to limit the size of the TCB to 512 byte. The maximal number of threads can be extended easily at the cost of a larger on-chip memory. For thread management, we use a circular queue to store the ID of each thread in the queue. Every time a new thread is created by the Java application through the execution of the start()

method of a Thread object, a new thread ID will be generated and entered into the end of the thread queue. When the time slice of the current thread ends, its ID will be moved to the end of the queue and the thread whose ID is pointed to by the 'next' pointer will become the current thread.

The ping-pong stack architecture works as follows. As soon as a thread is selected as the current thread and starts its execution, the multi-threading logic also picks the next thread to be executed and, while the first thread is running, swaps in the runtime stack of the second thread from the main memory. When the time slice of the first thread is up, JAIP can be switched to the second thread within a cycle since its stack has already been setup. In the rare case where the restoration of the runtime stack of the second thread takes longer than the predetermined time quantum of the first thread, the time quantum will be extended until the runtime stack of the second thread is in place. The average time it takes to backup or restore a runtime stack to/from the backing store (the main memory) for the target system used in this chapter (Xilinx ML 605) is less than 10 µs when the system clock is 83.3 MHz.

When the execution is switched to the second thread, the runtime stack of the first thread will be saved to the main memory in parallel to the execution of the second thread. As soon as the stack of the first thread is saved, the multi-thread control logic will proceed to the setup of the third thread. With this design, the overhead of saving/restoring the runtime stack can be overlapped with the execution of the current thread. According to our experiments, the time slice of the proposed architecture can be as small as 20 µs and the only overhead in context-switching is virtually the reset of the processor pipeline (similar to a branch instruction). Smaller time slice means the distribution of the CPU resources to each thread is more even. This level of multi-threading efficiency is very difficult to achieve with a software-based preemptive multitasking operating system.

2.4 The Memory Manager and Garbage Collector

Garbage collection (GC) is an important feature of the Java programming model. It takes the burden of memory management off the programmers and removes common memory-related bugs in programs. However, runtime GC may induce high overhead and affect the performance of an application [18, 19]. This is particular true for software-based VM. Therefore, for embedded applications, programmers must be careful to avoid triggering GC unintentionally or the whole application may stall until the GC process is finished.

On the other hand, for hardwired Java VM, the GC circuitry can run in parallel to the bytecode execution pipeline, it is possible to design hardware based GC that does not stall the execution of the Java applications [20]. Although hardware-based GC is an active research direction [20–22], most designs are simply technical investigations and have not been integrated into a complete Java system. For example, [20] presents a synthesizable GC hardware, but the GC is exclusively evaluated on an FPGA using test patterns that represent typical applications.

Although GC is a crucial component of a JVM, the JVM specification does not enforce of any type of GCs for memory management. The memory manager hardware in JAIP includes hardware controllers that handle memory allocation and object caching (see Fig. 6). To perform garbage collection, the VM must carry out two types of operations. First, the VM must be able to determine that an object on the heap is not pointed to by any Java reference variables. Secondly, the GC mechanism must return the object space to the unused memory block list and merging two consecutive unused memory blocks if possible. In JAIP, we adopt the tracing garbage collector since it has low overhead and is suitable for hardware implementation. Furthermore, it can be a pluggable component to existing memory manager of JAIP.

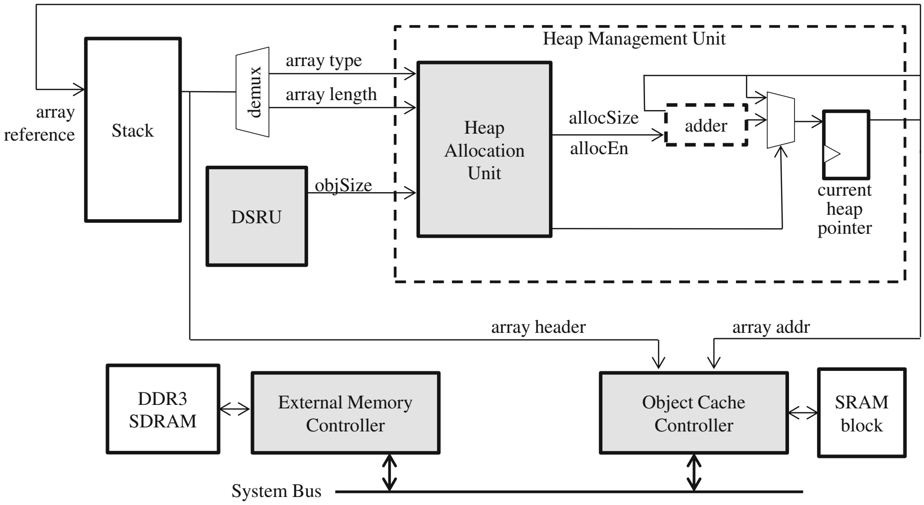


Fig. 6. The memory manager architecture of JAIP without the GC.

In short, the tracing collector returns all the local references to the unused memory block list unless the reference is a return value to the caller method. To achieve this goal, we expand the heap allocation unit in Fig. 6 to the architecture in Fig. 7. The object allocation controller is responsible for allocation of a new object on the heap and enters the object into the on-chip GC table. The GC table can be accessed by the GC controller for unused object collection upon the return of a method. Note that to hide the overhead of the GC, the GC controller must be able to access the GC table concurrently to the operation of the object allocation controller. Hence, we use a two-port memory for the GC table. Another on-chip memory in Fig. 7 is the GC method stack memory. The GC controller maintains this memory exclusively. During the execution of a method, this memory records all objects allocated locally and whether they are assigned to references outside the scope of this method. Upon the return of the method, the GC controller will go through the list of objects and return the memory blocks to the unused memory list if possible. Note that the collection process is executed in parallel to the normal bytecode execution pipeline.

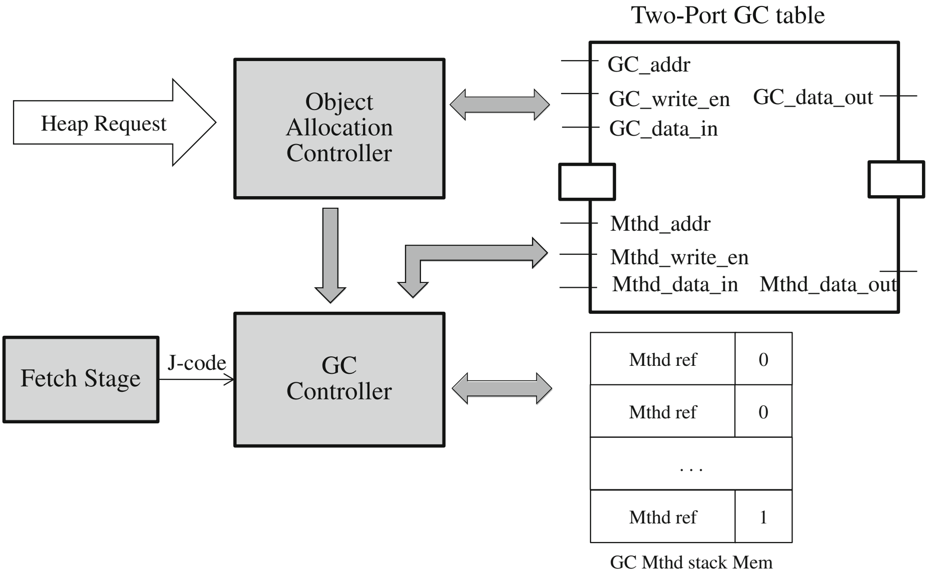


Fig. 7. The garbage collector of JAIP.

The GC controller will merges consecutive free memory block in the GC table into a larger block. However, it does not move the occupied memory blocks to create larger unused blocks because the cost would be too high for embedded applications. Note that the GC algorithm used in JAIP is not a complete garbage collector. It only collects unreferenced objects created by a method upon the return of the method to the caller. The reason this algorithm is chosen is mainly because it has very low runtime overhead and can be integrated into the existing memory manager of JAIP without major modification to the overall microarchitecture.

2.5 Dynamic Symbol Resolution Unit and the I/O Subsystem

In Sect. 1.2, we mentioned that the DSRU can provide a direct interface to the I/O subsystem of a hardwired Java VM. In this subsection, we use the JAIP DSRU as an example to explain the details. Since most modern operating systems and processors adopt the memory-mapped I/O model to manage I/O devices and accelerators, naturally, accesses to I/O devices are achieved using memory read/write operations in the I/O subsystem address space. Java uses the symbol space realized by the DSRU to replace the concept of the address space. Therefore, for a hardwired Java VM, the I/O subsystem can be integrated seamlessly into the DSRU logic. A method call in Java can be transform directly by the DSRU into control signals wired to a hardware device through some routing box (similar to the interconnect module of the ARM AXI bus protocols). Figure 8 shows the state-diagram of the controller of the DSRU of JAIP. When a program invokes a method, the controller begins at the 'IDLE' state and begins the symbol resolution process. When the DSRU determines that the target of the method invocation is for a native method implemented in hardware, it will enter the state of ‘Invoke HW Logic.’ This state will trigger the I/O subsystem manager to send appropriate hardware signals to the target device. Currently, all the hardware native methods of JAIP are determined at synthesis time. The string accelerators and the multi-thread managers of JAIP are invoked using such facility.

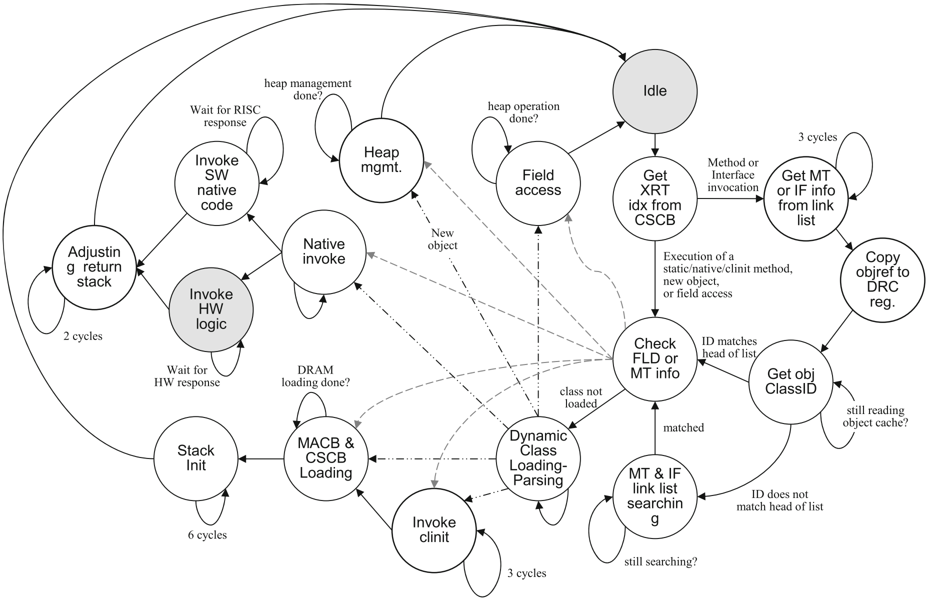


Fig. 8. The state diagram of the dynamic resolution controller of JAIP. XRT stands for ‘cross-reference table,’ MT stands for ‘method,’ IF stands for ‘interface,’ DRC stands for ‘dynamic resolution controller,’ and FLD stands for ‘field.’

3 Multi-core Integration of JAIP

3.1 The Multi-core Thread Manager

In order to integrate multiple JAIP cores into one application processor, we must modify the microarchitecture of JAIP. The multi-core capable JAIP core is shown in Fig. 9. The new addition to the original JAIP core is the Inter-Core Communication Unit (ICCU). The interactions between various components of the JAIP core and the ICCU are illustrated in Fig. 10. In the Java programming language, an object belongs to the “Thread” class can register its own execution context by invocation of the Thread.start() method. At runtime, the Dynamic Symbol Resolution Unit (DSRU) of JAIP will resolve the method invocation of start() and trigger a hardwired signal to the thread manager unit of the local JAIP core that executes the start() method. Such direct invocation of a hardwired logic through the dynamic resolution unit is called the Hardware Native Interface (HNI). In the original single-core JAIP, the local thread manager will handle the thread creation requests by itself and register a new entry in its local task queue. However, for a multi-core capable JAIP, the thread creation request cannot be handled locally. Instead, the request will trigger the HNI invocation of the ICCU, and the request signal will be passed to the Data Coherence Controller (DCC). The DCC then talks to a global thread manager to request for the creation of a new thread. The global thread manager will assign the new thread to a JAIP core based on the depth of its local task queue.

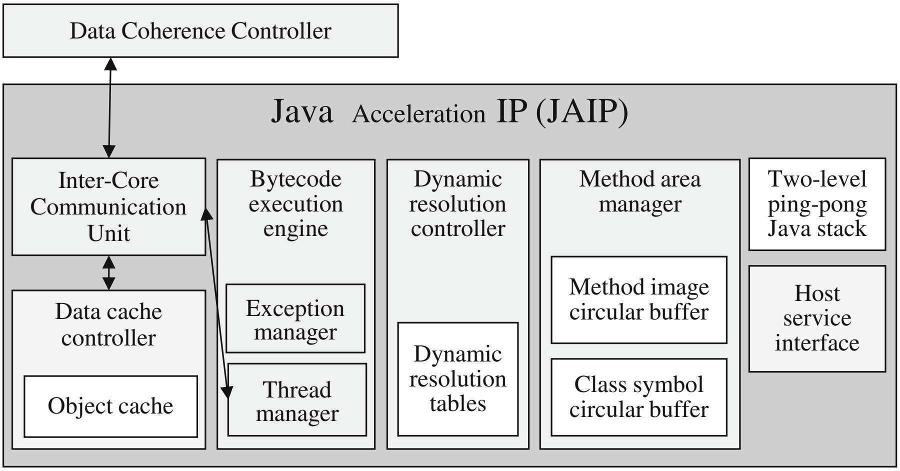

Data Coherence Controller
Java Acceleration IP (JAIP)
Inter-Core Communication Unit
Data cache controller
Object cache
Bytecode execution engine
Exception manager
Thread manager
Dynamic resolution controller
Dynamic resolution tables
Method area manager
Method image circular buffer
Class symbol circular buffer
Two-level ping-pong Java stack
Host service interface

Fig. 9. Modifications required to a JAIP core to enable multi-core integration.

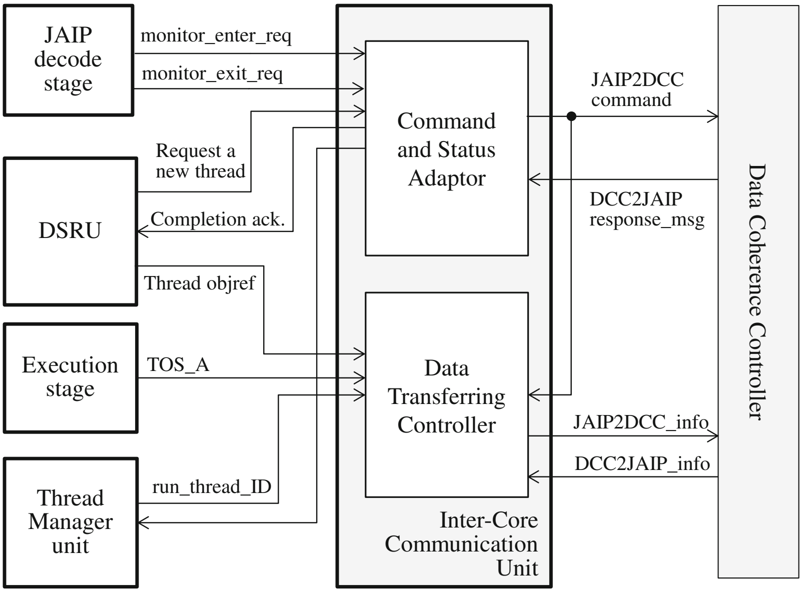


Fig. 10. Signaling between ICCU and other components of JAIP.

In addition to thread creations, the Java language also defines standard ways for synchronization. In short, each Java object contains a lock (similar to mutex in other programming language). Synchronization can be achieved explicitly through the acquisition of the lock in an object, or implicitly through invocation of a synchronized method. Similar to the thread creation problem, the acquisition of a lock cannot be handled locally since two threads requesting the same lock may be running on different JAIP cores. Therefore, such locking requests will also be passed to the ICCU for multi-core mutex operations. However, this time, the ICCU is not activated by a HNI invocation from the DSRU because the lock request is triggered by the execution of a "monitor" bytecode. Therefore, the lock request is originated from the decode stage of the bytecode execution engine, as shown in Fig. 10.

The integration of four JAIP cores into the multi-core application processor, JAIP-MP, is shown in Fig. 11. In the SoC, we only need one copy of DCC and global thread manager. The combination of these two hardware logic is referred to as the multicore coordinator of the JAIP-MP. Each JAIP core has its own ICCU. The local cache controller of each JAIP core will forward its cache block update status to the DCC so that the DCC can inform other cache controller to update their cache blocks if necessary. This is an efficient way to guarantee cache coherence when there are only few processor cores. However, to simplify the implementation of the coherent object cache, each cache controller adopts a write-through policy. This is different from the original single-core JAIP presented in [3], where a write-back policy is used. The write-through cache policy does hinder the single-core performance slightly. Nevertheless, the overall system performance still scales up fairly well.

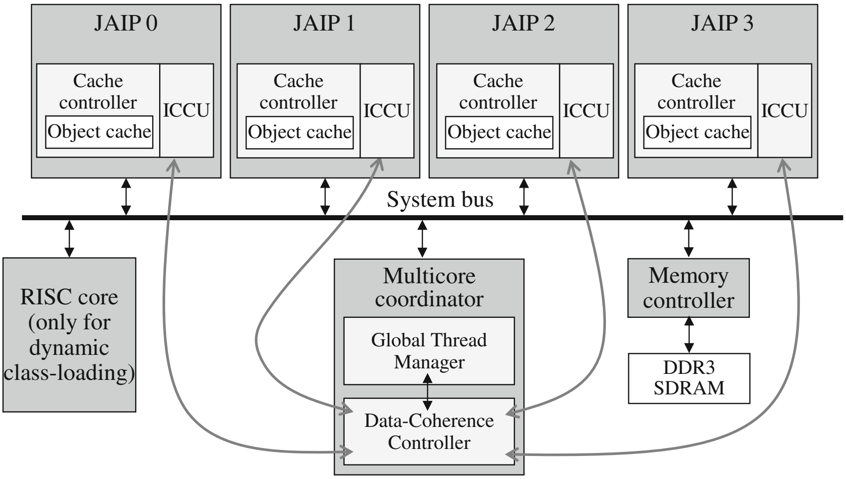


Fig. 11. Integration of four JAIP cores into a multi-core JAIP-MP SoC.

3.2 The Data Coherence Controller Architecture

The detail architecture of the DCC is shown in Fig. 12. It is composed of four submodules. The cache coherence controller maintains the data consistency across the object heap controllers of each core. The heap controller adopts the least-recently used policy and write-through strategy for caching of Java heap objects. The mutex controller serially decodes requests sent by the JAIP cores and activates corresponding submodule. The thread assignment controller (TAC) is responsible for load balancing among all JAIP cores. When a JAIP core invokes the Thread.start() method, the TAC will forward its special-purpose registers to the JAIP cores with the least number of ready threads. The Lock Object Accessing Controller (LOAC) shown in Fig. 13 maintains the information of waiting threads associated with each occupied lock object.

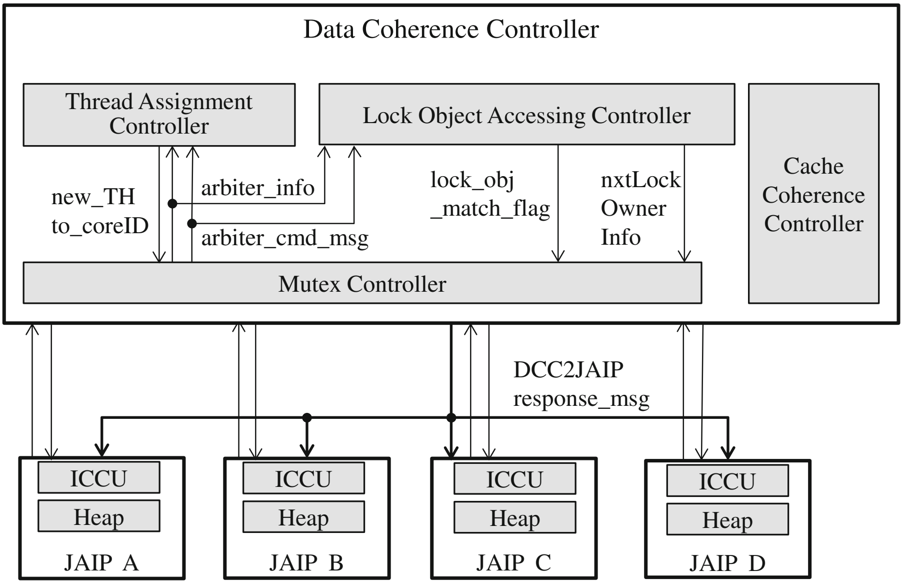


Fig. 12. The block diagram of the DCC.

When several JAIP cores try to request locks on the same mutex concurrently, the mutex controller uses a fixed-priority policy to determine which core can lock the mutex. Currently, the JAIP core with a smaller ID has a higher priority. The mutex controller supports three types of requests: dispatching a new thread, acquiring a lock object, and releasing a lock object. Either the TAC or the LOAC will be activated after the mutex controller determines the type of the request.

When any of the JAIP cores issues a request for the dispatching of a new thread, the TAC should determine a JAIP core to handle the new thread. In order to determine the

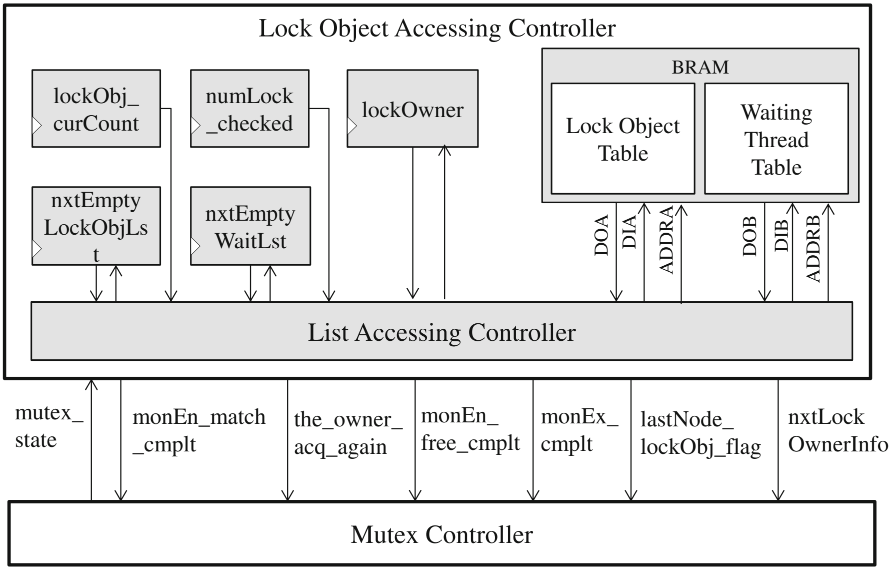
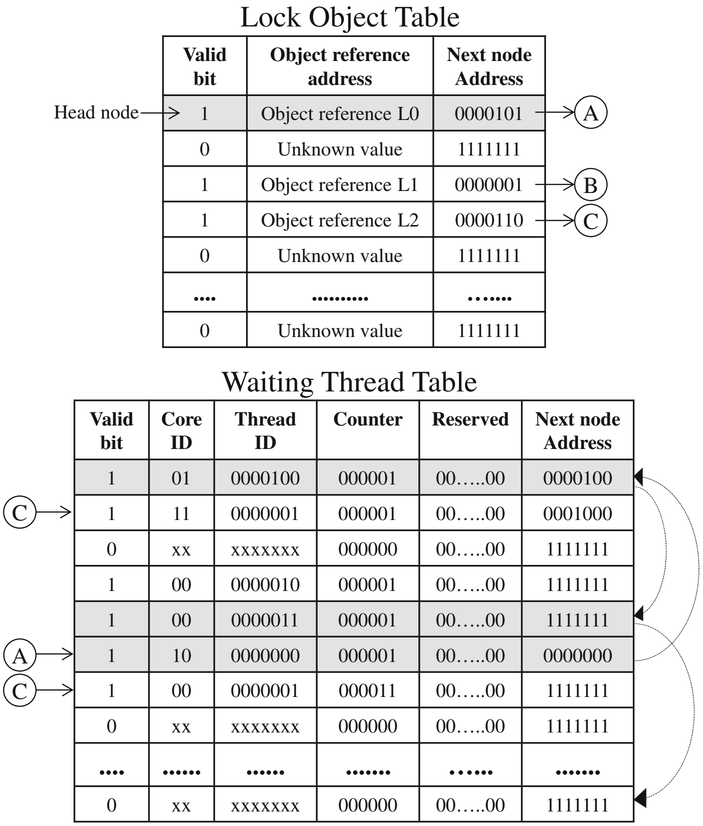


Fig. 13. The block diagram of the LOAC.

current number of active threads in each JAIP core, the TAC maintains a table. The table indexed is the ID of the JAIP core, and its entries store the current number of active threads of each core. The TAC will always assign the new thread to the lowest ID JAIP core that has the fewest number of ready threads. The TAC will inform the MHC to send a response signal to the chosen JAIP core with some essential information of the new thread. The ICCU of the JAIP core may process the information by decoding the response signal. Finally, the ICCU activates the thread manager unit of the JAIP core to add the new thread into its local thread queue.

Figure 14 is an example of the link lists maintained by the LOAC, which consists of a lock object table, a waiting thread table, and a few internal registers. Each occupied lock object maintains a linked list in these two tables. The head node of the linked list of a lock object begins at an entry in the lock object table, and the rest of the linked list nodes are entries of the waiting thread list. Each entry in the link list (except for the head node) represents a thread that is performing a lock operation on the object. The first thread in the linked list is the link list is the current owner of the lock. As soon as any thread in one of the JAIP cores tries to lock a Java object, the mutex controller will send a lock object L_n to the LOAC. The LOAC will look for the object address of an entry that matches L_n in the lock object table. Once the matched entry is found, the information must be recorded in the waiting thread table. Each entry contains the IDs of the JAIP core and the thread. The new entry is appended at the end of the link list. If the request from a thread is to release the lock object L_n, the LOAC will remove the thread from the link list. If any other thread is waiting for the same lock object L_n, the LOAC will make the second thread in the link list become the current owner of the lock object.



Lock Object Table

Valid bit	Object reference address	Next node Address
1	Object reference L0	0000101
0	Unknown value	1111111
1	Object reference L1	0000001
1	Object reference L2	0000110
0	Unknown value	1111111
....	....	....
0	Unknown value	1111111

Waiting Thread Table

Valid bit	Core ID	Thread ID	Counter	Reserved	Next node Address
1	01	0000100	000001	00…..00	0000100
1	11	0000001	000001	00…..00	0001000
0	xx	xxxxxxx	000000	00…..00	1111111
1	00	0000010	000001	00…..00	1111111
1	00	0000011	000001	00…..00	1111111
1	10	0000000	000001	00…..00	0000000
1	00	0000001	000011	00…..00	1111111
0	xx	xxxxxxx	000000	00…..00	1111111
....	......	......	....	......	....
0	xx	xxxxxxx	000000	00…..00	1111111

Fig. 14. Data structures maintained by the LOAC.

4 Experimental Results

The proposed architecture has been implemented on a Xilinx ML605 platform with a Xilinx Virtex6 XC6VLX240T FPGA. The RTL model of the JAIP core and the DCC logic are written in VHDL. Four JAIP cores and one DCC logic are integrated into the application processor using Xilinx XPS 13.4. The synthesis tool is Xilinx XST 13.4 and the target clock is 83.3 MHz. According to the place-and-route timing report of the Xilinx tools, the critical path of the system is currently at the execution stage of JAIP, from the customized four-port stack memory to ALU and then back to the four-port memory. The target frequency is chosen at 83.3 MHz due to some restrictions for DDR DRAM support on the development boards. The FPGA resource usages of JAIP and DCC are shown in Table 1.

Table 1. Logic usage of a JAIP core and DCC on a Virtex6 FPGA device.

FPGA logic units	LUT6s	Flip-flops	BRAMs
JAIP (per core)	12,580	5,912	34
DCC	663	449	1

Note: LUT6 means a six-input lookup table in a logic cell of a Xilinx device.

4.1 Single-Core Multithread Performance Evaluation

To evaluate the multi-threading performance of the proposed JAIP, we used the multi-threading benchmark programs from the JemBench suites [23]. These test programs are explained as follows. The 'Dummy' test creates multiple threads to execute busy loops for 5000 iterations. For the 'Matrix' test, each thread computes the multiplication of two 20-by-20 matrices. The 'N-Queens' test solves the N-Queens puzzle for N = 13 in each thread. For each test programs, the test scores roughly stand for the number of iterations each test program can execute per seconds by all threads. However, the scores are associated with quantization noises from the partition of subtasks across multiple threads and from the synchronization operations. In short, the drop in scores from single-thread test to multiple-thread tests is not entirely due to the context-switching overhead. Sun's CVM-JIT [24] running under Linux kernel 2.6.38 on an 83.3 MHz PowerPC 405 processor is used as the comparison point. JIT compilation is a very popular technique for Java program acceleration. Since the standard Java compilers (from Sun/Oracle) do not perform bytecode optimization on the compiled class files, a JIT-based VM could achieve significant speedup at runtime.

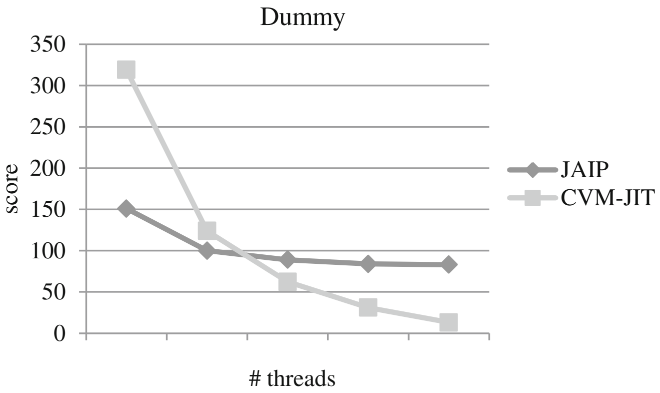


Fig. 15. JemBench scores of the 'Dummy' test on a single JAIP core.

From Figs. 15 and 16, one can see that the performance of CVM-JIT is higher for single-thread execution of the Dummy test and the Matrix test, JAIP has better performance when the number of threads becomes larger. Since in these tests, both JAIP and CVM-JIT executes using only one processor core, the scores drops naturally as the number of threads increases due to task division and synchronization overheads explained before. However, from these plots, it is quite clear that a software-based multithread mechanism such as the CVM-JIT has higher overhead in thread management. The performance drops significantly as the number of threads increases. For the Dummy test, JAIP outperforms CVM-JIT when the thread number is larger or equal to 4. For the Matrix test, JAIP matches the performance of CVM-JIT when the thread number is equal to 2 and outperforms CVM-JIT when the thread number is larger than 2. Finally, for the N-Queens test result shown in Fig. 17, JAIP outperforms CVM-JIT even if there is only one thread. This is probably because the NQueens program leaves little room for bytecode optimization by the JIT technique.

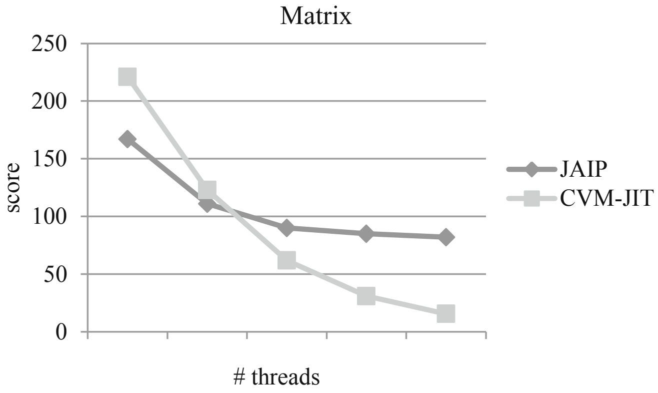


Fig. 16. JemBench scores of the 'Matrix' test on a single JAIP core.

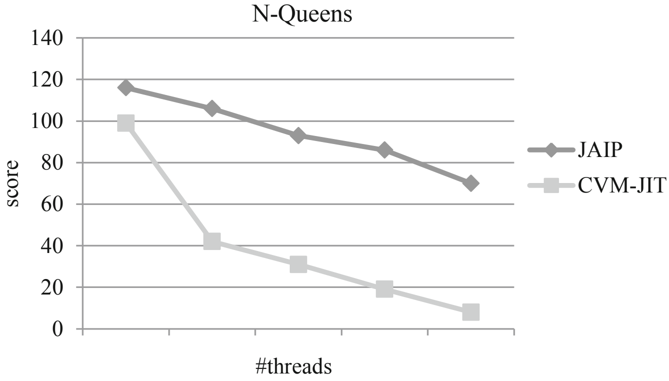


Fig. 17. JemBench scores of the 'N-Queens' test on a single JAIP core.

4.2 Multi-core Multithread Performance Evaluation

For multi-core multi-thread performance evaluation, we do not use CVM-JIT as a comparison point because the software platform does not support multi-core execution of Java applications. Here, we focus on the evaluation of performance scalability of JAIP-MP when the threads are distributed over multiple processor cores. As Table 2 shows, when the total number of threads is less than or equal to four, the JemBench score scales up fairly well (up to 3.69 times faster for the N-Queens test). When the total number of threads is more than four, the score of each benchmark naturally drops as the preemptive multi-threading mechanism of each JAIP core kicks in and there are synchronization overheads due to the way the benchmarks are designed. This is especially true for the N-Queens test.

Table 2. The multi-core JemBench scores of the parallel benchmarks.

# threads	Dummy	Matrix	N-Queens
1	151	167	116
2	298	240	225
3	374	395	330
4	491	498	428
5	410	425	311
6	373	412	251
8	362	399	262
12	359	366	212
16	340	327	195

Note: Larger number means better scores.

4.3 Synchronization Overhead

In the JemBench tests, context-switching overhead is not the only reason to cause the performance drop. If several requests are sent concurrently to the DCC of JAIP-MP, it takes several cycles for the mutex controller to decode the requests sequentially. In addition, to maintain data cache coherency, as soon as each entry is updated in any of the object heap controllers, the modified entry and its corresponding address are sent to the cache coherence controller and the main memory controller for cache validation among JAIP cores.

Tables 3 and 4 show the synchronization overhead of the proposed architecture under the Matrix test. The average overhead of a synchronization operation can be as small as tens of machine cycles.

Table 3. The execution time (in clock cycles) of acquiring a lock object.

# threads	4	6	8	12	16
Average	23.1	25.4	25.4	28.9	28.9
Worst-case	43	89	108	108	110
Best-case	9	9	9	9	9

Table 4. The execution time (in clock cycles) of releasing a lock object.

# threads	4	6	8	12	16
Average	20.2	21.6	21.8	28.4	28.7
Worst-case	43	76	85	92	98
Best-case	10	10	10	10	10

5 Conclusions and Future Work

In this chapter, we have presented an four-core Java processor, JAIP-MP. The uniqueness of JAIP-MP is that the key functions of an operating system kernel are implemented in hardware circuits. For thread management, the architecture supports arbitrary number of threads (limited by the on-chip TCB memory size), low context-switching overhead, small time quantum, and low synchronization overhead. The proposed architecture is implemented and verified on an FPGA platform. Experimental results show that the proposed design is very promising for embedded multi-thread applications.

For future work, we will look into the following directions. First of all, although the ping-pong buffer for context-switching is efficient performance-wise, it does impose heavy memory accesses. This may result in high power consumption. In the future, we will try to design a new architecture that can reduce the number of memory access per context-switch. Secondly, the coherent data cache in our current implementation only adopts one-level of cache hierarchy. Most general purpose processors nowadays adopt two or even three levels of cache hierarchy. It would be interesting to study the effects of a multi-level cache on the object-oriented programming model of the Java language.

Finally, current thread management design only uses a round-robin policy to maintain load balance. We will look into the design of a new architecture that can customize the thread distribution policy at runtime and allow for thread migration across different JAIP cores so that better runtime load balance can be achieved.

References

1. Ritchie, S.: Systems programming in Java. IEEE Micro **17**(3), 30–35 (1997)
2. Montague, B.R.: JN: OS for an embedded Java network computer. IEEE Micro **17**(3), 54–60 (1997)
3. Tsai, C.-J., Kuo, H.-W., Lin, Z., Guo, Z.-J., Wang, J.-F.: A Java processor IP design for embedded SoC. ACM Trans. Embed. Comput. Syst. **14**(2), Article 35 (2015)
4. Su, H.-C., Wu, T.-H., Tsai, C.-J.: Temporal multithreading architecture design for a Java processor. In: Proceedings of the IEEE International Symposium on Circuit and Systems (ISCAS 2014), Melbourne, Australia, June 2014
5. Schoeberl, M.: A Java processor architecture for embedded real-time systems. EUROMICRO J. Syst. Architect. **54**(1–2), 265–286 (2008)
6. Gruian, F., Schoeberl, M.: Hardware support for CSP on a Java chip multiprocessor. Microprocess. Microsyst. **37**(4), 472–481 (2013)

7. Brandner, F., Thorn, T., Schoeberl, M.: Embedded JIT compilation with CACAO on YARI. In: Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2009), Tokyo, 17–20 March, pp. 63–70 (2009)
8. Tyystjaervi, J., Saentti, T., Plosila, J.: Efficient bytecode optimizations for a multicore Java co-processor system. In: Proceedings of the 12th Biennial Baltic Electronics Conference, Tallinn, Estonia, 4–6 October 2010
9. Kreuzinger, J., Brinkschulte, U., Pfeffer, M., Uhrig, S., Ungerer, T.: Real-time event-handling and scheduling on a multithreaded Java microcontroller. Microprocess. Microsyst. **27**(1), 19–31 (2003)
10. Sun Microsystems: picoJava-II Microarchitecture Guide (1999)
11. Uhrig, S., Wiese, J.: jamuth: an IP processor core for embedded Java real-time systems. In: Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007), 26–28 September, Vienna, pp. 230–237 (2007)
12. Pitter, C., Schoeberl, M.: Towards a Java multiprocessor. In: Proceedings of the 5th ACM International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007), 26–28 September, Vienna, pp. 144–151 (2007)
13. Yan, L., Liang, Z.: An accelerator design for speedup of Java execution in consumer mobile devices. Comput. Electr. Eng. **35**(6), 904–919 (2009)
14. Hardin, D.S.: Real-time objects on the bare metal: an efficient hardware realization of the JavaTM virtual machine. In: Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2–4 May, Magdeburg, pp. 53–59 (2001)
15. Vijaykrishnan, N., Ranganathan, N., Gadekarla, R.: Object-oriented architectural support for a Java processor. In: Proceedings of the 12th European Conference on Object-Oriented Programming, Brussels, Belgium, pp. 330–354 (1998)
16. Lin, Z.-G., Kuo, H.-W., Guo, Z.-J., Tsai, C.-J.: Stack memory design for a low-cost instruction folding Java processor. In: Proceedings of the IEEE International Symposium on Circuit and Systems, 20–23 May, Soeul, Korea, pp. 3326–3229 (2012)
17. Lindholm, T., Yelling, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Longman Publishing Co., Inc., Boston (1999)
18. Chang, Y., Wellings, A.: Garbage collection for flexible hard real-time systems. IEEE Trans. Comput. **59**(8), 1063–1075 (2010)
19. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: an exercise in cooperation. Commun. ACM **21**(11), 965–975 (1978)
20. Bacon, D.F., Cheng, P., Shukla, S.: And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. Commun. ACM **56**(12), 101–109 (2013)
21. Srisa-an, W., Lo, C.-T.D., Chang, J.M.: Active memory processor: a hardware garbage collector for real-time Java embedded devices. IEEE Trans. Mobile Comput. **2**(2), 89–101 (2003)
22. Gruian, F., Salcic, Z.A.: Designing a concurrent hardware garbage collector for small embedded systems. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740, pp. 281–294. Springer, Heidelberg (2005)
23. Schoeberl, M., Preusser, T. B., Uhrig, S.: The embedded Java benchmark suite JemBench. In: Proceedings of the JTRES 2010, 19–21 August, Prague, Czech Republic (2010)
24. Oracle: Phoneme project webpage. Accessed 27 Sept 2011. https://java.net/projects/phoneme