國立交通大學

資訊科學與工程研究所

碩士論文

適用 3-D 互動人機圖形介面系統的雙核心 Java 處理器設計

Design of Dual-Core Java Processor for Interactive 3-D GUI Platform

研究生：黃建峰

指導教授：蔡淳仁　教授

中華民國九十九年七月

適用 3-D 互動人機圖形介面系統的雙核心 Java 處理器設計
Design of Dual-Core Java Processor for Interactive 3-D GUI Platform

研究生：黃建峰　　Student：Chien-Fong Huang

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

June 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年七月

Abstract

Java is becoming a more popular language for embedded systems. Modem embedded systems do not only execute single task, but often execute many interactive multimedia applications simultaneously. For example, complex UIs based on touch screen are very popular today. The goal of this thesis is to design a heterogeneous dual-core Java platform that can support complex 3-D virtual man-machine interaction. The Java platform is derived from previously published work. In this thesis, hardware and software components are added into existing platform to enable dynamic class loading so that the proposed dual-core Java platform can support large-scale Java programs. In addition, a camera calibration process is also developed in this thesis so that a pair of stereo cameras can be used to capture the 3-D motion of the human operator. With 3-D motion input capability, the proposed platform will be capable of efficient execution of 3-D interactive GUI systems.

The proposed dynamic class loading mechanism has been implemented on the Xilinx Virtex-5 ML507 FPGA development board, and verified the proposed design with a subset of Embedded Caffeine Mark. The performance of our design (without data cache) is about 3~ 9 times faster than CVM running on RISC processor with data cache at the same clock rate. We also verified the camera calibration algorithm using simulated image sequences and the errors of camera calibration process is about 5~10 mm for a virtual object located about 700 mm away from the camera.

Acknowledgement

在完成這篇論文的同時, 首先必須要感謝我的指導教授蔡淳仁教授, 老師在我求學的時期給我很多學習的經驗跟機會, 在平日的指導上也願意付出很多耐心教導傳承許多研究經驗, 以及增廣研究學習的角度, 也讓我了解到自己的缺點跟不夠細心, 讓我在研究所的這兩年期間能夠保持耐心, 也接觸到許多不同研究的課題, 讓我在實作上有較多的接觸跟許多不同的實作經驗, 也感謝實驗室的學長帶領我們一起執行國科會的計畫, 透過計畫的合作互相研討, 也認識了其他實驗室的同學並且能夠多看看別人努力的成果, 在整合計畫的過程中也學到與人溝通合作的重要性, 也感謝實驗室學弟願意配合一起努力學習, 也讓我了解自己仍有許多地方需要努力學習, 很高興在實驗室的夥伴願意和我一起努力研究, 讓我自己在這兩年中間能夠有所成長。

Content

List of Figures

List of Table

Introduction

In this chapter, the motivation behind the work in this thesis is presented. With the application platforms for embedded systems converging towards Java Runtime Environment (JRE), we try to design a heterogeneous dual-core SoC that can support 3-D interactive man-machine interface efficiently. The work done in this thesis is part of the ultimate goal. In the following sections, we will give an overview to the proposed JRE and the derivation of the camera calibration process.

1.1. Motivation for Proposed Java Runtime Platform

JRE (Java Runtime Environment) is well known for Java platform today especially for embedded systems such as mobile phones and set-top boxes. Sun Microsystems had defined Java 2 Micro Edition (J2ME) [5] framework and had different configurations and profiles depending on different embedded applications and devices such as Connected Limited Device Configuration (CLDC) [6] and Mobile Information Device Profile (MIDP), etc.

Traditional JRE is composed of a software-based Java Virtual Machine (JVM) [10] running on a full-blown operating system. The JVM must execute the bytecodes and provide system interface or dynamic linked library interface for method execution. For object-oriented Java language, there are many performance issues for embedded systems with a RISC CPU, For example, operations such as simulation of a stack-machine, dynamic symbol resolutions, and heavy dynamic memory allocations, are expensive for embedded processors. There are many solutions for improving the performance of JRE for embedded systems. Just-in-Time (JIT) compilers or the hardware-based co-processors are common approaches for embedded Java platform.

However, JIT requires extra memory and imposes extra compilation overhead for class loading. Architechture exetension such as ARM Jazella [10] are tied to specific processor architecture and are not generally available for any host processors.

The Java platform in this thesis is a heterogeneous dual-core system, which is composed of a generic RISC processor and a hardwired Java bytecode execution engine. The generic RISC processor works for tasks, such as I/O and control, which are inefficient for stack-based processors to execute. The Java core execution engine is responsible for general bytecode execution.

The dual-core Java application processor adopted in this thesis is based on the work done in [2][4]. However, previous implementations of the architecture only support statically linked classes. That is, all Java classes must be loaded into the on-chip memory blocks and all dynamic linking information is parsed and resolved before execution. In this thesis, full dynamic class loading and symbol resolution mechanism is proposed to enhance the function of previous system. The design is based on a software-hardware co-design principle such that one-time complex symbol resolution tasks are partitioned and assigned to the RISC core while repeated bytecode execution tasks are completely handled within the Java core. With this partition rule, the impact of inter-processor communication (IPC) cost is highly reduced and the overall performance is improved significantly, compare to a software-based JVM.

1.2. Camera Calibration Process

Since the proposed JRE is targeted for interactive multimedia applications with virtual 3-D man-machine interface, we have to design a subsystem within the dual-core Java SoC to capture human operator 3-D (hand) actions. When combined with a 3-D display device, the proposed system will enable virtual 3-D touch screen GUI. The technologies of 3-D display devices have been developed for a long time. In order to produce of 3-D visual effects for viewer a general approach is to generate two views In front of the views, one for each eye. The viewer may need to ware either red-cyan, polarized, or LCD shutter glasses [22] in order to see different images in each eye. The viewer is then able to have a synthetic feel of the depth of objects from 2-D frame images. The two views of a scene can be generated directly from two cameras or synthesized from only one view and a depth map. There are also new technologies based on lenticular or barrier screens that can display multi-views at simultaneously such that multiple viewers can all watch 3-D video together without wearing glasses.

In order to capture the hand operations of the human operator in front of the 3-D display, one approach is to adopt the techniques in stereo computational vision research. In short, a pair of stereo cameras can be used to estimate the 3-D position and motion of the operating hand by triangulation. The first step towards this goal is to set up a pair of calibrated cameras connected to the dual-core Java platform. The second part of this thesis is to design a simple camera calibration process for the proposed platform so that camera parameters can be estimated for the purpose of triangulation of 3-D objects.

The organization of the thesis is as follows. Previous work on Java processors and camera calibration process is presented in Chapter 2. Details of the dual-core Java application processor architecture is described in Chapter 3. The new hardware-software codesigned dynamic class loading mechanism is proposed in Chapter 4. Chapter 5 discusses the implementation of the camera calibration process. Chapter 6 shows experimental results of the dynamic class loading mechanism and the camera calibration process and finally, some conclusions and discussions are given in chapter 7.

Chapter 2. Previous Work

The work done in this thesis is part of a project that designs a Java-based interactive 3-D man-machine interface (MMI) system. Two of the key components implemented in this thesis are the dynamic class loader for a Java processor and the camera calibration algorithm for stereo cameras used in MMI.

2.1. Dynamic Class Loading in Java Runtime Environment

We have presented the motivation of the proposed dual core Java application processor in chapter 1. In this thesis, we propose a design of dynamic class loading mechanism for Java processors. Class loader is an important feature of Java environment. Class loader loads the class files which produced by Java compilers from Java program sources. Class file format defines the organization of Java method bytecodes, constant pool data, and method argument flags, etc., in an executable file. Java virtual machines execute bytecode and use class loaders to load class files. Through dynamic class loaders, a Java system can download new Java applications from the Internet or storage spaces. A Java system with dynamic class loading is more powerful and flexible.

The JVM provide several modes for class loading such as Lazy Loading or User-definable class loading policy[12][13]. JVM has an embedded default class loader in Lazy Loading mode. The class loader in Lazy mode loads class files on demand. There are usually two cases for JVM to do class loading. One creates the class object reference and the other one is for method invocation reference. The class loader is a very complex module in JVM. In the beginning of the class loading process, a JVM resolves the class name in constant pool from bytecode and follows the class loading flow as shown in Fig. 1.

Fig. 1 describes general class loading flow of JVM. The class loader searches the class method area, jar archives, and class paths for the target classes first. Then it computes new resolution information of the newly loaded class to the resolution structure and cache table. There are also verification steps for namespaces, method invocations, and security issues during class loading process. The class loading process is too complex for embedded systems. There are some approaches for optimization such as adjusting the search hash mapping function, collect runtime information of environment for static method acceleration, etc.

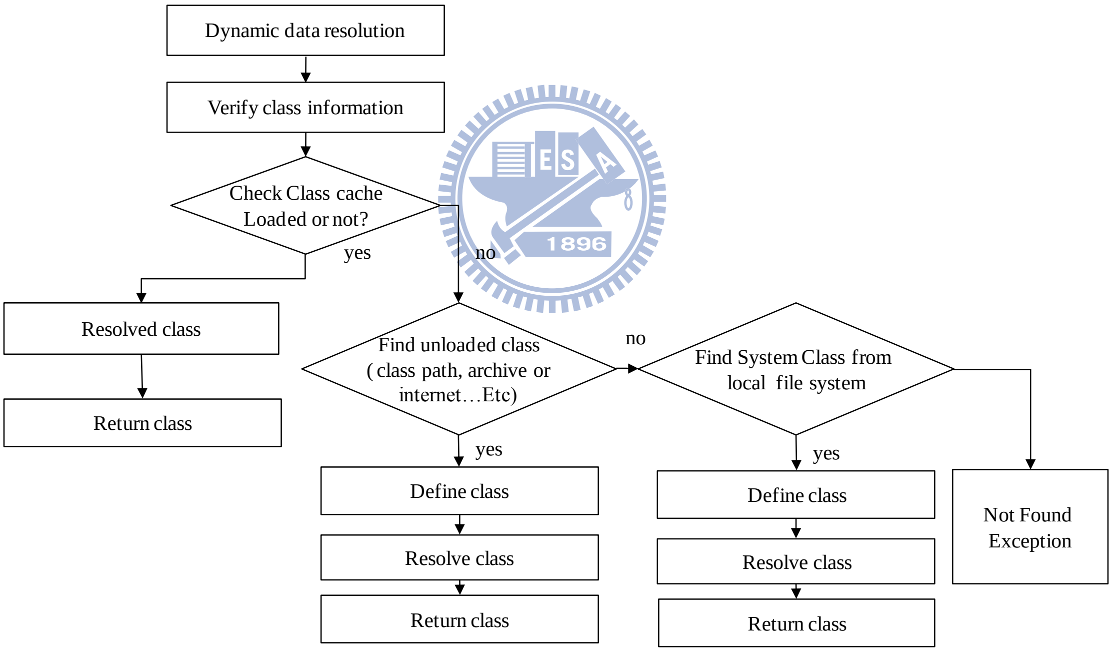


Fig. 1. Flow chart for class loading of JVM.

For the proposed embedded multimedia Java runtime execution environment, we adopt a heterogeneous dual-core SoC system with a Java bytecode execution engine and a RISC processor. In this thesis, we implement dynamic class loader for the proposed system. The implementation in this thesis is based on a previous Java system, which has a static class loader [2][4]. In the old system, the class loader parses all class files and convert them into runtime information images and reserves all resolution information in the constant pool of the image. Fig. 2 shows the system life cycle of the previous platform. After system boot up, host RISC processor initializes system memory and parses all class files before execution. After all class files are parsed into information images, the system loads the boot class into Java method area and enables the Java execution engine.

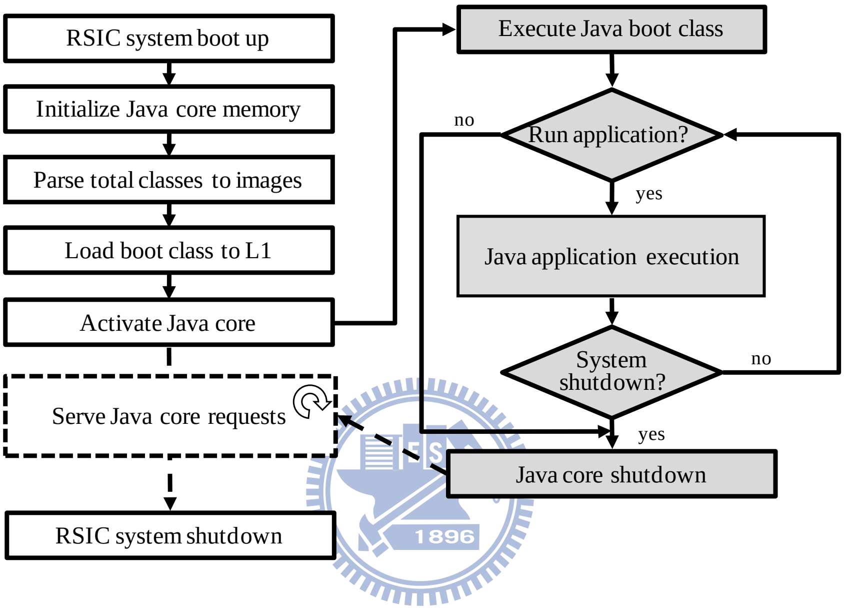


Fig. 2. System life cycle of previous Java platform.

Fig. 3 describes the architecture of previous version of Java core [4]. The Java core execution engine is composed of four pipeline stages which fetch, translate, decode, and execute bytecode and micro-instructions. There are some issues with previous system which will be described later.

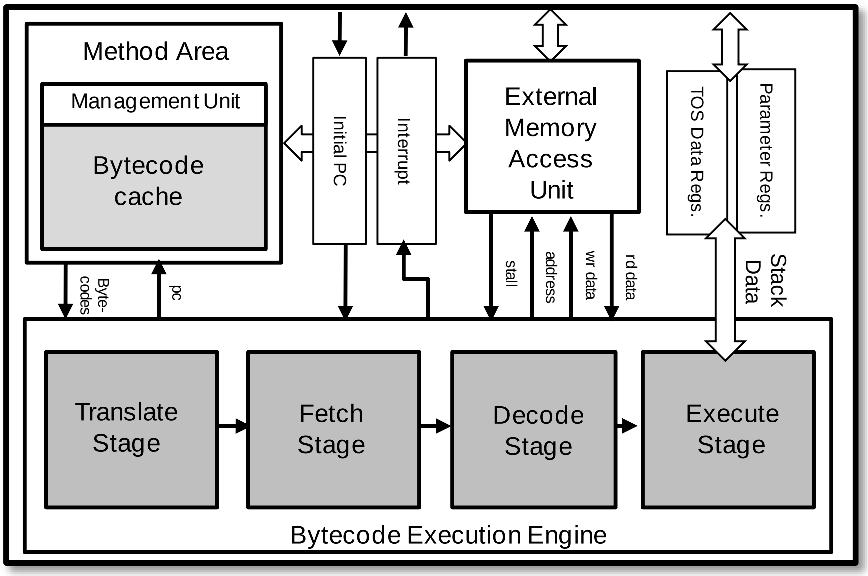


Fig. 3. Previous Java execution engine [4].

Some behaviors of Java bytecode are too complex for hard-wired implementation such as method invocation or field data access, etc. These behaviors require string resolution, which is too expensive for hardware execution. Previous design provides a dynamic resolution state machine to control the status of Java core execution engine. The main task of dynamic resolution controller is to handle symbol resolution of constant pool data. For example, the states of method invocation do resolution and get information which produced by class loader before execution as shown in Fig. 4. This state machine also controls and changes the program counter for referenced method bytecode.

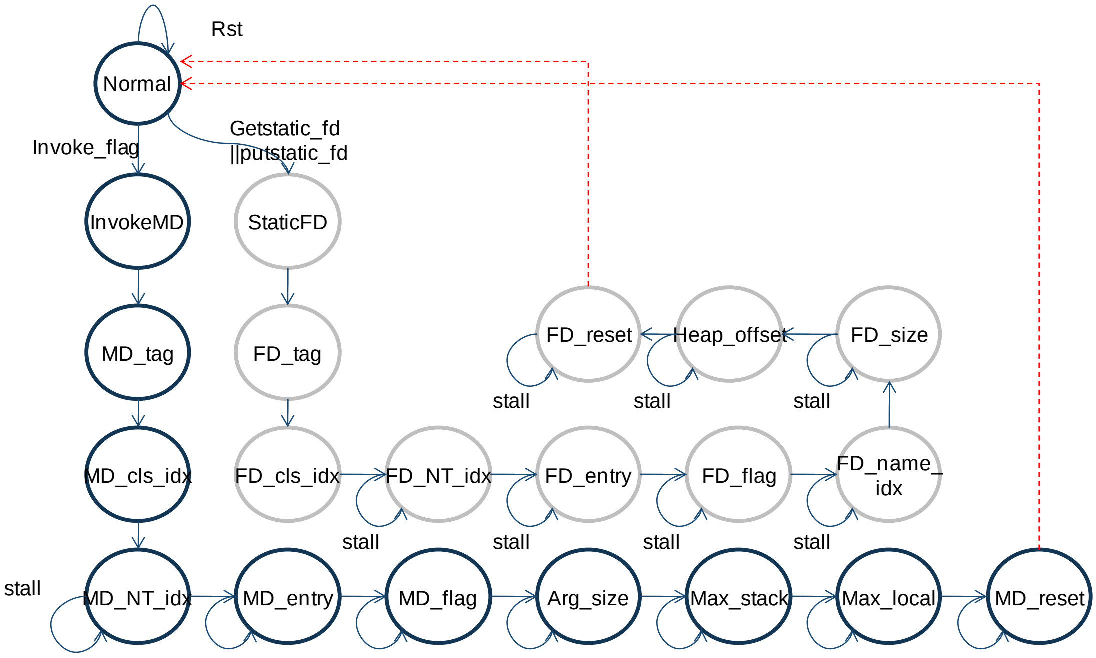


Fig. 4. Original controller state machine for method invocation resolution.

Finally, previous design does not reference to dynamic data information at run time and all essential information at runtime must be stored in runtime information image of a class. Previous design parses all class files before Java program execution. To make the Java SoC more flexible and stable, we redesign some subsystems and added the function of dynamic class loading. In chapter 4 I will present the design for dynamic class loading and discuss some additions to improve the Java execution engine.

2.2. 3-D MMI based on Stereo Cameras

2.2.1. 3-D and Image Coordinates Systems

We have presented the motivation of integration between the 3-D MMI and the proposed Java platform in chapter 1. A 3-D GUI system interacts with human operators' 3-D hand motions, captured by stereo cameras. The proposed MMI system computes the 3-D coordinates of the target object using the projection of the target object on the image plane and the camera parameters. These parameters contain camera intrinsic parameters such as effective focal length and sensor size, etc., and extrinsic parameters such as orientation and translation parameters. These parameters can be obtained by calibrating the camera using known landmark points. In this thesis, we use the camera calibration algorithm proposed by Tsai [15]. The camera model used here is the pinhole camera model. There are other camera models such as the orthographic projection model or the affine projection model.

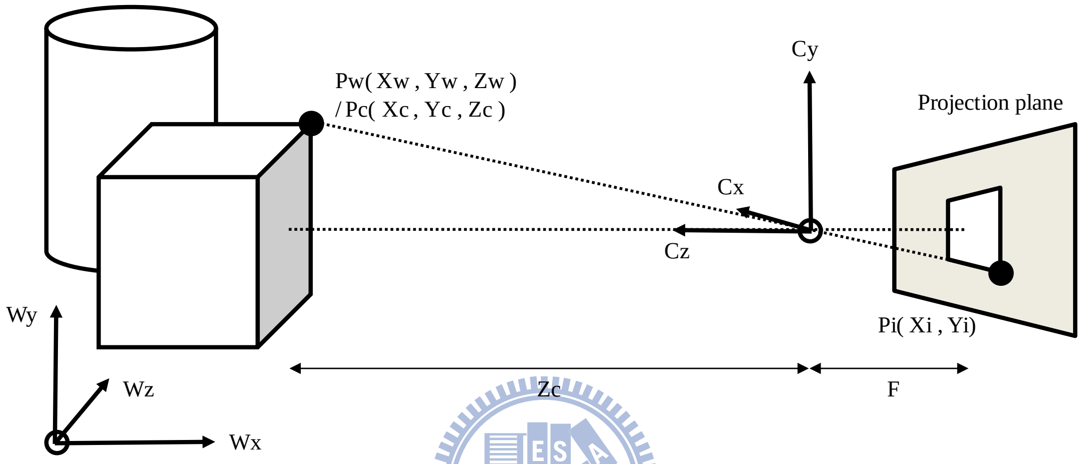


Fig. 5. Pin-hole camera model.

Fig. 5 illustrates the pin-hole camera model. There are two 3-D coordinate systems shown in Fig. 5, the camera coordinate system and the world coordinate system. Traditional pinhole camera model converts the world coordinate system (WCS) into camera coordinate system (CCS) first, then projects 3-D objects to image plane and computes its 2-D positions w.r.t. image coordinates system (ICS) of the computer.

The first step in 3-D WCS to 2-D ICS projection is the rigid body transformation from (X, Y, Z)$_{\text{WCS}}$ to (X, Y, Z)$_{\text{CCS}}$ with 3×3 Rotation matrix *R* and 3×1 Translation vector *T*. That is,

$$[X\ Y\ Z]_{CCS} = \boldsymbol{R} * [X\ Y\ Z]_{WCS} + \boldsymbol{T} \tag{1}$$

$$\boldsymbol{R} = \begin{pmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{pmatrix} \boldsymbol{T} = \begin{pmatrix} T_x \\ T_y \\ T_z \end{pmatrix} \tag{2}$$

The second step is to project the object, denoted in CCS, to ideal undistorted image coordinate (X, Y) $_{\text{Ideal}}$ using perspective projection of pinhole camera model as follows:

$$X_i = f * {}^{X_c}/_{Z_c} \tag{3}$$

The variable f is the focal length of the camera.

Thirdly, the effect of lens distortion is taken into account and compute the coordinate of image plane with lens distortion (X, Y) $_{\text{Distortion}}$

$$X_d + D_x = X_u, D_x = X_d(K_1 r^2 + K_2 r^4 + \cdots) \tag{4}$$

Fourth, we scale the image plain into computer image (X, Y) $_{\text{frame}}$ which is pixel based frame image by some scale factor.

$$X_f = S_x d_x'^{-1} X_d + C_x \; Y_f = d_x'^{-1} Y_d + C_y$$

$$d_x' = d_x \frac{N_{cx}}{N_{fx}} \tag{5}$$

After below steps, the 3-D real objects are projected onto the 2-D image plane. We can adopt these images from different viewpoints to compute the 3-D coordinates through triangulation as shown in Fig. 6. Note that we need at least two views in order to perform triangulation.

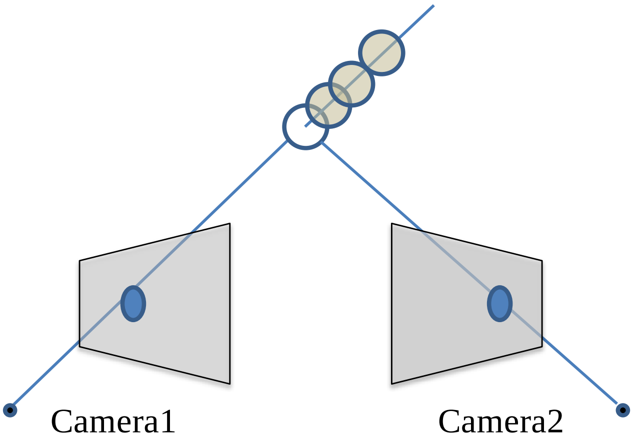


Fig. 6. Compute original 3-D coordinate from two views.

The triangulation process requires camera parameters, which can be obtained through camera calibration. Next sub-section describes the basic concept of camera calibration.

2.2.2. Overview of Camera Calibration

Camera calibration can be divided into two categories. One is photogrammetric calibration and the other one is self-calibration. Basic photogrammetric calibration approaches use a camera to observe a set of known patterns called calibration landmark points and use the known 3-D to 2-D correspondence of these landmark points to calculate the camera parameters. Self-Calibration can calibrate the camera without any calibration landmark points [21]. Self-Calibration captures the static scene and generates image sequence by moving the camera. This technology estimates the calibration parameters from image sequence which captured by the same camera. The former is more accurate and more efficient then letter. But the former is also much more expensive. However the interaction interface between human and machine needs more reliable information especially for the motion of hand. There are also other approaches for specialized environment. We give an overview of another algorithm of photogrammetric calibration.

The camera calibration technology developed by Z. Zhang is also well-known [20]. This algorithm provides a flexible calibration process without specialized 3-D geometry. The camera only captures the planar pattern from anywhere. However some parameters such as lens distortion should be modeled. This calibration technology adopts a nonlinear approach based on the maximum likelihood criterion to solve the matrix derived from homograph between pattern and image. Even though technology of Z. Zhang is flexible and accurate, the calculation is too complex for embedded systems to do nonlinear optimization. Therefore, we adopts the camera calibration algorithm by Tsai [15], which only use linear equations and general least square optimization which can easily be implement for embedded systems. The details of this algorithm will be presented in chapter 5.

Chapter 3. Overview of Java Processor

3.1. Overview

The dual-core Java application processor in this thesis is based on the architecture described in [2][4], which is a double-issue four-stage pipeline stack processor. The stages are Translate, Fetch, Decode, and Execute as shown in Fig. 7. The Java core also contains a method area and a cache management unit. Furthermore, Java core also supports three communication interfaces. One is the internal processor communication (IPC: interrupt) interface between RISC core and Java core. Java core request RISC for services through IPC interface. Another interface is bus master interface of Processor Local Bus (PLB v4.6). Java core can access other devices such as DDR memory through master bus interface. The third interface is a set of data registers. These data registers are used for information exchange with RISC processor. There is a thin kernel running on RISC processor and this kernel provides several services such as I/O, dynamic class loading or native function invocations. The IPC transmit function parameters to the RISC core through data registers and an interrupt service ID register. The system bus protocol is based on the Processor Local Bus (PLB) bus of the IBM CoreConnect bus [24]. The remaining sections of this chapter describe some details of the previous architecture. Some controller design and implementation flaws that lead to stable issues of existing Java core will also be discussed. The design and integration of dynamic class loading mechanism into existing architecture will be described in next chapter.

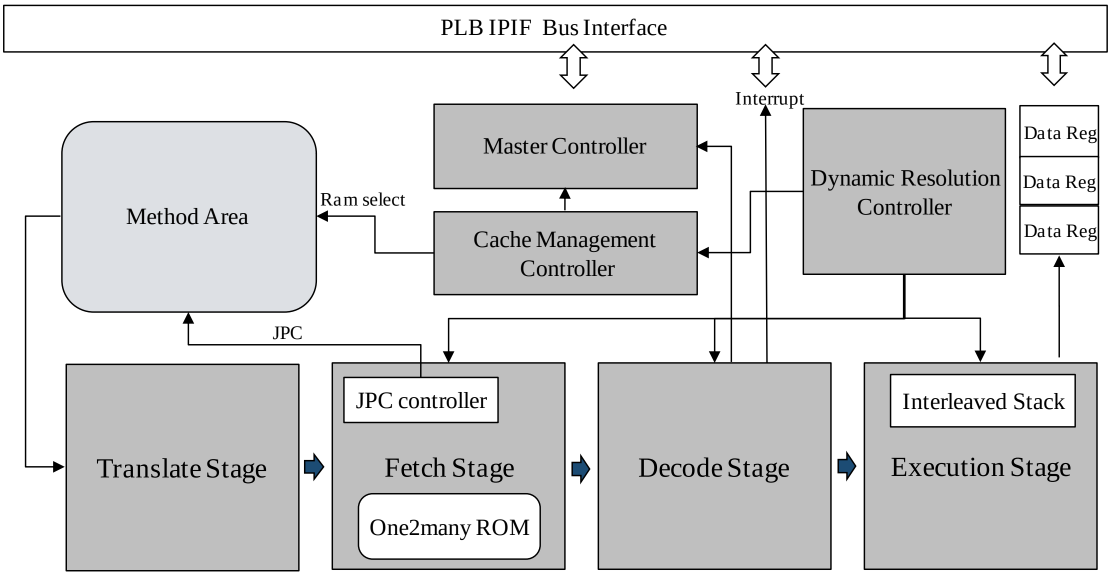


Fig. 7. Overall architecture of Java core.

3.2. Bytecode Execution Core

Fig. 7 shows the overall architecture of previous Java core. This Java core has a dynamic resolution controller unit which controls the status of Java core. Java core executes bytecode in normal mode. If a bytecode invokes dynamic resolution for data access in class runtime information image, the Java core will leave the normal mode and enters simple dynamic data resolution mode. The following sections describe the details for each component of Java core.

3.2.1. Translate Stage

This section describes the first stage of proposed pipeline architecture. Because the Java bytecode adopt variable code lengths and some bytecode behaviors are too complex to execute in one cycle, the proposed design achieves the same behavior of complex bytecode with simple instructions sequence. System separates the bytecode into two types in this stage. One type is one-to-one mapping type which the simple task can be done in one cycle. The other type is one-to-many mapping which complex task can only be done with a sequence of simple instructions. Simple type bytecode will directly be translated into one simple instruction for Java core. Then complex type bytecode will be translated into an ROM address mapping to an instruction sequence at Fetch stage.

This stage also determinates whether the bytecode are operands or not. If the bytecode are operands, these bytecode should not be translated and will be pushed into operands buffer at next stage. In order to follows double-issue, system fetch two bytecode at a time. Then system need to exactly know which one of the two bytecode should be executed, especially after jump, branch, and a data resolution for JPC variation. More details are described in [2].

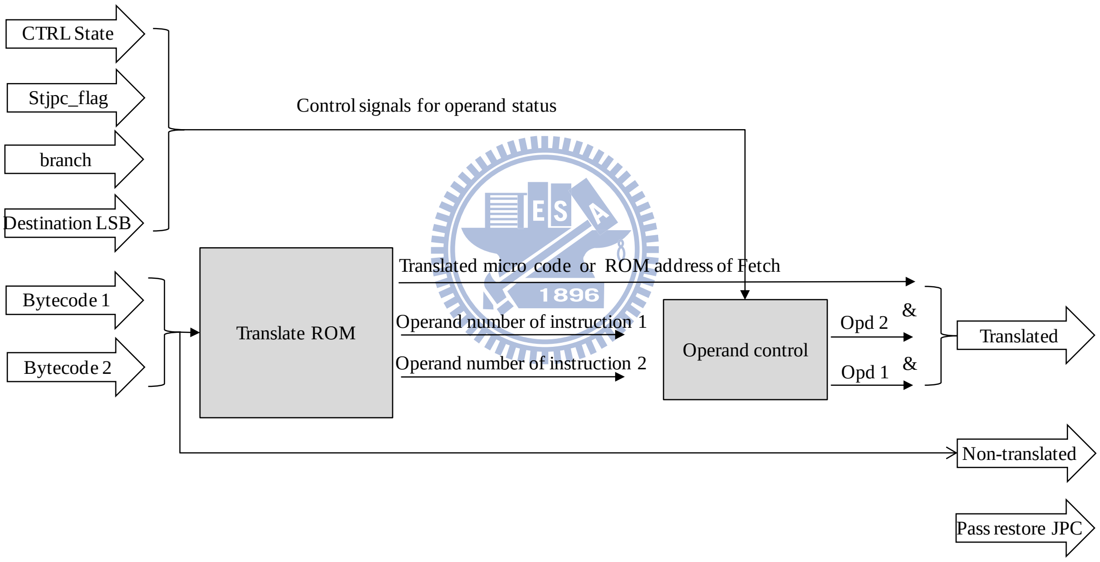


Fig. 8. Architecture of translate stage.

3.2.2. Fetch Stage

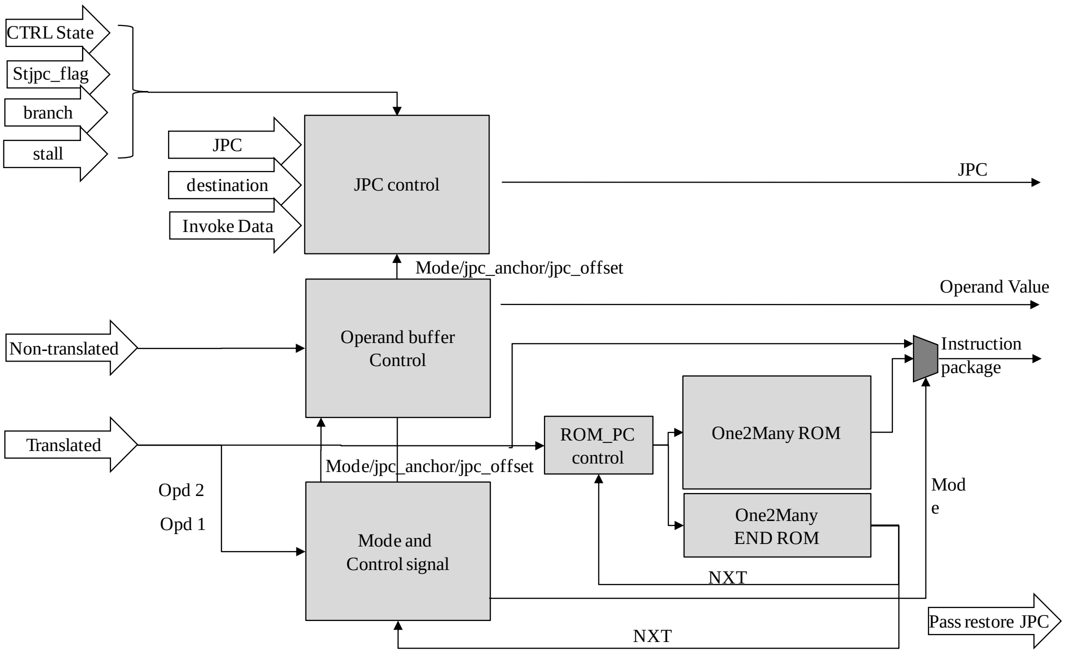


Fig. 9. Architecture of fetch stage

The Fetch stage is a most complex module as shown in Fig. 9. This stage has many important tasks such as Java core program counter (JPC) control, operands buffer control, and mode control (Complex bytecode execution mode/Simple bytecode execution mode), etc. Main purpose of Fetch stage is to generate correct instructions package for decode stage.

There are two cases of the instruction packages which transferred to Decode stage. First case, the bytecode of Translate stage is simple and the simple instructions package is obtained directly from former stage. Second case, the bytecode of Translate stage is complex bytecode and the instructions package is fetched from the one-to-many ROM by translated ROM address. Unfortunately, the former statge sometimes put the ROM address and the simple bytecode into the same package. Then the mode translation between passing simple bytecode or fetching bytecode from one-to-many ROM becomes a difficult problem as shown in Fig. 10.

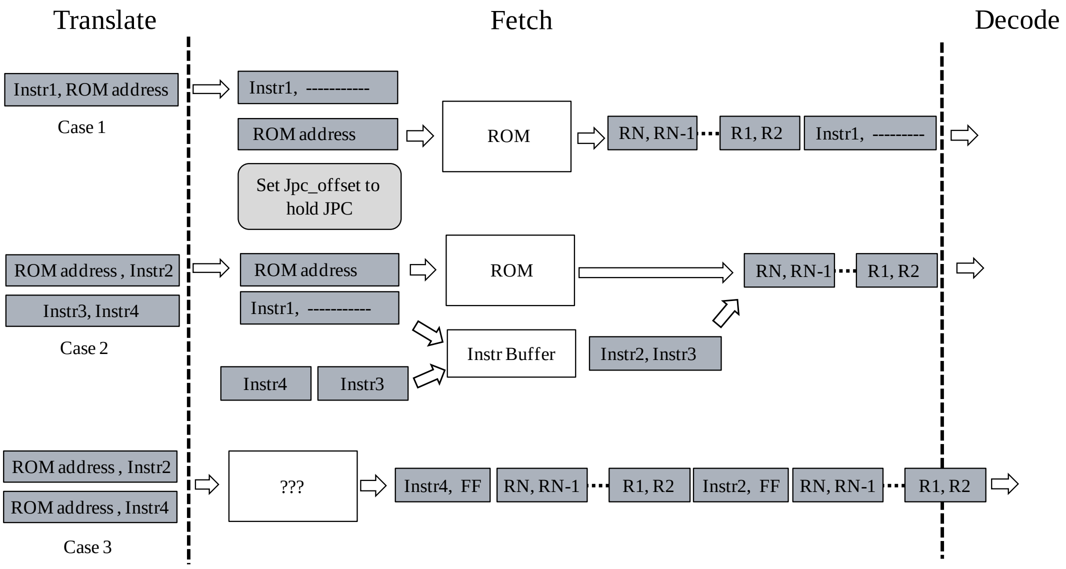


Fig. 10. Some special conditions for instruction packages.

As Fig. 10 shows, this stage holds JPC in order to fetch instructions from One2Many ROM. At the same time, system also stalls the Translate stage for complex bytecode execution. System packages the instructions from ROM or directly from last stage at complex bytecode execution mode. Moreover, the packages from last stage also can be one or more operands. Finally, this stage should package instructions and operands correctly no matter for any combinations. This stage handles more JPC operations from above reasons, so we directly put JPC control module into Fetch stage.

3.2.3. Decode Stage

Decode stage generate control signals from instruction packages from Fetch stage. And the Fetch stage ensures all packages transferred to Decode stage which must be composed of only simple instructions. This stage would focus on generating appropriate data paths for different behaviors of instructions packages. In the beginning, system classes the data paths with instruction package types. The details and data paths of instructions pair types can be found in [2]. This stage not only generates the data path control signals but also picks correct value from immROM or operands buffer for operations or stack access. The interrupt control module and bus data master access control module are also included in the stage. Some instructions like “new” or method invocation will invoke the interrupt signal at this stage. And the bus master access modules only control the target address and R/W request signal. The data transferred from the bus will handle at Execute stage.

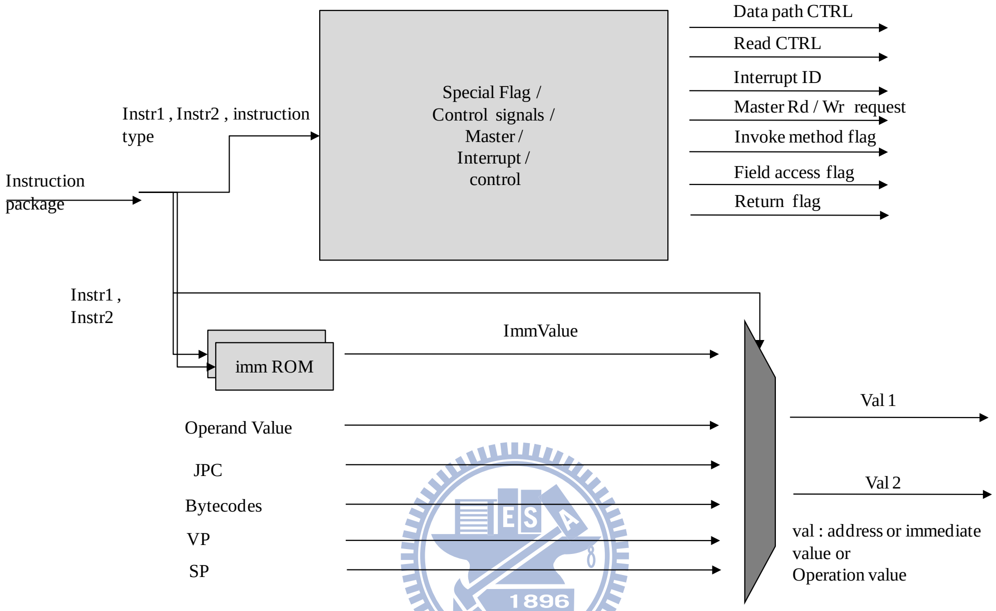


Fig. 11. Architecture of decode stage.

3.2.4. Execute Stage

Fig. 12 shows the execute stage of the Java core. This stage is for the execution of ALU operations, stack operations, and external data accesses. The control signals of data paths control data sources, operations, multiplexers, and interleaved stack operations. The control signals of decode stage handle stack operation and stack pointers, where the interleaving stack structure makes things very complex for operations. The stack structure will be described in next section. Execute stage executes operations indicated by the control signal. Moreover, this stage has the critical path here from many levels of multiplexers and the multiply operations.

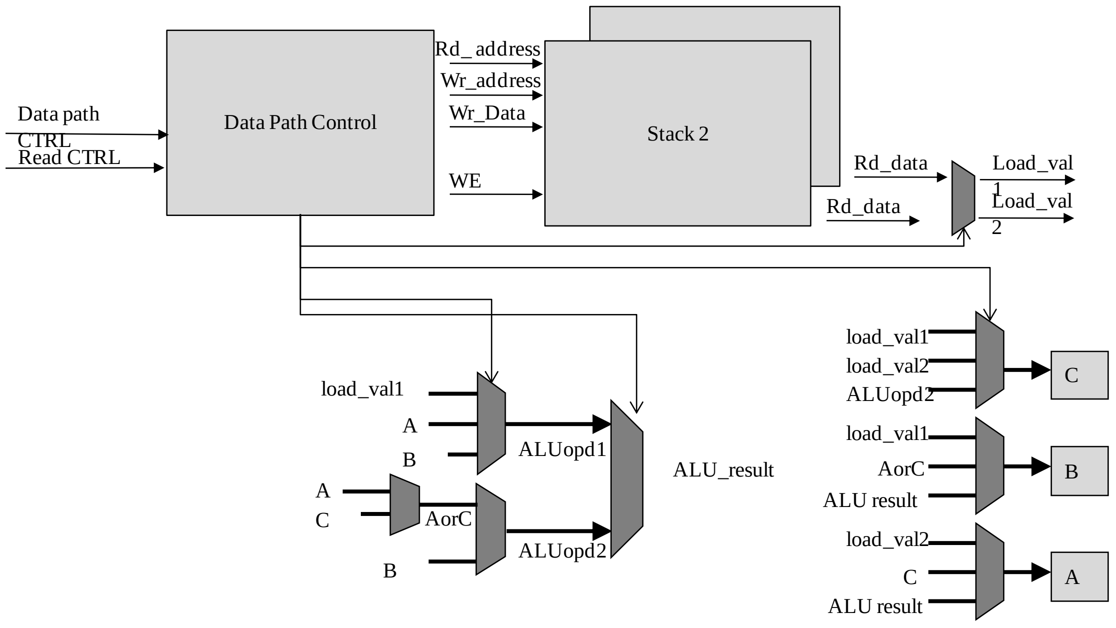


Fig. 12. Architecture of execute stage.

3.2.5. Stack Structure.

In this section, the interleaved stacks structure is resented, as shown in Fig. 13. In order to support double-issue, the stack memory is composed of two SRAM banks. And the two SRAM banks are interleaved together to enable simultaneous access of consecutive words of memory cells. All operations express with three top registers A, B, and C. The stack status can be expressed by two register SP and VP. The register SP is the stack pointer that points to the top of stack. The register VP points to the base address of local variables and arguments of current stack frame of the invoked method. The initial SP of each class should be placed after all local variables and method arguments from VP as shown in Fig. 13. The push and pop stack operations are described as follows.

- Push Operation: Load type instruction pushes data into top stack register and the original data of bottom register C will be pushed back into the stack base on SP pointer.
- Pop Operation: Store type instruction or ALU operations pop data from

stack and store data back to the locals or still store to top registers.

The concepts of the push and store operations are illustrated in Fig. 13 as well. The ALU operation is shown as Two loads and an ALU operation. If the numbers of top stack register are reduced, system needs pop data for SP pointer. We can just take the gray part as operand stack and the stack space between SP pointers and VP pointer stand for local variables or arguments.

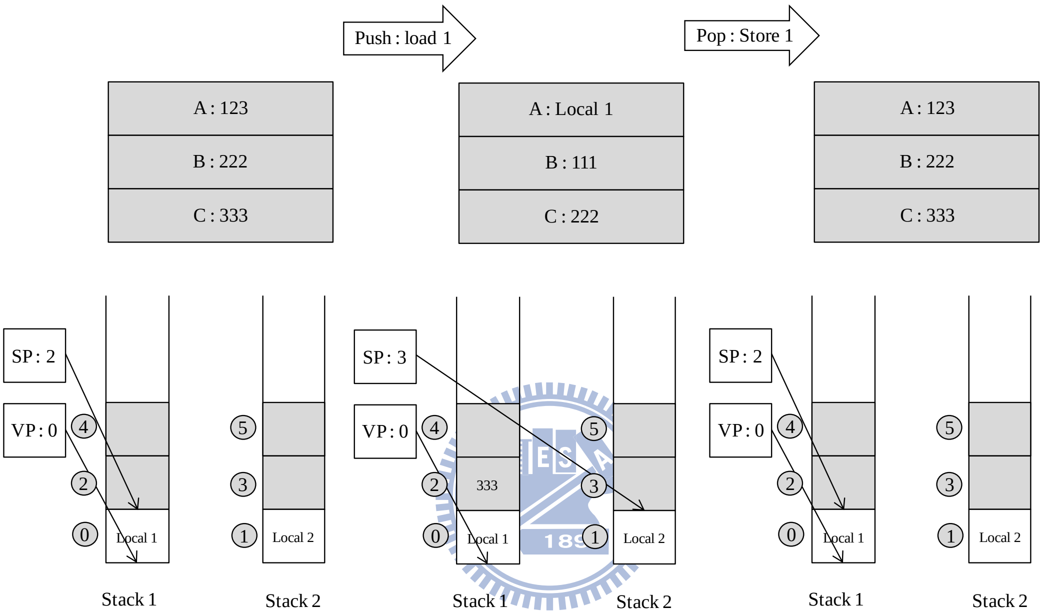


Fig. 13. Basic operations of interleaving stack memory.

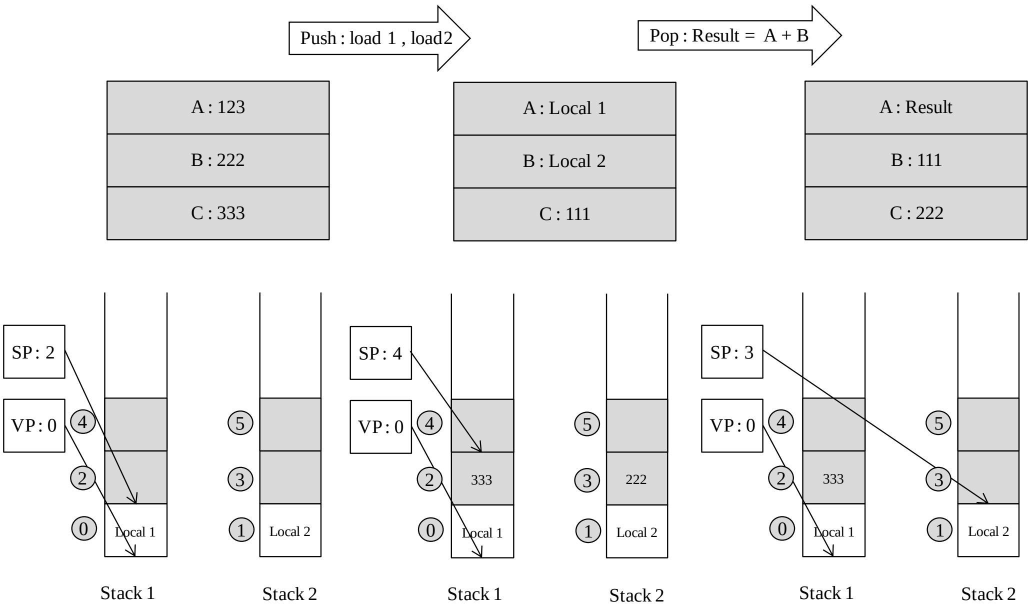


Fig. 14. Two loads and an ALU operation.

3.3. Cache Management Unit

In this section, I will introduce the cache mechanism for existing Java core. As Fig. 7 shows, Java core fetch bytecode directly from method area. Previous method area has 32 2KBytes cache blocks. And the average size of standard Java system library is about 8 Kbytes, the method area is enough for usage. Then Java core manages the cache mechanism with a cache management unit. The cache management table contains the information of class runtime image such as the image size, the start address of external memory, etc. An entry of the table is shown in Table 1. This table is included in the cache management unit.

Cache Manage Table			
Global Index Tag	Image Size	1st Cache Block Start Index	2st External Memory Start Address

Table 1. Management Table of Cache Management Unit

When the Java core performs method invocation, it can query this table using the class global index. If the referenced class has been loaded into the method area, the Java core can adjust the JPC to the target method area from the start block index. This method area start index indicates the first cache block index of referenced class. On the other hand cache management mechanism loads referenced class from external memory by external memory start address field. Then the cache mechanism enters a loading loop for the referenced class image until data counter over the image size. This cache mechanism for class image loading is often invoked by method invocation which will be introduced in section 4.5.

Chapter 4. Design of Java Dynamic Class Loading Mechanism

This chapter presents the proposed dynamic class loading mechanism. We implement the proposed dynamic class loading mechanism with hardware-software codesigned method. This proposed mechanism is composed of two steps. We use software (executed by the RISC core) to parse the class files and collect runtime information. This software will generate a bytecode image with reference pointers and a cross reference table. The pointers in this image point to this table directly. The table contains all runtime information which collected from the original class file. This first step only executed by a new class loading event. After the system collect essential information, we can just apply second step to load the same class. Moreover, the second step is executed completely by hardware. Java core can resolve all runtime information with the reference pointer in bytecode image. This mechanism only has one-time class loader software execution overhead for each class throughout the life cycle of the JRE. When the same class is referenced for the second time, the Java core will simply transfer the previously created class runtime image from the external DDR SDRAM to the on-chip method area without any runtime class loading software overhead. An overview of the proposed dynamic class loading mechanism and the software class loader analysis will be presented in later sections. Details of dynamic resolution for method invocation and field data accesses will also be described in details. The detail descriptions of these mechanisms are also presented in later sections.

4.1. Overview of Proposed Dynamic Class Loading

Mechanism

When a Java method invokes methods which belong to other class files, the referenced classes have to be loaded into the method area. Traditional JVM’s have software dynamic class loaders to perform the class loading process triggered by method invocations. The dynamic class loaders perform complex dynamic data resolution identified by bytecode operands at first. The dynamic data resolution mechanism gets the class and method information by referencing the const pool data in class files. Then dynamic class loader must check the class method area to make sure whether the referenced class is in the method area or not. If the class is not in the method area, the dynamic class loader loads the class with a series of verification processes as Fig. 1 shows.

It is obvious that the traditional JVM’s have high overhead for dynamic class loading. Even though the class is in the class method area, the JVM still have high overhead to do dynamic data resolution for runtime class information. Hence we propose a hardware-software co-designed dynamic class loading mechanism to deduce the overhead at runtime.

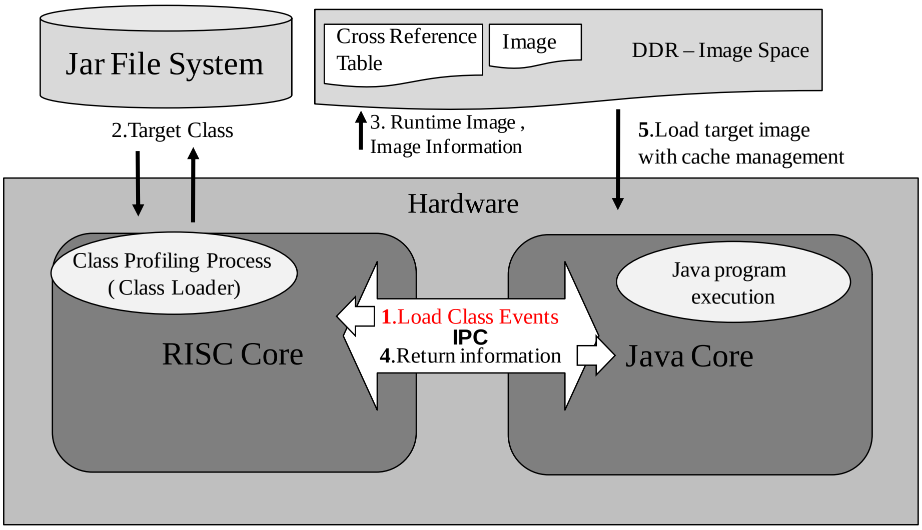


Fig. 15. Runtime dynamic class loading mechanism.

Our proposed hardware-software co-designed dynamic class loading mechanism has two steps. As Fig. 15 shows, when the Java program is executing at runtime and the program wants to do a method invocation from a different class. If the referenced class is invoked first time, Java core generates a “Load Class” event interrupt. Java core asks the RISC core to serve the “Load Class” request through IPC interface.

RISC core will perform the first step of proposed dynamic class loading mechanism in the proposed class loader. The detail analysis of class loader is in next section. This class loader will resolved the class path and class name of target class by querying the cross reference table with information from interrupt service. Then this class loader loads the target class from a Jar file system as the Fig. 15 shown.

This class loader continues to parse the referenced class file and generate the bytecode image and update our proposed cross reference table. Then cross reference table contains new information of referenced class. And the bytecode image also contains the reference pointers which point to these information fields of cross reference table. After the class loader collects essential runtime information of referenced class, RISC core responds to this “Class Load” interrupt service request with method invocation information. Java core enable our cache mechanism after the service returned. Then Java core follows cache mechanism and query cache management table by the returned method invocation. Finally, Java core can load referenced class from cache mechanism and the returned information.

When Java program wants to do method invocation and the referenced class has been loaded before, the Java core can access the cross reference table directly by the reference pointer in the bytecode image. Java core can get the same method invocation information as the information returned in the first step by accessing the cross reference table. If the class has been loaded before, but not in the method area, then Java core must enables cache mechanism to load the referenced class into the method area. The Java core can execute the method code directly if the referenced class is already in method area. The second step of the class loading mechanism only requires very simple hardware logic to implement. And it only requires one data bus transaction to retrieve the method invocation information. The overhead of repeated invocations of the same method is very small during runtime execution.

4.2. RISC-software of the Proposed Dynamic Class Loader

We introduce the software class loader which only executed in first step of proposed dynamic class loading mechanism. Proposed dynamic class loading mechanism construct the cross reference table by this class loader. This cross reference table provides all runtime information not only for method invocation but also other runtime data resolution. As the cross reference table has enough information for runtime execution, the Java core can reduce most overhead from the assistance of this table.

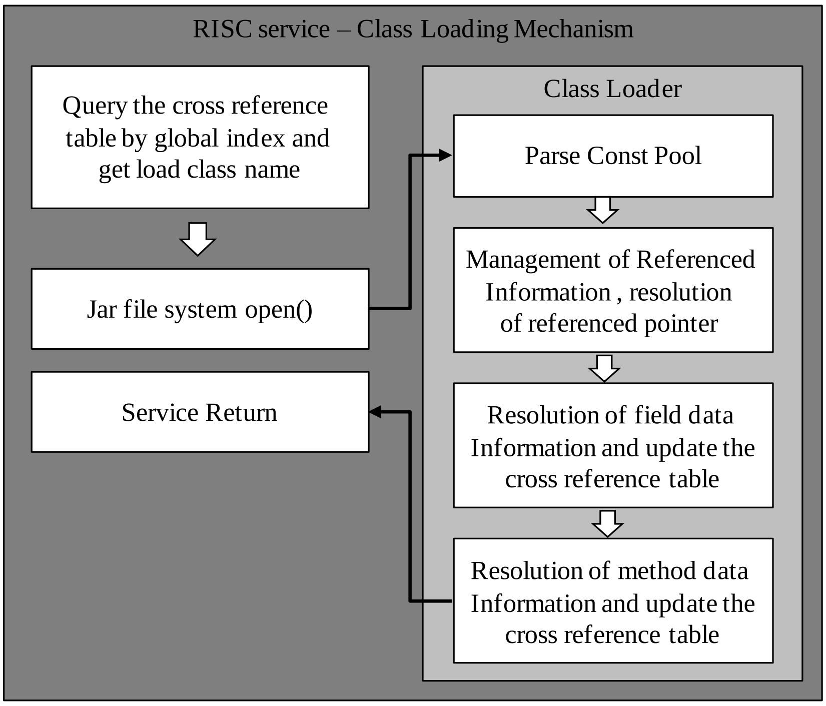


Fig. 16. Overview for the flow of class loader

In this section, we introduce this class loader process and the software overhead analysis. The Fig. 16 describes the general flow of proposed class loader process. The Java core invokes the RISC core service with referenced class global index. The global index of referenced class is allocated at the loading time of the running class. We can make sure the referenced class has registered in advance. The cross reference table already has registered the name of referenced class, but without other details information. Then RISC core get the referenced class name and class path by querying cross reference table with global index.

The function "Jar file system open "will find the referenced class file through the class path in Jar file system at first. Then this process also invokes the class loader process to generate the bytecode image and update the cross reference table. As the Fig. 16 shows, class loader first resolve the const pool of class file and get the information of method reference and field reference in this referenced class. The information will be referenced, when the referenced class is running and references to another class. Then we also get reference pointer for this class to reference other classes. Then class loader continues to resolve method data and field data of this loading class. Finally the cross reference table gets the method information from loading class (referenced class by running class). Then the service returns this information to the Java core and completes the onetime cost class loader process.

We can find pseudo-code and total operations count in Appendix section. The operations count depends on the total registered classes' number, method count of each registered class and field count of each registered class. If the complexity of method or field reference increases, the cost of software also increases. There are still a lot of acceleration issues for software class loader optimization.

4.3. Cross Reference Table with Memory Management

The cross reference table plays an important role of the dynamic class loading form section 4.1. Our proposed Java Runtime Environment registers information such as class or method or field data information to this cross reference table by profiling class file as shown in section 4.2. The cross reference table not only helps runtime resolution to the dynamic class loading mechanism, but also gives helps to method invocation and field data access and memory management. A typical cross reference table with all features is shown in Table 2.

Cross reference table			
Class [x]	**Attributes**	Class name, global index, image address, parent class index	
	Method[i]	Name	Global index & method1 offset
	Field Data[i]	Name	Indirectly pointer

Table 2. Cross Reference Table with Data Field Features

Table 2 shows all information registered on the table fields for one class. When class loader parses a new class, the loader allocates a new global index and creates a table space for new class. Basic table field includes class name, class path name, image start address of external memory, and image size. If the class is load yet, the cross reference table still has registered from other class would reference to it. When the class loader gets a loading class event with global index; the loader can query the table by global index to get the non-load class name. And class loader can find load correct class file from correct Jar file path [8].

The method invocation is the only behavior would invoke the "Load class event." Java core will invoke the interrupt service when the method invocation references to a new class. When the class has been loaded, Java core can directly accesses expected information for usage. And the field data access mechanism also can be a fast hardware mechanism. The sections 4.5 and 4.6 describe how Java core use the information of cross reference table to do method invocation and field data access. Section 4.4 shows current runtime image and how to do the dynamic resolution to their information.

4.4. Runtime Image Format and Resolution

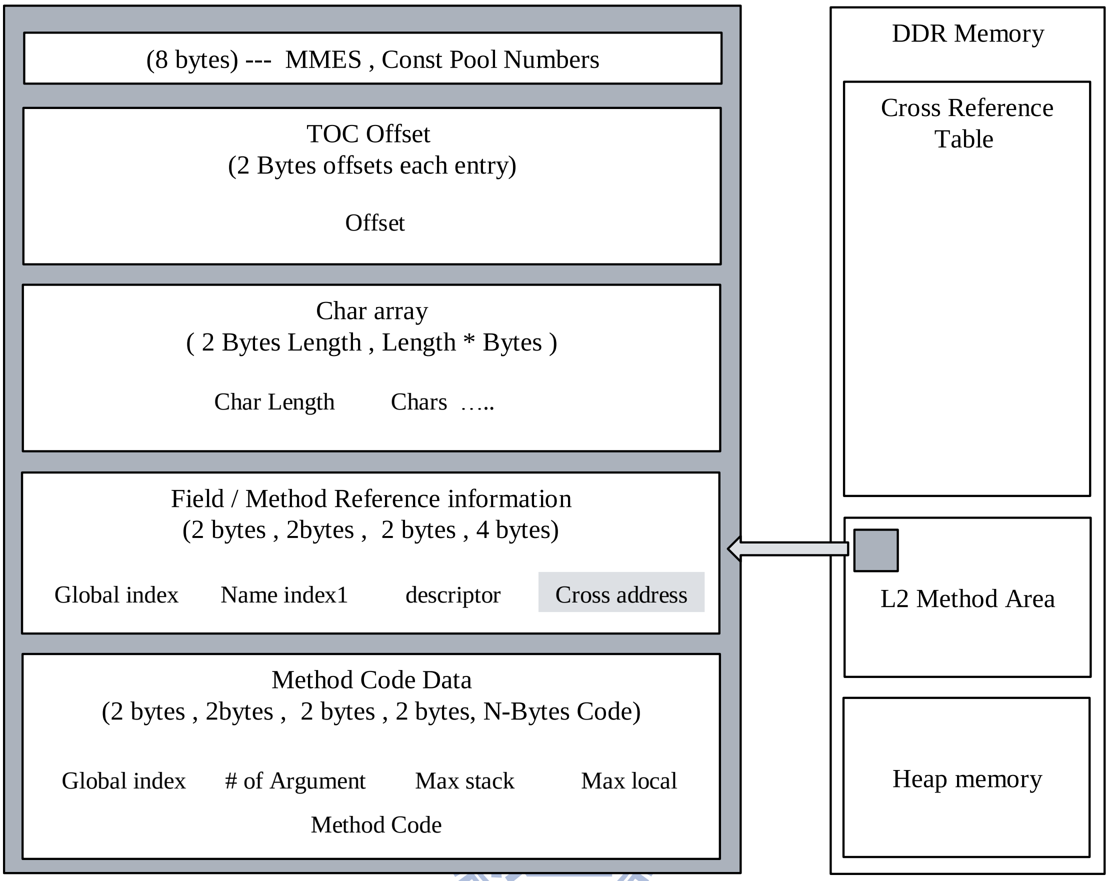


Fig. 17. Image format and memory allocation.

Fig. 17 shows the runtime memory allocation for Java environment and the gray region is the runtime image and its format. This image format can be divided into five parts. The first part contains the image header and numbers of information data and reference information. The second part is TOC offset table, and every entry of this table is two bytes. The TOC offset entries indicate the byte length from image's head to the resolution information data. The Java Core can directly jump to correct address for data resolution by TOC offset entries in this table. The third part is purely char arrays. The fourth part describes all reference resolution information and address links to the cross reference table of section 4.3. For example, method reference information field contains the method owner class's global index. And fields of name index and descriptor both are TOC index for their char arrays. The cross address fields are most important of all information. This field points to the cross reference table field address with method offset for method invocation or physical field data address. The method offset mean the byte length from image head to method bytecode section. The last part is method code data. The bytecode, arguments numbers, local variable numbers, and max stack numbers all have stored in this part for Java core execution.

4.5. Method Invocation Mechanism

This section introduces the method invocation mechanism of the Java core. There are two types of Java method invocations. One is static class method invocation and the other one is dynamic instance method invocation. Static method invocation such as invokestatic references to class method directly. Dynamic method invocation such as invokevirtual and invokespecial depend on their object references. Nevertheless, the proposed Java core executes them with the same procedure. The proposed class loader recognizes these method invocation types and inserts special parameters to indicate the object reference at parsing time. Moreover the invokespecial can be used under three conditions. First, the method is used for instance initialization. Second, the method is a private method. Third, the method is invoked from a “Super” keyword. Current version can support bytecode such as invokestatic, invokespecial, and invokevirtual. The complete method invocation procedure can be described by the state machine in Fig. 18. In a normal class file, a method invocation bytecode is followed by the class and method reference ID’s of the class name string and method name string that are used for computing the target address of the method. With the proposed dynamic resolution mechanism, the invocation bytecode in the class runtime image (translated by the class loader) is followed by a index operand. The operand simplifies the dynamic resolution process.

Fig. 19 illustrates how the operand can be used to access the target method address through indirect references. There is no need to perform any string comparisons as in traditional JVM dynamic resolution process..

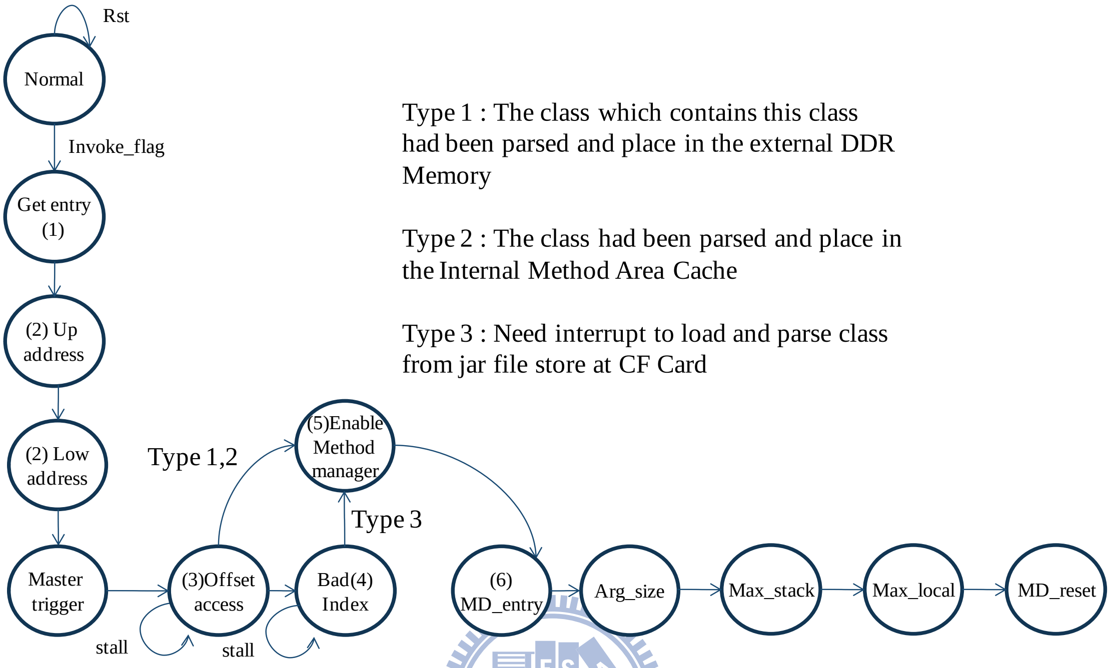


Fig. 18. State machine of method invocation mechanism.

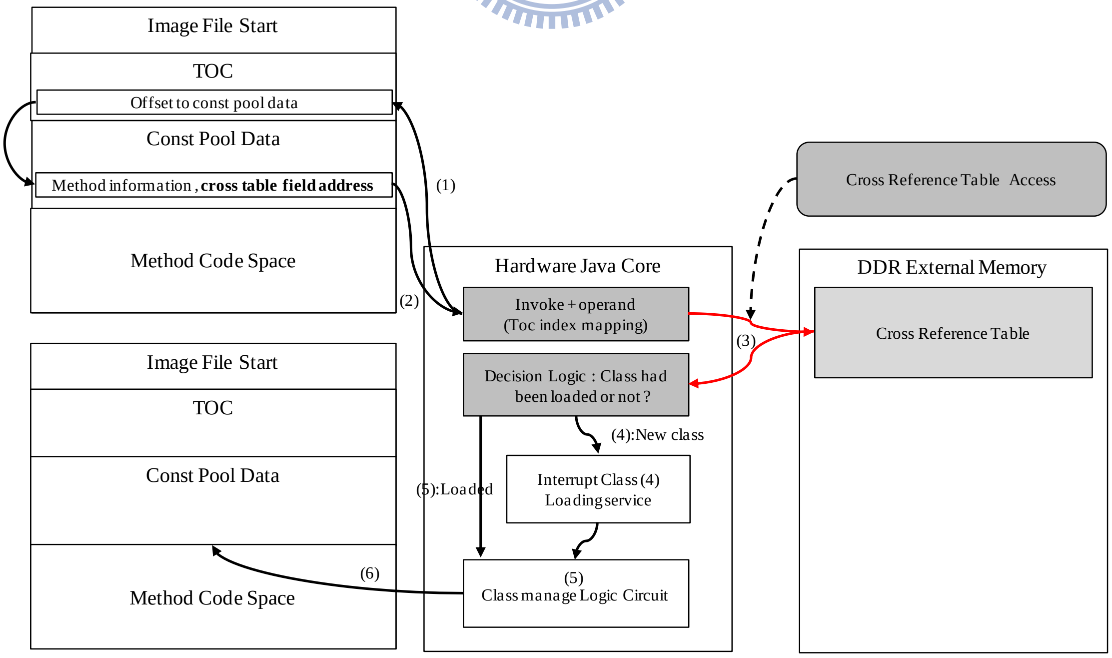
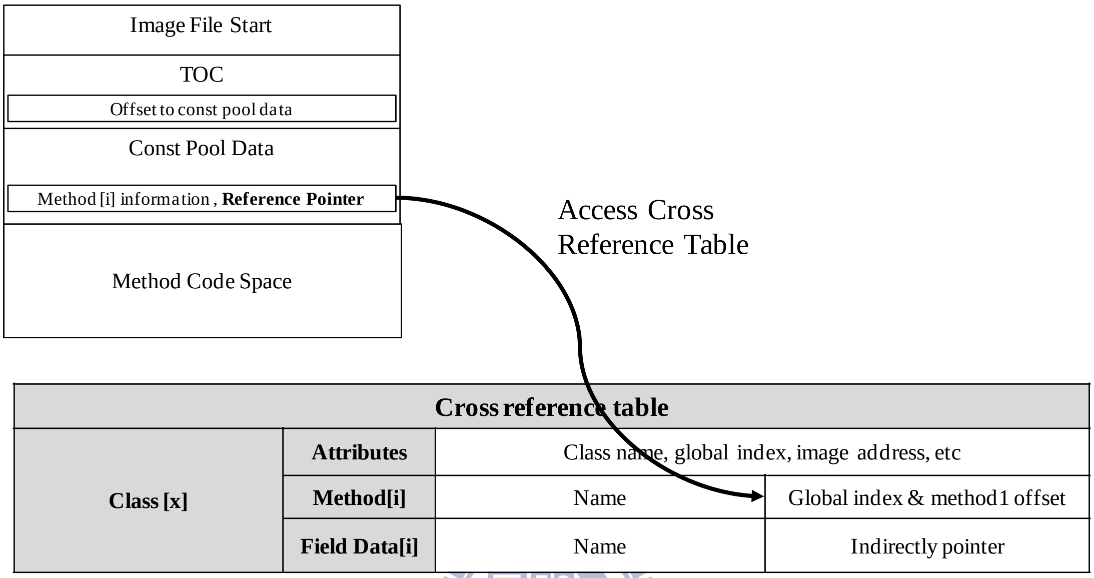


Fig. 19. Method invocation resolution steps.

Fig. 18 describes the state machine of method invocation mechanism. The state machine is controlled by dynamic resolution controller of Java core. The method invocation mechanism can be divided into four stages roughly. Java core can find referenced class and referenced method bytecode information by following this state machine.



Cross reference table			
Class [x]	**Attributes**	Class name, global index, image address, etc	
	Method[i]	Name	Global index & method1 offset
	Field Data[i]	Name	Indirectly pointer

Fig. 20. Reference pointer and method information access

The first stage of method invocation state machine is to find the reference pointer in bytecode image. The first 4 states of state machine are the first stage. Java core will exchange the program counter to obtain the reference pointer from these four stages.

The second stage will access the cross reference table by reference pointer from first stage. And the method invocation state machine performs this behavior at “Offset access” state. As the Fig. 20 shown, the field of cross reference table referenced is a 32-bits data field. The up 16-bits of this field indicate the referenced class’s global index. The low 16-bits of this field indicate the method offset of referenced method. The method offset stands for the method bytecode address offset of its bytecode image. In this stage we can judge whether this referenced class has ever loaded or not by the method offset. If the referenced method has correct method offset, we can realize that referenced class has loaded before. When Java core finds out the referenced class which not loaded before, Java core enters the bad index and waits for the RISC core service as above sections. The Java core enters the third stage until the RISC service finish. If the referenced class has loaded before, the Java core enters third stage directly.

Cache Management Table In Java Core			
Global Index Tag	Image Size	1st Cache Block Start Index	2st External Memory Start Address

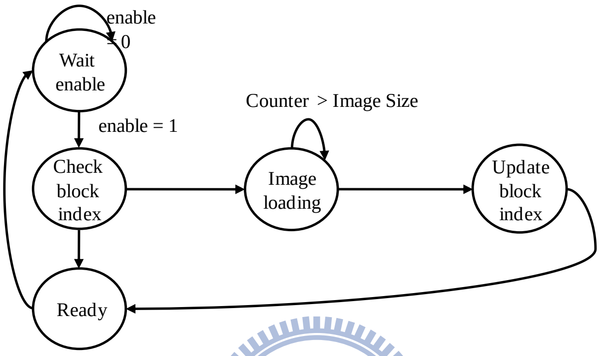


Fig. 21. The cache mechanism of third stage

Java core will get sufficient method information from RISC core service or cross reference table. The third stage enables our bytecode cache mechanism. Java core can query the cache table by the global index which is upper bits of method information field. The Java core follows the cache mechanism and checks method area cache block index at first. If the referenced class has allocated to method area, the cache table will register the start block index. If the referenced class has not allocated to method area, Java core can load its bytecode image with other information such as start address in external memory and the image size.

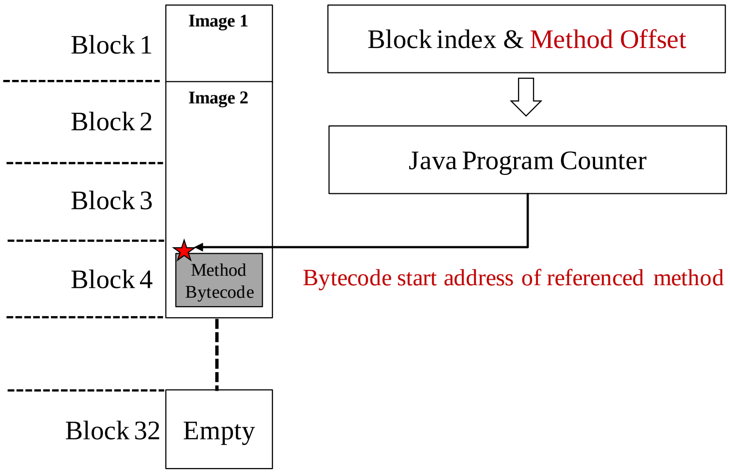


Fig. 22. Adjust Java core to access referenced method bytecode

After the referenced class has loaded into method area, Java core adjust its stack frame and program counter as the Fig. 22 shown. The method offset from cross reference table helps the Java core to update program counter. Then Java core continues to execute new method as usual.

Then this final section continuously describes the stack status during method invocation mechanism. The Fig. 23 describes the stack status of method invocation which the method invoked has two local variables and no argument. The Fig. 24 is also the method invocation with one argument and two local variables. The Fig. 25 and Fig. 26 are then method return which one is “Void return” and another is “Integer return.”

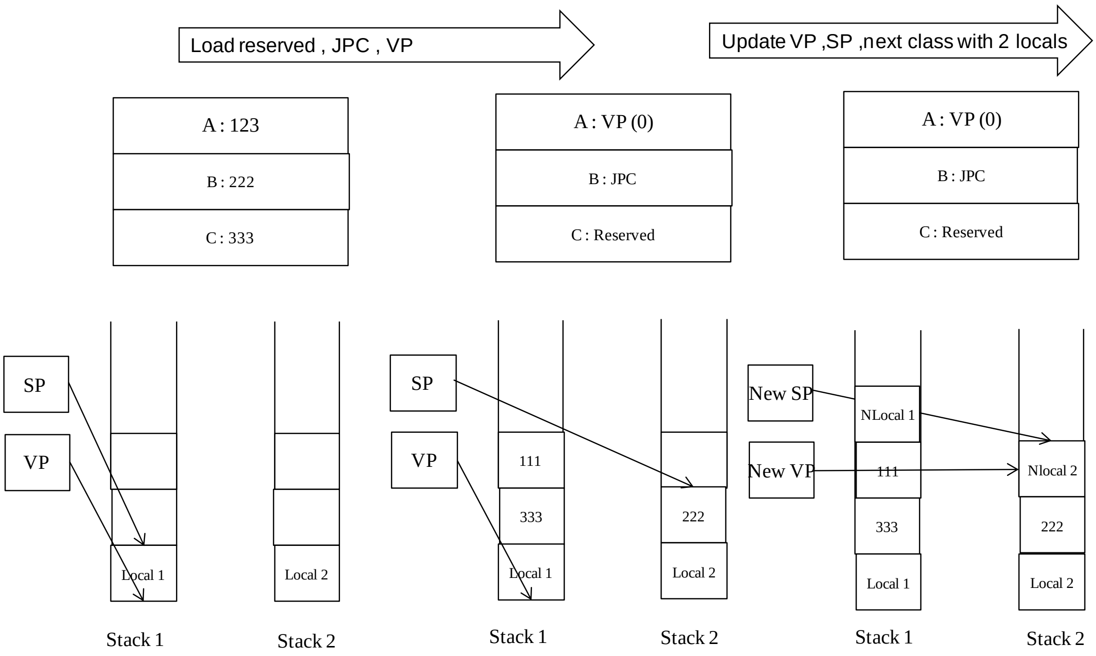


Fig. 23. Method invocation stack variation – no arguments, 2 local variables.

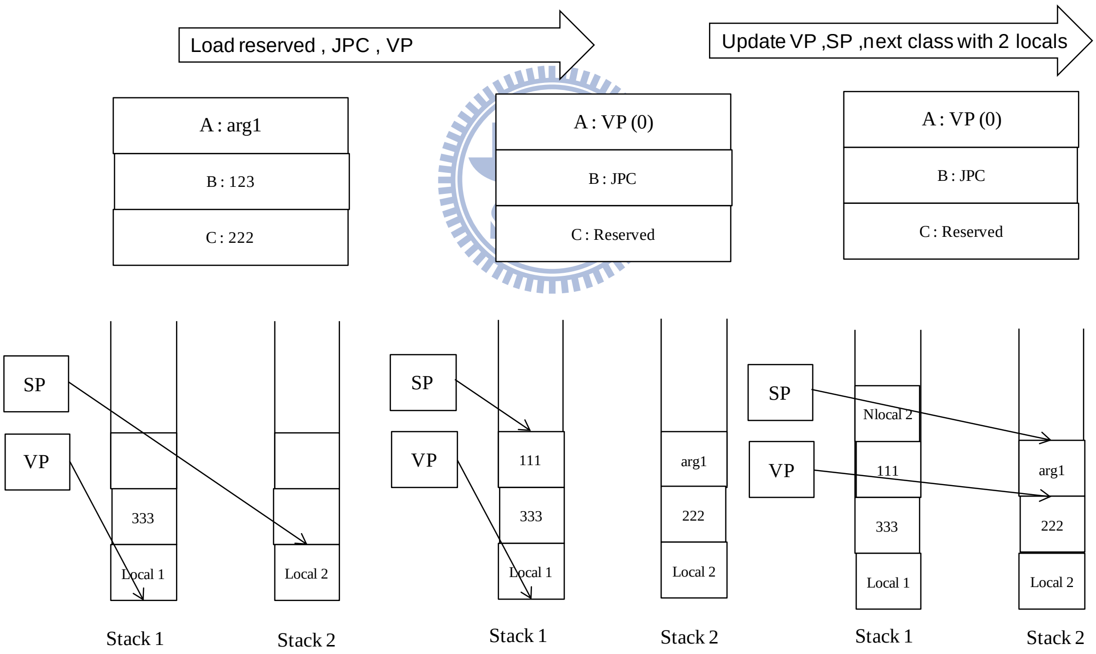


Fig. 24. Method invocation stack variation – 1 argument, 2 locals.

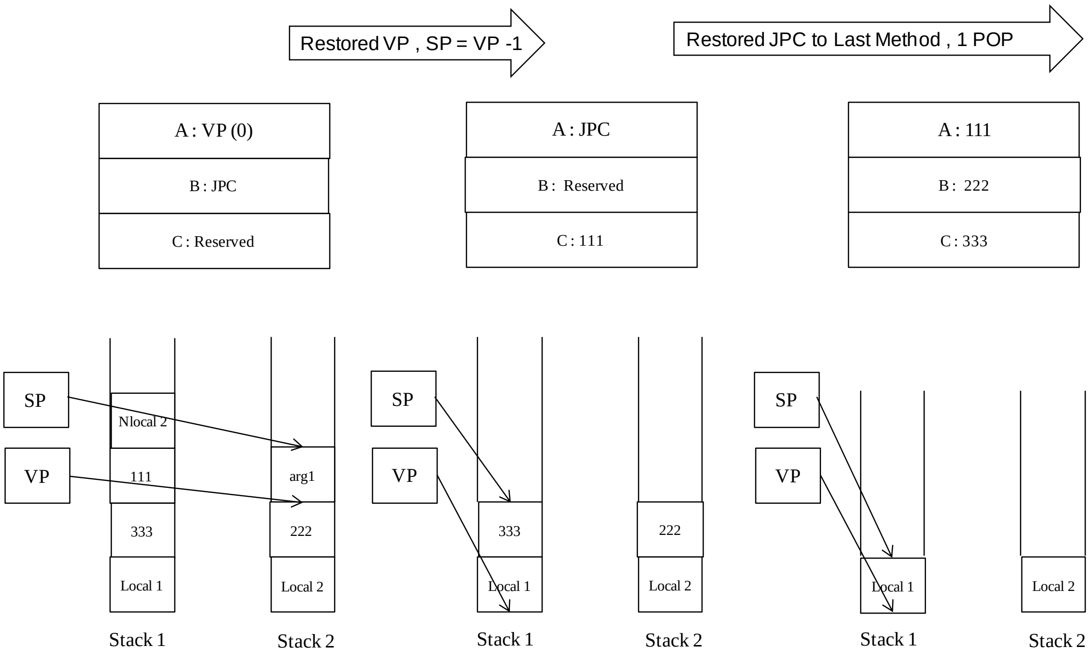


Fig. 25. Method invocation stack variation – Method Return without value (void).

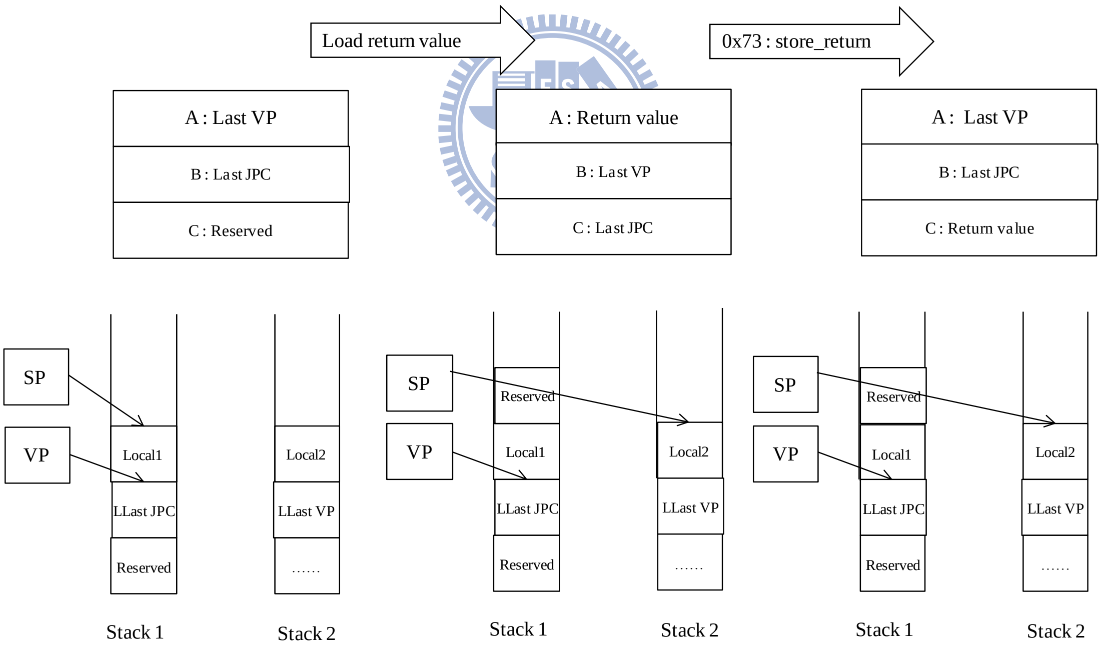


Fig. 26. Method Invocation Stack Variation – Method Return with value “ireturn.”

4.6. Field Data Access Mechanism

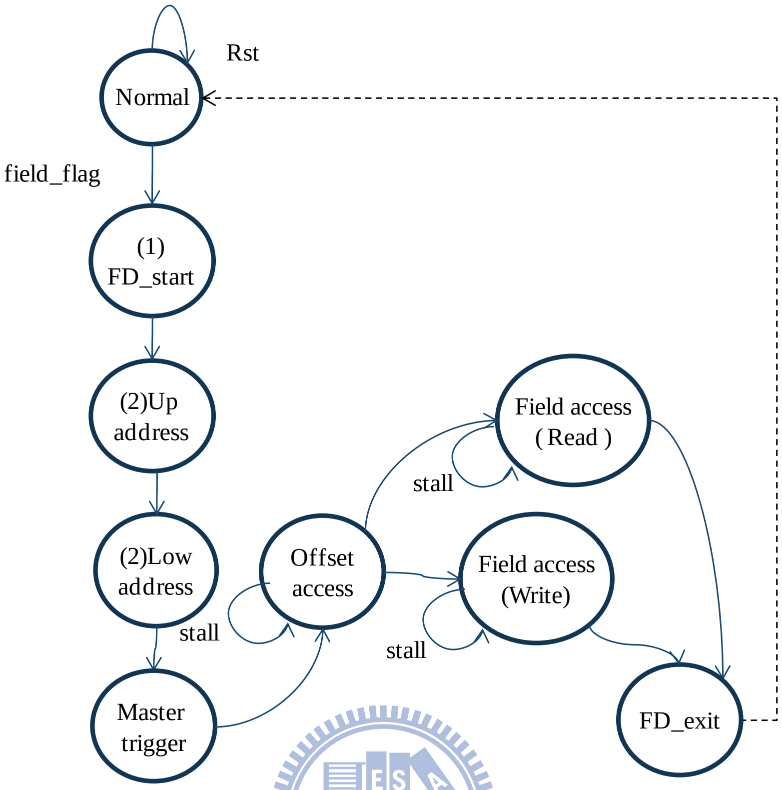


Fig. 27. Dynamic resolution state for field access mechanism.

The field access mechanism is somewhat like the method invocation mechanism. The state machine is also controlled by dynamic resolution controller. The Fig. 27 is the dynamic resolution state transaction flow for field data access, and the first three states are also for the reference pointer mapping. After Java core get the reference pointer, this field of table should contain a pointer of physical memory address for static field data or a variable order index for non-static variable. In the case of non-static variable, the Java core will calculate correct memory address of heap memory by object reference address of heap memory and the order index. The Fig. 28 describes a complete flow for non-static variable access.

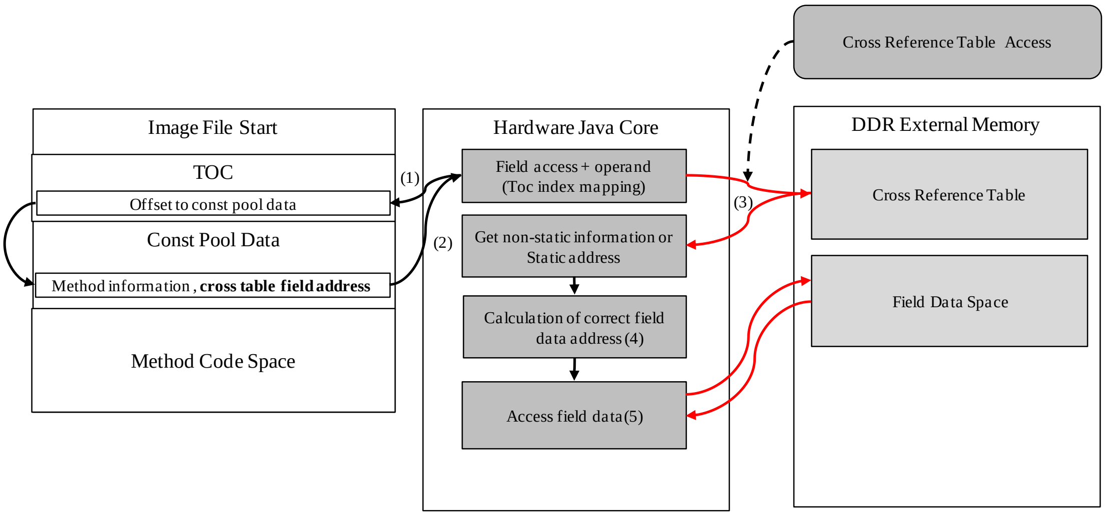


Fig. 28. Step flow for field data access mechanism.

Static field access mechanism has only one thing different form non-static. The cross reference table field has different meaning between static and non-static. In the static case, this field describes a physical memory address for static value which follows the image space, but order index for non-static field access.

Chapter 5. Camera Calibration

We presented basic concepts and some technologies of camera calibration process in Chapter 2. Then we chose calibration algorithm of Roger Tsai which is suitable for our proposed embedded Java platform. We continue to describe the details of our implementation of Roger Tsai's algorithm in this chapter. We verify our implementation with a virtual simulation environment. The virtual simulation environment is set by Blender Tool 2.49. This chapter describes the usage of Blender Tool at first. Then following section describes the image process for abstracting the orientation of landmark points from image. The final section describes the implementation of calibration process.

5.1. Camera Calibration Algorithm

In this section, we will describe the camera calibration algorithm by Tsai [15].

5.1.1. Calibration Algorithm

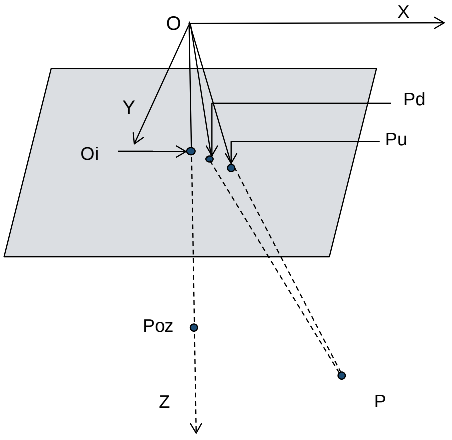


Fig. 29. Perspective Projection Model with Lens Distortion

This section introduces the motivation and basic concept of Roger Tsai's camera calibration algorithm. Main concept of this algorithm is to calibrate the camera by linear equation derived from the physical property of optics. And this algorithm can be divided into two parts. All calculation of this algorithm do not involved non-linear optimization, there is only linear problem need be solved. The one is to compute 3-D orientation position parameters. Another is to compute effective focal length, distortion coefficients, z-position and scale factor which more camera intrinsic parameters. The following sections describe the detail of two parts individually.

5.1.2. Computation of 3-D Orientation Position Parameters

We introduce first step in this section. We observe the Fig. 29 at first. The line segments O_iP_d, $P_{oz}P$, O_iP_u is parallel each other because of the Radial Alignment Constraint (RAC). Then we can find the vector (X_d, Y_d) of P_d and the vector (X, Y) of P are also parallel each other. And we also conclude the outer product of these two vectors is zero. Then we can derive an equation from (1).

$$X_d Y_{ccs} - Y_d X_{ccs} = 0 \tag{5}$$

$$X_d\left(r_4X_w + r_5Y_w + r_6Z_w + T_y\right) - Y_d\left(r_1X_w + r_2Y_w + r_3Z_w + T_x\right) = 0 \tag{6}$$

Finally we derive (5) from (1) and (2). Then we can rearrange the variables to five unknowns because of Z_w to be zero for coplanar calibration system or non-coplanar calibration system. Moreover the X_d, Y_d variables are available from (3). The model of coplanar or non-coplanar are introduced at next section. Then we can conclude two linear systems for different calibration landmark point models. We can get parameters of rotation and translation matrices from these linear systems with known position of landmark points. Equation (7) is for non-coplanar model and Equation (8) is for co-planar model.

$$[Y_{di}x_{wi}\ Y_{di}y_{wi}\ Y_{di}z_{wi}\ Y_{di}\ -X_{di}x_{wi}\ -X_{di}y_{wi}\ -Y_{di}z_{wi}]\begin{bmatrix} T_y^{-1}z_x r_1 \\ T_y^{-1}z_x r_2 \\ T_y^{-1}z_x r_3 \\ T_y^{-1}z_x T_x \\ T_y^{-1}r_4 \\ T_y^{-1}r_5 \\ T_y^{-1}r_6 \end{bmatrix} = X_{di} \quad (7)$$

$$[Y_{di}x_{wi}\ Y_{di}y_{wi}\ \ Y_{di}\ -X_{di}x_{wi}\ -X_{di}y_{wi}]\begin{bmatrix} T_y^{-1}r_1 \\ T_y^{-1}r_2 \\ T_y^{-1}T_x \\ T_y^{-1}r_4 \\ T_y^{-1}r_5 \end{bmatrix} = X_{di} \qquad (8)$$

5.1.3. Computation of Effective Focal Length, Len Distortion, Z-Position

After first stage computation we can estimate three rotation angles parameters, translation of x-axle and y-axle and scale factor in this stage. The second stage computes remain parameters as the title 2.4.1. From relating (3), (4), and (5) we can estimate the z-axle of translation matrix and the effective focal length by (9).

$$d_y * (Y_f - C_y) = f\left({}^{r_4X_w + r_5Y_w + T_y}\Big/_{r_7X_w + r_8Y_w + T_z} \right) \qquad (9)$$

Then we get approximations of focal length and z-axle by ignoring lens distortion from (9). This over-determined system of linear equation can be solved easily from least square method. Finally we compute exact values of focal length, z-axle and lens distortion and use the result of last computation as initial guess for standard optimization of(9). The (10) is also derived from (3), (4), and (5). And we can select some standard schemes such as Steepest descent, Newton method or Powell method for standard optimization. After this calibration algorithm we can get all essential parameters for computing the coordinates of 3-D object.

$$d_yY + d_yY\kappa r^2 = f\left({r_4X_w + r_5Y_w + T_y} \middle/ {r_7X_w + r_8Y_w + T_z} \right) \tag{10}$$

5.1.4. Basic Concepts of Landmark Points Arrangement for Calibration

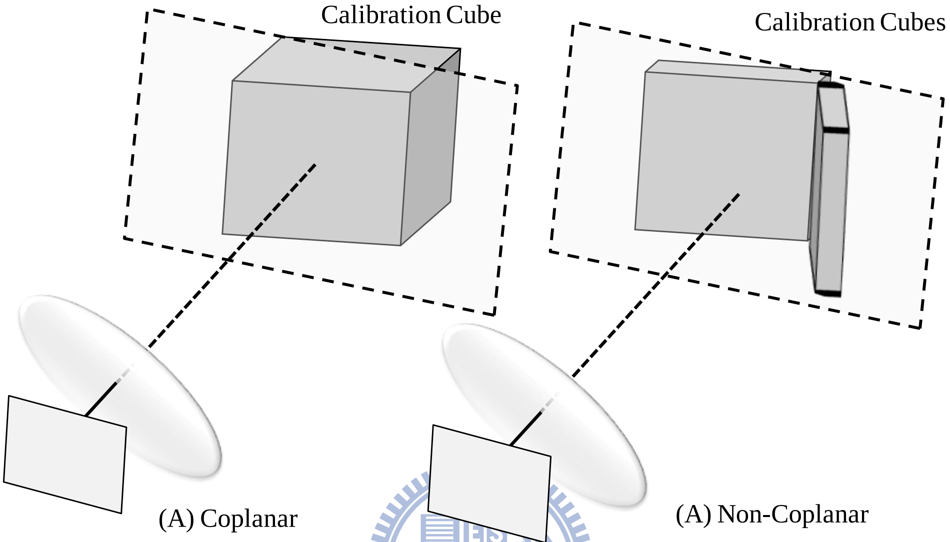


Fig. 30. Calibration Cube for Different Models

The Fig. 30 describes the difference between coplanar and non-coplanar calibration models. The coplanar model means that the landmark points for calibration are all on the same plane. On the other hand the non-coplanar model has landmark points on different plane non-parallel. The landmark points are attached on the face of Cube. The following section of implementation details describes the experiment environment with a simulation platform produced by Blender Tool-2.49.

5.2. Simulation Environment Setup with Blender Tool

I introduce the virtual simulation environment setup by Blender Tool 2.49 in this section. Before the implementation of camera calibration process we need to construct a stable environment for verification at first. The Blender Tool is convenient to construct a virtual 3-D space environment and we use this virtual environment to verify our implementation flow. More learning information can be found at its official web site [25].

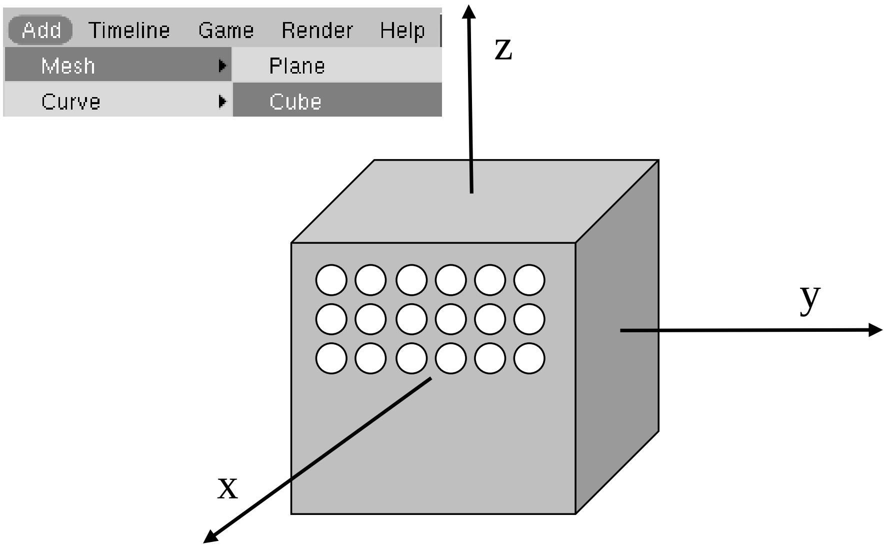
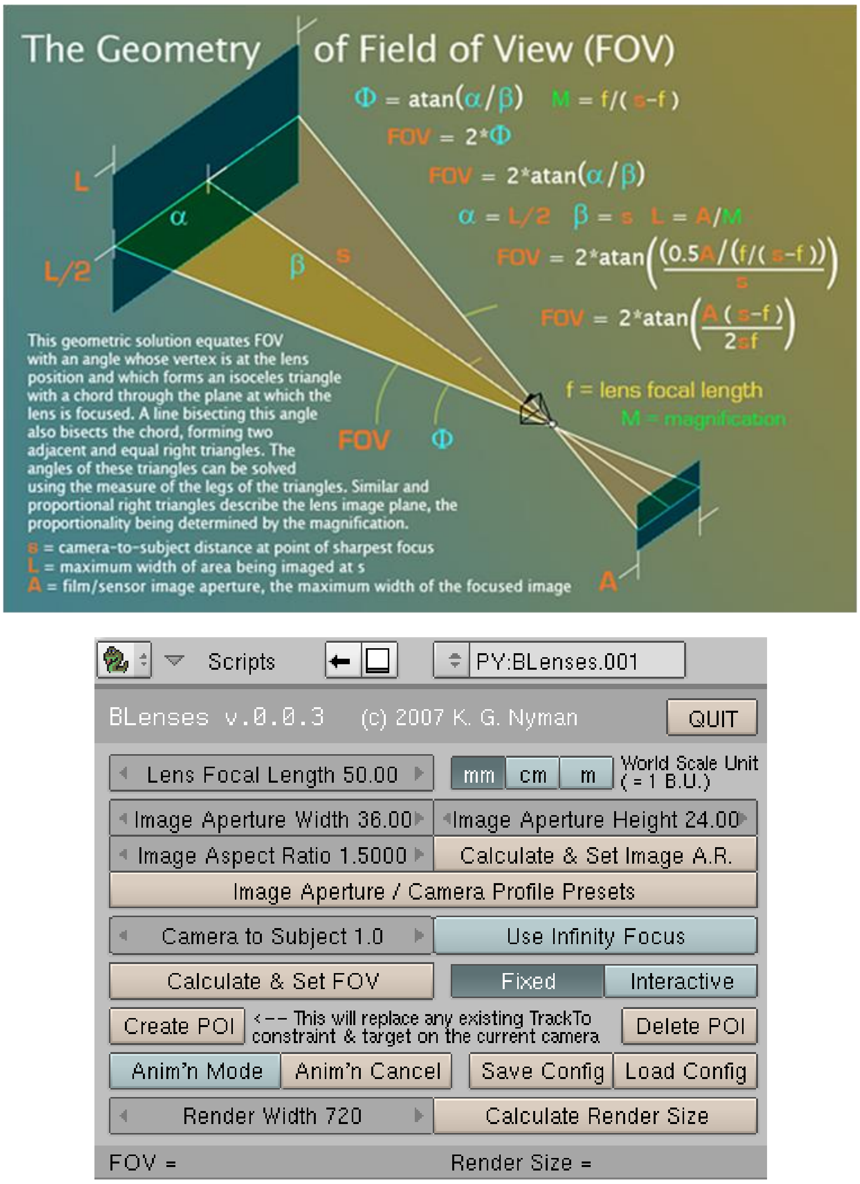


Fig. 31. Virtual Cube with Landmark Points for Calibration

We adopt a static feature points as calibration landmark points such as center of circle or corner of rectangle. Then we use this tool to add a virtual cube attached twenty-five landmark points each face as shown in Fig. 33 . Then we can add virtual camera into the 3-D virtual scene to generate the render image. We can download a template camera script which can be used to adjust camera options manually, and then we can adjust the camera intrinsic parameters for verification. Then we can configure the Field-of-View (FOV) of this camera through the options of script. The Fig. 32 describes the camera field view model and the script options of camera.



This geometric solution equates FOV with an angle whose vertex is at the lens position and which forms an isoceles triangle with a chord through the plane at which the lens is focused. A line bisecting this angle also bisects the chord, forming two adjacent and equal right triangles. The angles of these triangles can be solved using the measure of the legs of the triangles. Similar and proportional right triangles describe the lens image plane, the proportionality being determined by the magnification.

s = camera-to-subject distance at point of sharpest focus
L = maximum width of area being imaged at s
A = film/sensor image aperture, the maximum width of the focused image

Lens Focal Length 50.00
World Scale Unit (= 1 B.U.)
Image Aperture Width 36.00
Image Aperture Height 24.00
Image Aspect Ratio 1.5000
Camera to Subject 1.0
<--- This will replace any existing TrackTo constraint & target on the current camera
Render Width 720
FOV =
Render Size =

Fig. 32. Cameral FOV Model and the Options of Camera

Finally we configure the calibration objects and the camera parameters to a noiseless verification environment. After we arrange appropriate positions of calibration cube and the camera, we can verify the calibration implementation by render image captured from this virtual camera.

5.3. Landmark Points Image Process

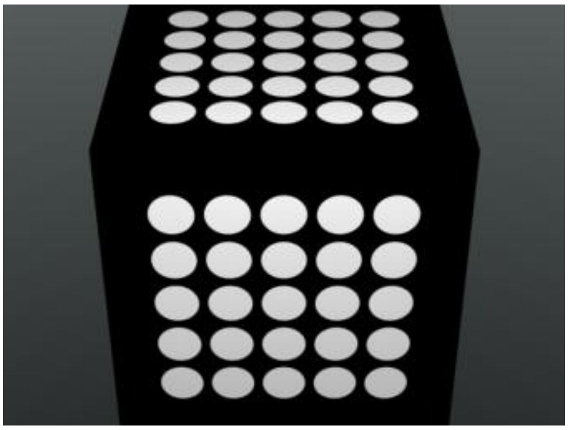


Fig. 33. Render Image of Captured Cube from Camera of Blender Tool

Fig. 33 is the render image captured from the virtual camera. In this thesis, we implement the Roger Tsai's camera calibration algorithm. In order to fit the linear function, we obtain the solutions by substituting the landmark point position of world coordinate system and the ellipse center position of image plane into the linear function. In this section, we describe how to obtain there essential information. We can easily get accurate position of word coordinate system by the Blender tool. Moreover we need get the correct position of ellipse center of landmark points from render image. In this section, I will introduce the basic image processing flow to get the ellipse center position.

The Fig. 34 describes the image processing flow to obtain the 2-D position of landmark points on image plane. The first step translates the color image into gray-level and grouping the histogram. Then we can choose an appropriate value for threshold to delete most other parts. However there are no efficient to reserve helpful parts from a complicated image. Hence I keep the view of calibration as clean as possible. After adopt an appropriate threshold value we can get a binary graph as shown in Fig. 33. Then we apply an edge filter from Laplacian Filter to get the edge pixels and group edge pixels into connected components as shown in Fig. 35.

$$aX^2 + bXY + cY^2 + dX + eY + f = 0 \tag{11}$$

The Fig. 35 shows edge pixels of one component. Then we approximate the ellipse equation (11) by substituting these edge points and normalize with f = 1. Finally, we can obtain the ellipse center from ellipse function approximated. All prepared data are ready for calibration process from this step.

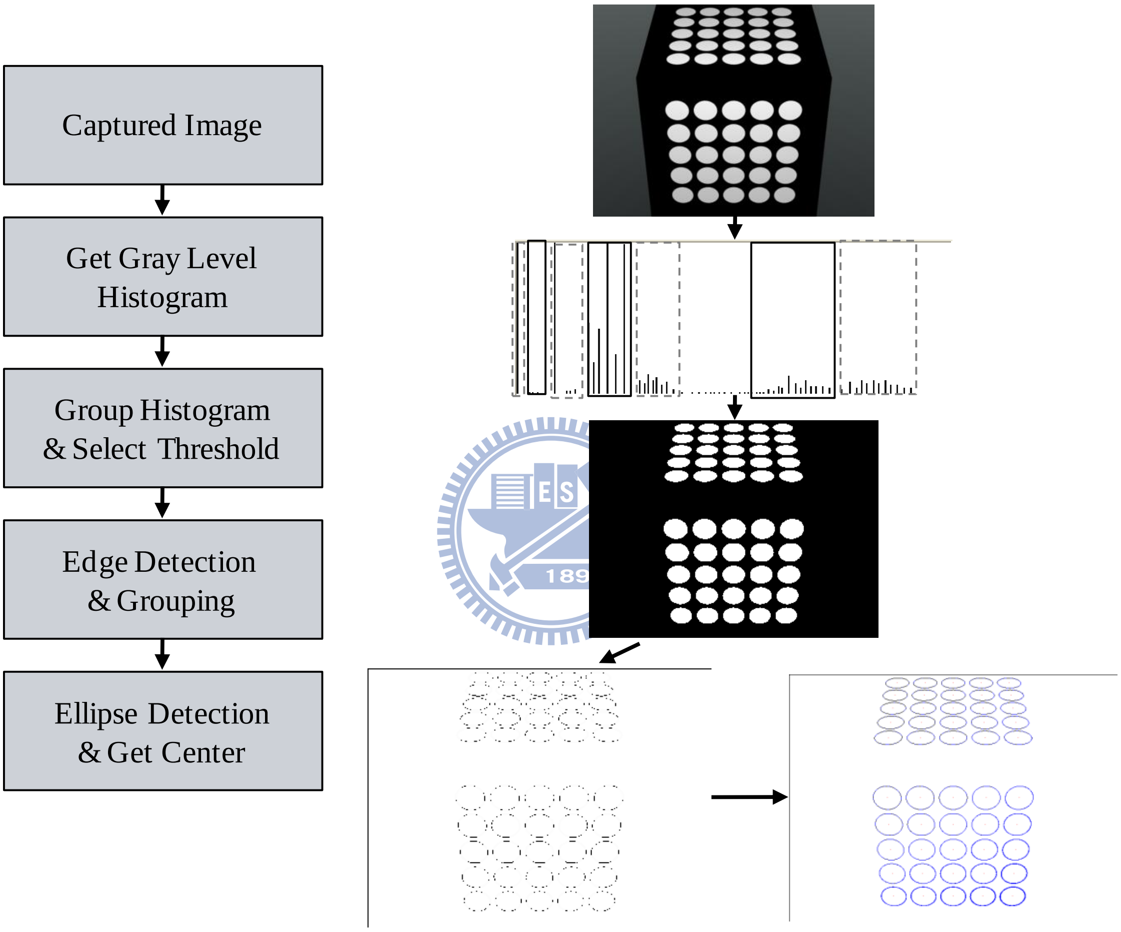


Fig. 34. Image Processing Flow for Landmark Points Center

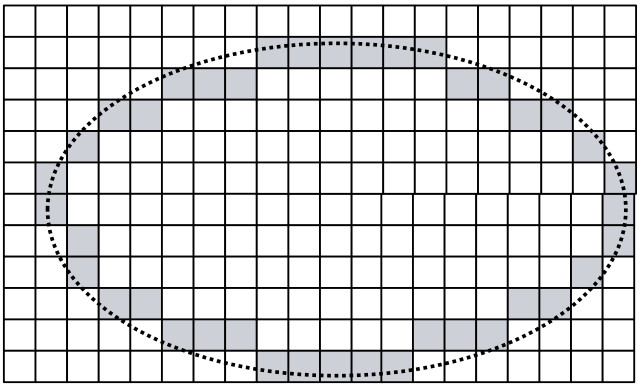


Fig. 35. The Edge Pixels Component for Ellipse Detection

5.4. Implementation Flow for Calibration Process from Roger Tsai's Calibration Algorithm

Proposed implementation of camera calibration process is based on the Roger Tsai's calibration algorithm of section 2.3. The calibration process can divide into two parts. As the section 2.3 shown, first part calculates the 3-D orientation and position. System uses (14) as the calculation function of this step. Then we rearrange the variables of (7) and (8) and obtain (14) from all landmark points. We substitute the known information of world coordinates of landmark points (center of landmark circle) and position of image plane (ellipse center) into this linear over-determined equation (13). The (14) is the partial differentiation of over-determined equation (13).

$$\begin{gathered} V_1 a + V_2 b + V_3 c + V_4 d + V_5 e = V_6 \\ V_1 = Y_{di} X_{wi}\ , V_2 = Y_{di}\ , V_3 = Y_{di}\ , V_4 = -X_{di} X_{wi}\ , V_5 = -X_{di} Y_{wi}\ , V_6 = X_{di} \end{gathered} \tag{12}$$

$$S = \sum (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6)^2 \tag{13}$$

$$\begin{gathered} \frac{\partial S}{\partial a} = 2 * \sum V_1\ (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6) = 0 \\ \frac{\partial S}{\partial b} = 2 * \sum V_2\ (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6) = 0 \\ \frac{\partial S}{\partial c} = 2 * \sum V_3\ (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6) = 0 \\ \frac{\partial S}{\partial d} = 2 * \sum V_4\ (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6) = 0 \\ \frac{\partial S}{\partial e} = 2 * \sum V_5\ (\ V_1 a + V_2 b + V_3 c + V_4 d + V_5 e - V_6) = 0 \end{gathered} \tag{14}$$

We can obtain most calibration parameters of 3-D orientation and position from the solution of above calculation. The rotation matrix, scale factor, x-axle translation and y-axle translation cab be obtained from this step. And the sign of parameters decided by set one of this parameter to be '1' positive, and the other parameters decided by the mutually orthogonal of any two rows or columns. Certainly we should be sure to the consistency of coordinate direction.

The second step computes focal length, distortion coefficients and z-axle position. We assume the distortion to be zero to get the initial value for next step at first. And we can derive a linear function by below functions with distortion is zero. Then the summery linear function is also an over-determined system and we estimate effective focal length by least square method.

$$S = \sum(y_i f - Y_{di} T_z - w_i Y_{di})^2$$

$$\frac{\Delta S}{\Delta f} = \sum 2y_i (y_i f - Y_{di} T_z - w_i Y_{di})$$

$$\frac{\Delta S}{\Delta T_z} = \sum 2Y_{di} (y_i f - Y_{di} T_z - w_i Y_{di}) \tag{15}$$

Then we compute more accurate focal length, distortion coefficients and z-axle position parameters in this section. The equation (15) is the summery and partial differentiation of (10). We continue use (15) and assume f, T_z and K as unknowns. And estimate the solution by standard optimization process as Powell’s Method with initial guess from last round solutions. Finally we can obtain the intrinsic parameters of camera and extrinsic parameters of 3-D orientation rotation matrix and translation matrix parameter.

Chapter 6. Experimental Results

6.1. Performance Analysis of Java Core Platform

6.1.1. Development Platform and Tools

We have implemented the proposed Java platform on the Xilinx Virtex5 ML507 development board. The FPGA XC5VFX70T contains hardcore PowerPC 440, 44800 slices, 128 DSP8E functional units and 5328 Kbits BRAM with 256 MB DDR2 memory. We use Xilinx Embedded Development Kit 10.1(EDK) as the development tool and Xilinx® Synthesis Technology (XST) as the FPGA synthesis tool. The design suite also provides full system simulation verification for EDK development platform and ISE.

We create an implementation platform from Base System Builder (BSP) wizard of Xilinx Platform Studio (EDK XPS) as shown in Fig. 36. We also add Java core IP into this implementation platform.

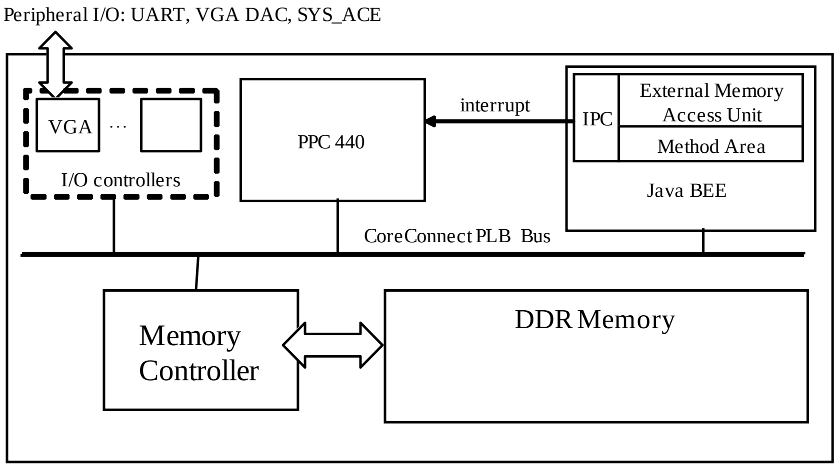


Fig. 36. **Architecture diagram of implementation platform.**

Fig. 37 describes the runtime emulation platform. The system bus (PLB) clock rate is running at 100 MHz. Table 1 shows the resource utilization of the FPGA device and the estimated working clock frequency. Although the estimated working frequency is a little bit lower than the actual bus frequency, the system seems quite stable when running benchmark programs. It is possible to lower the system bus frequency on ML-507 if necessary.

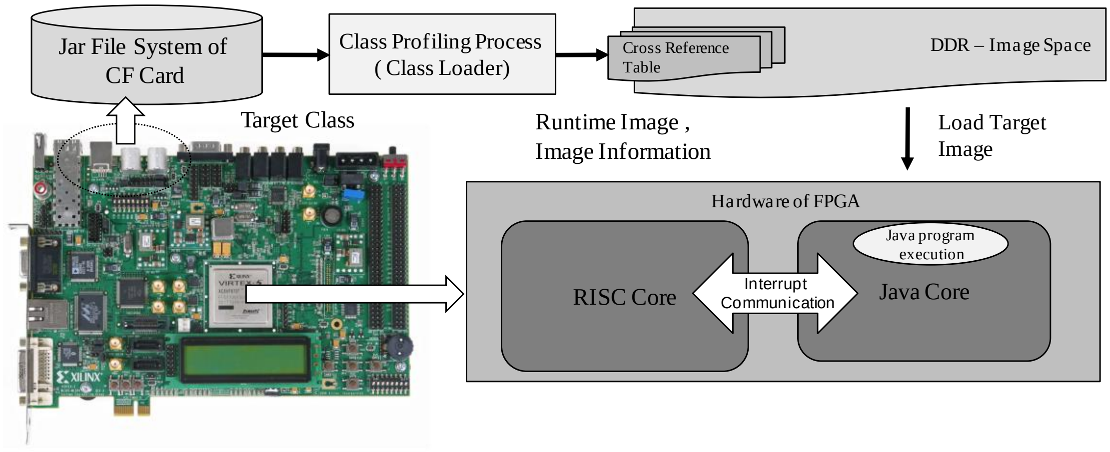


Fig. 37. Emulation platform of the proposed Java system.

Selected Device : 5vfx70tff1136-1		
Number of Slices:	9252(3528) out of 44800	20(7)%
Number of Slice 6 input LUTs:	8755(4044) out of 44800	19(9)%
Number used as logic:	8390(4404)	
Number of IOs:	212(0)	
Number of bonded IOBs:	120 (0) out of 640	18%
Number of Block RAM/FIFO	35 (19) out of 148	23(12)%
Number using Block RAM only	35(19)	
Number of PPC440:	1 out of 1	100%
Minimum period: 10.681ns	Maximum Frequency: 93.624MHz The value of parentheses is only Java execution engine	

Table 3. Synthesis report of the design on an XC5VF70T device.

6.1.2. Benchmark of Java Core

This section compares the performance between the CVM running on the RISC core alone and our proposed Java core. We use the benchmark of Embedded Caffeine Mark (ECM) 3.0. The benchmark result is shown in Table 4. The CVM Java VM interpreter is running on the PowerPC 440 CPU under MontaVista Linux. The CVM platform has 32KB instruction cache and 32 KB data cache in the PowerPC core.

However, current Java core does not have any data cache, which affects the performance significantly for the proposed Java processor. However, the performance of the proposed Java core is still much faster than that of CVM as shown in Table 4. And the score of performance is defined by Caffeine Mark, which is calculated based on the execution iterations of certain code in 1 ms and the scale factor of each benchmark. In short, the higher the score, the better the performance.

Items **Bus100MHZ** **Test Programs**	**Proposed JRE** **@100 MHz without data cache**			**CVM @PowerPC** **100MHZ with data cache**	
	Once [#K cycle]	Once [ms]	Score	Score	Improvement
SieveAtom			320	82	3.82 faster
Sieve(Load class from DDR)	868098	8.68			
Sieve(non-load)	866073	8.66			
Loop Atom			262	56	4.68faster
Loop (Load class from DDR)	1038833	10.38			
Loop (non-load)	1037778	10.37			
MethodAtom			228	79	2.88 faster
Method(Load class from DDR)	1343344	13.43			
Method(non-load)	1340759	13.40			
Logic			711	74	9.60 faster
Logic(Load class from DDR)	545558	5.45			
Logic(non-load)	535828	5.35			

Table 4. Benchmark between CVM and the proposed platform

As the Table 4 shows, we can notice that the performances for different benchmarks are quite different. For example, the performance of executing Method on the proposed Java core is only 2.88 (about 3) times faster than that on the CVM. However the performance of Logic is 9.60 times faster. There is more details analysis for each benchmark later.

Fig. 388, Fig. 39, Fig. 40, and Fig. 41 show the distributions of bytecode instructions of these benchmark programs. Note that in the proposed design (described in chapter 4), both method invocation and field data access need at least two external memory accesses. Access to an array in heap memory takes one external memory access. However current Java core does not have any data cache, external memory costs a lot of cycles for PLB bus transaction.

Sieve compute prime number below “512”. Logic changes the Boolean logic state 2400 times. Method calls recursive method invocation for a total of 10060 times. Loop counts the Fibonacci sequence below 64 for a total of 4036 iterations. As Fig. 39 shows, Logic is composed mostly of simple instructions (90%). The proposed Java core can execute two simple instructions per clock cycle. Therefore, the performance on Logic is much faster than others.

On the other hand, although the percentage of simple instructions of Loop is much more than that of Sieve, the performance on Loop is less than that on Sieve. The main reason for this issue is the number of external memory accesses. External memory accesses on PLB bus take about 15~20 cycles for one data transaction on average. The Sieve benchmark has about 17000 (18%) instructions for external memory data accesses. Moreover, the field accesses and method reference instructions requires two external memory accesses for each instruction. The Loop benchmark has about 16500 (16%) array access and 8317(8%) field access. We compare the percentages of external memory accesses between Sieve and Loop, and Loop has much more portion of programs to do external memory accesses. Finally, Method benchmark has very high percentages to do method reference and external memory data accesses and complex bytecode “ireturn.” Hence Method need more amount of time to do bus transaction and the performance improvement is less than others benchmark cases.

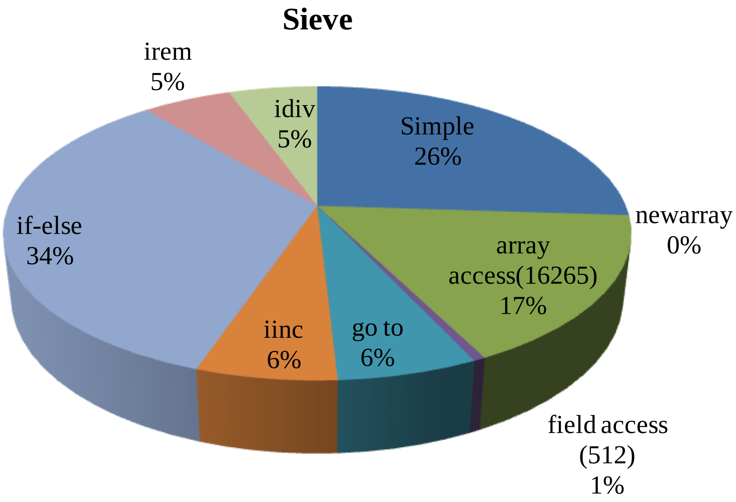


Fig. 38. **Sieve benchmark analysis**

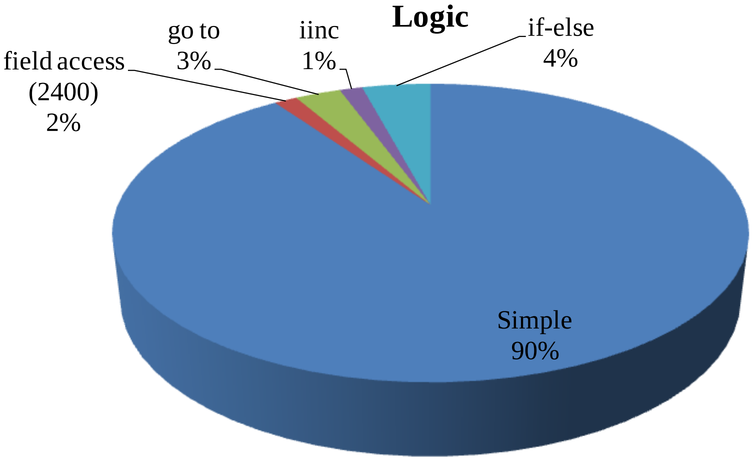


Fig. 39. **Logic benchmark analysis**

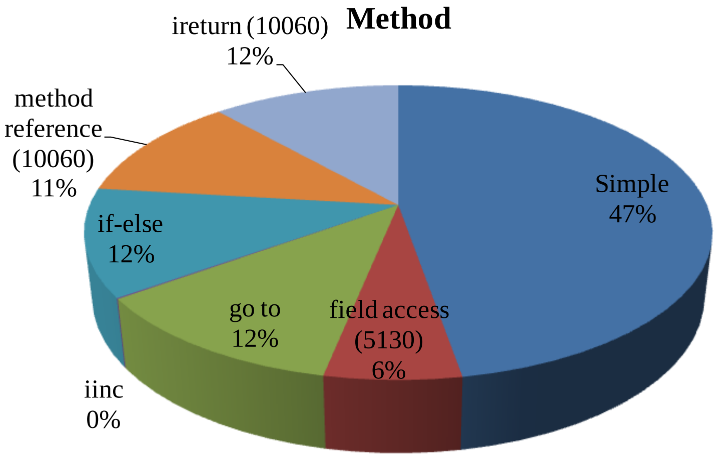


Fig. 40. **Method benchmark analysis**

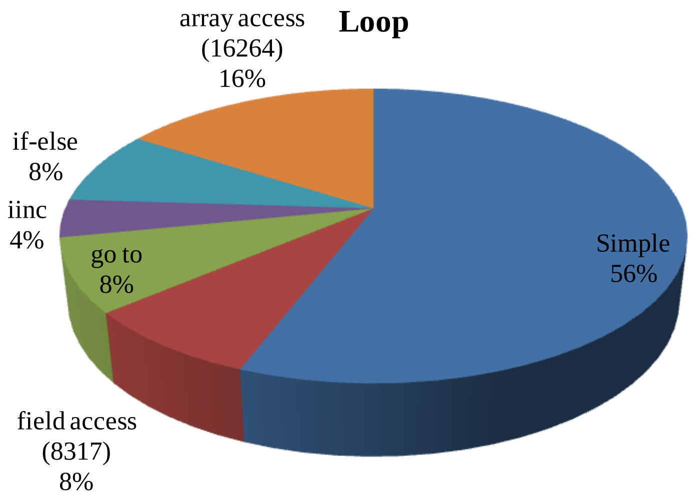


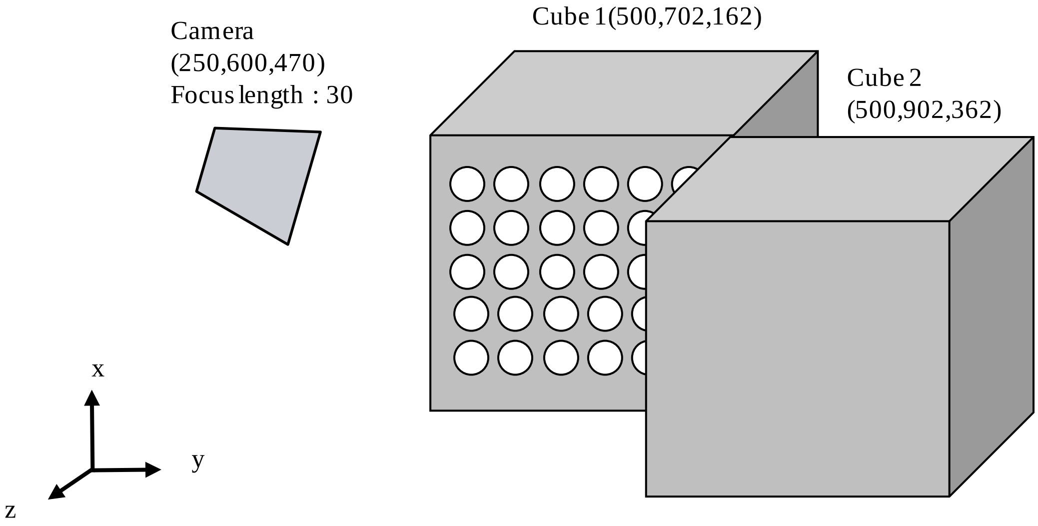
Fig. 41. Loop benchmark analysis

6.2. Experimental Results on Camera Calibration

We use a simulated virtual 3-D world created by Blender Tool 2.49 to verify the implementation of camera calibration process. Blender is used to create calibration cube, landmark points, and virtual camera in the virtual 3-D space. We can control accurate 3-D position of calibration objects and camera through the tool. Blender also provides more detail options such as real light render, text control, camera intrinsic parameters, etc. We can compare the result calculated by our calibration process from render image with the parameter configured by Blender for verification.

Fig. 42 describes the simulation model constructed by blender. There are two calibration cubes with landmark points on different faces in this virtual model. Fig. 42 also describes 3-D position of objects and the position and focal length of camera. Then the calibration process do the image process from render image captured by virtual camera at first and obtain the orientation information of image plane as shown in Fig. 43.

After system has the orientation information of image plane and the known 3-D position of landmark points, the camera calibration process calculates all parameters as described in chapter 5. Table 5 is the comparison between results of camera calibration implementation and configuration setting from simulation environment.



Coordinate(mm) Objects	Camera Coordinate System	Proposed World Coordinate System
Camera center (Rot x -35 , Rot y -55, Rot z 70)	(0,0,0)	(250,600,470)
Proposed world coordinate center	(110.7,-740,-289)	(0,0,0)

Fig. 42.　Virtual configurations of experiment environment.

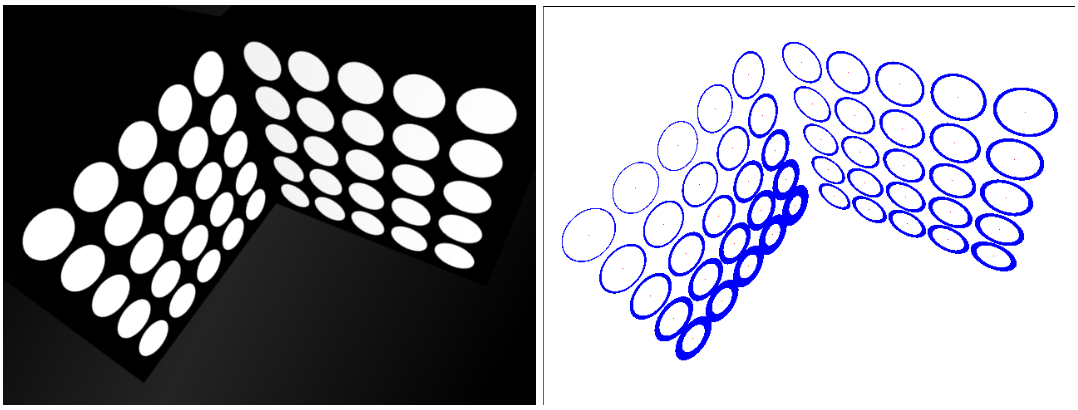


Fig. 43.　Render image and result of image process.

Calibration Parameters	Configure Value	Calibration Value
Focal length	30	27.4
Scale factor	--	1.3
Distortion	0	0
Rotation x	-35	-35.3
Rotation y	-55	-54.9
Rotation z	70	70.3
Translation x	757.43	758.19
Translation y	120.75	127.54
Translation z	231.91	233.69

Table 5. Comparison between configuration and calibration parameters.

Chapter 7. Conclusions and Discussions

In this thesis, we have presented the design of a dynamic class loading mechanism for Java processors using hardware-software co-design approach. For each class loading operation, the proposed system divides it into two different steps. The first step includes operations like locating the Java class files, parsing the class files, generating a runtime class image and cross-reference tables. This step is a one-time only process for the entire life cycle of the Java system and it greatly reduces the complexity of dynamic resolution and loading of a class image into method area. The first step in dynamic class loading is implemented in software and is executed by the RISC core.

The second step of dynamic class loading involves dynamic resolution of symbols in the class (symbols are turned into unique IDs during the first steps) and management of class images in the method area. The second step is implemented by hardwired logic for performance reason. The second step will be executed over-and-over every time a class is referenced in the Java application program.

There are some operations that are not completely supported in current design of dynamic class loading mechanism such as complex inheritance and interface invocation. Although the class parser only parses each class once during the life cycle of a Java system, the performance of the class parser is also important to reduce the overhead of dynamic class loading for Java processors.

Finally, for long-term goals, there are many issues such as sophisticated garbage collection, exception handling, and multi-threading for complete support of JVM for embedded systems, etc.

Another topics investigated in this thesis is the camera calibration process for 3-D

MMI. We have implemented a camera calibration algorithm and verified the implementation using simulated environment. The simulation result shows that the camera parameters are estimated quite accurately. For future work, we will construct a real environment for testing the camera calibration process. Then we can apply the triangulation method to capture real 3-D position of any object. Eventually, these technologies can be integrated together to construct a smart embedded Java platform with 3-D human interaction interface.

Appendix: Pseudo-code of Class Loader on RISC-side

```
total_file_of_FAT_entries          ; Total number of files of FAT file system
totol_registered_class             ; Total number of registered class
nConstPool                         ; Total number of const pool items
attribute_count                    ; Total number of attribute in a class
mehthod_numberofclass[x]           ; Number of methods of the x th class
field_numberofclass[x]             ; Number of fields of the x th class
nByteCode                          ; Number of bytecodes in current method
string_operation                   ; String comparison
directly_access_operation          ; String comparison

Open Class File of Jar file system (Jar file name, Class path/class name)
{
    Open Jar file with Xilinx Fat16 filesystem

    while (1)
    {
        if (Correct file header) comment: Check the header of Jar file system ; 1 string_operation
        {
            Resolution of class files information of Jar format
            if (Temp class name equal class path/class name)    ; 1 string_operation
            {

              Initialization of target class name of table ; 20 directly_access_operations
              For (i = 0; i < totol_registerd_class; i++)
               {
                   if (class name = cross_reference_table[i].class_name) ;1 string_operation
                   {
                       Start the class loader process for a registered class
                       ; 3 directly_access_operations
                   }
               }
```

```
                jf (flag = 0) comment:    new class register at first
                {
                      Start the class loader process for a non-registered class
                      ; 6 directly_access_operations
                }
            }
        }
    }
}
total_file_of_FAT_entries*(1/2)*(26+ totol_registerd_class) directly_access_operations
total_file_of_FAT_entries*(1/2)*(2+ totol_registerd_class) string_operations

ClassLoader (image baseaddress, &classfile , i(Class global index ) )
{
      Resolution of class header
      Resolution the number of referenced method and field
      ;3* nConstPool directly_access_operations
      Retreat information for cross reference table ; 2 directly_access_operations
      for (i = 0; i < Method_ref_count ; i++)
      {
            Resolution of referenced method information ; 10 directly_access_operations

            for ( j=0 ; j < totol_registerd_class ; j++)
              {
                  if (referenced Class not registered )
                  {
                       Table initialization process
                       Register method field both in this class and referenced class
                       ; 26 directly_access_operations
                       ; 1 string_operations
                  }
                  else {
                        if ( referenced class is this class)
                        {
                               Register the method information and get reference pointer
                               ; 14 directly_access_operations
                        }
                        Else
```

```
            {
                1.  The Referenced Class has registered the method information
                    ;Mehthod_numberofclass[x] + 2 directly_access_operations
                    ; Mehthod_numberofclass[x]/2 string_operations
                2.  The Referenced Class also has not register this method information
                    ;Mehthod_numberofclass[x] + 7 directly_access_operations
                    ; Mehthod_numberofclass[x]/2 string_operations
                Update image reference pointer
                ; 11directly_access_operations
            }
        }
    }
 }
 Retreat field information for cross reference table
for (i = 0; i < field_ref_count ; i++)
 {
     Resolution of referenced field information
     ; 10 directly_access_operations
     for ( j=0 ; j < totol_registerd_class ; j++)
       {
          if (referenced Class not registered )
          {
             Table initialization process
             Register method field both in this class and referenced class
             ; 26 directly_access_operations
             ; 1 string_operations
          }
          else {
             if ( referenced class is this class)
             {
                 Register the field information and get reference pointer
                 ; 14 directly_access_operations
             }
             Else
             {
                 1.  The Referenced Class has registered the field information
                     Mehthod_numberofclass[x]*2 operations
                     ;Mehthod_numberofclass[x] + 2 directly_access_operations
```

```
                        ; Mehthod_numberofclass[x]/2 string_operations
                    2.  The Referenced Class also has not register this field information
                        ;Mehthod_numberofclass[x] + 7 directly_access_operations
                        ; Mehthod_numberofclass[x]/2 string_operations
                    Update image reference pointer
                    ; 11directly_access_operations
                }
            }
        }
}
Resolution parent class index information
;2 directly_access_operations
Get the global index of parent index
; totol_registerd_class/2 string_operations
;1 directly_access_operations
Resolution of field data of this class
  ; 1 directly_access_operations
for (i = 0 ;i < total field data; i++)
{
    Parse field data information (Access flag) and resolve the field data name
    Register the field information to the cross reference table of this class
    ; 5 directly_access_operations
     Update field information of other class who will access this field data
     ; totol_registerd_class*Field_numberofclass[x]/2 string_operations
     ; 2 directly_access_operations
}
Resolution of method this class
; 1 directly_access_operations
for (i = 0 ; i < method count of this class ; i++ )
{
    Resolution of method access flag
    Resolution of method information
    ; 6 directly_access_operations
    Check the method whether has been registered or not 4 operations
    ; Mehthod_numberofclass[x]/2 string_operations
    ; 2 directly_access_operations
    if ( Method has not been registered )
    {
```

```
            Register the method field – 5 operations
            ; 4 directly_access_operations
        }
    Resolution of descriptor
    ; 4 directly_access_operations
    Descriptor checking for argument numbers
    ; 2 directly_access_operations
    ; 5 string_operations
    "Main" function checking – 5 operations
    ; 1 string_operations
    ; 1 directly_access_operations
    for (j = 0 ; j <attribute_count ; j++)
    {
        Resolution of attribute name and size
        ; 4 directly_access_operations
        if (attribute name is "Code")
        {
            ; 1 string_operations
            Resolution of Max_Local ,Max_Stack
            ; 5 directly_access_operations
            Update current method field offset    2*method count operations
            ; method count operations directly_access_operations
            ; method count operations/2 string_operations
            Update method field offset of other class
            2*totol_registerd_class*Mehthod_numberofclass[x] operations
            ; ( totol_registerd_class/2)*(1 +Mehthod_numberofclass[x]) directly_access_operations
            ;( totol_registerd_class/2)*(Mehthod_numberofclass[x]/2) string_operations
            Update the operand index of method code – nByteCode*4 operations
            ; nByteCode*5 directly_access_operations
        }
       else
       {
            Skip other attribute
       }
    }
    }
    Return Image Size;
}
```

Reference

[1] 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.

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

[3] 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.

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

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

[6] Sun Microsystems, *Connected, Limited Device Configuration Specification*, ver. 1.0a, Sun Microsystems White Paper, May 2000.

[7] Sun Microsystems, the Java Community Process Program, JSR 36: *Connected Device Configuration*, ver. 1.0b, Dec 20, 2005.

[8] Sun Microsystems, *Jar File Specification* URL: http://java.sun.com 1995

[9] PKWARE®, *Zip File Specification* ver.6.3.2, September 28, 2007. URL: http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[10] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd Ed., Addison-Wesley, 1999.

[11] 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.

[12] Z. Qian, A. Goldberg, and A.Coglio, “A Formal Specification of Java™ Class Loading,” Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304 July 21, 2000

[13] S. Liang and G. Bracha, "Dynamic Class Loading in the Java TM Virtual Machine," Sun Microsystem Inc. 901 San Antonio Road, CUP02-302 Palo Alto, CA 94303

[14] P. S. Corporation, Embedded Caffeine Mark 3.0, URL: http://www.benchmarkhq.ru/cm30/info.html, 1997.

[15] R. Y. Tsai," A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-shelf TV Cameras and Lenses," *IEEE Journal of Robotics and Autimation*, vol. RA-3, NO. 4, Aug. 1987

[16] N. GUIL and E. L. ZAPATA, "Lower Order Circle and Ellipse Hough Transform," 1997 *Pattern Recognition Society*. Published by Elsevier Science Ltd

[17] P. L. Rosin, "A note on the least squares fitting of ellipses," *Pattern Recognition Letters* 14 Elsevier Science Publishers (1993) 799-808

[18] M. Tapper, P. McKerrow, J. Abrantes, " Problems Encountered in the Implementation of Tsai's Algorithm for camera calibration," Proc. 2002 *Australasian Conference on Robotics and Automation Auckland*, 27 - 29, Nov. 2002

[19] J. Heikkilä, "Geometric camera calibration using circular control points," *IEEE transactions on pattern analysis and machine intelligence* [0162-8828] Heikkila yr:2000 vol:22 iss:10 pg:1066

[20] Z. Zhang, "A Flexible New Technique for Camera Calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence* [0162-8828] Zhang yr:2000 vol:22 iss:11 pg:1330

[21] O.D. Faugeras, Q.-T. Luong and S.J. Maybark, *Camera Self-Calibration: Theory and Experiments*, Institut National de Recherche en Informatique et Automatique (INRIA), 2004 Routs des Lucioles, 06560 Valbonne, France

[22] L. Edwards, "*Active Shutter 3D Technology for HDTV*," PhysOrg, Sept. 25, 2009

URL: http://www.physorg.com

[23] Xilinx LogiCore, *PLB IPIF (v2.02a)*, Xilinx Production Specification DS448, April, 2005.

[24] Xilinx LogiCore, *Processor Local Bus (PLB) v4.6*, Xilinx Production Specification DS531, Aug, 2007.

[25] Blender Home, Blender Tool 2.49, URL: http://www.blender.org/

[26] K. G. Nyman, “Real-World Camera Andscene Matching In Blender,” http://home.metrocast.net/~chipartist/BlensesSite/index.html, 2007.

[27] Security Engineering Research Group, ,*Analysis of Dalvik Virtual Machine and Class Path Library* , Institute of Management Sciences Peshawar, Pakistan Technical Report URL: http://serg.imsciences.edu.pk , Nov., 2009.