

國立交通大學

資訊工程學系

碩士論文

高度可擴充性 DVB-MHP 平台上的軟硬體協同設計
Java VM 之動態編碼最佳化

Dynamic Code Optimization for Java VM Hardware/Software

Co-design of a Highly Upgradeable DVB-MHP Terminal

研究生：林君玲

指導教授：蔡淳仁、李素瑛 博士

中華民國九十四年六月

高度可擴充性 DVB-MHP 平台上的軟硬體協同設計
Java VM 之動態編碼最佳化
Dynamic Code Optimization for Java VM Hardware/Software
Co-design of a Highly Upgradeable DVB-MHP Terminal

研究生：林君玲

Student : Chun-Ling Lin

指導教授：蔡淳仁、李素瑛

Advisor : Chun-Jen Tsai, Suh-Yin Lee

國立交通大學
資訊工程系
碩士論文



Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

高度可擴充性 DVB-MHP 平台上的軟硬體協同設計 Java VM 之動態編碼最佳化

學生：林君玲

教授：蔡淳仁、李素瑛 博士

國立交通大學資訊工程學系（研究所）碩士班

摘 要

多媒體家用平台(MHP)是由 Digital Video Broadcasting(DVB)所提出，作為互動電視家用娛樂平台上的中介軟體公開標準，這個平台使用 Java 為主要的程式語言，由 Java 虛擬機器(VM)負責程式的運作執行。採用軟硬體協同設計的方式能讓 Java 虛擬機器具有高度的可擴充性，功能也強大許多，但仍舊會受限於 Java 語言本身的效率不彰；而傳統的動態編碼最佳化雖然可以利用一些執行時期所得的資訊來加速系統，但對於嵌入式系統來說，這個代價是十分昂貴的。因此，在這篇論文中，我們提出了一個新的動態編碼最佳化演算法，用軟硬體協同設計的方式使這類系統的整體效能大大的提升，並且更加的省電。我們將這樣的想法實作在 Java Optimized Processor(JOP)上，並且在 Xilinx 的 Spartan-3 發展板上模擬執行，實驗結果顯示我們所提出的這套架構在整體上可增進 13.8% 的速度；在省電方面，也分別可以減少 10.4% 的微指令執行週期以及 11.1% 的外部記憶體存取。

Dynamic Code Optimization for Java VM Hardware/Software Co-design of a Highly Upgradeable DVB-MHP Terminal

Student: Chun Ling, Lin

Advisors: Dr. Chun Jen, Tsai
Dr. Suh Yin, Lee

Institute of Computer Science and Information Engineering
National Chiao-Tung University

ABSTRACT

Multimedia Home Platform (MHP) is the open middleware system for interactive television and related interactive home entertainment designed by the Digital Video Broadcasting (DVB) project. They use Java as the common programming language and embed the Java Virtual Machine (VM) that provides a stable and cross-platform java runtime environment in the system software layer. A hardware/software co-design approach makes Java VM more flexible and powerful, but it still suffers from the inefficiency of java system. Typical dynamic code optimization can save method lookup and constant pool searching time using the runtime information known in the first time we execute it. However, in such kind of embedded system, it is very expensive due to the overhead of external memory modification. In this thesis, we propose a new hardware/software co-design dynamic code optimization schema for this kind of approach that can significantly improve the efficiency of Java program execution. By analyzing the execution frequency of Java code segment, we can dynamically decide if the dynamic code optimization is needed. This approach can also cut down the power consumption with less microcode execution cycles and less external memory access. We implement this architecture on Java Optimized Processor (JOP) and simulate on Xilinx Spartan-3 developing board. Experiment Result shows that this proposed dynamic code optimization schema for Java VM hardware/software co-design of DVB-MHP terminal has 13.8% average speedup, 10.4% less microcode execution cycles and 11.1% less external memory access than the original system.

誌 謝

這篇論文能夠如此順利的完成，要感謝許多人對我的幫助。首先要感謝的是我的指導教授們，尤其是蔡淳仁老師，給予我許多寶貴的意見與指導，使我學到許多專業的知識、實作上的技巧，以及寫作英文科技論文的方法。還有要謝謝跟我一起合作的伙伴們，謝謝黃士嘉學長，對於硬體不熟的我幫助指導甚多。接著要感謝 JOP 的作者 Martin，他真的是一個很厲害又很親切的人，在這篇論文的實作期間，不厭煩的回答我任何問題，甚至幫我解決程式上的困難，沒有他我想我不會進行得如此順利。感謝交大資工，讓我在實驗室能夠擁有如此豐富的資源及優良的學習環境，實驗室的每個人都非常的和善又專業，對於我各式各樣奇怪又笨拙的問題總是不吝惜的教導以及包容，使我得以在一個非常快樂的氣氛中學習和做研究。最後，要感謝長期支持我的家人以及朋友們，尤其是洪智傑，在我論文實作以及撰寫的期間給予我非常多實質上的幫助，和我分享快樂的事情，也幫我分擔了許多的瑣事以及壓力下所產生的負面情緒，讓我能專心的完成論文。謝謝所有關心我的親朋好友們，僅將這篇論文，獻給各位。

Table of Content

摘要.....	i
ABSTRACT	ii
誌謝.....	iii
Table of Content	iv
List of Figures.....	vi
List of Tables	vii
1. Introduction	1
1.1. Why Dynamic Code Optimization (DCO)	1
1.1.1. Dynamically Typed Object-Oriented Languages	2
1.1.2. Dynamic Message Sending	2
1.2. DCO for Java VM Using HW/SW Co-design Approach	4
1.3. Advantages of DCO & HW/SW Co-design for DVB-MHP Applications	5
1.4. Overview of this Thesis	6
2. Related Work	7
2.1. Previous DCO Mechanisms	7
2.1.1. Lookup Cache Mechanism in Smalltalk-80	7
2.1.2. Inline Cache Mechanism in Smalltalk-80	9
2.1.3. Polymorphic Inline Cache in SELF System	10
2.1.4. Java Virtual Machine Reference Implementation	13
2.1.4.1. Sun's Java Virtual Machine Reference Implementation.....	14
2.1.4.2. Sun's K Virtual Machine Reference Implementation.....	18
2.2. Java Platform	20
2.2.1. Java Execution Flowchart.....	22
2.2.2. Java Class File Format.....	22
2.2.3. Java Virtual Machine (JVM)	24
2.2.4. JVM Instruction Set.....	25
2.3. Implementations of JVM	29
2.3.1. Interpreter	30
2.3.2. Just-In-Time (JIT) Compiler.....	31
2.3.3. HotSpot technology	32
2.3.4. Java Processor	34
3. Problem Formulation.....	37
3.1. Introduction	37
3.2. Java Optimized Processor - JOP.....	38
3.2.1. Software Layer Stack of JOP.....	40
3.2.2. System Architecture of JOP.....	41
3.2.3. Datapath of JOP	42
3.2.4. Hardware/Software Co-design of JOP	43
4. Proposed Dynamic Code Optimization System	45
4.1. Data Structure Using in Our Dynamic Code Optimization.....	45
4.1.1. Data Arrangement in the External Memory	45
4.1.2. Method Cache.....	48
4.1.3. Runtime Data Structure	49
4.1.3.1. Stack Frame	49
4.1.3.2. Data Layout	51

4.1.3.3.	Runtime Class Structure	51
4.2.	The Proposed Dynamic Code Optimization Scheme	52
4.2.1.	Analysis of Bytecode Execution Frequency.....	52
4.2.2.	Access Time of External Memory & Internal Memory.....	55
4.2.3.	Architecture Overview	57
4.3.	Implementation Details	59
4.3.1.	Hardware Implementation Modules	60
4.3.2.	Software Implementation Modules	62
5.	Performance Study	64
5.1.	Xilinx Spartan-3 Developing Board	64
5.2.	Java Benchmark Programs	65
5.2.1.	Sieve of Eratosthenes.....	66
5.2.2.	Kfl.....	66
5.2.3.	UDP/IP	67
5.3.	Experiment Results.....	68
5.3.1.	Execution Time.....	68
5.3.2.	Power consumption	69
5.3.2.1.	Microcode Execution Cycles.....	69
5.3.2.2.	External Memory Access Times	72
6.	Conclusion and Future Work	74
	REFERENCES	76



List of Figures

Fig 1. Indirect Access Example	2
Fig 2. Polymorphic Operations Example	4
Fig 3. DVB – MHP Functional Block	5
Fig 4. Selection Mechanism of Lookup Cache.....	8
Fig 5. Inline Cache	9
Fig 6. Polymorphic Inline Cache (PIC).....	10
Fig 7. Inlining a Small Method into Polymorphic Inline Cache	12
Fig 8. Impact of Polymorphic Inline Cache	12
Fig 9. Inline Cache Miss Ratios	13
Fig 10. Original Execution Flowchart	15
Fig 11. Java Class File	17
Fig 12. Execution with Fast Bytecodes	18
Fig 13. Java 2 nd Edition (Source: http://java.sun.com).....	19
Fig 14. Java Instruction Format Using DCO.....	20
Fig 15. Java Platform (Source: http://java.sun.com)	21
Fig 16. Java Execution Flowchart	22
Fig 17. Structure of Java Class File.....	23
Fig 18. Components of Java Runtime System.....	25
Fig 19. Implementations of JVM.....	30
Fig 20. Interpreter.....	31
Fig 21. Just-In-Time Compiler.....	32
Fig 22. HotSpot	33
Fig 23. Java Processor.....	34
Fig 24. Software Architecture of Jazelle Chip.....	36
Fig 25. Java Optimized Processor Runtime Environment.....	38
Fig 26. Software Layer Stack of JOP	41
Fig 27. Block Diagram of JOP	42
Fig 28. Datapath and Data Flow of JOP.....	43
Fig 29. Data Arrangement in the External Memory	46
Fig 30. Method Table Structure	47
Fig 31. Stack Change on Method Invocation	50
Fig 32. Object Format.....	51
Fig 33. Array Format	51
Fig 34. Runtime Class Structure.....	52
Fig 35. Transmeta Code Morphing Software Control Flow.....	53
Fig 36. Distribution of Bytecode Execution Frequency	55
Fig 37. Microcode Sequence of External Memory Read	56
Fig 38. Microcode Sequence of External Memory Write.....	56
Fig 39. Our JDCO Architecture Overview	58
Fig 40. Java Bytecode Fetch Stage of Our JDCO	60
Fig 41. Block Diagram of The Proposed JDCO.....	61
Fig 42. The Top Side of Xilinx Spartan-3	65
Fig 43. The Bottom Side of Xilinx Spartan-3	65
Fig 44. Pictures of a Kippfahrleitung Mast in Down and Up Position.....	67
Fig 45. Execution Time	69
Fig 46. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185	71
Fig 47. External Memory Access Times of Bytecodes 180. 181. 182. 185.....	73

List of Tables

Table 1. Fast Bytecodes in Sun’s Java VM Reference Implementation	14
Table 2. Support Data Type of Java VM	26
Table 3. The Providing Types of JVM Opcodes.....	27
Table 4. A Comparison of Different Implementations of <i>imul</i>	44
Table 5. The Number of Bytecodes under the Given Execution Frequency.	54
Table 6. Our Designed JDCO Bytecodes & Their Formats.....	63
Table 7. Execution Time	68
Table 8. Microcode Execution Cycles of Each Bytecode	70
Table 9. Execution Times of Bytecodes 180. 181. 182. 185	70
Table 10. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185.....	71
Table 11. External Memory Access Times of Each Bytecode.....	72
Table 12. External Memory Access Times of Bytecodes 180. 181. 182. 185	73



1. Introduction

DVB-MHP provides an open standard for interactive digital television and home entertainment. In DVB-MHP, the DVB-J functional block uses Java VM to construct a cross-platform Java runtime environment. However, this Java VM technology also makes the system less efficient.

In this Chapter, we first presented a popular method called dynamic code optimization (DCO) for speeding up Java VM. Using DCO in a hardware/software co-design approach is examined in section 2. In section 3, we list the advantages of DCO and hardware/software co-design for DVB-MHP applications. Finally, the overview of this thesis is given in section 4.



1.1. Why Dynamic Code Optimization (DCO)

Code optimization for dynamically typed object-oriented languages is more difficult than statically typed object-oriented languages. Research shows that the main bottleneck is in the unpredictability of dynamic message sending, which is determined at runtime for dynamically typed object-oriented languages.

In this section, we first illustrate the differences between statically and dynamically typed object-oriented languages, and then we focus on dynamic message sending. Optimizing code dynamically on this topic will significantly improve the efficiency of the system.

1.1.1. Dynamically Typed Object-Oriented Languages

Dynamically typed object-oriented languages, such as Smalltalk and Java, are much slower than statically typed languages like C++. The reason is that the reference variables in dynamically typed languages may potentially reference to any objects in the program at runtime. Therefore type checking of the references can only be done at runtime. Furthermore, the addresses of the dynamic objects are also unknown at compile time. As a result, indirect access must be used, which is again very expensive at runtime. [4]

Consider the Java program segment in Fig 1, integer i is a local variable in method $m(B)$, and f is an object field in class B . Object cc is sent to method $m(B)$, and the field f of object cc is retrieved and assigned to local variable i . Because the address of object cc is unknown at compile time, the address resolution of $cc.f$ must be done at runtime. When executing the statement $i = cc.f$, the address of object cc is retrieved first, and then the address of field f is calculated based on the address of the object cc . As a result, there are two indirect accesses in order to get the value of $cc.f$. These accesses cause the inefficiency of executing dynamically-typed object-oriented programs.

```
class A {  
    public void m(B cc) {  
        int i;  
        i = cc.f;  
        ...  
    }  
}
```

Fig 1. Indirect Access Example

1.1.2. Dynamic Message Sending

In object-oriented languages, message sending is the most frequent operations. When we invoke a method, a message is sent to a class or an object, which selects the method to

be executed. Message sending is also called method invocation in some languages.

Polymorphic operations from dynamic binding and inheritance make it easy for object-oriented language programmers to develop well-designed systems, but also result in the difficulty of efficient execution of these programs. Because the address of the method can only be determined at runtime. To perform a message sending we must extract the name of the method, use it as a key to find the method in the current class (or in the superclass that this method is inherited), continue in this way up the class hierarchy until we find the corresponding method or the top of the inheritance hierarchy is reached.

In Fig 2, we will show how the polymorphic operations make the execution of object-oriented programs more difficult.

Class *A* is the superclass of class *B*, and the *m1()* method of class *B* override the *m1()* method of class *A*. *m0(A)* is a method of class *A*, and *m1()* method is invoked in it. *m2()* is also a method of class *A*, which method is just directly inherited in class *B*. Note that in the main program, the two statements *x.m0(y)* and *x.m0(z)* will invoke *a.m1()* while execute method *m0(A)*. In the first message sending, the class of *y* is *A*, so the statement *a.m1()* will invoke the *m1()* of class *A*. While in the second message sending, the class of *z* is *B*, so the statement *a.m1()* will invoke the *m1()* of class *B*. Inheritance property also makes it difficult to determine the access addresses in object-oriented programs. Consider the statement *z.m2()* in Fig 2. The class of *z* is *B*, but we can not find the method *m2()* in class *B*, so we try to look it up in the superclass of *B*, i.e., class *A*. The address of method *m2()* in class *A* is then retrieved in order to execute this statement. From this example, one can realize that the dynamic message sending is the crucial property of dynamically typed object-oriented languages.

By analyzing the message sending behavior, we can develop dynamic code optimization techniques to improve the efficiency of the language systems. Using the

caching mechanism, some duplicated method lookup procedure can be prevented. In this thesis, an adaptive dynamic code optimization mechanism for a java virtual machine is developed. By modifying the runtime behavior, method invocation can be more efficient and the extra memory required for this technique is limited.

```
Class A {  
    public void m0(A a) {  
        a.m1();  
    }  
  
    public void m1() {  
        ...  
    }  
  
    public void m2() {  
        ...  
    }  
}  
  
Class B extends A {  
    public void m1() {  
        ...  
    }  
}  
  
main() {  
    A x = new A();  
    A y = new A();  
    B z = new B();  
  
    x.m0(y);  
    x.m0(z);  
    z.m2();  
}
```

Fig 2. Polymorphic Operations Example

1.2. DCO for Java VM Using HW/SW Co-design Approach

Java is also a dynamically-typed pure object-oriented language developed by Sun Microsystems in the early 1990. It has many features of modern programming languages,

such as simple, object-oriented, robust, secure, architecture neutral, automatic garbage collection, dynamic linking, multi-threaded, and portability. However, it loses the efficiency. Slow execution speed makes Java incapable of handling multimedia applications efficiently without resort to native code or hardware accelerator.

Pure hardware implementation approach, such as java processor, can improve the execution speed greatly. The disadvantages are high design cost and low upgradeability. Hardware/software co-design takes the advantages of both approaches: low cost, flexibility and efficiency, but the execution speed can not be as fast as the pure hardware approach. DCO can significantly improve the system efficiency, which makes this HW/SW co-design approach more useful and powerful.

1.3. Advantages of DCO & HW/SW Co-design for DVB-MHP

Applications

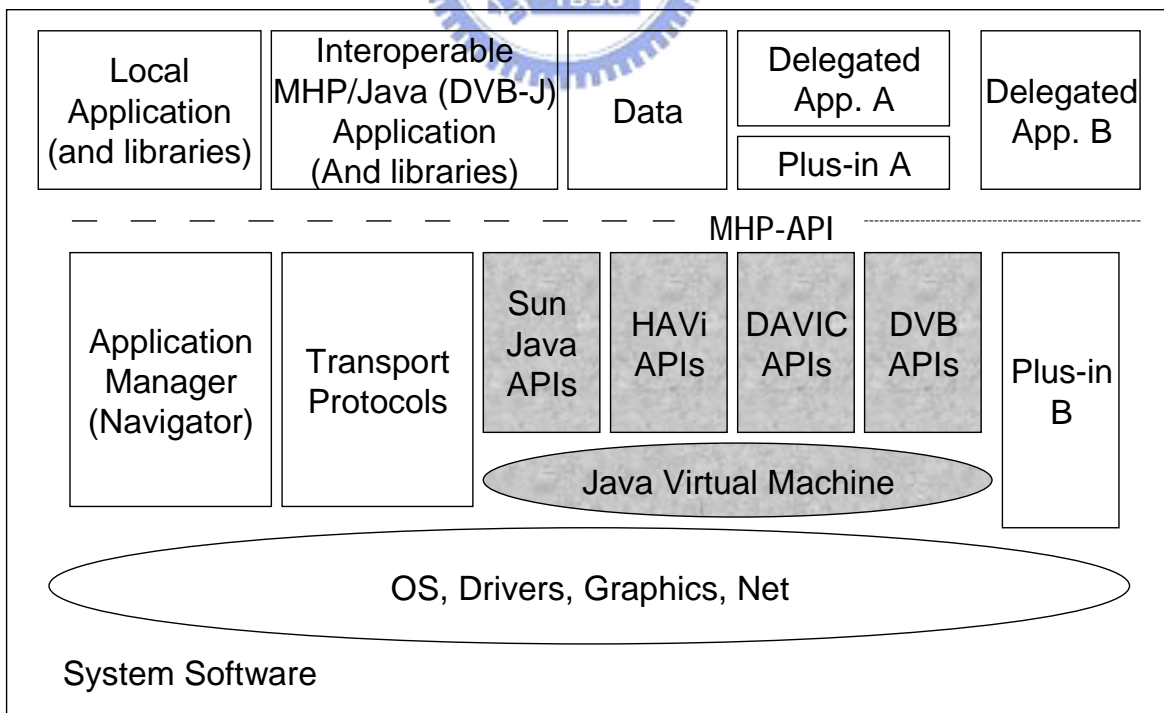
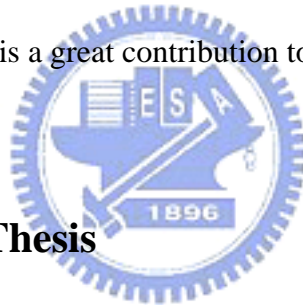


Fig 3. DVB – MHP Functional Block

In 2000, the Digital Video Broadcasting (DVB) organization proposed an open middleware system standard called Multimedia Home Platform (MHP), which is designed for interactive television and related interactive home entertainment applications. Java programming language is chosen as the common language of this platform, which is in the DVB-J functional block as the dark region in figure 3 (see [10]).

The underneath Java Virtual Machine plays an important role in DVB-MHP System. It provides a stable and cross-platform Java runtime environment. Java API developers do not need to know the underlying system software information so they can put more efforts on the libraries themselves. Because of the interactive and real-time demand, the execution speed is the crucial factor of DVB-MHP terminal. Using DCO and hardware/software co-design approach can make Java execution more efficiency. Furthermore, it can cut down the power consumption, which is a great contribution to such kind of embedded system.



1.4. Overview of this Thesis

The rest of this thesis is organized as follows. In chapter 2, several related works are listed and reviewed. Previous DCO mechanisms are also discussed here. In chapter 3, we formulate the problem, and introduce our target hardware/software co-design system – Java Optimized Processor (JOP) and the target developing board. The main ideas of the proposed dynamic code optimization scheme are presented in chapter 4. In chapter 5, the simulation result is shown and discussed. Finally, the conclusion and future work are given in chapter 6.

2. Related Work

In this chapter, we first list some papers and systems about dynamic code optimization. Then we introduce the Java platform including Java execution flow, Java class file format, JVM and its instruction set. In the next section, popular implementation approaches of JVM are discussed, including Java interpreter, Just-In-Time compiler, HotSpot, and Java processor.

2.1. Previous DCO Mechanisms

In this section, we will discuss several dynamic code optimization mechanisms for various dynamically typed object-oriented programming language systems. This concept was first proposed in 1983 [1], with implementation of the smalltalk-80 system. It is called lookup cache. In 1984, an efficient implementation of the Smalltalk-80 system that used a modified cache mechanism (called inline cache) was presented by Deutsch and Schiffman [2]. The inline cache concept now is adapted into many object-oriented language systems. One classical example is polymorphic inline cache, which is implemented in SELF system [3]. Another famous implementation is in the Java programming language. The Java virtual machine and K virtual machine of Sun's reference implementation which adopts this mechanism will be discussed in the end of this chapter.

2.1.1. Lookup Cache Mechanism in Smalltalk-80

The Smalltalk definition specifies that the source code is translated into a sequence of primitive operations called *byte codes*. Smalltalk-80 was originally run on virtual machines

which implemented the byte codes in microcode. Early implementations of Smalltalk-80 on hardware interpreted the byte code in software, which led to poor performance [5]. *Ungar* and *Patternson* proposed a lookup cache mechanism that can improve the performance of message sending for Smalltalk. [1]

Lookup caches are used to cache the previous lookup result. Method addresses are retrieved from the lookup cache, a hash table of the most recently used method addresses, via the pair (receiver class, message selector) as the key. The receiver class is the class that the called object belongs to, and the message selector selects the method to be executed. Fig 4 illustrates the selection mechanism of the lookup cache. When a method is invoked, the pair (receiver class, message selector) is used as a key to the lookup cache. If it hits this hash table, the message address will be extracted and the method lookup procedure can be avoided. Otherwise, the method lookup routine will be processed. And then the new address information will be kept in the lookup cache for next method invocation.

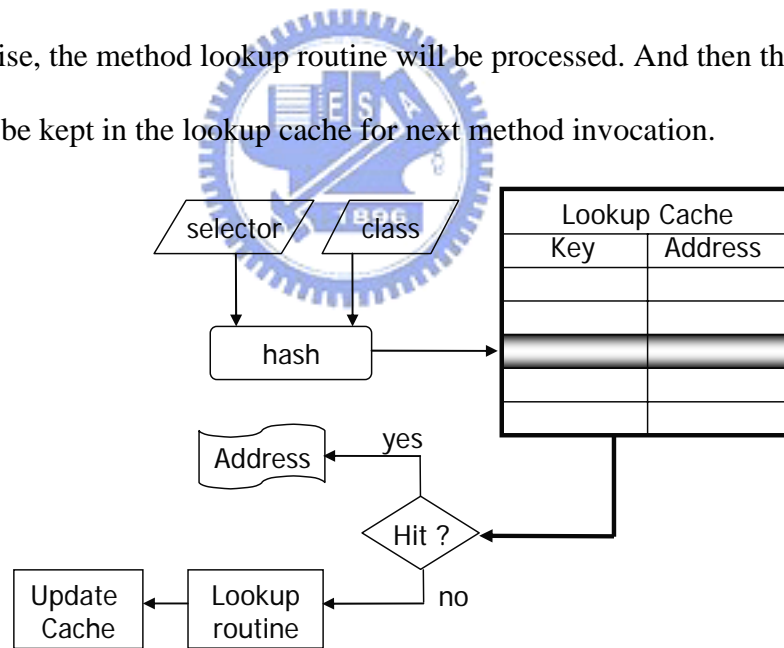


Fig 4. Selection Mechanism of Lookup Cache

Lookup cache is very effective in reducing the lookup overhead. Berkeley Smalltalk [1], for example, would have been 37% slower without a cache. Furthermore, if the hit ratio of the lookup cache is high, this advancement will be more observable.

2.1.2. Inline Cache Mechanism in Smalltalk-80

The inline cache mechanism proposed in 1984 [2] predicts the method addresses and places them in the message send site. Even with a lookup cache, sending a message still takes considerably longer than calling a simple procedure because the cache must be probed for every message sent. However, send operations can be sped up further by the observation that the class of the receiver at a given call site rarely varies; that is, if a message is sent to an object of class X at a particular call site, it is very likely that the next time the send is executed will also have a receiver class X.

This locality of receiver class usage can be exploited by caching the most recently look-up method address at the call site (e.g. by overwriting the call instruction). Fig 5 (see [5]) shows the modification using this technique. Subsequent executions of the sent code jump directly to the cached method, completely avoiding any lookup. Of course, the class type of the receiver could have changed, so the calling method procedure must verify that the receiver class is correct and call the lookup routine if the type test fails. After updating the method code of the receiver class, it may be matched and the method lookup cost can be saved next time. This form of caching proposed by *Deutsch* and *Schiffman* is called inline cache since the target address is stored at the sent point. [2]

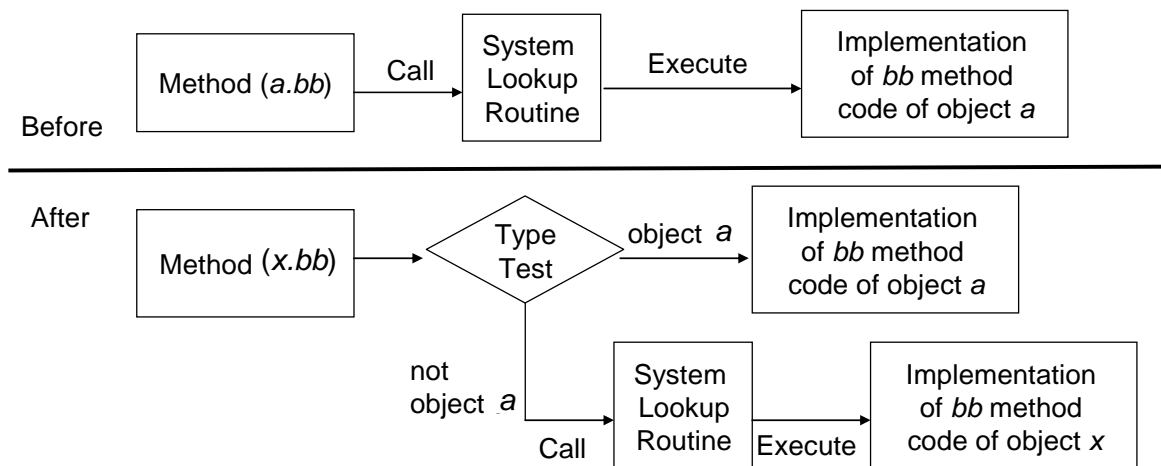


Fig 5. Inline Cache

Inline caching is surprisingly effective, with a hit ratio of 95% for Smalltalk code [2]. SOAR, a Smalltalk implementation for a RISC processor, would be 33% slower without inline cache [6]. Nowadays all compiled implementations of Smalltalk that we know is integrated with inline cache mechanism.

2.1.3. Polymorphic Inline Cache in SELF System

Inline cache mechanism is effective only if the receiver class remains relatively constant at a call site. Although it works very well for the majority of sends, it does not speed up a polymorphic call site with several equally likely receiver classes because the call target switches back and forth between different methods. Worse, inline cache mechanism may even slow down these sends because of the extra overhead associated with inline cache misses.

Based on the inline cache technique, Polymorphic Inline Cache (PIC) caches all method addresses, if the degree of polymorphism is less than ten [3]. The example in Fig 6 (see [3]) illustrates this.

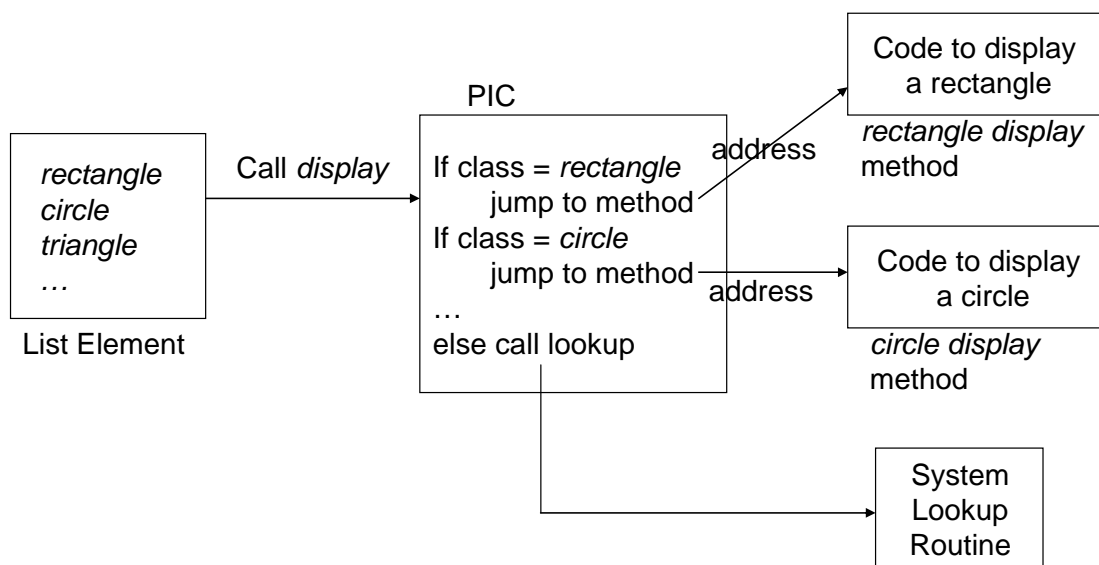


Fig 6. Polymorphic Inline Cache (PIC)

Suppose that the method *display* is sent to all classes in the list, the polymorphic inline cache mechanism will handle this method invocation. First, the list element is a *rectangle* class. Similar to the normal inline cache, the method address will be extracted and the calling code will jump to the direct method code to display a rectangle. It is the same with the class *circle*. Following the type test, a *triangle* class is passed. When the system finds that it is a new receiver class type that does not exist in current cache the Polymorphic Inline Cache handler will call the method lookup routine and construct a new branch routine for the *display* method to rebind the receiver class *triangle*. Next time the receiver class *triangle* is called, it can just branch to the corresponding code of the method.

If the cache misses again, the Polymorphic Inline Cache will simply be extended to handle the new case. Eventually, the Polymorphic Inline Cache handler will contain all cases seen in practice, and there will be no more cache misses or method lookup procedures. Thus, a Polymorphic Inline Cache is not a fixed-sized cache similar to a hardware data cache; rather, it should be viewed as an extensible cache in which no cache item is ever displaced by another newer item.

Since many methods are very short, the Polymorphic Inline Cache can be modified to be more effective and more space can be saved. At polymorphic call sites, short methods could be integrated into the Polymorphic Inline Cache handler instead of being called by it. For example, suppose the lookup routine finds a method that just loads the receiver's *x* field. Instead of using the stored method address to call this method from the handler, its code can be copied directly into the handler, eliminating the calling and return procedure. The figure in Fig 7 (see [3]) explains this example.

The hit ratio of the Polymorphic Inline Cache depends on the runtime behavior of the programs. In [3], this mechanism is implemented for SELF, a typical dynamically-typed pure object-oriented language. In SELF, all operations including variable accesses and basic

arithmetic operations are implemented by dynamically bound procedure calls.

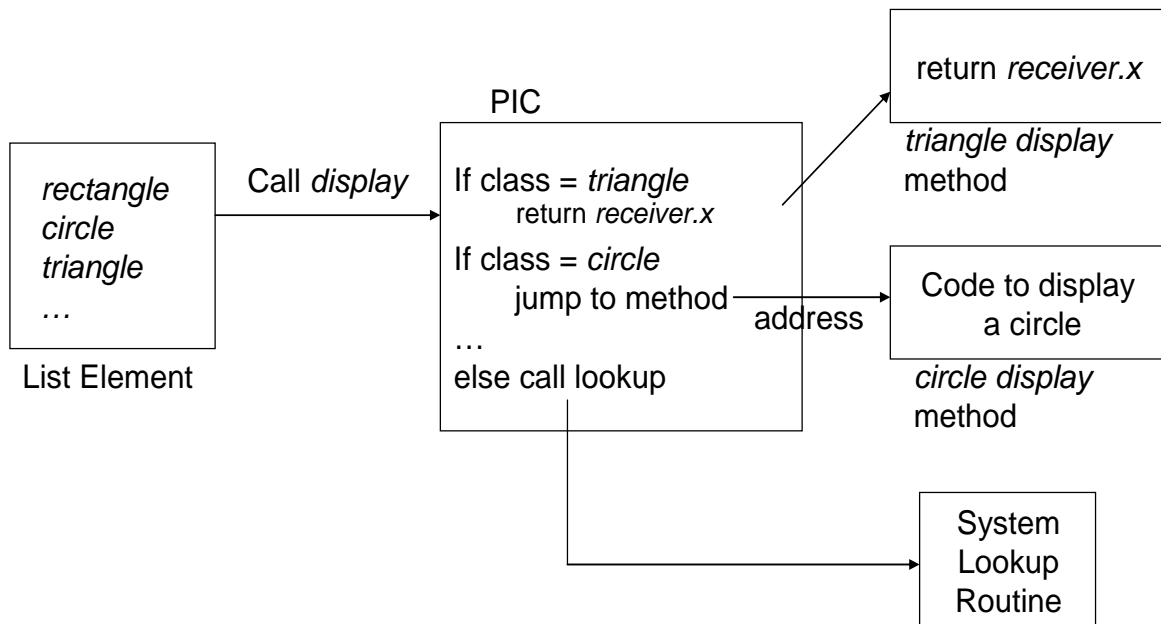


Fig 7. Inlining a Small Method into Polymorphic Inline Cache

Fig 8 (see [3]) shows the individual execution time with several benchmark programs. PolyTest is an artificial benchmark with only 20 lines that is designed to show the highest possible speedup with Polymorphic Inline Cache while all the others are produced by software in order to cover a variety of programming styles. The median speedup for the benchmark programs (without PolyTest) is 11%. And the space overhead of Polymorphic Inline Cache is very low, typically less than 2% of the compiled code.

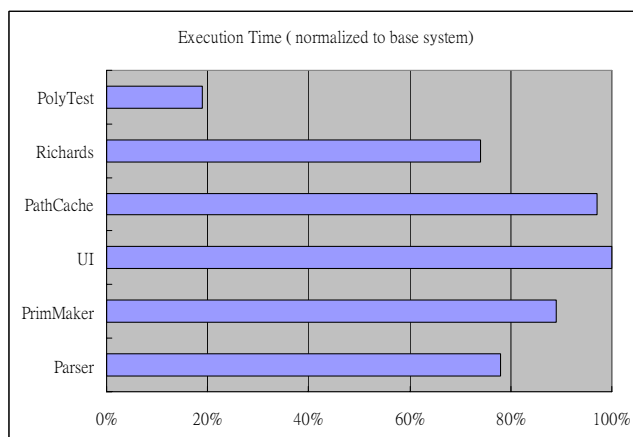


Fig 8. Impact of Polymorphic Inline Cache

This research also found an interesting observation. In Fig 9 (see [3]), there is no direct correlation between cache misses and the number of polymorphic call sites. For example, in these benchmark programs, one receiver type dominates at most call sites in PathCache, while the receiver class frequently changes in Parser's Inline Caches. Thus, ordering a Polymorphic Inline Cache Mechanism may win with programs like Parser.

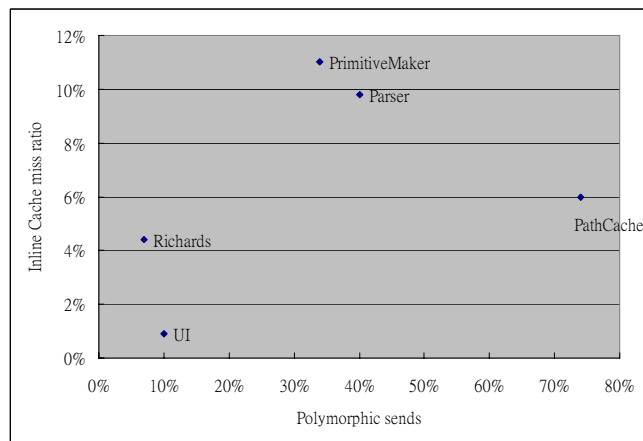


Fig 9. Inline Cache Miss Ratios

2.1.4. Java Virtual Machine Reference Implementation

The Java programming language relies on the simulated machine, known as Java Virtual Machine (JVM). JVM allows the computer programmer to communicate with the virtual machine instead of the real hardware system. This is advantageous, because it allows for portability. If the individual JVM are installed on two completely different machines, the Java programs should work well on both machines without any code modification, because it relies on the JVM and not the hardware system it is running on.

Sun Microsystems developed this powerful language system, and this language becomes very popular nowadays. Various Java VM were constructed by different teams that conform to the Java Virtual Machine Specification [7] but have independent implementations. For a reference implementation, Sun Microsystems also develop a Java

Virtual Machine and a K Virtual Machine for a part of the Java 2 Micro Edition (J2ME) called Connected Limited Device Configuration (CLDC) [8]. Dynamic code optimization is also used in these reference implementations to improve the efficiency of Java VM execution.

2.1.4.1. Sun's Java Virtual Machine Reference Implementation

In Sun's version of the Java Virtual Machine, compiled java Virtual Machine code is modified at runtime for better performance. This optimization takes the form of a set of pseudo-instructions that are distinguishable by the suffix `_quick` in their mnemonics. These are variants of normal Java Virtual Machine instructions that take advantage of information learned at runtime to do less work than the original instructions.

203	<code>ldc_quick</code>
204	<code>ldc_w_quick</code>
205	<code>ldc2_w_quick</code>
206	<code>getfield_quick</code>
207	<code>putfield_quick</code>
208	<code>getfield2_quick</code>
209	<code>putfield2_quick</code>
210	<code>getstatic_quick</code>
211	<code>putstatic_static</code>
212	<code>getstatic2_quick</code>
213	<code>putstatic2_static</code>
214	<code>invokevirtual_quick</code>
215	<code>invokenonvirtual_quick</code>
216	<code>invokesuper_quick</code>
217	<code>invokestatic_quick</code>
218	<code>invokeinterface_quick</code>
219	<code>invokevirtualobject_quick</code>
221	<code>new_quick</code>
222	<code>anewarray_quick</code>
223	<code>multianewarray_quick</code>
224	<code>checkcast_quick</code>
225	<code>instanceof_quick</code>
226	<code>invokevirtual_quick_w</code>
227	<code>getfield_quick_w</code>
228	<code>putfield_quick_w</code>

Table 1. Fast Bytecodes in Sun's Java VM Reference Implementation

To learn from inline cache mechanism [2], the Reference Implementation (RI) of Sun's JVM also uses the concept of caching the previous method lookup information and stores them in the instruction space. Only standard java bytecode instructions numbered from 0 to 201 may be generated by the java compiler. The optimization works by dynamically replacing occurrences of certain instructions by the reserved instructions (in the range of 202-255) after the first time they are executed. These new instructions listed in Table 1 have been loaded and linked the first time the associated regular instruction is executed.

Note that these new instructions (referred to as fast bytecodes) are not specified in the Java Virtual Machine Specification [7]. However, for the implementation of Java Virtual Machine the adoption of the fast bytecodes has been proven to be an effective optimization technique.

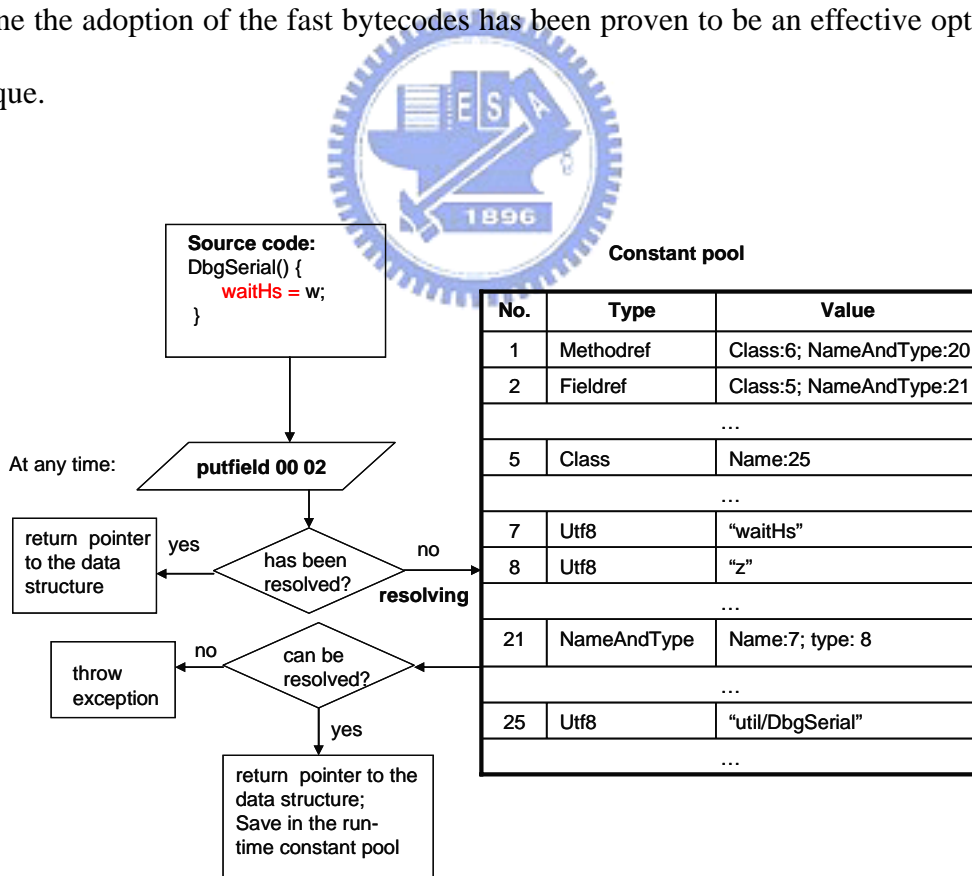


Fig 10. Original Execution Flowchart

Fig 10 shows the original execution flowchart if we do not enable fast bytecodes.

Consider the assignment instruction `waitHs = w` in Function `DbgSerial ()`. A sequence of java bytecodes will be generated after compilation, and `putfield 00 02` is the core instruction of this assignment. When Java VM fetches this instruction for execution, first it will check that if this constant pool component, indexed by 2 in this case, has been resolved. The java constant pool inside the java class file format (as the second block in Fig 11, which we will illustrate it in subsection 2.2.2) is designed to support dynamic linking. When the Java Virtual Machine encounters a use of a constant pool entry for the first time (e.g., when you first use the `new` statement to create a new object of a class, or in the first use of `getfield` to get a field), the constant pool entry is resolved [9].

The actions the JVM performs to resolve a constant pool entry depend on its type. Resolution of an entry involves two basic steps: checking that the item you are trying to access exists (possibly loading or creating it if it doesn't already exist), and checking that you have the right permissions to access the item (i.e., making sure that you don't access private fields in other classes, etc.). In Fig 10 the constant pool of the class `DbgSerial` is listed. The Java VM checks that the index 2 points to a field that belongs to the class `util/DbgSerial` (index 25), its name is `waitHs` (index 7), and its type is an integer ("z" in index 8). If any illegal situation happens, an exception will be thrown by the Java VM. After the entry is resolved, the address of this constant pool item will be returned for execution of the Java VM. At the same time, this address will be stored in the runtime constant pool of that class. Next time this constant pool is used, the Java VM will find that it has been resolved and use the direct address in the runtime constant pool.

format of operands are left up to the implementer. Just remember that the operands of the *_quick* pseudo-instruction must fit within the space allocated for the original instruction's operands.

With this dynamic code optimization, a significant amount of time is thus saved on all subsequent invocations of the pseudo-instruction.

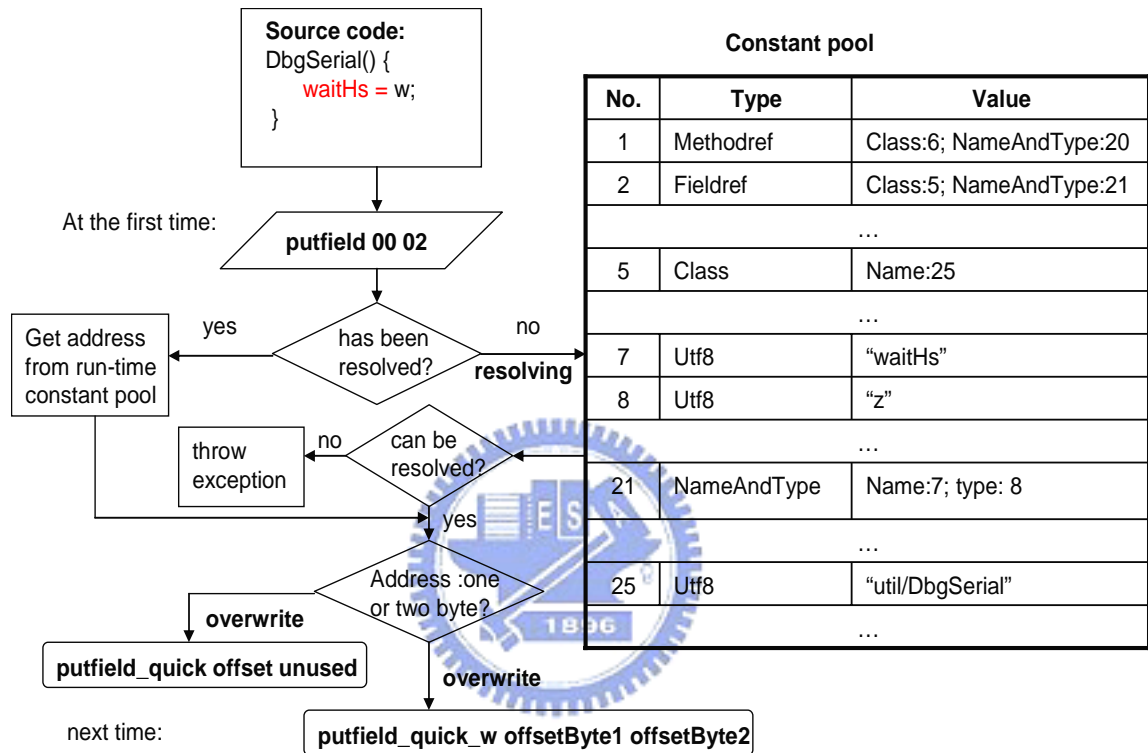


Fig 12. Execution with Fast Bytecodes

2.1.4.2. Sun's K Virtual Machine Reference Implementation

Recognizing that one size does not fit all, Sun Microsystems has grouped its Java technologies into three editions as in Fig 13, and each of them aimed at a specific area of today's vast computing industry. Java 2 Enterprise Edition (J2EE) is for enterprises needing to serve their customers, suppliers, and employees with solid, complete, and scalable Internet business server solutions. While Java 2 Standard Edition (J2SE) is for the familiar and well-established desktop computer market. The Java 2 Micro Edition (J2ME), targeted

at two broad categories of products: CDC (Connected Device Configuration) and CLDC (Connected, Limited Device Configuration), is specified for the consumer and embedded device manufacturers, service providers, and content creators.

For these three different Java editions, the underneath Virtual Machine also have different execution speed and ability. The K Virtual Machine (KVM) is developed for CLDC in Java 2 Micro edition, which is a compact, portable Java Virtual Machine specifically designed from the ground up for small, resource-constrained devices. The high-level design goal for the KVM was to create the smallest possible complete Java virtual machine that would maintain all the central aspects of the Java programming language, but would run in a resource-constrained device with only a few hundred kilobytes total memory budget.

In Sun's Reference Implementation, dynamic code optimization is also used in the K Virtual Machine. The implementation details are just like Sun's JVM RI, but `_fast` suffix is used as the new instructions instead of `_quick`. By caching the method lookup result in the call site, the time of searching constant pool and method table can be saved.

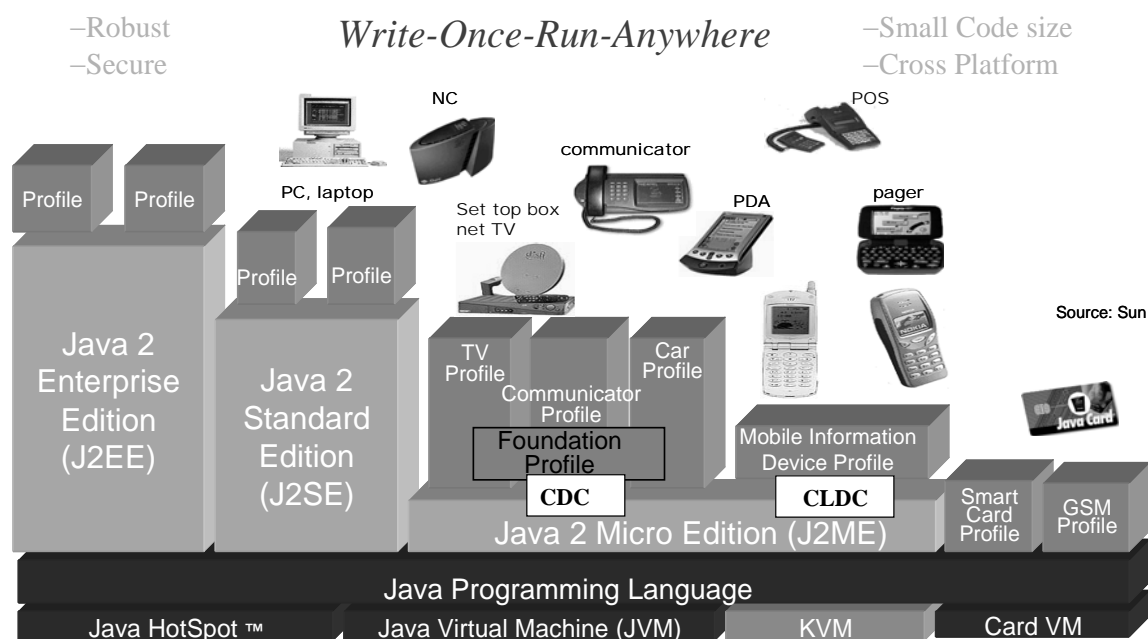


Fig 13. Java 2nd Edition (Source: <http://java.sun.com>)

As the restrictions of DCO in Java VM, the KVM implementers also need to assure that the executed java instructions are stored in RAM or other memory types that the stored data can be modified at runtime. The other important restriction is, the operands of the *_fast* pseudo-instruction must fit within the space allocated for the original instruction's operands. Instead of just saving the corresponding address, KVM provides a second technique to save more execution time. Some instructions need much information to be executed, such as *invokevirtual*, which instruction will invoke a method of an object instance. The information (e.g. parameter, method's return type, etc) now can be stored in an external memory called inline cache, and an index to the inline cache is used in the instruction operands. The new instruction format is illustrated in Fig 14.

This additional dynamic code optimization technique in Sun's KVM RI requires about 100 Kbytes extra memory, but it has been proven to be very efficient. The execution time with fast bytecodes enabled is two or three times faster than without it. Because using inline cache to execute method invocation, the performance of Sun's KVM RI is much better than the JVM RI.

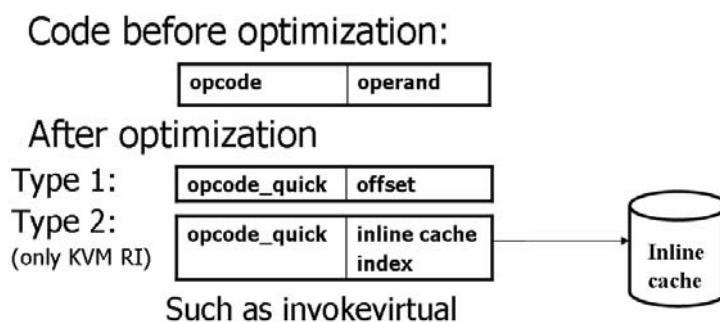


Fig 14. Java Instruction Format Using DCO

2.2. Java Platform

Fig 15 illustrates the layer structure of the Java Platform, which consists of six layers. The first underneath layer is Platforms layer. Sun provides implementations of Java

Development Kit (JDK) and Java Runtime Environment (JRE) for Microsoft Windows, Linux, and the Solaris operating systems. In addition, they can also run in any user-defined platforms if they have their own Java Virtual Machine, which is the second layer. Java Virtual Machine (VM) simulates the execution behaviors like a real machine, and it has its own instruction set. Java bytecodes can be executed by Java VM without knowing which platform behind it, so do the native programs. Up this structure, Java APIs and JNI provide basic features and fundamental functionalities for the Java platform. The fourth layer is development technologies, which enables applications written in other technologies and gives an integrated solution for that. Development Tools & APIs provide many useful tools such as Java compiler (javac), Java executer (java), document generator (javadoc), and etc. This structure provides the Java Programming Language a complete execution environment.

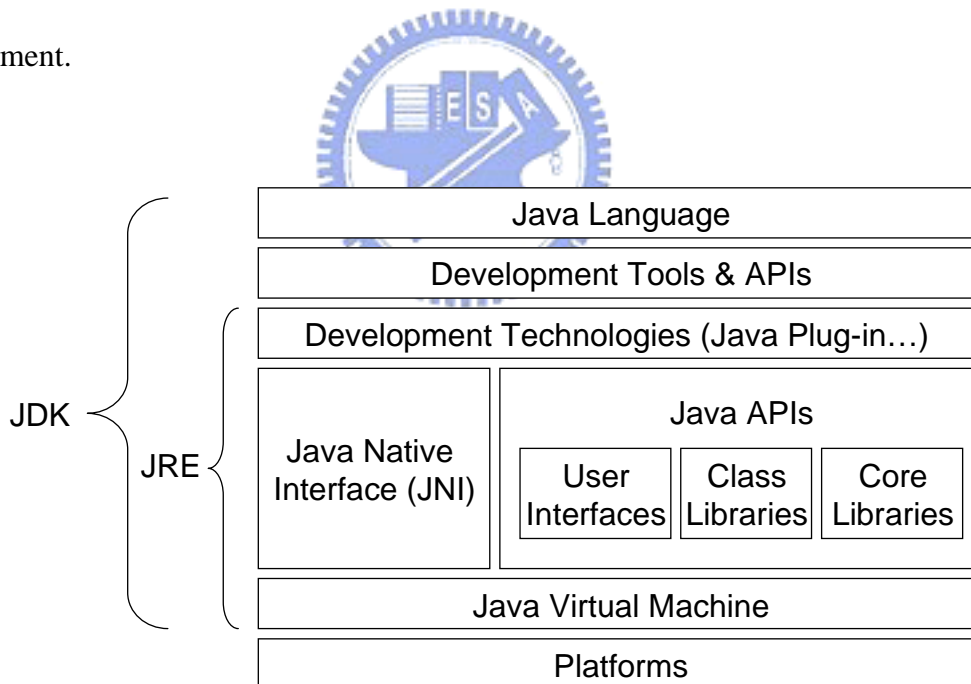


Fig 15. Java Platform (Source: <http://java.sun.com>)

In following subsections, we will give a detailed introduction to Java execution flowchart, Java class file format, Java Virtual Machine, and its instruction set. These knowledge are very important for the design of DCO.

2.2.1. Java Execution Flowchart

To understand the Java runtime system, the Java execution flowchart must be discussed first (see Fig 16). Java source programs are compiled by javac into Java bytecodes, and these bytecode sequences are organized into class files. Each class file contains exactly one class bytecodes and information including methods, fields and interface of this class. These class files are then loaded either from local storage or through data network into the Java Virtual Machine for execution.

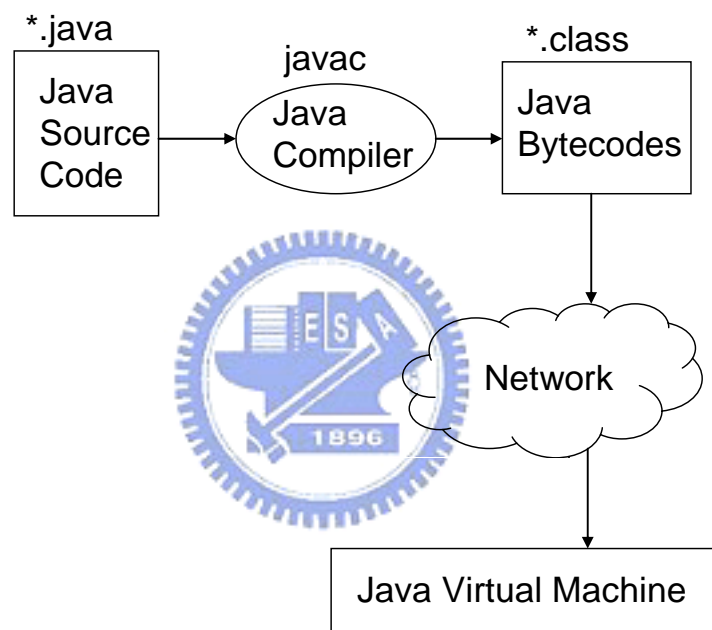


Fig 16. Java Execution Flowchart

2.2.2. Java Class File Format

We have mentioned the Java class file in subsection 2.1.4.1. Now let us look into more detail at the Java class file format. A format structure in Fig 17 illustrates it.

Java class files are structured in linear and record-based organization. Each class file contains seven sections in order: File Header, Constant Pool, Class Descriptor, Interface Table, Field Table, Method Table, and Attribute Table. File Format includes the magic

number, which is a signature that can be verified to make sure that it's a Java class file, and the version number. The version number indicates that which version of Java VM can execute it.

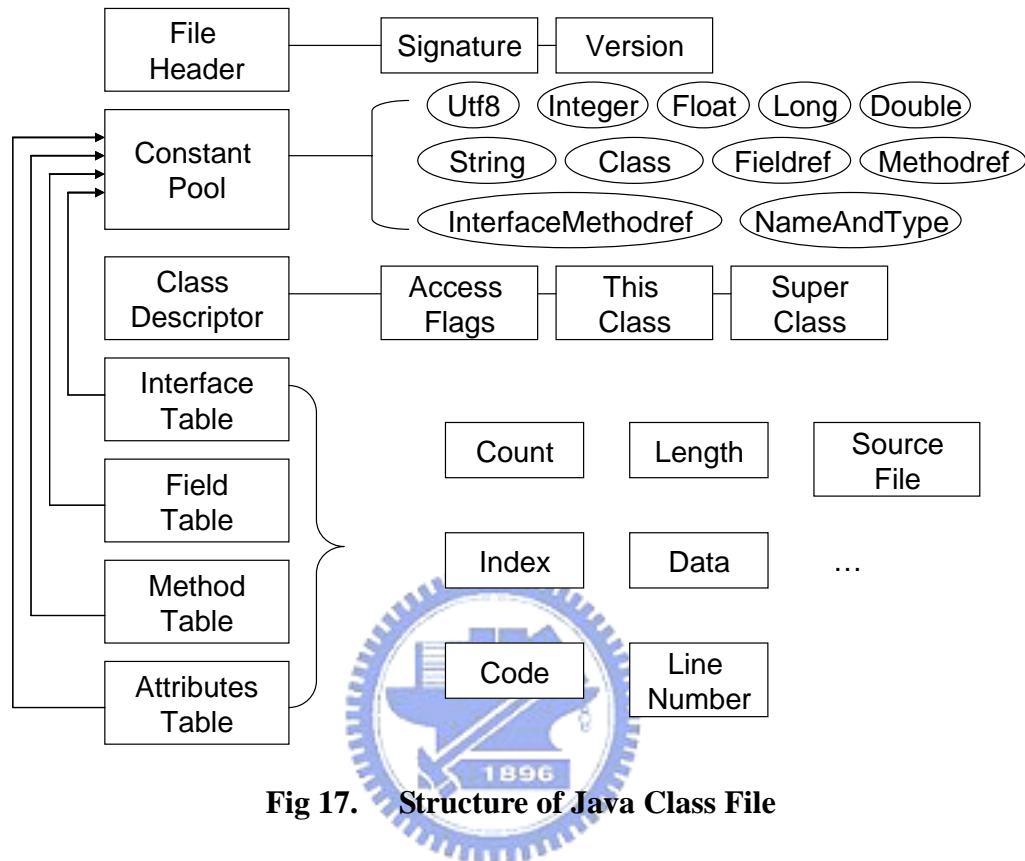


Fig 17. Structure of Java Class File

The Constant Pool acts as the symbol table of this class. It has eleven types: Utf8, Integer, Float, Long, Double, String, Class, Fieldref, Methodref, InterfaceMethodref, and NameAndType. First the count of this constant pool items is given. The items have variable length, and all multibyte data are in Big-Endian byte order. Java class files are written using the Unicode character encoding [11] which is a worldwide encoding standard. All strings in the constant pool are stored in the UTF-8 formats [12] in which Unicode characters are packed into bytes to reduce space usage.

The Class Descriptor section contains the Access Flags of this class, this class and super class. Next to the Class Descriptor are four tables: Interface Table, Field Table, Method Table, and Attribute Table. These tables contain all the information about interface,

field, method, and attribute. (e.g. count, length, index, data, code, and etc.)

Symbols and values presented in Class Descriptor, Interface Table, Field Table, Method Table, and Attribute Table are actually indexes which point to the constant pool. When the class file is loaded into Java VM, these symbols and values must be resolved from the constant pool before execution. Java uses this delayed symbolic resolution before execution to achieve dynamic binding. This binding is happened after class files are loaded into Java VM and before they are executed, but this also makes the inefficiency of VM so we will improve it in this research work.

2.2.3. Java Virtual Machine (JVM)

Java Virtual Machine is an abstract programmable computing machine with an instruction set called bytecode. It can be ported to different platforms to provide hardware and operating system independence.

Java VM is defined by the Java Virtual Machine Specification [7], which gives the details of the design such as Java class file format and the semantics of each instruction. Concrete implementations of Java VM specification are required to support these semantics correctly, and these implementations are known as Java runtime systems. Fig 18 (see [9]) shows the components in a typical Java runtime system.

Applet or application class files are loaded via local memory storage or network into Java VM. Dynamic Class Loader will handle the loading behavior and do the verifier, and then pass it to the Execution Engine with the standard-specified build-in Java classes. Execution Engine is the heart of any runtime system, which has many kinds of implementations. They can be hardware implementation, software implementation, or both of them. We will go deeply into it in section 2.3. Bytecodes are executed by Execution Engine with the simulated memory areas. Garbage collecting and other supporting codes

(e.g. Exceptions, Threads, Security, and etc.) are integrated into Execution Engine to enhance the ability of Java VM. If the programs use native programs, the native methods will be linked by Native Method Linker and acted like libraries for Java programs to be executed.

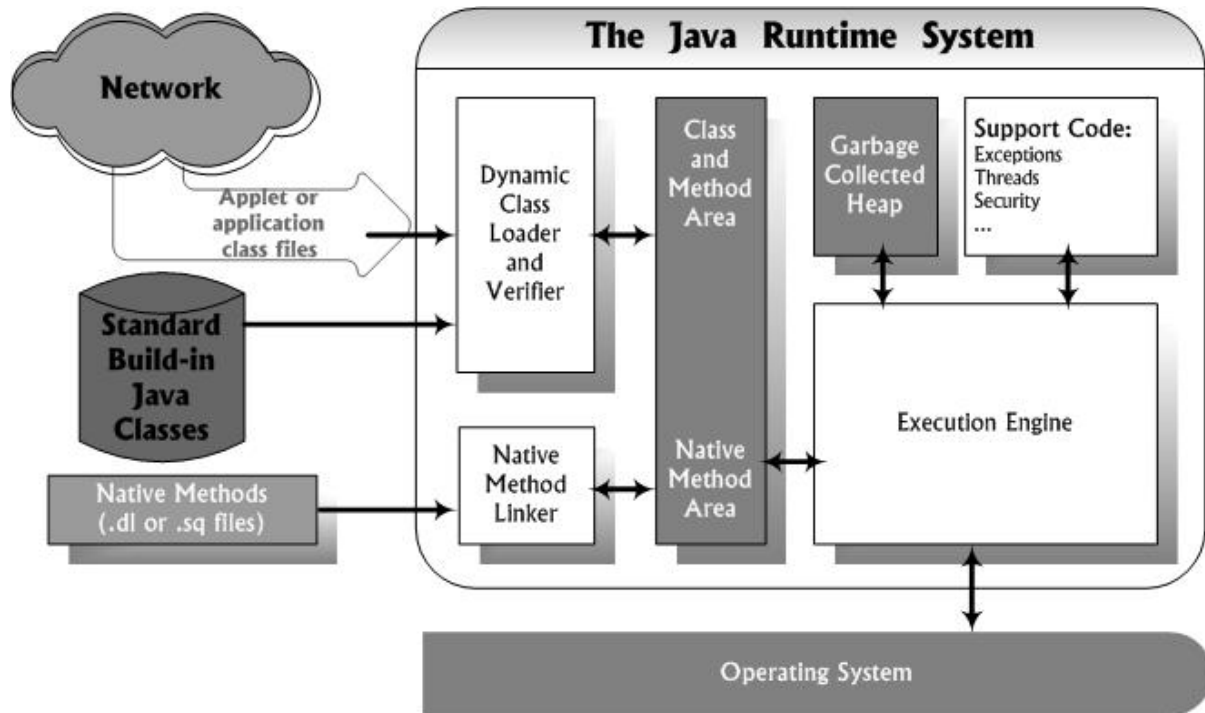


Fig 18. Components of Java Runtime System

2.2.4. JVM Instruction Set

JVM is a stack-based machine. It defines 201 standard instructions. Each instruction is represented by an 8-bit value, and this is the reason that the JVM instructions are called bytecodes.

JVM supports 9 primitive data types, which can be divided into two categories. One is numerical type, and the other is address type. Table 2 lists the 9 primitive data types and their respective lengths. [4]

Data Type	Length (Byte)
int	4
long	8
float	4
double	8
byte	1
char	2
short	2
reference	4
returnAddress	4

Table 2. Support Data Type of Java VM

reference and returnAddress are address types, and others are numerical types. char is unsigned, while byte, short, int, and long are signed. The floating-point types float and double represent single-precision 32-bit and double-precision 64-bit format IEEE 754 value. The values of reference types are pointers to class instances or fields. Arithmetic operations can not be applied to reference types, so do returnAddress types, which are pointers to opcode of JVM instructions. This type is used by jump instructions of JVM, and it is not corresponding to any data types in Java programming language.

There is one thing must be mentioned about the supported data types of JVM. Although Java programming language provides boolean data types, but JVM does not have boolean primitive data types. Instead, JVM uses int types to represent boolean values, and boolean arrays are represented by byte arrays.

	int	long	float	double	byte	char	short	reference
?2c	i2c							
?2d	i2d	l2d	f2d					
?2i		l2i	f2i	d2i				
?2f	i2f	l2f		d2f				
?2l	i2l		f2l	d2l				
?2s	i2s							
?add	iadd	ladd	fadd	dadd				
?aload	iaload	laload	faload	daload	baload	caload	saload	aaload
?and	iand	land						
?astore	iastore	lastore	fastore	dastore	bastore	castore	sastore	aastore
?cmp		lcmp						
?cmp{gll}			fcmp{gll}	dcmp{gll}				
?const_<n>	iconst_<n>	lconst_<n>	fconst_<n>	dconst_<n>			sconst_<n>	
?div	idiv	ldiv	fdiv	ddiv				
?inc	iinc							
?push					bpush		spush	
?load	iload	lload	fload	dload				
?mul	imul	lmul	fmul	dmul				
?neg	ineg	lneg	fneg	dneg				
?newarray								anewarray
?or	ior	lor						
?rem	irem	lrem	frem	drem				
?return	ireturn	lreturn	freturn	dreturn				areturn
?shl	ishl	lshl						
?shr	ishr	lshr						
?store	istore	lstore	fstore	dstore				astore
?sub	isub	lsub	fsub	dsub				
?throw								athrow
?ushr	iushr	lushr						
?xor	ixor	lxor						

Table 3. The Providing Types of JVM Opcodes

JVM instruction set is not orthogonal. In other words, operations provided for one data type are not necessarily provided for other data types. This lack of orthogonality is because each instruction is 8-bit opcode, so there are not enough opcode to offer the same support to all java's runtime types. The providing types of JVM opcodes are listed in Table 3. [9]

The instructions of JVM are variable-length, and they depend on the instructions. We can categorize the instructions of JVM into 9 groups. The following paragraphs describe them briefly. [4]

1. Load and Store

This group of instructions is responsible for the data movement between the operand stack and local variable area. Besides, there are instructions for loading constants onto the operand stack.

2. Arithmetic

Type specific arithmetic instructions are supported by JVM instruction set as mentioned in the previous paragraph. We can see that there is no arithmetic instruction for byte, short and char types. If we want to do arithmetic operations, we should first cast them to int types and use integer arithmetic instructions to perform what we want to do.

3. Type Conversion

JVM provides several instructions to do numeric data type conversion. These instructions can be divided into two categories. One is widening the data length. For example, the *i2l* instruction converts 4-byte integer to 8-byte integer. The other category is narrowing the data length. The *l2i* instruction acts like that.

4. Object Creation and Manipulation

This group of instructions deals with object-related operations. For example, create class instances, create array objects, access object variables, access array elements, and check object types.

5. Operand Stack Manipulation

As mentioned before, JVM is a stack-based machine, so there are instructions for manipulating the data in operand stack. This instruction group includes push, pop, duplication of top element, and swap of top two elements instructions.

6. Flow Control

Except conditional branch and unconditional branch instructions, JVM also provides two compound conditional instructions: *tableswitch* and *lookupswitch*. These two compound conditional instructions are used to choose an address out of a list of addresses according to specific conditions.

7. Exception

Java provides exception handling mechanism. An exception is occurred by *athrow* instructions thrown by JVM.

8. Synchronization

Because Java is a multi-threaded programming language, there are synchronization problems. Two instructions, *monitorenter* and *monitorexit*, are provided to support method-level and block-level synchronization.

9. Method Invocation and Return

JVM provides 4 different Method invocation instructions: *invokevirtual*, *invokestatic*, *invokeinterface* and *invokespecial*, and 6 different method return instructions: *return*, *ireturn*, *lreturn*, *freturn*, *dreturn*, and *areturn*.

2.3. Implementations of JVM

JVM is the key point to platform-independence. Once there is an implementation of JVM on a platform, all Java programs can be run on this platform without any recompilation. So there is a slogan of Java technology: Write Once, Run Anywhere.

Fig 19 shows the four kinds of implementations. The first kind is Interpreter, and this is also the original version implementation of JVM. Then the second kind is Just-In-Time (JIT) compiler. The technology using dynamic compiler is called HotSpot. Finally, the fourth

kind is Java processor, which is also our basic target implementation. We will introduce these four implementations in the following subsections.

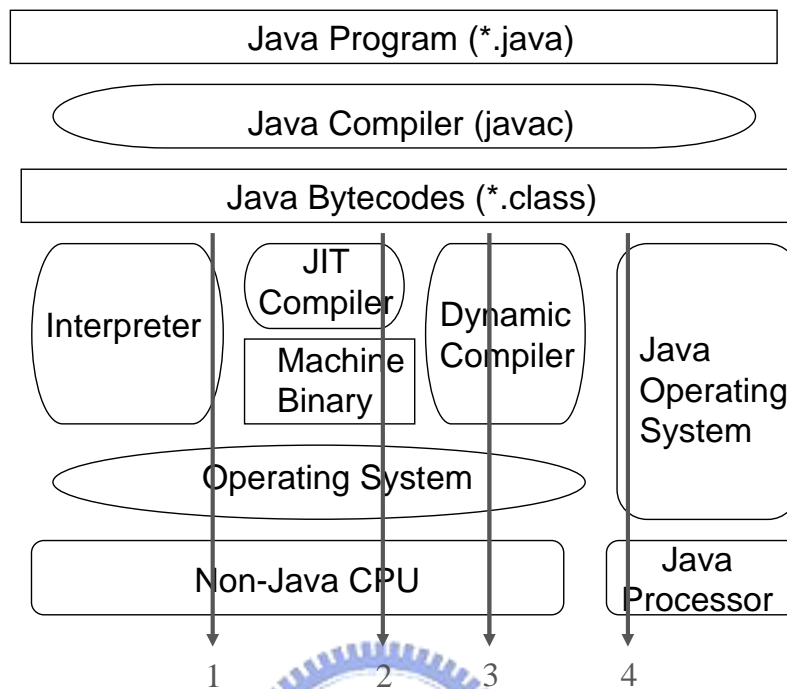


Fig 19. Implementations of JVM

2.3.1. Interpreter

The first JVM implementation is interpreter, which includes a big loop in that every instruction is read and executed in order. Fig 20 illustrates the flowchart of the interpreter. This kind of implementation is very simple, but it suffers from inefficiency. Consider a loop code section. If this loop executes 100 times, this code should be interpreted 100 times and executed 100 times. Compared with fully compiled codes, 99 out of 100 interpretations are actually overhead.

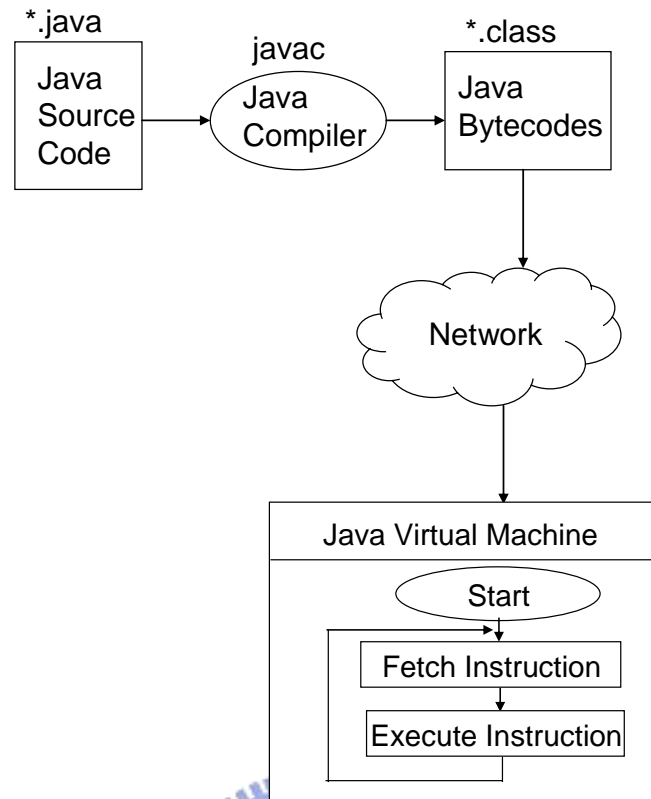


Fig 20: Interpreter

2.3.2. Just-In-Time (JIT) Compiler

Just-In-Time (JIT) compiler takes the bytecodes and compiles them into native code for the machine that you are running on before the first time you execute it. This is shown in Fig 21. The native machine codes exist only in the memory. When the program terminates, the native machine codes are destroyed rather than restored for next execution. Compilation must be done for each execution to ensure that the Java bytecodes are portable. This is the key difference between JIT compiler execution and fully-compiled execution.

Because JIT compiler translates the whole programs into native machine codes before executing, it can do some optimization of the entire programs. A Java program usually runs 50 times faster on the JIT compiler than on the interpreter. [4] However, the start-up time of JIT is very long. They should wait for the whole program loaded and compiled. If some

optimization option is adjusted, the start-up time will even longer. Nevertheless, JIT may spend a lot of time on useless optimizations, such as the instruction that is only executed one time. Nowadays, Many research works study on this topic to make JIT more efficient.

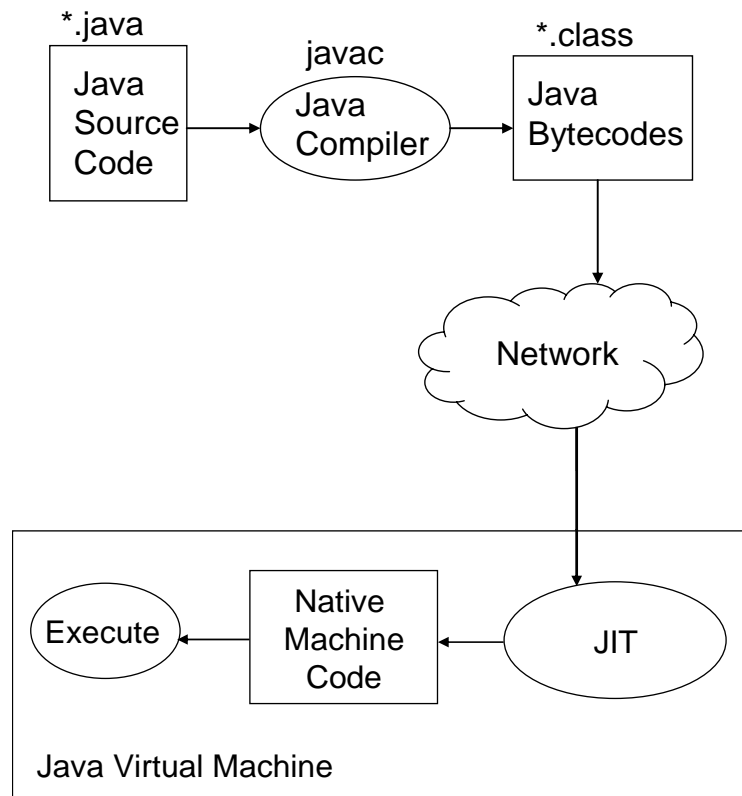


Fig 21. Just-In-Time Compiler

2.3.3. HotSpot technology

HotSpot is a dynamic compiler that integrates a compiler with an interpreter. The concept of dynamic compilation is based on the research done over the past 10 years at Stanford University and the University of Californiam Santa Barbara (UCSB).

Fig 22 illustrates the architecture of HotSpot. Java bytecodes are first loaded into HotSpot and executed by the interpreter. During the execution, the profiler keeps the runtime information and determines which method to be compiled into native machine codes and optimizes them. The control component links the other four together, and

provides the shared information. An apparent important function of the control component is that it must keep whether a method has a native version or not, and their addresses.

The dynamic compiler can perform some tasks to improve the performance of program execution that a normal static compiler can not perform. The first is optimistic compilation. Compilation during execution is very expensive, so we can choose which ones are needed to be compiled and others remained to the interpreter. (e.g. the code section that is executed only one time) By ignoring these cases, the dynamic compiler makes significant performance improvement with only a small investment in optimizing time. The second advantage is the run-time information can be taken into account for compilation and optimization. For object-oriented programs, runtime information is more useful than static information. The other advantage is that dynamic compilation can perform inlining the frequently-invoked methods according to the runtime information.

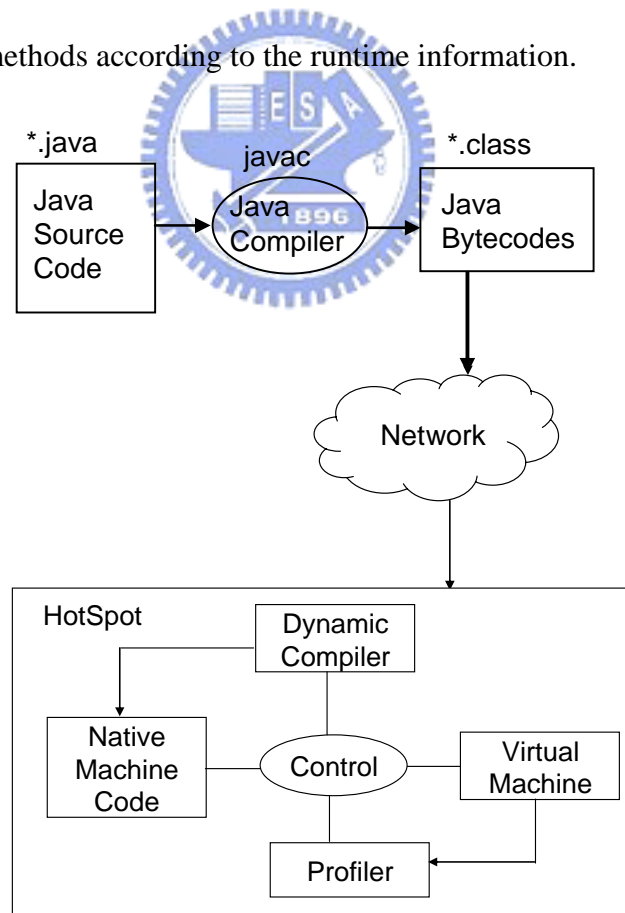


Fig 22. HotSpot

HotSpot is implemented in Java Virtual Machine by Sun in 1998. The details of the internal workings are not open to the programmers, but many experiments show that HotSpot has a great improvement of the execution efficiency of JVM.

2.3.4. Java Processor

Java Processors are primarily used in embedded system [14]. The native programming language of such systems is Java, and all operating system related code, such as devices drivers, are implemented in Java. Java processors are also stack-based machines with their own instruction set, which bytecode will translate to and be executed in directly. As a result, this pure hardware implementation has the best performance of the four. Fig 23 shows the flowchart of Java programs that execute on the Java processor.

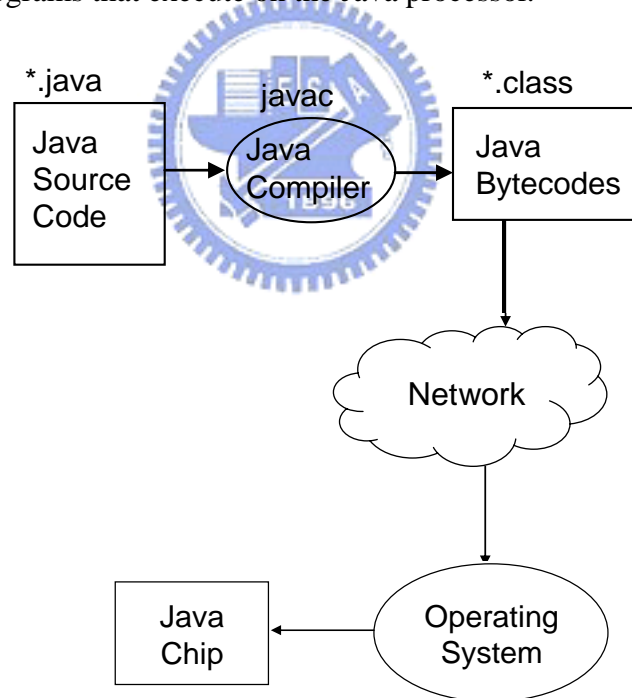


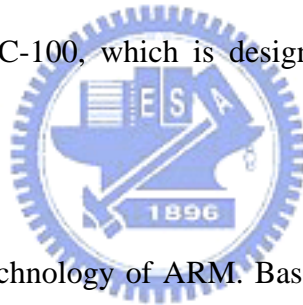
Fig 23. Java Processor

picoJava is the most well-known Java processor developed by Sun. It always serves as the reference for new Java processors and as the basic for research into improving various

aspects of a Java processor. The first version of picoJava is presented by Sun in 1997 [15]. This processor was targeting at the embedded systems market as a pure Java processor with restricted support of C programming language. PicoJava-I contains four pipeline stages. A redesign followed in 1999, known as picoJava-II. picoJava-II now is freely available with a rich set of documentation. In the following, we will briefly introduce four famous Java processors or Java chips.

1. Zucotto

Zucotto Wireless Inc. is a new company, which established in 1999. The target market of this company aims wireless communication. Their main product is Zucotto XC-100. Zucotto XC-100 implements the Garbage Collector into their hardware architecture, so they can manage the memory usage more powerfully. The lower power consumption is also the main advantage of Zucotto XC-100, which is designed a power saving mode that idle blocks can enter in.



2. ARM Jazelle

Jazelle is a Java Chip technology of ARM. Based on the RISC architecture, Jazelle executes bytecodes directly using the translated microcode sequences. Now it can support 95% bytecodes. Besides the two basic instruction set of ARM processor, ARM 32 bits instructions and ARM 16 bits Thumb instructions, Jazelle adds a third instruction set, that is Java Bytecode instruction set. These three can switch while needed. The software architecture of Jazelle chip is shown in Fig 24 (see [16]).

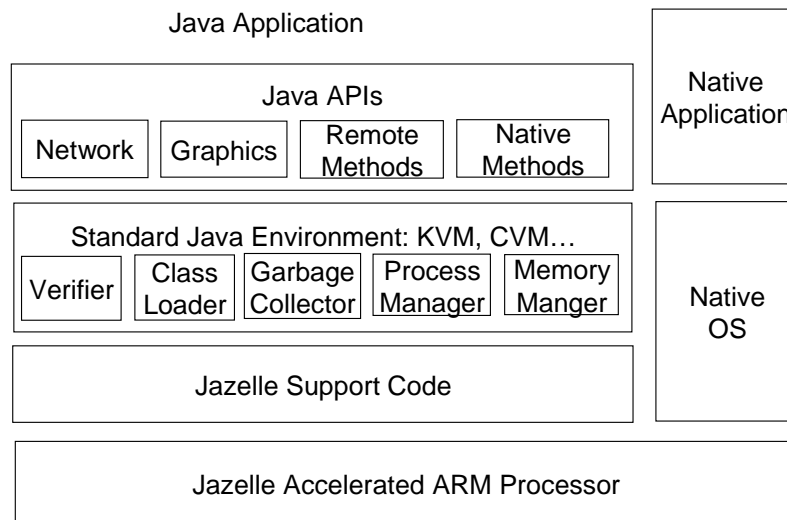
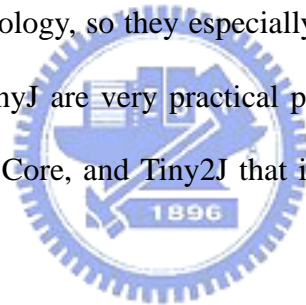


Fig 24. Software Architecture of Jazelle Chip

3. Tiny J

TinyJ is developed by Advancel Logic Corporation. The advantage of TinyJ is the support of cryptographic technology, so they especially suited the JavaCard and e-business devices. The derivations of TinyJ are very practical product, shch as TinyJDSP processor that integrates TinyJ and DSP Core, and Tiny2J that is designed for Java Card and Smart Card.



4. MOON

MOON is a Java specified chip developed by Vulcan Machines. The basic architecture of MOON is a traditional Von Neumann machine, so their instruction set and data are stored in the same address space. MOON core size is very small and it has a good performance. The only disadvantage of Moon is that the Garbage Collection must be implemented in software, which makes it weak than others.

3. Problem Formulation

In this chapter, we first formulate our problem. Then we will give an overview to the open source system, Java Optimized Processor (JOP), upon which our proposed system is built. The software layer stack, system architecture, datapath, and hardware/software co-design of JOP will be discussed in section 3.2.

3.1. Introduction

In order to make the DVB-MHP system stable and condensable, we choose JOP as our code base of Java VM in DVB-J functional block. JOP is an hardware/software co-design system, which we will state in the next section. As we mentioned before, although interpreter is the most suitable kind of embedded Java VM implementation due to its simplicity and low resource requirement, it has a big problem in efficiency. Typical dynamic code optimization (Sun's JVM RI described in subsection 2.1.4.1) can speed up the execution of this approach, but it suffers from the overhead of external memory access. Furthermore, it sometimes is dispensable because the modified codes will never be executed again.

In this research, we want to design a new hardware/software co-processing dynamic code optimization scheme that is more suitable for embedded system. Our goal is to make the Java VM more efficient and significantly cut down on the power consumption.

3.2. Java Optimized Processor - JOP

Java is seldom used in embedded systems. Actually, many features of Java, such as thread support in the language, could greatly simplify development of embedded systems. Based on this concept, Java optimized Processor (JOP), which is part of a Ph.D. thesis at the Technical University of Vienna in Austria, is developed by Martin Schoberl In Oct 2001. [14]

JOP is basically a hardware implementation of JVM with predictable execution time for embedded real-time systems. The goal of this development is a simple and small Java processor optimized to execute Java bytecodes. Due to the small size of this processor, it can be implemented in a low cost FPGA. The flexibility of an FPGA can be of more importance for low volume systems compared to conventional Java processors.

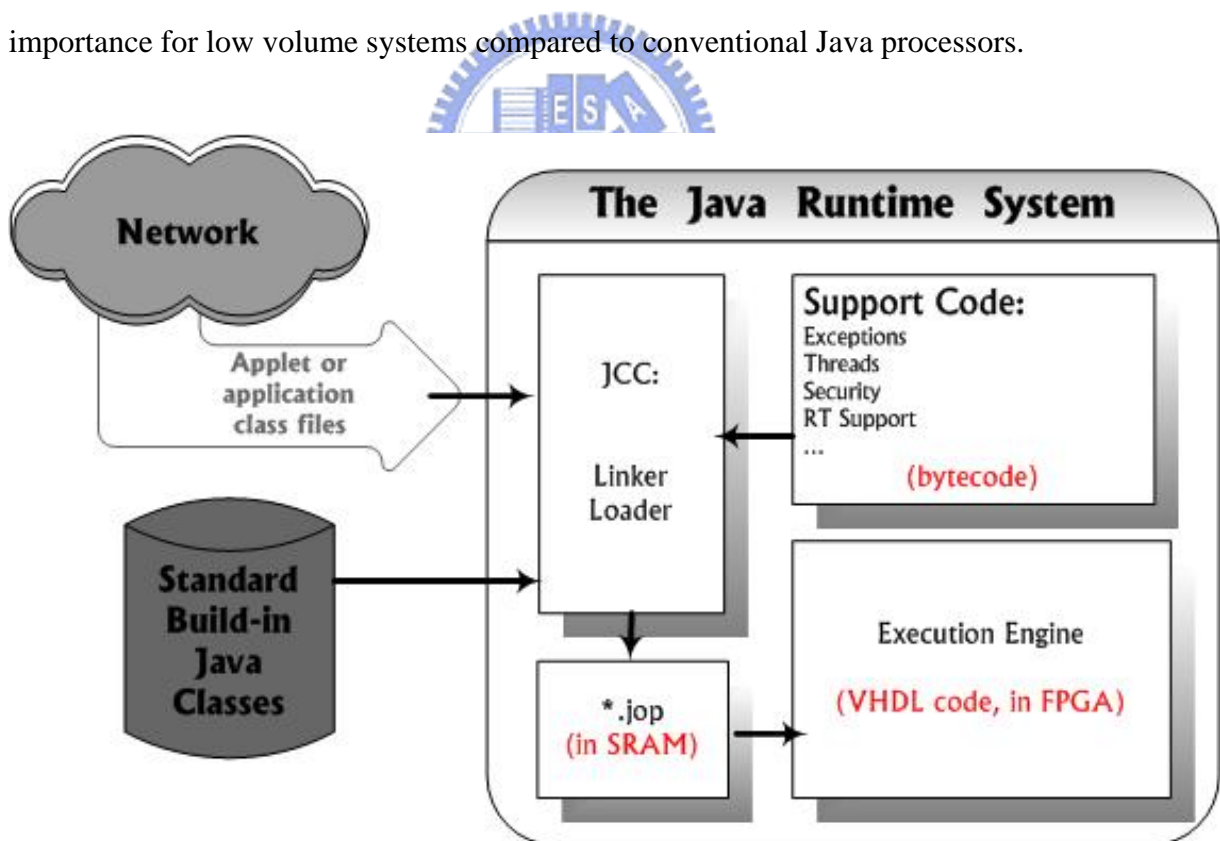


Fig 25. Java Optimized Processor Runtime Environment

JOP is one way to use a configurable Java processor in small embedded real-time

systems. It shall help to increase the acceptance of Java for these systems. However, it suffers from the restrictions of embedded systems. Because of the memory limitation and security concerns, JOP is compatible with the Java Virtual Machine but has following restrictions:

- No support for floating point data types (float and double).
- No support for the Java Native Interface (JNI).
- No user-defined, Java-level class loaders.
- No reflection features.
- No support for thread groups or daemon threads.
- No support for finalization of class instances.
- No weak references.
- Limitations on error handling.

The simplified Java Runtime System is illustrated in Fig 25. Compared to the typical Java Runtime Systems in Fig 18, we can see that there are no Java Native Interface and dynamic class loader and verifier support. Furthermore, garbage collection is not allowed because it is not suitable for such real-time systems.

The important step of executing Java programs on JOP is JavaCodeCompact (JCC), which is also known as the class prelinker, preloader or ROMizer. This utility allows Java classes to be linked directly in the JVM and reduces JVM startup time considerably. Bytecodes of Java programs and Java APIs including support codes that are used in this program do the JavaCodeCompact and output a file to be stored in external memory. Then execution engine starts to execute this program. [8]

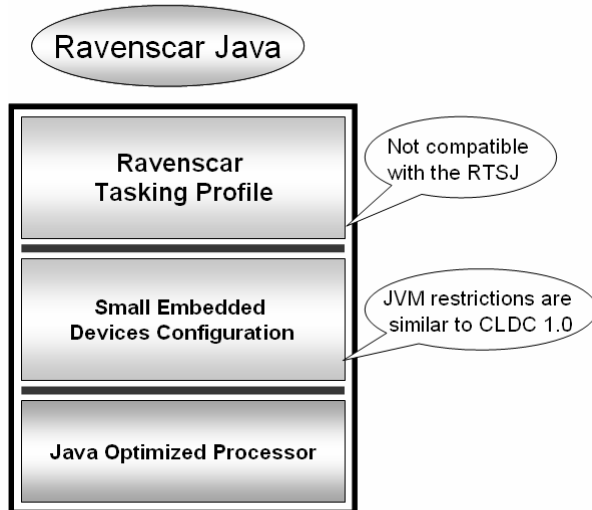
3.2.1. Software Layer Stack of JOP

In Java 2nd platform we mentioned in subsection 2.1.4.2, JOP is targeted at the Java 2 Micro Edition (J2ME). J2ME is a four-layered structure. Upon the operation system and Java Virtual Machine, configuration and profile are presented. A J2ME configuration defines a minimum platform for a “horizontal” category or grouping of devices, each with similar requirements on total memory budget and processing power. A configuration defines the Java language and virtual machine features and minimum class libraries that a device manufacturer or a content provider can expect to be available on all devices of the same category, such as Connected Device Configuration (CDC) and Connected, Limited Device Configuration (CLDC). On the other hand, a J2ME profile is layered on top of (and thus extends) a configuration. A profile addresses the specific demands of a certain “vertical” market segment or device family. The main goal of a profile is to guarantee interoperability within a certain vertical device family or domain by defining a standard Java platform for that market. Profiles typically include class libraries that are far more domain-specific than the class libraries provided in a configuration. The most famous profile that we know is MIDP (Mobile Information Device Profile). [8]

Due to the features of embedded system, JOP must have its own configuration and profile. Fig 26 digests the configuration and profile that JOP are compatible. Small Embedded Devices Configuration (SEDC) is intended for small embedded devices with a 16-bit (or even 8-bit) microprocessor and a low memory budget (below 128 kB). The JVM restrictions of SEDC are similar to CLDC 1.0 but smaller than. SEDC use JCC to simplify the application with preverified and preloaded mechanisms. Threads are not part of SEDC, and there are no stream input/output facilities. [17] Ravenscar Tasking Profile is designed in the concept of the ADA Ravenscar Profile [19]. It resembles the ideas from [18] and [20] but is not compatible with the RTSJ. This profile addresses the same devices as SEDC. Java

language run on this profile and configuration also called Ravenscar Java.

➤ JOP layer stack:



➤ J2ME software layer stack:

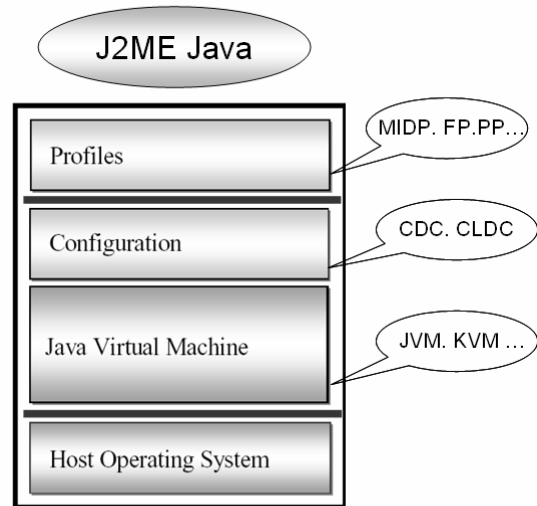


Fig 26. Software Layer Stack of JOP

3.2.2. System Architecture of JOP

A typical configuration of JOP contains the processor core, a memory interface, a number of IO devices, and the module extension which provides the link between processor core, memory and IO modules. Block diagram of JOP is illustrated in Fig 27 (see [14]).

The processor core contains four pipeline stages: bytecode fetch, microcode fetch, decode and execute, which we will discuss in next subsection. As we see, there is no direct connection between the processor core and the external world. The memory interface provides a connection between the main memory and the processor core. It also contains the bytecode cache, which caches the whole method code of one method. The I/O interface controls peripheral devices, such as the system time, the timer interrupt, a serial interface and application-specific devices.

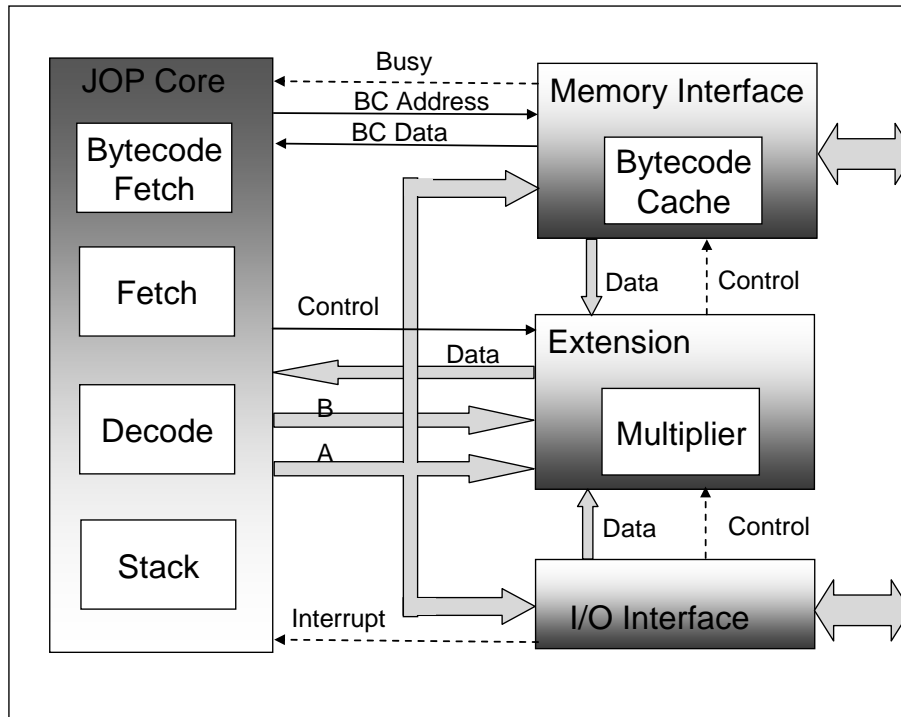


Fig 27. Block Diagram of JOP

The division of this processor into those four modules greatly simplifies the adaptation of JOP for different application domains or hardware platforms. For example, in order to port JOP to a different FPGA device, one only needs to modify the memory module alone, but not the processor core.

3.2.3. Datapath of JOP

In previous subsection, we said that JOP uses a four-stage pipeline architecture and every instruction in JOP is exactly executed in one single cycle. Look at Fig 28 (see [14]). Bytecode is fetched in order by pc register, and then looks it up in the jump table to get the start address of the translated microcode sequence. This is done in the first stage. In the second pipeline stage, JOP uses the start address to jump to the corresponding microcode. It is decoded and executed until the *next* instruction in next two pipeline stages.

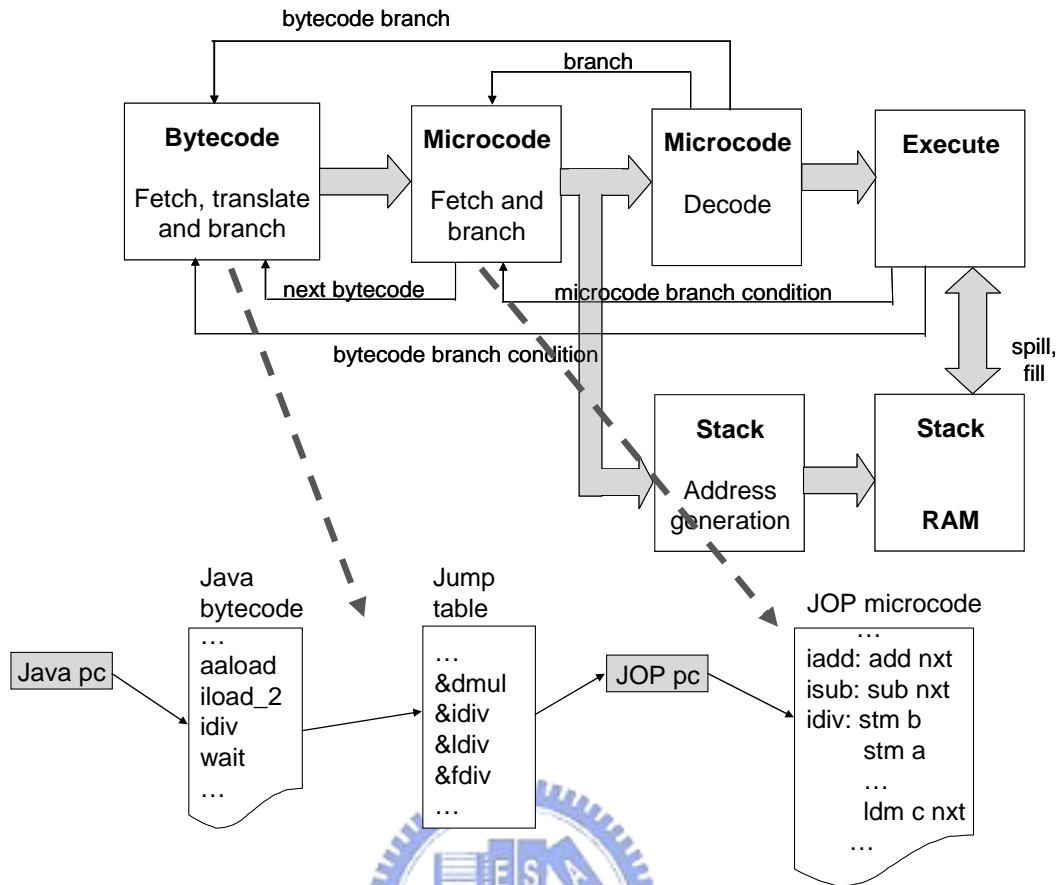


Fig 28. Datapath and Data Flow of JOP

This stack architecture allows for a short pipeline, which results in short branch delays. Two branch delay slots are available after a conditional microcode branch. All the needed memory while execution, such as the method cache (bytecode cache), microcode ROM, and stack RAM, are implemented with single cycle access in the FPGA's internal memories. [14]

3.2.4. Hardware/Software Co-design of JOP

JOP is a hardware/software co-design Java processor. Moving functions between hardware and software is very easy, and this feature is resulting in a highly configurable design. If the execution speed is the important issue, more functions are realized in

hardware; if the cost is the primary concern, these functions are moved to software and a smaller FPGA can be used.

There are three implementations of bytecodes. They can be VHDL code implementation, microcode implementation, and Java code implementation. Bytecodes that are not implemented in VHDL or microcode result in a static Java method call from a special class. The additional overhead for this implementation is a call and return with method cache refills. A comparison of resource usage and execution time for the three implementations of *imul* is listed in Table 4. We can see that the implementation in Java is slower than the microcode implementation and consequently VHDL implementation as the Java method is loaded from main memory into the bytecode cache.

	Hardware (LC)	Microcode (Byte)	Time (cycle)
VHDL	156	10	35
Microcode	0	73	750
Java	0	0	2300

Table 4. A Comparison of Different Implementations of *imul*

4. Proposed Dynamic Code Optimization System

Due to the demand of efficiency in DVB-MHP applications, we need to further improve the performance of the JOP system. By analyzing the execution frequency, we observed an important feature and use it to design our new dynamic code optimization scheme.

In this chapter, we first discuss the data structure using our framework. Then we analyze the bytecode execution frequency and give an overview to our scheme. Finally, the hardware and software modules of our design are respectively illustrated.

4.1. Data Structure Using in Our Dynamic Code Optimization

In this section, the data structure using dynamic code optimization is given. These include the data arrangement in the external memory, method cache and each of the runtime data structure.

4.1.1. Data Arrangement in the External Memory

The application programs are compiled into Java class files by the Java compiler (javac), with all the linked library programs recompiled, and then passed to JavaCodeCompact (JCC).

In conventional class loading, javac is used to compile Java source files into Java class files, which are loaded into a Java system, either individually, or as part of a jar archive file. Upon demand, the class loading mechanisms resolve references to other class

definitions. JCC provides an alternative means of program linking and symbol resolution. First the multiple input class files will be combined, and JCC will determine the layout and size of an object instance. Only the designated class members will be loaded and linked with the Java Virtual Machine in order to reduce JVM's bandwidth and memory requirements. Resolution of symbols is also performed in this stage, which reduces the start-up time of JVM.

The output of JCC is a C file and its format can be arranged by the user-defined writer. In JOP system, the writer is redesigned to have JCC output a data layout file like the data arrangement in the external memory (SRAM in Spartan-3) and loaded it directly to the external memory. An illustration of it is shown in Fig 29.

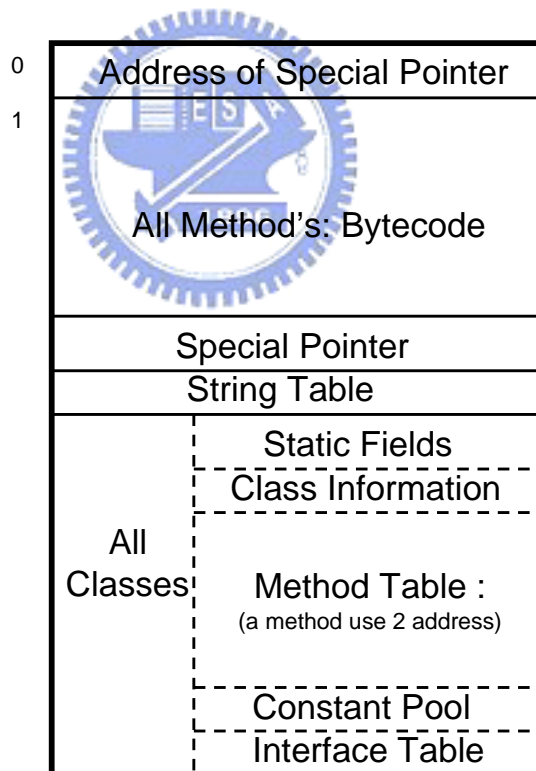


Fig 29. Data Arrangement in the External Memory

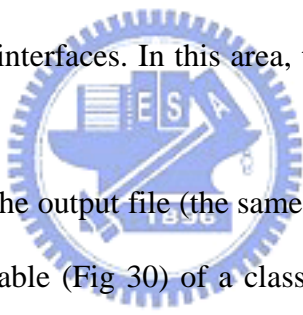
All of data in this output file are united in 32 bits of an address. This means that the address 0 has 32 bits data, and the 33rd bit is the first bit in address 1. After collected all the

designated method bytecodes, JCC has the bytecode size in 32-bits. The JOPWriter writes this size added one in the first address, and then all the designated method bytecodes. Finished all the writing of bytecodes, the next writing address must be the data saved in address 0, because it is the size of bytecodes added one.

Then we save four special pointers: a pointer to boot code, a pointer to first non-object method structure of class JVM, a pointer to first non-object method structure of class JVMHelp, and a pointer to main method structure. We can easily get special pointers by using the data in address 0, because it is also the address of first special pointer. For example, the data in address 0 adds three is the address of main method structure.

The next area is the string table area, followed by the all-class data area. The all-class data area contains the static fields, class information, method table, constant pool, and interface table if this class has interfaces. In this area, the data related to all the classes are listed one after another.

All of the information in the output file (the same as in external memory) will be used while execution. The method table (Fig 30) of a class is the key data structure to get the address to other class information. Note that a method table occupies two address space, and an address is 32 bits.



Start Address	Method Length	
Constant Pool	Local Count	Arg. Count
0	22	27 31

Fig 30. Method Table Structure

The highest 10 bits in the first address of method table are the length of method bytecodes with 32 bits a unit. By shifting right 10 bits of the first address we can get the

method bytecodes' start address that points to the second block in Fig 29. The start address has 22 bits and it is in Big-Endian byte order. The second address stores the constant pool pointer in 22 bits, the number of local variables in 5 bits, and the number of arguments in 5 bits.

4.1.2. Method Cache

Method cache is also called bytecode cache which we had mentioned in subsection 3.2.2. Because the fetch of external memory is very expensive, the concept of method cache is created in JOP. During one external memory fetch, the whole bytecodes of one executing method are fetched and loaded to the method cache, which is usually a memory area synthesized on FPGA. The external memory fetch time can be smaller than fetching one address a time. For example, assume that we fetch one address in external memory takes 3 microseconds ($= 10^{-6}$ seconds). We will spend 30 microseconds if we want to fetch a method with 10 units (32 bits a unit) address bytecodes. However, if we fetch all bytecodes of that method (10 units address) one time, we may just spend 22 microseconds in fetching external memory.

Method cache is designed to cache just one method bytecodes. Consider this example program [14]:

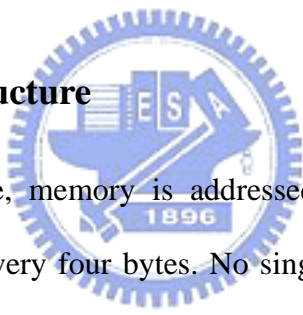
```
Foo () {  
    A();  
    B();  
}
```

We will have the following cache loads:

1. method *Foo* is load on invocation of *Foo()*
2. method *A* is load on invocation of *A()*
3. method *Foo* is load on return from *A()*
4. method *B* is load on invocation of *B()*
5. method *foo* is load on return from *B()*

It should refill the method after returned from its internal method. This is the main drawback of the method cache. But by that we can almost make sure that the method cache will reload when executing the same method next time. As a result, we do not need to reflash the method table when we modified the executing method bytecodes in our dynamic code optimization scheme. This also saves much time in doing optimization.

4.1.3. Runtime Data Structure



As we mentioned before, memory is addressed as 32 bits data, so the memory pointers are incremented for every four bytes. No single or 16 bits access is necessary in our JOP system. The reference data type is a point to memory that represents the object or an array, which is pushed on the stack before an instruction operating on it. A null reference is represented by the value 0 [14].

In the following we are going to see each runtime data structure.

4.1.3.1. Stack Frame

First we look into the stack frame. On a method invocation, the information of the invoker is saved in a newly allocated frame on the stack. It is restored when the method returns. The information consists of five registers: SP (Stack Pointer), PC (Program Counter), VP (Variable Pointer), CP (Constant Pool Pointer), and MP (Method Table

Pointer).

SP, PC and VP are registers in JOP while CP and MP are local variables of JVM. Fig 31 (see [14]) provides an example of the stack change before and after invoking a method. The caller has two arguments and the called method has two local variables. The arguments that we want to pass into the invocated method can be accessed in the same way as local variables. As in this example, the arguments *arg_0* and *arg_1* will become *var_0* and *var_1* with the original *var_0* and *var_1* shifted to *var_2* and *var_3*. The start address of the frame can be calculated with the information from the method table:

$$\text{Frame address} = \text{VP} + \text{Arg. Count} + \text{Local Count}$$

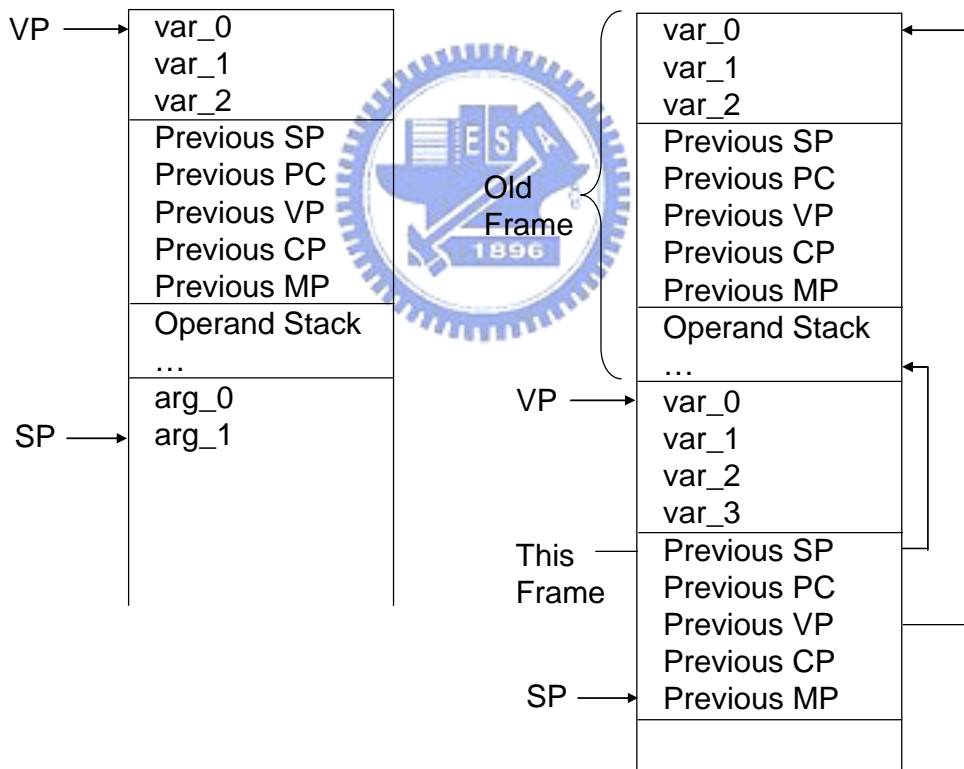


Fig 31. Stack Change on Method Invocation

4.1.3.2. Data Layout

In JOP, objects are stored in memory during runtime in the Fig 32 (see [14]) format. Note that the object reference points directly to the first reference of the object to speedup. We can access the class information pointer by object reference subtracted one.

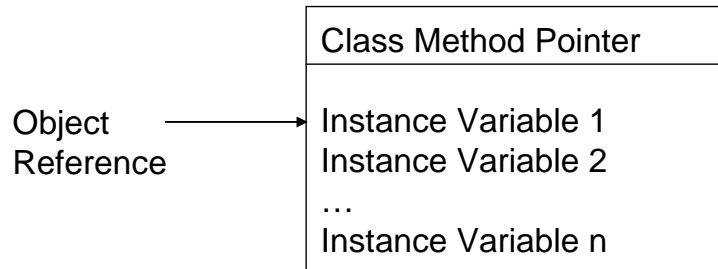


Fig 32. Object Format

The array layout in memory is just like an object. We showed the array format in Fig 33 (see [14]). Also, if we want to access the array length, just take object reference subtracted one.

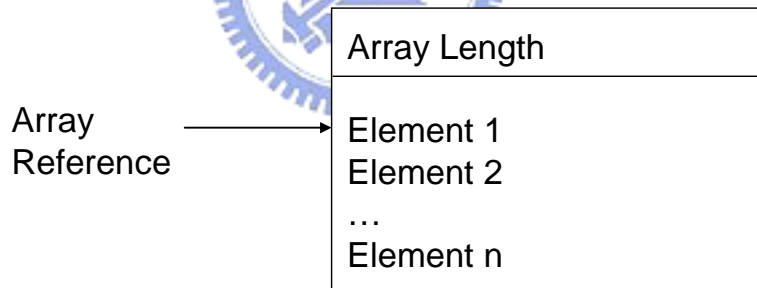


Fig 33. Array Format

4.1.3.3. Runtime Class Structure

The runtime class structure of JOP is shown in Fig 34 which had discussed in 4.1.1 as all classes' information. This class structure is stored in the external memory. For indicating the pointers in previous data structure, we drew this class structure again with pointers Class Reference, Class Method Pointer, MP, and CP.

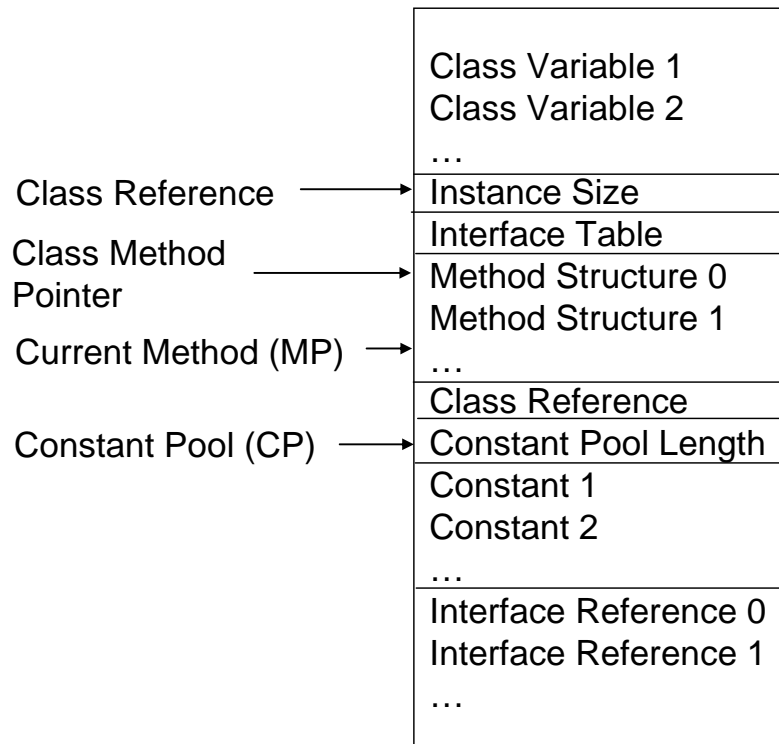
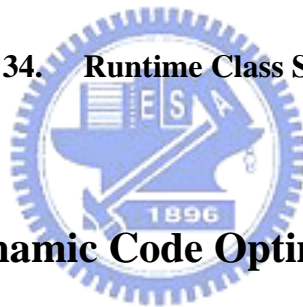


Fig 34. Runtime Class Structure



4.2. The Proposed Dynamic Code Optimization Scheme

In this section, we propose our dynamic code optimization scheme. First we analyze the bytecode execution frequency, from which we get the new idea of improvement. Then we compare the access time of the external memory and the internal memory. By reducing the number of dynamic code modifications that do not improve the performance, we can make the system more efficient. The architecture overview is illustrated in the last subsection.

4.2.1. Analysis of Bytecode Execution Frequency

In subsection 2.3.3, we have mentioned that Hotspot uses optimistic compilation

which can dynamically choose which instructions needs to be compiled and the rest are executed by the interpreter. The decision is based on the execution frequency. This concept is used in many systems. For example, the famous code morphing processor from Transmeta also uses execution frequency to decide whether the code is to be interpreted or to translated. As Fig 35 (see [22]), the translation threshold is decided by the code execution frequency. When the number of executions of a section of x86 machine code reaches a certain threshold, its address is passed to the translator.

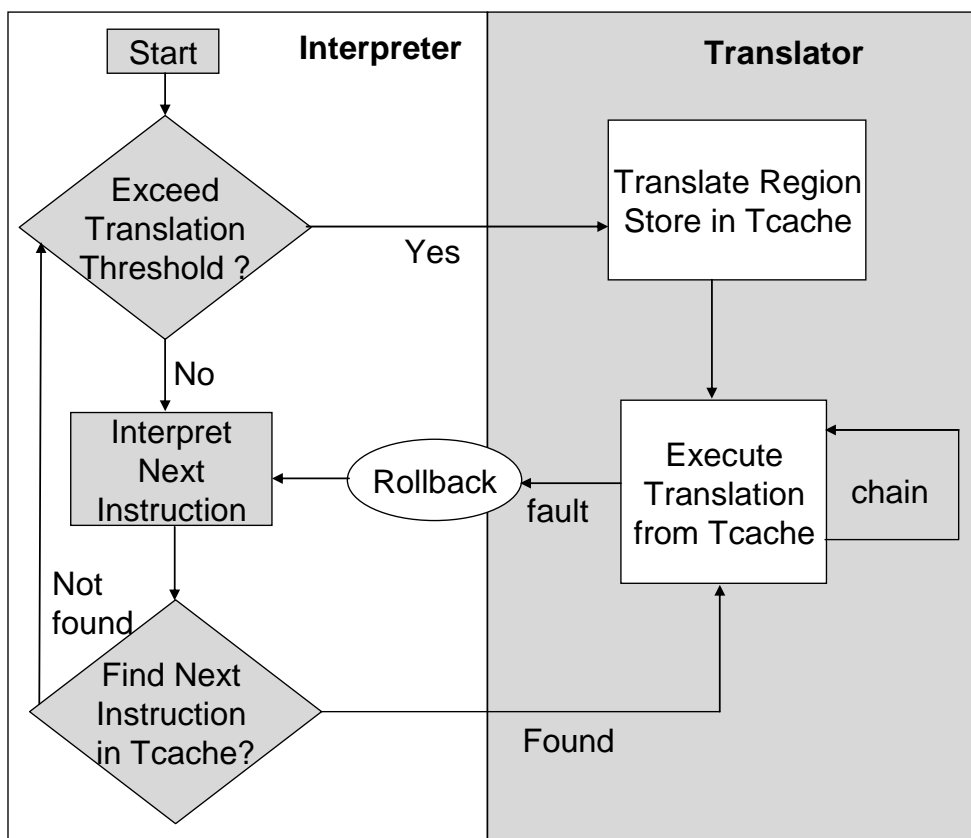


Fig 35. Transmeta Code Morphing Software Control Flow

We analyze the bytecode execution frequency using the three benchmark programs described in section 5.2. The distribution of bytecode execution frequency is listed in Table 5 and Fig 36 shows the diagram. The number of bytecodes is counted under the given execution frequency. Consider the following analysis data. When executing the UDP/IP

program, there are 385 bytecodes that are executed 9 times and 210 bytecodes are executed 13 times. For the same bytecodes (e.g. aload_0), they are different in different methods or sequences.

benchmark frequency	Sieve	Kfl	UDP/IP
1	487	267	292
2	1	2	0
3	0	5	3
4	24	0	1
5	15	13	8
6	129	65	72
7	283	142	69
8	145	259	141
9	382	284	385
10	425	378	287
11	622	241	342
12	319	389	356
13	386	165	210
14	245	121	368
15	77	242	224
16	164	86	165
17	231	69	82
18	89	32	114
19	54	44	32
20	32	3	46
21	2	0	15
22	17	2	3
23	0	0	0
24	1	0	2
25	2	0	0
all	4132	2809	3217

Table 5. The Number of Bytecodes under the Given Execution Frequency.

Look at the curves in Fig 36. We observe a very important rule. The bytecodes are almost executed exactly once or much more than twice. This observation is the critical point in our design.

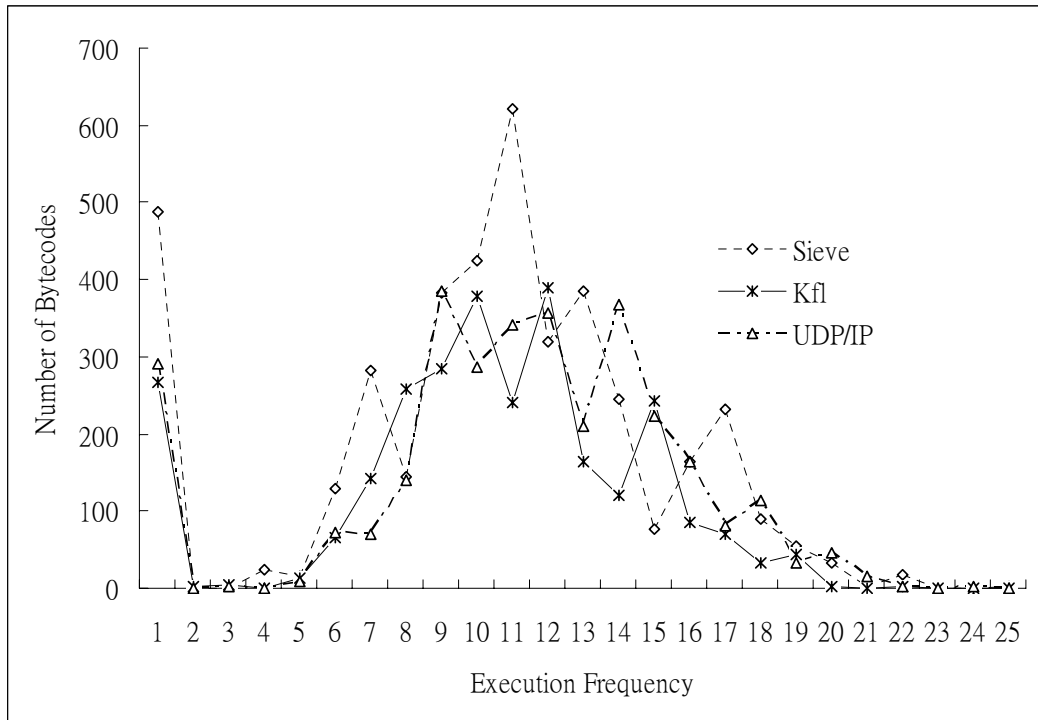


Fig 36. Distribution of Bytecode Execution Frequency

4.2.2. Access Time of External Memory & Internal Memory

As we mentioned before, typical dynamic code optimization (Sun’s JVM RI described in subsection 2.1.4.1) can speed up the execution of embedded Java VM, but it suffers from the overhead of external memory accesses.

Consider the JOP system. The clock frequency of both FPGA and SRAM is 50MHz, so the clock time is calculated as following.

$$\frac{1}{50M} = \frac{1}{50} \times 10^{-6} = 0.02 \times 10^{-6} = 2 \times 10^{-8} \text{ seconds}$$

The internal memory access only needs 1 cycle. But if it is the external memory, it needs 5 cycles for memory read and 7 cycles for memory write on JOP because JOP is designed for various developing boards. The microcode sequence of external memory read is shown in Fig 37.

```
stmra
nop
wait
wait
ldmrd
```

Fig 37. Microcode Sequence of External Memory Read

Upon execution of a memory read, the address is stored and the processor waits for the value to arrive and then pushed the value to the top of the operand stack as in Fig 31. Each microcode executes in a single cycle, so the external memory read needs 5 cycles. For the microcode sequence of external memory write shown in Fig 38, it needs 7 cycles.

```
stmwa
nop
stmwd
nop
wait
wait
nop
```

Fig 38. Microcode Sequence of External Memory Write

As a result, if we can reduce the number of dynamic code modifications that do not give us any advantages, e.g. the codes that are exactly executed once, we can make a big improvement of execution time and cut down the power consumption. In next subsection we are going to introduce the design of our dynamic code optimization module.

4.2.3. Architecture Overview

In subsection 4.2.1, we knew that bytecodes are almost executed exactly one time or much more than two times. Then in subsection 4.2.2, we analyzed the memory access time, and found that the access time of external memory is a big overhead of the traditional dynamic code optimization scheme. Based on these two observations, we designed the new dynamic code optimization architecture called JDCO.

To speed up the execution and cut down the power consumption, we only modify the codes when it is necessary. That is, if the code is executed exactly one time, we do not do the dynamic code optimization – constructing a new bytecode to replace the original bytecode and storing the field or method offset in the operand of new bytecode. Because the method bytecodes are stored in external memory in most embedded system and also our JOP system (described in subsection 4.1.1), this new module can execute the Java programs with dynamic code optimization in a more efficient way.

However, if the execution frequency can not be determined upon the first encounter of a bytecode (unless we do a “fast-forward” to check whether the bytecode will be executed again, which has unacceptable overhead). Another possible way is to perform a pre-pass counting of the execution of the bytecodes, but this is also very expensive. We proposed a simple algorithm that reduces unnecessary modifications with very low overhead. The proposal is as follows. A small memory is synthesized in the FPGA to count the number of execution of each bytecode during execution. For the first execution, no dynamic code modification is performed. The DCO is only done at the second time the code is executed, because we assume that it will be executed again and again base on the observation of subsection 4.2.1. For third execution and above, we can directly use the operand of new

bytecode to speed up the performance and cut down the power consumption.

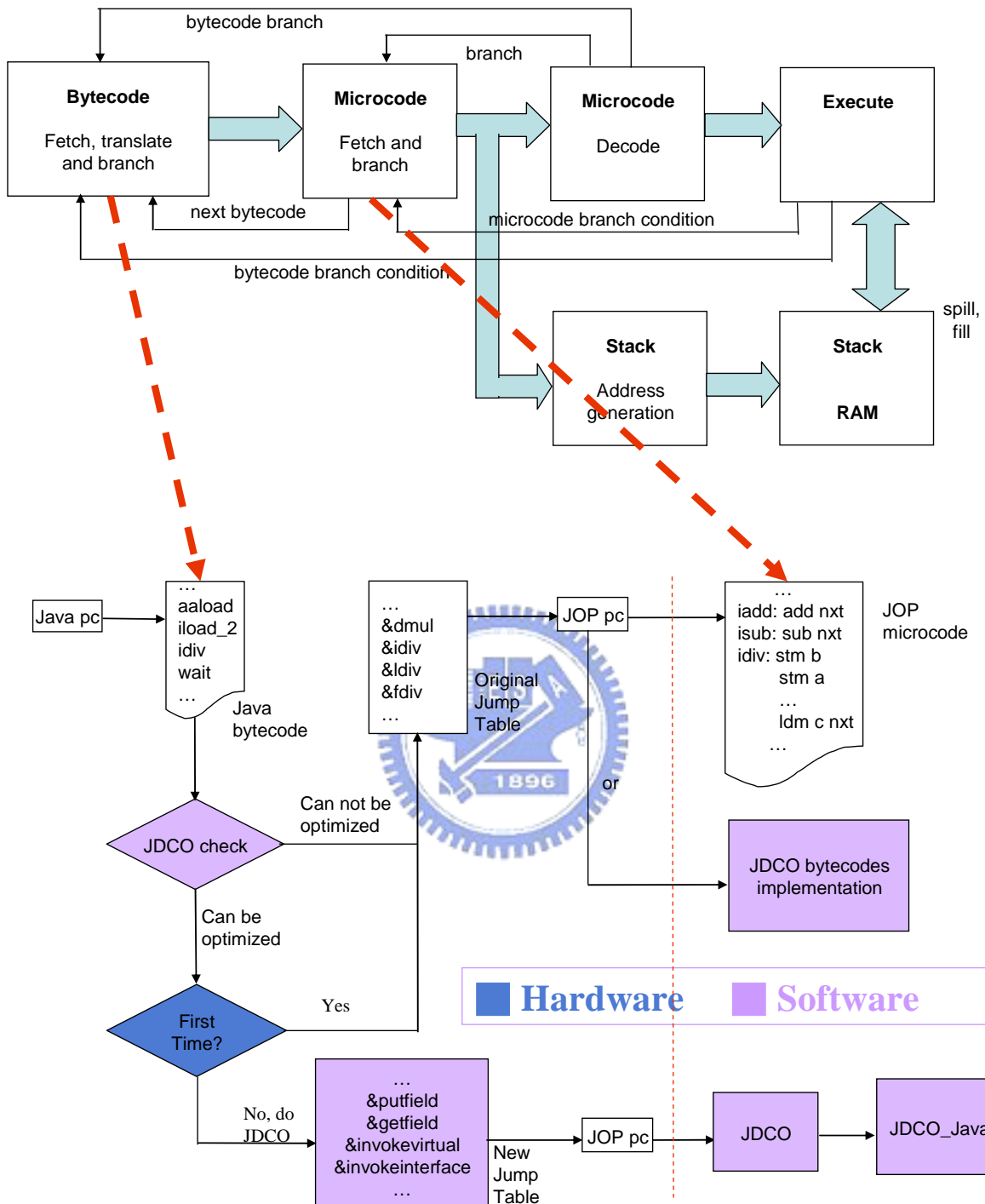
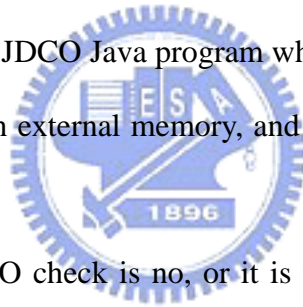


Fig 39. Our JDCO Architecture Overview

The flowchart of our JDCO architecture is shown in Fig 39. We mark our new modules in colored background with distinguishing hardware and software implementation

modules. In the beginning of the first stage, bytecode fetch, a bytecode is pointed by Java pc to be executed. The bytecode will pass to a JDCO check module, which will check if this bytecode can be optimized or not. For example, if the bytecode has the information that can be recorded for speeding up the next execution (e.g. getfield. putfield. etc.), we say that it can be optimized. If the answer of JDCO check is yes, our system will further check if it is the first time to execute this bytecode to decide whether we should perform DCO or not. If it is not the first time of execution, the new JDCO optimization will look up the bytecode in our new jump table to get the JOP pc, which points to our new JDCO module in the second stage, microcode fetch. JDCO will execute this bytecode and get the runtime information depending on the specific bytecode. It may be the offset of an object field, or of the class method that will not change when next time we execute the same bytecode. The runtime information will be passed to a JDCO Java program which will construct a new bytecode to replace the original bytecode in external memory, and store the runtime information in the operand of this new bytecode.



If the answer of the JDCO check is no, or it is yes but this is only the first time of execution of the byte code, our architecture will follow the original procedure. Looking up in the jump table, the JOP pc is retrieved for execution. The corresponding bytecode implementation is executed whether it is a newly implemented bytecode that we constructed or not. The implementation of the bytecode may be the VHDL implementation, microcode implementation, or Java Code implementation.

4.3. Implementation Details

In this section, we are going to look into more details of the implementation. The description is divided into two parts: hardware implementation modules and software

implementation modules, which distinguished in Fig 39.

4.3.1. Hardware Implementation Modules

In our design, a hardware module is needed for first time of execution checking. We need to synthesis a small on-chip memory that can count the execution times of each bytecodes, and then decide to do the original bytecode implementation or the JDCO module.

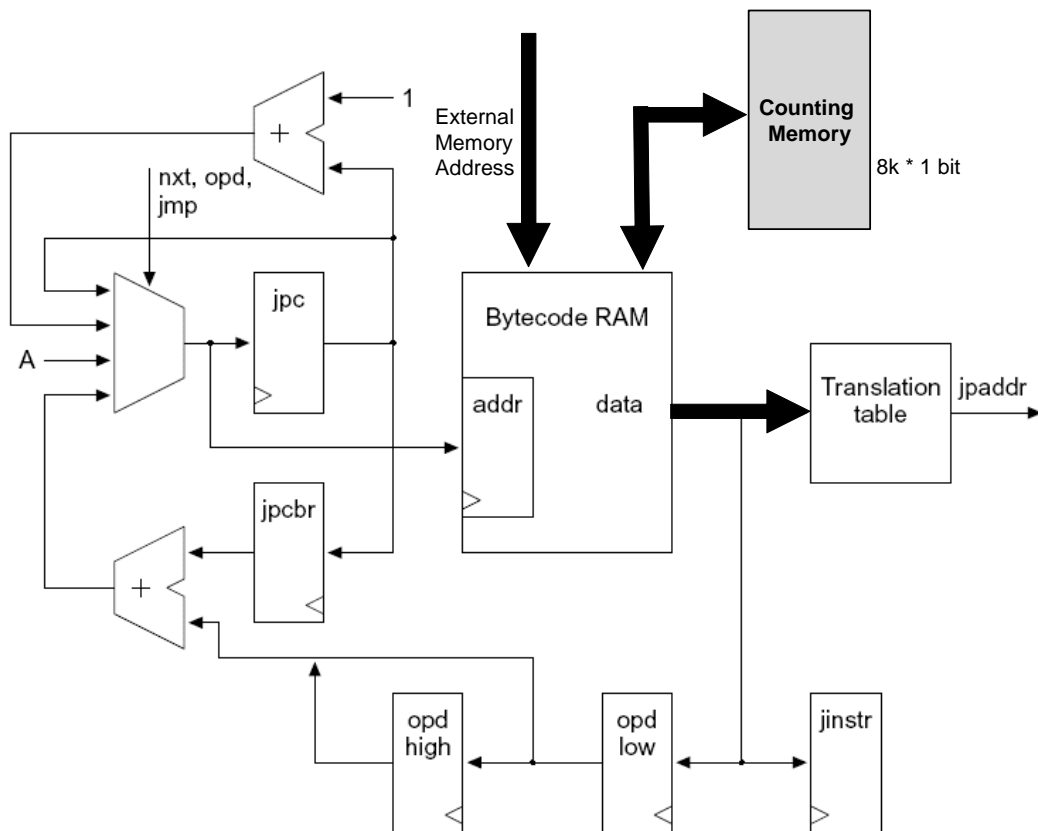


Fig 40. Java Bytecode Fetch Stage of Our JDCO

We have mentioned in subsection 3.2.3 that JOP has four pipeline stages. In the first pipeline stage as in Fig 40, the Java bytecodes are fetched from the internal memory

(Bytecode RAM). The bytecode is mapped through the translation table into the address (*jpaddr*) for the microcode RAM in next stage.

We synthesize an $8K * 1$ bit memory called *Counting Memory*, in which one bit map to an address of method bytecode in external memory (see Fig 29). When Java bytecodes are fetched from the internal memory, we use its start address of method as an index to see if the bit in *Counting Memory* is set or not. If it is set, we know that it is the second time executed. Then the address of the modified bytecode implementation (e.g. *putfield_modify* in next subsection) is mapped through the translation table and passed to next stage. If the bit is zero, then the address of original bytecode (e.g. *putfield*) is mapped and passed. Finally the corresponding bit in *Counting Memory* is set.

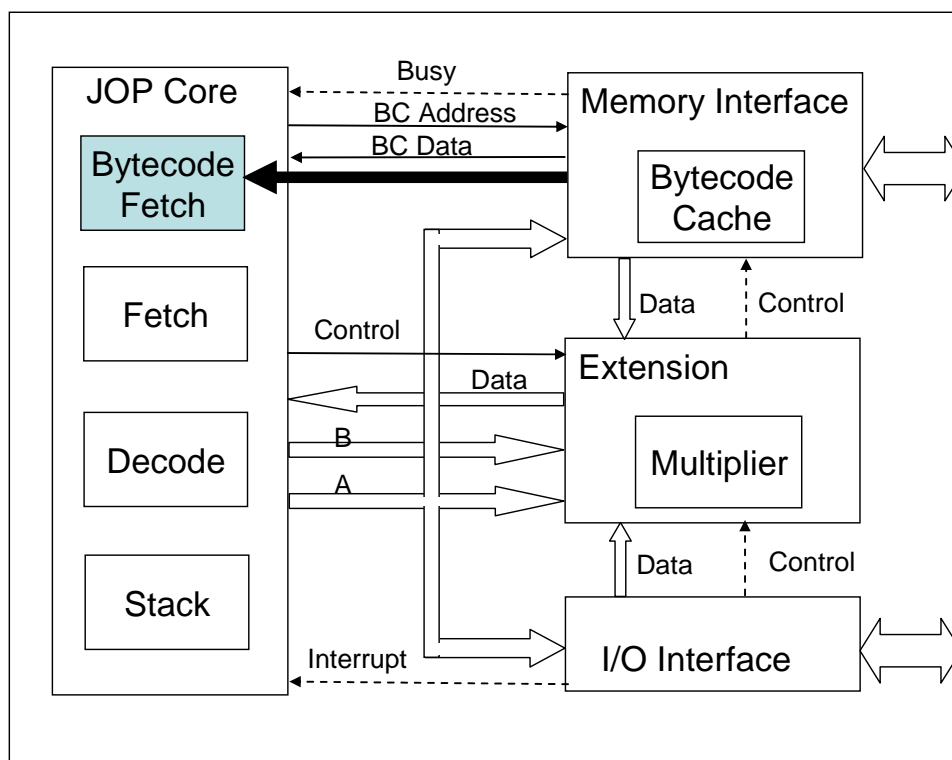


Fig 41. Block Diagram of The Proposed JDCO

The block diagram of JOP has been shown in Fig 27. The main modification is in the Bytecode Fetch stage of JOP core. But we do not have the method start address of external

memory because method bytecodes are fetched from the bytecode cache missing the original address in external memory. So we need to map this port (external memory address) from memory interface to JOP core, and map to the Bytecode Fetch stage. We summarize our modification and redraw this as in Fig 41.

4.3.2. Software Implementation Modules

In bytecode level, first we should figure out what bytecodes are needed to do our JDCO. Based on Sun's JVM Reference Implementation, we may have 25 bytecodes (Table 1) that can be considered. But most operands of them are not modified. The new bytecodes of these bytecodes just indicate that they have been resolved. In section 3.2, we have introduced that all bytecodes are passed to JavaCodeCompact (JCC) first, and then the output is loaded into the external memory in JOP system. In other words, all method bytecodes in external memory have been resolved, and the DCO is not useable for these. As a result, we only have four bytecodes needed to do JDCO: getfield (180), putfield (181), invokevirtual (182) and invokeinterface (185).

To fulfill our JDCO modules, two bytecodes need to be constructed for one bytecode without changing the instruction length. We list the new bytecodes of our architecture with their format in Table 6.

Upon the first time of executions, we will executed the original bytecodes. Modified bytecodes are used in the second executions and above, and replaced itself in the new bytecodes. The offset of object field or class method will be stored in the operand of the new bytecodes for next execution.

	bytecode	format				
Original Bytecodes (not changed)	180	getfield	indexbyte1	indexbyte2		
	181	putfield	indexbyte1	indexbyte2		
	182	invokevirtual	indexbyte1	indexbyte2		
	185	invokeinterface	indexbyte1	indexbyte2	nargs	0
Our JDCO Bytecodes	228	getfield_modify	indexbyte1	indexbyte2		
	229	putfield_modify	indexbyte1	indexbyte2		
	230	invokevirtual_modify	indexbyte1	indexbyte2		
	231	invokeinterface_modify	indexbyte1	indexbyte2	nargs	0
	233	getfield_new	offserbyte1	offsetbyte2		
	234	putfield_new	offserbyte1	offsetbyte2		
	235	invokevirtual_new	offserbyte1	offsetbyte2		
	236	invokeinterface_new	offserbyte1	offsetbyte2	nargs	0

Table 6. Our Designed JDCO Bytecodes & Their Formats



5. Performance Study

In this chapter, we first introduce our development environment – Xilinx Spartan-3 Developing Board, and then we state the Java benchmark used in this research. Finally, the experiment results are shown and discussed. We analyze the performance on both execution time and power consumption.

5.1. Xilinx Spartan-3 Developing Board

The Xilinx Spartan-3 Developing Board is used for the development of the proposed Java VM accelerating algorithm. The top side and bottom side of the board are shown in Fig 42 and Fig 43 (these figures are taken directly from the user guide. [21]).

The equivalent gate counts of the target Spartan-3 device are 200,000 gates, and the logic utilization of JOP on the FPGA is 64 percent. The data path of Spartan-3 is 32 bits with an 8-bit memory interface. Shift instruction can be computed in exactly one single cycle. The external memory devices of JOP on Spartan-3 is a 32-bit SRAM block of 1M bytes and an 8-bit flash of 2M bits. Java program is compacted by JCC to *.jop file which is loaded into SRAM. Configuration data is stored in flash. Finally, the maximum working frequency of this processor is 194.621 MHz, according to the synthesizer.

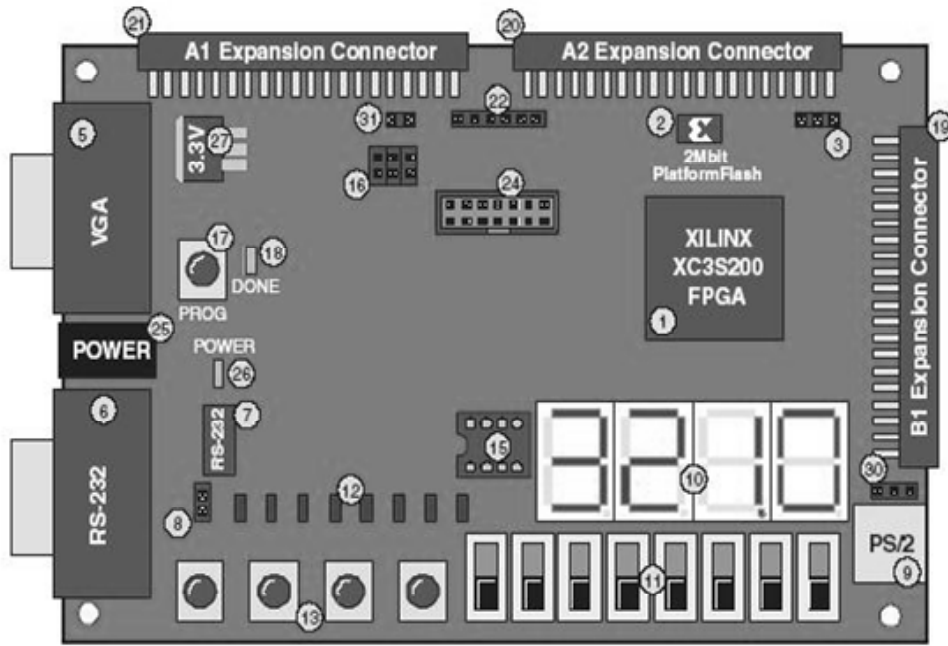


Fig 42. The Top Side of Xilinx Spartan-3

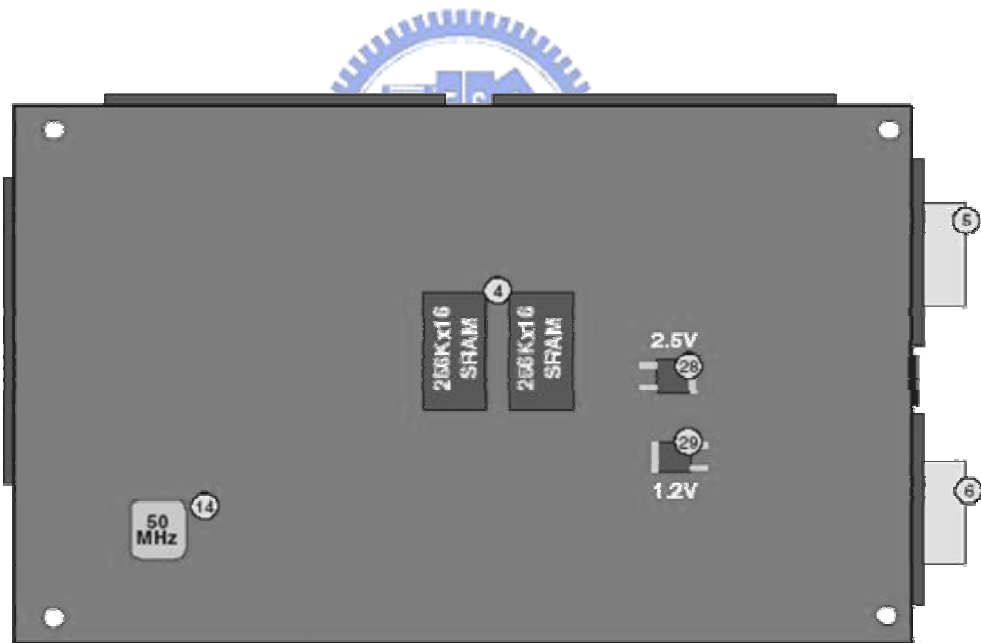


Fig 43. The Bottom Side of Xilinx Spartan-3

5.2. Java Benchmark Programs

In this research, we use three small Java benchmark programs, which contain a

synthetic benchmark (Sieve of Eratosthenes) and two application benchmarks, Kfl and UDP/IP. [14] We describe them in the following subsection.

5.2.1. Sieve of Eratosthenes

This program will produce a list of prime numbers. The algorithm is proposed by Eratosthenes. His method is as following. First, write down a list of integers. Then mark all multiples of 2. The next step is, move to the next unmarked number, in here is 3, and mark all its multiples. Continue to mark all multiples of the next unmarked number until there are no new unmarked numbers. The numbers which survive from this marking process (the Sieve of Eratosthenes) are primes.

5.2.2. Kfl



Kfl is adopted from a real-time application which is taken from one of the nodes of a distributed motor control system. The motor control system is a solution to rail cargo. During loading and unloading goods from wagons, a large amount of time is spent due to the obstacle of contact wires. Balfour Beatty Austria developed and patented a technical solution called *Kippfahrleitung* to tilt up the contact wire. An asynchrony motor on each mast is used for this titling. However, it has to be done synchronously on the whole line. [23]

Each motor is controlled by an embedded system. This system also measures the position and communications with a base station. We show the mast with the motor and the control system in down and up positions in Fig 44 (see [14]). The base station need to control the deviation of individual positions during the tilt. It also includes the user interface for the operator. In technical term, this is a distributed, embedded real-time control

system, communication over an RS 485 network.

A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-time workload.

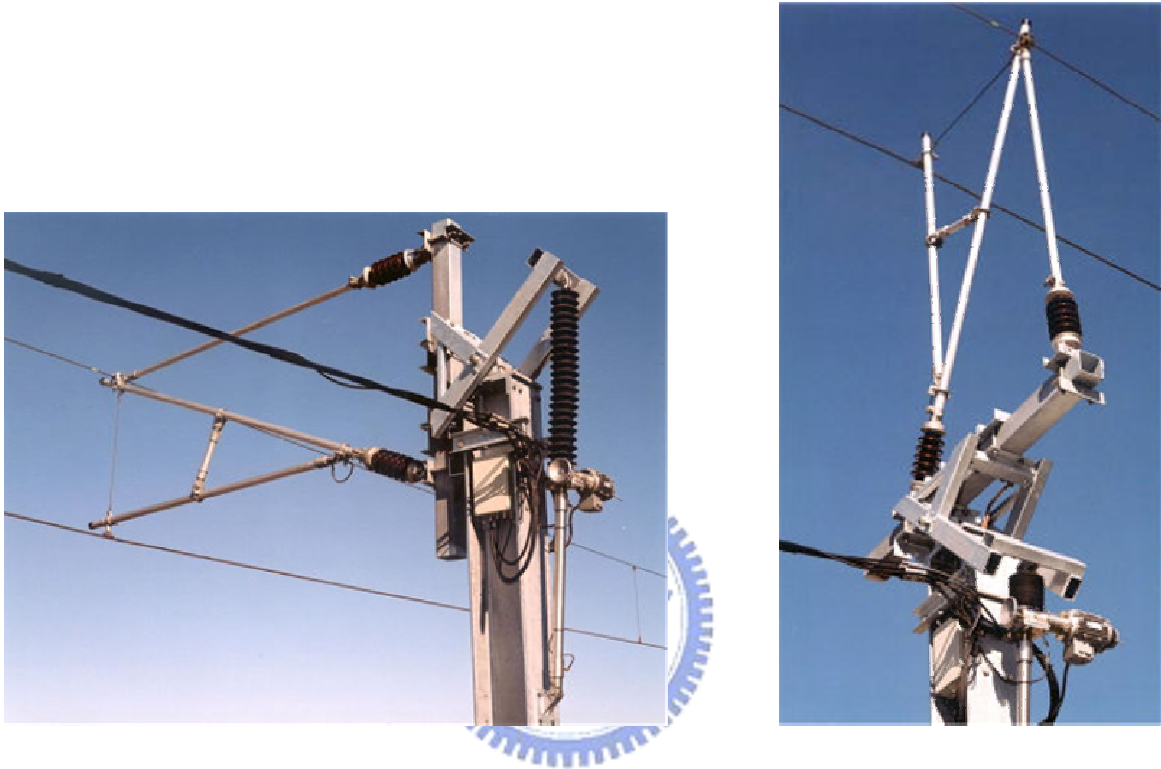


Fig 44. Pictures of a Kippfahrleitung Mast in Down and Up Position

5.2.3. UDP/IP

UDP/IP benchmark is composed of a tiny TCP/IP stack (Ejip) for embedded Java. This benchmark contains two UDP server/clients, exchanging message via a loopback device.

5.3. Experiment Results

We simulated our dynamic code optimization scheme on Spartan-3. The percentage of logic utilization increment is less than 1%, but we have made a big improvement in both execution time and power consumption. Now we are going to discuss in these two aspects.

5.3.1. Execution Time

We synthesize our JDCO system with comparisons to DCO (no frequency check) and the original JOP system. The execution time is listed in Table 7 and shown in Fig 45. In the table, we can see that the average speedup of our system is 13.8%, and compare to DCO system, we also have 7.1% execution time speedup.

Considering the execution time of each benchmark, we find an interesting phenomena. Let us focus on the results of UDP/IP benchmark. In our JDCO system, it has 9.7% speedup compared to DCO system, while other two benchmarks only have 6.0% and 5.6% speedup. The reason is that the UDP/IP benchmark has many initialization and executed-only-once code, so our JDCO system can make a big improvement by avoid that cases. Actually, the performance of this system is dependent on the Java program behavior.

system benchmark	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	11813	11043	10382	0.935	0.879	0.940
Kfl	2719	2419	2284	0.890	0.840	0.944
UDP/IP	4813	4627	4179	0.961	0.868	0.903
average	6448.3	6029.7	5615.0	0.929	0.862	0.929

Unit: millisecond

Table 7. Execution Time



Fig 45. Execution Time

5.3.2. Power consumption

To estimate the power consumption savings, we can analyze the microcode execution cycles and the external memory access times. We discuss the two aspects in the following subsections.



5.3.2.1. Microcode Execution Cycles

As we know that the less microcode execution cycles, the less power consumption will be. We analyze the microcode execution cycles of each bytecode and separate them by the number occurrences. The analyzed data is in listed in Table 8.

Because we have different microcode execution cycles in different number of occurrences, we should know the total execution times of the modified bytecodes of each benchmark separating by the number of occurrences, which is listed in Table 9. But these are the sum of the four modified bytecodes (putfield, getfield, invokevirtual, and invokeinterface), we should know the percentages of each of them. By analyzing the benchmark programs, we assume the percentages of the bytecodes as following:

$$180 : 181 : 182 : 185 = 40 : 20 : 20 : 1$$

# occurrences bytecodes	For JDCO			For DCO	
	first	second	third and later	first	second and later
getfield	20	33	7	33	7
putfield	23	36	10	36	10
invokevirtual	106	119	98	119	98
invokeinterface	118	131	110	131	110

Unit: cycles

Table 8. Microcode Execution Cycles of Each Bytecode

We can calculate the microcode execution cycles by the following formulation:

$$\sum_{\# \text{ occurrence}} (T * (\sum_{\text{ bytecode}} (\text{cycles} * P)))$$

T is the execution times in Table 9, and P is the percentage of bytecodes. For example, P of **getfield** is $40 / (40+20+20+1)$. The principle of this formulation is to calculate the sum of the execution cycles multiply the execution times. The execution cycles are calculated according to the percentage of each bytecode. Note that the microcode execution cycles of original JOP are always the same as the first time of JDCO.

# occurrences bytecodes	For JDCO			For DCO		For JOP
	first	second	third and later	first	second and later	all
Sieve	1874	1592	17462	1874	19054	20928
Kfl	1129	1001	12384	1129	13385	14514
UDP/IP	1342	1128	14287	1342	15415	16757

Table 9. Execution Times of Bytecodes 180. 181. 182. 185

We still calculate the execution cycles of our JDCO system with comparison to DCO and original JOP system. The experimental results are listed in Table 10 and shown in Fig

46. Because we only calculate on the modified bytecodes, we need to know the percentage of them of all bytecodes. By analyzing the benchmark programs, we get that the roughly percentage is 1/2. That is, $(180 + 181 + 182 + 185) \div \text{all} = \frac{1}{2}$.

As in Table 10, our JDCO has average 20.8% less execution cycles for the modified bytecodes, so for the all bytecodes, we have 10.4% less execution cycles than the original system. However, our JDCO has a little more microcode execution cycles than DCO system. This can be easily explained. By comparing between our JDCO and DCO system, we have less execution cycles for the executed-only-once bytecodes, but the needless first time overhead is happened to all the other bytecodes.

benchmark \ system	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	903779.6	705139.2	720105.5	0.780	0.797	1.021
Kfl	626789.8	484812.7	494864.1	0.773	0.790	1.021
UDP/IP	723654.1	560687.6	571107.3	0.775	0.789	1.019
average	751407.8	583546.5	595359.0	0.776	0.792	1.020

Unit: cycles

Table 10. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185

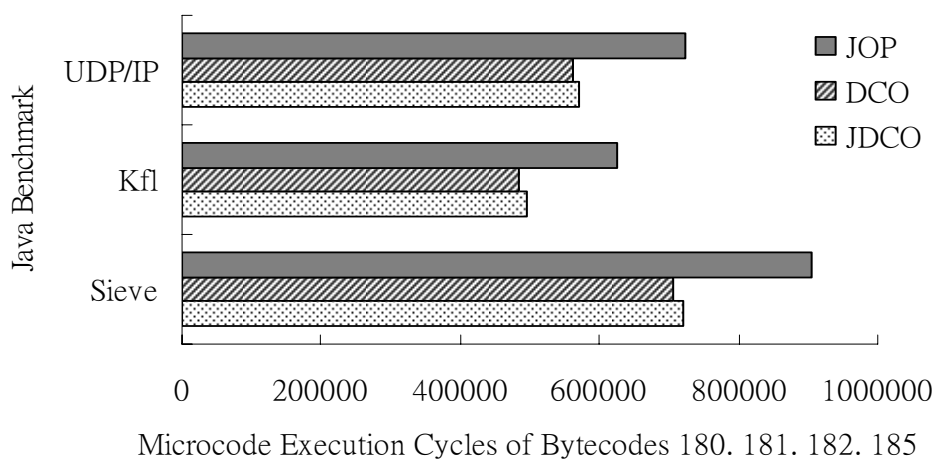


Fig 46. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185

5.3.2.2. External Memory Access Times

In addition to the microcode execution cycles, there is another important factor of power consumption. That is the external memory access times. Like the microcode execution cycles, the less external memory accesses, the more power consumption saving.

The calculation is similar to the microcode execution cycles. We also list the external memory access times of each bytecode and separate them by the number occurrences as in Table 11, in which we calculate the sum of memory read and memory write. The times 3 or 5 is based on the number of address we modified because an address is of 32 bits. For example, if the address of modified bytecode is “42 1 2 181”, we should modify the next address because it contains the operand of bytecode 181. For calculating, we use the average 4. Use this information and the total execution times of the modified bytecodes of each benchmark separating by the number occurrences in Table 9, we can calculate the external memory access times by the following formulation:

$$\sum_{\# \text{ occurrence}} (T * (\sum_{\text{bytecode}} (\text{times} * P)))$$

# occurrences bytecodes	For JDCO			For DCO	
	first	second	third and later	first	second and later
getfield	2	2+3/5	1	2+3/5	1
putfield	2	2+3/5	1	2+3/5	1
invokevirtual	4	4+3/5	3	4+3/5	3
invokeinterface	6	6+3/5	5	6+3/5	5

Unit: times

Table 11. External Memory Access Times of Each Bytecode

The experiment results are listed in Table 12 and showed in Fig 47. Our JDCO system has 22.2% less external memory access times of the modified bytecodes, so for the system of total bytecodes, we have 11.1 % less external memory access. If comparing to DCO

system, we still have a little more external memory access times. The reason is as we mentioned in the previous subsection.

system benchmark	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	53224.3	41666.3	42130.3	0.783	0.792	1.011
Kfl	36912.2	28043.1	28532.1	0.760	0.773	1.017
UDP/IP	42616.6	32569.6	32841.6	0.764	0.771	1.008
average	44251.0	34093.0	34501.3	0.769	0.778	1.012

Unit: times

Table 12. External Memory Access Times of Bytecodes 180. 181. 182. 185

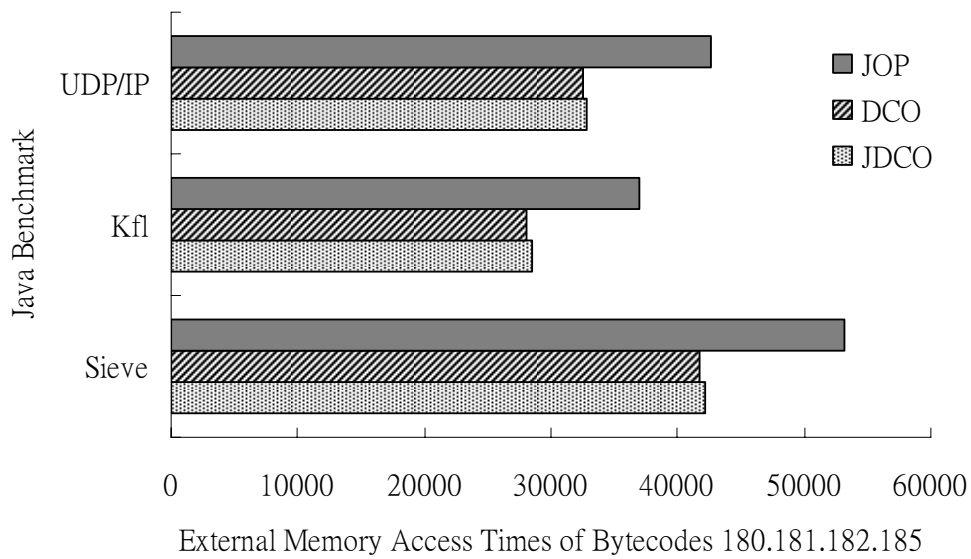


Fig 47. External Memory Access Times of Bytecodes 180. 181. 182. 185

6. Conclusion and Future Work

In this thesis, we propose a dynamic code optimization scheme which can significantly improve the efficiency of Java program execution and cut down on the power consumption for a hardware/software co-designed Java VM. As we mentioned above, typical dynamic code optimization can save method lookup and constant pool searching time using the runtime information at the first time a bytecode is executed. However, in the embedded system such as DVB-MHP terminal, code modification and saving of the runtime information is very expensive due to the overhead of external memory accesses. By analyzing the execution frequency of Java code segment, we can dynamically decide if the dynamic code optimization is needed. This JDCO architecture can make Java execution more efficient and more suitable to the DVB-MHP terminal due to less power consumption.

We implement this architecture based on the Java Optimized Processor (JOP) and verified the design on a Xilinx Spartan-3 development board. It is shown by our experimental results that the proposed dynamic code optimization scheme for Java VM hardware/software co-design has 13.8% average speedup of execution time. Furthermore, the power consumption of the proposed system can be reduced due to 10.4% less microcode execution cycles and 11.1% less external memory accesses compared to the original system.

Future researches can improve on recognizing the pattern of the relationship between frequency code and non-frequency code (maybe can learn from HotSpot). By doing this, the overhead of needless first time searching as describe in subsection 5.3.2 can be avoided. It may give a great improvement in power consumption. Then the format of other bytecodes will be designed and implemented for target systems that do not use JCC. In the future, the proposed system will be port to other more powerful developing board, such as the Xilinx

ML 310. It can be expected to have better performance for the Java VM.



REFERENCES

- [1] David Ungar and David Patterson, “Berkeley Smalltalk: Who Knows Where the Time Goes? ,” In *Smalltalk-80: Bits of History, Words of advice*, Addison-Wesley, Reading, MA, 1983.
- [2] Peter Deutsch and Alan M. Schiffman, “Efficient implementation of the Smalltalk-80 system,” In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297-302, ACM Press, January 1984.
- [3] Urs Hölzle, Craig Chambers, and David Ungar, “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches,“ In *Proceeding America, editor, Proceedings ECOOP '91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.
- [4] Ming Chan, Lin, “Runtime Profiling and Analysis of Java Program Execution,” Master Thesis, Computer Science and Information Engineering, National Chiao-Tung University, Taiwan, June 1998.
- [5] Anders Dellian, “Dynamic Code Optimization for Statically Typed OO Languages in An Integrated Incremental System,” In *Proceeding of NWPER'94, Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1994.
- [6] David Ungar, “The Design and Evaluation of a High Performance Smalltalk System,” In *MIT Press*, Cambridge, MA, 1986.
- [7] Sun Microsystems Inc., “The Java Virtual Machine Specification,” [Online] Available: <http://java.sun.com>.
- [8] Sun Microsystems Inc, “The K Virtual Machine White Paper,” [Online] Available: <http://java.sun.com> , June 1999.
- [9] Jon Meyer and Troy Downing, “Java Virtual Machine,” published by *O'REILLY*, 2000.
- [10] DVB project, “Digital Video Broadcasting (DVB): Multimedia Home Platform (MHP)

- Specification 1.1.1,” [Online] Available: <http://www.mhp.org>, Jun 2003.
- [11] “The Unicode Standard: Worldwide Character Encoding, “ [Online] Available: <http://unicode.org>
- [12] “UCS Transformation Format 8 (UTF-8), “ [Online] Available: <http://www.stonehand.com/unicode/standard/wg2n1036.html>
- [13] Eric Armstrong, “HotSpot: A New Breed of Virtual Machine,“ [Online] Available: <http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html>, 1998.
- [14] Martin Schoberl, “JOP: A Java Optimized Processor for Embedded Real-Time Systems”, Vienna, Jan 2005.
- [15] J.Michael O’Connor and Marc Tremblay, “picoJava-I: The Java Virtual Machine in Hardware,” In *IEEE Micro*, 17(2):45–53, 1997.
- [16] ARM, “ARM Jazelle Technology,” [Online] Available: <http://www.arm.com/products/solutions/Jazelle.html>
- [17] M. Schoeberl, “Restrictions of Java for Embedded Real-Time Systems, “ In *Proceeding of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2004*, Austria, Vienna, May 2004.
- [18] P. Puschner and A.J. Wellings, “A Profile for High Integrity Real-Time Java Programs,” In *Proceeding of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001
- [19] A.Burns and B. Dibbing, “The Ravenscar Tasking Profile for High Integrity Real-Time Programs,” In *Proceeding of the 1998 annual ACM SIGAda international conference on Ada*, pp.1-6, Washington, USA, 2002.
- [20] J.Kwon, A. Wellings and S. King, “Ravenscar-Java: a High Integrity Profile for Real-Time Java,” In *Proceeding of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 131-140, Seattle, Washington, USA, 2002.
- [21] Xilinx, “Spartan-3 Starter Kit Board User Guide”, Jul 2004.

- [22] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson, “The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges,” In *Proceeding of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, San Francisco, California, 27-29 March 2003.
- [23] Martin Schoeberl, “Using a Java Optimized Processor in a Real World Application,” In *Proceeding of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.

