

國立交通大學

資訊科學與工程研究所

碩士論文

嵌入式爪哇硬體加速器中
位元碼與陣列資料預先抓取之研究



Prefetching for Bytecode and Array Data in Embedded Java
Hardware Accelerators

研究生：吳易叡

指導教授：單智君 博士

中華民國九十六年八月

嵌入式爪哇硬體加速器中位元碼與陣列資料預先抓取之研究
Prefetching for Bytecode and Array Data
in Embedded Java Hardware Accelerators


研究生：吳易叡

Student：Yi-Ruei Wu

指導教授：單智君

Advisor：Jean Jyh-Jiun Shann

國立交通大學
資訊科學與工程研究所
碩士論文



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

August 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月

國立交通大學

博碩士論文全文電子檔著作權授權書

(提供授權人裝訂於紙本論文書名頁之次頁用)

本授權書所授權之學位論文，為本人於國立交通大學 資訊科學與工程 所
系統設計 組， 九十五 學年度第 二 學期取得碩士學位之論文。

論文題目：嵌入式爪哇硬體加速器中位元碼與陣列資料預先抓取之研究

指導教授：單智君

同意 不同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：

| | |
|-----------------|--|
| 本校及台灣聯合大學系統區域網路 | <input checked="" type="checkbox"/> 中華民國 年 月 日公開 |
| 校外網際網路 | <input checked="" type="checkbox"/> 中華民國 年 月 日公開 |

授權人：吳易叡

親筆簽名：_____

中華民國 年 月 日

國立交通大學

博碩士紙本論文著作權授權書

(提供授權人裝訂於全文電子檔授權書之次頁用)

本授權書所授權之學位論文，為本人於國立交通大學 資訊科學與工程 所
系統設計 組， 九十五 學年度第 二 學期取得碩士學位之論文。

論文題目：嵌入式爪哇硬體加速器中位元碼與陣列資料預先抓取之研究

指導教授：單智君

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：_____，請將論文延至____年____月____日再公開。

授權人：吳易叡

親筆簽名：_____

中華民國 年 月 日

國家圖書館博碩士論文電子檔案上網授權書

ID:GT009455591

本授權書所授權之論文為授權人在國立交通大學資訊學院資訊科學與工程所系統設計組 九十五學年度第二學期取得碩士學位之論文。

論文題目：嵌入式爪哇硬體加速器中位元碼與陣列資料預先抓取之研究

指導教授：單智君

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：吳易叡

親筆簽名：_____

民國 年 月 日



1. 本授權書請以黑筆撰寫，並列印二份，其中一份影印裝訂於附錄三之二(博碩士紙本論文著作權授權書)之次頁；另一份於辦理離校時繳交給系所助理，由圖書館彙總寄交國家圖書館。

To my parents



嵌入式爪哇硬體加速器中 位元碼與陣列資料預先抓取之研究

學生：吳易叡

指導教授：單智君 博士

國立交通大學資訊科學與工程研究所碩士班

摘 要

減少資料存取時的記憶體等待(memory stall)一直是改進程式執行效能的重要課題。傳統程式常在執行時花費了許多時間在等待較低層記憶體的存取，爪哇(Java)程式亦然。為了要減少在記憶體上的等待時間，預先抓取(prefetching)所需要的資料是種可行的方案。我們觀察到爪哇程式中，位元碼(bytecode)之抓取與陣列資料之存取都有明顯的特性，可以利用這些特性去預先抓取它們。這篇論文設計了適用於嵌入式爪哇硬體加速器中，位元碼與陣列資料預先抓取的機制，以減少等待記憶體讀取資料的時間。我們分析了它們的特性，並設計了一些合適的方法。平均而言，我們的方法可降低約一半的位元碼抓取所造成的停滯；在某些以陣列存取為主的程式上，也能減少約一半因陣列存取所造成之記憶體等待。

Prefetching for Bytecode and Array Data in Embedded Java Hardware Accelerators

Student: Yi-Ruei Wu

Advisor: Dr. Jean Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

For improving speed of program execution, it is important to reduce stalls caused by memory accesses. Traditional programs usually spend much time on memory stalls during accessing lower-level memory, and so do Java programs. In order to reduce memory stall time, prefetching is a feasible solution. We observed that there exist obvious properties of bytecode fetchings and array accesses, so we can try to prefetch them by taking advantage of their properties. This thesis proposes novel prefetching mechanisms for embedded Java hardware accelerators to prefetch bytecode and array data, so that the time spent on memory stalls can be reduced. We analyzed their properties and designed suitable approaches. Our approaches can reduce half of bytecode stall time on an average; for some array-based programs, about half of array stall time can also be eliminated.

誌 謝

終於結束一邊吃著學校餐廳賣的便當，一邊猜測裡頭到底放什麼詭異食材的日子了。

記得第一次 meeting 時坐在底下聽著學長報告，即使前所未有的認真與專心，但仍是鴨子聽雷。看著老師與博士班學長們的無情批判，就覺得研究生涯真是可怕，總有一天被釘在牆上的會是我。

碩一修課繁重，曾有段時間，回到宿舍就開始啃書本、論文，連電腦都不敢開機；曾有段時間，為了期末作業，在實驗室忙到半夜，隔天拖著疲憊和黑眼圈展示結果。這一年，我不斷地看著學長報告、不斷地自修、不斷地累積實力、不斷地成長。升了碩二，學長們「突然」畢業了，開始輪到我這毛頭小鬼上台報告，學習接受考驗。

做研究的過程，真的很累：過程方法不嚴謹、結果數據不好看，就得重新檢視設計，重頭來過；每次做投影片時，還要自己在腦海不斷預演，猜想會被質疑的問題。日日排山倒海的壓力，如連續劇般一波接一波，常常躺在床上難以成眠，絕不只是因為宿舍很吵。

現在，總算一切都熬過去了。

能完成這篇論文，最要感謝的是單智君老師與陳裕生學長，能夠不厭其煩地給我指導與建議。並感謝鍾崇斌老師的教導、實驗室的各位學長姐與同學對我的幫助；也感謝各位口試委員的寶貴意見，讓論文更加嚴謹完整。

最後，我必須感謝父母從小到大的栽培；未來我仍背負著期許，不斷往前。

我在交大，編織了兩年的回憶，帶著滿滿地感謝；祝福陪伴我的每一個人，都有美好的人生旅程。

Contents

| | |
|--|------|
| 摘要..... | i |
| Abstract..... | ii |
| 誌謝..... | iii |
| Contents | iv |
| List of Figures..... | vi |
| List of Tables..... | viii |
| Chapter 1 Introduction..... | 1 |
| 1.1 Introduction to Java..... | 1 |
| 1.2 Motivation..... | 3 |
| 1.3 Objective | 3 |
| 1.4 Thesis Organization | 4 |
| Chapter 2 Background..... | 5 |
| 2.1 JVM's Internal and Acceleration Technologies | 5 |
| 2.1.1 Java Interpreter..... | 5 |
| 2.1.2 Memory Access Types in Java Programs..... | 6 |
| 2.1.3 Technologies to Speedup Java | 8 |
| 2.1.4 Java Hardware Acceleration | 8 |
| 2.2 Concept of Prefetching | 10 |
| 2.2.1 Introduction to Prefetching | 10 |
| 2.2.2 Side-effects of Prefetching..... | 12 |
| 2.2.3 Prefetch Buffer..... | 13 |
| 2.2.4 Policy Design of Prefetching | 14 |
| 2.3 Hardware-Based Instruction Prefetching..... | 14 |
| 2.3.1 Related Work: Next-Line Prediction Table (NLPT) | 15 |
| 2.3.2 Timing Difficulty to Traditional Embedded Programs | 16 |
| 2.4 Hardware-Based Data Prefetching..... | 17 |
| Related Work: Reference Prediction Table (RPT) | 17 |
| Chapter 3 Designs..... | 21 |
| 3.1 Bytecode Prefetching | 21 |
| 3.1.1 Observations and Main Design Idea..... | 21 |
| 3.1.2 Overview of Bytecode Prefetching..... | 22 |
| 3.1.3 Non-sequential Block Prediction Table (NBPT) | 23 |
| 3.1.4 The State Design of NBPT..... | 24 |
| 3.1.5 Case-by-case Discussions | 27 |

| | |
|--|----|
| 3.2 Array Prefetching | 35 |
| 3.2.1 Observations | 35 |
| 3.2.2 Stride Table | 36 |
| 3.2.3 Stride-Adaptive Prefetching..... | 38 |
| 3.2.4 Array-Base-Tagging..... | 40 |
| 3.2.5 Trigger Block | 40 |
| 3.2.6 Circular Prefetching..... | 42 |
| Chapter 4 Experiments and Results..... | 43 |
| 4.1 Evaluation Environment | 43 |
| 4.2 Benchmarks..... | 44 |
| 4.3 Analysis on the Benchmarks..... | 46 |
| 4.3.1 Memory Stalls | 46 |
| 4.3.2 Experiments on Bytecode | 47 |
| 4.3.3 Stride Distributions of Arrays | 49 |
| 4.4 Results of Prefetching..... | 50 |
| 4.4.1 Prefetching for Bytecode | 50 |
| 4.4.2 Reference Prediction Table for Data Prefetching..... | 53 |
| 4.4.3 Stride Table for Array Prefetching | 55 |
| 4.5.1 Bytecode Traffic..... | 62 |
| 4.5.2 Array Traffic..... | 63 |
| Chapter 5 Conclusion and Future Works..... | 65 |
| 5.1 Conclusion | 65 |
| 5.2 Future Works..... | 66 |
| 5.2.1 Prefetching More Bytecode Blocks at a Time | 66 |
| 5.2.2 Adaptive Mechanisms..... | 68 |
| 5.2.3 Prefetching for Other Data Types | 69 |
| 5.2.4 Next-Block Prediction for Low Power Caches..... | 70 |
| References..... | 71 |

List of Figures

| | |
|--|----|
| Figure 1.1 Java 2 platform editions and their target markets [22] | 2 |
| Figure 2.1 Relationship between Java and processor caches..... | 6 |
| Figure 2.2 picoJava-I stack cache [25] | 9 |
| Figure 2.3 picoJava-I pipeline [25]..... | 10 |
| Figure 2.4 Illustration of data prefetch [30]..... | 12 |
| Figure 2.5 Prefetch buffer | 13 |
| Figure 2.6 Target table | 16 |
| Figure 2.7 Examples for next-line prediction table | 16 |
| Figure 2.8 Reference prediction table [30] | 18 |
| Figure 2.9 State design of RTP | 19 |
| Figure 3.1 Flow path of bytecode prefetching | 22 |
| Figure 3.2 Structure of non-sequential block prediction table..... | 23 |
| Figure 3.3 Method invocation and return | 24 |
| Figure 3.4 The state design of NBPT..... | 26 |
| Figure 3.5 NBPT: Case of forward branch | 28 |
| Figure 3.6 NBPT: Case of loop..... | 29 |
| Figure 3.7 NBPT: Case of method invocation..... | 30 |
| Figure 3.8 NBPT: A trace of method invocation and return | 32 |
| Figure 3.9 Additional prefetch during method invocation and return | 33 |
| Figure 3.10 NBPT: Case of multiple transition targets..... | 34 |
| Figure 3.11 NBPT: An example of selection between 2 candidates | 34 |
| Figure 3.12 The array structure in KVM | 36 |
| Figure 3.13 Stride table for Java | 37 |
| Figure 3.14 The 2-state design of stride table..... | 38 |
| Figure 3.15 Trigger block | 41 |
| Figure 3.16 Circular prefetching..... | 42 |
| Figure 4.1 Datapath of JOP [24] | 44 |
| Figure 4.2 Memory stall time over total execution time..... | 46 |
| Figure 4.3 Stall distributions..... | 47 |
| Figure 4.4 Average stay time per bytecode block | 48 |
| Figure 4.5 Sequential strength of bytecode..... | 49 |
| Figure 4.6 Stride distributions of arrays..... | 50 |
| Figure 4.7 RSR(bytecode)s to the sizes of NLPT..... | 51 |
| Figure 4.8 RSR(bytecode)s to the sizes of NBPT..... | 52 |

| | |
|---|----|
| Figure 4.9 A comparison of RSR(bytecode) | 52 |
| Figure 4.10 RSR(InstanceField)s of 128-entry RPT..... | 53 |
| Figure 4.11 RSR(array)s to the sizes of RPT..... | 54 |
| Figure 4.12 RSR(array)s to the sizes of RPT..... | 55 |
| Figure 4.13 Average RSR(arrays)s of original RPTs and array-only RPTs | 55 |
| Figure 4.14 Results of configurations of <i>H</i> and <i>prefetch depth</i> in ST | 56 |
| Figure 4.15 RSR(array)s to the sizes of ST | 58 |
| Figure 4.16 RSR(array)s to the sizes of PC-tagged ST | 58 |
| Figure 4.17 Comparison of tagging approaches for ST | 59 |
| Figure 4.18 Comparisons of ST and array-only RPT | 60 |
| Figure 4.19 Effects of each design idea of ST | 61 |
| Figure 4.20 Effects of trigger-block..... | 62 |
| Figure 4.21 Bytecode traffic | 63 |
| Figure 4.22 Array traffic | 64 |
| Figure 5.1 Timing issue of bytecode prefetching..... | 67 |



List of Tables

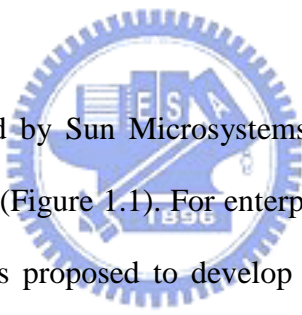
Table 4.1 RSR(array) differences between using the recommended H and *prefetch depth*,
and their optimal configurations..... 57



Chapter 1 Introduction

This chapter gives an overview, and describes the motivation and the objective of this thesis. Section 1.1 introduces Java technology which is applied in popularity and its performance issue. Section 1.2 and 1.3 describe the motivation and the objective of this thesis, respectively. Section 1.4 introduces the organization of this thesis.

1.1 Introduction to Java



Java [27] was introduced by Sun Microsystems. The 2nd version (Java 2) has been widely applied in many fields (Figure 1.1). For enterprise, Java Platform Enterprise Edition (Java EE) industry standard is proposed to develop portable, robust, scalable and secure server-side applications. For desktop, Java Platform Standard Edition (Java SE) provides plenty of APIs for developing applications. For embedded devices such as PDAs, mobile phones, TV set-top boxes, Java Platform Micro Edition (Java ME) provides a well-defined virtual platform that fit for heterogeneous embedded environments. In this region, the K virtual machine (KVM) is designed for products with approximately 128K of available memory. In addition, there is also Java Card technology for IC card applications.

In order to accomplish “write once, run anywhere”, Java programs are not immediately compiled to machine code (say, native code), but an intermediate code called bytecode instead. Java bytecode executes on target platform through a phase of translation. In other words, a Java program must run below a virtual platform, called Java Virtual Machine (JVM). A JVM interprets Java bytecode and does operations on behalf of Java programs.

The simplest implementation of JVM is software interpretation. However, interpretation is much slower than the direct execution of native code. It can only be applied in small embedded environments that don't care performance. Even if we try to use a more sophisticated interpreter, the effect is very limited for most programs. Thus, how to solve the performance issue is an essential topic to Java researchers. Now there are many studies on this topic such as dynamic compilation technologies.

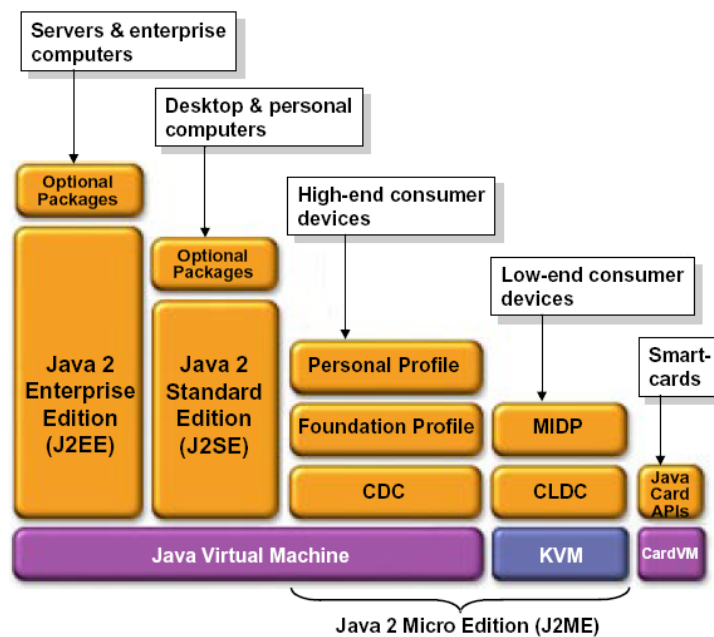


Figure 1.1 Java 2 platform editions and their target markets [22]

Besides software solutions to speedup Java execution, hardware accelerators have been proposed to be an alternative choice; for example, picoJava [19, 25], JOP [29], ARM's Jazelle DBX [3, 4] technology and so on. By these hardware solutions, some simple Java instructions can be executed directly. Now ARM's Jazelle solution has been applied in many embedded environments such as smart phones.

For improving the performance of Java execution, in addition to improving JVM components such as execution engine or memory manager, another way is to reduce

memory stall time. Adding caches, data re-layout [15, 17, 21, 28] or prefetching [1, 30] are all possible approaches. [11] also presents some technologies to enhance memory systems. This thesis focuses on prefetching to eliminate memory stalls.

1.2 Motivation

Traditional program wastes much time on waiting for memory accesses, and the same as Java. Adl-Tabatabai et al [1] indicated the data memory stalls take up to 45% of execution cycles when running the SPEC JBB2000 and SPEC JVM98 benchmarks on Itanium 2. In the experiments of F. Li et al [17], array-based embedded programs, on an average, spend about 45% of execution cycles in memory access. In our experiments on Sun's CLDC HI and EEMBC's GrinderBench benchmarks running on the Intel x86 ISA plus JOP [29], a Java processor, it takes more than 20% of execution time on data cache stalls in average, maximum up to 34%, where the average miss penalty is around 50 cycles. Such situation is being deteriorating as the data requirements or the code size of multimedia applications is continually increasing. So reducing the time on memory stalls shall be very effective in practice.

Bytecode and array data usually take more than 50% of stall time (see Section 4.3), but they have obvious properties for prefetching. Bytecode has sequential-access property and frequent branch targets. For array, distances between two consecutive accesses of an array are usually stable values, which are called *strides*.

1.3 Objective

In order to reduce memory stall time during Java execution, we propose some prefetching mechanisms, which are suitable in embedded Java hardware accelerators, to

prefetch bytecode and array data. For this purpose, there will be two key points: prediction of future accesses and timing determination. I.e., the prefetching mechanisms have to predict where future accesses will locate on and then issue prefetch signals at appropriate time points before the real accesses.

1.4 Thesis Organization

Chapter 2 describes the background and the related work of this thesis. Chapter 3 presents our prefetching mechanisms for Java bytecode and array elements. Chapter 4 shows some experiments and the results of previous designs and our designs. Chapter 5 discusses some variations, future works, and makes conclusions finally.



Chapter 2 Background

This chapter introduces the necessary background for this thesis. In Section 2.1, we introduce JVM's internal and the acceleration technologies. Section 2.2 introduces the concept of prefetching, its potential side-effects and prefetch buffer to solve cache pollution. Then we will introduce some related works. Section 2.3 introduces the next-line prediction table (NLPT) for instruction prefetching, and then discusses the timing difficulty of instruction prefetching to traditional embedded programs. Section 2.4 introduces the reference prediction table (RPT) which is used for data prefetching.



2.1 JVM's Internal and Acceleration Technologies

A JVM consists of many components, such as class loader, execution engine, memory manager..., and so on. In following we briefly introduce how a JVM works, especially focus on implementations of the execution engine, and discuss the data types from the JVM's view.

2.1.1 Java Interpreter

An interpreter is the easiest implementation of the execution engine. Java programs usually do not be compiled to machine code immediately but an intermediate form called bytecode instead. Java achieves its portability based on virtual machine technology. All Java program must be executed on a Java Virtual Machine [18]. A Java compiler reads Java

source code and generates classfiles. Classfiles contains information of classes, including method tables, constant pools, and bytecode of each method... etc. The JVM contains a class loader to load classfiles, resolve names and link them together.

After the initializations of necessary classes, an interpreter in most JVMs is launched to execute the main method of entry class. Classes are dynamically loaded during runtime. An interpreter fetches Java instructions, decodes it, and maps them to corresponding machine codes for emulation. Note that the Java bytecodes are considered as data to interpreter, the same as Java data and stored in data cache (see Figure 2.1).

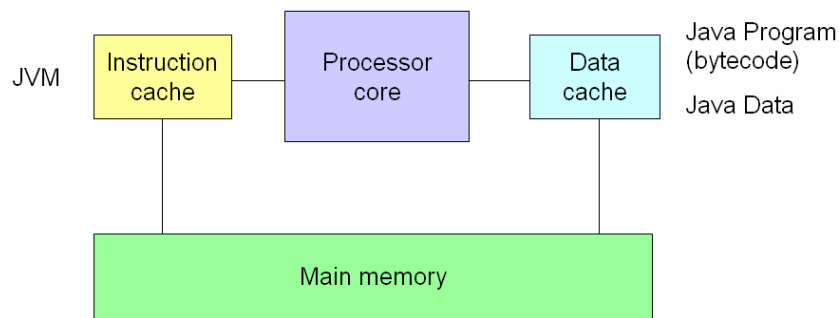


Figure 2.1 Relationship between Java and processor caches

In this case, both Java bytecode and Java data contend with the data of other processes in data cache. Only JVM itself (including interpreter, memory manager ... etc) is in instruction cache.

2.1.2 Memory Access Types in Java Programs

JVM fetches bytecodes and operates data on behalf of them. Each data type has its properties of accessing. The data operated by JVM can be roughly categorized into 3 types:

- Java bytecode fetches

A JVM fetches bytecodes from memory and execute them.

- Object accesses

Including instances and classes, note that an array is a type of instance in Java.

- Stack operations

Computation instructions operate data on the top of stack rather than registers.

Local variables are also stored in stack frames. Arguments passing and method return values are both by way of the stack as well.

Object accesses can further be categorized by Java instruction types:

- Array elements

Accessed by array load/store instructions such as *iaload*, *iastore*, *aaload* ...etc.

- Instance headers

Accessed explicitly (*checkcast*, *instanceof*) or implicitly for type testing, or *monitorenter*, *monitorexit* instructions.

- Instance fields

Accessed by instance field read/write instructions, only *getfield* and *putfield* is associated to this type.

- Static fields

Accessed by static field read/write instructions, only *getstatic* and *putstatic* is associated to this type.

- Class structures other than fields

A class structure is a large structure that stores class information, constant pool, method table, and so on. Class structures are usually implicitly accessed during type testing, name resolution ...etc.

This thesis focuses on bytecode and array. By taking advantages of their properties, we can design some mechanisms to prefetch them.

2.1.3 Technologies to Speedup Java

Interpreter is relatively simple, easy to be implemented and only small memory required. However, a Java program runs on an interpreter is much slower than a traditional program. Thus how to improve Java execution speed is an important issue. There are some well-known approaches to improve Java's performance. For example, ahead-of-time (AOT) compilation [20], just-in-time (JIT) compilation [7], or hardware acceleration [3, 8, 19, 25].

An AOT compiler converts Java bytecode into native code after downloaded. It simply compiles all Java program before execution. On the other hand, a JIT compiler translates Java bytecode into native instructions on the fly. Since JIT compilers work during runtime of Java program, they also introduce additional compilation overhead. Thus, a JIT compiler is usually only allowed to do simple optimization rather than complicated optimization in traditional compiler. Even so, JIT technologies still significantly speedup Java execution.

However, either AOT or JIT compilers are not always suitable in all applications. First, an extra, large amount of memory is required for either compiler itself or compiled code. It may be infeasible in many embedded systems that only have small memory. Second, dynamic compilation may result in a short period of pause during program execution. Pauses are sometimes bad for user experience or real-time systems.

2.1.4 Java Hardware Acceleration

JIT compilation technologies are the most frequently used approach, but sometimes infeasible under some circumstances. Java hardware acceleration is another solution. An accelerator can be a separate Java processor, a hardware translation unit or highly integrated with the processor core. Java instructions can directly run on accelerators, so that the speed

of Java execution can approximate to native programs. Furthermore, for example, if the accelerator supports garbage collection that not typically found on conventional processors, Java programs run on the accelerator can be faster than software-only approaches. Simple instructions are usually executed by hardware, either directly implemented or emulated by microcodes. Complex instructions, such as *new* or *athrow*, must still be emulated by software.

Sun's picoJava-I [19, 25] microprocessor is the first hardware accelerator for Java. It is a small, configurable core designed to support the Java Virtual Machine specification. Most instructions execute in one to three cycles. For complex instructions, it traps to software to keep the complexity and size of the core manageable. The picoJava-I has a dedicated cache to handle stack operations as Figure 2.2. Stack operations, unless during filling or spilling, merely access the stack cache instead of the data cache. The picoJava-I has a 4-stage, RISC-style pipeline (Figure 2.3). It has an instruction buffer and also has the capability of operation folding and monitor support. It fetches 4 bytes of bytecodes into the buffer at a time rather than merely one instruction. Therefore it fetches bytecode from the buffer instead of accessing the cache. After the picoJava-I published, Sun soon announced the picoJava-II [26] for next generation of Java processor. Even if they have never been realized, the picoJava series became foundations of modern Java processors.

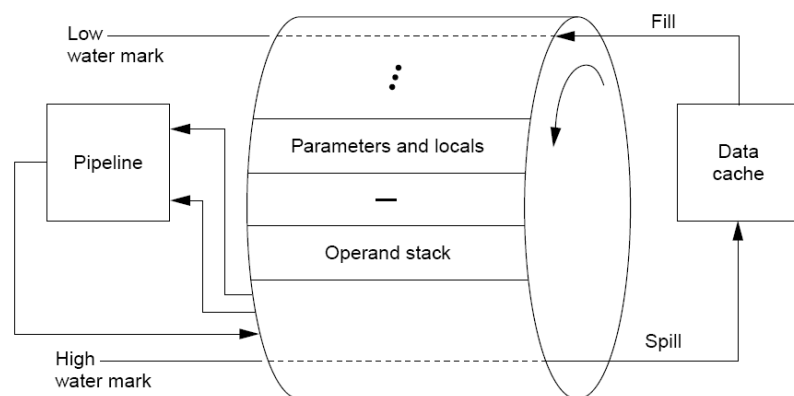


Figure 2.2 picoJava-I stack cache [25]

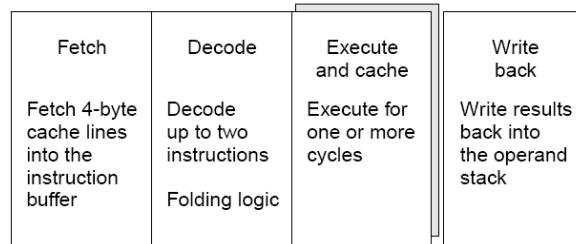


Figure 2.3 picoJava-I pipeline [25]

There are many implementations of Java accelerators [2, 4, 23, 24, 29] today. For example, ARM Jazelle DBX [4] takes Java bytecodes as an instruction set extension. The instruction set of Jazelle DBX technology creates a new state similar to Thumb in which the processor fetches and decodes Java bytecodes and maintains the Java operand stack. Now ARM's Jazelle has been applied in many embedded devices such as smart phones.

Some Java processors expect that the OS can run directly below them. However, up to present, no OS porting is developed to achieve this purpose. So they must still co-work with a conventional processor core now.

2.2 Concept of Prefetching

Prefetching can be aimed at instructions or data. Generally speaking, in stored program computer, instructions can be viewed as a type of data. This section introduces the concept of prefetching and its derived issues.

2.2.1 Introduction to Prefetching

Rather than cache uses history of running program, prefetching predicts future based on data properties. Prefetching [30] anticipates cache misses and issues a fetch to memory

system in advance of actual memory reference. Prefetches proceed concurrently with processor computation. See Figure 2.4(a), the processor has to stall after memory read finished. This is because traditional cache only fetches data “on demand,” namely, issue data request to memory system only on cache misses. In Figure 2.4(b), the prefetching effectively hides all memory latency since memory accesses go in parallel with the computation. When the processor requires data, they have been ready. However, actually, nothing is so perfect. Real cases are like Figure 2.4(c), some prefetches are issued too late so that the processor still must wait for data to be ready. Some prefetches proceed too early and may result in “cache pollution.”

Data prefetching instructions can be inserted manually by programmer but increases the programmer’s work. There are 2 approaches for automatic data prefetching: one is compiler-directed approach, the other is hardware-based approach. Compiler-directed prefetching, either statically [32] or dynamically [1, 13], inject additional computation for miss or address prediction and prefetch instructions into compiled code. Additional computations, however, slowdown the normal execution slightly. Furthermore, extra instructions may a bit degrade instruction cache performance.

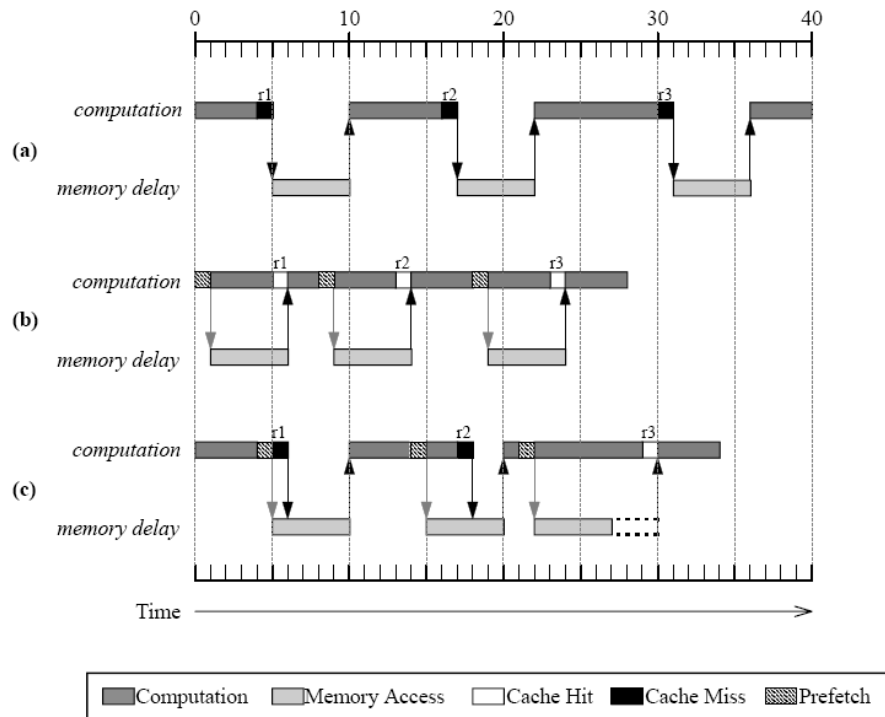


Figure 2.4 Illustration of data prefetching [30]

(a) No prefetching (b) Perfect prefetching (c) Degraded prefetching

Hardware-based prefetching, different to compiler-directed approach, produces no performance overhead. However, since dynamic approaches, including dynamic compiler prefetching, usually lack for high-level language semantics, it's tough to them to foresee longer so that they usually make more inaccurate decisions.

We will call the basic unit of prefetching “block” rather than line in order to avoid confusing with cache lines. Basically, the block size equals to the cache line size.

2.2.2 Side-effects of Prefetching

Prefetching brings not only positive effects but side-effects that play a decisive role. Prefetching is a kind of non-blocking load, so we need hardware supports of course. In addition, as we described above, software-based prefetching will expand code size, may increase execution stream and degrade instruction cache performance. Again, consider we

prefetched data into some cache line and the original data in the line was replaced. If afterwards the processor requires the original data replaced, an underserved miss is produced. It is also possible that our prefetched data has never been used by the processor. This phenomenon is called “cache pollution.” Note that this effect is different from normal cache replacement miss.

2.2.3 Prefetch Buffer

Instead of putting data into cache directly which may result in cache pollution, it is a good idea that temporarily putting data in a relatively small memory. Such a small memory is called prefetch buffer [14]. We can check the prefetch buffer first when cache misses. If the requested data is found in the prefetch buffer, it would be written into the cache directly so that the processor can go in proceed. In case that it also misses in prefetch buffer, the processor is obliged to wait on main memory accessing eventually.

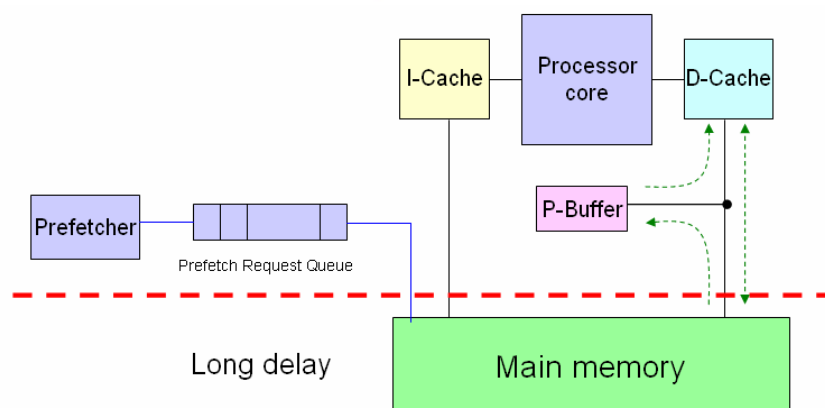


Figure 2.5 Prefetch buffer

A prefetch buffer can be a tiny cache which has high associativity. With such a buffer, we are able to utilize necessary data only and avoid unnecessary prefetches polluting the cache. Note that before a prefetch be really issued to the memory system, it has to check the

cache and the prefetch buffer first.

2.2.4 Policy Design of Prefetching

Prefetching is really a tricky approach. There are too many issues we have to take into account. For example, memory latency, cache size and hierarchy, implementation of prefetch buffer, number of processes which may run concurrently, priorities of current process ...and so on. The effects and designs of prefetching extremely depend on the platform and what to run.

Note that a prefetch may contend with other prefetches either on timing, or in the prefetch request queue or in the prefetch buffer. Roughly speaking, we might be able to use a more aggressive policy for prefetching if the contention is slight. That is, we can prefetch more data even if we don't have much confidence. However, if the contention is so obvious that the effect of prefetching degrades, we tend to use a conservative policy, only prefetch the data we confide in to ease the contention.

2.3 Hardware-Based Instruction Prefetching

Some high-performance processors will fetch following instructions into a buffer beforehand when fetch some instruction. For example, when a processor fetch an instruction I_i , it also fetches I_{i+1} , I_{i+2} , ... I_{i+k} into a instruction queue for future use or issuing in parallel. The simplest design is sequential fetching and no speculation. Sophisticated speculative processors also can make use of a branch prediction table to get better accuracy [31]. Nevertheless, they fetch instruction from cache into instruction queue rather than from memory into cache. Conventionally, they usually stop speculation and maybe stall during high penalty misses.

2.3.1 Related Work: Next-Line Prediction Table (NLPT)

Instruction cache misses, different from data cache misses that can be effectively hidden by a large instruction window and out-of-order execution, the processor usually has to stall and stop speculation. Hsu and Smith [12] studied instruction prefetching approaches for scalar supercomputer pipelines and programs. The simplest method is sequential prefetching, which is called fall-through prefetching in [12]. A table (called target table) can also be used to record the history of block switches. Each entry of the table consists of a pair of (current-line, next-line) as Figure 2.6. When the program counter changes to a new block, the prefetch unit looks-up the table and issues a prefetch for the next block if hit. [12] also proposed a combined algorithm for block prediction. In the combined algorithm, the target table only records non-sequential pairs. It is also updated at every block switches; however, when a sequential transition is detected, it will not be inserted or the corresponding entry will be removed. When the program counter enters a new block, if current block address of is found in the table, the corresponding next-line is used for prefetching. Otherwise, sequential prefetching is adopted. Hsu and Smith indicated that such hybrid approach can get better effect than fall-through or target table only. Such hybrid approach is called “next-line prediction table (NLPT)” in this thesis.

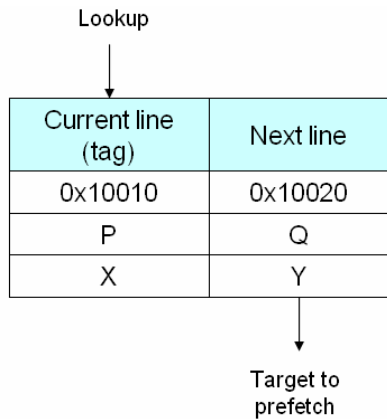


Figure 2.6 Target table

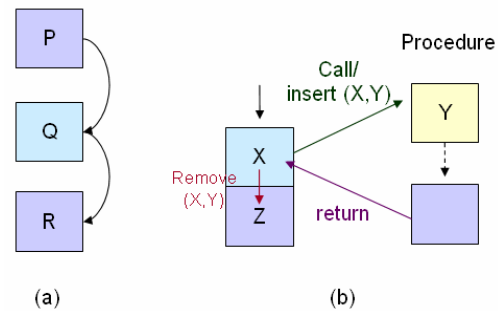


Figure 2.7 Examples for next-line prediction table

Consider (a) in Figure 2.7, the program counter is on block Q at present and transferred from P non-sequentially. The pair (P, Q) will insert into the NLPT. The prefetch unit will also look up Q in the table and make a prediction for the future block R.

NLPT has a weakness. Suppose Figure 2.7 (b), there is a procedure call in X to another block Y. After the call is made and the program counter transferred to Y, a pair (X, Y) would be inserted into the NLPT. However, after the procedure returned, the program counter would move back to X then Z. Note here, the transition of (X, Z) is sequential, so (X, Y) will be removed from the table. If the procedure call is in a loop, the (X, Y) will be absent in the table at the next iteration so that Y will never be prefetched.

2.3.2 Timing Difficulty to Traditional Embedded Programs

If we want to prefetch instruction block into the cache or a buffer, there are 2 important issues for us to take account of. One is the prediction of future fetches. It is easy for coming instructions since the spatial locality of instructions is so obvious. The other is to determine the timing. We have to prefetch an instruction before a period of the instruction is really required. This is very difficult for traditional embedded program. For instance, suppose an embedded RISC processor which has 32-bit instructions and 16-byte cache line, the time of

the processor stays per line is only about 4 to 6 cycles in average. Because hardware-based prefetching lacks for high-level language semantics and is unable to foresee too longer, in the environments which have decades of memory latency, it very difficult to have a good hardware-based approach and obtain good effect of speedup. Even if we use a CISC processor to get higher code density, the compiler still tends to generate simple instructions since the complex instructions are not supported in high-level languages. On the other hand, it usually stays average 40 to 60 cycles per cache line which stores Java bytecodes. So the opportunity of bytecode prefetching will be much more than instructions in traditional embedded programs.

2.4 Hardware-Based Data Prefetching

Sequential prefetching seems effective for instructions because of the high locality of it, but much less for data. Baer and Chen proposed the reference prediction table (RPT) design for data prefetching [5].



Related Work: Reference Prediction Table (RPT)

An RPT is a hardware table. It is similar to an instruction cache tagged by the program counter address, but records the generated addresses of load/store instructions. An entry of RPT has following fields (See Figure 2.8):

- Instruction tag:
The address of a load/store instruction.
- Previous address:
The address which was referenced by the instruction.
- Stride

The difference of the 2 most recently generated address.

- State

A 2-bit encoding of 4 states that indicates how further prefetches should be generated.

An entry in the RPT will be in one of the 4 possible states (Figure 2.9):

- Initial

Start state and no prefetching.

- Transient

The stride may be in transition. A tentative prefetch is issued.

- Steady

The stride is stable. We can issue a prefetch if stride $\neq 0$.

- No prediction

No fixed stride is detected. It won't issue any prefetch in this state.

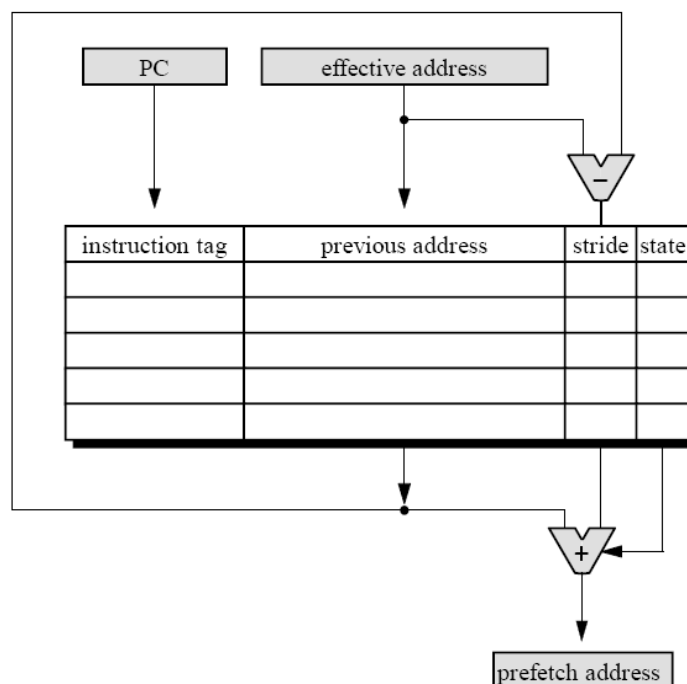


Figure 2.8 Reference prediction table [30]

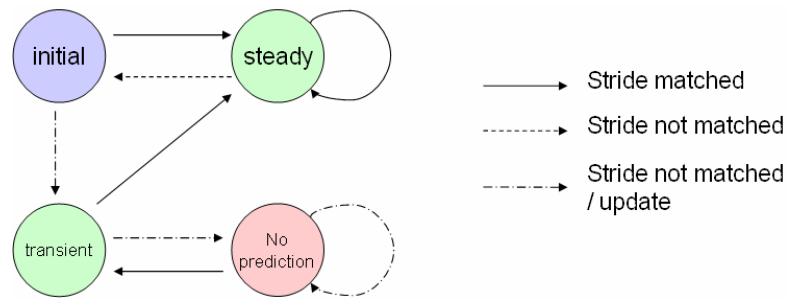


Figure 2.9 State design of RTP

When the program counter encounters a load/store instruction, the instruction is inserted into the RTP and its PC address is used for tag, the location for accessing is stored in the previous address, the state will be set to “initial” and the stride is 0 initially. The instruction may be enclosed in a loop or a subroutine; in this situation, it will be encountered more than once when the loop goes back or the subroutine is entered again. Therefore, we can find the corresponding entry that we filled previously in the RPT. We are able to obtain a stride value by calculating the difference between current address and the previous address, and then compare it to the stride field. If the calculated stride matches the stride field, the state will go toward “steady”. After we mispredicted the stride twice or more, the stride field will be modified to the newest. The state goes toward “no prediction” if we mispredicts repeatedly.

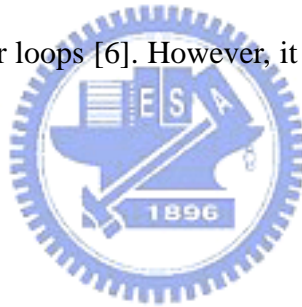
RPT prefetches the data seems be used in next iteration where the address for prefetching P is calculated by

$$P = \text{current address} + \text{stride}$$

(Figure 2.8). Then P is issued for prefetching if the state is “steady” or “transient”.

There exist some potential weaknesses of RPT for traditional programs. First, a drawback is that all load/store instructions will be also inserted into the RTP no matter what type of data they access. For example, the loads/stores for local variables or structure fields

which have no fixed strides will be recorded in the RTP and result in unnecessary waste of entries. Second, the prefetch may cross over the boundary of array since it cannot know where the array begins and ends, so that some unnecessary data would also be prefetched. Third, in case of small stride, even if the prefetch for next required block has been issued by previous access, RTP still tries to issue prefetch for the same block. This results in unnecessary cache lookups which consume more power. Subsequent prefetches may also wait for previous cache checks completed so that they are postponed. Finally, if the loop body is too small, the prefetched data may arrive too late for the next access. In a large loop, the prefetched data maybe wait too long and contend with other data in the cache or the prefetch buffer. Chen and Baer proposed a dual-ported RTP approach, with a look-ahead program counter taking advantage of branch target buffer (BTB) which has dual real ports, to improve the timing issue for loops [6]. However, it seems too complex and too expensive for embedded devices.



Chapter 3 Designs

By observations of Java properties, we designed some mechanisms for bytecode and array data prefetching. Section 3.1 describes the mechanism of bytecode prefetching and discusses the design strategies case-by-case. Section 3.2 presents the design of array prefetching.

3.1 Bytecode Prefetching

A Java hardware accelerator usually fetches bytecodes and executes them directly. Bytecode is very similar to traditional program code, but has some different properties that we should care or can make use of.



3.1.1 Observations and Main Design Idea

In small line size environments, because of the fleeting stay per cache line and the limited prediction ability of hardware-based prefetching, it can not gain too much benefit by hardware-based instruction prefetching for traditional embedded program as we have mentioned in Subsection 2.3.2. In contrast, Java programs have more complex instructions and much high code density, so that Java takes average 40 to 60 cycles per bytecode line (see Section 4.3). Because of the longer line stay per cache line of Java bytecodes, we can have more adequate time than traditional programs to prefetch bytecode blocks.

Similar to traditional programs, Java bytecodes also have strongly sequential property.

About half of cross-block fetches are sequential, so we can apply sequential prediction for those cases. For non-sequential ones, we use a table, which is named non-sequential block prediction table (NBPT), to record them similar to next-line prediction table.

However, a Java program usually has more method invocations than a traditional program. NLPT can not handle such situation well as we have discussed in Subsection 2.3.1. So NBPT must have some special design for method invocations and returns.

3.1.2 Overview of Bytecode Prefetching

Bytecode prefetching is triggered when the program counter transfers to a new block, namely, at the point of block switching. Suppose the program counter was on block P previously and is on Q at present, and will transfer to block R in the future. After the bytecode prefetching is triggered, the prefetch unit looks up Q in the NBPT firstly. Then it gets a prediction for the next block. Finally, we update the NBPT by (P, Q). The flow path of bytecode prefetching is depicted in Figure 3.1.

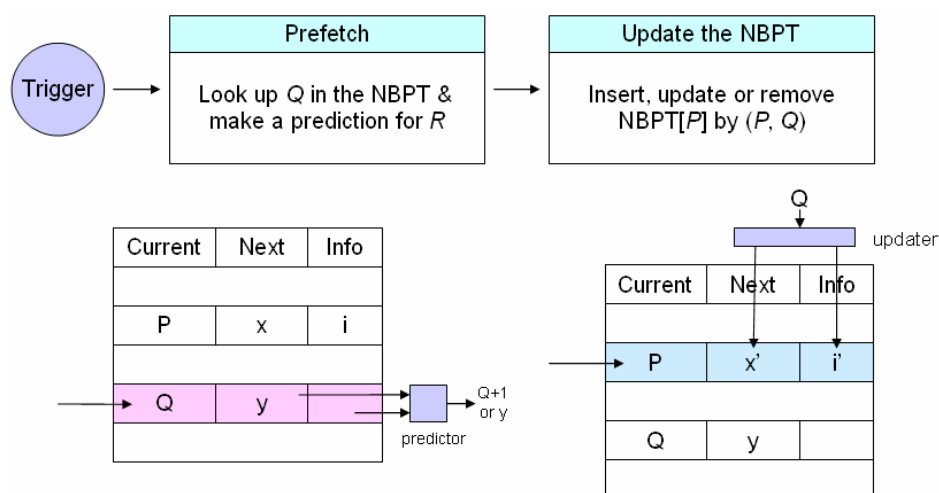


Figure 3.1 Flow path of bytecode prefetching

3.1.3 Non-sequential Block Prediction Table (NBPT)

The design of non-sequential block prediction table (NBPT) is very similar to next-line prediction table. The NBPT records block pairs of non-sequential cross-block fetches of bytecodes, however, has some additional fields designed to obtain better performance for Java which is shown in Figure 3.2.

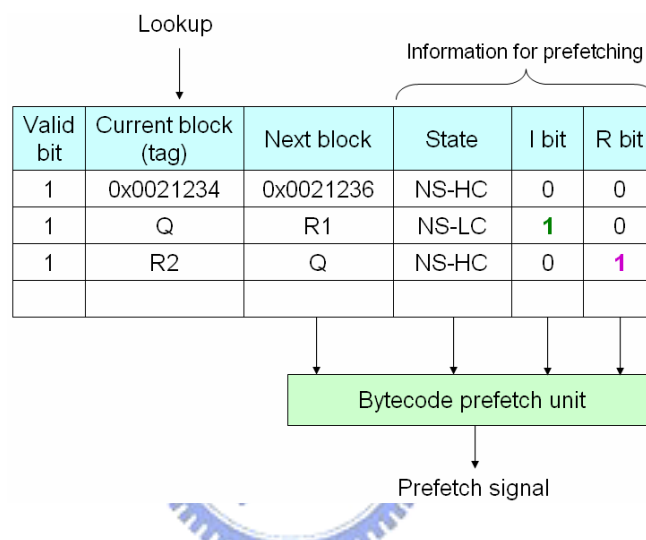


Figure 3.2 Structure of non-sequential block prediction table

Besides the valid bit, an NBPT entry has following fields:

- Current-block

The tag of an entry. If a non-sequential block transition is from P to Q, P will be stored in this field.

- Next-block

The corresponding non-sequential block of the current-block. In other words, if a non-sequential block transition is from P to Q, Q will be stored in this field.

- State

The state of an entry. This field decides what prediction we make.

- I-bit

Set if the block transition is caused by a method invocation.

- R-bit

Set if the block transition is caused by a method return.

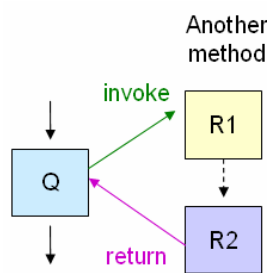


Figure 3.3 Method invocation and return

Figure 3.3 is an example of method invocation and return. There is an invoke instruction in block Q. When the program counter encounters this instruction, a block transition occurs from Q to R1. Thus an entry of (Q, R1) will be inserted into the NBPT. Because this transition is caused by a method invocation, the I-bit will be set to 1 during insertion as in Figure 3.1. When the method ends on R2 and returns back to Q, then (R2, Q) will be inserted and the R-bit is set to 1.

3.1.4 The State Design of NBPT

The state design of NBPT follows some principles. First, we amend our prediction after mispredicted twice rather than the once-policy taken by NLPT. This will be contributive for us to choose a more frequent path. Second, we should prevent the entry due to method invocation from being removed immediately after return. Especially for Java programs which usually have more method invocations, this will help improve prefetching. Finally, we will see in Subsection 3.1.5, there is a period of latency between the decoding of

an invoke/return instruction and the beginning of the target method. By taking advantage of the latency, we may issue an additional prefetch during method invocation or return.

NBPT predicts next block by the state of current block, and updates the state of previous block by current block. The 4-state design of NBPT is shown in Figure 3.4:

- Sequential with High Confidence (S-HC)

If we can not find a corresponding entry for a given block, it is considered in this state, and vice versa. This state represents a higher probability of that the next block of a given block is sequential.

- Sequential with Low Confidence (S-LC)

Given a block in the S-LC state, we tend to believe the next block of it is sequential even if it was non-sequential previously. The previous non-sequential consecutive block is recorded in the next-block field.

- Non-Sequential with Low Confidence (NS-LC)

We tend to believe the next block of a given block is non-sequential if it is in this state. However, we have less confidence in the next-block field of its corresponding entry and are ready to refresh it at any moment.

- Non-Sequential with High Confidence (NS-HC)

We confide in the next-block of the corresponding entry of a block highly when it is in this state.

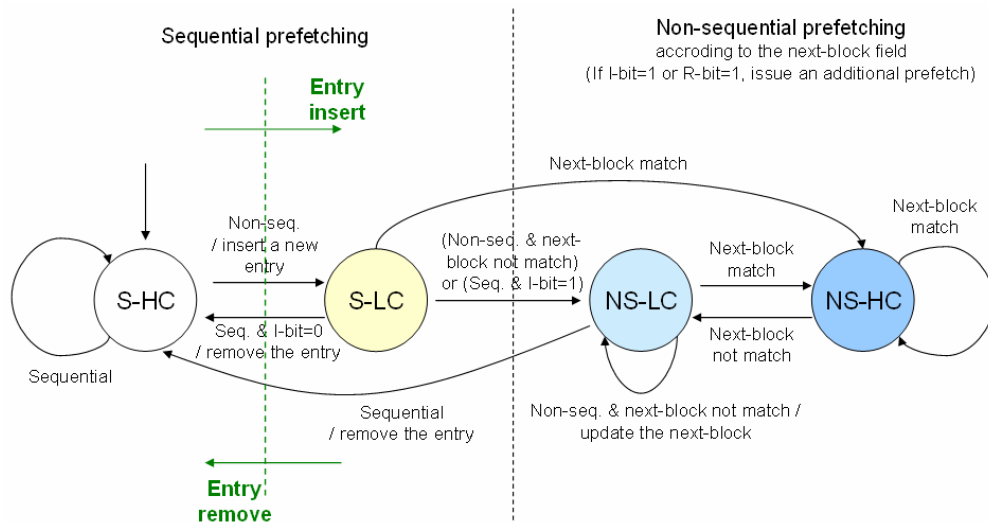


Figure 3.4 The state design of NBPT

Suppose a block transition is from block P to block Q and on Q at present, and will then transfer to block R in the future. Firstly, the prefetch unit looks up Q in the NBPT and makes a prediction r for R by the state of the corresponding entry of Q:

- Sequential with High Confidence (S-HC)

Q is not found in the NBPT, do sequential prediction. I.e., our prediction r is $Q + 1$.

- Sequential with Low Confidence (S-LC)

Q is found in the NBPT but in state S-LC, do sequential prediction. Prediction $r = Q + 1$.

- Non-Sequential with Low Confidence (NS-LC)

Q is found in the NBPT and in state NS-LC. Predict R by the corresponding next-block. That is, prediction $r = \text{NBPT}[Q].\text{next-block}$. If the I-bit or the R-bit of $\text{NBPT}[Q]$ is set, we may make an additional prediction s for the block consecutive to r. For simply, we can let $s = r + 1$. In this case, the prefetch unit can issue 2 prefetches in this state. Moreover, if the transition (P, Q) is caused by a method return and the I-bit of entry Q is 1, we don't prefetch the target method again

since it is unnecessary.

- Non-Sequential with High Confidence (NS-HC)

Q is found in the NBPT and in state NS-HC. The action is the same as that in state NS-LC.

Afterwards, the NBPT should be updated by (P, Q). P may be in following states:

- Sequential with High Confidence (S-HC)

P has no corresponding entry in the NBPT is considered in this state. If (P, Q) is sequential, it remains in the S-HC state and won't be put into the NBPT.

- Sequential with Low Confidence (S-LC)

If (P, Q) is non-sequential and any entry of P is not found in the NBPT, it will be inserted and the state will be set to S-LC initially.

- Non-Sequential with Low Confidence (NS-LC)

After the next-block not matched, the entry which corresponds to P moves to this state for updating the next-block.

- Non-Sequential with High Confidence (NS-HC)

If the next-block continuously matches, it goes toward this state.

3.1.5 Case-by-case Discussions

We will see how NBPT works case-by-case in this section. We go through the NBPT design by following cases and consider each pattern will be encountered more than once:

- I. Forward branch
- II. Loop (backward branch)
- III. Method invocation and return
- IV. Multiple transition targets

Case of Forward Branch

Here we consider *if* instructions in Java. They are: *ifeq*, *ifne*, *iflt*, *ifge*, *ifgt*, *ifle*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpge*, *if_icmpgt*, *if_icmple*, *if_acmpeq*, *if_acmpne*, *goto*, *ifnull* and *ifnonnull*. Their destination can be forward or backward. In case of backward, it usually forms a loop and we will discuss in next subsection. Now we consider the forward case.

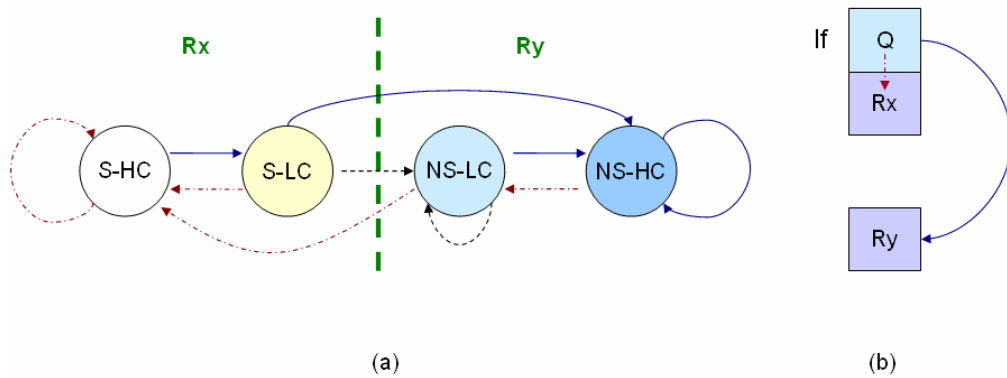


Figure 3.5 NBPT: Case of forward branch

See Figure 3.5 (b), there is an *if* instruction in block Q. If the branch is taken, the program counter will jump to block Ry and restart execution from Ry. Otherwise, it continues to execute the instructions after the *if* and then transfer to Rx. If the Ry case appears more frequently than Rx, the state goes toward the right hand side along the solid line in Figure 3.5 (a). Note that we predict Ry for the right part and Rx for the left part. If the Rx case is more common, the entry will be invalidated or just stays in the left part of Figure 3.5 (a).

Case of Loop (Backward Branch)

A loop is formed by a backward *if*. Consider a loop pattern as Figure 3.6 (b). The program counter transfers from Q to Ry every iteration so that the state goes toward and stays in state NS-HC. However, it will leave the loop and transfer to Rx eventually. Then the state will become NS-LC. If the program counter enters the loop again, NBPT still predict Ry at the first iteration and then the state is set back to NS-HC at the second iteration.

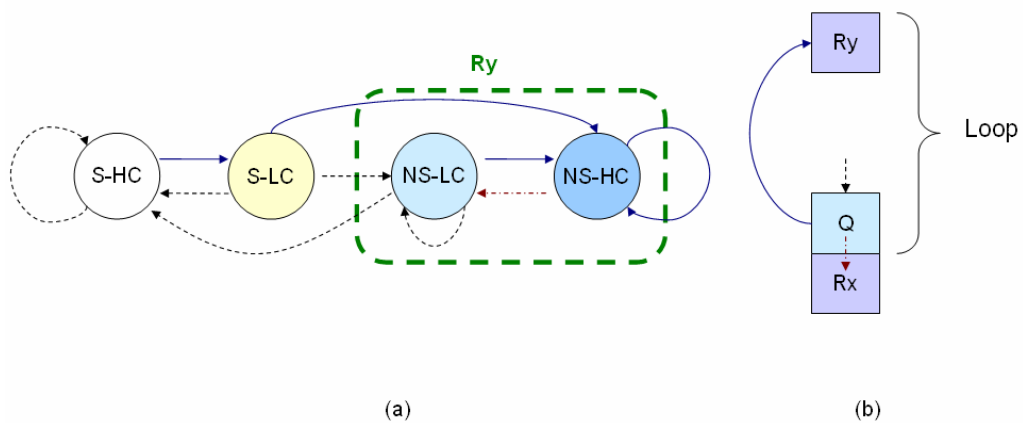


Figure 3.6 NBPT: Case of loop

Case of Method Invocation and Return

Method invocations do not be handled well by NLPT as we have mentioned in Subsection 2.3.1. Thus, a key point of NBPT is to prevent an entry of method invocation from being invalidated immediately. Figure 3.7 (b) depicts the program flow of a method invocation and its return. There is an invoke instruction in block Q. When it encounters the instruction, the JVM determines the method location and restart execution from the first block R1 of the target method. After finished the duty of the method, it will return eventually. Thus the program counter transfers back to the subsequent instruction after the

invoke site when the method returns. Then it will move to block R3 after Q completed. Here we consider the case of R3 is sequential to Q. For case that (Q, R3) is non-sequential, this is a situation of multiple targets. The state machine will choose the frequent one between R1 and R3 in this situation as we will describe later.

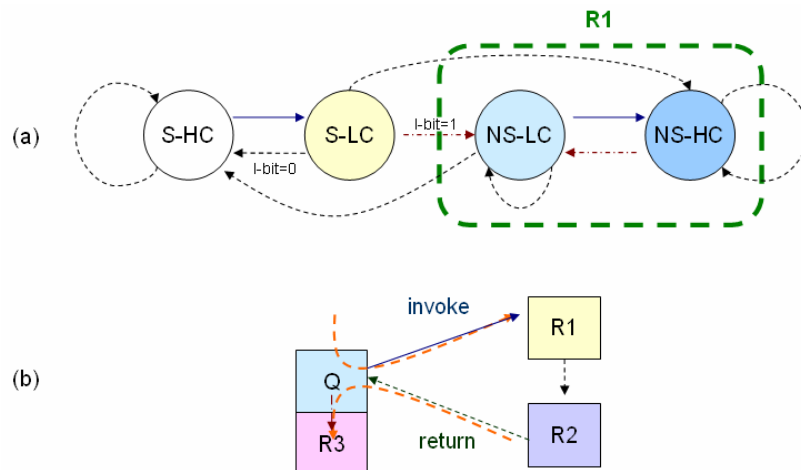


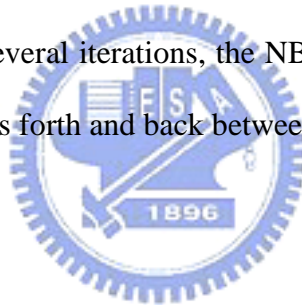
Figure 3.7 NBPT: Case of method invocation

Now refer to Figure 3.8:

- (a) Initially, (Q, R1) and (R2, Q) are both not in the NBPT. After entered Q, because there doesn't exist any corresponding entry of Q, the prefetch unit predicts R3 sequentially.
- (b) After encountered the invoke instruction in block Q, the program counter transfers to R1. At the same time, an entry of (Q, R1) is inserted into the NBPT, where the state is S-LC initially and the I-bit is set to 1.
- (c) After the work of the invoked method finished, the program counter returns back to Q from R2. (R2, Q) is put into the NBPT where the state is S-LC and the R-bit is set to 1.
- (d) After the program counter left from Q and moved into R3, the corresponding entry of Q should be updated. Note the condition of the arc from S-LC to S-HC is

“sequential and I-bit=0”. Because the I-bit of (Q, R1) is 1, it will go along another arc toward the NS-LC state.

- (e) Consider the program counter entered Q again. Nothing has been changed yet if they were not replaced out. Now the prefetch unit predicts R1 for the next block because (Q, R1) is in the NBPT and in state NS-LC.
- (f) The program counter entered R1 because of the method invocation. The state of (Q, R1) became NS-HC from HS-LC since its next-block matched.
- (g) Afterwards the program counter returned back to Q from R2, (R2, Q) became NS-HC from S-LC since its next-block matched.
- (h) The program counter left from Q and entered R3, (Q, R1) became NS-LC because its next-block did not match. Note it will become NS-HC when the invocation occurs again. After several iterations, the NBPT will be like (g) or (h) finally. The state of (Q, R1) moves forth and back between NS-HC and NS-LC.



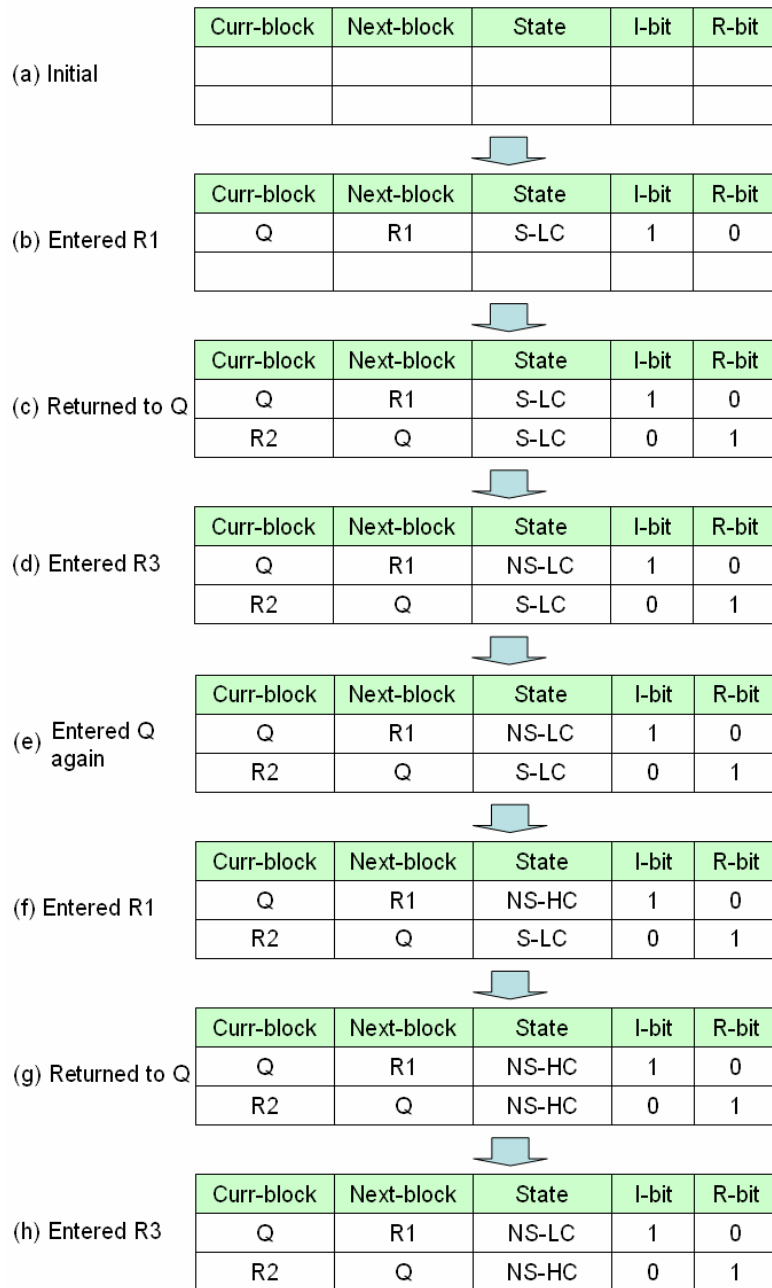


Figure 3.8 NBPT: A trace of method invocation and return

Besides, we may issue an additional prefetch during method invocation and return. This is because during invocation and return, it traps to software JVM to do some duties. Then there would be a period of time for us to prefetch one block extra. See Figure 3.9 (b), block A which has a invoke or return instruction is in method X(). The program counter will transfer to block B which is in another method Y() after met the instruction and then enter C.

Before entering B, it will trap to the software JVM to fix up frames, determine where B is, and do some checks. Compare to traditional programs, these works have been done before the call/return instruction, so a conventional processor is able to jump to the target address directly. Thus, we may prefetch an additional block C during this period of time. For simply, we can just speculate that block C is sequential to block B. See Figure 3.9 (a) as an example, we prefetch block B1 and B2 when entering A2 from A1. At the entry of B3, we prefetch block A2 as well as A3. Note that the software JVM may also produce misses, however, we can still obtain some advantages.

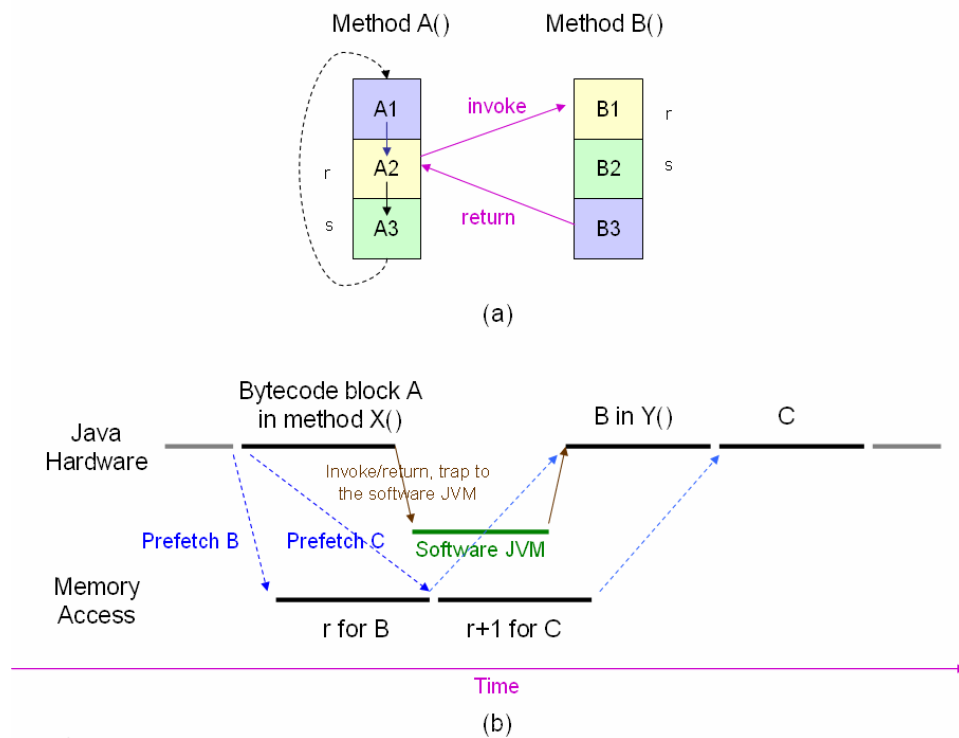


Figure 3.9 Additional prefetch during method invocation and return

Case of Multiple Transition Targets

Finally, we consider the case that the program counter may transfer from one block to multiple target blocks. This is probably caused by a virtual method invocation, an indirect

branch, or there are multiple branches in a block. Take Figure 3.10 (b) for example, if we have 2 possible targets Rx and Ry, we would like to choose the most frequent one since our NBPT only records one target. The selection mechanism is designed in the left part which is circled in Figure 3.10 (a). See Figure 3.11, if (Q, Rx) appears frequently and (Q, Ry) appears occasionally, NBPT will tend to select Rx. However, if (Q, Ry) continually occurs twice, the next-block of the NBPT entry will be replaced by Ry. At this moment, Ry is considered as the most frequent block.

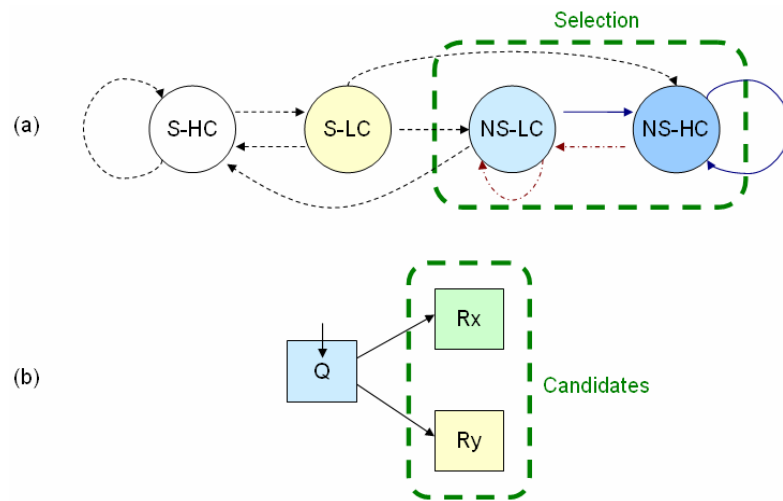


Figure 3.10 NBPT: Case of multiple transition targets

The entry for Q in the NBPT

| Exec Pattern | Curr | Next | State |
|--------------|------|------|-------|
| Q → Rx | Q | Rx | S-LC |
| Q → Rx | Q | Rx | NS-HC |
| Q → Ry | Q | Rx | NS-LC |
| Q → Rx | Q | Rx | NS-HC |
| | | ⋮ | |
| Q → Ry | Q | Rx | NS-LC |
| Q → Ry | Q | Ry | NS-LC |
| Q → Ry | Q | Ry | NS-HC |
| Q → Rx | Q | Ry | NS-LC |

Figure 3.11 NBPT: An example of selection between 2 candidates

3.2 Array Prefetching

Arrays in Java have some properties that not found in traditional programs. By taking advantages of the observations, we can achieve better performance than traditional prefetching approaches.

3.2.1 Observations

Loads and stores of different data types in traditional program code have the same binary form. Hardware can not tell what data type a load/store instruction is associated to. For example, hardware is difficult to determine a loaded data is a local variable, an array element or an indirect pointer. Thus if we want to prefetch data based on their properties we have to ask compiler's assistance or instrument manually. For array prefetching in Java, fortunately, JVM Specification [18] defines array access instructions that operate array only. Thus, we can concentrate on array accesses and get rid of interferences from other data types.

Most C or C++ programmers prefer to visit an array via pointers to achieve better performance. However, there is no pointer in Java but reference for substitution. Programmers are disallowed to operate a Java reference arithmetically unlike pointer operations. All accesses to certain object are always done through a fixed reference necessarily. An array is an instance of object, so if a programmer wants to access an element of an array, he must give the array reference and an index to the JVM. The JVM can then do boundary check and calculate the actual address of the element. In this situation, the JVM can know an access is associated to which array.

The JVM specification claims that an array operation access data out of the array is disallowed. If such condition occurs, the JVM will throw a

java.lang.ArrayIndexOutOfBoundsException to stop further actions. Thus, the JVM has to know the length of the array which is referenced.

Figure 3.12 shows how an array structures in the Sun’s KVM implementation. Given an array reference and an index, the JVM retrieves the length field first to check whether the index lays in the array. Afterward the JVM calculates the array base from the reference. Finally it can access the element by calculating

$$element\ address = array\ base + element\ size \times index .$$

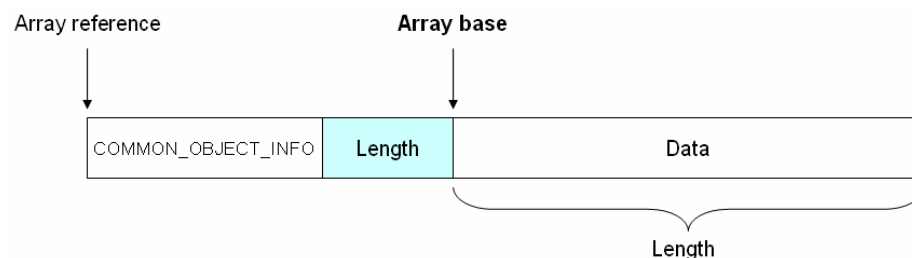
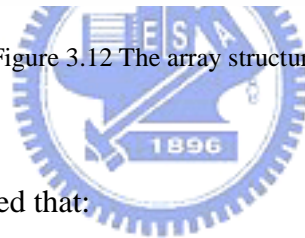


Figure 3.12 The array structure in KVM



In conclusion, we observed that:

- a) Array accesses can be distinguished from other data types by the JVM or the hardware accelerator.
- b) The JVM or the hardware accelerator can determine which array an access is associated to and know the array size during executing an array instruction.

3.2.2 Stride Table

Having the advantageous information described in previous subsection, we can design our array prefetching mechanism for Java. We construct a table, called stride table (ST) to record accesses of each array. See Figure 3.13, a stride table is similar to the RPT design, but an entry of the ST has following fields besides the valid bit:

| Valid bit | Tag (Array base) | Previous offset | Stride | State | Trigger block |
|-----------|------------------|-----------------|--------|--------|---------------|
| 1 | 0x00800100 | 100 | 64 | Init | 0x0080120 |
| 1 | 0x00802300 | 23 | 1 | Steady | 0x008023a |
| 1 | 0x00802305 | 18 | -3 | Steady | 0x00803f2 |
| | | | | | |

Figure 3.13 Stride table for Java

- Array base

The base address of an array. Since all array access should be done via the array base, we can use it alternative to the program counter, for our tag to distinguish from other arrays. Thus, one entry is associated to one array exactly.

- Previous offset

The distance of the address of previous access to the array base in bytes. This field is calculated by $element\ size \times index$.



- Stride

The difference of the addresses of last 2 accesses. If an entry is inserted at the first time, this field is set to the element size initially, because most indexes of array are increased by 1 every iteration in loops.

- State

ST adopts a 2-state design rather than 4 states in RPT as we will describe later.

- Trigger block

This field is optional and will be described in Subsection 3.2.4. In order to avoid producing unnecessary prefetch signals and result in more unnecessary cache checks, we can add this field. If the trigger block is enabled, a prefetch is generated only when an access enters it.

Because an entry will always be mapped to signal array, we may use a simpler 2-state design rather than the conservative 4-state design of RPT. An entry in the ST has 2 possible states shown in Figure 3.14:

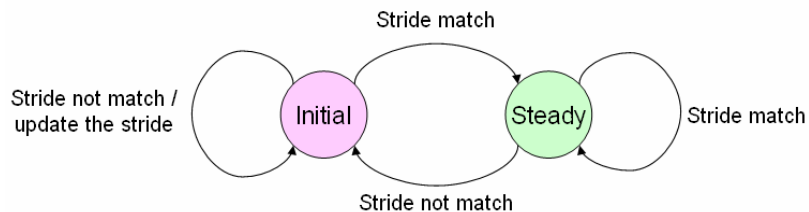
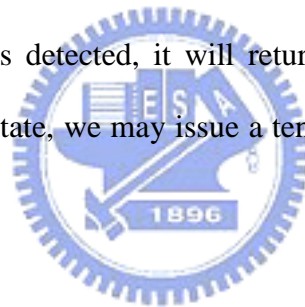


Figure 3.14 The 2-state design of stride table

- Initial

If an entry is inserted at the first time, it would be set to this state initially. When an irregular stride is detected, it will return back to this state for updating the stride field. In this state, we may issue a tentative prefetch and disable the trigger block.



- Steady

If the stride matches, it will go into this state. ST would issue a prefetch for next element here. However, the prefetch signal may be filtered by the trigger block.

Every time of an array access, its corresponding entry will be inserted or updated. Furthermore, since the hardware can know the locations of where an array begins and ends, we don't need to insert an array when its entire body is inside a block as it is unnecessary to be prefetched.

3.2.3 Stride-Adaptive Prefetching

Different to RPT which always prefetches the data of next iteration, ST determines

which block to be prefetched and how many blocks it should prefetch according to the magnitude of stride. The index of current access is denoted by i , its associated element is denoted by $[i]$, and the block number of an element $[j]$ is denoted by $B([j])$:

```

if  $|stride| \leq H$  then // small stride, H is predefined
    if  $stride > 0$  then
        prefetch(  $B([i]+1)$  )
         $Trigger\_Block = B([i]+1)$ 
    else if  $stride < 0$  then
        prefetch(  $B([i]-1)$  )
         $Trigger\_Block = B([i]-1)$ 
    end if
else // large stride
    for  $k = 1$  to  $Prefetch\_Depth$  do
        prefetch(  $B([i+stride*k])$  )
    end
     $Trigger\_Block = B([i+stride*Prefetch\_Depth])$ 
end if

```




If the stride magnitude is smaller than or equal to a predefined value H , we prefetch the next block. However, if the stride magnitude is larger, that means a block is needed only for a shorter period, then another block is required. For this case, we can try to prefetch more blocks at one time. But note that any unnecessary prefetch may make subsequent useful prefetches be postponed.

The trigger block is updated to the last block of prefetching during prefetch signal generation, we will describe it in Subsection 3.2.5.

3.2.4 Array-Base-Tagging

Array base is an alternative option to program counter for tagging entries. Stride table uses array base for its entry tag. An array-base-tagged approach can be better than a PC-tagged approach in some common cases of Java program:

- a) One instruction may manipulate multiple arrays. For example, array utility methods, multi-threaded codes, or more cases that many instances have their own arrays. Their common feature is multiple arrays may share the same instruction.
- b) Multiple instructions manipulate the same array, but there exists a constant stride between them. Loop-unrolled code is an instance:



```
int[] a=new int[100], b=new int[100];  
.....  
// copy b[] to a[]  
for (int i=0; i<a.length; i+=4) {  
    a[i] = b[i] ;  
    a[i+1] = b[i+1] ;  
    a[i+2] = b[i+2] ;  
    a[i+3] = b[i+3] ;  
}
```

In this case, a PC-tagged approach needs more entries to record an array.

3.2.5 Trigger Block

The trigger block field is optional; its purpose is to prevent unnecessary prefetches from being generated. Although most unnecessary prefetches would be gated by cache or buffer checks, however, they consume additional power and might make subsequent prefetches a little delay if the cache or the buffer is in busy. If the trigger block is enabled and an access does not enter it, any prefetch signal will not be produced. Our algorithm always sets the trigger block to be the last prefetched block of an array.

Consider Figure 3.15 (a), when the program access an array element [i] in block A, the prefetch unit tries to prefetch block B. Here we set the trigger block to be the last prefetched block, namely, block B. When it accesses element [j] consecutively, the prefetch unit would also try to prefetch block B. However, it is unnecessary since the block has been prefetched, so will be gated by the trigger block. When an access [k] crosses onto the trigger block, a prefetch for block C will be issued eventually (Figure 3.15 (c)). After the prefetch is issued, the trigger block is also updated to be block C. If an irregular stride is detected, the trigger block should be disabled.

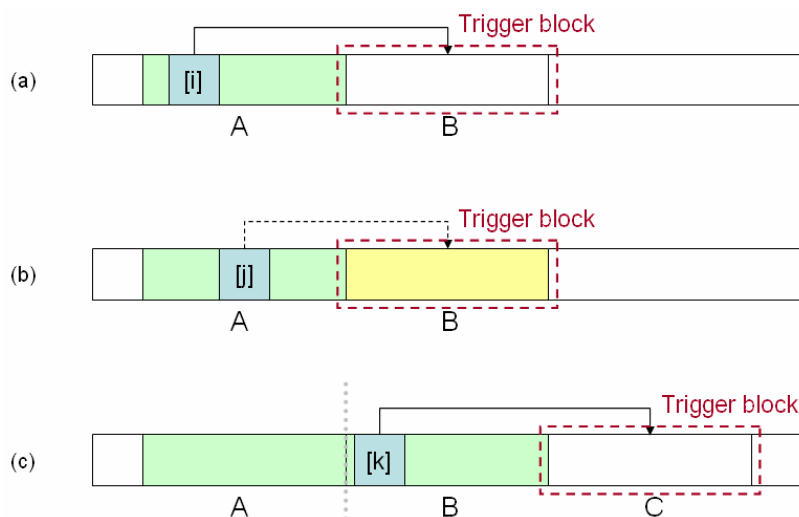


Figure 3.15 Trigger block

3.2.6 Circular Prefetching

Consider a *for* loop in a loop as following:

```
while ( k > 0 ) {  
    .....  
    for (int i=0; i<a.length; i++) {  
        Read a[i]  
        .....  
    }  
    .....  
}
```

When the index i approaches the array tail, the RPT will try to prefetch the data over the array (Figure 3.16 (a)). However, since the hardware can know the array length, we can avoid this situation by a simple comparison. Further, we may prefetch the head of the array for the next entry of *for* as Figure 3.16 (b). This may gain some benefit for case that the *for* loop is entered repeatedly.

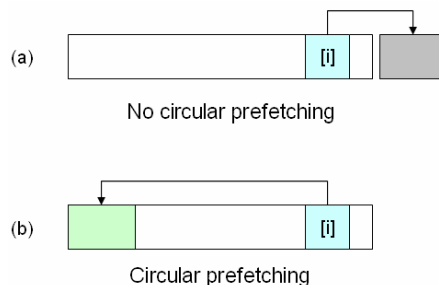


Figure 3.16 Circular prefetching

Chapter 4

Experiments and Results

This chapter presents the experiments and results on 2 benchmark suites: Sun's CLDC HotSpot Implementation Evaluation Kit 1.0.1 and EEMBC's GrinderBench 1.0. Section 4.1 introduces our environment setting for evaluations. Section 4.2 gives introductions of the benchmarks. Section 4.3 presents some analysis on these benchmarks. Section 4.4 shows the results of applying our prefetch mechanisms and compares them to the previous studies. Section 4.5 analyzes memory traffics resulted by the related works and our designs.



4.1 Evaluation Environment

We use the cycle parameters of Java Optimized Processor (JOP) [29] for the simulation of hardware accelerator. JOP is an embedded Java processor implemented on FPGA. It has 4-stage pipeline and handles stack in the internal memory (Figure 4.1). Most bytecode are translated to microcodes. A simple bytecode instruction can be mapped to single microcode; however, a complex instruction must be synthesized by several microcodes. For bytecodes not implemented by JOP, we trap to Sun's KVM 1.1 on Intel x86 processor core for software emulation. We use a 4k bytes data cache with 16 bytes per line; the prefetch buffer is configured to be an 8-line fully associative cache. Average memory latency is set to 50 cycles.

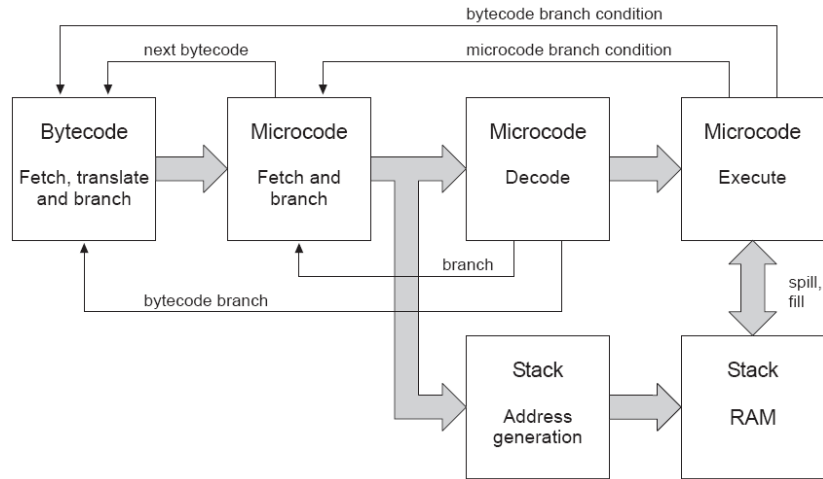


Figure 4.1 Datapath of JOP [24]

4.2 Benchmarks

We use 2 CLDC benchmark suites for our evaluation. One is Sun's CLDC HotSpot Implementation Evaluation Kit (CLDC HI) version 1.0.1, the other is EEMBC's GrinderBench (GB) version 1.0 [9].

Sun's CLDC HI Evaluation Kit 1.0.1 includes 4 benchmarks, following is their brief descriptions:

- Richard

Simulating the task dispatcher in the kernel of an operating system.

- Delta Blue

Solving one-way constraint systems.

- Image Manipulation (Processing)

Reading an image file (Sun raster image format) and performs various transformations on it, such as Sobel, threshold, 3x3 convolver, and so forth. After each transformation, it compares the result with an expected result to confirm that the transformation was done properly.

- Queen

A solver of the n -queens problem, where the objective is to place n queens in a chess board so that no queen can attack another. It is a classical problem used to illustrate several techniques such as general search and backtracking.

EEMBC's GrinderBench 1.0 [9] contains 5 benchmarks:

- Chess

It only performs the logical parts of a chess program, as no graphical output is available. It plays a preset number of games with itself.

- Crypto

It contains multiple encrypt/decrypt engines. The following encryption engines are exercised: DES, DESede, IDEA, Blowfish and Twofish.

- kXML

It processes a command script which specifies XML documents to parse and DOM tree manipulations to do.

- Parallel

This benchmark is used to test the performance of KVM threading capabilities. It accomplishes this by dividing computational tasks among several threads that must then cooperate with each other to complete those tasks. Two parallel algorithms are used: a merge-sort algorithm and a parallel matrix multiplication algorithm.

- PNG

PNG is the standard format for image representation in J2ME implementations. This benchmark does the decoding of a PNG image, including decompression, and stores the result internally as header info, color palette(s), and image data.

4.3 Analysis on the Benchmarks

In order to understand the properties of Java programs, we analyzed the benchmarks. This section presents the experimental results: stall analysis, array stride analysis to each benchmark.

4.3.1 Memory Stalls

Figure 4.2 shows the stall time over the total execution time of each benchmark. In the average of Sun's CLDC HI benchmarks, it takes 15.9% execution time on stalls; In EEMBC's GrinderBench, average 25.7% execution time are spent on stalls. So it is worth reducing memory stall time in order to speedup Java execution.

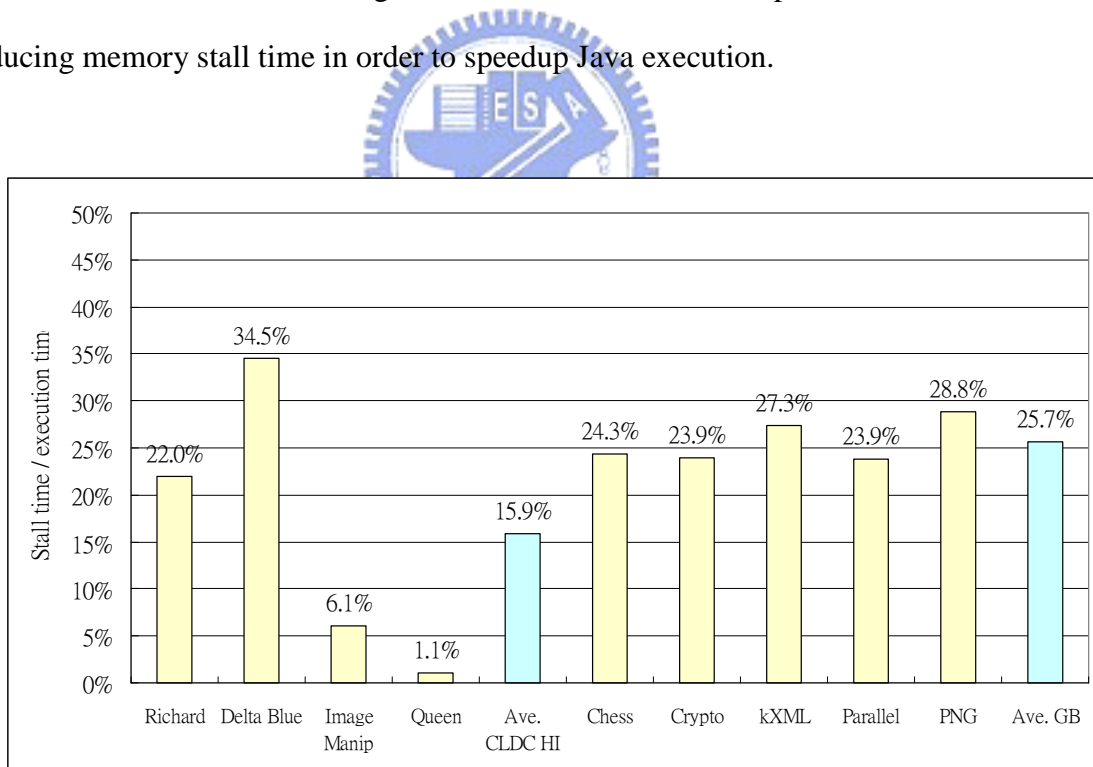


Figure 4.2 Memory stall time over total execution time

Now we consider the composition of stall time. Stall distribution depends on program type. A computation-intensive program will often spend more time on bytecode stalls, such

as Richard, Chess and kXML. An array-based program will usually have a larger proportion of time spent on array stalls; Image Manipulation, Crypto and Parallel belong to this type. On an average, stall time caused by bytecode misses and array misses take more than 50% of total stall time.

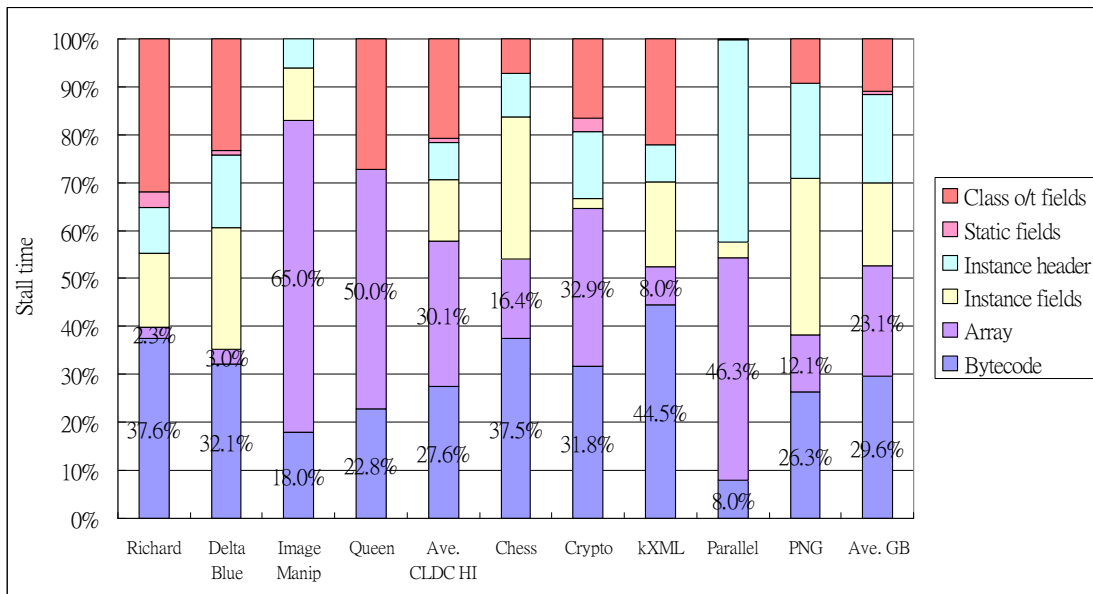


Figure 4.3 Stall distributions

4.3.2 Experiments on Bytecode

As we have mentioned, because the complex instruction design and the high code density of Java programs, there are more opportunities for bytecode prefetching than for instructions of traditional embedded programs. See Figure 4.4, on an average, if we eliminate all stalls, the number of stay cycles per bytecode block distributes from 30 to 40 cycles. If plus stalls, the average numbers of stay cycles of each benchmark are between 40 to 60 cycles.

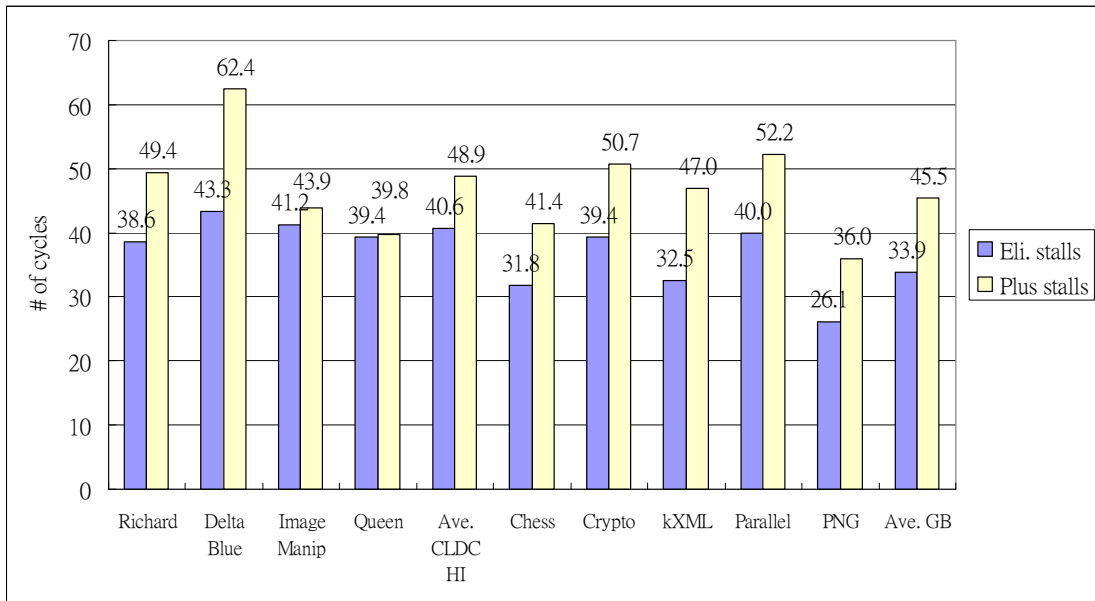


Figure 4.4 Average stay time per bytecode block

If a consecutively-fetched block pair is sequential, then we call it is a *sequential cross*; otherwise, it is a *non-sequential cross*. Figure 4.5 shows the proportions of sequential crosses and non-sequential crosses of each benchmark. On an average, sequential crosses occupy around half of all crosses. Especially in Image Manipulation, the proportion of sequential-crosses is up to 77.3%. Thus, we can apply sequential prefetching for most blocks.

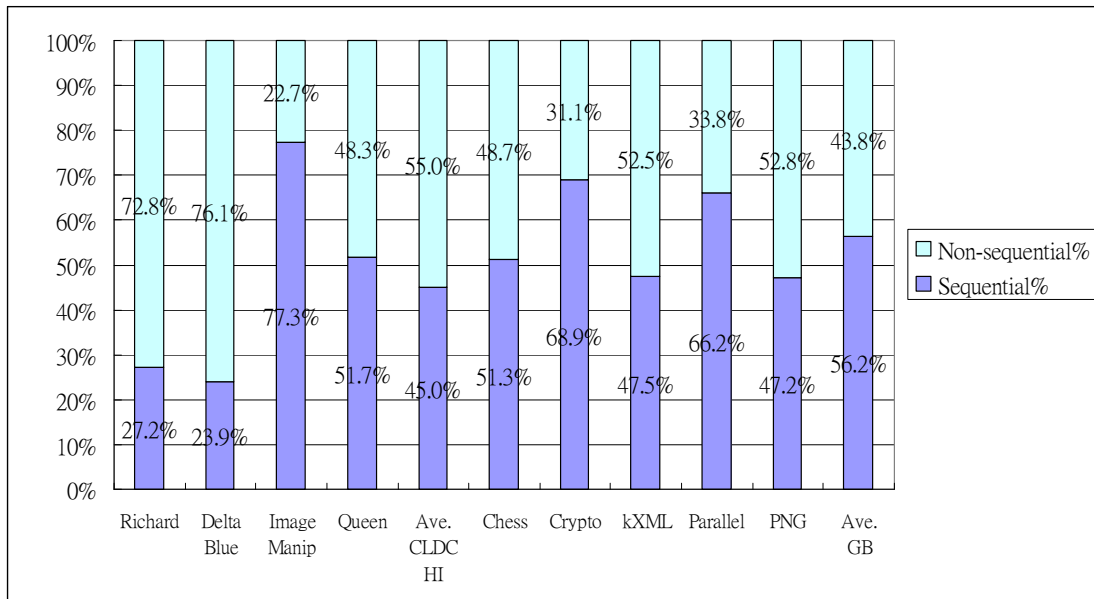
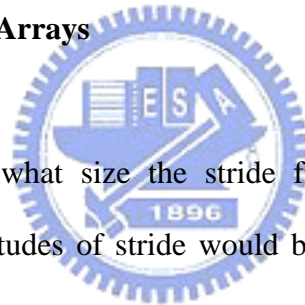


Figure 4.5 Sequential strength of bytecode

4.3.3 Stride Distributions of Arrays



We may concern with what size the stride field of a stride table is needed, or concentrating on what magnitudes of stride would be effective sufficiently if we want to simplify our design. The stride distributions of each benchmark are shown in Figure 4.6. The x axis is the absolute values of strides in bytes; the y axis represents the accumulating proportion of strides. Most magnitudes of strides of the benchmarks are less than or equal to 4 bytes. That is, if our prefetching works for strides less than or equal to 4 bytes, it works for more than 90% of strides.

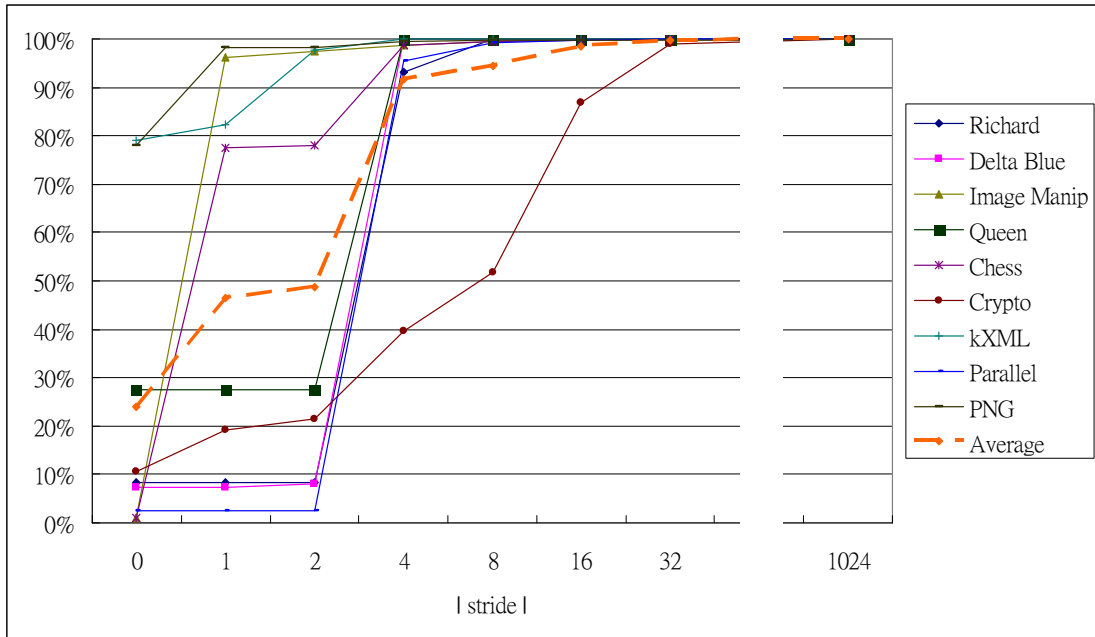


Figure 4.6 Stride distributions of arrays

4.4 Results of Prefetching



In order to evaluate effects of prefetching, we define the *remaining stall ratio* (RSR).

$$\text{Remaining stall ratio}(\text{certain data}) = \frac{\text{Number of stall cycles due to } \textit{certain data} \text{ when its prefetch is enabled}}{\text{Number of stall cycles due to } \textit{certain data} \text{ without prefetching}}$$

For example:

$$\text{Remaining stall ratio}(\text{bytecode}) = \frac{\text{Number of stall cycles due to bytecode when the bytecode prefetching is enabled}}{\text{Number of stall cycles due to bytecode without prefetching}}$$

4.4.1 Prefetching for Bytecode

Firstly, we discuss the size of NLPT and its effects. See Figure 4.7, the x axis represents the number of entries of NLPT and the y axis represents the remaining stall ratio of bytecode. The left-most points of each benchmark are RSR(bytecode)s of sequential

prefetching only. As the table size grows, the RSR(bytecode)s degrade but slightly in most benchmarks. It even has no effect for Queen and gets worse than sequential prefetching for Parallel.

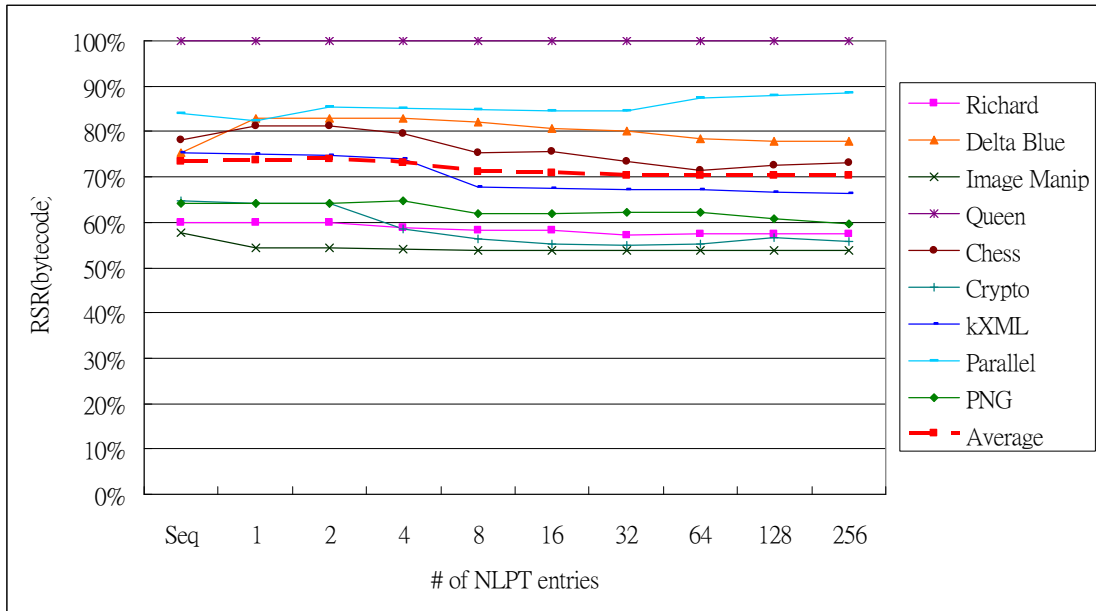


Figure 4.7 RSR(bytecode)s to the sizes of NLPT

Now we see how NBPT performs. See Figure 4.8, we can see that the RSR(bytecode)s start to degrade slowly when the NBPT is larger than 8 to 16 entries. For Queen and PNG, NBPT introduces good stall reductions.

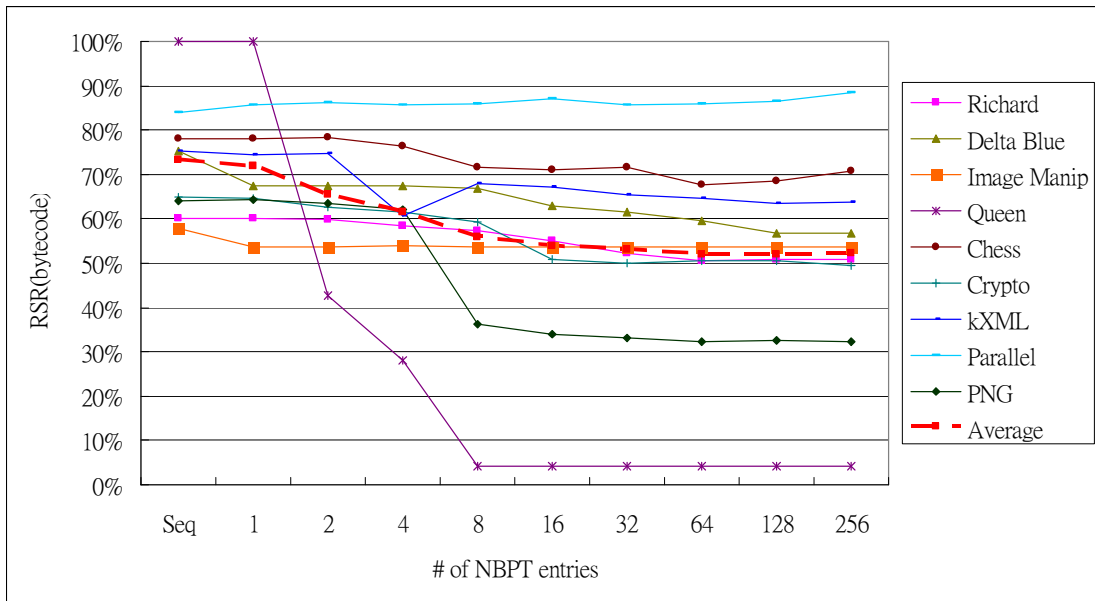


Figure 4.8 RSR(bytecode)s to the sizes of NBPT

In Figure 4.9, we pick the 16-entry NBPT, compare to a 16-entry NLPT and sequential prefetching. When we add a table to record non-sequential crosses rather than sequential prefetching, we can improve the prefetching further. If we adopt NBPT for Java bytecode prefetching, we can obtain better performance than NLPT, especially for Queen and PNG.

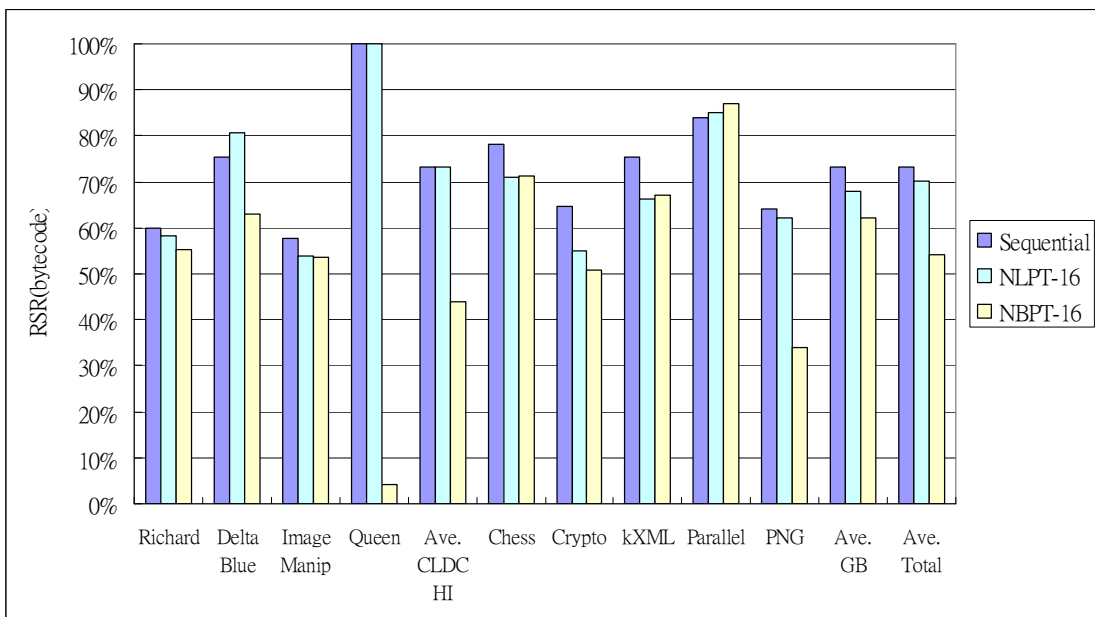


Figure 4.9 A comparison of RSR(bytecode)

4.4.2 Reference Prediction Table for Data Prefetching

Following discusses the effects of reference prediction table(RPT) which for data prefetching. RPT records all load/store instructions, including instance-field accesses, static-field accesses and array accesses. However, our simulations show RPT is only effective for instance-field accesses in some special programs and doesn't have any improvement for static-field accesses. Figure 4.10 shows a 128-entry RPT and its effects for instance fields. Sometimes there are strides between instance-field accesses as indicated in [32]. This property is obvious in Delta Blue, as a result, RPT also introduces a good stall reduction for it. But strides between instances are uncommon in most programs, so RPT usually cannot effectively eliminate the stalls of instance-field accesses.

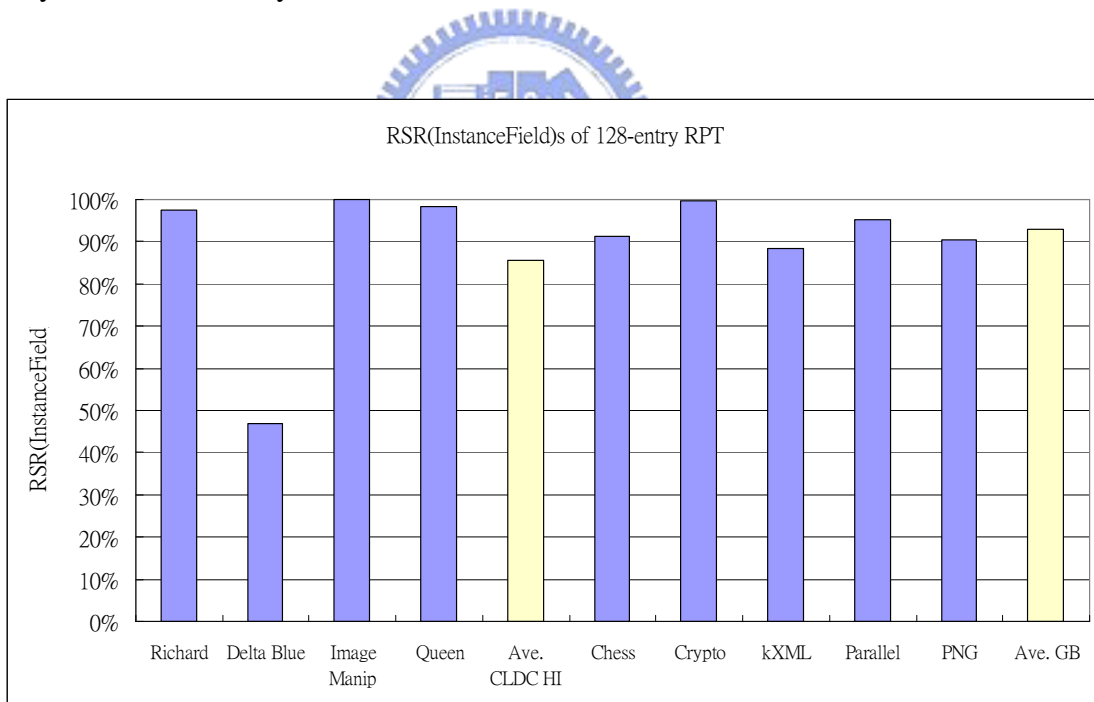


Figure 4.10 RSR(InstanceField)s of 128-entry RPT

Figure 4.11 shows the RSR(array)s to the sizes of RPT. We can see RPT performs well for array prefetching, especially for Delta Blue, kXML and PNG.

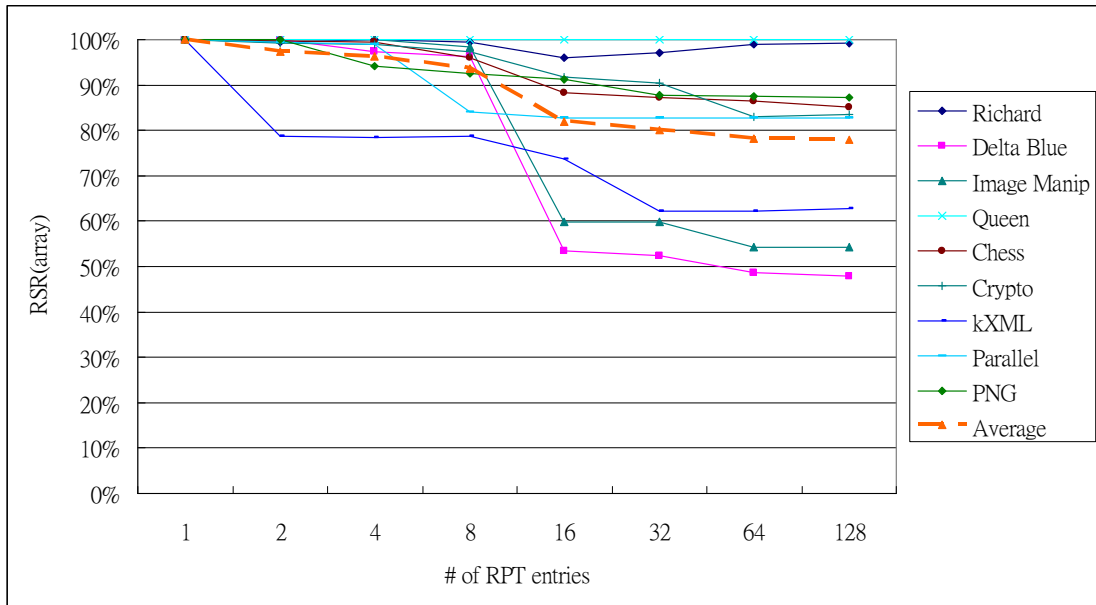


Figure 4.11 RSR(array)s to the sizes of RPT

A possible variation of RPT is, letting RPT only record array instructions since it is usually not effective for other data types. Figure 4.12 shows the effects for array data of the array-only RPT design; Figure 4.13 depicts the average RSR(array)s of original RPTs and array-only RPTs together. Because instructions of other data types occupy spaces in the RPT, unsurprisingly, a small-size array-only RPT performs better than an original RPT which has the same number of entries. However, if we are able to provide a larger size for RPT, their effects will be very close.

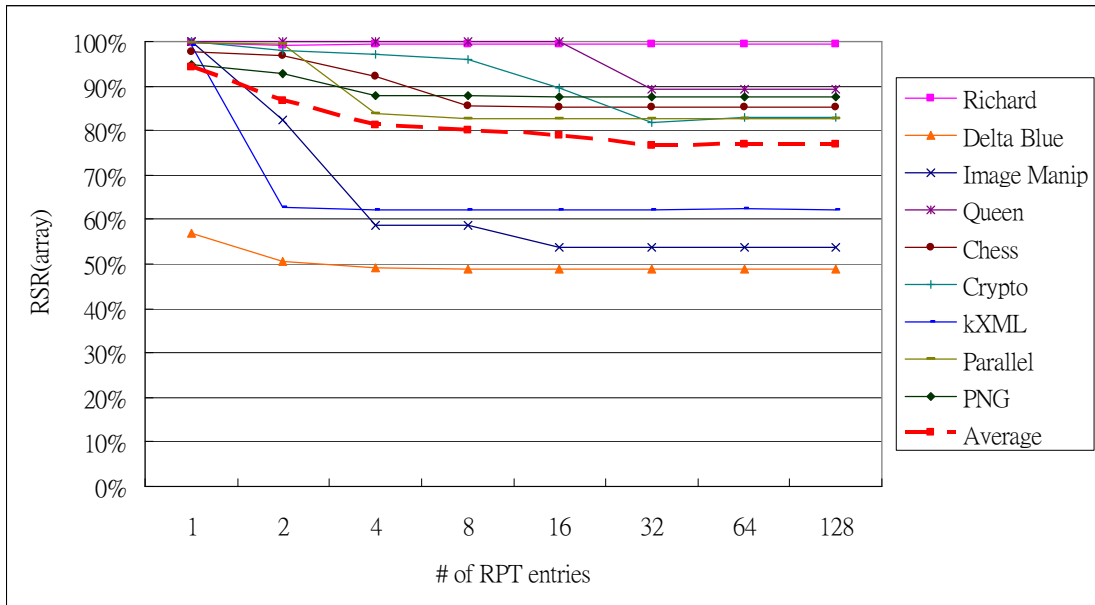


Figure 4.12 RSR(array)s to the sizes of RPT

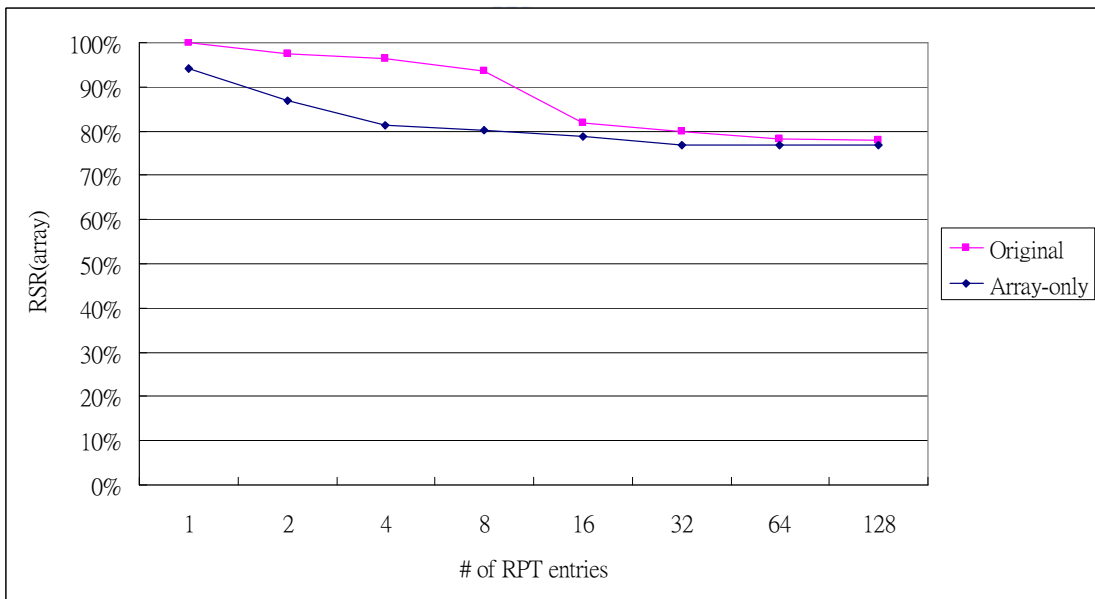


Figure 4.13 Average RSR(array)s of original RPTs and array-only RPTs

4.4.3 Stride Table for Array Prefetching

Firstly, we may care about what values of the predefined H and the *prefetch depth* of ST should be. These 2 variables very highly depend on individual program. We may profile

a program offline and embed the appropriate settings into its classfiles. However, we could try to find the appropriate values for most programs by experiments. Figure 4.14 shows the average RSR(array)s of all benchmarks by using 8-entry STs. We can see when the H and the *prefetch depth* both equal to 2, the average RSR(array) would be the minimum. Now we apply $H=2$ and *prefetch depth*=2 to each benchmarks and compare the result to their optimal configurations. See Table 4.1, the differences of RSR(array)s between the recommended configurations and their optimal configurations are less than 1.1%. So we usually can already get good effects when using $H=2$ and *prefetch depth*=2 in comparison to using their individual optimal configurations. Nevertheless, note the appropriate values of these 2 variables may highly depend on the platform.

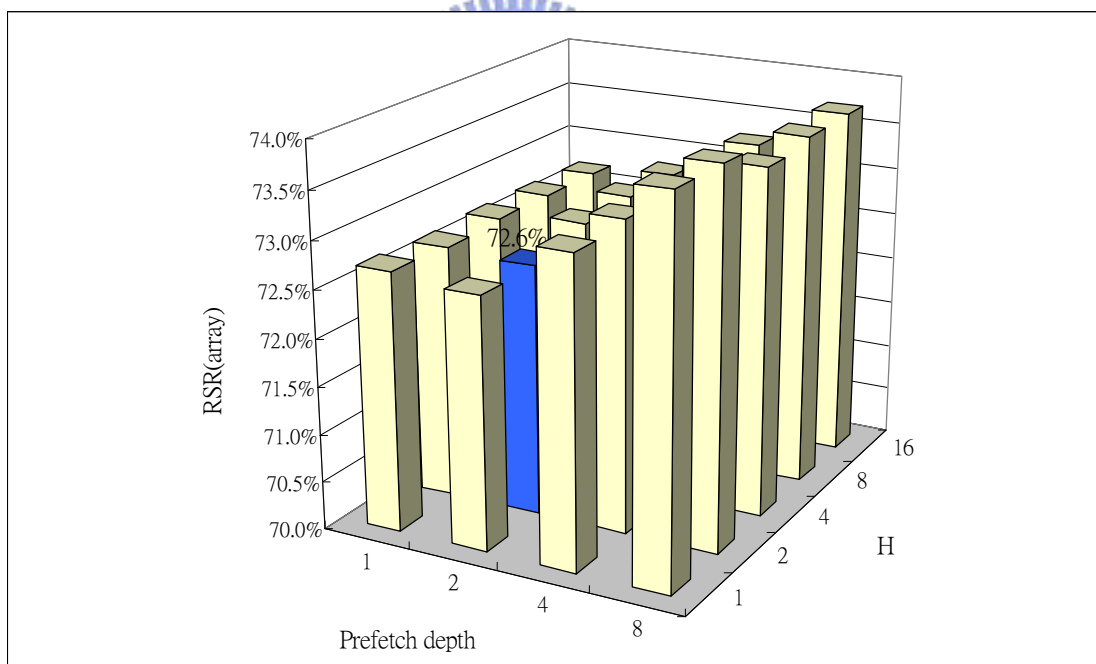


Figure 4.14 Results of configurations of H and *prefetch depth* in ST

| | | | | | |
|--|---------|------------|--------------------|----------|---------|
| Benchmark | Richard | Delta Blue | Image Manipulation | Queen | |
| Optimal (H , $prefetch\ depth$) | (8, 1) | (1, 1) | (16, 1) | (2, 8) | |
| Optimal RSR(array) | 97.544% | 49.493% | 51.574% | 78.921% | |
| RSR(array) when ($H=2$, $prefetch\ depth=2$) | 97.545% | 49.667% | 51.898% | 79.985% | |
| Difference | 0.001% | 0.174% | 0.324% | 1.064% | |
| Benchmark | Chess | Crypto | kXML | Parallel | PNG |
| Optimal (H , $prefetch\ depth$) | (1, 8) | (2, 2) | (2, 2) | (2, 1) | (1, 1) |
| Optimal RSR(array) | 78.002% | 77.257% | 43.528% | 83.171% | 90.905% |
| RSR(array) when ($H=2$, $prefetch\ depth=2$) | 78.687% | 77.257% | 43.528% | 83.260% | 91.838% |
| Difference | 0.685% | 0% | 0% | 0.089% | 0.933% |

Table 4.1 RSR(array) differences between using the recommended H and $prefetch\ depth$, and their optimal configurations

Now we may want to know what size a stride table should be. Figure 4.15 shows that the average RSR(array) almost doesn't degrade if the stride table is larger than 8 or 16 entries. So a stride table has 8 to 16 entries is usually sufficient for most embedded Java programs.

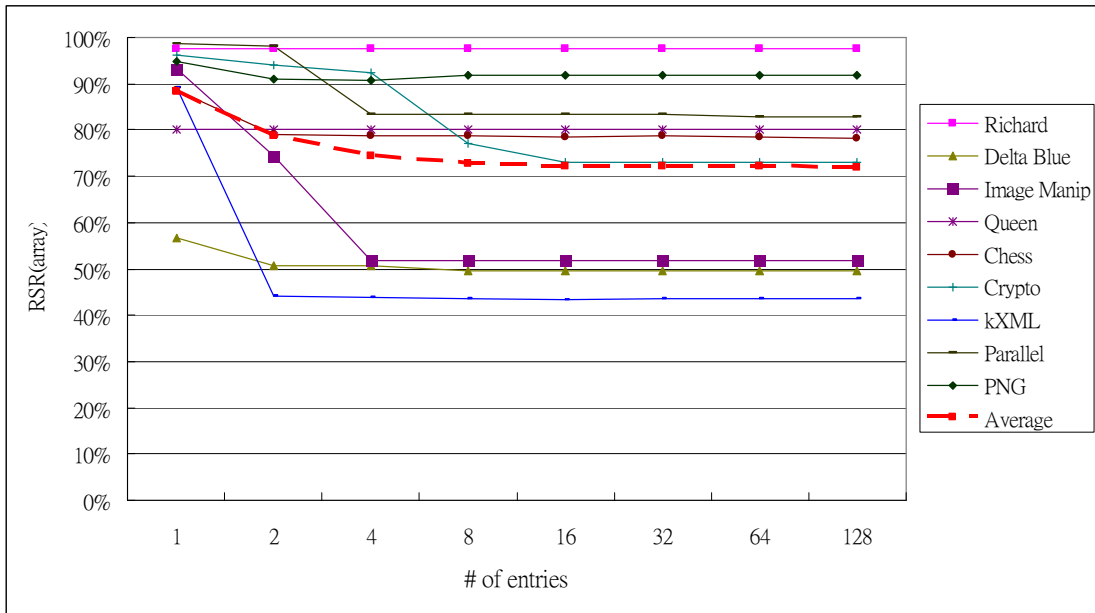


Figure 4.15 RSR(array)s to the sizes of ST

If we use program counter for tagging, we can see the performance of prefetching almost doesn't promote after a 32-entry stride table (Figure 4.16).

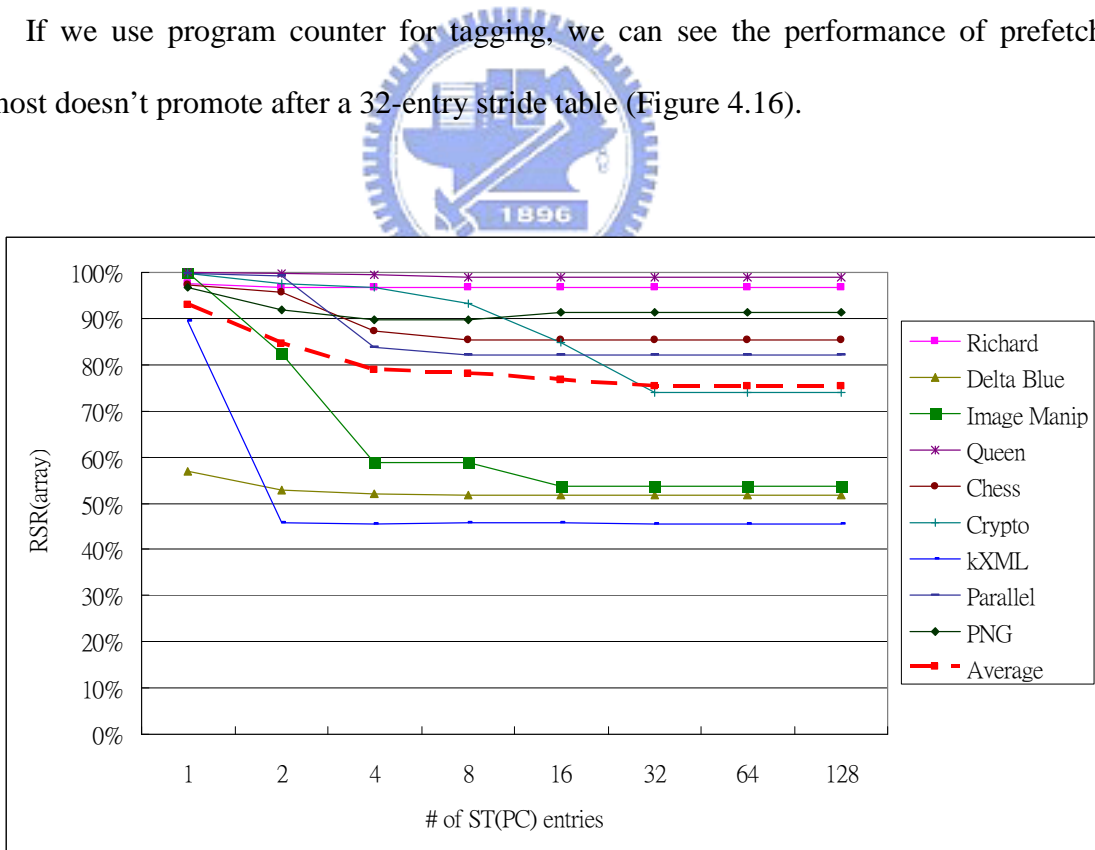


Figure 4.16 RSR(array)s to the sizes of PC-tagged ST

Then we compare the array-base-tagged ST and the PC-tagged ST, the simulation

result is shown in Figure 4.17. In the condition of the same number of entries, an array-base-tagged ST usually has a lower RSR(array) than a PC-tagged ST. The possible reason we have discussed in Subsection 3.2.4. However, if most strides appear only in individual instruction but not arrays, a PC-tagged ST will be better than an array-base-tagged ST, such as Richard, Parallel and PNG.

ST(both)-256 is a 256-entry ST but tagged by both PC and array base. Both-tagging can eliminate the interferences of several instructions or several arrays to an entry. However, it can only perform better than PC-tagging and array-base-tagging slightly for Richard and Delta Blue. This is maybe because an entry of both-tagged ST needs longer time to get a stable stride.

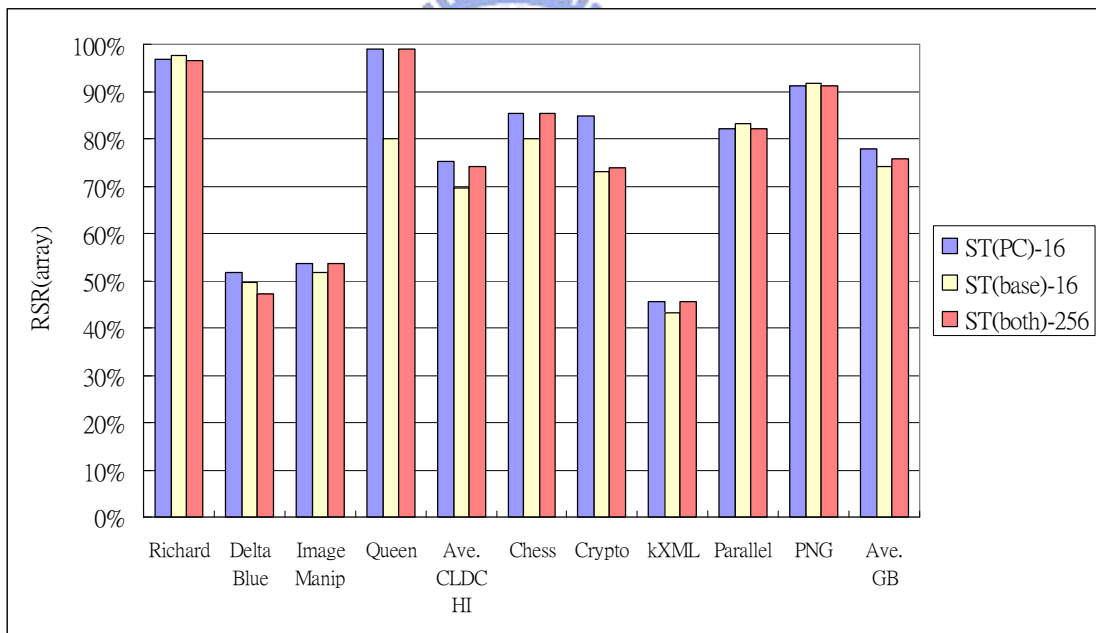


Figure 4.17 Comparison of tagging approaches for ST

In following we compare the efficiencies of ST to array-only RPT. As Figure 4.18, ST usually obtains better array stall reductions than RPT for Java programs, especially for Queen, Crypto and kXML.

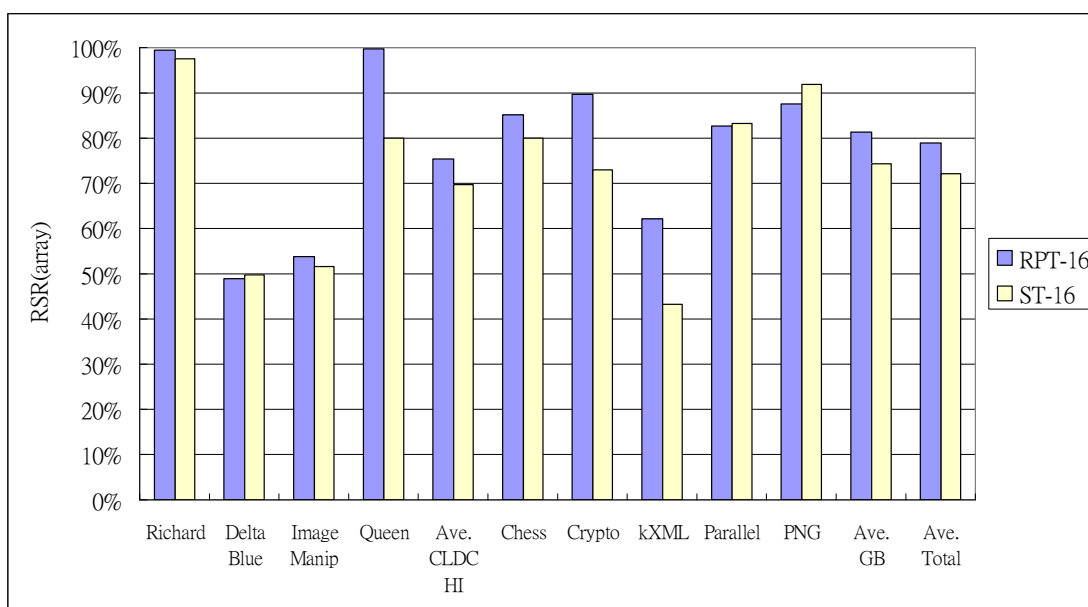


Figure 4.18 Comparisons of ST and array-only RPT

Figure 4.19 presents the effects of each design idea of ST. The bars of “Array-base” are RSR(array)s of RPT but adopting array-base-tagging and our 2-state design. We can see array-base-tagging is effective for Queen and Crypto, although it is a little bad for Delta Blue, Chess and PNG. The bars of “Base+S” are RSR(array)s when using array-base-tagging plus stride-adaptive prefetching. The stride-adaptive approach is much effective for Chess and kXML, but gets worse for Richard, Crypto and PNG. “Base+S+C” is base-tagging plus stride-adaptive prefetching and circular prefetching; that is, our final ST design. The results show circular prefetching slightly improves the performance of prefetching. But note that no design absolutely suits every case and sometime may result in negative effects.

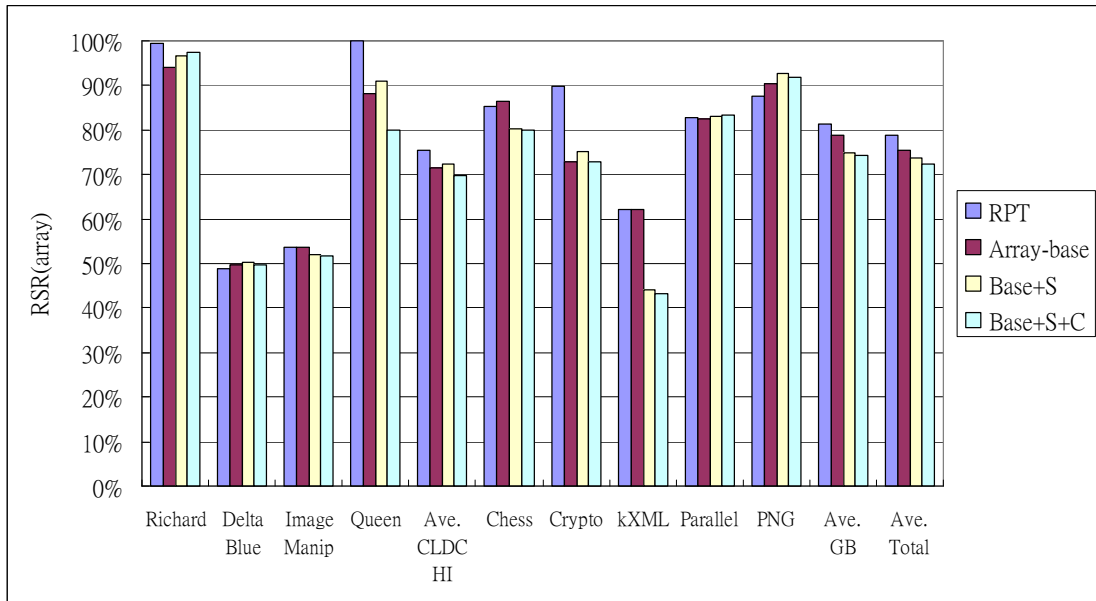


Figure 4.19 Effects of each design idea of ST

In conclusion, our design can achieve better performance for array prefetching than RPT. On an average, ST is 6% better for Sun's CLDC HI and 8% better for EEMBC's GrinderBench than RPT in RSR(array).

Finally, Figure 4.20 shows the fractions of unnecessary prefetch signals that can be eliminated by trigger-block. The trigger-block design eliminates more than 50% of unnecessary signals for Image Manipulation, kXML and PNG; 17.2% for Sun's CLDC HI and 33.9% for EEMBC's GrinderBench in average.

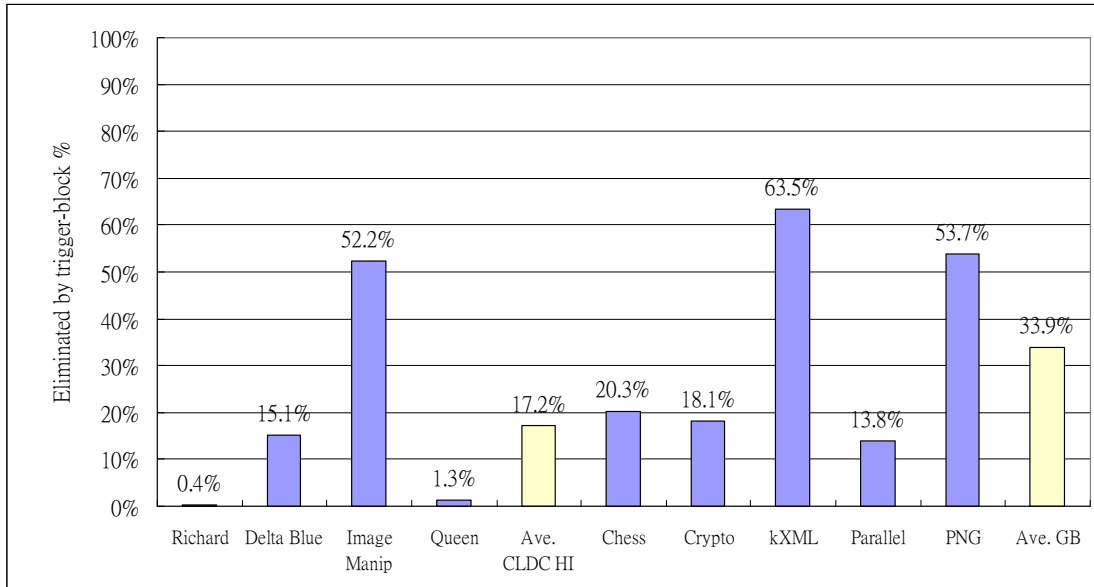


Figure 4.20 Effects of trigger-block

4.5 Analysis of Memory Traffic

Besides stall reduction, another issue we may concern with is memory traffic. Useless prefetches will produce additional traffics.

Firstly, we define some terms. A *true miss* is a memory request where the data accessed is not found either in the cache or in the prefetch buffer. If a prefetched block is really required and submitted into the cache, we call the prefetch a *useful prefetch*; otherwise, it is called an *unused prefetch*. An unused prefetch is never needed before being replaced out from the prefetch buffer. The memory traffic caused by bytecode fetches or prefetches is called *bytecode traffic*; the *array traffic* is similar.

4.5.1 Bytecode Traffic

We compare the bytecode traffics of the sequential prefetching, a 16-entry NLPT and a 16-entry NBPT. The bytecode traffic without prefetching is normalized to 100%. Their

traffics are shown in Figure 4.21, as well as the traffics resulted by useful prefetches (U.P) only. The fractions larger than 100% are caused by unused prefetches and the rests are true misses. As we see, the traffic resulted by the 16-entry NBPT is a little larger than the 16-entry NLPT in some benchmarks, but no more than the sequential prefetching. However, the useful prefetches produced by the NBPT-16 are more than both the NLPT-16 and the sequential prefetching in average, especially for Queen and PNG. So despite of including the effects of unused prefetches, the performance of NBPT is still better than NLPT.

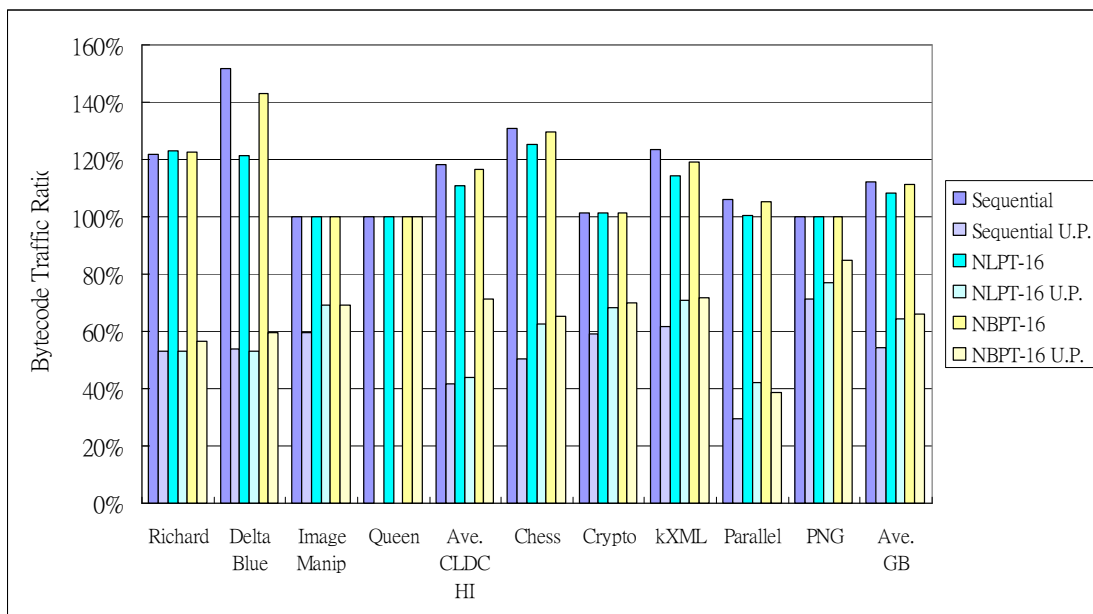


Figure 4.21 Bytecode traffic

4.5.2 Array Traffic

Similarly, we compare the array traffics of a 16-entry RPT and a 16-entry ST. See Figure 4.22, 100% are the array traffics without prefetching. We can find the RPT design results in almost no additional traffic. This is because RPT is very conservative and only prefetch with high confidence. ST does prefetch a little earlier than RPT, and also issues more tentative prefetches. However, ST can issue more useful prefetches, especially for

Queen, Chess, Crypto and KXML, to achieve more stall reductions. So if the prefetch buffer is absent or the contention is significant, the conservative policy of RPT is worth of being used or we can just disable the tentative prefetches of ST; otherwise, the aggressive policy of ST can provide a better performance. A possible variation will be described in Subsection 5.2.2.

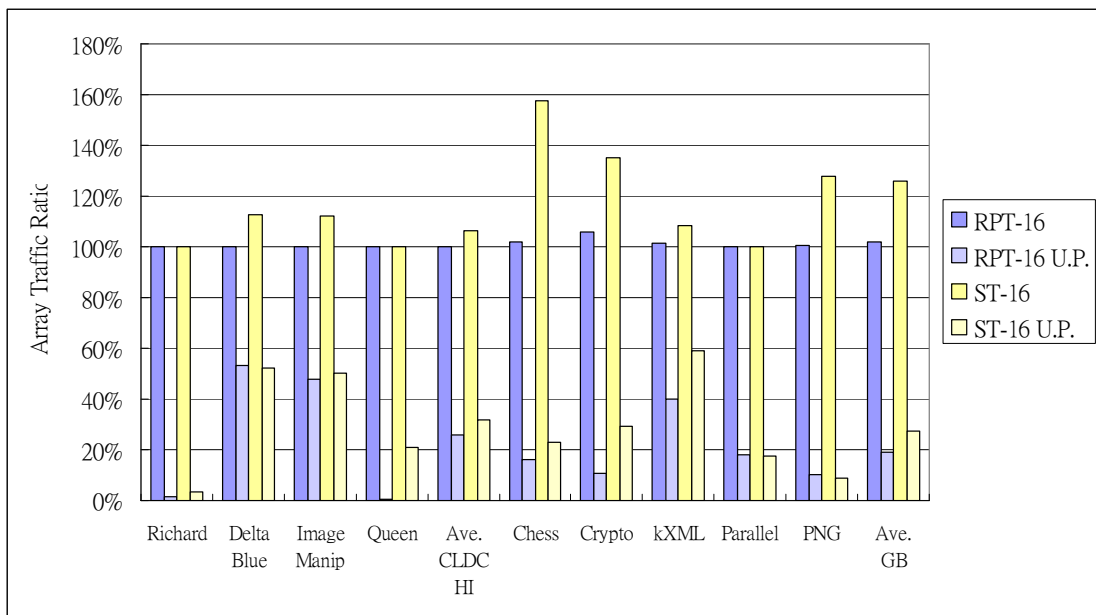


Figure 4.22 Array traffic

Chapter 5

Conclusion and Future Works

This chapter includes the conclusion and the discussions of some future works in Section 5.1 and 5.2, respectively.

5.1 Conclusion

As the continuously growing of multimedia applications, the requirement of memory is increasing because of their large amount of code and data. This problem also exists in embedded devices. In order to reduce memory stall time and speedup execution, prefetching is a feasible solution. We studied bytecode and array prefetching approaches for Java hardware accelerators. Because there are usually more small method invocations in a Java program, NBPT has some subtle designs to handle them. Strides exist between array accesses, we indicated using array-base to tag stride entries is an alternative approach to PC-tagging for Java. By cooperating to our 2-state design and stride-adaptive algorithm, it performs better than the PC-tagged RPT. We also had some analysis on Sun's CLDC HI and EEMBC's GrinderBench benchmarks. On an average, NBPT can reduce 40% of time spent on bytecode stalls. The ST design can reduce 25% of array stall time; for some array-based programs, around 50% of array stall time is eliminated.

We can try to apply our mechanisms to mixed-mode JVMs in advanced environments. A mixed-mode JVM has a selective JIT compiler. It detects hotspots in running Java programs, and compiles them into machine code dynamically. For non-compiled code, it

still executes them by interpretation. A hardware accelerator can also be used to accelerate the interpretation.

5.2 Future Works

This section includes some possible variations and applications of our mechanisms, and future study directions.

5.2.1 Prefetching More Bytecode Blocks at a Time

Note that NBPT only makes a prediction for the next continuously-fetched block. In case of shorter memory latency, it's adequate that we issue prefetch for the next block. However, if the memory latency is longer or the accelerator is improved further, the arrival time of our prefetched block may be too late to hide the memory stall. Thus we may want to initiate the prefetch earlier.

For simply, suppose we have a block sequence ... A, B, C..., where B is in the cache and we merely have to prefetch block C. If the memory latency is not too long, we can initiate the prefetch of C when entering B as Figure 5.1 (a). However, if the memory latency is longer, our prefetch for C would arrive too late, so that there is still a period of stall does not be hided (Figure 5.1 (b)). Thus we may want to issue the prefetch for C earlier, for example, at the entry of A (Figure 5.1 (c)). In this case, we may assume that block B is already in cache or has been prefetched, or also try to prefetch B and let C be a little postponed. Note that since all prefetches will check the cache and the prefetch buffer before being really issued out to the memory, the prefetches for blocks present in the cache or the buffer won't degrade the performance of prefetching too much.

To predict C, we can have 2 different policies. One is looking up the NBPT only once

for B and assume (B, C) is sequential, in other words, our prediction is $NBPT[A]+1$. The other is looking up the NBPT twice continuously, make a prediction for B and then for C, i.e., predict $NBPT[NBPT[A]]$ for C.

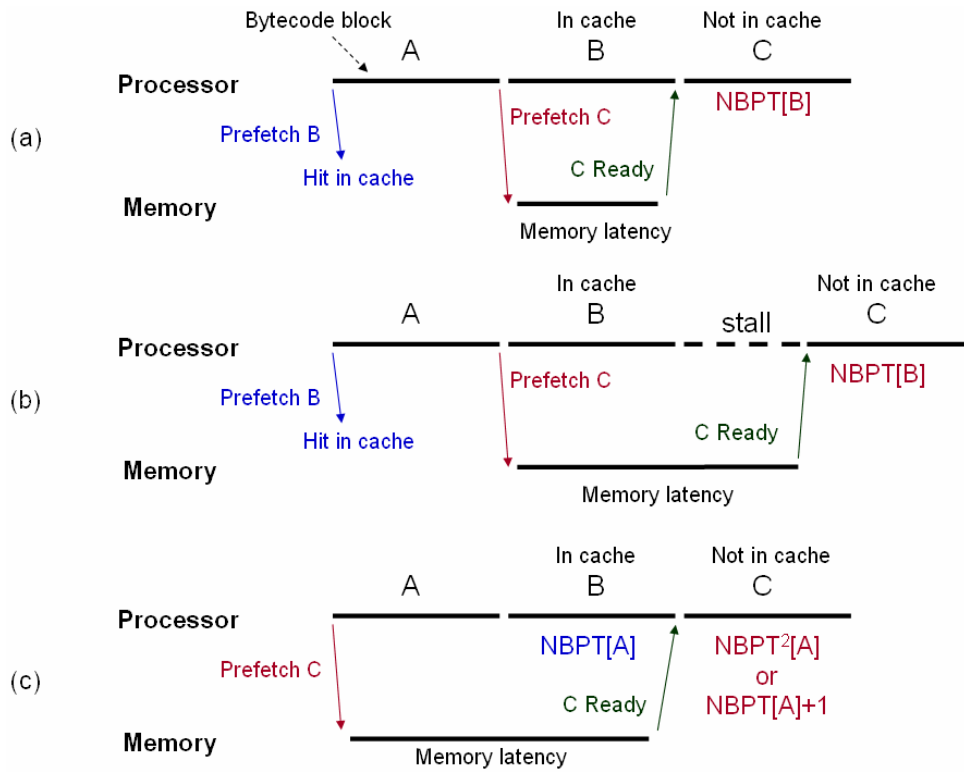


Figure 5.1 Timing issue of bytecode prefetching

(a) Short memory latency (b) Prefetch too late (c) Prefetch earlier

More generally, if the program counter is on block A currently and we want to prefetch certain block B which is n blocks later. Suppose the accuracy of NBPT prediction for one later block is p , the accuracy of sequential prediction for next block is q , and the accuracy of our prediction for B denoted by r . We can have 2 policies:

- Single lookup of NBPT

We only look up the NBPT once and assume the followings are sequential. That is, our prediction

$$b_s = NBPT[A] + (n-1)$$

Thus the accuracy of our prediction

$$r_s = p \cdot q^{n-1}$$

In this case, the prediction could be obtained instantly after the lookup.

- Multiple lookups of NBPT

We look up the NBPT repeatedly. Our prediction is

$$b_m = \text{NBPT}[\text{NBPT}[\dots \text{NBPT}[A] \dots]] = \text{NBPT}^n[A]$$

And its accuracy

$$r_m = p^n$$

In this case, the prediction will be known after n lookups.

Note that both r_s and r_m are exponential functions of n and usually diminish quickly, so it is only effective if a small n and the prediction is very accurate. In case that we didn't predict correctly and prefetched a block not required, the arrival time of useful prefetches will be delayed; or even some useful prefetches already in the buffer are discarded.

For processors that have less memory latency, prefetching several possible paths into the buffer may be a possible implementation.

5.2.2 Adaptive Mechanisms

As we have described in Subsection 2.2.4. If the contention on timing or prefetch buffer is slight, we can issue more tentative prefetches; otherwise, we may deteriorate the contention and degrade the performance of prefetching. Thus, we can switch between conservative or aggressive policies by monitoring contentions. Or we may track the utilization of tentative prefetches to decide what policy we should select. If the utilization of tentative prefetches is low, we may change to use a conservative policy and issue less tentative prefetches to keep a fine work.

In another aspect, some design idea may not suit certain applications. We may develop

some detection mechanisms to collect program behavior, to choose a suitable algorithm or enable/disable unsuitable prefetch mechanisms.

5.2.3 Prefetching for Other Data Types

Subsection 2.1.2 lists several data types; however, our prefetching mechanisms only focus on bytecode and array data. The other data types are:

- Instance headers
- Instance fields
- Static fields
- Class structures other than fields

Instance fields and static fields have specific instructions associate to. The data belong to these 2 types usually distribute on the heap randomly and have less regularity. Stalls caused by static field accesses are relative fewer, so perhaps we don't need to prefetch them. Stalls caused by instance field accesses, however, will take a large proportion in instance-based programs. For example, Delta Blue, Chess and PNG will visit plenty of instances during traversals of linked structures. Some researchers try to find compiler solutions by some complex analysis [13, 32], but their approaches have some strict restrictions. So it is still a difficult problem until today because of the huge amount of instances. Instance headers and classes structures other than fields are usually accessed implicitly. The difficulty of prefetching instance headers is the same as instance fields. Class structures are relative large structures, a class structure usually occupies several blocks. The regularities of their accesses are unobvious and difficult to catch. [1] proposed a JIT-compiler approach, by analyzing object metadata and the aid of a hardware monitor for misses, to inject prefetch instructions into compiled code.

In the future, we may dynamically profile these data types by hardware and launch a

software analyzer to analyze them. By co-working of hardware and software, similar to [1], perhaps there will be more chances for dynamic prefetching.

5.2.4 Next-Block Prediction for Low Power Caches

The prediction approaches of our designs can also be used to other fields. For example, line-decayed cache [16] and drowsy cache [10] are designed to reduce leakage power of cache. Our prediction approaches may be applied to prefetch data from the next-level cache into a decayed cache line, or pre-activate cache lines for a drowsy cache, so that the performance will not degrade too much in a low-power cache.



References

- [1] A. Adl-Tabatabai et al, "Prefetch Injection Based on Hardware Monitoring and Object Metadata." In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 267-276, 2004.
- [2] aJile, "aj-100 Real-time Low Power Java Processor." Preliminary data sheet, 2000.
- [3] ARM Jazelle, http://www.arm.com/products/esd/jazelle_home.html
- [4] ARM, "Jazelle DBX Technology: ARM Acceleration Technology for Java Platform," *Jazelle DBX white paper*, 2005.
- [5] J.-L. Bare and T.-F. Chen, "An Effective On-chip Preloading Scheme to reduce Data Access Penalty." In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp.176-186, Nov, 1991.
- [6] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors." In *IEEE Transactions on Computers*, Vol. 44, No. 5, May 1995.
- [7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko, "Compiling Java Just In Time." In *IEEE Micro*, pp. 36-43, 1997.
- [8] M. W. El-Kharashi and F. Elguibaly, "Java Microprocessors: Computer Architecture Implicants." In *IEEE Communications, Computers and Signal Processing*, Vol. 1, pp. 277-280, 1997.
- [9] EEMBC's GrinderBench, <http://www.grinderbench.com/>
- [10] K. Flautner and et al, "Drowsy Caches: Simple Techniques for Reducing Leakage Power." In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, pp. 148-157, 2002.
- [11] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative*

Approach 3ed, Morgan Kaufmann Publishers, 2003.

- [12] W.-C. Hsu and J. E. Smith, "A Performance Study of Instruction Cache Prefetching Methods." In *IEEE Transactions on Computers*, Vol. 47, No. 5, pp. 497-508, May 1998.
- [13] T. Inagaki et al, "Stride Prefetching by Dynamically Inspecting Objects." In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 269-277, Jun 2003.
- [14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." In *Proceedings of the 17th annual international symposium on Computer Architecture*, Vol 18, Issue 3a, pp. 364-373, 1990.
- [15] T. Kistler and M. Franz, "Automated data-member layout of heap objects to improve memory-hierarchy performance." In *ACM Transactions on Programming Languages and Systems*, Vol. 22, Issue 3, pp. 490-505, Oct 2000.
- [16] S. Kaxiras and et al, "Cache-line Decay: A Mechanism to Reduce Cache Leakage Power." In *IEEE Workshop on Power Aware Computer Systems 2000*, pp. 82-96, 2000.
- [17] F. Li, P. Agrawal, G. Eberhardt, E. Manavoglu, S. Ugurel, and M. Kandemir, "Improving Memory Performance of Embedded Java Applications by Dynamic Layout Modifications." In *Preceeding of the 18th International Parallel and Distributed Processing Symoisium*, pp. 159-166, 2004.
- [18] T. Linholm and F. Yellin, *The Java Virtual Machine Specification 2ed*, Sun Microsystems, 1999.
- [19] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java Virtual Machine In Hardware." In *IEEE Micro*, pp. 45-53, 1997.
- [20] T.A. Proebsting et al., "Toba: Java for Applications. A Way Ahead of Time (WAT) Compiler." In *Proc. 3rd USENIX Conference on Object-Oriented Technologies and*

Systems (COOTS), pp. 41-53, 1997.

- [21] S. Rubin, R. Bodik, and T. Chilimbi, “An Efficient Profile-Analysis Framework for Data-Layout Optimizations.” In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Vol. 37, Issue 1, pp.140-153, Jan 2002.
- [22] R. Riggs, A. Taivalsaari, and M. VandenBrink, *Programming Wireless Devices with the Java 2 Platform, Micro Edition*, Pearson Education, 2001.
- [23] M. Schoeberl, “JOP: A Optimized Java Processor.” In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pp. 346-359, 2003.
- [24] M. Schoeberl, “Evaluation of a Java Processor,” In *Tagungsband Austrochip 2005*.
- [25] Sun Microelectronics, “picoJava-I Microprocessor Core Architecture,” white paper, 1996.
- [26] Sun Microsystems, “picoJava-II: Java Processor Core,” write paper, 1997.
- [27] Sun Microsystems, Java Technology, <http://java.sun.com/>
- [28] H. Tomiyama and H. Yasurra, “Code placement techniques for cache miss rate reduction.” In *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, Issue 4, pp. 410-429, 1997.
- [29] Y. Y. Tan et al, “Design and implementation of a Java processor.” In *Computers and Digital Techniques, IEE Proceedings*, Vol. 153, Issue 1, pp. 20-30, Jan 2006.
- [30] S. P. Vanderwiel and D. J. Lilja, “Data Prefetch Mechanisms.” In *ACM Computing Surveys*, Vol. 32, No. 2, Jun 2000.
- [31] T. Y. Yeh and Y. N. Patt, “A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Instruction.” In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium*, pp. 129-139, 1992.
- [32] Y. Wu, “Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching.” In *Proceedings of the ACM SIGPLAN 2002 Conference on*

Programming language design and implementation, pp. 210-221, 2002.

