國立交通大學

資訊科學與工程研究所

碩 士 論 文

嵌入式系統異質雙核心 Java 處理器設計

Design of Dual-Core Java Application Processor for Embedded Systems

研 究 生：郭子敬

指導教授：蔡淳仁 教授

中 華 民 國 101 年 7 月

嵌入式系統異質雙核心 Java 處理器設計
Design of Dual-Core Java Application Processor for Embedded Systems

研 究 生：郭子敬　　Student：Zi-Jing Guo

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

July 2012

Hsinchu, Taiwan, Republic of China

中華民國 101 年 7 月

摘要

本論文為 Java 處理器提出了一個異常處理的設計。雖然目前有許多關於 Java 處理器設計的相關研究，但是大部分的研究簡單地忽略了異常處理，而有些聲稱，符合 Java 語言的異常處理機制牽涉到複雜的行為而難以實作在硬體上。在這篇論文中，我們檢視了異質雙核心 Java 處理器對於例外處理的機制有優秀的硬體加速效果，進而提出了一個以硬體設計為主的異常處理機制，除此之外，為了能夠支援有效率的例外處理機制，我們還改善了此平台上二層級方法區域的設計，在實作出來的例外處理機制設計中，並不會影響正常程式的執行效能，而只有當程式真正發生異常時，才會增加異常處理的時間，更重要的是，由於異常處理的程序主要執行於 Java 核心，因此大大節省了處理器之間的溝通時間，我們實作上述完整的 Java 執行環境在 Xilinx ML-507 FPGA 開發板上，在最後的實驗數據結果顯示，我們提出的硬體異常處理機制非常適合用在 Java 嵌入式平台的環境並且有非常好的效能。

Abstract

This thesis presents the design of the exception handling architecture of a Java processor. Although there are many research publications on Java processor designs, there is no efficient implementation on Java exception handling circuitry. Most Java processor design papers simply ignore exception handling while some claims that a hardwired implementation of exception handling conforming to the Java language specification is quite complex to implement. In this thesis, we have proposed an efficient design of the Java exception handling mechanism and the associated two-level method area. We have also integrated the design into a heterogeneous dual-core Java processor. With the proposed two-level method area, the exception handling overheads are delayed to the time after an exception actually occurs. More importantly, the process of exception handling is mostly performed in the Java core with very little runtime overhead from the RISC core. As a result, the proposed design reduces the amount of inter-processor communication and circuit design cost of the Java core while enabling full support of Java exception handling. We have implemented the design on a Xilinx ML-507 FPGA platform. As the experiments show, the proposed design is very promising for embedded applications.

Acknowledgement

這篇論文的完成，首先必須要感謝我的指導教授蔡淳仁教授。在研究所的這兩年期間，老師提供了很多學習的經驗跟機會，在與老師討論的過程中，常常能了解到自己思考中的盲點與研究過程的缺失，此外，老師也提供參與國際會議的機會，讓我了解到國際上研究的趨勢，還有不同問題的解決方法，這些在面對自己的研究問題時都是很寶貴的參考經驗。在解決問題的過程中，學長們的意見也助益良多，透過跟學長們的討論，可以反覆檢視自己的設計，以及訓練表達的方法。另外還要感謝實驗室的同學，透過計畫的合作與研討，讓我在這兩年內認識了不同領域的知識。

Content

Chapter 1. Introduction............................1

1.1 Motivation...1
1.2 Java Runtime Systems 2
1.3 Scope of this Thesis 4

Chapter 2. Previous Works 5

2.1 Double-Issue Java Core 5
2.2 Dynamic Class Loading Mechanism 7

Chapter 3. Dynamic Class Parsing and Method Loading.....................8

3.1 Overview of the Dual-Core Java Application Processor 8
3.2 Overview of Dynamic Class Parsing and Method Loading.........10
3.3 Dynamic Class Parsing 12
3.4 Two-Level Runtime Image Design...13
3.5 Dynamic Method Loading...18
3.5.1 Class Lookup Table and Method Lookup Table 19
3.5.2 Architecture of the Method Area Manager 20

Chapter 4. The Implementation of the Exception Handling Unit......26

4.1 Overview...26
4.2 Exception Handling in the Java Language..................................27
4.3 Exception Handler Design for Dual-Core Java Processor 29
4.3.1 Stage 1 : Extraction of Exception Information 31
4.3.2 Exception Lookup Table 33
4.3.3 Stage 2 : Finding The Exception Routine 35
4.3.4 Stage 3 : Dynamic Method Loading 40

Chapter 5. Experimental Results...........41

5.1 Development Platform and Tools 41
5.2 Benchmark Analysis 42
5.2.1 Scenario One Benchmark Analysis..42

5.2.2 Scenario Two Benchmark Analysis 46
5.2.3 Scenario Three Benchmark Analysis 48

Chapter 6. Conclusions and Future Works52

Appendix: CLDC Exception Library 54

Reference 58

List of Figures

Fig 1. Traditional Java runtime system. 2
Fig 2. Four-stage pipelines of bytecode execution engine 5
Fig 3. Architecture of the two-level Java stack. 6
Fig 4. Control state machine for method invocation resolution in [6]. 7
Fig 5. Overall system diagram of the dual-core Java processor. 8
Fig 6. Processes of dynamic class parsing and method loading. 10
Fig 7. Processes of class parsing. 12
Fig 8. Two-Level runtime image. 14
Fig 9. Method area format. 15
Fig 10.Adjust the stack frame when invoking the method 'A' with the method image headers: parameter_count and max_local. 16
Fig 11. The processes of dynamic method loading. 18
Fig 12. Class lookup table and method lookup table 19
Fig 13. Architecture of the method area manager. 21
Fig 14. FSM in the method area controller. 23
Fig 15. An example of a Java program that throws an exception. 29
Fig 16. Three stages of exception handling process. 31
Fig 17. Architecture of exception lookup table. 34
Fig 18. Architecture of exception handler that cooperates with method area manager. 36
Fig 19. Exception handling steps in the stage two. 37
Fig 20. Processes of Java stack unwinding. 39
Fig 21. A test program with exception thrown by bytecode "athrow". 43
Fig 22. A test program with exception thrown by JVM. 43
Fig 23. A test program containing five methods and the last method triggers an exception. 46
Fig 24. A test program containing five methods and each method contains an exception routine. 49

List of Tables

Table I. Synthesis information of Java core. 41
Table II. Time overheads of exception handling process in Scenario One programs........ 44
Table III. Time overheads of adding an exception routine in the same try block. 45
Table IV. Overheads of exception handling process in the benchmark of scenario two. 47
Table V. Overheads when adding an additional searched method. 47
Table VI. Time overheads of exception handling process in the benchmark of scenario two. 50
Table VII. Overheads of additional try block in each method. 50
Table VIII. The Sun CVM runs on a 300 MHz PowerPC platform and the Java processor in dual-core Java remains 83.3 MHz. 501

Chapter 1. Introduction

1.1 Motivation

For the past few years, Java is getting considerable attentions as a programming language for embedded system application platform because of the strong portability of applications across different operating systems and processors. Adopting Java as a standard application langrage also can enjoy the rich API supports. Not to mention the fact that the Java language is a well-designed object-oriented programming language and up to 40% more productivity than C++ [1]. As a result, many embedded multimedia systems are Java-based platforms, such as DVB set-top boxes, Blu-ray players, and smart mobile phones.

In addition, the Java language provides a comprehensive exception handling mechanism. Without exception handling, runtime errors can only be checked and processed by numerous if-then-else statements. Such coding style is both inefficient and error-prone. For now, there are many techniques to accelerate the executions of Java programs in different models of Java runtime systems. Many models adopt a hardware Java processor to accelerate the regular Java programs. However most researches do not cover the design of exception handling circuitry [2][3][4]. In [5], a hardware-software co-design of a Java Virtual Machine is proposed with the exception handling capability. The paper admits the complication of designing special circuitry for handling Java exceptions due to call stack searching. Thus, it implements exception handling on the general-purpose processor core. However, such design makes exception handling very inefficient. This might be fine for non-recoverable exceptions. However, for exceptions that will not stop the execution of the program, the design would be unacceptably inefficient.

1.2 Java Runtime Systems

A Java runtime system is composed of a Java virtual machine (JVM) [14] and a set of standard class libraries. Sun Microsystems has defined Java Micro Edition (Java ME) [6] framework which is designed specifically for embedded systems. Target devices range from industrial controllers to mobile phones and set-top boxes. Furthermore, the Java ME has been divided into two base configurations, one to fit small mobile devices and one to target towards more capable devices like smart-phones and set top boxes. The configuration for small devices is called the connected limited device configuration (CLDC) [5] and the more capable configuration is called the connected device configuration (CDC) [7].

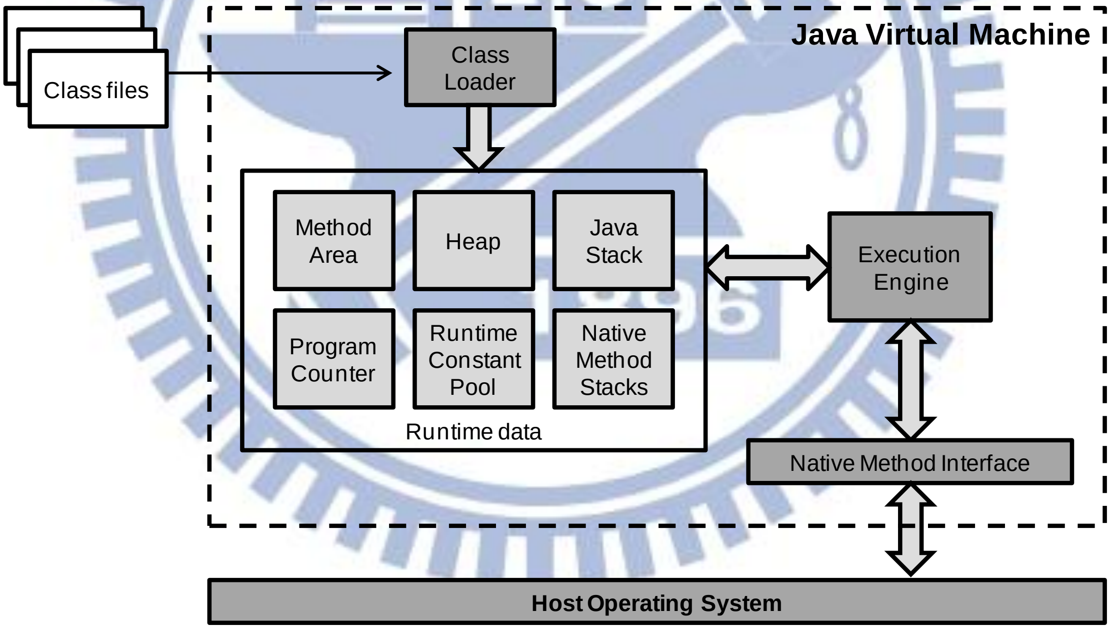


Fig 1. Traditional Java runtime system.

A traditional JRE is shown in Fig 1. The JRE contains software JVM [7] which relies on a full-blown OS to execute a Java program. In the Java virtual machine specification, the three major components are the class loader, the runtime data, and the execution engine [9]. The class loader is a component for loading specific classes or interfaces. The runtime data area is the major memory space that the JVM organizes to execute a program. And the execution engine is the component that is responsible for executing the instructions contained in the methods of loaded classes

In addition to the traditional JRE, there are many solutions for improving the performance of JRE for embedded systems. The solutions for improving the time and space overheads of JRE can be roughly divided into three approaches. The software Just-in-Time (JIT) compilers, the hardware-based co-processors (e.g. ARM Jazelle), and stand-alone Java processors (e.g. picoJava) are three common approaches for embedded environments.

The execution is improved significantly by using JIT compilation techniques [10][11] to translate Java bytecodes to native codes at runtime. Although, the speedup by the JIT compiler is high, JIT requires extra memory [11] and imposes extra compilation overheads for class loading. The compiler itself along with the memory footprint for the compilation may require a few megabytes of storage [10]. Therefore, this approach is less suitable for embedded devices, which have strong memory constraints. An interesting effort is taken on by Google when picking a solution for their Android application environment. The Dalvik VM [12] is a register-based virtual machine which is not binary compatible with the JVM. Java application class files must be converted into Dex file format before execution. It is shown that a register-based VM can be 32.3% more efficient than a stack-based VM when executing standard benchmarks by an interpreter, at the expense of 25% larger binary code size of the benchmark programs [13].

Another approach is to build a purely hardwired Java processor to handle nontrivial tasks. There are several Java processor solutions such as picoJava [11], Komodo [15], jHISC [12], and JOP [17]. Because the processor is custom-designed to match the stack machine model of the JVM, it can deliver better bytecode execution performance than that of a general-purpose processor running a Java interpreter. However, some operations are inefficient in pure hardware solutions such as class parsing, memory managing, file accesses …etc .Therefore a hardware software co-design approach is proposed in [18][19]. The co-design JRE includes a RISC core and Java core. The Java core is designed to efficiently execute the bytecodes instructions. And the RISC core handles the complicate services that are better performed in software solutions.

1.3 Scope of this Thesis

In this thesis, we propose the hardware exception handling architecture for the dual-core Java processor [18][19]. The exception handling architecture adopts the stack unwinding approach to find a proper exception routine. The whole process is mainly performed in the Java processor. Therefore, the overhead of exception handling is much lower than that of the design in [6]. The proposed design is based on our previous work of a Java processor [7]. The prototype of the dynamic class loading mechanism and method area management was previously proposed in [6]. However, the design in [7] used a class-based method area caching system, which is not very efficient for exception handling. To reduce the on-chip memory requirement for various lookup tables for the exception handling support, we have changed the method area buffering system from a class-based scheme to a method-based scheme.

The organization of this thesis is as follows. Chapter 2 describes the previous design of the Java embedded platform. Chapter 3 describes the modifications of the two-level runtimes images designs. Chapter 4 describes the exception handling unit for the Java processor in detail. And the performance analyses and benchmark results is discussed in chapter 5. Finally, conclusions and discussions are given in chapter 6.

Chapter 2.   Previous Works

2.1 Double-Issue Java Core

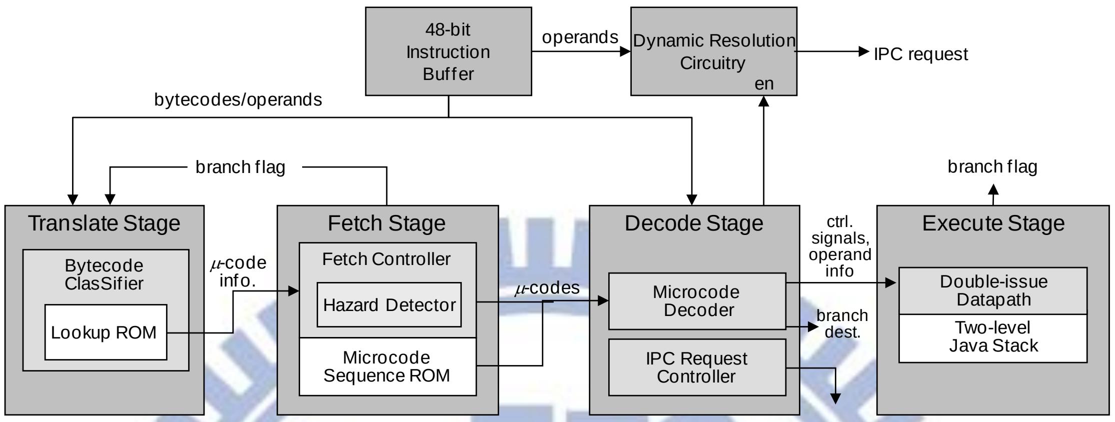


Fig 2. Four-stage pipelines of bytecode execution engine

Ko et al [18][19]. proposed a design of double-issue Java bytecode execution engine (BEE) core. The architecture of Java bytecode execution engine is shown in Fig 2. The BEE core adopts four-stage pipeline architecture with translate, fetch, decode, and execute stages. The Java BEE core is a stand-alone IP not tied to any host processor architecture. Therefore, it is easy to integrate the BEE core into any processor that supports interrupt-driven inter-processor communications.

In the translate stage, each byte of incoming two Java bytecodes will be classified into a simple instruction, a complex instruction or an operand. If the incoming byte is a simple instruction, the translate stage will translate it to a single microcode instruction and pass it to the fetch stage. If the incoming byte is a complex instruction, the translate stage will pass the starting address of the corresponding microcode sequence to the fetch stage. If the incoming byte is an operand, the operand values will be extracted directly from the instruction buffer at the decode stage. The rest pipeline will then try to execute two complete microcode instructions per cycle whenever possible.

The fetch stage is in charge of sending two microcode instructions per clock cycle to the decode stage. If a complex instruction address is obtained from the translate stage, a sequence of paired microcodes deriving from ROM will be sand to the decode stage. Note that any one sequence of microcodes in ROM is designed to be double-issued (some paired microcodes may have one NOP). The decode stage will then decode the microinstructions (up to two instructions per cycle) and setup the data path accordingly. Finally, the execute stage performs operations on Java stack.

To enable more double-issued instruction, a special two-level Java stack memory is purposed in [24]. As shown in Fig 3.The first level of Java stack is composed of three registers that store the top-three elements of the Java stack. The second level of Java stack is composed of two dual-port on-chip memory blocks organized in an interleaving structure that implements a contiguous Java stack. With this design, accesses to the most frequent local variables will not cause structure hazard to the double-issued datapath. As the result, the execution of Java program can gain more double-issued rate.

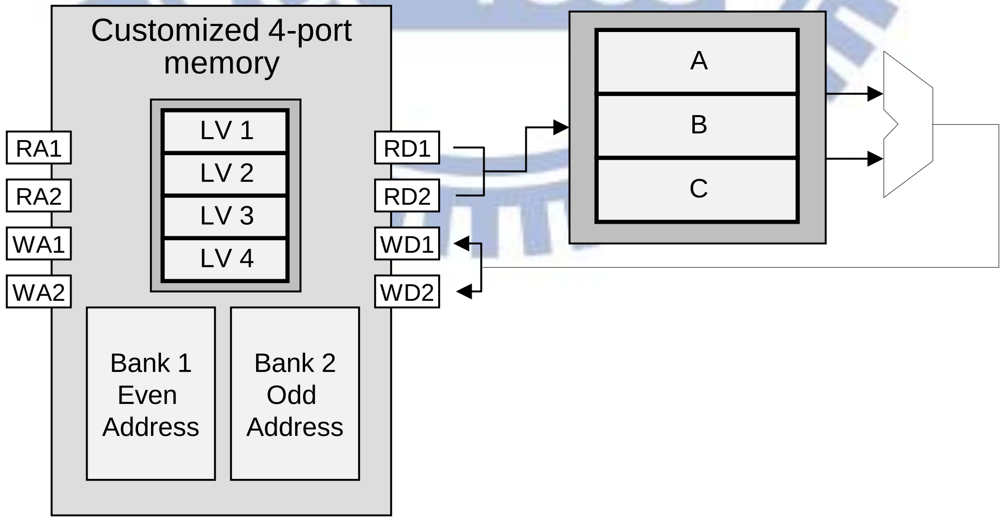


Fig 3. Architecture of the two-level Java stack.

2.2 Dynamic Class Loading Mechanism

Hwang et al. proposed a dynamic class loader [20][21] for the dual-core Java processor [22][23]. In the previous system [20][21][22][23], the class loader parses all class files and converts them into runtime information images and reserves all resolution information in the constant pool of the image. Hwang also designed a dynamic resolution state machine to handle symbol resolution of constant pool data. The states of symbols resolutions triggered by method invocation and field data access are shown in Fig 4.In the design, the class file is firstly parsed by the software parser in the RISC core before the class is accessed by Java core. Such class parsing process involves translating each symbol to a reference data and locating them to runtimes images. The design offers the bytecode execution engine a faster way to result the operand to its reference data in runtime execution.

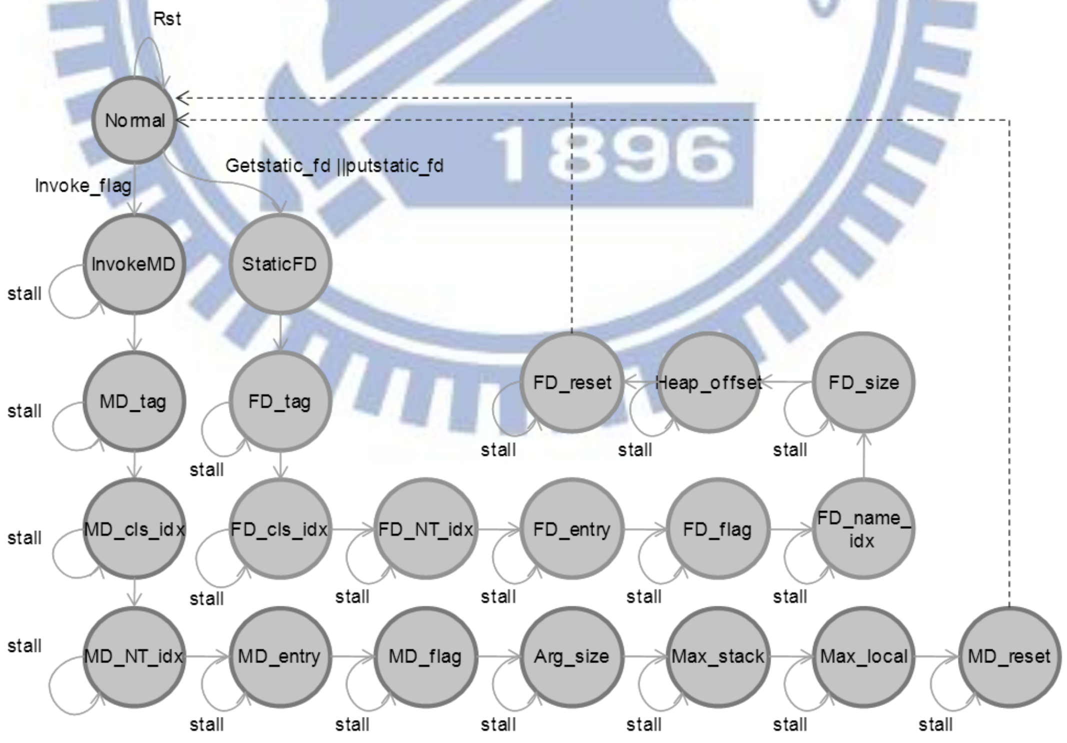


Fig 4. Control state machine for method invocation resolution in [3].

Chapter 3. Dynamic Class Parsing and Method Loading

3.1 Overview of the Dual-Core Java Application Processor

In this section, we present the overview of modified dual-core Java application processor, which is derived from the previous work [20][21][22][23]. To run a Java program in the proposed system, the instruction bytecodes in the program are fully executed in a high performance in the accelerator Java core. If some of the executed bytecodes encounter complicated processes that are not suitable in hardware solutions, such process will be performed by the software interrupt service routines in the RISC core. For an example, the Java language behavior of the instruction bytecode “new” is to instantiate a class object. The process of object instantiation involves heap memory managing and class files information searching, which are better for software solutions. Therefore when the Java core executes the bytecode “new”, an interrupt is triggered to RISC core to execute the specific service routine to accomplish the bytecode “new”.

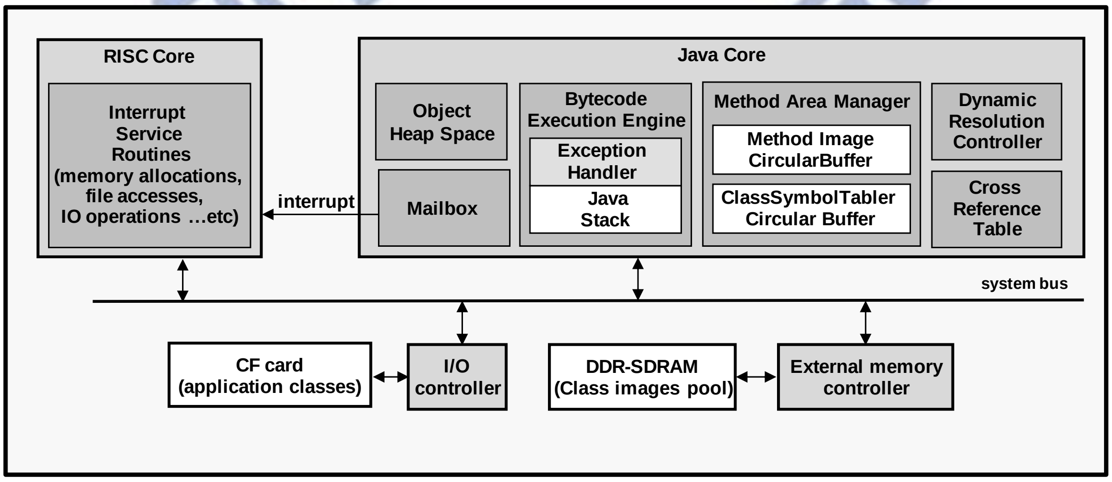

RISC Core
Interrupt Service Routines (memory allocations, file accesses, IO operations …etc)
interrupt
Java Core
Object Heap Space
Mailbox
Bytecode Execution Engine
Exception Handler
Java Stack
Method Area Manager
Method Image CircularBuffer
ClassSymbolTabler Circular Buffer
Dynamic Resolution Controller
Cross Reference Table
system bus
CF card (application classes)
I/O controller
DDR-SDRAM (Class images pool)
External memory controller

Fig 5. Overall system diagram of the dual-core Java processor.

As shown in Fig 5, the RISC core provides interrupt service routines that handle complicate tasks such as memory allocations, file accesses, IO operations …etc. In the Java core, each specific interrupt routine is triggered through the mailbox. There are another five main components in the Java core. The object heap space is the memory space allocated for class instantiation objects in program runtime execution. The instruction bytecodes are executed in an instruction-folding approach by a four-stage-pipeline bytecode execution engine. The bytecode execution engine contains a Java stack which is a special two-level stack memory proposed in [24]. In addition, the design of the exception handling architecture proposed in this thesis is based on the Java stack. The detail architecture of exception handler is discussed in the Chapter 4.

Another proposed feature, the method area manager, is the key component to facilitate the exception handling mechanism and to adopt a more efficient approach to buffer the essential runtime resources. To be memory-efficient in usage of the runtime resources, the runtime resource is design to a two-level memory hierarchy. The first-level space is on-chip circular buffers and the second-level space is the class images pool locating at the external DDR-SDRAM. In more detail, a runtime image in the first-level space is named as method area composed of class symbol table and method image. To be efficient retrieve the data in the method area, the class symbol table and the method image are separately buffered into two circular buffers. In the method image circular buffer, each method image contains the instruction bytecodes of the method and eight bytes method headers. The method image circular buffer is directly accessed by the bytecode execution engine to efficiently retrieve and execute the instruction bytecodes.

Another essential runtime resource, the class symbol table, contains reference data that may be referenced by some instruction bytecodes. The class symbol table provides a fast way to resolute class symbols from operand following by a bytecode. Some complicate class symbols are referenced indirectly and placed at the cross reference table. In this case, the reference data in the class symbol table is the address of the cross reference table. The process of symbol resolution is handled by the component dynamic resolution controller, which is proposed in [20][21]. The following sections discuss the usage of the two-level runtime images. And the method area manager dealing with the buffing process is described in section 3.5.

3.2 Overview of Dynamic Class Parsing and Method Loading

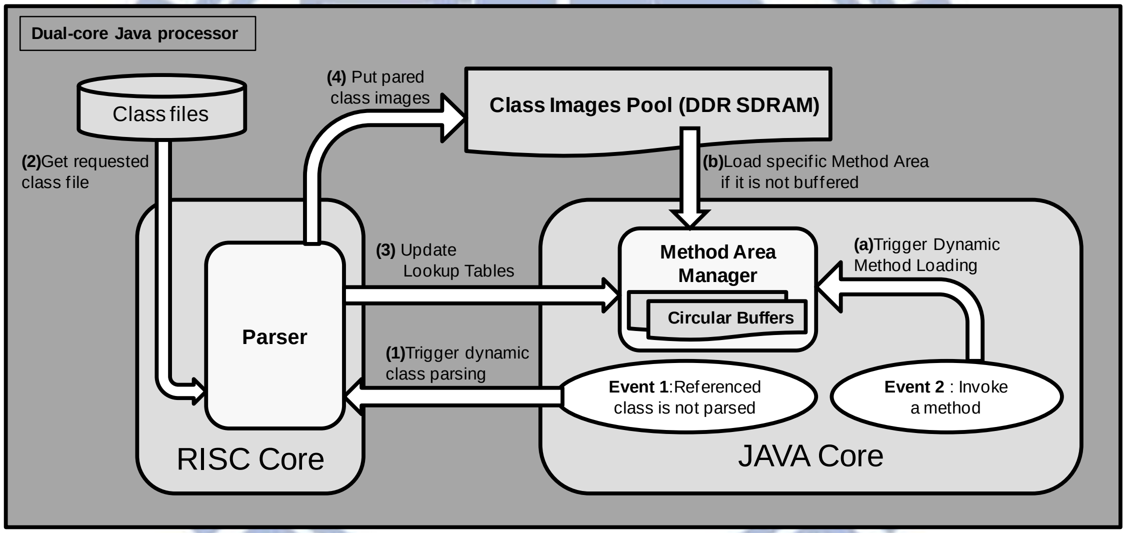

Dual-core Java processor
Class files
(2)Get requested class file
(4) Put pared class images
Class Images Pool (DDR SDRAM)
(b)Load specific Method Area if it is not buffered
(3) Update Lookup Tables
Method Area Manager
Circular Buffers
(a)Trigger Dynamic Method Loading
Parser
(1)Trigger dynamic class parsing
Event 1:Referenced class is not parsed
Event 2 : Invoke a method
RISC Core
JAVA Core

Fig 6. Processes of dynamic class parsing and method loading.

In the proposed design, the class file is parsed to a runtime image before the class can be referenced. In the life-time of a Java program, there are two important events related to the runtime image. As shown in Fig 6, the first event, dynamic class parsing, is triggered when a referenced class has not been parsed yet. And the second event, dynamic method loading, is to ensure the parsed image of the invoked method is buffered in the circular buffers before the method is executed by the bytecode execution engine.

In the first event of dynamic class parsing, the process of parsing a class file involves retrieving class files, class symbols linking, creations of tables…etc, which are not suitable for implementation using hardware IP. Therefore, such class parsing process is performed by the interrupt service routine on the RISC side. After parsing a class file, the generated class runtime image will be located at the class images pool. In the framework of runtime images, the class image pool located in the in the DDR SDRAM is the second-level memory space. And the first-level space contains images accessed by the Java core, which is the subset containing essential data for executing a method of the class image in second-level memory space.

The second event, dynamic method loading, is triggered to ensure a specific method area is buffered in the Java core before a method is executed. If the request image has been buffered before and the buffer containing the image is not overwritten, such process is to update a register locating the method area in the buffer. In the other case, if the runtime image is not buffered, the specific image will be dynamically loaded into the circular buffers. In the end of dynamic method loading, the essential data for executing a method are ensured in the circular buffers and waits to be executed.

The process of the dynamic method loading is handled by the method area manager. The method area which is a subset of a class image contains only the essential data to execute a method. In our previous work [20][21][22][23], the whole runtime image of a class is loaded into on-chip buffers. Note that the class image contains all methods' instruction bytecodes, the disadvantage of this approach is obvious since some method bytecodes that are not used would also be loaded. To reduce the loading overheads, the class image is divided into two parts: class symbol table and method images. In the proposed design, the method area containing only the class symbol table and the invoked method image are loaded in dynamic method loading, which is discussed in section 3.5.

3.3 Dynamic Class Parsing

The dynamic class parsing mechanism is implemented in a hardware-software co-designed approach. In the Java core, the method invocation bytecode may reference to an unparsed class. For example, a bytecode "invokstatic" is used to invoke and execute a specific method of a class object. The invoked method is identified by the index operand followed by the "invokstatic" instruction. Through dynamic symbol resolution, the index operand is resolved to an invoked class ID and a method ID. If the retrieved method ID is equal to zero, it means that the invoked class is not parsed. And the dynamic resolution controller will trigger an interrupt service to the software parser in the RISC core. The interrupt is triggered with a class ID argument so that the parser can parse the specific class file. After accomplishing the parsing process and locating the runtime image to class images pool, the Java core can reference the requested class and continues execution.

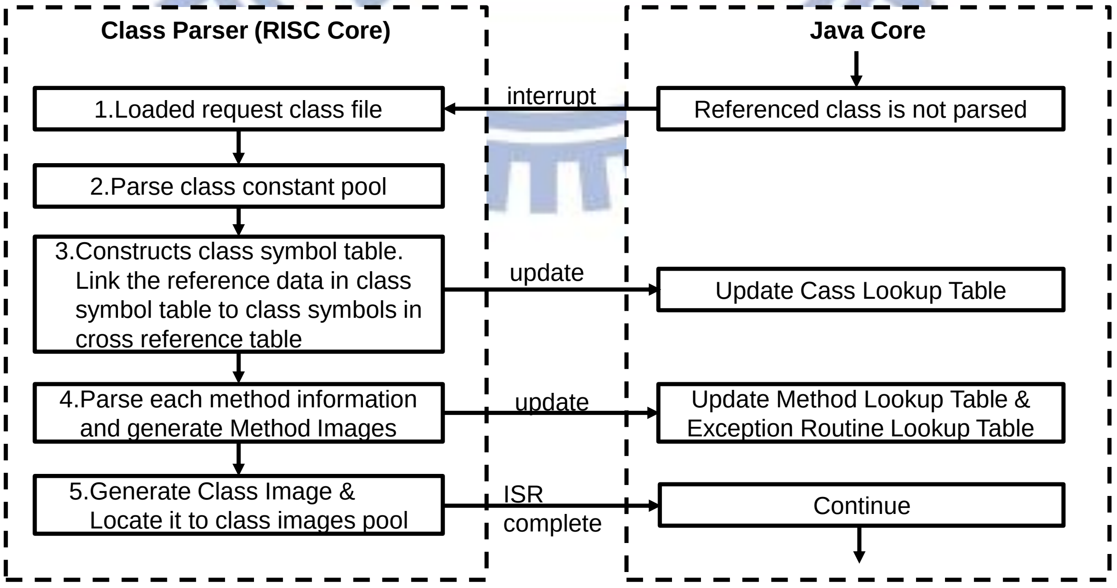

Class Parser (RISC Core)
Java Core
1.Loaded request class file
interrupt
Referenced class is not parsed
2.Parse class constant pool
3.Constructs class symbol table. Link the reference data in class symbol table to class symbols in cross reference table
update
Update Cass Lookup Table
4.Parse each method information and generate Method Images
update
Update Method Lookup Table & Exception Routine Lookup Table
5.Generate Class Image & Locate it to class images pool
ISR complete
Continue

Fig 7. Processes of class parsing.

The parsing process has series processes of class symbols resulting and is mainly for constructing the class image and updating the cross reference table, which is shown in Fig 7. At the beginning of the class parsing, the requested class file is firstly loaded from the CF cards to the DDR SDAM to speed up the file accesses. In the second step, the constant pool data are parsed and resulted to different class symbols. In the third step, a class symbol table is constructed and is linked with the cross reference table. After accomplishing the step 3, the class symbol table contains the reference data which locates a class symbol in the cross reference table. In the same time, the on-chip class lookup table is also updated, which has the location and size information of each class symbol table. In the next step 4, each method in the class is parsed to the method images and two on-chip tables, including the method lookup table and the exception lookup table, are also updated. The method lookup table contains the location and size information of each method image and the exception lookup table contains the information of each exception routines. The class lookup table and method lookup table are two key components in dynamic method loading, which will be discussed in section 3.5.1. Finally in the last step, the class image is generated by merging the class symbol table and each method images and is located at the class image pools.

3.4 Two-Level Runtime Image Design

As mentioned before, the runtime information of a method is loaded into the Java core before the method is executed. The runtime image contains all information for the Java core to execute a method, which includes the class symbol table and the method image. In our previous design, the whole class image is loaded as the runtime image to the Java core. The disadvantage of this approach is obvious since some method bytecodes that are not used would be loaded. In addition to being memory-inefficient, such class-based buffering system also makes it more complex to implement exception handling and multi-threading due to lack of some method-based information. For the exception handling mechanism, the exception routines are a section of Java bytecodes in a method image. To be efficient to find such exception routine in a method when an exception happens, the information locating the method's exception routines is needed.

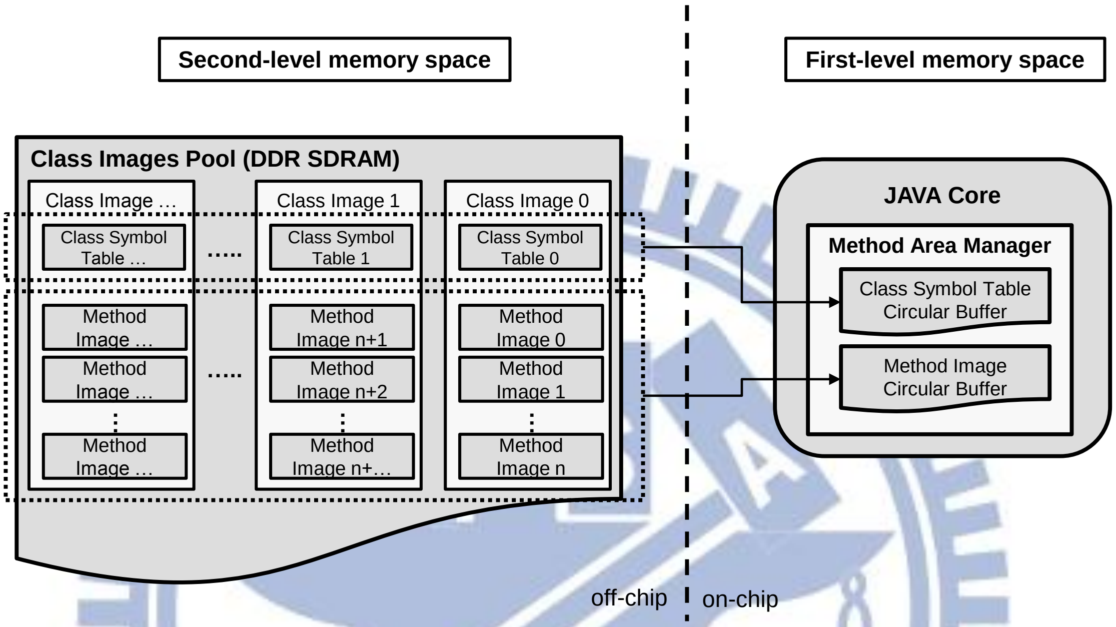


Fig 8. Two-Level runtime image.

In the current design shown in Fig 8, the class images remain at the second-level memory space and each class image is divided into two parts: the class symbol table and the method images. The class symbol table contains reference pointers that are used in the mechanism of dynamic resolution. And the method images contain method's instruction bytecodes and eight-byte headers. To execute a method in a class, the required data are only a class symbol table and a method image. Therefore, only the class symbol table and a method image are required to be loaded into the buffers to execute a method. The essential runtime image is called the method area containing a class symbol table and a method image, which is a subset of class image. To locate and identify each class symbol table and each method image in both the RISC side and the Java core side, a dedicated class ID is related to a class symbol table and a dedicated method ID is related to a method image. In the process of loading the method area in the Java core, the class ID and method ID are used as indexes of lookup tables to retrieve information to locate class symbol table and method image. In addition the IDs also facilitate the design in exception handing to fast access the specific method or class information from lookup tables.

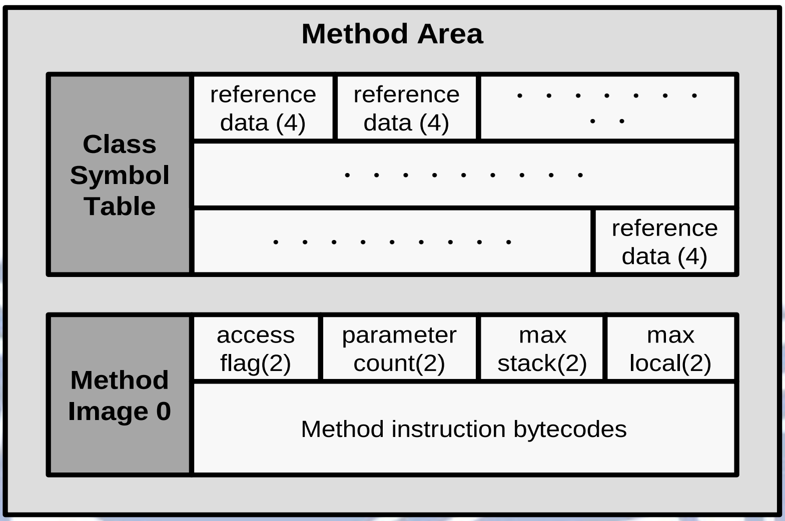


Fig 9. Method area format.

The format of the method area is shown in Fig 9. The method image contains eight bytes headers and the method instruction bytecodes. The header contains the access_flag, parameter_count, max_stack, and max_local. The access_flag indicates the method's declared feature, such as PUBLIC, INTERFACE, ABSTRACT…etc. The parameter_count indicates the numbers of parameters with the invocation of this method. The max_stack is the maximum number of stack elements in the method runtime. And the max_local is the maximum number of local variables in the method runtime. The other two header items, parameter_count and max_local, are two keys to adjust the stack frame during method invocation

In the Java language behavior of method invocation, the top operands may be used as parameters when a method is invoked. And the parameters then become the local variables in the invoked method. In the implementation of the Java core, the process of changing stack frame to the invoked method's stack frame are achieved by adjusting the stack pointer (SP) and variable pointer (VP). In the design of the Java stack, a typical method frame contains local variables, return frames (three stack elements that contain for return information) and operands. The SP points to the base address of the current method stack frame and the VP points to the top element in the current method stack frame.

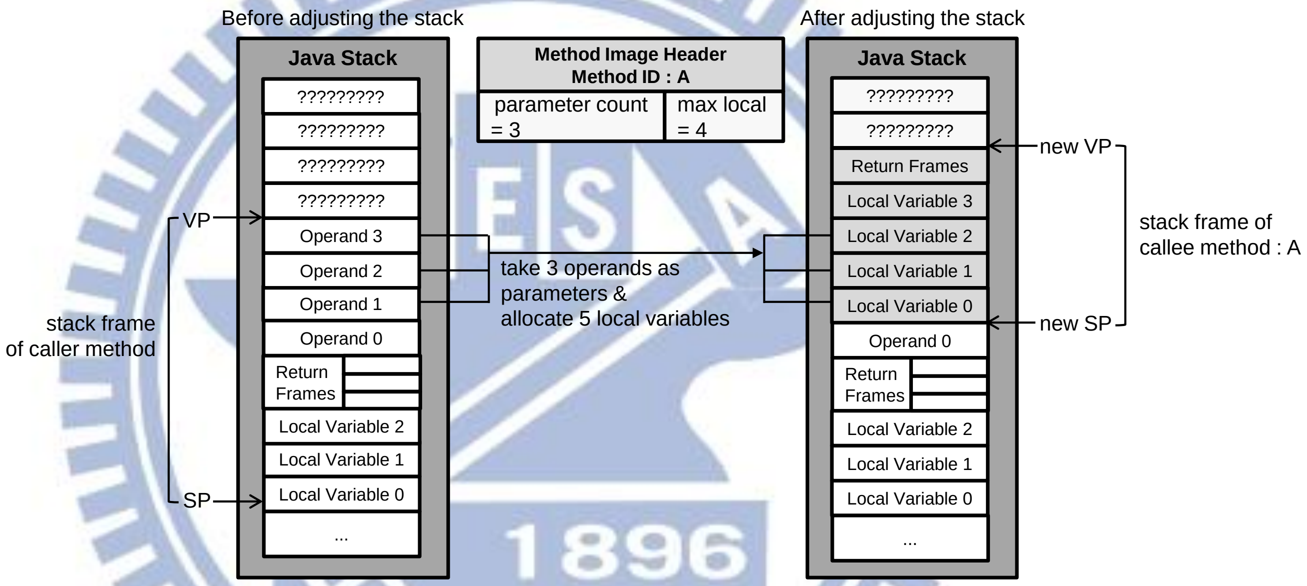


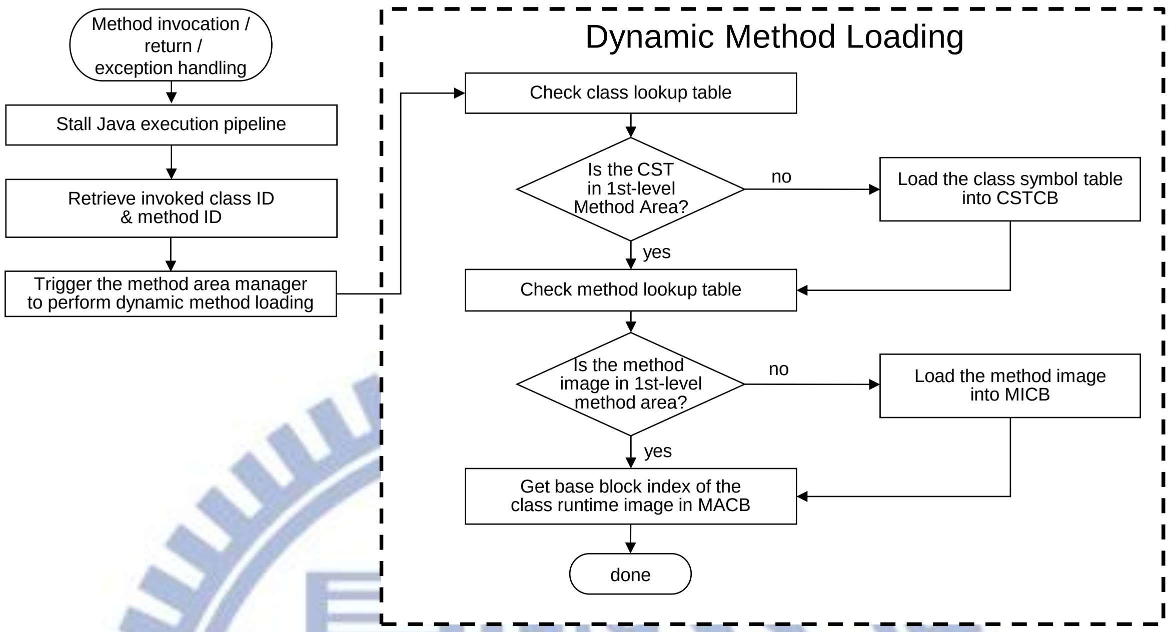
Fig 10.Adjust the stack frame when invoking the method 'A' with the method image header entries: parameter_count and max_local.

An example of adjusting the stack frame in method invocation is shown in Fig 10. When a method "A" is invoked, the method image of this method will firstly be loaded to the buffers in the Java core. Before executing the invoked method, the Java stack is allocated for the invoked method, which is achieved by adjusting the SP and VP. The new SP is adjusted to the value that the old VP minus the parameter count. And the new VP is adjusted to value that the new SP plus max_local and plus three for the return frames. After the process of adjusting the VP and SP, the parameters are placed at the local variables in invoked method stack frame, and the left local variables and return frame are also allocated. In the end, the new stack frame for the invoked method is ready for method execution.

The other part of the method area, class symbol table contains reference data, as shown in Fig 10. For some instruction bytecodes, the operand followed by an operator bytecode is a constant pool index that references to the class symbol. If the referenced class symbol is CONSTANT_Fieldref_info type, CONSTANT_Methodref_info type, or CONSTANT_InterfaceMethodref_info type, the reference data will be a 32-bit table address indexing the class symbols location in the cross reference table. If the class symbol is CONSTANT_Class type, CONSTANT_String_info type, or CONSTANT_Integer_info type, the class symbol data is directly stored at the reference data.

For example, the bytecode “new” is followed by an index indicating which of the class should be instantiated. The index operand indexes to the class symbol table to retrieve the reference data which is the CONSTANT_Class type of class symbol. The reference data is a class ID that is used as an interrupt service routine argument to the software instantiate routine. In the instantiation routine, the object space is then allocated in the heap memory. After allocating the object space, the interrupt service routine signals the Java core with a data which is an object reference (the heap address pointer to the object space). Then the Java core pushes the object reference onto the Java stack accomplishing the bytecode “new”. Such process to result the reference data to the class symbols is performed by the dynamic resolution controller. The dynamic symbol resolution provides a faster way to access class symbols for bytecode execution. The linking process from the class symbol table to the cross reference table is done by the software class parser. In the next section, the process to buffer the method area containing class symbol table and method image is discussed.

3.5 Dynamic Method Loading



Method invocation / return / exception handling
Stall Java execution pipeline
Retrieve invoked class ID & method ID
Trigger the method area manager to perform dynamic method loading
Dynamic Method Loading
Check class lookup table
Is the CST in 1st-level Method Area?
no
Load the class symbol table into CSTCB
yes
Check method lookup table
Is the method image in 1st-level method area?
no
Load the method image into MICB
yes
Get base block index of the class runtime image in MACB
done

Fig 11. The processes of dynamic method loading.

As mentioned in section 3.4, the runtime adopts a two-level memory space hierarchy. To be memory-efficient, only the method area containing class symbol table and method image will be loaded from the second-level class image pool to the first-level on-chip buffers in the Java core. The loading mechanism is only performed in the Java core, which is handled by the method area manager module. The method area manager relies on two lookup tables to determine the class symbol table or method image is whether buffered in circular buffers. If the request class symbol table or method image is not in the buffer, then the unloaded data of method area will be dynamically loaded.

As shown in Fig 11, there are three situations that the Java core will trigger the dynamic method loading process. These three situations are the method invocation from a bytecode, the method return and the exception. When these situations happen, a new method is to be invoked and ready to be executed. Therefore, the method area of the method must be ensured in the buffers. To load the request method's method area, the class ID and the method ID is given to identify the request class symbol table and the method image. With the class ID and the method ID, the method area manager checks the lookup tables to locate the buffered data in the circular buffers. If the class symbol table or the method image is not buffered, a dynamic loading process will be performed by the manager to load the un-buffered data. The two key components of the lookup tables are first introduced in the next section and the detail architecture of the method area manager is presented in section 3.5.2.

3.5.1 Class Lookup Table and Method Lookup Table

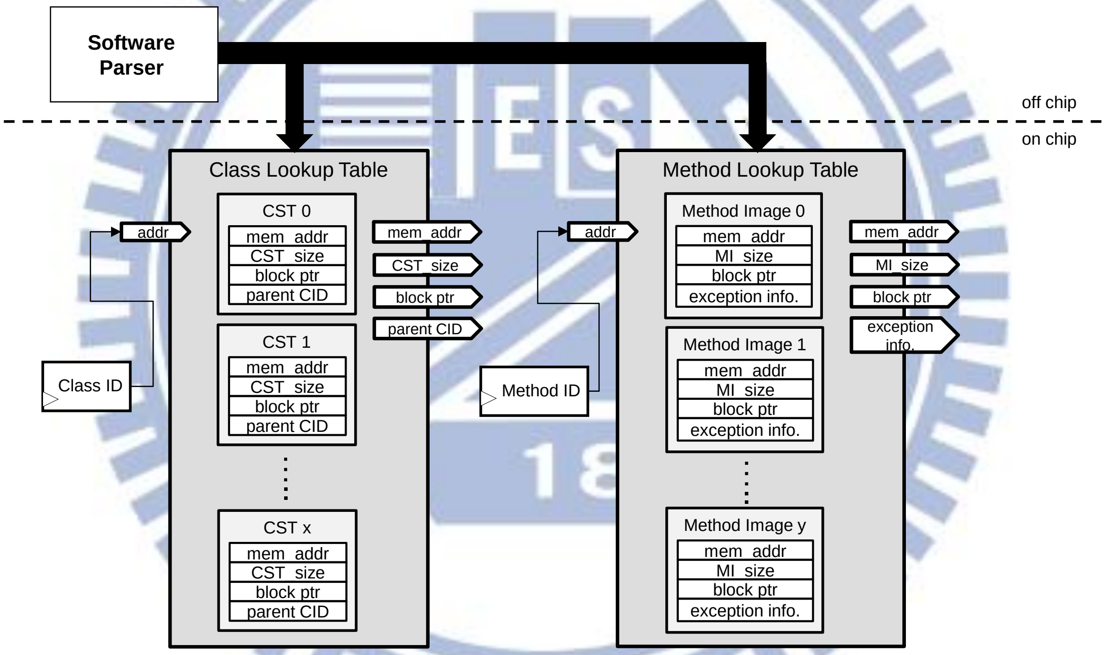


Fig 12. Class lookup table and method lookup table

In the dynamic method loading process, the method area manager relies on two key on-chip tables. The two tables are the class lookup table and the method lookup table, which are shown in Fig 12. The class lookup table contains the class-based information and the method lookup table contains the method-based information. The information is not only used in the process of dynamic method loading but also the process of exception handling, which will be discussed in next chapter.

As shown in Fig 12, to retrieve the specific class's information, the class lookup table (CST) is indexed by the class ID and each parsed class has four items in the table. The four items include base memory address of the class symbol table (mem_addr), size of class symbol table (CST_size), block pointer (block_ptr) and the parent class ID (parent_CID). The first and second items are used in the loading process when the CST is not buffered. The third item, block_ptr, points to the block in the circular buffer where the CST is buffered. And the fourth item parent class ID is used in the process of exception handling.

Another on-chip table, method lookup table, is indexed by the method ID and has four items for each method in the parsed classes. The four items include the base memory address of the method image (mem_addr), size of the method image (MI_size), the block pointer (block_ptr) and the exception information (exception_info). The first and second items are used in the loading process when the method image is not buffered. The third item, block_ptr, points to the block in the circular buffers where the method image is buffered. And the fourth item exception_info is used in the process of exception handling. In the next section, it shows the detail that how the method area manager cooperates with two lookup tables in the process of dynamic method loading.

3.5.2 Architecture of the Method Area Manager

In the framework of two-level runtime image design, the first-level memory space is composed of two circular buffers buffering the method area. To buffer method area, the class symbol table and the method image are separately buffered into two circular buffers which have different block sizes. The class symbol table circular buffer (CSTCB) has 32 blocks and each bock is of 256 bytes. Another circular buffer, the method image circular buffer (MICB), has 64 blocks and each bock is 256 bytes. The circular buffers are designed to buffer the data in a FIFO manner. We choose to implement the FIFO policy rather than other sophisticated replacement techniques (e.g., LRU) due to its simplicity. Furthermore, some studies show that the method-based FIFO caching policy has reasonable performance for Java applications [25][26]. Note that a CST and MI may occupy more than one block in circular buffers. Therefore, a register, CSTCB_Pointer, is used to point the start block occupied by the specific CST in the CSTCB. And a register, MICB_Pointer, is used to point the start block occupied by the specific MI in the MICB. With the CST_Pointer, the specific block in CSTCB can be selected and accessed by the dynamic resolution controller. With the MI_Pointer, the specific block in MICB can be selected and accessed by the bytecode execution engine.

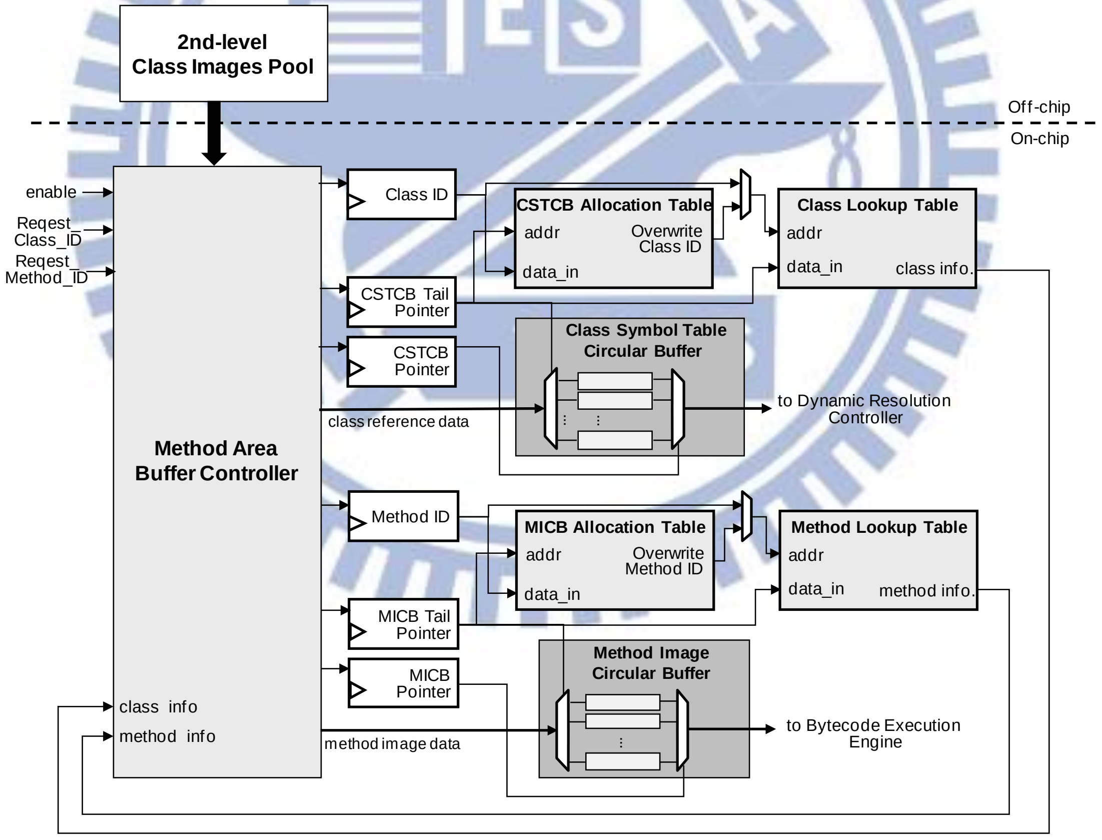


Fig 13. Architecture of the method area manager.

As mentioned before, there are three situations (method invocations, the method returns and the exceptions) that the Java core may trigger the dynamic method loading.

In these situations, an enable signal raised by other components in Java core will trigger the method area manager to perform dynamic method loading. When the enable signal is raised, two input signals, Request_Method_ID and Request_Class_ID, are valid in the inputs, which are shown in Fig 13. The register Class_ID indicates which of the class symbol table should be located and the register Method_ID indicates which of the method image should be located. After the required method area is ensured in the circular buffers, the manager raises a loading_done signal to notify the Java processor the process of dynamic method loading is finished.

To perform the process of dynamic method loading, the method image buffer manager relies on one controller and two similar sets of tables and registers, which is shown in Fig 13. One set is for cooperating with the CSTCB and another one is for cooperating with the MICB. These two sets of tables and registers have similar functional components and each set have three registers and two tables. In the set cooperating with CSTCB, three registers are Class_ID, CSTCB_Tail_Pointer and CSTCB_Pointer. The register Class_ID indicates which of the CST is loaded or ready to be loaded. The CSTCB_Tail_Pointer points to the last block that is ready to be allocated. The third register CSTCB_Pointer points to the first block that locates the request CST. In addition, the set contains two tables CSTCB allocation table and the class lookup table. The CSTCB allocation table records the class ID of CST that is allocated in blocks. The input of CSTCB allocation table is block pointer and the output is a class ID indicating that which of the CST occupies the block. Another table, the class lookup table, has the information of parsed class information and the start block pointer of CSTCB, which is discussed in section 3.5.1.

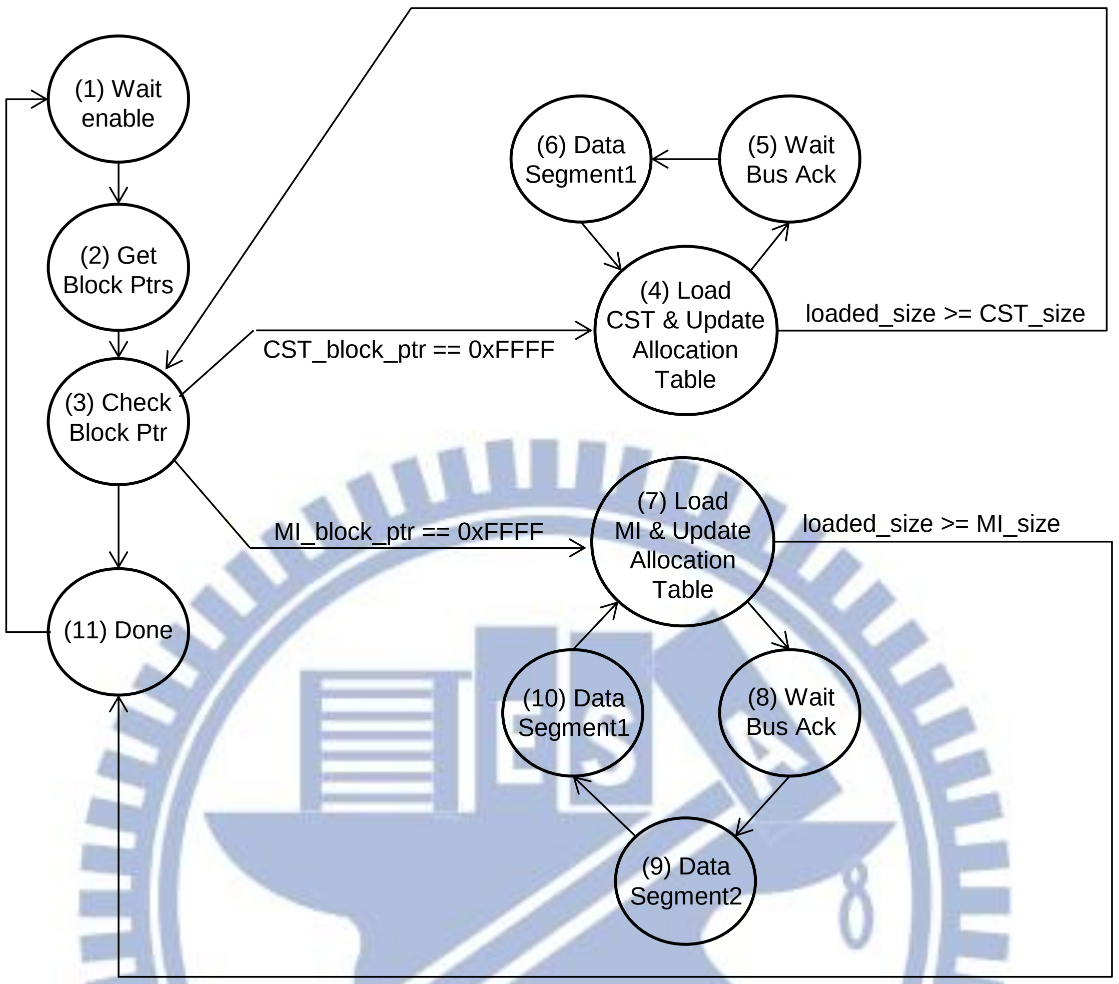


Fig 14. FSM in the method area controller.

The finite state machine in the method area controller controls the process of dynamic method loading, which is shown in Fig 14. In the state (1), the controller waits for the enable signal to activate the process of dynamic method loading, which is requested from the bytecode execution engine or the exception handling unit. When the enable signal is raised, another two signals, Request_Class_ID and Request_Method_ID, are valid and separately stored to the registers Class_ID and Method_ID. After the enable signal is raised, the control state moves to the next state, Get Block Ptrs. In this state, the registers Method_ID and Class_ID are updated to requested IDs and separately index the class lookup table and method lookup table. After one cycle delay, the information of block pointers of CSTCB and MICB are valid at the outputs of the two lookup tables in state (3) Check Block Ptrs. If the value of block pointer is equal to 0xFFFF, it represents that the requested data are not buffered and the controller will start to load the requested data from the second-level memory. If the value of the block pointer is not equal to 0xFFFF, it represents the requested data is buffered and the value is the base pointer of the requested data in the circular buffers. According to the outputs of block pointers from the two lookup tables, the state (3) moves to one of three states (4) Load CST, (8) Load MI and, (13) Done. If the requested method image and class symbol table are already buffered, the state machine moves to the state (13) Done. If the class symbol table are not buffered, the state machine moves to the state (4) Load CST. If the class symbol table are buffered and the method image is not buffered, the state machine moves to the state (8) Load method image.

Before moving to the state (4) from state (3), the item block pointer (0xFFFF) in class lookup table is updated from the CSTCB_Tail_Pointer which points to the blocks that locates the newly loaded CST. In the same time, the register CSTCB_Pointer is also updated from the register CSTCB_Tail_Pointer so that it can select the allocated block which is ready to be loaded in following processes. In the next state (4), (5) and (6), they are mainly to load each one word data of the requested class symbol table. The state (5) raises the external memory loading signal and waits for the 32-bits external loaded data from the bus. After the ACK signal is raised, the external loaded data is valid and is stored to the pointed circular block in state (6). The states (4) (5) and (6) continue the loading processes until the whole requested CST data is buffered.

In the states (4) (5) and (6), a requested loaded CST data may occupy more than one block in the CSTCB. In this case, the register CSTCB_Tail_Pointer will plus one to point to the allocated block. And if the newly loaded data overwrites the block that contains another CST data in the circular buffer, the item block pointer of the overwritten CST in the class lookup table must be updated to 0xFFFF. To retrieve the overwritten Class_ID, it relies on the CSTCB allocation table which records the occupied class ID of each block in the circular buffer. The CSTCB allocation table is indexed by the CSTCB_Tail_Pointer so that the class ID indicating the overwrite CST data can be retrieved from the output signal overwrite_Class_ID. The overwrite_Class_ID re-index the class lookup table and update the block pointer to 0xFFFF indicating that this CST is not buffered.

After loading the requested CST to the CSTCB, the controller state is changed back to the state (3) Check Block Ptr. If the requested method image is not buffered, a series of loading process similar to loading class symbol table will be performed again. In the end, the request CST and MI are ensure to be buffered in the CSTCB and MICB. The controller state is changed to state (11) Done and the controller raises the done signal to notify the Java processor to resume execution.

Chapter 4. The Implementation of the Exception Handling Unit

4.1 Overview

In the Java programming language, exception handling is a crucial component for general runtime error handlings. Without exception handling, unhandled errors may cause incorrect results or unpredictable program behaviors. Therefore, a comprehensive exception handling mechanism is supported in the Java language. The Java reacts to errors by throwing exception events which may be caught by an exception handling routine. The exception handling routines are written by the programmer to deals with the error condition, which is an isolated code segment of a Java program. The details of exception mechanism in the Java language will be described in the section 4.2.

Although there are many papers on Java processor designs, most researches do not cover the design of the exception handling circuitry [1][2][3]. In [4], a hardware-software co-design approach of a Java virtual machine is proposed with the exception handling capability. The paper admits it is not easy to design special circuitry for handling Java exceptions due to the complex exception handling process. The exception handling process involves the call stack unwinding, exception routines searching and instantiating/passing the exception object. Thus, it implements full exception handling on the general-purpose processor core. However, such design makes exception handling very inefficient. This might be fine for non-recoverable exceptions. However, for exceptions that will not stop the execution of the program, the design would be unacceptably inefficient.

In this chapter, a hardware-based design of exception handling unit for a dual-core Java processor is proposed. The proposed design is based on previous work published in [20][21][22][23] and it is facilitated by the new method area manager design described in chapter 3. In the proposed design, the exception handling process relies on two key components, the exception lookup table and the exception handling unit. The exception routine lookup table is an on-chip block-RAM that stores the information of each exception routines contained in the parsed class. The other component is the exception handling unit which performs a sequence of stack frames unwinding and routines searching to find a suitable exception routine from the exception lookup table. The whole exception handling mechanism is mainly performed in the Java core to reduce the handling overheads.

If the exception event is triggered by bytecode "athrow", there is no inter-processor communication overhead in such handling process. In this case, the performance of exception handling has a best performance on the exception handling process. Moreover, the proposed exception handling architecture only costs 320 LUTs and its performance can be up to 525 times better than the performance of the Sun CVM running on a processor at the same clock rate. The experimental results will show that the exception handling architecture is suitable for hardware solutions for the dual-core Java processor, which will be discussed in chapter 5.

4.2 Exception Handling in the Java Language

In the Java language, an exception event is activated by throwing an object instantiated from a subclass of the Throwable class. Programmers can define their own exceptions and exception handlers by creating subclasses of the Throwable class. According to two different types of operations that cause an exception, the exception can be thrown by a bytecode instruction "athrow" or thrown by the Java runtime system due to runtime errors. After an exception throws, a specific exceptions object which carries the information of exception type will be created and is handed over to the exception handling mechanism in the Java runtime system. The exception handling mechanism then triggers the exception handling process to find a suitable exception handler. Note that the suitable exception handler may be located in one of the callers' method.

In a Java program with exception routines, a try statement specifies a try block which is a program region associated with one or more exception handlers. For each one exception handler, the class name in the catch statement specifies what exception types it can catch/handle. If the thrown exception object is the instance of sub class or the class in the catch statement, it represents that this exception can be handled by the exception routine. If the exception cannot be caught by any exception routine that is associated with the try block, an outer try block of the current try block will be examined. The outer try block may exist within the same method or in a caller method. Such the searching processes continue until an exception routine that can catch the thrown exception is found.

```
1: public class Example extends Throwable {
2:     public static void main(String[] args) {
3:         try{
4:             method_throw();
5:         }catch(Throwable  e){
6:             //routine codes
7:         }
8:         //remaining programs
9:     }
10:   static void method_throw() throws Example {
11:       try{
12:           Example object = new Example();
13:           throw object;
14:       }catch(ArithmeticException e){
15:           //exception routine codes
16:       }catch(NullPointerException e){
17:           // exception routine codes
18:       }
19:   }
20:
21:}
```

Fig 15. An example of a Java program that throws an exception.

An example of a Java program defined with two exception routines is shown in Fig 15. In the program, the method method_throw() throws an exception object at line 13. And the exception object is not caught by the exception routines in the method_throw() because the thrown Example object is not the instance of the class or sub-class of ArithmeticException or NullPointerException. Next, the exception object is caught by the exception routine defined in line 5, which is in the caller method main(). It is because the thrown Example object is the sub class of Throwable. In the end, the Java program jumps to the exception routine in line 6 and continues executing the remaining Java program.

4.3 Exception Handler Design for Dual-Core Java Processor

According to the behavior of the Java language, a suitable exception routine must be located and executed after an exception event occurs. Note that the exception routine may be resident in one of the callers' method, which means that if the current method has no suitable routine, the routines in caller methods should also be checked.

To execute a regular program efficiently, one should not incur extra exception handling overheads until an exception actually happens. Therefore we choose the stack unwinding approach to search for the suitable exception routine. In addition to the runtime overheads caused by exception, the performance of dealing with the exception is also crucial. The process of exception handling involves a series of steps of type matching, program counter range checking, stack unwinding.

With the stack unwinding approach, the handling process is only performed after an exception event occurs. This handling process is performed through a sequence of exception table searches and stack unwinding. In the proposed architecture, the handling processes are performed by the exception handling unit that residents in the execute stage module in the bytecode execution engine. The exception handling unit retrieves exception routine information from an on-chip exception lookup table and determines a method whether contains a suitable exception routine. If a method does not contain a suitable exception routine, the exception handling unit will unwind the Java stack to search the caller method for a suitable exception routine. Such process of unwinding stack and routine searching are only performed in the Java core so that it has no inter-processors communication overheads.

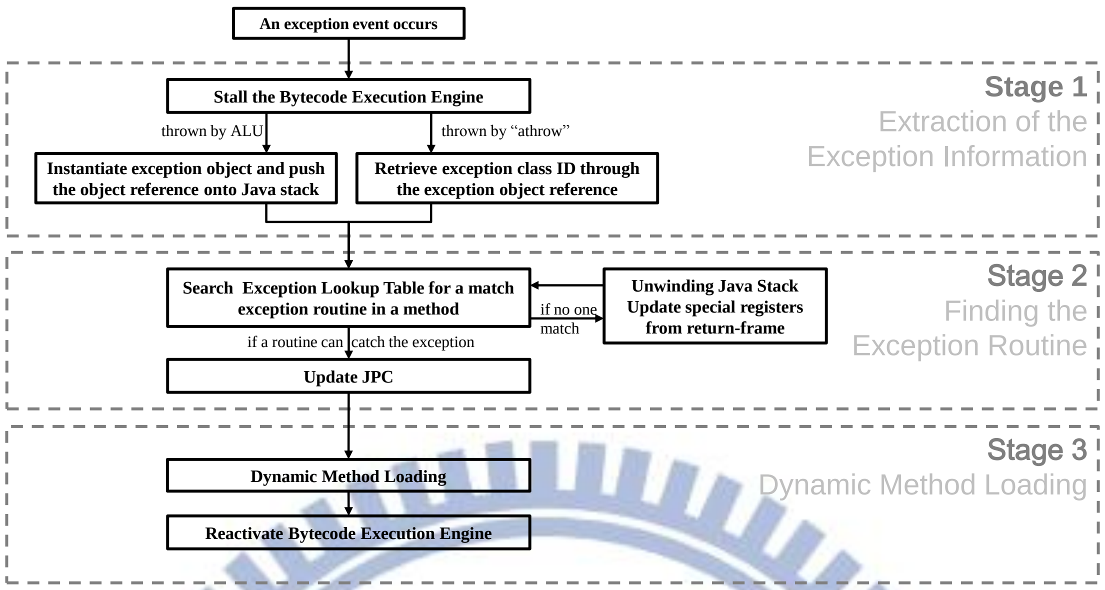

An exception event occurs
Stall the Bytecode Execution Engine
thrown by ALU
thrown by "athrow"
Instantiate exception object and push the object reference onto Java stack
Retrieve exception class ID through the exception object reference
Stage 1
Extraction of the Exception Information
Search Exception Lookup Table for a match exception routine in a method
if no one match
Unwinding Java Stack
Update special registers from return-frame
if a routine can catch the exception
Update JPC
Stage 2
Finding the Exception Routine
Dynamic Method Loading
Reactivate Bytecode Execution Engine
Stage 3
Dynamic Method Loading

Fig 16. Three stages of exception handling process.

When an exception event happens during runtime, the Java core should stall the bytecode execution engine immediately. In the same time, the operation is handed over to the exception handling unit. Then the exception handling unit starts to perform the exception handling process, which is shown in Fig 16. The handling process can be divided into three stages: extracting the exception information, finding the exception routine, and dynamic method loading. The details of each stage are described in the following sections.

4.3.1 Stage 1 : Extraction of Exception Information

In the Java language, it acts the exception event by throwing an exception object. According to the thrower of the exception, there are two types of exception events, the exception thrown by the JVM and the exception explicitly thrown by the instruction bytecode "athrow". In the first type, the exception is thrown due the error in the Java runtime system. For example, an arithmetic error happens due to that an operand is divided by zero. In the Java core, the computing unit is performed by the ALU located in the bytecode execution engine. To detect the arithmetic exception, the operand is examined when a division operation is executed.

The second type of exception is thrown by the bytecode “athrow”, which triggers an exception event directly through the bytecode. To trigger such exception event, the thrown exception object is instantiated by the bytecode before the “athrow”. Note that for the first type of exception, the object instantiation operation is performed by the JVM. Before the process to search for the suitable exception routine, the thrown exception object is ensured to be instantiated and be pushed onto the Java stack. And the thrown exception type identified by a class ID is also needed to be prepared for routine searching process.

For the JVM-type exception, the exception classes defined in CLDC are listed in appendix 1. An example that triggers an arithmetic exception is discussed as follows. As mentioned before, the arithmetic exception is triggered and detected by ALU in the bytecode execution engine. For such the JVM-type exceptions, the exception class is predefined in the system class library. In the proposed design, the predefined classes are pre-parsed in Java system booting by default. Therefore when the JVM detects the arithmetic exception event, the class ID of ArithmeticException is known and saved to a register named Thrown_EID. The register Thrown_EID is a key comparing feature to match an exception routine in later stages. Another task in the stage one is to instantiate a specific exception object and to push it to onto the Java stack. In this task, the exception handling unit will trigger an interrupt service to instantiate the exception object. When the triggering the instantiation services, the Thrown_EID is used as the service argument to instantiate a object of specific exception class. After accomplishing the service, the exception object reference is sand back to Java core and pushed onto the Java stack so that the handling process is ready for entering to the stage two.

The other exception type is thrown by bytecode “athrow”. For this type, the steps of instantiating the throwing exception object and pushing the object reference are already done by the instruction bytecodes before executing the bytecode “athrow”. In this case, the thrown exception class ID is turned out to be unknown, which is an important search key in the next exception handling stage. To retrieve the exception class ID, the exception handling unit requests a heap memory access by the object reference which is on the top of stack. In the previous work, the object space is allocated in the heap space and the first item in an object space is the class ID that instantiates this object space. To accomplish the exception handling process in stage one, the thrown exception class ID is retrieved from the allocated object space and is saved to the register Thrown_EID.

In the end of the stage one, the thrown object reference is ensured on the top of stack and the Thrown_EID is ready for the process of searching a routine to catch this thrown exception. In the stage two, the exception handling unit performs a sequence of stack frame unwinding and the table searching to find a suitable exception routine. One of the key components is the design of exception lookup table, which is an on-chip block ram that stores the exception routines information. The design of exception lookup table is firstly described in the next section 4.3.2 and the details of stage two will be described in the section 4.3.3.

4.3.2 Exception Lookup Table

To find a matching exception routine to catch the thrown exception, the exception handling unit relies on an on-chip block ram named exception lookup table. The exception lookup table stores the information that lists all exception routines in parsed classes, which is shown in Fig 17. Each exception routine in the exception lookup table has four items including exception routine start offset (ER_start), exception routine end offset (ER_end), exception routine address (ER_addr), and exception routine class ID

(ER_EID). The two items ER_start and ER_end indicate the range in the Java bytecode offset at which the exception catch area is active. The item of the ER_addr indicates the start offset of the exception routine in the method image. The last item ER_EID represents the class type of this exception routine. The three items ER_start, ER_end and ER_EID are the key features for an exception routine to determine whether it can catch the thrown exception. And the ER_addr is used to set up the new program counter locating the exception routine that can catch the thrown exception object.

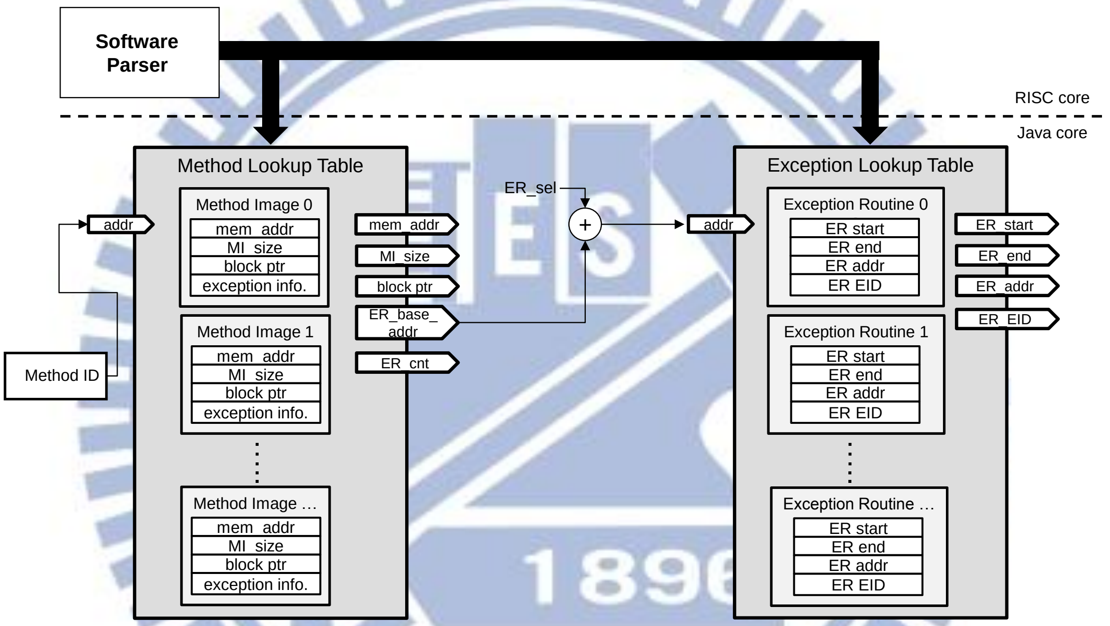


Fig 17. Architecture of exception lookup table.

Due to the advantage of dynamic class parsing, the exception routines information of the class is added to the exception lookup table only when the class is really referenced and parsed. In the process of parsing a class, the exception information in a class file is arranged to a table that residents behind of each method information. The table format in the class file is similar to the exception lookup table. The only difference is in the fourth data item, namely exception type. The exception type in the table of the class file is a constant pool number indexing to the class information. After parsing the exception table for a method, each exception type is resulted to a dedicated class ID. Once a routine is parsed, the four items of a routine are added to the exception lookup table. After parsing a class, each method's exception routines information is compacted and added to the exception lookup table.

In the process of finding a method containing the suitable exception routine, a sequence of routine searches in each calling method is performed. To reduce the overheads in such process, the ability to efficiently retrieve the exception routine list for a method is required. In chapter 3, the method lookup table is designed to have the capability of recording such information for each parsed method. In the proposed design, a method item in the method lookup table is used for locating the first exception routine of a method. Note that a method may have several exception routines. Therefore, the item is divided into two entry including the exception routine base index (ER_base_index) and exception routine count (ER_cnt). The ER_base_index is the table address pointing to method's first exception routine in the exception lookup table and the ER_cnt is the numbers of exception routines contained in that method. When parsing each method of a class, the two items ER_base_index and ER_cnt related to each method are also updated to the method lookup table, which is also mentioned in section 3.3.

4.3.3 Stage 2 : Finding The Exception Routine

In the Java language, the exception event can be handled locally or globally, which means the thrown exception can be caught by the routine located in one of the caller methods. To find the proper routine in such caller method chains, it needs to trace through the chains of each called method. The information related to the caller method is saved as return frame in the Java stack. To try not to incur extra exception handling overheads in normal Java program until an exception happen, we adopt the stack unwinding approach to trace the caller methods.

To check that a method whether contains a suitable exception routine, the exception handling unit relies on the exception lookup table and some control logics, which is shown in Fig 18. In the proposed design, the exception lookup table is indirectly indexed by method ID so that the information of exception routines belonging to a specific method can be retrieved efficiently. The method ID firstly indexes to the method lookup table to retrieve a method's ER_base_index. Then the ER_base_index indexes to the first exception routine of the method in the exception lookup table. To select one of the exception routine information of a method, the ER_base_index pluses the ER_sel as the input address of the exception lookup table. The indexed exception routine information will be used in the comparing processes to check whether it can catch the thrown exception.

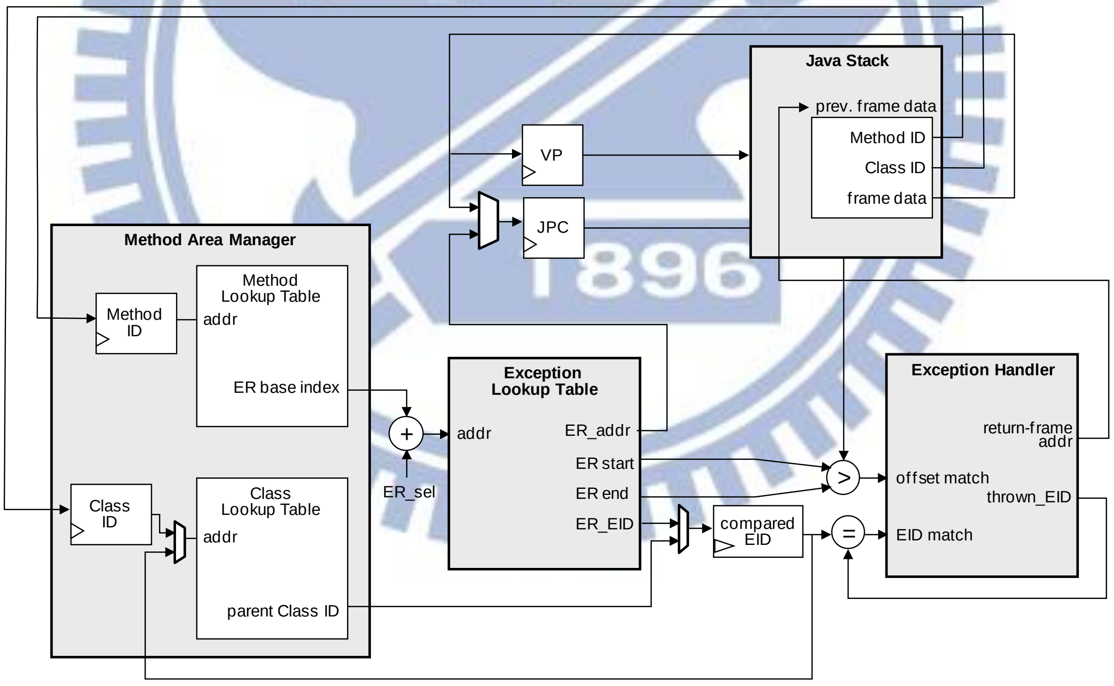


Fig 18. Architecture of the exception handling unit that cooperates with method area manager.

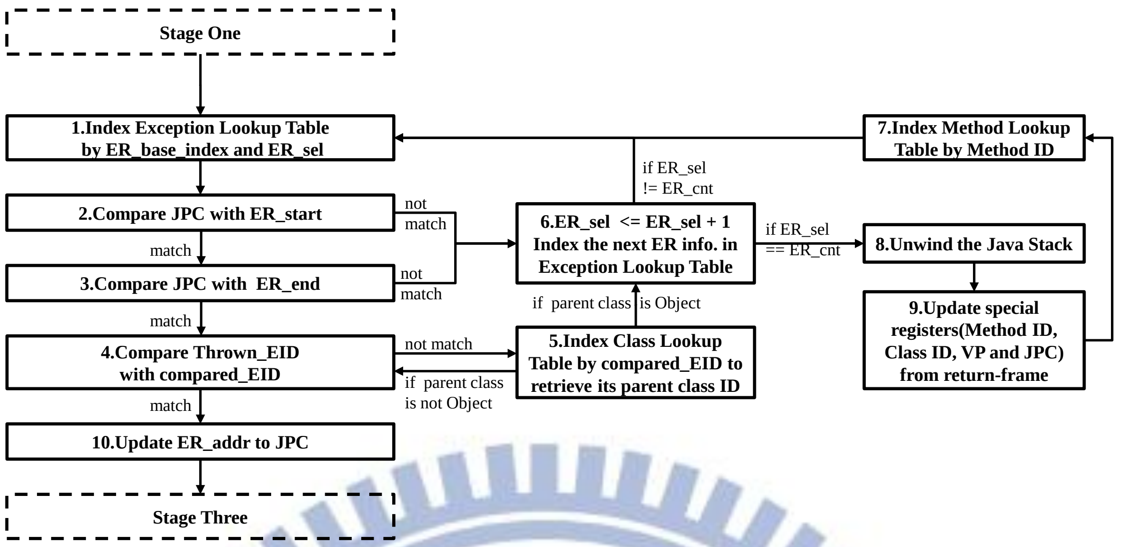

Stage One
1.Index Exception Lookup Table by ER_base_index and ER_sel
2.Compare JPC with ER_start
not match
match
3.Compare JPC with ER_end
not match
match
4.Compare Thrown_EID with compared_EID
not match
if parent class is not Object
match
10.Update ER_addr to JPC
Stage Three
5.Index Class Lookup Table by compared_EID to retrieve its parent class ID
if parent class is Object
6.ER_sel <= ER_sel + 1 Index the next ER info. in Exception Lookup Table
if ER_sel != ER_cnt
if ER_sel == ER_cnt
7.Index Method Lookup Table by Method ID
8.Unwind the Java Stack
9.Update special registers(Method ID, Class ID, VP and JPC) from return-frame

Fig 19. Exception handling steps in the stage two.

To check whether the selected exception routine can catch the thrown exception or not, two runtime registers are compared with the related items in the exception lookup table. These two runtime registers are the thrown_EID and the Java program counter (JPC). The JPC implies the location of the instruction that throws an exception or the location of the instruction that invokes one of the callee methods throwing the exception. If the thrown_EID is the same as the compared_EID and the JPC falls within the range between ER_start and ER_end, it means the exception routine can catch the thrown exception. To reduce the hardware cost of comparing logics, the comparing process for the exception handling unit is conducted sequentially, which is shown in Fig 19.

Note that if the thrown exception object is the instance of the sub class of an exception routine, then it can also be caught by the routine. Therefore the relation of the class inheritances is needed in such comparing process. To efficiently retrieve such information, the item named parent class ID (parant_CID) is added to the class lookup table. To retrieve the grand parent’s class ID, the parant_CID can used to re-index the class lookup table. In such processes of re-indexing from each class’s parant_CID, the relationship of the class inheritance can be easily retrieved from the class lookup table. As shown in Fig 19, if an exception routine is found that the EID is not matched in step 4, the parent_CID will be retrieved and saved in the register compared_EID in step 5. In step 5, if the retrieved parent_CID is not the class ID of the Object class, then the handling process will go back to the step 4 to comparing the EID again. Examining the parent class ID in step 4 and 5 continues until the Throwable class is examined. The Throwable class is defined as the top exception class in the inheritance relations and its parent class is the Object class. Thus in the step 5, when the retrieved parent Class is the Object class, the exception handling process will go to step 6 to select the next exception routine information in the exception lookup table. In step 6, it determines whether the exception routines in a method are all checked. If the method has routines that have not been checked, the next exception routine information in the method will be selected and the step 1, 2 and 3 will be performed again to check this exception routine is match for a suitable exception routine.

Once no routine in a method can catch the exception in step 6, the thrown exception should be handed over to the caller method. To perform the steps to check whether the caller method contains a suitable routines or not, three registers including class ID, method ID, and JPC must be updated. The class ID is used to retrieve the parent class ID in step 5. The method ID is used to indirectly re-index the exception lookup table to retrieve the exception routines information. And the JPC is the key comparator to match the thrown exception in step 2 and 3. In the proposed design, the registers are saved as the return frame data when a method is invoked. To retrieve the frame data, a return-like process is performed, which is called stack unwinding. After the special registers are updated to the data of the caller method, the processes to check whether the caller method contains a suitable routine can be performed again.

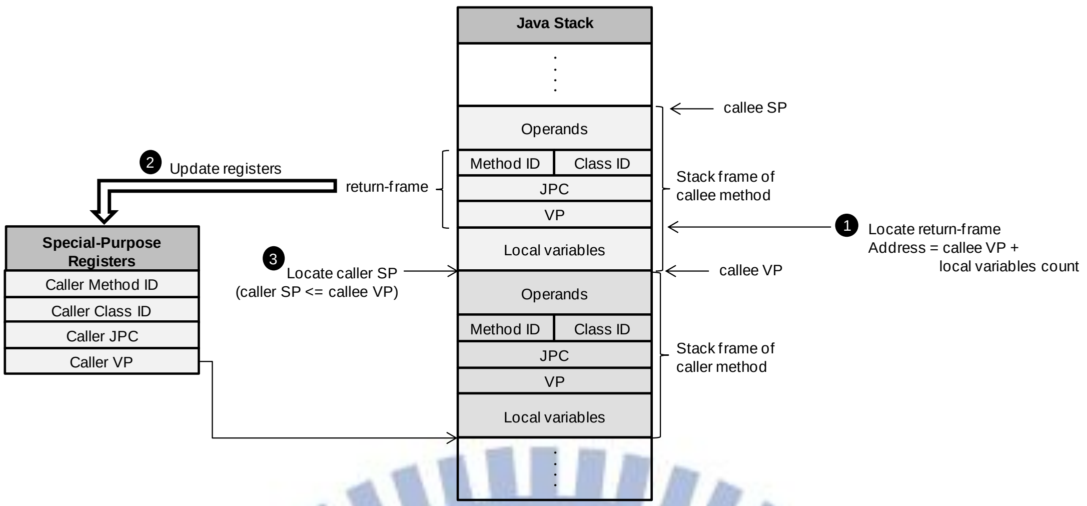


Fig 20. Processes of Java stack unwinding.

To unwind the Java stack, the first step is to locate the return-frame. As shown in Fig 20, the Java stack contains each stack frame of each invoked method at runtime. In a typical stack frame of a method, it contains local variables, return-frame and operands. The local variables are variables declared in the method and the operands are the intermediate computation data in runtime execution. The return-frame saves the runtime status of registers of the caller method, which is used to restore special-purpose registers when returning to the caller method. The return-frame items include the VP (the Java stack pointer that points to the start address of the caller's stack frame), JPC (Java program counter), the method_ID, and the class_ID. To locate the return-frame, the local variable count is firstly loaded from the method image in the external class images pool. As shown in Fig 20, the stack address of return-frame is the VP plus the local variable count. After locating the return-frame, the special registers are updated from the data in the return-frame. And the exception handling unit is ready to search the method's exception routines.

In the next, the exception lookup table is indirectly re-indexed by the register method_ID so that the routine information of the method can be located and the JPC is updated to the invocation instruction location in caller method that passes the thrown exception. The procedure of finding a suitable exception routine for a method is then performed again. The sequence of table searching and stack unwinding continues until a matching exception routine is found. Once an exception routine is match to catch the exception, the ER_addr of the matched routine will update the register JPC to indicate the location of the exception routine in the method image. Finally in the end of stage 2, the statuses of updated special registers and stack frame are ready for Java core to execute the exception routine. To execute the exception routine by Java core, the last stage is to trigger the dynamic method loading to ensure the method area of the method containing the match exception routine is in the circular buffers.

4.3.4 Stage 3 : Dynamic Method Loading

As the final stage of exception handling, the method area manager which is discussed in chapter 3 is triggered to locate the class symbol table and method image of the method containing the match exception routine. The enable signal of the dynamic method loading is raised and the registers method_ID and class_ID are also signaled to method area manager as inputs of request IDs. After the process of dynamic method loading is finished, the bytecode execution engine is ready to retrieve the instruction bytecodes form method image circular buffer and the dynamic resolution controller is ready to result runtime class symbols from class symbol table circular buffer. In the stage two, the Java stack is also adjusted and ready for accessed by bytecode execution engine. In the end, the whole exception handling routine is finished and Java core is re-activated and to execute the remaining Java program.

Chapter 5. Experimental Results

5.1 Development Platform and Tools

We have implemented the proposed dual-core Java platform on the FPGA Xilinx Virtex5 ML507 development board. We use Xilinx Embedded Development Kit 13.1(EDK) as the development tool and Xilinx® Synthesis Technology (XST) as the FPGA synthesis tool. We use the ISim simulator for verification, which is also provided in the EDK. We create the implementation platform from Base System Builder (BSP) wizard of Xilinx Platform Studio. The platform contains a Microblaze soft-IP core and the Java core IP. And the RTL model of the Java core is written in VHDL.

The resource utilization of the FPGA device is shown in Table I. Note that the numbers of Virtex-5 LUTs and BRAMs in Table I only include the logic for the proposed Java core. In this table, we can see that the proposed exception handling unit only costs 267 LUTs which occupy 4% of LUTs in the whole Java IP. And an additional BRAM is added as the exception lookup table for the exception handling unit, which is mentioned in section 4.3.2.

Device : vertex-5 5vfx70tf1136-1		
	Without Exception Handler Unit	**With Exception Handler Unit**
Number of LUTs	6,270	6,537
Number of 2K BRAM	36	37
Maxinum frequency	83.3Mhz	

Table I. Synthesis information of Java core.

5.2 Benchmark Analysis

To evaluate the performance of the proposed exception handling unit, benchmark programs are also tested in the Sun CVM. The Sun CVM is running under MontaVista Linux which is ported on the Xilinx ML-405 development board using the PowerPC core at 83.3 MHz. In following sections, we present three scenarios of benchmarks to observe the time overheads of exception handling process. The scenario one benchmark focuses on the time overheads of searching a suitable exception routine through multiple exception routines that are containing in a same method. The benchmark of scenario two focuses on the time overheads of searching a suitable exception routine in multi methods searching, which involves the processes of unwinding Java stack. And the programs in scenario three have a little difference to the programs in the scenario two. In the scenario three, each of the middle searched methods contains an exception routine, which will have additional exception routine matching process.

5.2.1 Scenario One Benchmark Analysis

The first benchmark focuses on observing the time overheads of exception handling process that searches the exception routines containing in a same method. Each program in the scenario one has different numbers of catch statements. A test program in scenario one benchmark is shown in Fig 21. The program contains five catch statements resident to a try block that throws an ArithmeticException object by bytecode “athrow”. The Exception_x classes in the first to fifth catch statements are child classes of Throwable class. In other words, the thrown exception object cannot be handled by the first four exception routines. However the thrown object can be caught in the fifth exception routine in line 13 because the routine is defined to catch the

ArithmeticException object. The Java program then jumps to the exception routine five in line 14 to execute the exception routine. To observe the time overheads due to numbers of searched exception routine, each test program in scenario one benchmark has different numbers of searched exception routines.

```
1:  public class Scenario_one_thrown_by _athrow{
2:     public static void main(String args[]){
3:           try{
4:                throw new ArithmeticException();
5:           }catch (Exception_1 e ){
6:                //exception routine 1
7:           }catch (Exception_2 e ){
8:                //exception routine 2
9:           }catch (Exception_3 e ){
10:               //exception routine 3
11:          }catch (Exception_4 e ){
12:               //exception routine 4
13:          }catch (ArithmeticException e ){
14:               //exception routine 5
15:          }
16:    }
17: }
```

Fig 21. A test program with exception thrown by bytecode “athrow”.

```
1:  public class Scenario_one_thrown_by _JVM{
2:     public static void main(String args[]){
3:           try{
4:                int a = 999999/0;
5:           }catch (Exception_1 e ){
6:                //exception routine 1
7:           }catch (Exception_2 e ){
8:                //exception routine 2
9:           }catch (Exception_3 e ){
10:               //exception routine 3
11:          }catch (Exception_4 e ){
12:               //exception routine 4
13:          }catch (ArithmeticException e ){
14:               //exception routine 5
15:          }
16:    }
17: }
```

Fig 22. A test program with exception thrown by JVM.

In addition to the exception events thrown by bytecode “athrow”, the exceptions thrown by JVM are also tested in the scenario one benchmark. Another test program with the exception thrown by JVM is shown in Fig 22. The test program triggers the ArithmeticException type exception implicitly in line 4 due to the wrong arithmetic operation in Java runtime system. Same as the test programs shown in Fig 21, the exception event is also caught by the fifth catch statement in line 13.

The results of scenario one benchmark are shown in Table II. The column “# of searched routines” shows the number of routine comparisons before the suitable exception handling routine is located. It is obvious from Table II that the exception handling overheads of the Java core is much smaller than that of a software-based Java VM. More important, the Sun CVM with the Just-In-Time compilation has even worse performance than the Sun CVM without JIT which is a common technique to accelerate execution of regular Java programs. In the table II, it shows that the JIT has additional overheads when a Java program encounters the exception handling process. As shown in the row one, the overheads of CVM_JIT are 356 times larger than the overheads of dual-core Java and the overheads of CVM are 183 times larger than the overheads of dual-core Java.

	# searched routines	Dual-Core Java # exception handling cycles	Sun CVM # exception handling cycles	Sun CVM_JIT # exception handling cycles
Case 1: Thrown by "athrow"	1	11	2017	3920
	5	73	2300	4487
	10	158	2773	4818
	15	243	3260	5082
Case 2: Thrown by JVM	1	2979	10433	12021
	5	3042	10808	13418
	10	3126	11440	13853
	15	3208	11727	14544

Table II. Time overheads of exception handling process in Scenario One programs

	Dual-Core Java	Sun CVM	Sun CVM_JIT
Averaged overheads of exception handling process when adding a searched exception routine (cycles)	17	87	89

Table III. Time overheads of adding an exception routine in the same try block.

The averaged overheads of exception handling process when adding an additional searched exception routine are shown in Table III. The table shows that it costs 87 cycles on additional exception routine comparing process in Sun CVM. To the dual-core Java, such overheads only cost 17 cycles. It is smaller than the overheads than the software JVM because the comparing process is mainly performed by sequentially comparing the registers with items in a compact on-chip table, which is efficient in hardware solution. And such the process is performed entirely in hardware Java core without external memory or involving interrupt service routines. In the row two in table II, the test program involves five times of searching process. The searching process cost 62 cycles which occupy 84% of the exception handling process. To the CVM, the searching process cost 5*87 = 435 cycles which occupy 19% of the exception handling process. In tells that the software CVM has extra overheads to be ready to search for a suitable exception routine. To the hardware Java core, the process of searching for an exception event in lookup table is performed immediately after it extracts the thrown exception information which only costs 3 cycles. Therefore the exception event can be fast handled by the exception handling unit when the Java program encounters an exception.

In the Table II, we can see that the programs of case two have larger overheads than the programs with same numbers of searched routines in the case 1. The reason is simple that for the handling process for the exception event thrown by JVM, it has an extra step to instantiate the exception object. To Java core, such the process of instantiation of an object is performed by the interrupt service routine which has large overheads due to inter processors communication. However the CVM and the CVM-JIT also has such larger overheads on object instantiation. In the programs of case two, the overheads of exception handling process in CVM are at most 4 times larger than the overheads in the dual-core Java. And the overheads of exception handling process in CVM-JIT are at most 5 times larger than the overheads in the dual-core Java. In the next section, a more complicate scenario will be discussed.

5.2.2 Scenario Two Benchmark Analysis

```
public class Scenario_two_thrown_by _athrow{
   public static void main(String args[]){
        try{
              method1();
        }catch (ArithmeticException e ){
             //exception routine 0
        }
   }

   public static void method1(){
        method2();
   }
   public static void method2(){
        method3();
   }
   public static void method3(){
        method4();
   }
   public static void method4(){
        throw new ArithmeticException ();
   }
}
```

Fig 23. A test program containing five methods and the last method triggers an exception.

In the benchmark of scenario two, the suitable exception handling routine is designed to be located in one of the caller methods. A test program that has five called methods is shown in Fig 23. In the test program, an arithmetic exception is thrown in the last called method. And to find the suitable exception routine to catch this exception, the exception is handed over to each caller method. Finally in the main method, an exception routine defined in line 5 can catch the ArithmeticException type exception event. Next, the program jumps to the line 6 and executes exception routine 0. Other test programs in the scenario two have different numbers of called methods.

The time overheads of exception handling process in each test program are shown in Table IV. In the Table IV, the column "# stack unwinding" stands for the number of stack that be unwind to find the suitable routine. A count of '0' means that the search stops at the current main method where the exception is thrown. Again, the proposed dual-core Java outperforms the CVM and the CVM_JIT. And the CVM_JIT also has the worst performance. Comparing to the scenario one, the programs in scenario two have larger overheads on exception handling process. The reason is simple because that the exception handling processes in scenario two involve the processes of stack unwinding. In the row one in table IV, it shows that the overhead of exception handling process in the dual-core Java is only 11 cycles, which is 356 times smaller than the overheads in Sun JVM_JIT. Note the program tested in row one only contain a main method so that the thrown exception is caught in the first try block, which does not involve the process of stack unwinding.

Thrown Exception Type	# stack unwinding	Dual-Core Java # exception handling cycles	Sun CVM # exception handling cycles	Sun CVM_JIT # exception handling cycles
Thrown by "athrow"	0	11	2017	3920
	4	137	3448	6415
	9	286	5418	7473
	14	443	7313	9965

Table IV. Overheads of exception handling process in the benchmark of scenario two.

	Dual-Core Java	Sun CVM	Sun CVM_JIT
Overheads of exception handling process when adding a method without a try block (cycles)	36	377	445

Table V. Overheads when adding an additional searched method.

In other test programs, the overheads start to grow larger due to the processes of stack unwinding. In the row two of the Table IV, the overheads of exception handling process in the dual-core Java grows to 137 cycles. Comparing to the row one, the difference of overheads 137 – 11 = 126 cycles are the overheads that incurs 4 times of stack unwinding process. The average overheads when adding a searched method are calculated and shown in Table V. In the Table V, we can see that it costs average 36 cycles for exception handling unit to perform the stack unwinding process in Java core. In such process, about 80% overheads are for an external memory access which average costs 27 cycles. As mentioned in chapter 5, the external memory access is to load the local variables count of a method, which is used to locate the return stack frame in the stack unwinding process. If the data of local variables count of a method can recorded in the on-chip table, the overheads for stack unwinding can be reduced to only 7 cycles. However concerning the precious usage on-chip memory space, we leave the data of local variables count to be saved in DDR-SDRAM.

5.2.3 Scenario Three Benchmark Analysis

Similar to the programs in scenario two, the test programs in scenario three also triggers exception event in the last called method. The difference is that the each called methods contains a try block with an exception routine. A test program that has five called methods is shown in Fig 24. In the test program, an arithmetic exception is thrown within a try block in method4. And the resident exception routine cannot handle the exception because the ArithmeticException class is not the sub class of Exception_4. Then the exception is handed over to the caller method to find a suitable exception routine. Again in method3, the exception cannot be handled and is handed over to the call method. The actions of handing over the exception continue until the exception is handed over to the main method. The main contains an exception routine which can catch the ArithmeticException type exception. In the end, the program jumps to the line 6 and executes exception routine 0. Other test programs in scenario three have different numbers of called methods.

```
1:  public class Scenario_three_thrown_by _athrow{
2:     public static void main(String args[]){
3:         try{
4:              method1();
5 :        }catch (ArithmeticException e ){
6:              //exception routine 0
7:         }
8:     }
9:
10:   public static void method1(){
11:       try{  method2();  }
12:       catch (Exception_1 e ){ //exception routine 1 }
13:   }
14:   public static void method2(){
15:       try{  method3();  }
16:       catch (Exception_2 e ){ //exception routine 2 }
17:   }
18:   public static void method3(){
19:       try{  method4();  }
20:       catch (Exception_3 e ){ //exception routine 3 }
21:   }
22:   public static void method4(){
23:       try{  throw new ArithmeticException (); }
24:       catch (Exception_4 e ){ //exception routine 4 }
25:   }
26: }
```

Fig 24. A test program containing five methods and each method contains an exception routine.

The time overheads of exception handling process in each test program are shown in Table VI. In the Table VI, the column "# stack unwinding" stands for the number of stack that be unwind to find the suitable routine. A count of '0' means that the search stops at the current main method where the exception is thrown. Again, the proposed dual-core Java outperforms the CVM and the CVM_JIT. And the CVM_JIT has the worst performance. Comparing to the scenario two, the programs in scenario three have larger overheads on exception handling process. The reason is simple that the exception handling processes in scenario three has additional processes of check exception routine in each called methods.

Thrown Exception Type	# stack unwinding	Dual-Core Java # exception handling cycles	Sun CVM # exception handling cycles	Sun CVM_JIT # exception handling cycles
Thrown by "athrow"	0	11	2017	3920
	4	197	3741	7548
	9	435	6224	11837
	14	675	8481	15567

Table VI. Time overheads of exception handling process in the benchmark of scenario two.

	Dual-Core Java	Sun CVM	Sun CVM_JIT
Overheads of exception handling process when adding a method without a try block (cycles)	36	377	445
Overheads of exception handling process when adding a method containing a try block (cycles)	55	460	837

Table VII. Overheads of additional try block in each method.

The average overheads when adding a called method are shown in Table VII. The results of scenario two remain in the row one and the results of scenario three are shown at row two. Comparing to the programs in the scenario two, we can see that the programs in the scenario three costs additional 19 cycles in dual-core Java. The 16 cycles are used to check an exception routine whether can catch the exception. Such the process involves class ID comparing or JPC ranges checking, which are efficient in hardware solution. For Sun CVM-JIT, we can see that such process has large additional costs if a searched method contain an additional exception routine, which adds ((837-445)/445= 88% overheads.

Considering the fact that the software-based JVM may runs on a general-purpose RISC processor at a frequency higher than that of the Java processor, we provide another experimental result that also tests the benchmark of scenario three. We compare the proposed dual-core Java with Sun CVM that runs on a 300 MHz PowerPC processor, which is the maximal processor clock rate for the Virtex-4 FPGA. In the dual-core Java platform, the Java processor remains at 83.3 MHz. The experimental results are shown in Table VIII. Note that the programs are executed 10000 times and the results are in milliseconds (ms). In table VIII, we can see that the dual-core Java processor still outperforms Sun CVM. By comparing the results in Table VI and Table VIII, we can see the performance of Sun CVM improved by the higher clock rate. In row one of table VIII, one can see that the exception handling overheads of the dual-core Java processor is 148 times smaller than that of the CVM-JIT, while the overhead ratio in Table VI is 356 times. Therefore, although CVM runs at 300/83.3 = 3.6 times higher clock rate, the performance only increases by 356/148 = 2.4 times. In any cases, the dual-core Java processor is a clear winner in exception handling.

Thrown Exception Type	# stack unwinding	Dual-Core Java Exception handling time (ms)	Sun CVM Exception handling time (ms)	Sun CVM_JIT Exception handling time (ms)
Thrown by "athrow"	0	13	903	1928
	4	236	1706	3412
	9	522	2438	4931
	14	810	3167	6696

Table VIII. The Sun CVM runs on a 300 MHz PowerPC platform and the Java processor in dual-core Java remains 83.3 MHz.

Chapter 6. Conclusions and Future Works

In this thesis, we have proposed the design of the architecture to support the exception handling mechanism for a dual-core Java processor. Two of the key architecture proposals are the two-level runtime image hierarchy and the method area manager. The first-level method area offers the Java core a memory-efficiently way to execute a specific method. And to buffer the specific method routine recourses, the method area manager is designed to cooperate with two lookup tables. The two on-chip lookup tables in the method area manager also offer the Java core the capability to retrieve the class-based information and method-based information. For the dynamic method loading, the tables are used to locate the runtimes images in second-level memory space. For the exception handling process, the tables provide the class inheritance information and method exception routines information.

With the two tables and an additional exception lookup table, the proposed architecture can perform exception handling process mainly in Java core. In addition, we adopt the stack unwinding approach in exception handling processes which is suitable for hardware solution. By the stack unwinding approach, the exception overheads can be delayed to the time that the Java program actually throws an exception event. In the final experimental results, it shows that the proposed dual-core Java processor only cost 320 LUTs on the exception handling unit. And it also shows that the dual-core Java processor outperforms the software Sun CVM in exception handling process. Moreover, the Sun CVM with the support of JIT compilation has even worse performance on exception handling than the CVM without the JIT. Not to mention the additional memory overheads on JIT compilation. The dual-core Java may be better suitable for deeply-embedded smart devices.

In the future, the dual-core Java processor will supply more functional features for the Java language. For now, it is not capable of multi-threading executions. To take advantages of the hardware Java core, the hardware-based context-switching for multi-threading is considered to be performed well in Java core. However, the design may increase the memory space overheads such as the space of Java stack, which is another important issue in embedded application processer designs. Therefore a context-switching design without significantly increase the logic size would be investigated in the future.

Appendix: CLDC Exception Library

Class name	Description
Exception	The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.
ClassNotFoundException	Thrown when an application tries to load in a class through its string name using: The forName method in class Class. The findSystemClass method in class ClassLoader . The loadClass method in class ClassLoader. But no definition for the class with the specified name could be found.
IllegalAccessException	An IllegalAccessException is thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor.
InstantiationException	Thrown when an application tries to create an instance of a class using the newInstance method in class Class, but the specified class object cannot be instantiated because it is an interface or is an abstract class.
InterruptedException	Thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class Thread.
NullPointerException	Thrown when an application attempts to use null in a case where an object is required. These include: Calling the instance method of a null object. Accessing or modifying the field of a null object. Taking the length of null as if it were an array. Accessing or modifying the slots of null as if it were an array. Throwing null as if it were a Throwable value. Applications should throw instances of this class to indicate other illegal uses of the null object.
RuntimeException	RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

ArithmeticException	Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.
IndexOutOfBoundsException	Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. Applications can subclass this class to indicate similar exceptions.
ArrayIndexOutOfBounds Exception	Extends IndexOutOfBoundsException. Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
StringIndexOutOfBounds Exception	Extends IndexOutOfBoundsException. Thrown by String methods to indicate that an index is either negative or greater than the size of the string. For some methods such as the charAt method, this exception also is thrown when the index is equal to the size of the string.
ArrayStoreException	Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects. For example, the following code generates such exception: Object x[] = new String[3]; x[0] = new Integer(0);
ClassCastException	Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates such exception: Object x = new Integer(0); System.out.println((String)x);
IllegalMonitorState Exception	Thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.
IllegalArgumentException	Thrown to indicate that a method has been passed an illegal or inappropriate argument.
IllegalThreadState Exception	Extends IllegalArgumentException. Thrown to indicate that a thread is not in an appropriate state for the requested operation. See, for example, the suspend and resume methods in class Thread.
NumberFormatException	Extends IllegalArgumentException. Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

NegativeArraySize Exception	Thrown if an application tries to create an array with negative size.
SecurityException	Thrown by the security manager to indicate a security violation.
EmptyStackException	Thrown by methods in the Stack class to indicate that the stack is empty.
NoSuchElementException	Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.
IOException	Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.
EOFException	Extends IOException. Signals that an end of file or end of stream has been reached unexpectedly during input. This exception is mainly used by data input streams to signal end of stream. Note that many other input operations return a special value on end of stream rather than throwing an exception.
InterruptedIOException	Extends IOException. Signals that an I/O operation has been interrupted. An InterruptedIOException is thrown to indicate that an input or output transfer has been terminated because the thread performing it was interrupted. The field bytesTransferred indicates how many bytes were successfully transferred before the interruption occurred
UnsupportedEncoding Exception	Extends IOException. The Character Encoding is not supported
UTFDataFormatException	Extends IOException. Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by any class that implements the data input interface.
Error	An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The ThreadDeath error, though a "normal" condition, is also a subclass of Error because most applications should not try to catch it.

LinkageError	Subclasses of LinkageError indicate that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
NoClassDefFoundError	Extends LinkageError. Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found. The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.
VirtualMachineError	Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.
OutOfMemoryError	Extends VirtualMachineError. Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

Reference

[1] H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine for Java Bytecode," Computer, Vol. 31, Issue 10, pp. 22-30, Oct. 1998.H.-J. Ko, *A Double-issue Java Processor Design for Embedded Application, Mater thesis, NCTU, 2007.*

[2] W. Puffitsch and M. Schoeberl, "picoJavaII in an FPGA," Proc. of the 5th Int. Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), vol. 231. Sep. 2007, pp. 213-221.K.-N. Su, *Design of Heterogeneous Dual-Core Java Application Processor for Embedded Applications, Mater thesis, NCTU, 2009.*

[3] M. Schoeberl, "A Java Processor Architecture for Embedded Real-Time Systems," The EUROMICRO Journal of System Architecture, 54, 1-2, 2008, pp. 265-286.C.-F. Hwang, *Design of Dual-Core Java Processor for Interactive 3-D GUI Platform, Mater thesis, NCTU, 2010.*

[4] L. Yan and Z. Liang, An accelerator design for speedup of Java Execution in Consumer Mobile Devices," Journal of Computers and Electrical Engineering, 35 (2009), pp. 904-919.Dan Bornstein, "Dalvik VM Internals," *Googol Developer Conference (Google I/O 2008)*, San Francisco, May 2008.

[5] K. B. Kent and M. Serra, "Hard/Software Co-Design of a Java Virtual Machine," Proc. of IEEE Int. Workshop on Rapid Systems Prototyping (RSP), June, 2000.Sun Microsystems, *Connected, Limited Device Configuration Specification*, ver. 1.0a, Sun Microsystems White Paper, May 2000.

[6] Sun Microsystems, J2ME Technology, Sun Developer Network URL: http://Java.sun.com/javame/technology/, 1994-2009.

[7] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, 2nd Ed., Addison-Wesley, 1999.B. R. Montague, "JN: OS for an Embedded Java Network Computer," *IEEE Micro*, 17, 3, 1997, pp. 54-60.

[8] C. Porthouse, High performance Java on embedded devices, Jazelle DBX technology: ARM acceleration technology for the Java Platform, White paper of ARM Ltd., Oct. 2005.C. Porthouse, *High performance Java on embedded devices, Jazelle DBX technology: ARM acceleration technology for the Java Platform,*

White paper of ARM Ltd., Oct. 2005.

[9] Bill Venners, Inside the Java 2 Virtual Machine, New York: McGraw-Hill, 2001, ch.5 ch.6 ch.7 ch.8.A. Krall, "Efficient Java Just-in-Time Compilation," *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 205-212, Paris, Oct. 1998.

[10] C.-H. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," Proc. of 29th Annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO'29), pp. 90-99, Paris, Dec. 1996.H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine for Java Bytecode," *Computer*, Vol. 31, Issue 10, pp. 22-30, Oct. 1998.

[11] A. Krall, "Efficient Java Just-in-Time Compilation," Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 205-212, Paris, Oct. 1998.Sun, *picoJava-II Microarchitecture Guide*, Sun Microsystems, March 1999.

[12] Dan Bornstein, "Dalvik VM Internals," Googol Developer Conference (Google I/O 2008), San Francisco, May 2008.Y. Y. Tan, C. H. Yau, K. M. Lo, W. S. Yu, P. L. Mok, and A. S. Fong, "Design and Implementation of a Java processor," *IEE Proceedings*, Vol. 153, pp. 20-30, 2006.

[13] Y. Shi, D. Gregg, A. Beatty, and M. Anton Ertl, "Virtual Machine Showdown: Stack versus Registers," ACM Transactions on Architecture and Code Optimization (TACO), Vol. 4, Issue 4, pp.153-163, Jan. 2008.Xilinx LogiCore, *LogiCORE IP Multi-Port Memory Controller (MPMC) (v6.03.a)*, Xilinx Production Specification DS643, March, 2011.

[14] Sun, picoJava-II Microarchitecture Guide, Sun Microsystems, March 1999.Xilinx LogiCore, *PLB IPIF (v2.02a)*, Xilinx Production Specification DS448, April, 2005.

[15] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling," Proc. of 1999 Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99), pp. 34-39, Newport Beach, Oct. 1999.

[16] Y. Y. Tan, C. H. Yau, K. M. Lo, W. S. Yu, P. L. Mok, and A. S. Fong, "Design and Implementation of a Java processor," IEE Proceedings, Vol. 153, pp. 20-30, 2006.

[17] M. Schoebel, "Evalution of a Java Processor," Tagungsband Austrochip 2005, pp. 127-134, Oct. 2005.

[18] H.-J. Ko and C.-J. Tsai, "A Double-issue Java Processor Design for Embedded Application," Proc. of IEEE Int. Symp. on Circuits and Systems(ISCAS'08), Seattle, May. 2007.

[19] H.-J. Ko, A Double-issue Java Processor Design for Embedded Application, Mater thesis, NCTU, 2007.

[20] C.-F. Hwang, K.-N. Su and C.-J. Tsai," Low-Cost Class Caching Mechanism for Java SoC," Proc. of IEEE Int. Symp. on Circuits and Systems(ISCAS'10), Paris, May. 2010.

[21] C.-F. Hwang, Design of Dual-Core Java Processor for Interactive 3-D GUI Platform, Mater thesis, NCTU, 2010.

[22] K.-N. Su and C.-J. Tsai, "Fast Host Service Interface Design for Embedded Java Application Processors," Proc. of IEEE Int. Symp. on Circuits and Systems (ISCAS'09) ,Taipei, May, 2009.

[23] K.-N. Su, Design of Heterogeneous Dual-Core Java Application Processor for Embedded Applications, Mater thesis, NCTU, 2009.

[24] H.-W. Kuo, Design of Java Accelerator IP for Embedded Systems, Mater thesis, NCTU, 2011.

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

[26] R. Radhakkrishn, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, "Architectural Issues in Java Runtime Systems," Proc. 6th Int. Symp. on High-Performance Computer Architecture, 387-398, 2000.