國立交通大學

資訊工程系

碩士論文

快取誤失類型辨認
及其在動態預測快取誤失位置之用途

Cache Miss Type Identification
and Its Use in Dynamic Miss Address Prediction

研 究 生：葉文涵

指導教授：鍾崇斌　教授

中華民國九十五年八月

快取誤失類型辨認

及其在動態預測快取誤失位置之用途

Cache Miss Type Identification
and Its Use in Dynamic Miss Address Prediction

研　究　生：葉文涵　　　　Student：Wen-Han Yeh

指導教授：鍾崇斌　　　　Advisor：Chung-Ping Chung

國立交通大學
資訊工程系
碩士論文

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

Computer Science

August 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年八月

快取誤失類型辨認及其在動態預測快取誤失位置之用途

學生：葉文涵　　　　　　　　　　　指導教授：鍾崇斌　博士

國立交通大學資訊科學學系(研究所)碩士班

摘　　要

微處理器的時脈隨著半導體製程與微架構技術的進步快速增加的同時，主記憶體的時脈卻無法以相同的幅度進展，唯一的辦法是使用快取記憶體來填補兩者間越來越大的時脈差距。因此快取記憶體的誤失(cache miss)影響系統的效能甚巨。各種快取架構加速機制雖然能降低快取誤失造成的延遲時間，但通常一種快取架構加速機制只能針對某種特定的快取誤失類型。在本篇論文，我們改進現有靜態(static time)辨認快取誤失類型的機制：將虛擬快取的置換策略改為有限預知置換法，使辨認結果更加正確。同時，我們也利用不同誤失類型具有不同的誤失長度與頻率的特性，提出適合在執行時期(run time)辨認快取誤失類型的新方法，其正確性與 static time 相比，達到 93%的正確性，並且較之前相關研究之方法更省硬體資源，也更正確。

此外，我們也示範如何利用快取誤失類型來改善快取的效率，並以動態預測快取誤失做為例子：當我們使用一種對於特定的快取誤失類型較為有效的加速機制時，可以針對此種快取誤失類型再啟動此加速機制；最後，我們整合數種快取加速機制並將它們所預測的快取區塊放入一個共享的緩衝區，稱之為 PV buffer。在我們的實驗中它能有效改善 89%的快取誤失造成的延遲。我們並利用執行時期快取誤失類型辨認的結果，來改善其緩衝區的使用效率，以及降低多種加速機制所造成的記憶體階層流量增加的問題。

Cache Miss Type Identification and Its Use in Dynamic Miss Address Prediction

Student：Wen-Han Yeh　　　　Advisor：Chung-Ping Chung

Department of Computer Science

National Chiao Tung University

Abstract

Semiconductor technology and micro-architecture evolutions are driving micro-processors toward higher clock frequencies. Meanwhile, main memories hadn't significantly reduced access time. To prevent large performance losses caused by long main memory latencies, micro-processors rely heavily on cache memories. Unfortunately, cache memories are not always effective due to the various cache misses. To overcome cache memories' limitations, there are several cache-based architectural optimizations which can reduce the miss penalty when cache misses. But some Optimizations cannot handle all types of misses well.

In this thesis, we improve existing static time cache miss type identification scheme by using finite look-ahead replacement policy in the pseudo-cache to make identification results more accurate. We also propose two low hardware cost, low complexity cache miss type identification approaches which achieve more than 93% average identification accuracy. Then, we demonstrate the application of run time cache miss type identification by applying it to several cache-based architectural optimizations. In each case, the architecture benefits from applying different policies to different types of misses. In addition, we combine several cache optimizations to cover 89% of cache misses with a sixteen-entry buffer, called PV buffer. By using cache miss type information, we can reduce unnecessary memory traffic and fetch operations to increase effectiveness of this cache-assist buffer.

Table of Contents

中文摘要 i

Abstract iii

Table of Contents iii

Lists of Figures v

List of Tables vi

Chapter 1 Introduction 1

1.1 The Three C in cache misses 3

1.2 Identification of Conflict misses 4

1.3 Research Motivation 7

1.4 Research Goal 8

1.5 Organization of this thesis 9

Chapter 2 Related Works and Backgrounds.. 10

2.1 Related Works – Runtime Identification of Cache Conflict Misses: The Adaptive Miss Buffer 11

2.1.1 Using MCT to classify misses 11

2.1.2 Multiple tags per entry 11

2.1.3 Applications of conflict miss filtering 12

2.2 Backgrounds – Cache Prefetch Methods 13

2.2.1 Sequential prefetching 13

2.2.2 Table-Based prefetching 13

2.2.3 Cache Prefetching using a Global History Buffer 15

2.3 Backgrounds – Victim cache 17

Chapter 3 How to Better Identify Cache Miss Types.. 18

3.1 Simulation methodology 18

3.1.1 Experimental flow 18
3.1.2 Experimental benchmarks 19
3.1.3 The instruction driven simulator 20
3.1.4 Trace driven simulator 21
3.2 Using FLA Replacement in Pseudo Cache to Identify Cache Miss Type in Static Time 22
3.3 Run time Approach 1 – Miss Frequency Spectrum 25
3.4 Run time Approach 2 – Miss Distance 29
3.5 Evaluation of Run Time Cache Miss Type Identification ... 33
Chapter 4 Dynamic Cache Miss Address Prediction and PV Buffer 35
4.1 Cooperating the Cache miss type identification and miss address prediction 36
4.2 Filtering Victim Cache with cache miss type 38
4.3 Cache Prefetching with cache miss type 40
4.3.1 Sequential Prefetching 40
4.3.2 Correlation Prefetching 41
4.4 Put it all together – PV Buffer 44
Chapter 5 Conclusion and Future Works 46
Reference 49
Q&A 51

Lists of Figures

Figure 1-1: Norman Jouppi's scheme to identify conflict misses 4

Figure 1-2: An example of the worst case of LRU replacement policy 5

Figure 2-1: MCT stores old tags which were evicted form L1 cache recently 11

Figure 2-2: The structure of Stride Prefetching table 14

Figure 2-3: Markov Prefetching 15

Figure 2-4: GHB Global / Address Correlation 16

Figure 3-1: Experimental flow 18

Figure 3-2: Static time identification results 23

Figure 3-3: Cache miss frequency spectrums 26

Figure 3-4: Identification accuracy in different base frequencies and cooldown periods 28

Figure 3-5: Structure of Miss Distance approach 29

Figure 3-6: Structure of improved Miss Distance approach 30

Figure 3-7: Identification accuracy in different time window sizes and hit numbers ... 31

Figure 3-8: Identification accuracy using different run time approaches 33

Figure 4-1: Block Diagram of our dynamic cache miss address prediction 36

Figure 4-2: Filtering Victim Cache with cache miss type 38

Figure 4-3: Sequential Prefetching accuracy on a direct-mapped L1 instruction cache 40

Figure 4-4: Sequential Prefetching accuracy on a 4-way set-associative L1 data cache 41

Figure 4-5: Simplified Correlation Prefetching 42

Figure 4-6: Correlation Prefetching using 16 and 4 entries 43

Figure 4-7: Prediction Accuracy of PV buffer 44

Figure 4-8: Fetching rate of PV buffer 45

Figure 5-1: an example of cache coherence protocols 47

Figure A-1: Prediction Accuracy per fetch of PV buffer 52

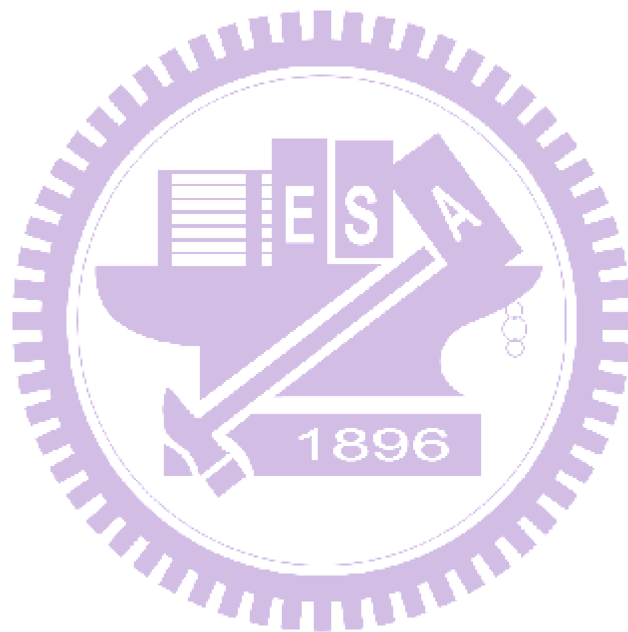
List of Tables

Table 3-1: The Benchmark programs we used form SPEC INT 2000 19

Table 3-2: The processor configuration of Simplescaler 3.0 in our experiment 20

Table 3-3: The memory configuration of Simplescaler 3.0 in our experiment 20

Table 3-4: table size comparison for related works and our approache 34



Chapter 1 Introduction

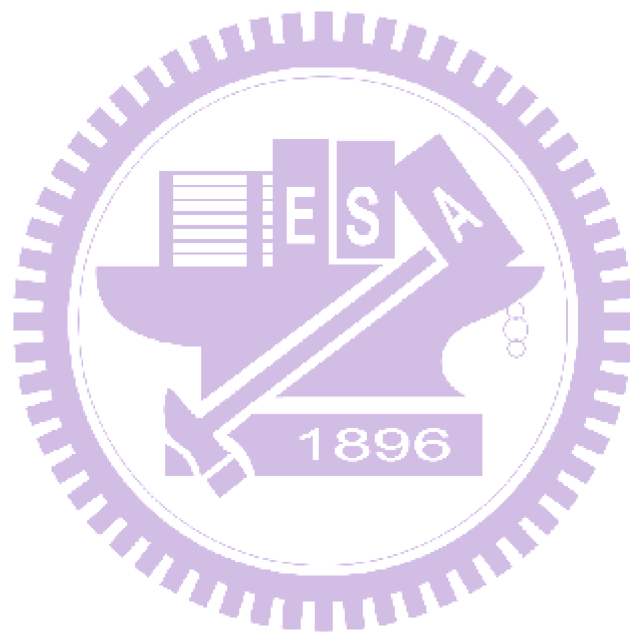
Semiconductor technology and micro-architecture evolutions are driving micro-processors toward higher clock frequencies and higher integration scale. Meanwhile, the technology trend in main memories has been a move toward higher densities rather than significantly reduced access time. Combines together, these trends have relative increased main memory latencies as measured in processor clock cycles. To prevent large performance losses caused by long main memory latencies, micro-processors rely heavily on hierarchies of cache memories.

But cache memories are not always effective, either because they are not large enough to hold a program's working set, or because too many memory blocks in reference stream map to a certain set. But we can't simply increase cache size, nor use more set-associative cache, because these would increase the processor clock period, leading to lower overall performance.

To partially overcome cache memories' limitations, the cache memory controller can predict memory addresses likely to be accessed soon and determine that the data of those addresses are in the cache memory or not. If not, the cache controller can prefetch those blocks which likely to be used soon from lower level of memory hierarchies into the cache. If we can let the prediction precise, we can hide the main memory latencies well. However, we shouldn't do the prediction and prefetching every time we access the cache, it would cause too much time and energy. We should do the prediction only when a cache miss occurs and predict what blocks may cause next cache miss.

As we know, there are three main kinds of cache misses. The fourth type of cache miss called the *coherence misses* will be discussed in Chapter 5. Each cache miss type has different

miss address patterns and has proper way to predict the next cache miss address. If we can identify the miss type when a cache miss occurred, and activate the suitable prediction mechanisms based on that miss type, the prediction results would be more accurate.



1.1 The Three C's cache misses

To gain better insights into the causes of misses, [Hennessy et al. 2003] sorts all misses into three categories:

- *Compulsory* – The very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called *cold misses* or *first-reference misses*.
- *Capacity* – If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because the cache size is not sufficient to hold data between references.
- *Conflict* – If the block placement strategy is set associative or direct mapped, conflict misses will occur because a block may be evicted and retrieved if too many blocks map to its set.

Because compulsory and capacity misses have very similar reference patterns and since we identify miss types for miss address prediction, we group compulsory and capacity misses together to the same type of miss for simplicity. That means misses are classified to fallowing two types in this thesis: *Conflict miss* and *Non-conflict miss*

1.2 Identification of Conflict misses

Although textbook had a definition of conflict misses, we still hard to determined a miss is a conflict miss or not by above definition.

Norman Jouppi, [Jouppi 1995] who proposed a scheme to identify conflict misses (which is very similar to the [Hill 1987]'s definition), said conflict misses are misses that would not occur if the cache was fully-associative and had LRU replacement. So, a particular miss is considered a conflict miss if it would have been a hit in a fully associative cache of the same size.

Figure 1-1 gives a overview that how this scheme works: when a cache miss occurs in the real cache but the virtual fully-associative cache hit, that means the real cache misses because its own organization can't hold the requested block but the fully-associative cache can. In other words, that is a conflict miss. If a requested block couldn't be found in both caches, which have the same size, means it's a very first access or the cache size is not sufficient to hold data between references. In other words, that is a non-conflict miss.

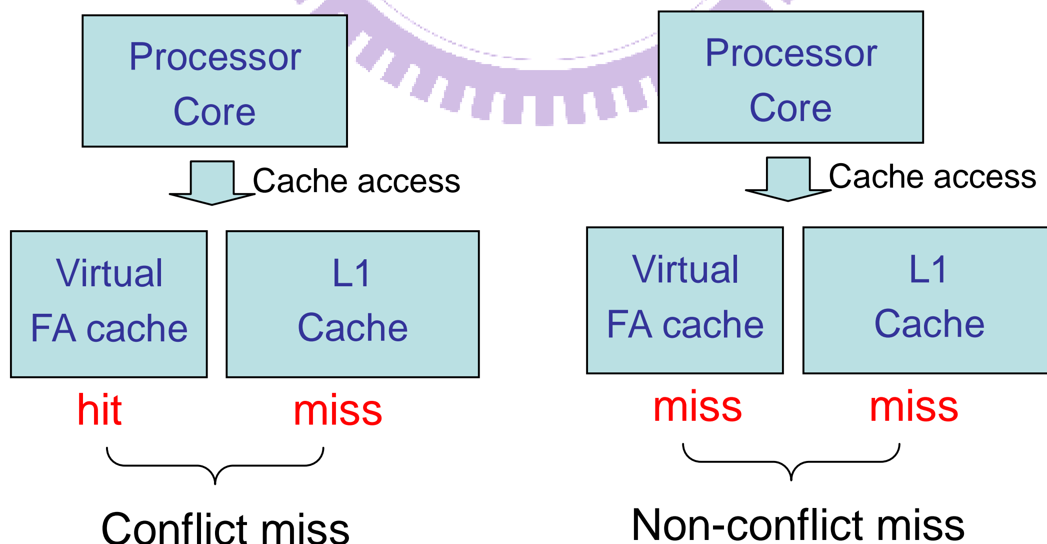


Figure 1-1: Norman Jouppi's scheme to identify conflict misses.

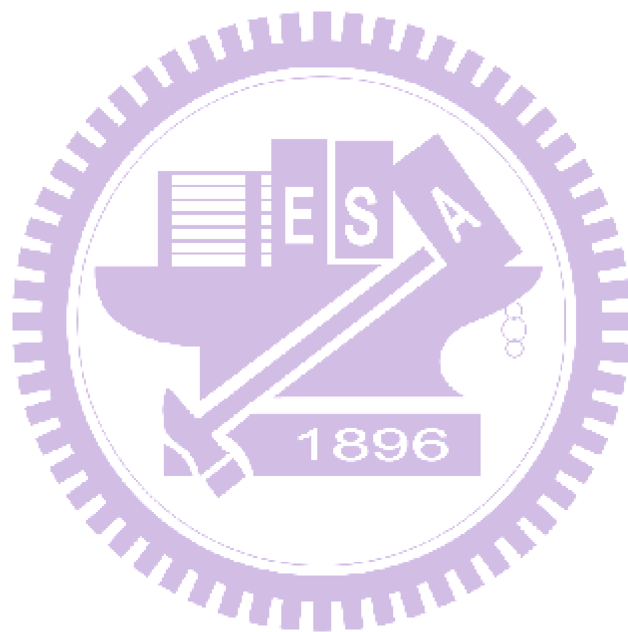
However, Jouppi’s conflict miss identification method shouldn't be done in run-time. It needs an extra fully-associative cache working together with original cache, causes longer cache access time and more energy consumption. The cost of run-time method needs to be lower, and the timing is critical.

Besides, Jouppi used LRU because it is the best-known fully-associative cache replacement policy. But sometimes, a fully-associative with LRU replacement cache may have its own problems. For example: Assume the real cache is a four blocks, direct-mapped cache, and use a four-entries, fully-associative, LRU replacement cache to identify conflict misses. The accessed memory addresses are sequential, five addresses per iteration. Figure 1-2 shows the detail.

Timestamp	Accessed memory address	Fully-associative LRU Cache				Direct-mapped Cache			
1	1	1				1			
2	2	1	2			1	2		
3	3	1	2	3		1	2	3	
4	4	1	2	3	4	1	2	3	4
5	5	5	2	3	4	5	2	3	4
6	1	5	1	3	4	1	2	3	4
7	2	5	1	2	4	1	2	3	4
8	3	5	1	2	3	1	2	3	4
9	4	4	1	2	3	1	2	3	4
10	5	4	5	2	3	5	2	3	4
11	1	4	5	1	3	1	2	3	4
12	2	4	5	1	2	1	2	3	4
13	3	3	5	1	2	1	2	3	4
14	4	3	4	1	2	1	2	3	4
15	5	3	4	5	2	5	2	3	4
16	1	3	4	5	1	1	2	3	4

Figure 1-2: An example of the worst case of LRU replacement policy. A red block means that access was a miss in that cache. A green block means that access was a hit in that cache.

The fully-associative cache misses all the time in this example, because the LRU always replaces the block which will be used in next access. This example is the worst case of LRU replacement policy. In that situation, it can't classify miss types accurately because it will consider all cache misses are capacity misses.

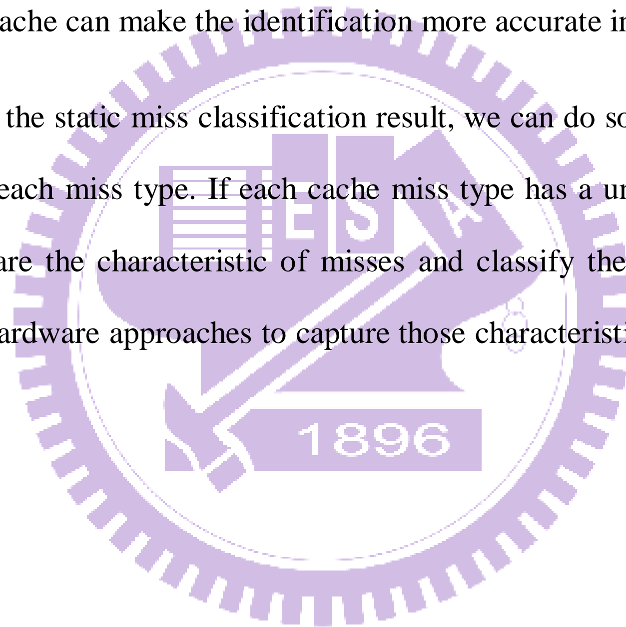


1.3 Research Motivation

Jouppi's conflict miss identification method shouldn't be done in run-time. The cost of run-time method needs to be lower, and the timing is critical. We need to find a more practical and more efficiency scheme to identify conflict misses.

Although Jouppi's scheme is too expensive in run time, it should work fine in simulation time. The worst case mentioned in last section is due to the LRU replacement policy. So, using a finite look-ahead replacement rather than LRU in the virtual fully-associative cache can make the identification more accurate in static time.

Once we have the static miss classification result, we can do some analysis to find the characteristics of each miss type. If each cache miss type has a unique characteristic, we can simply compare the characteristic of misses and classify them. In other words, we want to develop hardware approaches to capture those characteristics to classify misses in run-time.



1.4 Research Goal

There are three main goals in this research:

1. To classify cache misses accurately at static time with low overhead. On one hand, the static time classification results would be used as reference answers compared to the run-time scheme's results. Therefore, the static time results should be the more accurate the better. On the other hand, benchmark problems we used in this thesis would occur more than billions of cache misses. An efficient implementation in simulation time is necessary to classify that huge amount of misses.

2. To develop some run-time practicable miss type classification schemes to approximate static-time cache miss results using low computing and space complexity.

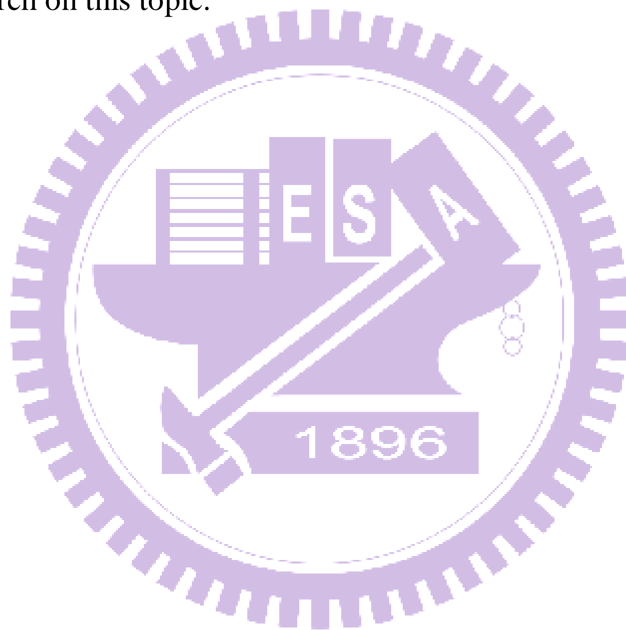
3. To identify possible applications of run-time miss type identification. There are not many previous works on the run-time miss type identification. Therefore, the exploration of possible applications is an essential work.

For example, the run-time cache miss type can be used as a parameter to decide which cache block should be replaced.

Another possible application is using miss type to enhance the performance or efficiency of memory hierarchies. Cache-based architectural optimizations are aimed at particular types of cache misses. Some Optimizations can't handle all types of misses well. If we can identify the cache miss type at run time and activate the proper cache-based architectural optimizations which handle this miss type well. In this research, we chose three different kinds of cache-based architectural optimizations and tried to cooperate with our cache miss type classifier for the example.

1.5 Organization of this thesis

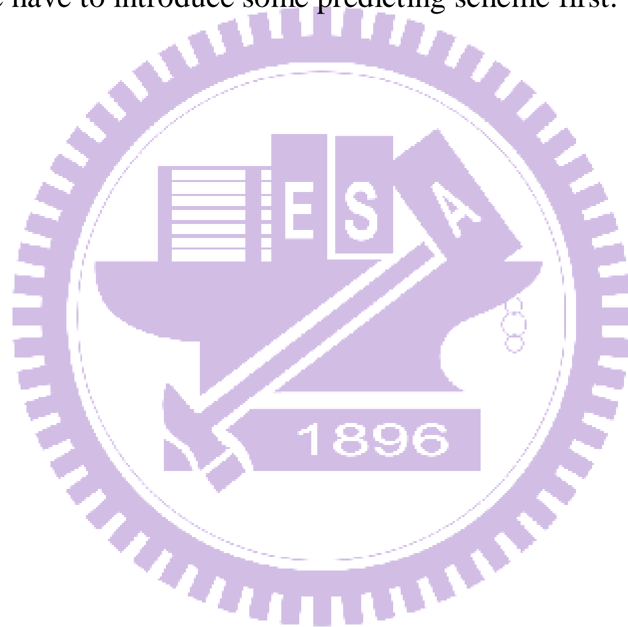
The remaining chapters of this thesis are organized as follows. In the next chapter, we provide the related work of runtime conflict misses identification and necessary background for cache miss address prediction. In Chapter 3, we present the ideas, designs and evaluations of static time and run time cache miss type identification. We demonstrate how the run time miss type identification cooperates with several cache optimizations by using cache assist buffers in Chapter 4. Finally, in Chapter 5, we summarize our major contributions and give an outline of future research on this topic.



Chapter 2 Backgrounds and Related Works

In this chapter, we present previous works of run-time cache miss type classification. In fact, there are not many researches about run-time cache miss type classification. In Nov 2001, Collins and Tullsen introduced a runtime identification of cache conflict misses. Their ideal and motivation are very similar to us.

We also provide the necessary background for cache miss address prediction in this chapter. Because we want to use run-time miss type classification to enhance the cache miss address prediction, we have to introduce some predicting scheme first.



2.1 Related Works - Runtime Identification of Cache Conflict Misses: The Adaptive Miss Buffer

This paper presents a technique to classify misses as either conflict misses or capacity misses at runtime. They use a Miss Classification Table (MCT) which may separate from the cache to store the extra tags.

2.1.1 Using MCT to classify misses

On the subsequent miss to this cache set, the tag of the newly accessed line is compared the tags of the most recently evicted lines from that set. If they are identical, it identifies this miss as a conflict miss.

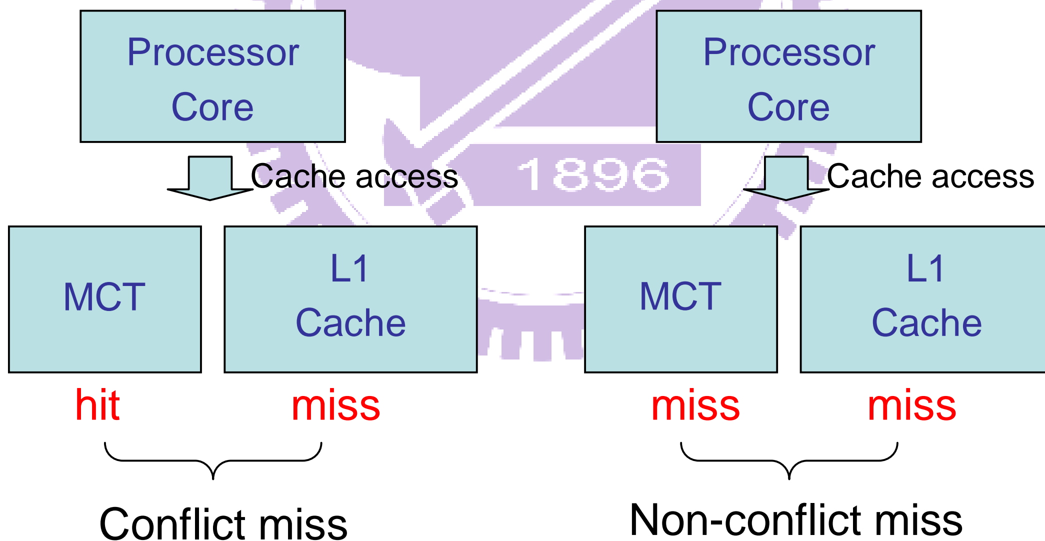


Figure 2-1: MCT stores old tags which were evicted form L1 cache recently.

2.1.2 Multiple tags per entry

Storing one old tag per MCT entry, MCT is able to identify short term conflict behavior by identifying cache misses which would have been hits if the cache had an

additional level of associativity. This simple scheme, however, fails to identify complex conflict patterns, such as those involving more than 2 cache lines. By saving multiple old tags per MCT entry they can detect misses which would have been hits in an arbitrarily associative cache, enabling MCT to identify these more complex patterns. For example, with a direct-mapped cache, if MCT stores 2 old tags per MCT entry, misses which would have been hits in a 3-way associative cache will be identified

2.1.3 Applications of conflict miss filtering

This knowledge can be used in various ways to improve the performance of the cache hierarchy. For example, it could be used to restrict the misses that can write into a victim buffer. This potentially protects two critical resources, the entries themselves, hopefully leaving high probability entries in the buffer longer, and the buffer access ports, increasing its availability.

They demonstrate the utility of this information by applying it to victim cache design, cache pre-fetching, cache exclusion mechanisms, and pseudo-associative caches. In each case, the architecture benefits from applying different policies to different types of misses. It does so in some cases by ignoring accesses unlikely to benefit from the particular architecture. Three of these techniques can be combined into a single architecture, which we call the adaptive miss buffer. The adaptive miss buffer uses the victim/prefetch/ exclusion buffer in a different way depending on the classification of each miss. This uses a single structure to optimize buffer performance for the elimination of both conflict and capacity misses. This greatly increases the effectiveness of a cache-assist buffer, providing twice the performance gain of any single optimization using the same size buffer.

2.2 Backgrounds - Cache Prefetch Methods

Hardware cache prefetching predicts future memory access patterns based on current or past access patterns, and attempts to move data likely to be accessed in the near future closer to the processor.

Hardware prefetchers range from very simple next-line prefetchers to more sophisticated stride or even repeated-pattern based predictors. Those more advanced prefetch methods use tables to record history information related to data accesses. We present several common prefetch methods in this section. And in chapter 4, we choose both a simple sequential prefetcher and a modified *correlation* prefetcher to cooperate with our cache miss type classifier.

2.2.1 Sequential prefetching

The simplest prefetch methods are sequential prefetching. They access cache lines that immediately follow the current cache line. The sequential prefetch is also called next-line prefetch. Early sequential methods always prefetch after each cache miss, while more recent sequential methods wait to issue prefetching until a sequential access pattern is detected. Once sequential prefetching is issued and turns out to be correct, the *degree* of the prefetching is increased until the prefetch can completely hide the latency of a miss to main memory. *Prefetch degree* is the maximum number of cache lines prefetched in response to a single prefetch request. For longer memory latencies, a higher degree is required in order for prefetched data to be returned in time to avoid a cache miss.

2.2.2 Table-Based Prefetching

There are two main kinds of table-based prefetching, Stride Prefetching and Correction Prefetching. Stride Prefetching uses a table (**Figure 2-2**) to store stride-related

local history information. The program counter (PC) of a load instruction indexes the table. Each table entry holds the load's most recent stride (the difference between the two most recently preceding load addresses), last address (to allow computation of the next local stride), and state information describing the stability of the load's recent stride behavior. When a prefetch is triggered, addresses ***a+s, a+2s, …, a+ds*** are prefetched – where ***a*** is the load's current target address, ***s*** is the detected stride and ***d*** is the prefetch degree, an implementation dependent prefetch look-ahead distance; more aggressive prefetch implementations will use a higher value for ***d***. Originally Stride Prefetching used a look-ahead PC (LA-PC) to prefetch ahead.

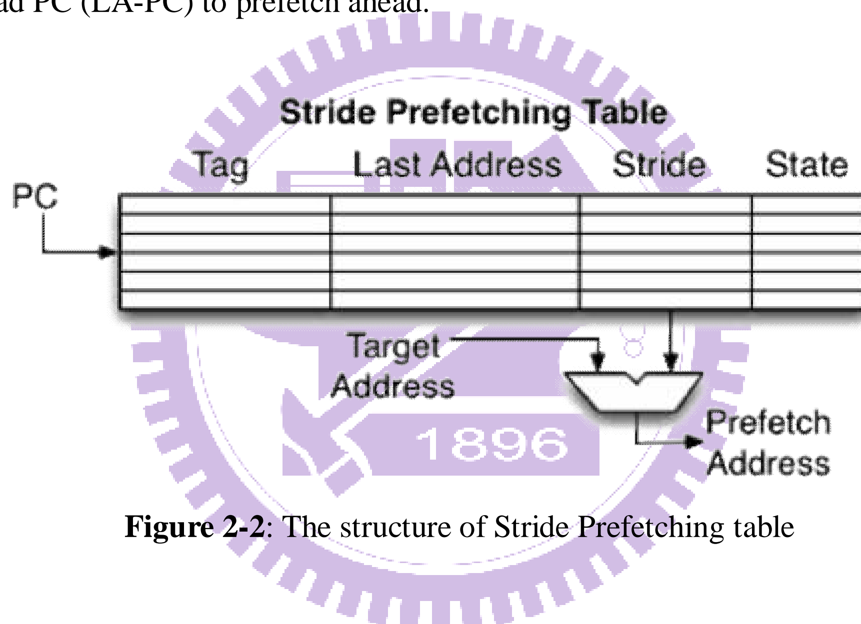


Figure 2-2: The structure of Stride Prefetching table

Markov Prefetching is an example of a correlation prefetching method. Correlation prefetching uses a history table to record consecutive address pairs. When a cache miss occurs, the miss address indexes the correlation table, **Figure 2-3**. Each entry in the Markov correlation table holds a list of addresses that have immediately followed the current miss address in the past. When a table entry is accessed, the members of its address list are prefetched, with the most recent miss address first. The left side of **Figure 2-3** illustrates the state of the correlation table after processing the miss address stream shown at the top of the figure Markov prefetching models the miss address stream as a Markov graph – informally, a probabilistic state machine. Each node in the Markov graph

is an address and the arcs between nodes are labeled with the probabilities that the arc's source node address will be immediately followed by the target node address. Each entry in the correlation table represents a node in an associated Markov graph, and its list of memory addresses represents arcs with the highest probabilities. Hence, the table maintains only a very crude approximation to the actual Markov probabilities. The right side of **Figure 2-3** is the Markov transition graph that corresponds to the example miss address stream.

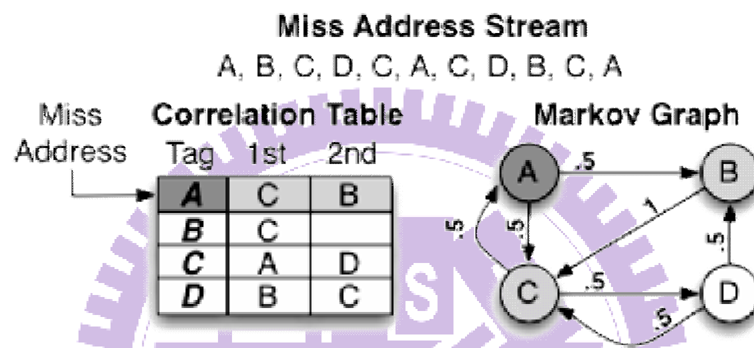


Figure 2-3: Markov Prefetching

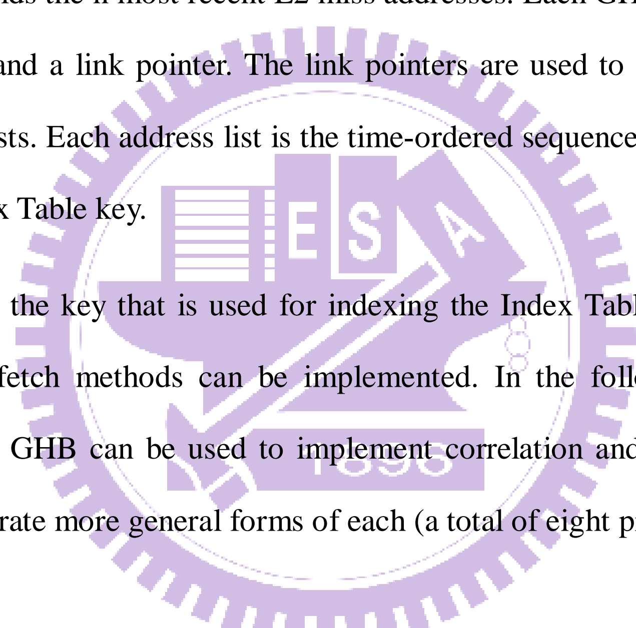
2.2.3 Cache Prefetching using a Global History Buffer

In general, prefetch tables store prefetch history inefficiently. First, table data can become stale, and consequently reduce prefetch accuracy (the percent of prefetches that are actually used by the program before being evicted). Second, tables suffer from conflicts that occur when multiple access keys map to the same table entry. The most common solution for reducing table conflicts is to increase the number of table entries. However, this approach increases the table's memory requirements, and increases the percentage of stale data held in the table. Third, tables have a fixed (and usually a small) amount of history per entry. Adding more prefetch history per entry creates new opportunities for effective prefetching, but the additional prefetch history also increases the table's memory requirements and its percentage of stale data, which together can negate the advantages.

To provide more efficient prefetchers we propose an alternative prefetching structure that decouples table key matching from the storage of prefetch-related history information. The overall prefetching structure has two levels (**Figure 2-4**).

- An Index Table (IT) that is accessed with a key as in conventional prefetch tables. The key may be a load instruction's PC, a cache miss address, or some combination. The entries in the Index Table contain pointers into the Global History Buffer.

- The Global History Buffer (GHB) is an n-entry FIFO table (implemented as a circular buffer) that holds the n most recent L2 miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers are used to chain the GHB entries into address lists. Each address list is the time-ordered sequence of addresses that have the same Index Table key.



Depending on the key that is used for indexing the Index Table, any of a number of history-based prefetch methods can be implemented. In the following subsections we illustrate how the GHB can be used to implement correlation and stride prefetching. In addition, we illustrate more general forms of each (a total of eight prefetching methods).

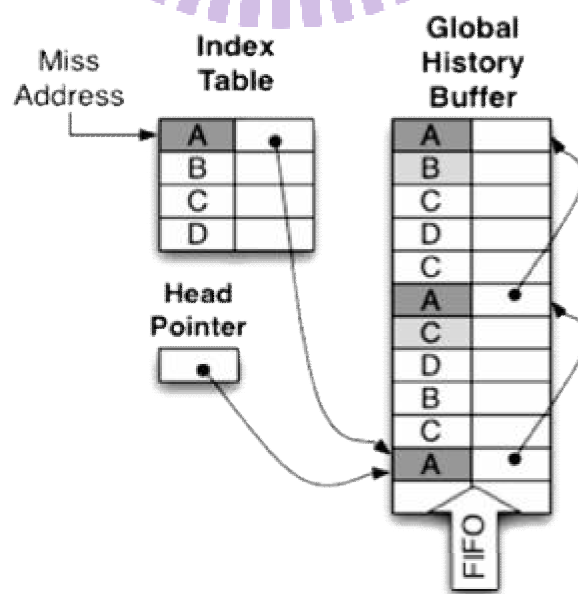
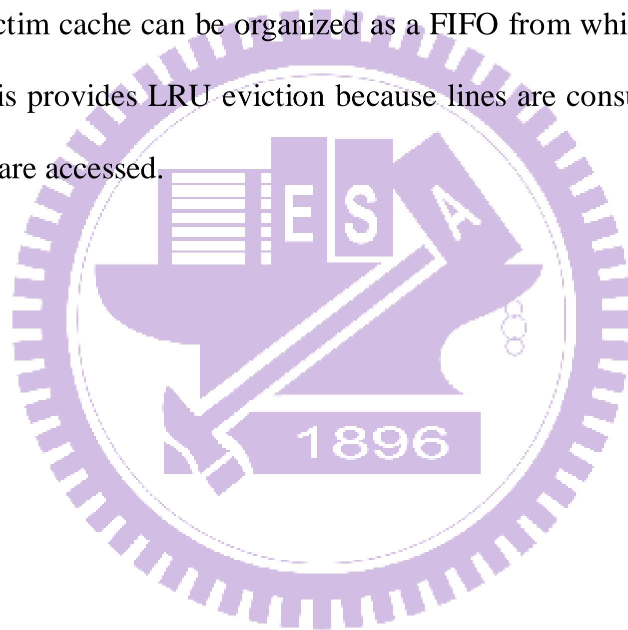


Figure 2-4: GHB Global / Address Correlation

2.3 Backgrounds – Victim cache

The victim cache is a small buffer that holds cache lines recently evicted from the cache. The victim cache is probed when the main cache misses, and when the data is found it can be returned much more quickly than a full cache miss. It targets conflict misses, and is most effective when just a few cache sets are heavily contended for.

Normally, a victim cache hit requires a swap of the two affected lines, the newly evicted line now becoming the first entry in the victim buffer (thus the last to be evicted from the victim cache). The victim cache can be organized as a FIFO from which entries can be taken out of the middle. This provides LRU eviction because lines are consumed out of the victim cache as soon as they are accessed.



Chapter 3 How to Better Identify Cache Miss Types

In this chapter, we present the static time and run time designs of cache miss type identification. Because we also show the evaluations after each design, we have to introduce the methodology of our experiments first.

After introducing of our ideas and designs, we compare the accuracy of our run-time approaches with the related works, The MCT. We also compare the hardware cost of each approach.

3.1 Simulation methodology

The Simulation methodology includes the experimental flow, the benchmark programs and simulators we used in this thesis.

3.1.1 Experimental flow

In this section, we present our simulation methodology. The whole experimental can be divided into follow parts: 1. a benchmarks set. 2. An instruction driven simulator to execute the benchmark programs and produce some trace files. 3 a trace driven simulator to simulate the hardware operation of run time cache miss type identification. Figure 3.X shows the experimental flow of this thesis and following sessions will explain the details of each part.

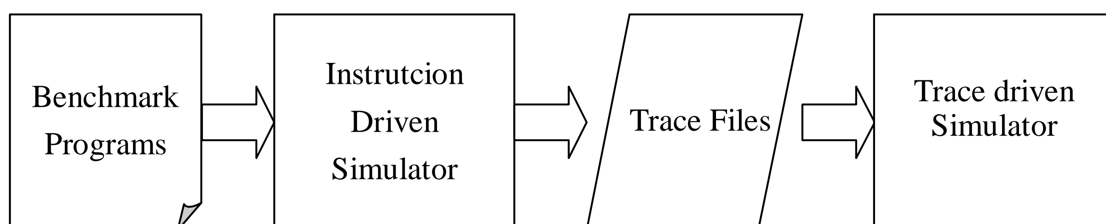


Figure 3-1: Experimental flow

3.1.2 Experimental benchmarks

We chose the SPEC INT 2000 to be our benchmark set. The benchmark programs were complied as Alpha binaries to fit the simulator. But some benchmarks in the sets were not used in the experiment because our simulator lacked for support of necessary system calls. The Table 3.1 shows the details and descriptions of the benchmark we used.

Benchmark name	Description	Number of Instr.
bzip2	bzip2 is based on Julian Seward's bzip2 version 0.1. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.	8.8 billion
crafty	crafty is a high-performance Computer Chess program that is designed around a 64 bit word. It runs on 32 bit machines using the "long long" data type. It is primarily an integer code, with a significant number of logical operations such as and, or, exclusive or and shift.	4.2 billion
gap	gap implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).	1.1 billion
gcc	gcc is based on gcc version 2.7.2.2. The benchmark runs as a compiler with many of its optimization flags enabled.	2.0 billion
gzip	gzip is a popular data compression program written by Jean-Loup Gailly for the GNU project. gzip uses Lempel-Ziv coding (LZ77) as its compression algorithm. SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory.	3.3 billion
parser	The Link Grammer Parser is a syntactic parser of English, based on link grammer, an original theory of English syntax.	4.2 billion
perlbmk	perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language. SPEC's version of Perl has had most of OS-specific features removed.	2.0 billion
twolf	The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. The placement problem is a permutation. Therefore, a simple or brute force exploration of the state space would take an execution time proportional to the factorial of the input size.	0.9 billion
vpr	VPR is a placement and routing program; it automatically implements a technology-mapped circuit in a Field- Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs.	1.5 billion

Table 3-1: The Benchmark programs we used form SPEC INT 2000

3.1.3 The instruction driven simulator

The next part is an instruction driven simulator. We modified the Simplescaler 3.0 simulator to log the information of each cache miss. The **Table 3.2** and **3.3** shows the CPU and memory hierarchies configuration of the simulator.

Simulation Parameter	Value
Instruction fetch queue size	4
Branch predictor type	bimod
Instruction fetch width	4 (instr./cycle)
Instruction decode width	4 (instr./cycle)
Instruction issue width	4 (instr./cycle)
Instruction commit width	4 (instr./cycle)
RUU size	16
LSQ size	8

Table 3-2: The processor configuration of Simplescaler 3.0 in our experiment

Cache name	Cache Size	Number of set	Number of way	Block size	Replacement policy
Instruction L1	16K	512	1	32Bytes	LRU
Data L1	16K	128	4	32Bytes	LRU
Unified L2	2M	2048	4	32Bytes	LRU

Table 3-3: The memory configuration of Simplescaler 3.0 in our experiment

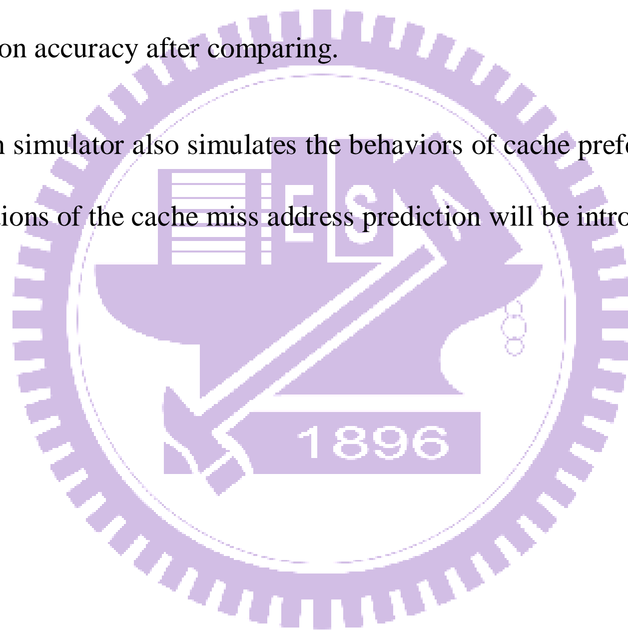
During the simulation, we saved the information of each cache miss including cache name, timestamp, miss address, memory operation (read/write), and the static time cache miss type identification result. In this thesis, we used the cache miss from L1 instruction cache for example. Although our experiment doesn't include other caches like L1 data cache or L2 cache, we believe that our designs can be used on them by change some

parameter. The producing of static time cache miss type identification result will be described in next session. All information was saved into trace files.

3.1.4. Trace driven simulator

The final part of simulation is a trace driven simulator. It handles every cache miss from trace files one after one. The purpose of the second simulator is to simulate the hardware behaviors of run-time cache miss type identification approach which will show be introduced in Chapter 3.3. In the end, we have the run-time cache miss type identification results and we can compare them to the static-time results. Because we assume the static-time results are 100% correct, we can get the run time identification accuracy after comparing.

Our trace driven simulator also simulates the behaviors of cache prefetching and victim cache. Designs and evaluations of the cache miss address prediction will be introduced in Chapter 4.


1896

3.2 Using FLA Replacement in Pseudo Cache to Identify Cache Miss Type in Static Time

As mentioned in Chapter 1.2, conflict misses would not occur if the cache is fully-associative. Therefore, to implement Jouppi's conflict miss identification scheme, we added a pseudo fully- associative cache which has the same size as original cache into the Simplescaler simulator. This pseudo cache only needs the tag part and work together with the original cache.

To avoid the worst case of the LRU replacement policy, we use a finite look-ahead (***FLA***) replacement in this pseudo fully-associative cache. ***FLA*** checks future memory access information and decides which block should be evicted to minimize cache miss rate. Checking the future access information would take a lot of time and space if we simply do the linear search in program traces. Instead, we want to use the ***FLA*** *on the fly*. When the pseudo cache encounters a miss, it won't decide which block will be evicted immediately. We delay the decision until we know which block will be used in the latest future. In fact, all memory accesses are delayed in the pseudo cache. That means we actually maintain a psuudo cache in the status of ***n***-cycle ago (***n*** is the delay cycle).

The algorithm below is the ***FLA*** replacement policy we used in this thesis:

Save the address of memory access of cycle ***i***

If the memory access of cycle (***i*** - ***n***) missed in the pseudo cache then

Evicted block set = { For All blocks weren't used in cycle (***i*** – ***n***) to cycle ***i*** }

If Evicted block set is not empty then

Evicted block = the least recently used block in **Evicted block set**

Else

Evicted block = the least recently used block in the pseudo cache

Figure 3.1 shows the results of static time identification, we experimented both two replacement policies. After we ran the SPEC INT 2000 benchmarks, we compared three kind of misses occurred in the benchmark programs. The cream-color parts mean the misses were not occurred in both ***FA-FLA*** (fully-associative using Finite Look-Ahead replacement policy) and ***FA-LRU*** (fully-associative using Least Recently Used replacement policy) caches but only in real cache (in this experiment is a direct-mapped cache). The purple parts mean those cache accesses hit in ***FA-FLA*** cache but missed in both ***FA-LRU*** and real caches. The blue ones mean those cache accesses were missed in all three cache configurations.

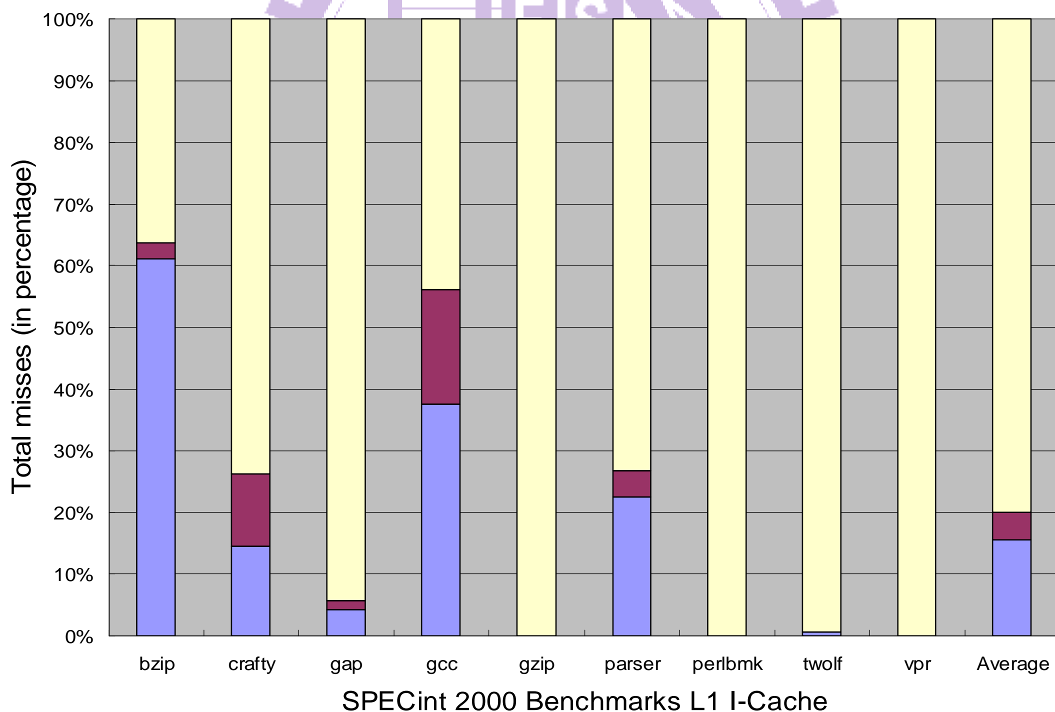
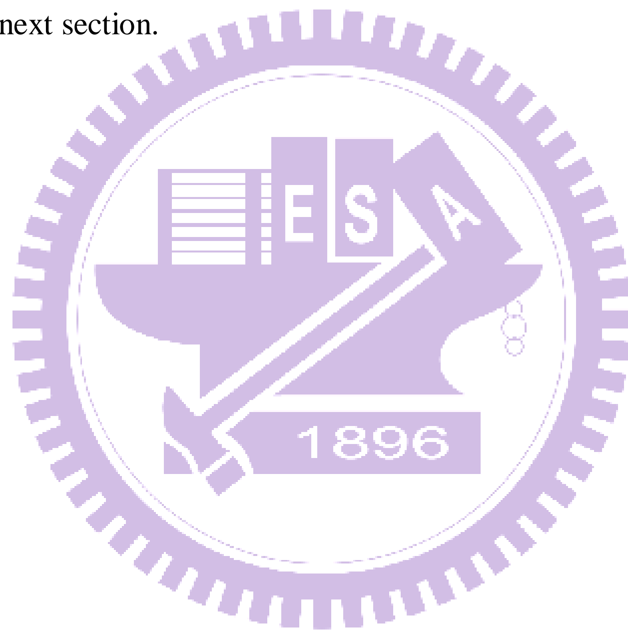


Figure 3-2: Static time identification results

To explain further, the cream-color part of misses will be identified as conflict misses if we use ***LRU*** as the replacement policy of pseudo cache. The cream-color part combines with the purple parts of misses will be identified as conflict misses if we change the replacement

policy to ***FLA***. After changing replacement policy of pseudo cache, almost 5% more of misses were identified as conflict misses. Because finite look-ahead replacement policy doesn't have the problem which we mentioned at **section 1.2**, we can make our identification results more accurate with little overhead.

We had run lots of programs of SPEC INT 2000 benchmarks on the modified Simplescaler simulator. It logged essential information of every miss occurred in instruction cache. The information include time stamp, miss address, operation, and miss type we identified by ***FA-FLA*** pseudo cache. All those information were saved into one trace files which will be used in next section.



3.3 Run time Approach 1 – Miss Frequency Spectrum

Before we descript our first approach, we have to define some symbols and terms using in following sections.

1. We use *M(s, t)* as the symbol for a cache miss occurred in cache set s at time t. The time *t* is not corresponding to the real time, but the index of all misses.

2. ***Set miss frequency***, notated as *MF(s, t, N),* is the number of misses occurred in cache set *s* during time period from *t-N+1* to *t*. Finally, *N* is a constant. The value of *N* will discuss later. For example: *MF (50, 10000, 1000)* = 10 means there are 10 misses occurred in the *cache set 50* during the time period from the 9001th miss to the 10000th miss of the whole cache.

3. ***Set miss distance***, notated as *MD(s, t)*. When a miss occurred in cache set s at time *t*, and *Set miss distance* is the number of misses occurred in the whole cache during time period from the last one miss of cache set *s* to this miss. To speak more precisely, if $M(s, t_0)$ is the last one miss occurred in cache set s and $M(s, t_1)$ is the new one, MD(s, t1) = $t_1 - t_0 - 1$

In last section we had logged cache miss statistics into trace files. Hence, we can do some analysis to find the characteristics of each miss type.

According to the statistics, we had found the cache misses which were classified as conflict misses usually have higher set miss frequency than the misses which were classified as capacity misses. **Figure 3-3** shows two ***frequency spectrums***. The blue one is measured by the cache misses which occurred in benchmark programs and were classified as conflict misses. The purple one is similar but classified as capacity misses.

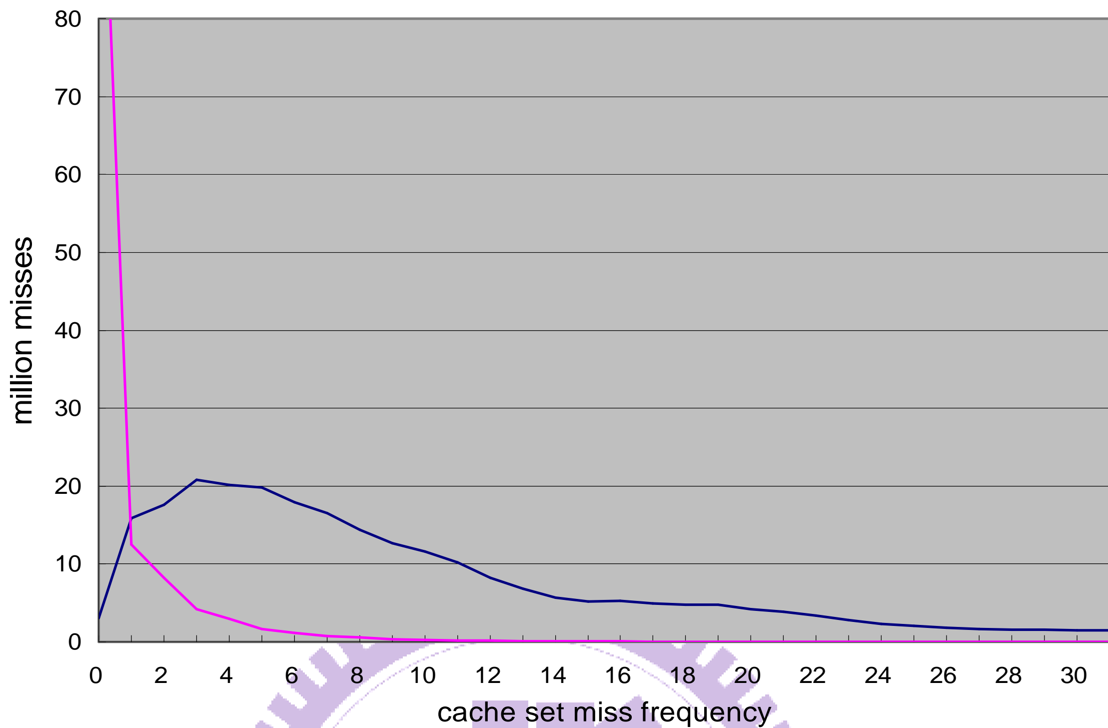


Figure 3-3: Cache miss frequency spectrums

It is obvious that most of conflict misses have higher cache set miss frequency. That is means cache misses occur in a cache sets which have high miss frequency usually are conflict misses, and vice versa. We may use this characteristic to classify cache miss types in run time. But there are still two challenges to overcome. First, how we know the cache set miss frequency in run time. Second, what number of cache set miss frequency is "high" enough to make a cache miss become a conflict miss.

For the first challenge, we add a 3-bits counter to count the set miss frequency for each set. The counter increases when its cache set miss. When the counter reaching the maximum value that it can represent, it will maintain the saturated value. All counters' value cut down to half every N misses by shifting right the counters 1 bit. The number N is the same as last page one and we call this N miss period ***Cool-down period***. By combining above operation, we can get the approximate cache set miss frequency easily and efficiently.

For the second challenge, we have set the value of **base frequency**. When a cache miss occurs, we compare the miss frequency of the cache set (which miss occurs) and the base frequency. If the set's frequency is higher than the base frequency, we identify this cache miss a conflict miss. The value of base frequency will affect miss type classification accuracy. If we set base frequency higher, more cache miss will be classified as capacity miss, and vice versa. To make this approach accurate, the proper value of base frequency will be determined by experiment and shows in following session.

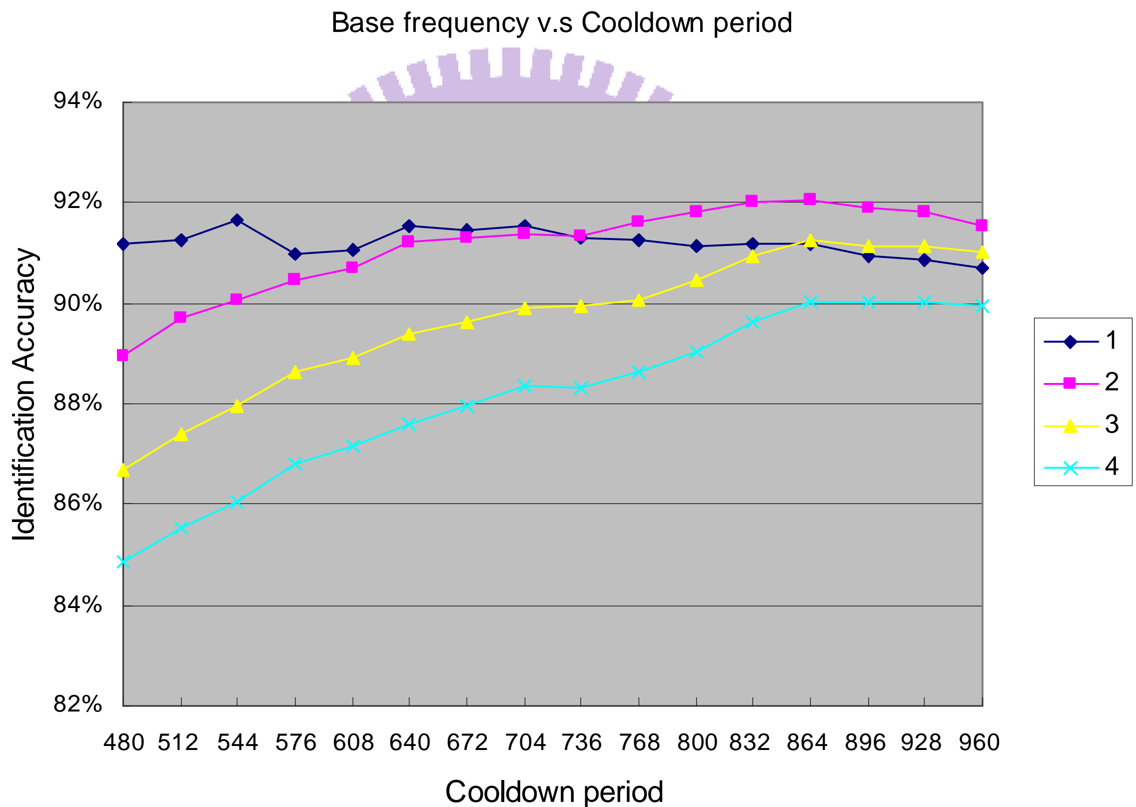
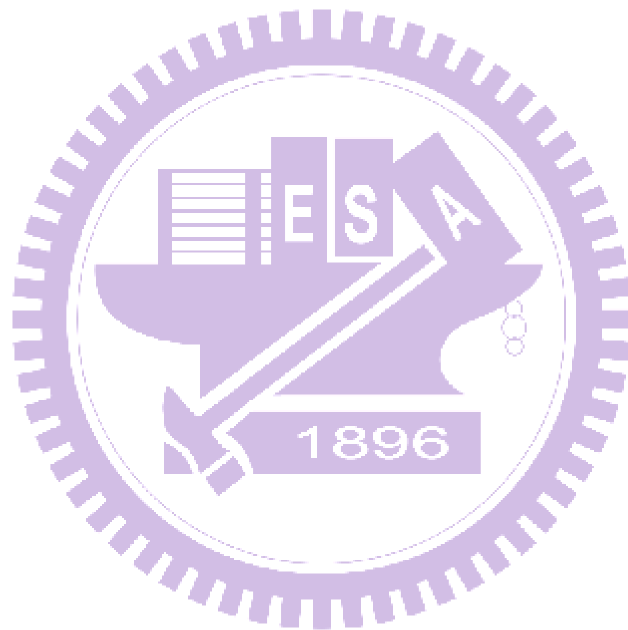


Figure 3-4: Identification accuracy in different base frequencies and cooldown periods

As mentioned, we set the value of base frequency and cooldown period by experiment. In another words, we tried a lots of values and picked the pair which have the highest identification accuracy. The definition of identification accuracy is simple, the percentage of cache miss which have the same result as the static time identification scheme. We

show the experiment results below: different color lines represent different base frequency; and the X-axis represent the cooldown period. As the **Figure 3.6** shows, if we set the base frequency to 2 and set the cooldown period to 864 would get the highest accuracy, which is about 92.05%.

Because the idea of this approach was formed from Figure 3-3 which is a graph of frequency spectrums, we called this approach Miss Frequency Spectrum.



3.4 Run time Approach 2 – Miss Distance

In addition to cache miss frequency, we want to use more characteristics of cache miss to classify different types of them. In this approach, we though miss distance would be useful. If every cache miss is non-conflict miss, a cache set will not have a miss again until all others cache set had encountered one cache miss. Base on this theory, every cache set has the same set miss distance MD(s, t) = the number of cache sets if there is no conflict miss occurred. If one conflict miss occurred, the corresponding cache set will have shorter set miss distance than other sets. Like approach 1, we have to find a way to get the set miss distance in run time and set a threshold distance.

To get the set miss distance in run time, we need a FIFO queue called **cache miss history table** to store the information of each cache miss. And set the size of **time windows**. The size of time windows is equal to the threshold distance and the number of entries of this table. When a cache miss occurs, we search the history table to find the lines which have the same set index. We don't search the entire history table but only within the time window. If hit, we assume this cache miss is a *conflict miss* because that means there was a cache miss occurred in the same cache set within the time window. In other words, this cache miss has shorter set miss distance.

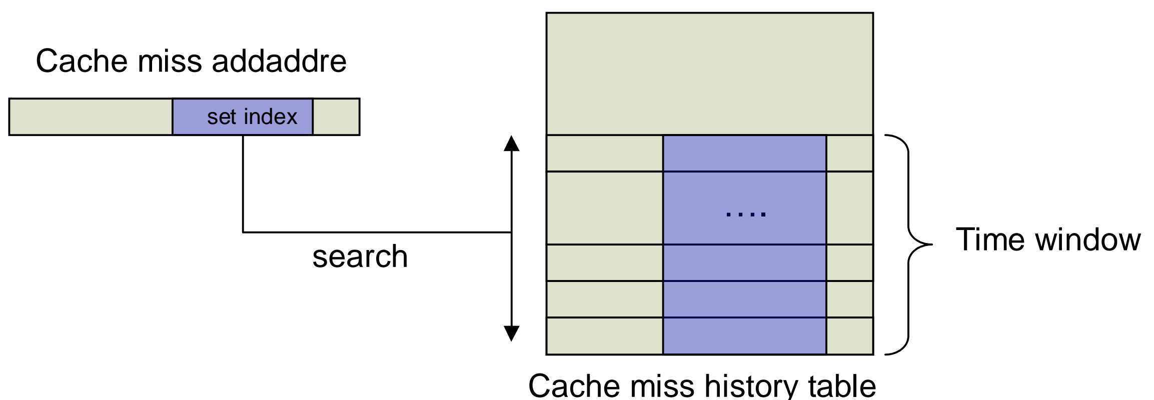


Figure 3-5: Structure of Miss Distance approach

But there is a drawback of above scheme. The operation of searching the history table must be in time. Implementing the cache miss history table by using Content Addressable Memory (CAM) is a possible solution. But the cost of CAM will be too high if we have a large history table.

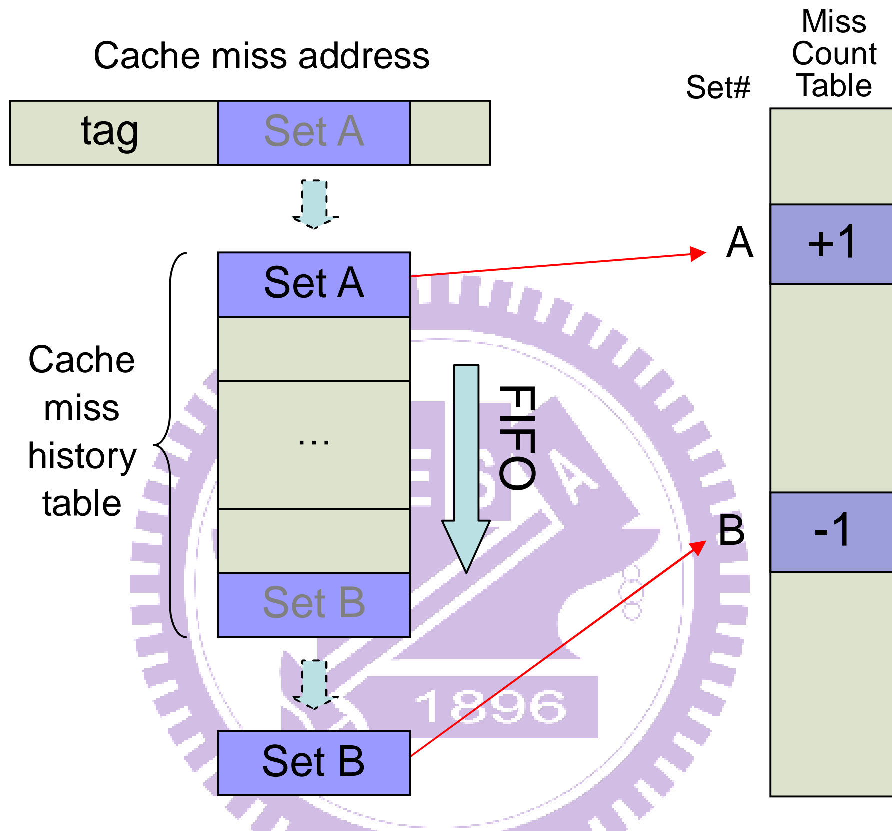


Figure 3-6: Structure of improved Miss Distance approach

So we have to improve the scheme to lower the hardware cost. Instead of parallel searching the history table, we can maintain a **Miss Count Table** to count the number of miss times in the time window of each cache set. The Figure 3.6 shows the idea. Each cache set has a **Miss Counter**. For example, a cache miss occurs in cache set “**A”.** Then we push the set index into the cache miss history table and increase corresponding miss counter by one. At the same time, another set index “B” has been pushed out from the history table and corresponding miss counter decreased by one. We can know how many number of certain set in the FIFO queue by check the value of corresponding miss counter

instead of parallel searching the whole cache miss history table. The Overhead of Maintaining the Miss Count Table is quite low, only two counter have to be updated per cache miss. This improvement effectively lower the hardware cost of this approach.

To make this approach accurate, we have to set two parameters proper. The first one is the size of cache miss history table, i.e. the time window size. Bigger size means more old cache misses can be recorded, and longer set miss distance can be measured. Therefore, bigger size also means more cache misses would be identified as conflict miss by this approach.

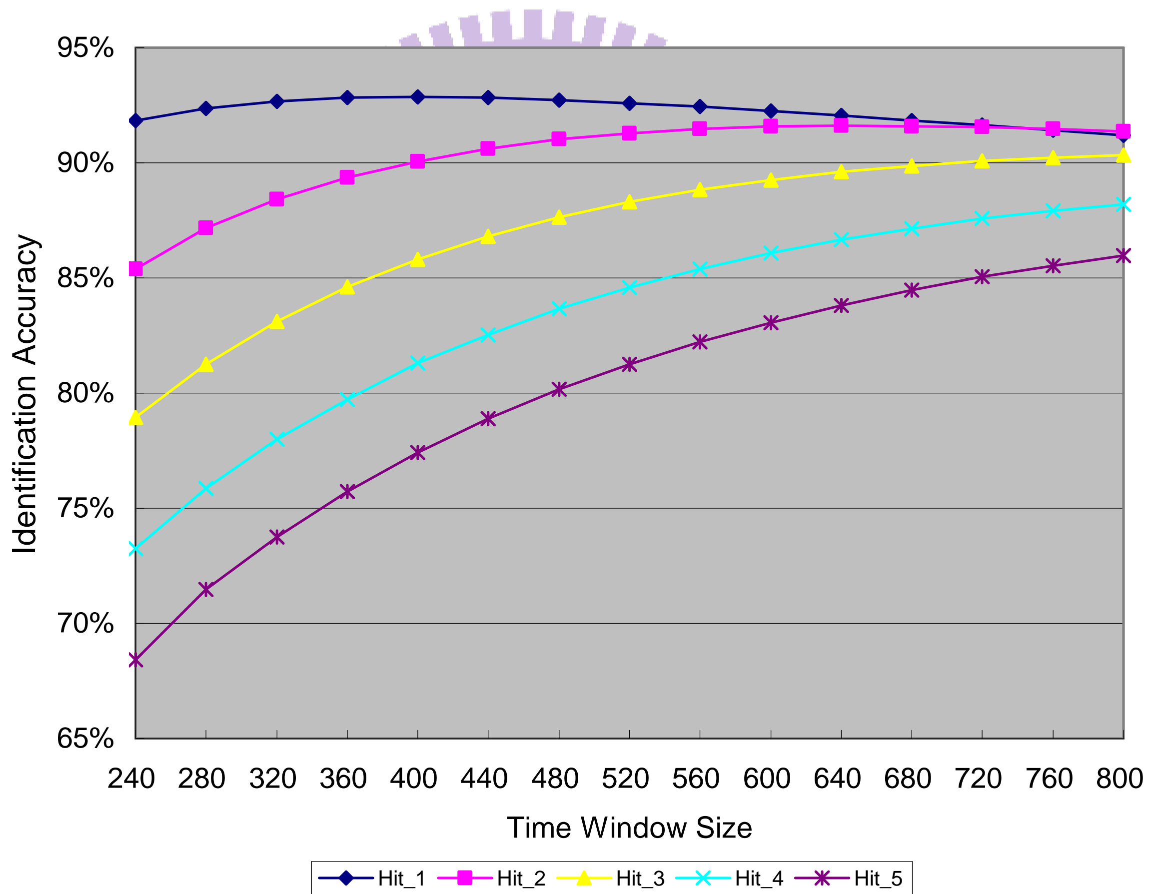
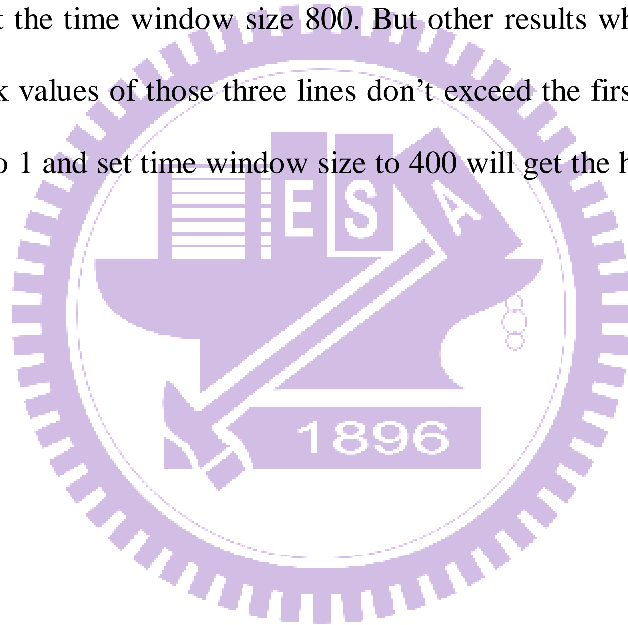


Figure 3-7: Identification accuracy in different time window sizes and hit numbers

The second parameter is due to the improvement of this approach. Because we use miss counters instead of parallel searching, the output of this approach is the number of

miss times in the time window instead of hit or not hit. Therefore, we also have to determine the threshold value to identify the coming cache miss is a conflict miss or not.

Just like the approach 1, we tested many different combination value of time window size and miss counter threshold. **Figure 3.7** shows part of our results. Different colors of lines represent different value of miss counter threshold. For example, the yellow line, "Hit_3", means setting the threshold to 3 and a conflict miss must have more than three entries of the coming cache miss in the cache miss history table. The X-axis represents the value of time window size. You can see the trends of "Hit_3", "Hit_4", and "Hit_5" are not going down at the time window size 800. But other results which not presented here show that the peak values of those three lines don't exceed the first one. Therefore, if we set the threshold to 1 and set time window size to 400 will get the highest accuracy, which is about 92.86%.



3.5 Evaluation of Run Time Cache Miss Type Identification

In this session, we compare our run time cache miss identification approaches to related works, the Miss Classification Table. We used the trace driven simulator to simulate our two run-time cache miss type identification approaches and the MCT approach. Figure 3.8 shows the results. The MCT1 to MCT4 represent the Miss Classification Table uses one to four old tags, respectively. You can see that not the more MCT entries the better, MCT3 has the best average accuracy among them. But even MCT3 isn't as good as our approaches. Our Miss Distance approach has the best average accuracy among all the run time cache miss type identification.

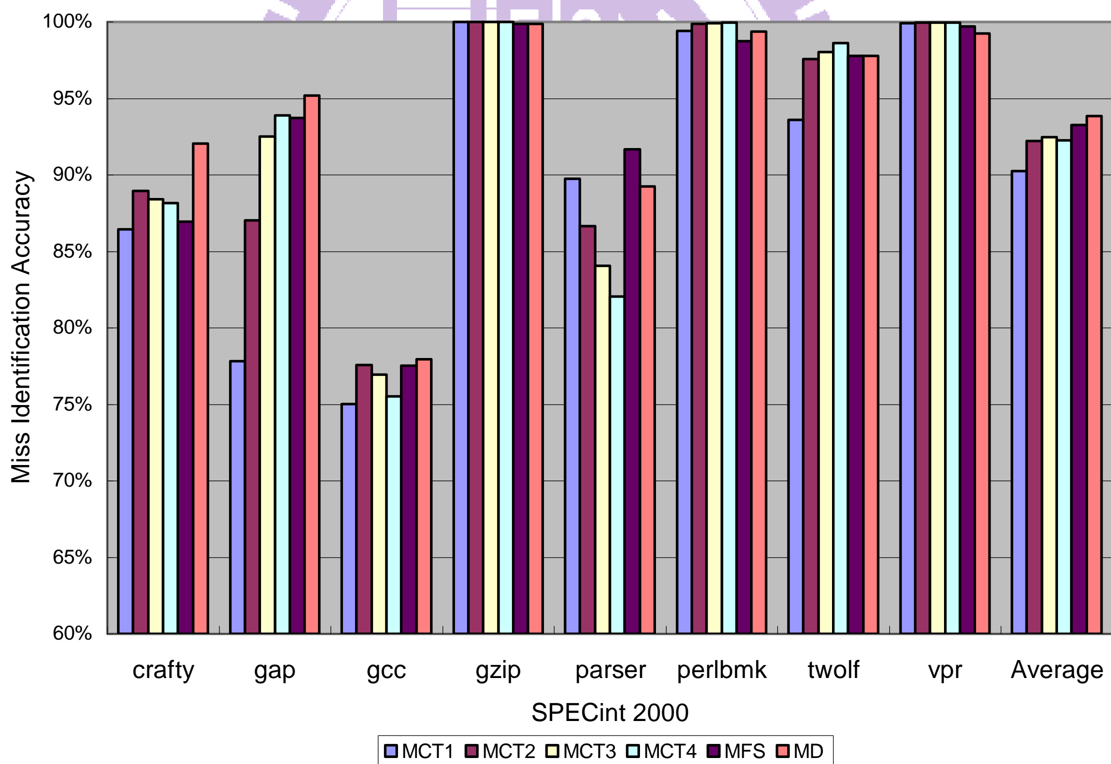


Figure 3-8: Identification accuracy using different run time approaches

If you back to the previous session and check **Figure 3.1**, you can find that cache misses form benchmark programs like gzip, perlbmk, twolf, and vpr are almost conflict

misses. All approaches have no problem to identify them. Cache misses form gcc are not very easy to identify them in run time, all approaches have lower accuracy in that benchmark because the program complexity of gcc is higher than others.

Approach	Size per cache set	Total table size for following cases	
		A 16KB, 32 bytes line size, Direct- Mapped cache	A 16KB, 32 bytes line size, 4-way set-associative cache
MCT	1 ~ 4 tags per cache set, 18~20 bits per tag	9 ~ 36 Kbits	2.5 ~ 10 Kbits
MFS	3 bits per cache set	1.5 Kbits	0.3 Kbits
MD	Cache miss history table: time window size * width of set index Miss Counter: 3 bits per cache set	5 Kbits	1.74 Kbits

Table 3-4: table size comparison for related works and our approaches

The accuracy differences between our approaches and MCT are not large. But if we consider the hardware cost, especially the table size needed, our approaches are much lower. Table 4.x show the table size needed in different cache configuration. IF MCT simply store the full-length tags, it needs 18~20 bits per tag. They also proposed that saving only the lower bits of the evicted tag can reduce the table size, but also reduce the identification accuracy at the same time. Our Miss Frequency Spectrum approach only needs a 3-bits counter per cache set. Besides the 3-bits counter per cache set, our Miss Distance approach needs another table - cache miss history table. And the size of cache miss history table depends on the cache configuration. But the total table size needed by Miss Distance approach is still smaller than the MCT1.

Chapter 4 Dynamic Cache Miss Address Prediction and PV Buffer

In this thesis, we treat cache-based architectural optimizations such as cache prefetching, and victim cache as cache miss address predictors. When cache encounters a miss, those optimizations predict memory addresses which would be used in near future. Those cache optimizations are aimed at particular types of cache misses. Some Optimizations can't handle all types of misses well. For example, victim cache serves conflict misses almost exclusively; next-line prefetching works for compulsory and capacity misses. Correlation prefetching handles capacity and conflict misses mixed pattern well. Therefore, if we can combine several cache optimizations to predict miss addresses, the prediction accuracy will higher than just using one kind of cache optimizations. All cache optimizations will fetch predicted blocks into a single small buffer, called PV buffer (**P**refetching and **V**ictim buffer).

But, using more cache optimizations means fetching more cache blocks per cache miss. That means more bandwidth and energy consumption. By using the cache miss type identification results as a filter or a parameter to cache optimizations, we can reduce the unnecessary traffic of the memory hierarchy and make buffer entries more efficiency.

This Chapter first demonstrates how the run time miss type identification cooperates with those cache optimizations by using the cache assist buffer. Then we present the usage of miss type on each miss address prediction. We evaluate the benefits of PV buffer at the end of this chapter.

4.1 Cooperating the Cache miss type identification and miss address prediction

In next sections, we will model several kinds of cache assist buffer, which will serve at different times as a prefetch buffer, victim buffer, or PV buffer. In each case, the structure is very similar. In most cases it will have four to sixteen fully-associative entries. Each entry has the same size as the cache block size. The purpose of the cache assist buffer is to fill the latency gap between two levels of cache hierarchy.

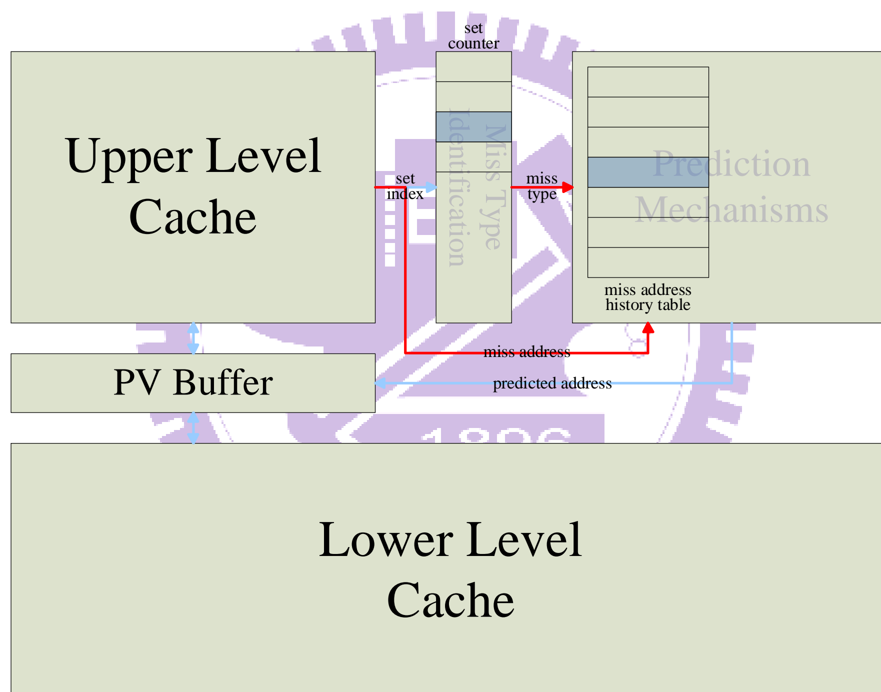
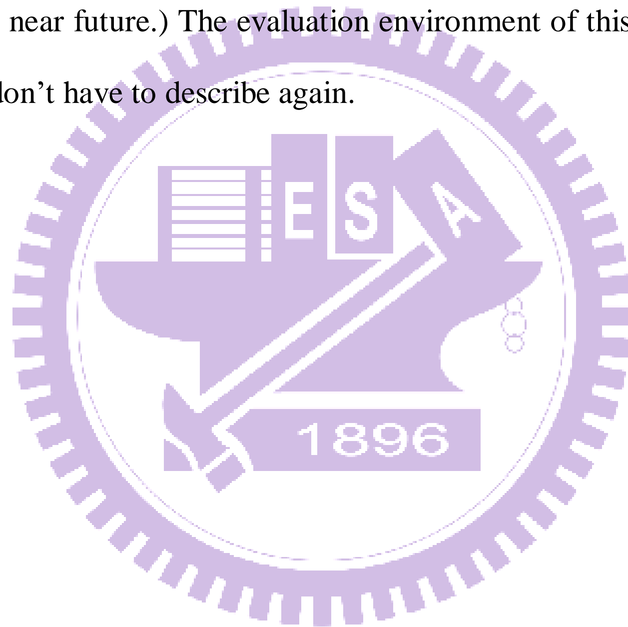


Figure 4-1: Block Diagram of our dynamic cache miss address prediction

We demonstrate how the run time miss type identification cooperates with miss address prediction by using the PV buffer (**Figure 4-1**). When a miss occurs in the upper level cache, the miss address would be sent to both miss type identification unit and miss address prediction unit. In last chapter, we had introduced our two run time cache miss type identification approaches. Both approaches use counters to determine the miss type. After the miss type determined, it will be sent to the miss address prediction unit. The

prediction unit activates appropriate optimization by the miss type, and sends the predicted addresses to the PV buffer. Finally, the predicted blocks will be fetched from lower level cache into the PV buffer. All actions should be done before the next upper level cache miss.

Before we introduce the applications of cache miss type, we have to define the *prediction accuracy* first. That is the probability of requested cache block can be found in the cache assist buffer on a cache miss. (Although victim cache is not actually predicting anything, we can consider that victim cache predicts the evicted block which will be requested again in near future.) The evaluation environment of this chapter is the same as Chapter 3, so we don't have to describe again.



4.2 Filtering Victim Cache with cache miss type

As we mentioned, victim cache serves conflict misses almost exclusively, and is most effective when just a few cache sets are heavily contended for. We can use the cache miss type identification result as a filter. That means we put the evicted block into the victim buffer only if a miss was identified as a conflict miss. There are two possible benefits of this filtering: first, we can remove unnecessary fetching operations when a non-conflict miss occurs; second, the entries of victim buffer can be saved for conflict miss only, that may increase the hit rate.

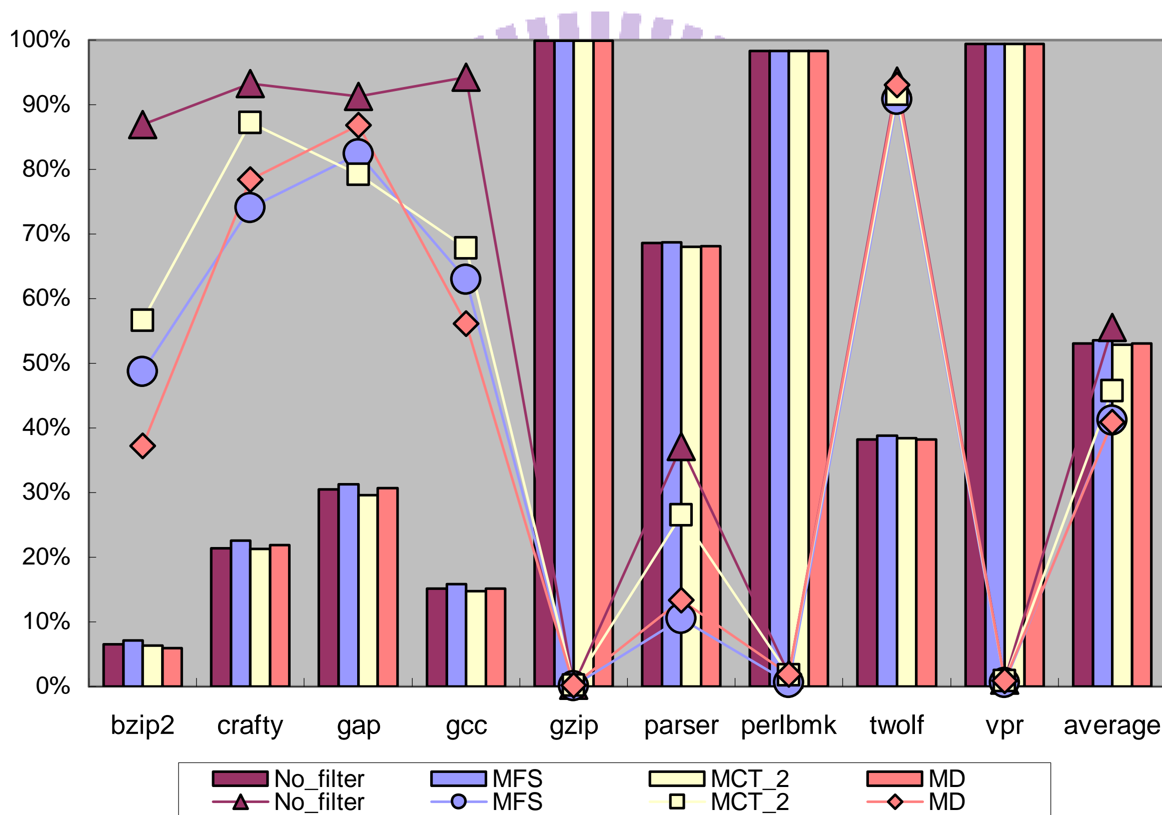
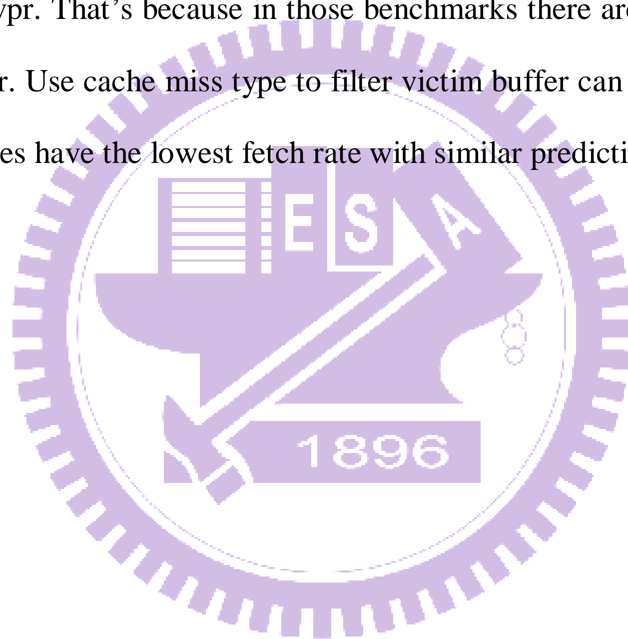


Figure 4-2: Filtering Victim Cache with cache miss type.

In this experiment, we use sixteen entries of victim buffer on L1 Instruction cache. Normally, a victim buffer hit requires a swap of the two affected blocks. To simplify, we decide not to swap the two blocks and use the FIFO as replacement policy. **Figure 4-2** shows the experiment results. The Different color bars represent the prediction accuracy of victim

buffer (from left to right, respectively) using no filter, using MFS results as the filter, using MCT2 results as the filter, and using MD results as the filter. (The definition of prediction accuracy had shown in last section.) In fact, the difference between no filtering and filtering by the results of all cache miss type identification is quite small. Although there is no performance gain, there is no performance loss either. The Different color lines represent the fetch rate of victim buffer using different filtering policy. No_filter policy always put the evicted block into the victim buffer unless the block is already there. This situation is possible when we use No_Swap and FIFO replacement policy and cause the low fetch rate in gzip, parser, perlbmk, and vpr. That's because in those benchmarks there are only a few cache sets heavily contending for. Use cache miss type to filter victim buffer can lower the fetch rate. In average, our approaches have the lowest fetch rate with similar prediction accuracy.



4.3 Cache Prefetching with cache miss type

4.3.1 Sequential Prefetching

The sequential prefetch simply pre-fetch the next cache line on a cache miss. It always prefetch one block per cache miss. Therefore, the prefetch buffer only needs one entry if we using sequential prefetch. While all misses can benefit from next line prefetching, we expect non-conflict misses to be more amenable to predict via pattern analysis than conflict misses. On that assumption, we can activate the sequential prefetching only when encountering a non-conflict miss. But after experiment, we found that sequential prefetching predicts well both on conflict and non-conflict misses for a direct-mapped L1 Instruction cache (**Figure 4-3**). That is because the nature of program execution, which is sequential. Even when two code blocks conflict in a direct-mapped cache, the cache miss address patterns were still sequential. Therefore, we suggest that always do sequential prefetching whatever the type of cache miss is on a direct-mapped L1 Instruction cache.

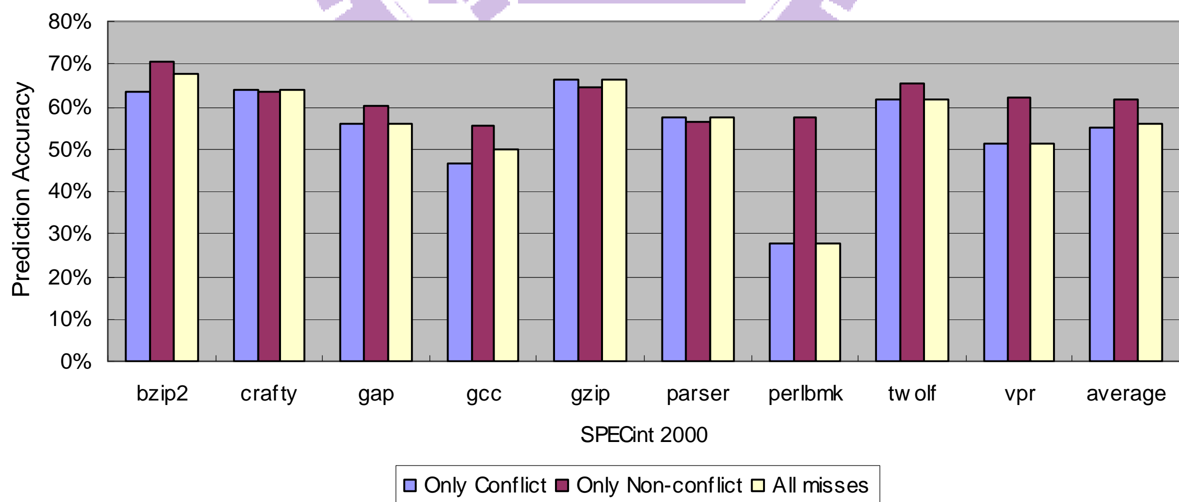


Figure 4-3: Sequential Prefetching accuracy on a direct-mapped L1 instruction cache

But on a 4-way set-associative L1 data cache (**Figure 4-4**), we did found the prediction accuracy of non-conflict misses is much higher than conflict ones. In that

situation, we can activate the sequential prefetching only when encountering a non-conflict miss.

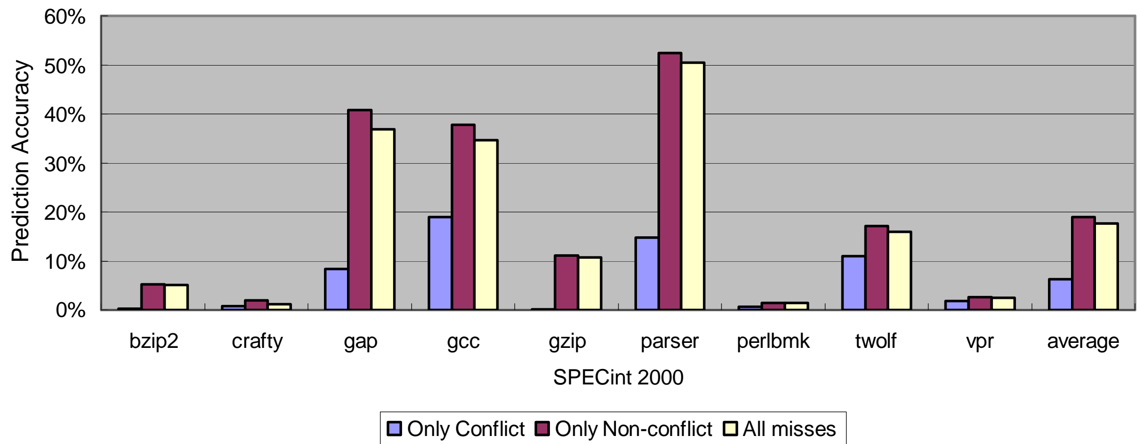


Figure 4-4: Sequential Prefetching accuracy on a 4-way set-associative L1 data cache

4.3.2 Correlation Prefetching

We use a simplified correction prefetching which has a single Miss Address History Table (MAHT) to record consecutive miss addresses rather than build a Markov graph and multiple tables. When a cache miss occurs, correction prefetcher index MAHT by miss address and put the following blocks into prefetching buffer. For example (**Figure 4-5**), there were three consecutive cache misses which were recorded into the MAHT. The addresses are 0100, 0120, and 0140, respectively. When cache block '0100' misses again, we use the miss address index the table, and fetch cache blocks 0120, 0140, … into the prefetching buffer. The number of block is depended on prefetch degree $\boldsymbol{d}$. If $\boldsymbol{d}$ =2, the prefetcher will prefetch two blocks on a single miss.

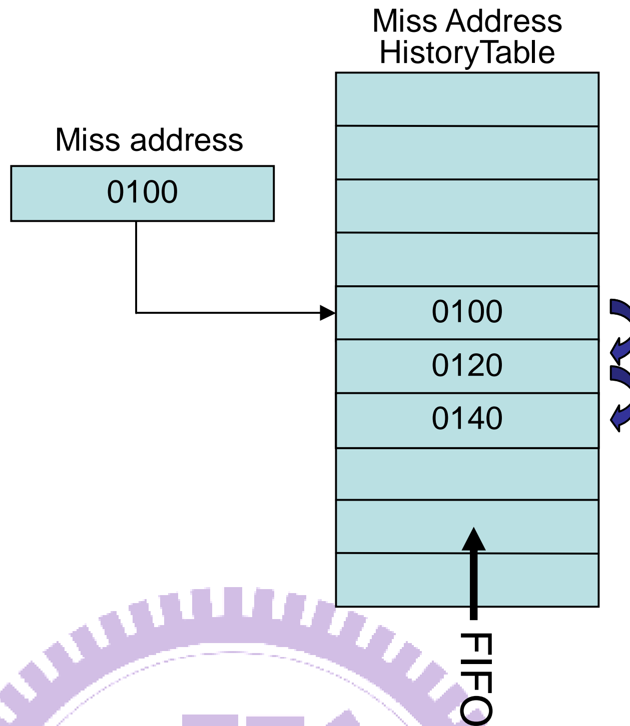
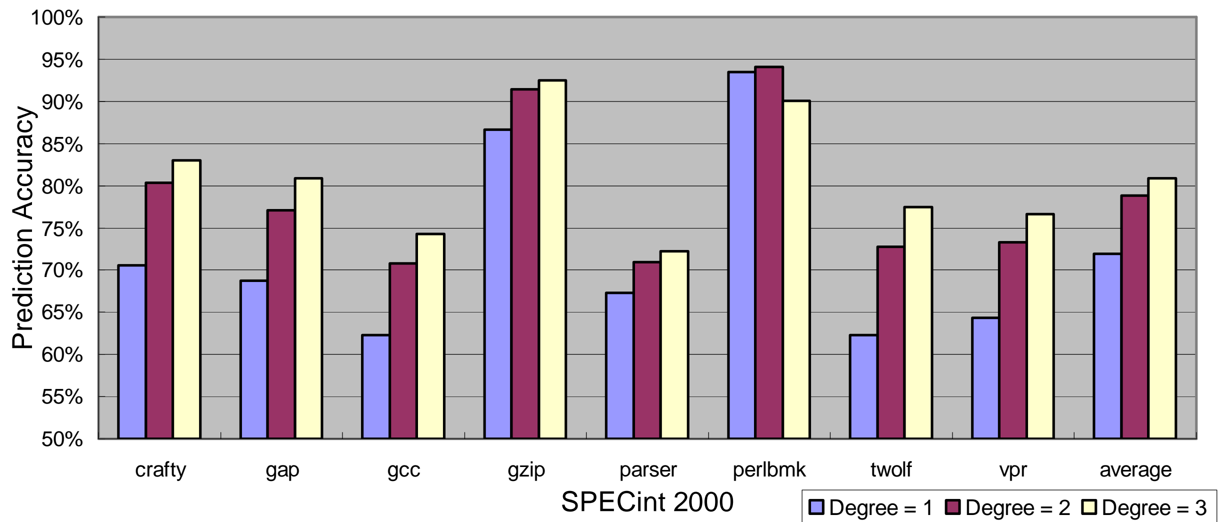


Figure 4-5: Simplified Correlation Prefetching

If memory bandwidth and prefetching buffer are large enough, higher prefetching degree generally has higher prediction accuracy. That is because the cache miss stream is filled by mixed miss types. Our experiment result shows that point of view (**Figure 4-6a**). But when the size of prefetching buffer is smaller, setting prefetching degree too high would not be always good. That is because the buffer is too small to let the blocks stay long enough to be used. **Figure 4-6b** shows average prediction accuracy decreased when we increased prefetching degree from two to three. Therefore, we introduce the **Selective 3** prefetching policy: we add an additional field which records the miss type at each MAHT entry; when predicting, the prefetcher checks miss types of incoming and following misses; if they are the same type, we set prefetching degree to 2; otherwise, we set prefetching degree to 3. As results show at **Figure 4-6b,** our **Selective 3** policy avoid the decreasing accuracy in benchmarks like gap, gcc, perlbmk, and have the highest overall prediction accuracy.



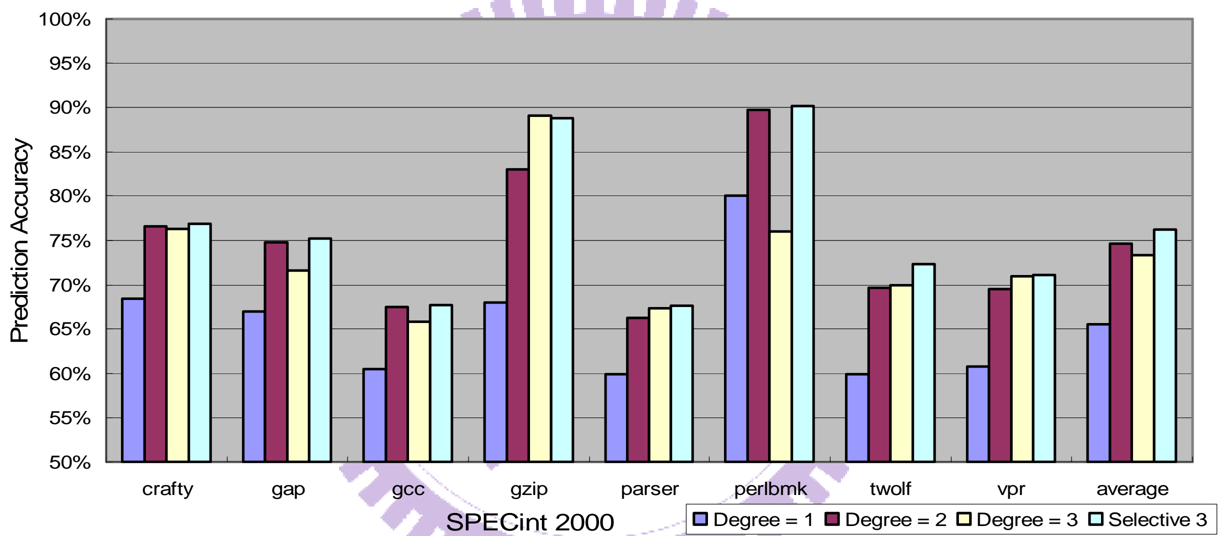


Figure 4-6: a. Correlation Prefetching using 16 entries on a 16KB direct-mapped L1 instruction cache. **b**. Correlation Prefetching using 4 entries on the same cache.

4.4 Put it all together – PV Buffer

In this session, we combine three cache optimizations above mentioned. All cache optimizations will fetch predicted blocks into a single small buffer, called PV buffer. We use two kinds of allocation policies to allocate buffer entries: Separated and Mixed policy.

- Separated policy: each cache optimizations fixed number of buffer entries. For example: sequential prefetching has one buffer entry. Correction prefetching has four entries. Victim cache has eleven entries. Each optimization can only replace its own entries. And it's possible that different optimizations fetch the same block into their own entries.

- Mixed Policy: All three cache optimizations share 16 buffer entries. When a cache optimization wanted to fetch a block into PV buffer and found the block was already there (possibly fetched by other cache optimizations), it won't fetch again.

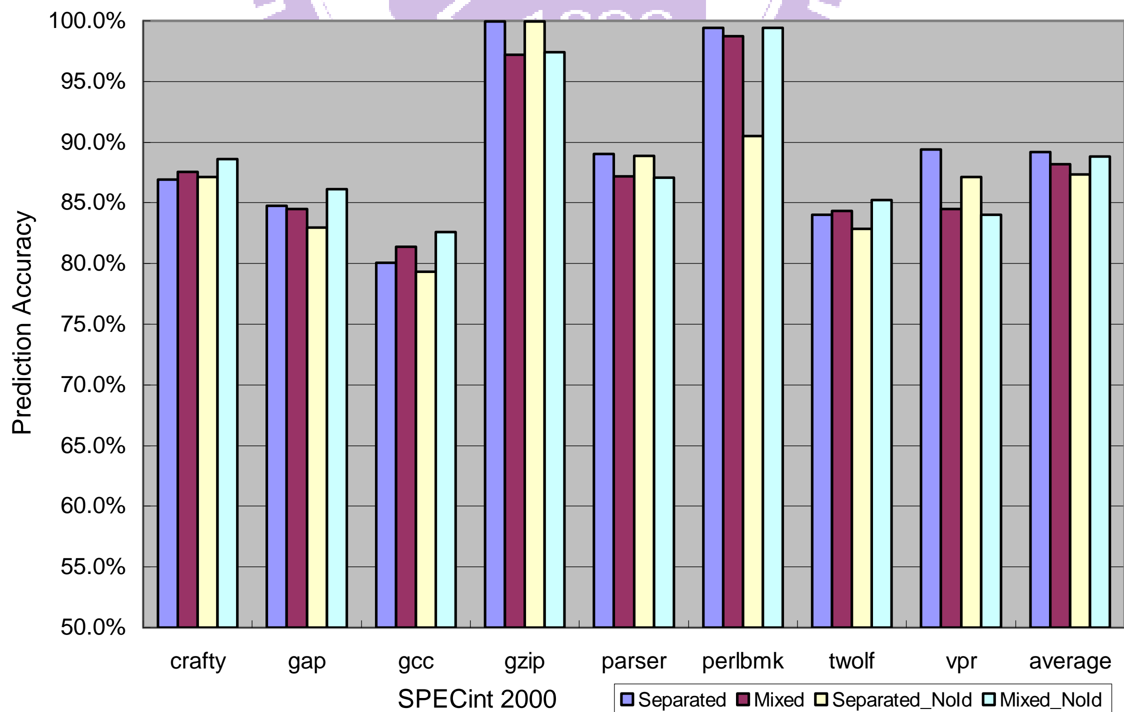


Figure 4-7: Prediction Accuracy of PV buffer

Figure 4-7 shows the prediction accuracy of PV buffer. The left two bars are results of using cache miss type as filtering on victim cache and **Selective 3** policy on correlation prefetching. The right two bars are results without using any filtering and prefetching degree 3 on correlation prefetching. Our separated policy has average 89.2% prediction accuracy with sixteen buffer entries. No single cache optimizations can reach that accuracy with the same size of buffer.

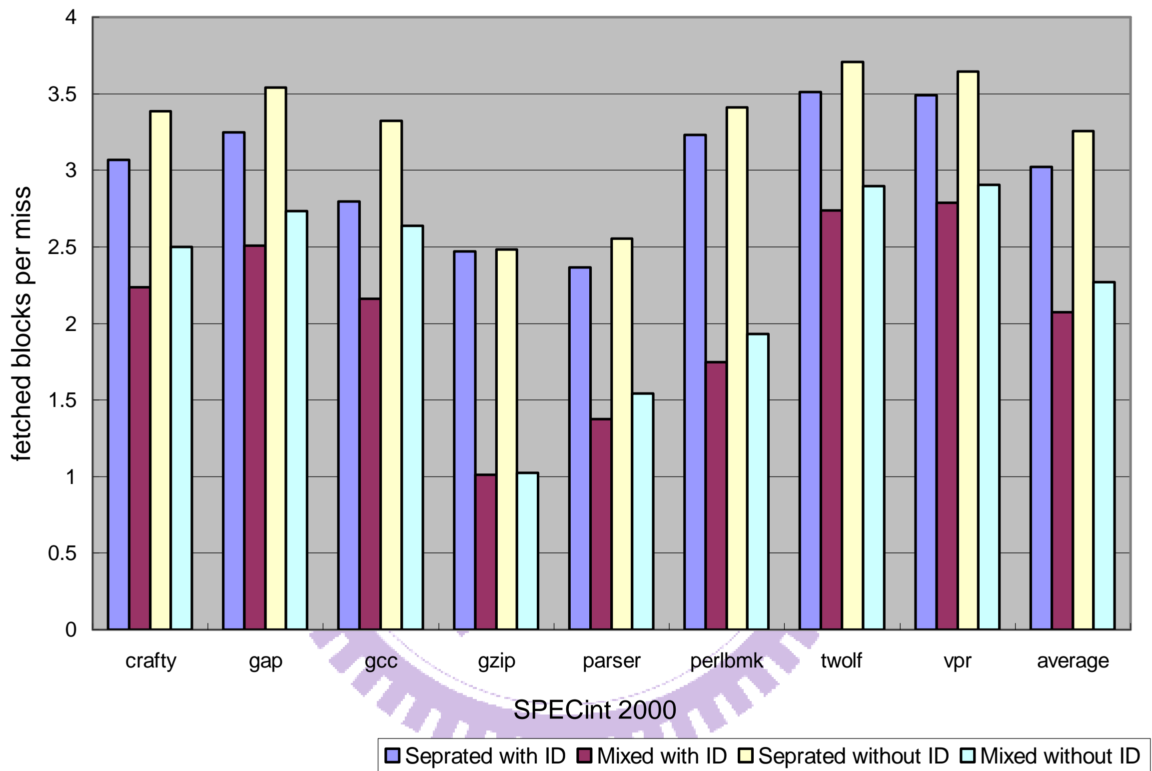


Figure 4-8: Fetching rate of PV buffer

Although separated policy has higher prediction accuracy, mixed policy has similar prediction accuracy with much lower fetching rate. **Figure 4-8** shows the numbers of fetched blocks per cache miss. Using cache miss type to filtering fetching operations, both policies can reduce about 0.2 blocks per miss. The reason that mixed policy having lower fetching rate is because all cache optimizations use the same buffer entries and they won't fetch a block if it was fetched by other cache optimizations.

Chapter 5 Conclusion and Future Works

In this thesis, we explored many issues of cache miss type identification, including modified Jouppi's conflict miss identification scheme, which is suitable for static-time miss type identification, by changing the replacement policy of pseudo cache to the on-the-fly finite look-ahead replacement policy. This modification avoids the worst case of LRU and makes the identification results more accurate.

For run-time miss type identification, we proposed two low hardware costs, low complexity cache miss type identification approaches which categorize cache miss types according to its *frequency* and *distance*. Both approaches achieve more than 93% average identification accuracy.

We also demonstrated the application of this information by applying it to victim cache design, sequential prefetching, and correlation prefetching. In each case, the architecture benefits from applying different policies to different types of misses. In addition, we combined several cache optimizations to cover 89% of cache misses with a sixteen-entry buffer, called PV buffer. No single cache optimization can do that with the same number of buffer entries. This design uses a single structure to optimize buffer performance for the elimination of both conflict and capacity misses. Although multiple cache optimizations need fetching multiple cache blocks to PV buffer, by using cache miss type information, we can reduce unnecessary memory traffic and fetch operations to increase effectiveness of this cache-assist buffer.

This work can be extended in multiple ways:

- *Use miss type information to determine cache replacement policy*: different types of cache misses occur at different frequencies. We log cache miss types to each cache block, and we can explore a new cache replacement policy which selects replacement scheme base on the miss types.

  For Example: Conflict misses generally have shorter **Miss Distance** than other types of cache misses. A cache block which was marked as conflict miss has less chance to be used again if its cache set has a long miss distance at that moment. Therefore, we can increase the replacement priority of this cache block.

- *Reduce miss address history table size by not saving consecutive miss addresses*: both Miss Distance approach and simplified correlation prefetching use a large miss address history table. But lots of miss addresses are sequential. We can reduce miss address history table size by not saving consecutive miss addresses, and just saving first and the last one of a consecutive address stream for instead.

- *Explore the way to identify coherence misses*: coherence misses are misses that occur as a result of invalidation to preserve multiprocessor cache coherence. The most common way to preserve multiprocessor cache coherence is by adding some states at each cache block and changing states by **cache coherence protocols** (**Figure 5-1**). We can probably identify coherence misses by cache block state checking. But accurately predicting when and where another processor modifies a data line is a very difficult problem. It requires a complete understanding of the communication and synchronization patterns of an application.

  But still, we can improve cache performance using cache miss type information. If coherence miss is detected, we can only invalid the sub-block which was written by

another processor. The sub-block mechanism could decrease the false sharing miss ratio and miss stall time. Another possible approach is cooperating with cache replacement policy we mentioned above. A cache block marked as coherence miss has different priority to be replaced.

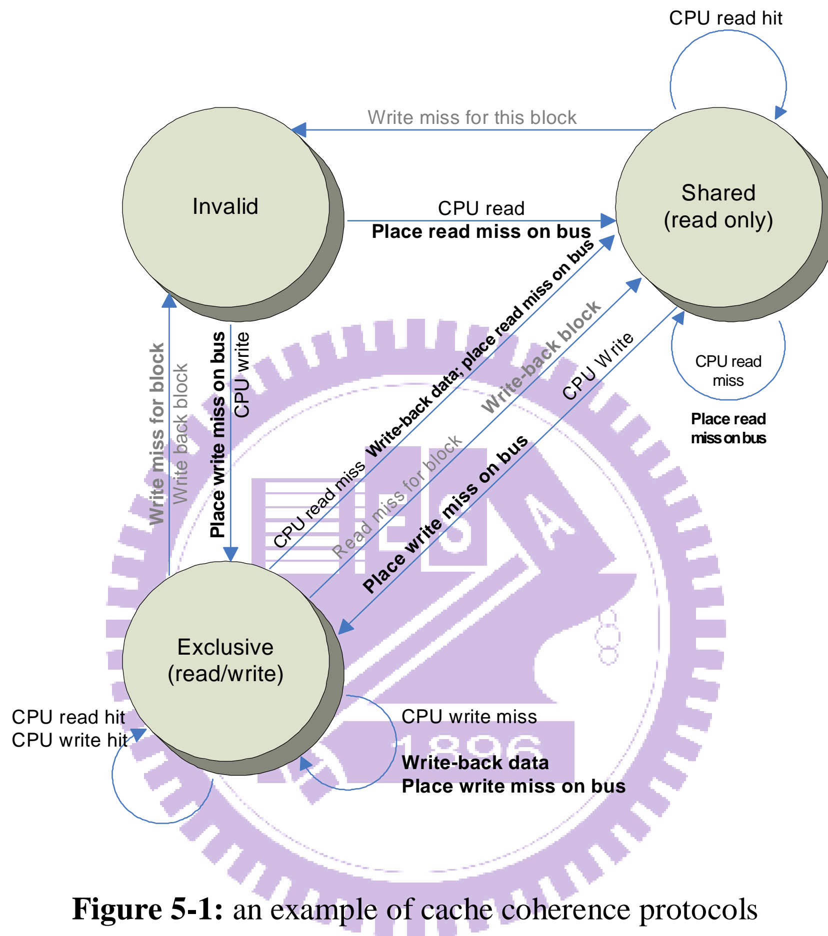


Figure 5-1: an example of cache coherence protocols

Reference

[1]A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies," IEEE Trans. Computers. vol. 11, no. 12, Dec. 1978, pp. 7-21.

[2]Jouppi, N. 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 (May 1990), 364–373

[3]JOHNSON, T. L. AND HWU, W. W. 1997. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (May 1997), 364–373.

[4]J.D. Collins, D.M. Tussen. 1999. Hardware identification of cache conflict misses. In *Proceedings of the 32nd International Symposium on Microarchitecture* (Nov. 1999), 126–135.

[5]J.D. Collins, D.M. Tussen. Runtime Identification of Cache Conflict Misses: The Adaptive Miss Buffer. In Proceedings of the ACM Transactions on Computer Systems, 2001

[6] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," Proc. 29th Ann. Int'l Symp. Computer Architecture (ISCA 02), IEEE Press, 2002, pp. 171-182.

[7] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, page 96, Madrid, Spain, February 2004

[8] Simplescaler LCC http://www.simplescalar.com/

[9] SPEC CPU 2000 Benchmarks http://www.spec.org/

[10] J. L. Hennessy and D. A. Patterson. ComputerArchitecture - A Quantitative Approach. Morgan Kaufmsmn, 1990.

[11] HILL, M. D. 1987. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph. D. dissertation, University of California, Berkeley.