


REPLACEMENT AND PLACEMENT POLICIES FOR
PREFETCHED LINES



By
SZE SIU CHING

A DISSERTATION
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF PHILOSOPHY
DIVISION OF COMPUTER SCIENCE AND ENGINEERING
THE CHINESE UNIVERSITY OF HONG HONG
JUNE 1998





Copyright © 1998 by Computer Science and Engineering, the Chinese
University of Hong Kong
All right reserved.

Acknowledgements

I want to give my hearty thanks to my family.

I also want to give my hearty thanks to Dr. Gilbert H. Young, Mr. S. C. Lau, Mr. S. Y. Yiu and especially to Ms. Priscilla K. Chan.

It is clear there are many to whom I owe my thanks and acknowledgments. In order not to miss anyone of them, I just want to say thanks.

Thanks for those who have helped me, taught me, carried me and supported me during my studies. Thanks a lot.

Abstract

As a result of technology advance, there is a widening gap between the rate at which a processing unit can consume operands and the rate at which the memory system can supply them. The introduction of cache helps alleviate this problem, and the design of cache memory is very critical to the overall system performance. Due to the limited space on the processors, on-chip caches are usually small. Therefore, the cache space should be used carefully and efficiently. Accurate prefetching and careful replacement of cache lines¹ are essential to improve the performance. In order to further improve the cache performance, different prefetching algorithms for cache have been proposed[BaC91] [KIL91] [Smi78a]. With prefetching, data could be available before their actual use. However, due to the large volume and the random behaviour of data usage, it is difficult to prefetch data accurately and this results in cache pollution.

Lau [Lau96] has proposed an accurate prefetching scheme, the Instruction Opcode and Addressing Mode Prefetching (**IAP**), which makes use of the future reference patterns embedded in certain instructions. Further to their study, it is also found that most prefetched data by the IAP scheme are likely to be referenced only once. Therefore, we proposed to use a mixed replacement policy to use together with the IAP scheme, to minimize the number of thrashing misses. Using both Least Recently Used (**LRU**) and Instant Zero (**IZ**) replacement algorithms with the IAP scheme outperforms the result of using LRU only.

Furthermore, in order to optimize the benefit of temporal locality and minimize

¹A **line** is a **block** of data in context of the cache. Usually, line size is the same as block size. In our paper, *block* refers to data located in memory or lower-level caches, *line* refers to the data in level 1 cache. However, they are interchangeable.

the cache pollution problem, another hardware replacement design is presented in this thesis. We propose a priority pre-updating scheme, which is used to update the priorities of cache lines prior to their normal updating situation. Simulation experiments are done with this priority pre-updating scheme in cache model with prefetch-on-miss prefetching scheme. From the results, it is found that priority pre-updating helps minimizing the number of thrashing misses, optimizes the benefit of temporal locality and reduces cache pollution. In order to obtain promising cache performance improvement, we add a victim cache to hold those fresh prefetched lines that displaced from the data cache. The experimental results show that using priority pre-updating with the victim cache can achieve up to 50% reduction in memory delay.

Beside the research on replacement of cache lines, we propose another hardware design, which concerns the placement of IAP lines. The cache lines prefetched by the Instruction Opcode and Addressing Mode Prefetching pose a referenced-once property, i.e., most of them are referenced one and only one time before the program terminates. Owing to this special reference behavior, a prefetch cache, which is dedicated to prefetched lines by Instruction Opcode and Addressing Mode Prefetching scheme, is implemented separately. The prefetch cache can reduce memory delay time up to 99%.

Contents

1	Introduction	1
1.1	Overlapping Computations with Memory Accesses	3
1.2	Cache Line Replacement Policies	4
1.3	The Rest of This Paper	4
2	A Brief Review of IAP Scheme	6
2.1	Embedded Hints for Next Data References	6
2.2	Instruction Opcode and Addressing Mode Prefetching	8
2.3	Chapter Summary	9
3	Motivation	11
3.1	Chapter Summary	14
4	Related Work	15
4.1	Existing Replacement Algorithms	16
4.2	Placement Policies for Cache Lines	18
4.3	Chapter Summary	20
5	Replacement and Placement Policies of Prefetched Lines	21
5.1	IZ Cache Line Replacement Policy in IAP scheme	22
5.1.1	The Instant Zero Scheme	23
5.2	Priority Pre-Updating and Victim Cache	27
5.2.1	Priority Pre-Updating	27
5.2.2	Priority Pre-Updating for Cache	28

5.2.3	Victim Cache for Unreferenced Prefetch Lines	28
5.3	Prefetch Cache for IAP Lines	31
5.4	Chapter Summary	33
6	Performance Evaluation	34
6.1	Methodology and metrics	34
6.1.1	Trace Driven Simulation	35
6.1.2	Caching Models	36
6.1.3	Simulation Models and Performance Metrics	39
6.2	Simulation Results	43
6.2.1	General Results	44
6.3	Simulation Results of IZ Replacement Policy	49
6.3.1	Analysis To IZ Cache Line Replacement Policy	50
6.4	Simulation Results for Priority Pre-Updating with Victim Cache .	52
6.4.1	PPUVC in Cache with IAP Scheme	52
6.4.2	PPUVC in prefetch-on-miss Cache	54
6.5	Prefetch Cache	57
6.6	Chapter Summary	63
7	Architecture Without <i>LOAD-AND-STORE</i> Instructions	64
8	Conclusion	66
A	CPI Due to Cache Misses	68
A.1	Varying Cache Size	68
A.1.1	Instant Zero Replacement Policy	68
A.1.2	Priority Pre-Updating with Victim Cache	70
A.1.3	Prefetch Cache	73
A.2	Varying Cache Line Size	75
A.2.1	Instant Zero Replacement Policy	75
A.2.2	Priority Pre-Updating with Victim Cache	77
A.2.3	Prefetch Cache	80

A.3	Varying Cache Set Associative	82
A.3.1	Instant Zero Replacement Policy	82
A.3.2	Priority Pre-Updating with Victim Cache	84
A.3.3	Prefetch Cache	87
B	Simulation Results of IZ Replacement Policy	89
B.1	Memory Delay Time Reduction	89
B.1.1	Varying Cache Size	89
B.1.2	Varying Cache Line Size	91
B.1.3	Varying Cache Set Associative	93
C	Simulation Results of Priority Pre-Updating with Victim Cache	95
C.1	PPUVC in IAP Scheme	95
C.1.1	Memory Delay Time Reduction	95
C.2	PPUVC in Cache with Prefetch-On-Miss Only	101
C.2.1	Memory Delay Time Reduction	101
D	Simulation Results of Prefetch Cache	107
D.1	Memory Delay Time Reduction	107
D.1.1	Varying Cache Size	107
D.1.2	Varying Cache Line Size	109
D.1.3	Varying Cache Set Associative	111
D.2	Results of the Three Replacement Policies	113
D.2.1	Varying Cache Size	113
D.2.2	Varying Cache Line Size	115
D.2.3	Varying Cache Set Associative	117
	Bibliography	119

List of Figures

2.1	Operations of <i>LOAD-UPDATE</i> and <i>STORE-UPDATE</i> (a) using the <i>index-displacement</i> addressing mode and (b) using the <i>index-based registers</i> addressing mode	7
2.2	Control flow for IAP scheme	10
3.1	Percentage of Prefetch-On-Miss lines that are not referenced in IAP scheme	14
5.1	A theoretical representation of a set in a four-way set associative cache	24
5.2	Before a reference to an IAP line	25
5.3	After reference to an IAP line	25
5.4	Control Flow of the IZ Replacement Policy	26
5.5	Architectural model of IAP scheme with PPU	28
5.6	Illustration of PPU	29
5.7	Control Flow of PPUVC	30
5.8	Cache Support in the IAP architecture	32
5.9	Control Flow of the Prefetch Cache Scheme	33
6.1	Trace-driven simulator using <i>xtrace</i>	35
6.2	Memory Model of the simulator: (a) Interleaved memory (b) Timing of data access	38
6.3	Comparison of number of prefetch-on-miss lines in IAP cache and prefetch-on-miss-only cache	53

6.4	Percentage of prefetch-on-miss lines referenced in total number of prefetched lines	54
6.5	The effect of Prefetch Cache size on cache performance	58
6.6	An illustration of the performance difference between LRU and FIFO	61
6.7	An instance of line activities in IZ prefetch cache	63
A.1	MCPI by varying cache size in IZ scheme	69
A.2	MCPI by varying cache size in PPUVC with IAP scheme	71
A.3	MCPI by varying cache size in PPUVC with prefetch-on-miss scheme	72
A.4	MCPI by varying cache size in prefetch cache scheme	74
A.5	MCPI by varying cache line size in IZ scheme	76
A.6	MCPI by varying cache line size in PPUVC with IAP scheme . . .	78
A.7	MCPI by varying cache line size in PPUVC with prefetch-on-miss scheme	79
A.8	MCPI by varying cache line size in prefetch cache scheme	81
A.9	MCPI by varying set associative in IZ scheme	83
A.10	MCPI by varying set associative in PPUVC with IAP scheme . . .	85
A.11	MCPI by varying set associative in PPUVC with prefetch-on-miss scheme	86
A.12	MCPI by varying set associative in prefetch cache scheme	88
B.1	Results of the first group programs in IZ	89
B.2	Results of the second group programs in IZ	90
B.3	Results of the third group programs in IZ	90
B.4	Results of the first group programs in IZ	91
B.5	Results of the second group programs in IZ	91
B.6	Results of the third group programs in IZ	92
B.7	Results of the first group programs in IZ	93
B.8	Results of the second group programs in IZ	93
B.9	Results of the third group programs in IZ	94

C.1	Results of the first group programs in PPUVC	95
C.2	Results of the second group programs in PPUVC	96
C.3	Results of the third group programs in PPUVC	96
C.4	Varying line size	97
C.5	Results of the second group programs in PPUVC	97
C.6	Results of the third group programs in PPUVC	98
C.7	Varying set associative	99
C.8	Results of the second group programs in PPUVC	99
C.9	Results of the third group programs in PPUVC	100
C.10	Results of the first group programs in PPUVC	101
C.11	Results of the second group programs in PPUVC	101
C.12	Results of the third group programs in PPUVC	102
C.13	Results of the first group programs in PPUVC	103
C.14	Results of the second group programs in PPUVC	103
C.15	Results of the third group programs in PPUVC	104
C.16	Results of the first group programs in PPUVC	105
C.17	Results of the second group programs in PPUVC	105
C.18	Results of the third group programs in PPUVC	106
D.1	Results of the first group programs	107
D.2	Results of the second group programs	108
D.3	Results of the third group programs	108
D.4	Results of the fourth group programs	108
D.5	Results of the first group programs	109
D.6	Results of the second group programs	109
D.7	Results of the third group programs	110
D.8	Results of the fourth group programs	110
D.9	Results of the first group programs	111
D.10	Results of the second group programs	111
D.11	Results of the third group programs	112

D.12 Results of the fourth group programs	112
D.13 Results of the first group programs	113
D.14 Results of the second group programs	113
D.15 Results of the third group programs	114
D.16 Results of the fourth group programs	114
D.17 Results of the first group programs	115
D.18 Results of the second group programs	115
D.19 Results of the third group programs	116
D.20 Results of the fourth group programs	116
D.21 Results of the first group programs	117
D.22 Results of the second group programs	117
D.23 Results of the third group programs	118
D.24 Results of the fourth group programs	118

List of Tables

6.1	SPEC Benchmark Applications used	34
6.2	Percentages of LOAD/STORE-UPDATES in SPEC92 Benchmark Suite	39
6.3	Baseline CPIs of SPEC92 Benchmark Suite	43

Chapter 1

Introduction

Cache memory is a special high speed memory designed to supply the processor with the most frequently requested instructions and data. Instructions and data located in cache memory can be accessed many times faster than instructions and data located in main memory. The more instructions and data the processor can access directly from cache memory, the faster the computer runs as a whole.

Memory caching is effective because most programs access the same data or instructions over and over. By keeping as much of this information as possible in cache memory (which is usually implemented with faster SRAM), the computer avoids accessing the slower main memory (which is usually implemented with slower DRAM). Some memory caches are built into the architecture or microprocessors. Such internal on-chip caches are often called Level 1 (L1) caches. Cache memory makes use of the principal of locality. Locality of reference states basically that even within very large programs with several megabyte of instructions, only small portions of this code generally get used at once. Programs tend to spend large periods of time working in one small area of the code, perform the same job many times with slightly different operands, and move on to another area of code for another batch of routine jobs. This occurs because of *loops*, which are what programs use to do work many times in a rapid succession.

Generally, there are two kinds of localities – temporal locality and spatial locality. Temporal locality describes the likelihood that a recently-referenced address

will be referenced again soon, while spatial locality describes the likelihood that a close neighbor of a recently-referenced address will be referenced soon. Conventional cache memories rely on a program's temporal and spatial localities to reduce the average memory access latency.

The gap between main memory and processor clock speeds is growing at an alarming rate. As a result, the system performance is increasingly dominated by the latency of servicing memory accesses, particularly those accesses which are not easily predicted by the temporal and spatial localities captured by conventional cache memory organizations [Smi82] [HeP95].

One obvious way to reduce number of the cache misses is enlarging the cache as much as possible, however, it is often difficult to achieve practically. There are two main reasons that limit the size of Level 1 cache: [1] The performance gained is not enough to compensate the cost for cache, which typically uses fast but expensive static RAM chips. The speed for SRAM is approximate 4 times faster than DRAM, however, SRAM chips cost more than six times as much as the DRAM chips normally used for main memory. Besides, the performance improvement is not linearly proportional to the size of cache, that is, a 512K bytes cache memory may not obtain 2 times better performance than a 256K bytes one. [2] The CPU chip is usually small while SRAM size is comparable large, thus only limited space for Level 1 cache, while maintaining a reasonable processor chip size.

Due to the large speed gap between the processor and main memory, it is obvious that performance of the system will then be largely determined by [1] how effectively the on-chip memory is able to manipulate operands, minimize the frequencies of off-chip accesses, and [2] the rate at which the external memory system can supply operands.

The main aim of cache memory is to reduce the CPU's idle waiting time. Improving cache performance of programs is one way of increasing the systems throughput. The effectiveness of the on-chip cache to maintain useful operands and minimizing the frequencies of off-chip accesses is one of the main factor to determine the performance of the system.

In order to reduce the disparity between processor speed and memory access time, many solutions have proposed to tackle this problem. Some have proposed adding additional features such as non-blocking fetches [Kro81], victim caches [Jou90], and sophisticated hardware prefetching [ChB92] to alleviate the access penalties for those references that have locality characteristics that are not captured by most conventional designs.

1.1 Overlapping Computations with Memory Accesses

Many solutions have been proposed to reduce the memory access and/or hide memory latency. An important approach is cache prefetching [Smi78a][Smi78b][Smi82][HeP95], that is, the action of bringing data to the cache before they are actually needed. Prefetching is similar to speculative loads in the sense that it is non-blocking and behaves like a hint without incurring semantic faults. The main difference between prefetching and speculative loads is that data are loaded into the caches rather than registers.

Depending on how prefetch requests are determined and initiated, prefetching can be either hardware-controlled [BaC91][FuP91][FuP92] or software-directed [Por89][KIL91][MoL92]. The hardware approach detects accesses with regular patterns and issues prefetches at the run time of the programs, whereas the software approach relies on the compiler to analyze programs and to insert prefetch instructions during compilation of the programs.

However, because of the low accuracy of some prefetching algorithms, there is a risk that the prefetched data that are never used before they are displaced from the cache. This leads to waste of memory space and bandwidth, thus poorer performance results. The problem become worse when the prefetched data displace some useful data in the cache. The phenomenon is called *cache pollution*.

1.2 Cache Line Replacement Policies

Different replacement policies are employed to manage operands in the memory. Replacement takes place when a particular cache line or a set of cache lines is already full, and the line has to be evicted its contents to make room for the new incoming line. There is still no ideal replacement policy being invented, and it is unlikely that one would exist. Replacement policy in cache is different from the problem of replacement in paged main memories because the cache replacement algorithm must be implemented entirely in hardware and must execute very quickly so as to catch up with the processor speed.

Least Recently Used and *First-In-First-Out* are two most commonly used replacement policies. Beside these two well-known algorithms, there are *Random*, *Pseudo-Least Recently Used* which are used in some special systems.

Not knowing whether a line will be accessed soon, *Least Recently Used* strategy is usually used in conventional cache as the replacement scheme. However, if the displaced line is referenced by the processor again, a thrashing miss¹ will occur. The situation may become worse, since one thrashing miss can lead to another thrashing miss. A good cache line replacement policy should try to find out the best candidate to be displaced and will help minimize these thrashing misses.

1.3 The Rest of This Paper

In this dissertation, we focus on techniques on better management of the pre-fetched lines for cache with the IAP scheme and that with a traditional prefetch scheme – prefetch-on-miss only.

IAP is an accurate prefetching scheme, in which it makes use of the information provided by the instruction opcodes and addressing modes for prediction. A brief review on IAP scheme will be given in Chapter 2.

Chapter 3 will briefly describe the cache pollution problem brought by conventional prefetching schemes, the reference-once property of IAP lines and how

¹A thrashing miss occurs when the line which was replaced must itself be reloaded

IAP works.

Chapter 4 will briefly describe previous research on prefetching, and stream buffers.

The implementation of the replacement policy, Instant Zero (**IZ**) cache line replacement policy, will be discussed in Chapter 5. The details of another replacement policy, Priority Pre-Updating (**PPU**), which is designed to be used with different types of prefetching algorithms, will be given in the same chapter. The effect of using Priority Pre-Updating with victim cache (**PPUVC**) is also discussed in this chapter.

In the same chapter, details on the placement policy of prefetched lines, prefetch cache, will be given.

Chapter 6 will present the simulation methodology, performance metrics that used and the results of performance evaluation. The designs are evaluated by simulating the some benchmark program in a uniprocessor environment. The results show that either IZ, PPU with victim cache and prefetch cache alone can obtain significant improvement in system performance.

Finally, Chapter 7 will give an insight on future directions, and we will conclude this paper in Chapter 8.

Chapter 2

A Brief Review of IAP Scheme

2.1 Embedded Hints for Next Data References

In the design of latest processor architectures, instruction opcodes and the addressing modes of the architecture definition usually have built-in mechanism to support the address calculation of future data references while the current datum is being referenced. It is also found that compound instructions are commonly used in RISC architecture to reduce the program execution path length. As it can be found from program instrumentation and tracing, certain simple RISC instructions are executed in pair. So it might be useful to define a single compound or extended opcode to execute the instruction pair, and this is particular useful if the new instruction opcode does not affect the processor clock cycle. Up to now, there are several machines have such kind of opcodes. For example, *ADD-AND-BRANCH*, *COMPARE-AND-BRANCH*, *LOAD-WORD-AND-UPDATE*, etc. in HP's Precision Architecture 1.1 [HP94]. IBM and PowerPC has *LOAD-UPDATE*, *LOAD-MULTIPLE*[IBM89] [Mot92] [IBM94] [WeS94], etc.. The total number of instructions defined in current RISC processors range from 150 and 200, which is much larger than that of early RISC processors (about 50 to 70 instructions). The reason is that latest processors find these compound or extended opcodes to be very useful, and this embedded them into the instruction set. Among these compound instructions, it is found that the *LOAD/STORE-UPDATE* (or

LOAD/STORE-MODIFY), are very helpful to manage on-chip cache activities.

Array or pointer references to a large set of data are one of the major types of data references in typical programs. Data will be referenced one after another successively, *index-displacement* and *index-based register* addressing modes are usually employed for this type of accesses. Because these accesses occur very frequently, as a result, many systems tend to use compound opcodes like *LOAD-UPDATE* and *STORE-UPDATE* for the accesses. Beside loading or storing a datum into the register, the content of the index register, which is used in the address calculation of current data reference, will be updated by each of these instructions. The operations of the *LOAD-UPDATE* and *STORE-UPDATE* instructions using either *index-displacement* or *index-based* registers addressing mode are shown in Figure 2.1.

$LOAD\ R_T(R_x + Disp)$ Equivalent to $Eff.\ Addr. = (R_x) + Disp$ $R_T = (Eff.\ Addr.)$ $R_x = Eff.\ Addr.$ (a)	$LOAD\ R_T(R_x + R_y)$ Equivalent to $Eff.\ Addr. = (R_x) + (R_y)$ $R_T = (Eff.\ Addr.)$ $R_x = Eff.\ Addr.$ (b)
---	---

Figure 2.1: Operations of *LOAD-UPDATE* and *STORE-UPDATE* (a) using the *index-displacement* addressing mode and (b) using the *index-based registers* addressing mode

The updating action of the *LOAD-UPDATE* or the *STORE-UPDATE* instruction is the preparation of the content of register R_x which is used in calculating of the effective address of the next expected datum. R_x is equal to the sum of the current data reference address *Eff. Addr.* (or the updated content of register R_x) and the displacement *Disp* (in the *index-displacement* addressing mode) or the register content R_y (in the *index-based register* addressing mode). Thus, accurate data cache prefetching can be carried out and the address of prefetched data is equal to $(Eff.\ Addr. + Disp)$ or $(Eff.\ Addr. + R_y)$. It should be noted that values

of *Eff.Addr.* and *Disp* (or R_y) are available to the cache prefetching unit during the execution of $LOAD\ R_T(R_x + Disp)$ or $LOAD\ R_T(R_x + R_y)$ instruction.

2.2 Instruction Opcode and Addressing Mode Prefetching

By using these hints of data references provided by the instruction opcode and addressing modes, Instruction Opcodes and Addressing Mode Prefetching (**IAP**) scheme which provides accurate data prefetching for on-chip cache is proposed by Lau [Lau96]. In Lau's study, IBM POWER architecture (or the PowerPC architecture) is used as an example to show how IAP scheme should be designed and implemented.

Figure 2.2 shows the control flow of IAP scheme. For each instruction i that is decoded and executed, its opcode will be checked first to determine if it belongs to *LOAD-UPDATE* or *STORE-UPDATE* instruction. If such a case is detected, the address of next datum expected to be referenced in the near future will be re-calculated. Using the same addressing mode as i but with the updated contents of all registers used in the address calculation of i . Afterwards, this new address will be sent to the cache prefetch unit for accurate data prefetching. Beside these basic ideas, two enhancements have been integrated into the IAP scheme:

- **Default Prefetching vs. Selective Prefetching**

When executing each *LOAD/STORE* instruction i , if this instruction i belongs to *LOAD/STORE-UPDATE* instructions group, then the IAP scheme will be used for data prefetching, else the prefetch-on-miss is used as the default prefetching scheme for data prefetching.

- **Cache Block Prefetching vs. Next Data Reference Prefetching**

For each data prefetch requested by IAP scheme, if the target prefetched block j containing the candidate datum is not the same as current data

referencing line i , then a prefetch of block j will be issued. If they are the same, then a prefetch request of block $i + 1$ will be issued.

The above IAP scheme together with the two enhancements are the *combined IAP* that we use in this paper.

2.3 Chapter Summary

In this chapter, a general design for hardware controlled prefetching, which was proposed by Lau [Lau96], is introduced. By using information embedded in the instruction opcodes, Lau's design is able to single out the data references with constant strides from the pool of all data references and also able to find the corresponding stride values. With this valuable information, accurate prefetching can be accomplished and consequently, the CPU stall time due to data cache misses can be reduced. The cache block prefetching is introduced to tackle the problem of limited memory bus bandwidth. In order to exploit the spatial locality, the combined IAP scheme is equipped with default prefetching to issue prefetch requests for the non-*LOAD/STORE UPDATE* instructions.

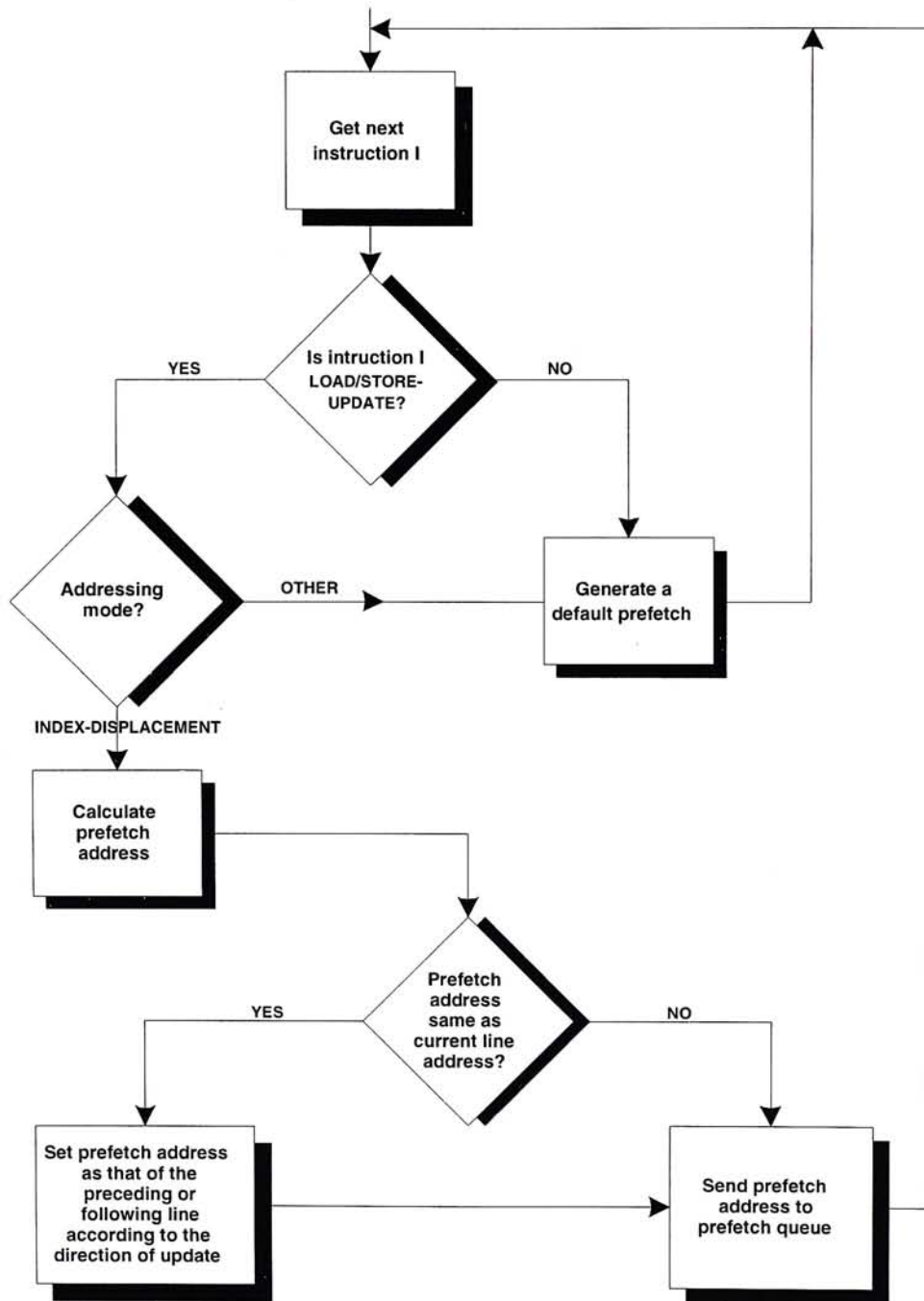


Figure 2.2: Control flow for IAP scheme

Chapter 3

Motivation

Over the last two decades, the CPU clock cycle time has been decreasing at a much faster rate than the main memory access time. The average number of cycles per instruction has also been decreasing dramatically. The effect is more obvious for RISC machines with higher clock speed and data consumption rate.

Unfortunately, a high bandwidth of the microprocessor is meaningless unless it is matched by a similarly powerful memory subsystem. Most of current microprocessors rely on caches to reduce their effective memory access time. However, cache miss affects the overall performance of a system, i.e., if either the instruction or the operand required by the operations is not found in cache(s), the actual performance would decline for the large amount of cache misses.

With current VLSI developments, several functional units, instruction and data caches, and some special hardware functional units can be included on the processor chip. Therefore, a first obvious method for reducing the average memory access time is to implement multi-level cache hierarchies [BaW89] with an on-chip first level cache. However, under the usual caching mechanism, the processor will still be stalled on a first-level cache miss and of course also on misses on any of the next levels of the memory hierarchy with an even larger penalty time, until the miss is resolved. Since a processor must stall on a cache miss, caches do not totally hide memory latency but, instead, they eliminate many off-chip memory accesses. In order to make further progress towards the reduction of

memory latency, memory accesses due to cache misses must proceed in parallel with the processor execution. As a result, a number of different solutions have been proposed to allow computations to be overlapped with memory accesses. They basically provide efficient mechanisms to allow buffering and pipelining of memory references.

Various data prefetching algorithms exist, some are hardware-assisted, some are software-directed and others are hybrid. The main fault of many of these algorithms is that they do not integrate replacement algorithms with prefetching methods. There is often a large penalty for prefetching into the cache because the wrong line was replaced.

When incorporating the prefetching algorithms in a processor, several things have to take into consideration. First, it is possible to prefetch data into the cache that will never be used by the processor. This not only pollutes the cache, but also increases memory traffic. Second, if the data is prefetched too early, it can become stale before it is referred, this may also increase memory traffic. Therefore, in designing a processor with prefetching, careful balance between performance gains and tradeoff like cache pollution and memory traffic increase are required.

From a different viewpoint, a conventional cache's hardware does not know the likelihood of whether a line will be accessed soon. A blind strategy is usually used to choose the line to be replaced when a miss has occurred, e.g. choose the *least recently used* line. However, if it happens that the displaced line is referenced by the processor again, a thrashing miss will occur. The problem becomes more serious since one thrashing miss can lead to another thrashing miss.

It is estimated that 50% of all misses are thrashing misses, and that most of these can be avoided. A good cache replacement policy will help minimize these thrashing misses.

There exists an accurate prefetching, the Instruction Opcodes and Addressing Modes Prefetching, which is proposed by Lau [Lau96]. IAP, which making use of the run-time information provided by the instruction opcodes and addressing modes, prefetches data accurately. However, it is found that data prefetched by

IAP scheme have not posed the temporal locality property, a large portion of the data prefetched by the IAP scheme ¹ are likely to be referenced one and only one time ² before the program terminates. In order to handle the replacement of the data lines in the cache, a new strategy, tentatively termed as *Instant Zero (IZ)*, is proposed. This new strategy aims at replacing the IAP lines intelligently, which will be explained in detail later.

On-chip cache is usually small ³, thus the precious cache space should be used carefully. Cache pollution problem highly affects the system performance, one obvious solution to solve the cache pollution problem is to kick out useless data in case of conflict or capacity misses. However, which data is useless and how to determine which should be kicked out is really a difficult problem. A poorly designed prefetching algorithm aggravates cache pollution problem, and wrong displacement of useful data degrades system performance. From Figure 3.1, we can find out that in some benchmark programs, more than 90% of prefetch-on-miss lines are unreferenced. It is obvious that most prefetch-on-miss lines are useless, i.e., they are not referenced before the program terminates. It is beneficial to shorten the life time of those possibly erroneously prefetched lines in the cache to minimize cache pollution. We propose a *Priority Pre-Updating (PPU)* scheme to tackle the problem, PPU helps determining the data to be kicked out and reduce cache pollution.

As mentioned above, IAP lines are likely to be referenced one and only one time before the program terminates. Therefore, placing them in a separate cache space can localize their effects and minimize the cache pollution problems. As a result, we proposed to use an on-chip *prefetch cache* to hold all those data prefetched by IAP scheme.

¹lines prefetched by IAP scheme will tentatively called IAP lines in later sections.

²Termed as reference-once property

³Usually ranging from 4K bytes to 32K bytes. Though large on-chip cache is also found in current architecture, it is not common.

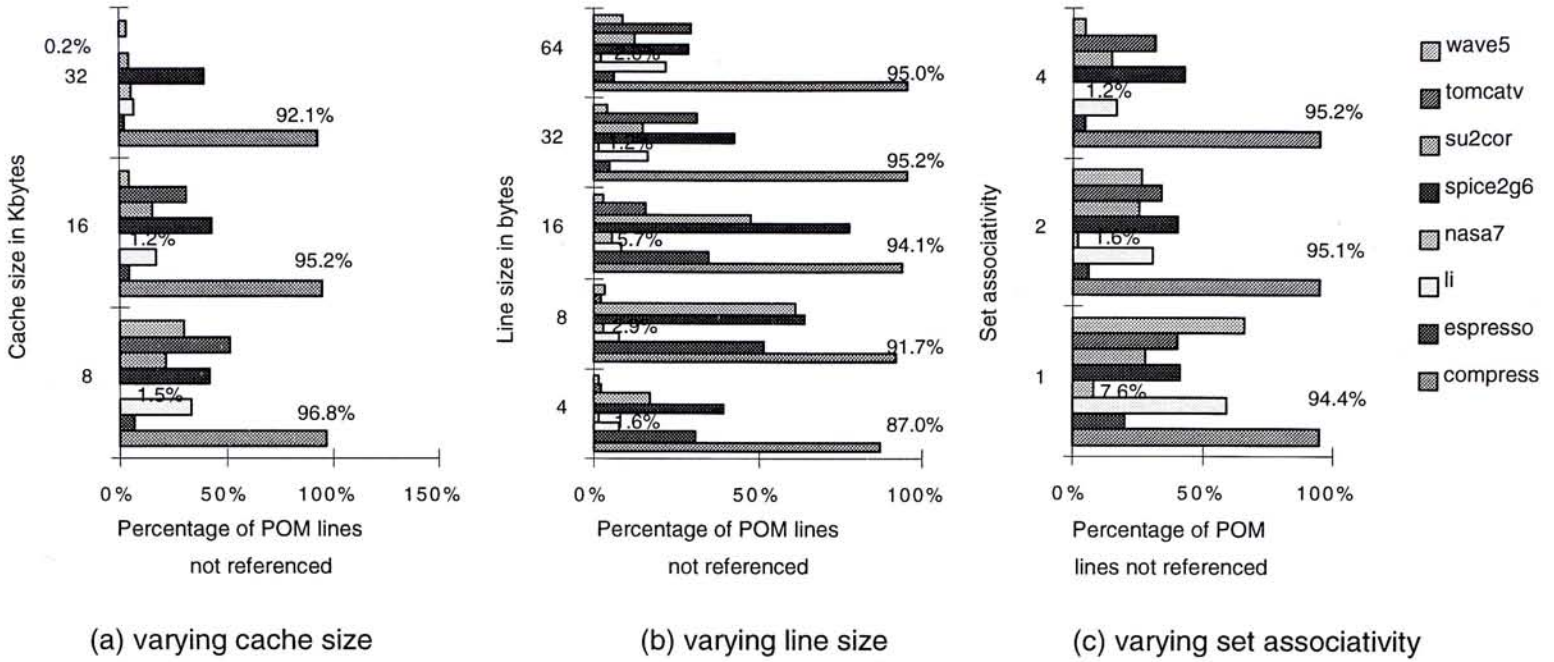


Figure 3.1: Percentage of Prefetch-On-Miss lines that are not referenced in IAP scheme

3.1 Chapter Summary

Cache pollution is a side-effect of data prefetching, a poorly designed prefetching algorithm aggravates cache pollution problem. This problem has diverse effect on cache performance. A blind replacement strategy increase the thrashing misses, reduce the utilization of cache and indirectly cause cache pollution. The techniques, *Instant Zero* replacement policy and *Priority Pre-Updating with victim cache*, tend to alleviate this problems and improve the cache performance. The cache lines prefetched by IAP scheme have a reference-once property, and placing them in a separate cache space, prefetch cache, is able to localize their influence and also reduce the cache pollution problem.

Chapter 4

Related Work

For recent computer applications, it is common that there are many matrix manipulations with highly regular and sequential data references, and a lot of data are needed in performing computation. If the operands are not found in cache, the actual performance of the system would decline for the large amount of cache misses.

In order to reduce the number of cache miss penalty, data should be prefetched into the cache before their actual usage. Prefetching techniques consist of both hardware and software approaches. Existing cache prefetching schemes, either hardware-driven [BaC91] [Smi78b] or software-assisted [Tha81][Bre87][Por89] [GoG90][CaK91][ChM91][KIL91][MoG91][MoL92], are not very effective in reducing the processor idle time due to memory accesses.

Hardware prefetching typically uses dynamic stride detection to perform run-time calculation of prefetch addresses to be issued [ChB92] [FuP91] [FuP92]. The overheads of hardware prefetching are the cost for the additional hardware, and the limited ability of the dynamic units to perform any prefetching other than through arrays with linear strides.

The prefetching accuracy of traditional hardware driven data prefetching schemes is low (though it is relatively easier to be implemented), thus cannot get significant improvement in data cache performance. Though we can find some accurate

hardware-driven prefetching schemes of constant stride array elements, they usually require some complicated add-on hardware such as a prediction table. As a result, they are not suitable to be implemented as the first-level on-chip cache as the space on the CPU chip is very limited. Chen and Baer [ChB94] evaluated the effectiveness of lockup-free caches and hardware prefetching, and proposed a hybrid scheme based on a combination of these approaches.

Software prefetching is more flexible than hardware prefetching, having the advantage of compile-time knowledge, but pays the price of software overhead, both in instructions issued and code size [CaK91] [KIL91] [MoL92]. Software-assisted cache prefetching schemes can also achieve high accuracy in prefetching array data references with constant strides, but the runtime overhead introduced is a big obstacle to their popularity. Furthermore, architectural and compiler supports are needed for the software-assisted prefetching schemes. These also restrict the usage of software prefetching scheme in current processors and computer systems. Besides these, some promising approaches use hybrid hardware and software techniques, issuing limited instructions that provide hints to the prefetch hardware [Chi94].

When the cache or a particular set is full, and information is requested by the CPU from the lower level memory, some information in the cache must be selected for replacement. This implies that a cache miss needs not only a fetch but also a replacement. Cache replacement policies should implement totally in hardware and execute very quickly, so it will not have bad influence on the system performance. The replacement algorithms are mainly classified into usage-based and non-usage-based. Section 4.1 will give a brief description of some known replacement policies.

4.1 Existing Replacement Algorithms

In brief, cache misses can occur for three reasons: [1] the requested data have never been accessed before (*compulsory miss*), [2] the requested data have been accessed before, but the size of the working data set exceeds the cache size (*capacity miss*),

or [3] the requested data had been in the cache but was displaced by an intervening reference to another address (*conflict miss*). Information resident in the cache has to be removed to bring in future information in the event of cache misses. The replacement algorithm determines the information to be discarded. The algorithm may be *Least Recently Used (LRU)*, *First In First Out (FIFO)*, *Random*, *Pseudo-LRU*, etc.

A truly random strategy is completely unacceptable for production test reasons, as it is difficult to run test vectors on a chip that does not have completely deterministic behavior. Some relatively common replacement algorithms are as follows:

1. *Least Recently Used (LRU)*: An usage-based algorithm under which the line which has not been accessed for the longest time is replaced with the hope of reducing the chance of throwing out information that will soon be needed again. Its implementation requires every line to have extra bits to keep track of the age of its contents thus making the controller design more complicated.
2. *First in First Out (FIFO)*: The First In-First Out replacement policy chooses the page which has been in the memory the longest to be the one replaced, i.e., the page to be replaced is the *oldest* page in the cache, the one which was loaded before all the others. A pointer into the line space is maintained. On replacement, the line pointed by the pointer is ejected, and the pointer is incremented. The pointer is set to zero when the end of the line space is reached.
3. *Clock (or Second-chance)*: A pointer in the line space is maintained. On replacement, the *used* bit of the line pointed to by the pointer is checked. If it is set, it will be cleared and the pointer is incremented. The last step is repeated until a line with the *used* bit cleared is found and that line is ejected. The *used* bit is set on every access, and is cleared periodically. This method can be used to approximate LRU, but the periodicity of the clearing needs to be carefully set. It will be difficult to find an eject-able

line if the period is too long. If the period is too short, locality will be lost and thrashing will occur frequently.

4. *Least Recently Modified*: The *LRU* bits of lines is modified only on writes.
5. *Not-Most-Recently-Used*: The most recently used line is kept in the cache, one of the remaining lines is selected and replaced.

6. *Least Frequently Used (LFU)*

The page to be replaced is the one used least often of the pages currently in the cache.

7. *Last In First Out (LIFO)*

The page to be replaced is the one most recently loaded into the cache.

8. *Optimal (OPT or MIN)*

The page to be replaced is the one that *will not be* used for the longest period of time. This algorithm requires future knowledge of the reference string which is not usually available. Thus, this policy is used for comparison studies.

Among all of the above algorithms, the usage-based LRU is most commonly-used in current memory design. As mentioned above, implementation for LRU has to keep track of the age of every cache line, and thus requires every cache line to have extra bits. Though this makes the controller design more complicated and expensive, it works well in most architecture. FIFO and Random are non-usage-based algorithms, non-usage-based algorithms use basis other than usage for replacement decision. It is shown that non-usage-based algorithms all yield comparable performance [Smi82].

4.2 Placement Policies for Cache Lines

Techniques on holding prefetched data in intermediate space other than the first-level cache has also been proposed. Jouppi [Jou90] proposed to use a stream

buffer to hold the prefetched data. Stream buffers prefetch cache lines starting at a cache miss address. The prefetched data is placed in the buffer instead of the cache. Stream buffers are useful in removing capacity and compulsory cache misses, as well as some instruction cache conflict misses. However, the stream buffer that proposed is actually a simple FIFO queues, and thus each time only the *oldest* element is visible to the processor. However, the newest replaced lines instead of older one are needed sometimes. As a result, the expected performance improvement in data cache is slight or nil. Therefore, multi-way stream buffer, which consists of four parallel stream buffers in a multi-way stream buffer and with LRU replacement policy, is proposed to solve the limited ability of stream buffer. When a miss occurs in the data cache that does not hit in any stream buffer, the least recently hit stream buffer is cleared and it is started fetching at the miss address. However, the utilization of buffer is still low, as only the first entry in each buffer can be searched.

Jouppi [Jou90] has proposed another technique, miss caching, to minimize the miss penalty during a cache miss. A miss cache is a small fully-associate cache containing two to five cache lines of data. When a miss occurs, data is returned not only to the normal (upper) cache, but also to the miss cache under it, where it replaces the least recently used item. Each time the upper cache is probed, the miss cache is probed as well. If a miss occurs in the upper cache but the address hits in the miss cache, then the directed mapped cache can be reloaded in the next cycle from the miss cache. This replaces a long off-chip miss penalty with a short one-cycle on-chip miss.

To make better use of the miss cache, victim caching is further proposed by Jouppi [Jou90]. Victim caching use a different replacement algorithm for the small fully-associative cache. Instead of loading the requested data into the miss cache on a miss, load the fully-associative cache with the victim line from the direct-mapped cache instead. With victim caching, no data line appears both in direct-mapped cache that hits in the victim cache, the contents of the direct-mapped cache line and the matching victim cache are swapped.

4.3 Chapter Summary

In this chapter, a brief review on different prefetching algorithms is given. Besides, review on existing replacement and placement policies is also introduced.

Chapter 5

Replacement and Placement Policies of Prefetched Lines

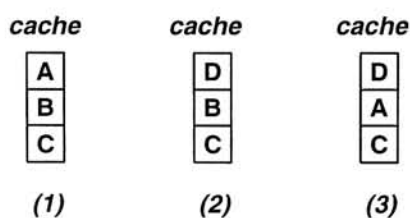
In order to minimize the cache pollution and localize the influences brought by prefetched lines, we have tried different approaches. The different schemes try to focus on handling the life time of prefetched lines in the cache, and placement of IAP lines. Firstly, we propose to use a mixed replacement policy with both IZ and LRU replacement policies, this scheme helps to shorten the life time of referenced and useless IAP lines and it retains the temporal locality of demand-fetched lines. Secondly, the Priority Pre-Updating scheme (PPU) is proposed to shorten the life time of possibly erroneously prefetched lines. In addition to IAP scheme, we found that PPU can work well on caches employing different prefetching algorithms. Thirdly, we use a on-chip prefetch cache to hold the prefetched data to localize the influences of IAP lines, Following sections will give details on these three schemes.

5.1 IZ Cache Line Replacement Policy in IAP scheme

Least Recently Used (LRU) is the most commonly used cache line replacement policy in both traditional cache designs and the IAP scheme mentioned in previous chapters. Though LRU is very popular in current cache designs, it still has many drawbacks. It is known that LRU cannot always replace the best line in the cache, and replace a wrong candidate line may cause cache miss afterwards. For example, consider the following code segment:

```
for(count = 0; count < 3; count++)
    for(i = 0; i < 4 ; i++)
        a[i] += b[i];
```

let blocks A, B, C and D contain the data $b[0]$, $b[1]$, $b[2]$ and $b[3]$ respectively. If the cache is fully associated, with LRU chosen as the replacement policy, then the data blocks will be loaded into the cache in the following sequences:



In (2), due to limited capacity, A is replaced by D as it is the least recently used one. However, block A is immediately in need after D when the outer loop is entered again (in (3)), and thus a cache miss follows. This situation continues until the end of the loops. The same situation occurs when FIFO policy is used.

Though there is no perfect cache line replacement scheme found so far, a good replacement algorithm should try to reduce the probability of wrongly replacing a useful cache line.

Although the IAP scheme can prefetch data accurately, the cache may not large enough to accommodate all lines brought in by demand fetch and prefetch

requests. As a result, conflict misses occur frequently when the working set of the program is larger than the cache size, so we have proposed the Instant Zero replacement policy [SzY97] is proposed to handle the problem. IZ scheme is aimed at managing the cache replacement more efficiently and reducing the thrashing misses. It is found that the data in IAP lines are most likely to be referenced once only. If these prefetched lines are placed in the cache and obey the LRU cache line replacement policy, as they have lost the benefit of referencing in the near future once after being referenced, they are just occupying the precious space in cache without any contribution. As a result, most of them should be the best candidate to be replaced when either capacity or conflict misses occur. That is the reason why the proposed IZ replacement policy is used to handle these IAP lines. As a result, after the requested data of an IAP line has been referenced, it should be the best candidate to be discarded when there are conflicts among the cache lines.

5.1.1 The Instant Zero Scheme

In the IAP scheme, conflict misses are the main concern. If a line (say line i) is being referenced, and was not found in the cache, then a miss occurs. The idea of IZ is not to reduce the miss penalty caused by the reference to line i , but to minimize the probability of cache misses in the future. We can easily observe that most cache lines that generated by demand fetch or prefetch-on-miss possess certain degrees of localities, and those lines prefetched by the IAP scheme are likely to be referenced once only. As a result, the IAP lines that have been referenced should be the best candidate to be discarded if conflicts on the cache lines occur. Therefore, the replacement strategy in the IAP scheme is a mixed strategy by using both IZ and LRU.

The mixed replacement policy of LRU and IZ can be summarized as follows. [1] The replacement policy for non-IAP scheme cache lines (either by demand fetch or by default prefetching) still obey the LRU policy. [2] For those lines prefetched by the IAP scheme will follow either LRU or IZ according to the following rule :

- If the prefetch address is not the same as the current data reference line, then this prefetched line will follow the LRU policy. On the other hand, if the prefetch is the same as current data reference line, then as mentioned before, the block preceding it or following it will be prefetched. And this prefetched line will obey the IZ policy, in which the priority of the line will set to 0 immediately after its reference.

To indicate whether a line is prefetched by the IAP scheme, 1 extra bit for each cache line is needed. The cache lines in a four-way set associative cache will look like Figure 5.1.

D ₁	H ₁	I ₁	S ₁	Ptag ₁	Data ₁
D ₂	H ₂	I ₂	S ₂	Ptag ₂	Data ₂
D ₃	H ₃	I ₃	S ₃	Ptag ₃	Data ₃
D ₄	H ₄	I ₄	S ₄	Ptag ₄	Data ₄

1 2 1 1

D Dirty bit

H Hot bit, which indicates the priority of the corresponding data line

I IAP bit, set if the line is prefetched by the IAP scheme

S Line status bit (Valid bit):

Ptag Physical tag

Data Cache Data

Figure 5.1: A theoretical representation of a set in a four-way set associative cache

The hot bits are used to indicate the priorities of the lines following LRU replacement policy. When a line in the cache is referenced or a new line is brought into the cache, the hot bits of the lines will be updated. In the former case, the referenced line will have the hot bits updated to the largest number within the same set. Those lines with hot bits value larger than the original value of that of the referenced line will have their hot bits decreased by 1, while others remain unchanged. In the latter case, the hot bits value of each cache line will be decreased by 1, and the one with new value equal to 0 will be displaced. The new line will have its hot bits value set to the highest number within the set. Therefore, lines

with a lower priority are more likely to be kicked out, as they are the *Least Recently Used* one. In case of conflict in the cache lines, the line with hot bits equal to 0 will be displaced in order to bring in a new line.

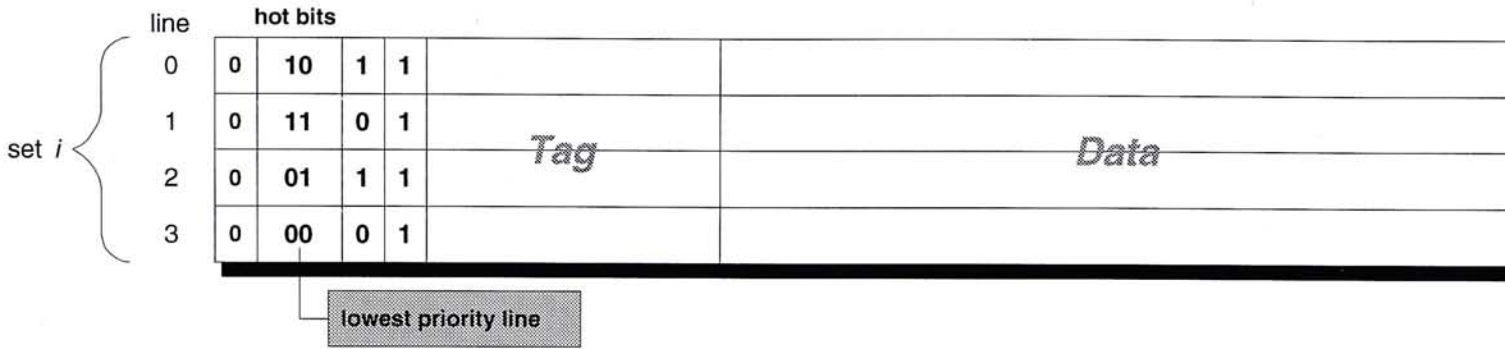


Figure 5.2: Before a reference to an IAP line

Now, let us have a look on how the IZ works within a set (say set i). Refer to Figure 5.2, if there has a miss to set i , then line 3 will be displaced. Since it is the one with lowest priority (the *least recently used* one). Suppose there is a reference go to line 0 before such a miss occurs, as it is an IAP line (with IAP bit set), it is then considered useless and will be the most likely one to be kicked out in future conflicts. Other than setting the priority of line 0 to 0, those lines with priorities lower than the original priority of line 0 should be incremented by 1. Those lines with priorities higher than the original priority of line 0 remain unchanged. As a result, the lines in the set with the priorities updated will look like Figure 5.3. Therefore, if there is a miss now, then line 0 instead of line 3 is the first one to be displaced.

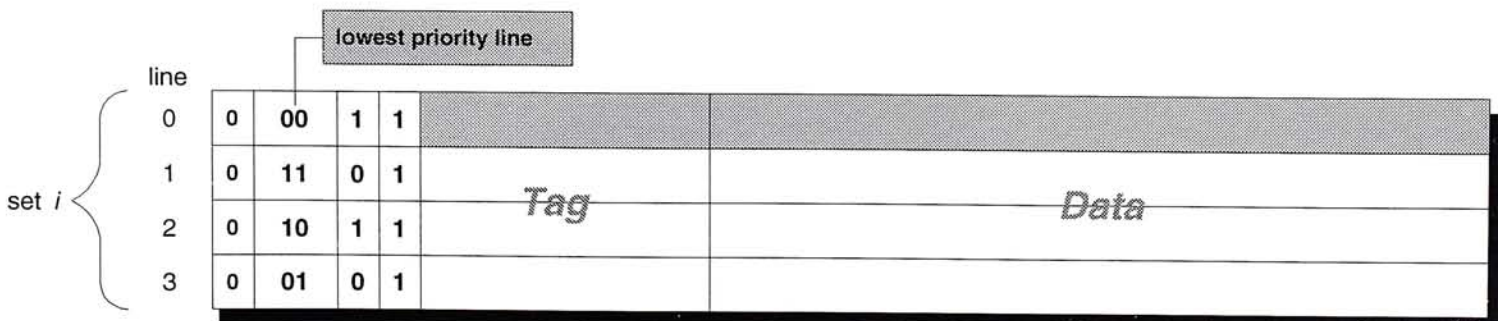


Figure 5.3: After reference to an IAP line

The control flow of the proposed scheme is shown in Figure 5.4.

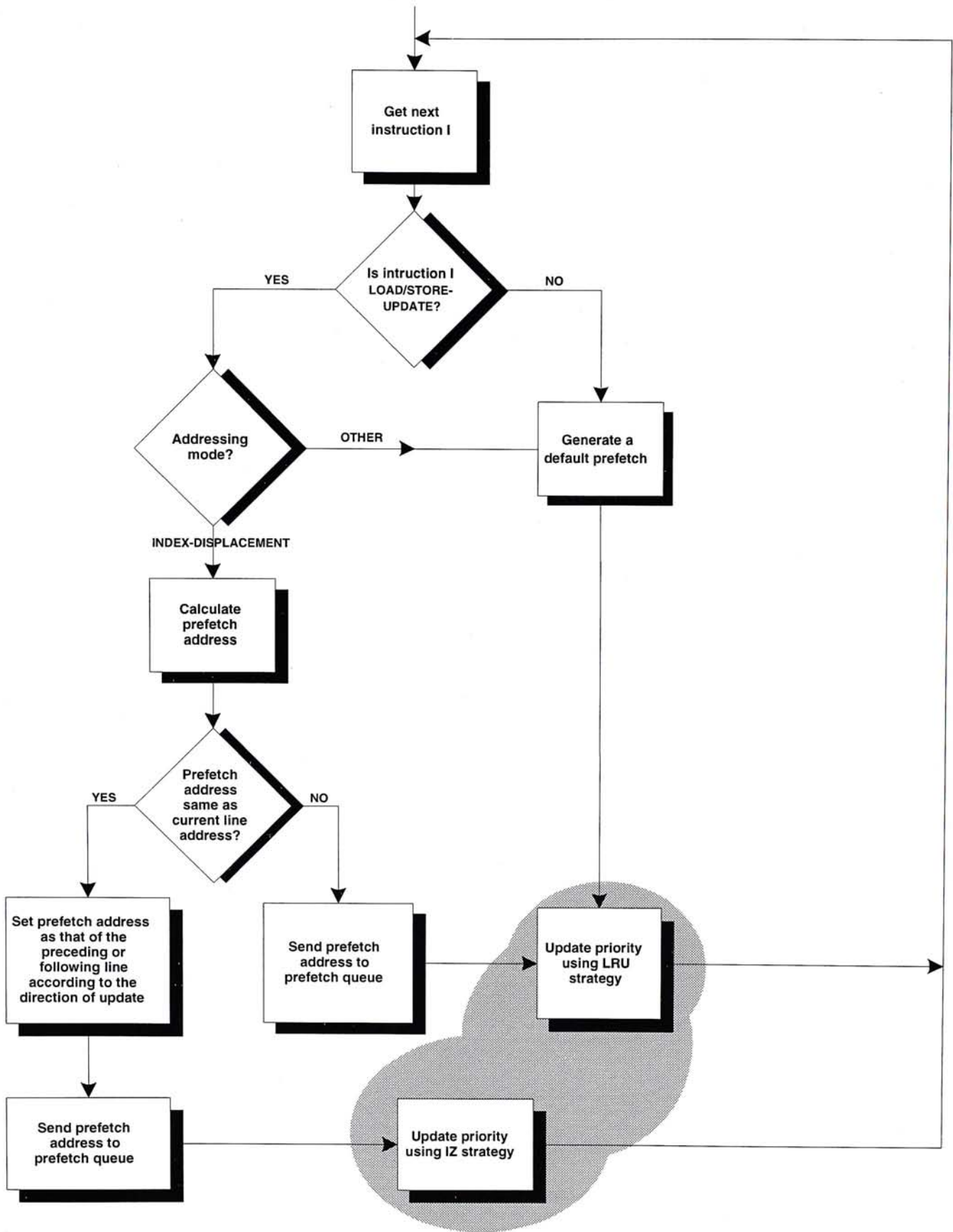


Figure 5.4: Control Flow of the IZ Replacement Policy

If a line is prefetched by the IAP scheme, then the IAP bit in the corresponding entry will be set to 1. These IAP lines will be treated as normal cache lines until there is a reference to it. When an IAP line is referenced, its priority will be set to 0 immediately after its reference to obey the IZ replacement policy.

5.2 Priority Pre-Updating and Victim Cache

In order to minimize the cache pollution and localize the influences brought by prefetched lines, we propose the Priority Pre-Updating scheme (**PPU**) to shorten the life time of possibly erroneously prefetched lines. The effect of adding a small fully-associative victim cache to hold the unreferenced prefetched lines ejected from cache is also discussed here.

5.2.1 Priority Pre-Updating

A PPU unit is added to keep track of all the prefetched lines according to their time of prefetching. Whenever there is a reference to a prefetched line, priorities of those former prefetched lines, which are unreferenced and precede the current referenced line in the PPU unit, will be decremented by a constant stride. Normally, the priority of a line updates only when there are references to any lines within the *same set*. However, under the PPU scheme, the priorities of other lines in different sets may also be changed.

In fact, the PPU aims at reducing the cache pollution problem as well as retaining the potential temporal locality of other lines. Figure 5.5 shows the overview of the architectural model of the cache under the PPU scheme.

As shown in Figure 5.6, each entry in PPU maps to a prefetched line in the cache. When there is a reference go to a cache line, say $line_{03}$ (entry 3 in PPU), since $line_{01}$ and $line_{10}$ precede $line_{03}$ in PPU, their priorities will be decreased by 1. The cache lines following $line_{03}$ in PPU, such as $line_{n0}$, will not be affected.

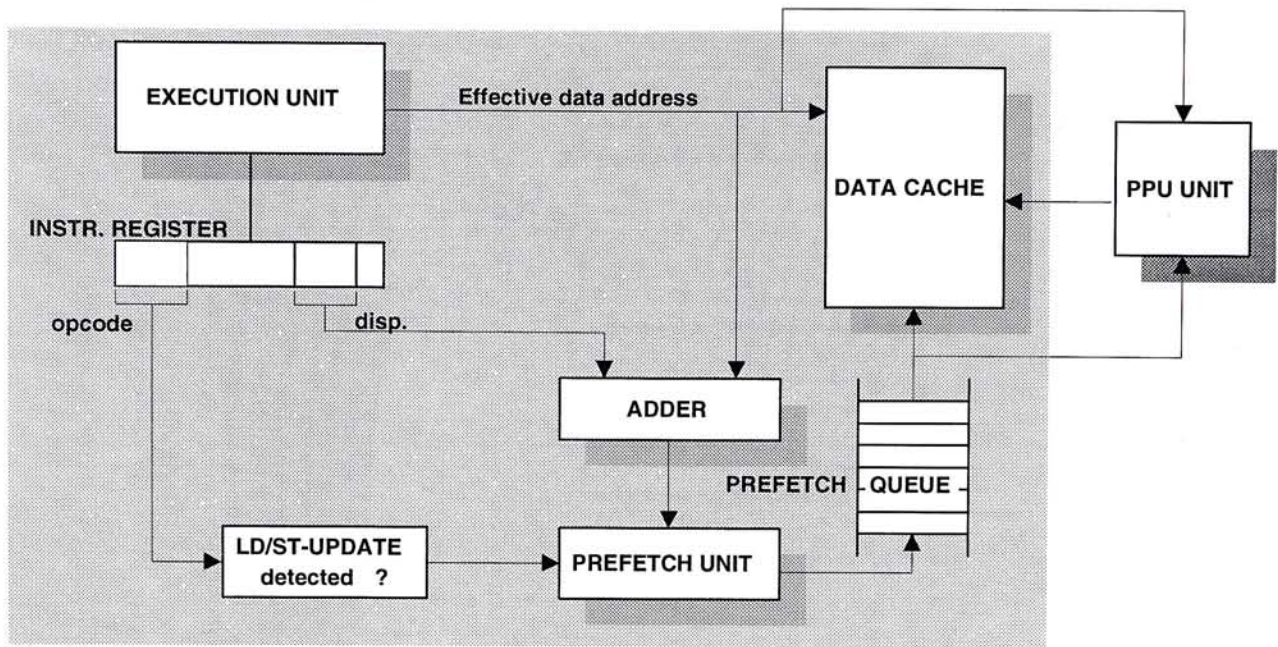


Figure 5.5: Architectural model of IAP scheme with PPU

5.2.2 Priority Pre-Updating for Cache

The basic rules of PPU are – [1] All prefetch-on-miss lines will be recorded in PPU unit, and [2] When there is a reference, if the requested line is found in PPU unit, those preceding *unreferenced* lines will have their priority decrement by 1. [3] The corresponding entry in the PPU unit for this reference line will be deleted.

5.2.3 Victim Cache for Unreferenced Prefetch Lines

Owing to the long delay between references of successive data, the priority pre-updating scheme cannot obtain significant improvement in the cache performance, Data which have high potential to be referenced in the near future, are sometimes trashed out before their actual reference. In order to minimize the influence of this situation, we use a small victim cache with few entries as a secondary buffer for the unreferenced prefetch line. Therefore, a small fully-associative victim cache with four entries, each with the same size as a cache line, is integrated with cache. As experimental results show that victim caches with FIFO and LRU replacement policies have similar performance, FIFO is chosen as replacement policy in victim cache, for the ease of implementation in hardware.

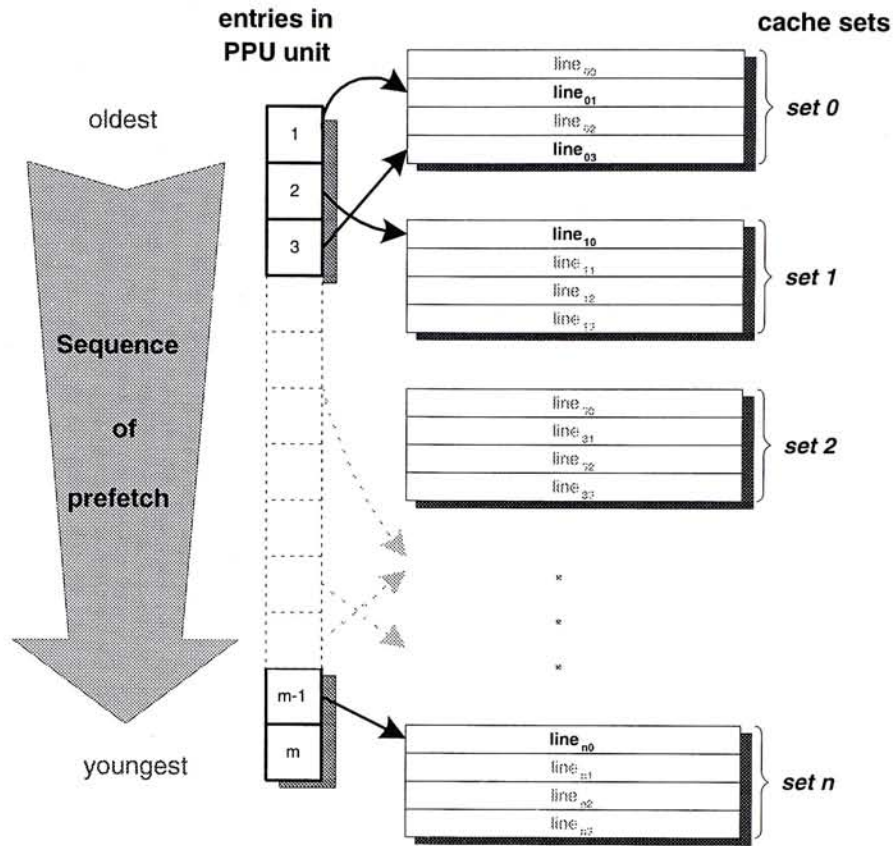


Figure 5.6: Illustration of PPU

For each prefetched line discarded from the cache, it will first be checked if this line is unreferenced. If such a case is detected, this line will be placed in the victim cache. When the victim cache is full, the *oldest* line will be displaced. When there is a data reference to the cache, it will first check if the data appears in the data cache. If the data is not found in data cache, then it will check if the data is in the victim cache. If such a case happens, then one extra clock cycle will be spent to fetch the data from victim cache.

Actually, the victim cache itself does not prefetch data but act as a secondary buffer to store prefetched data. Experimental results show that the cache performance is significantly improved by using PPU with victim cache (PPUVC). Figure 5.7 gives a summary of the control flow of the PPUVC.

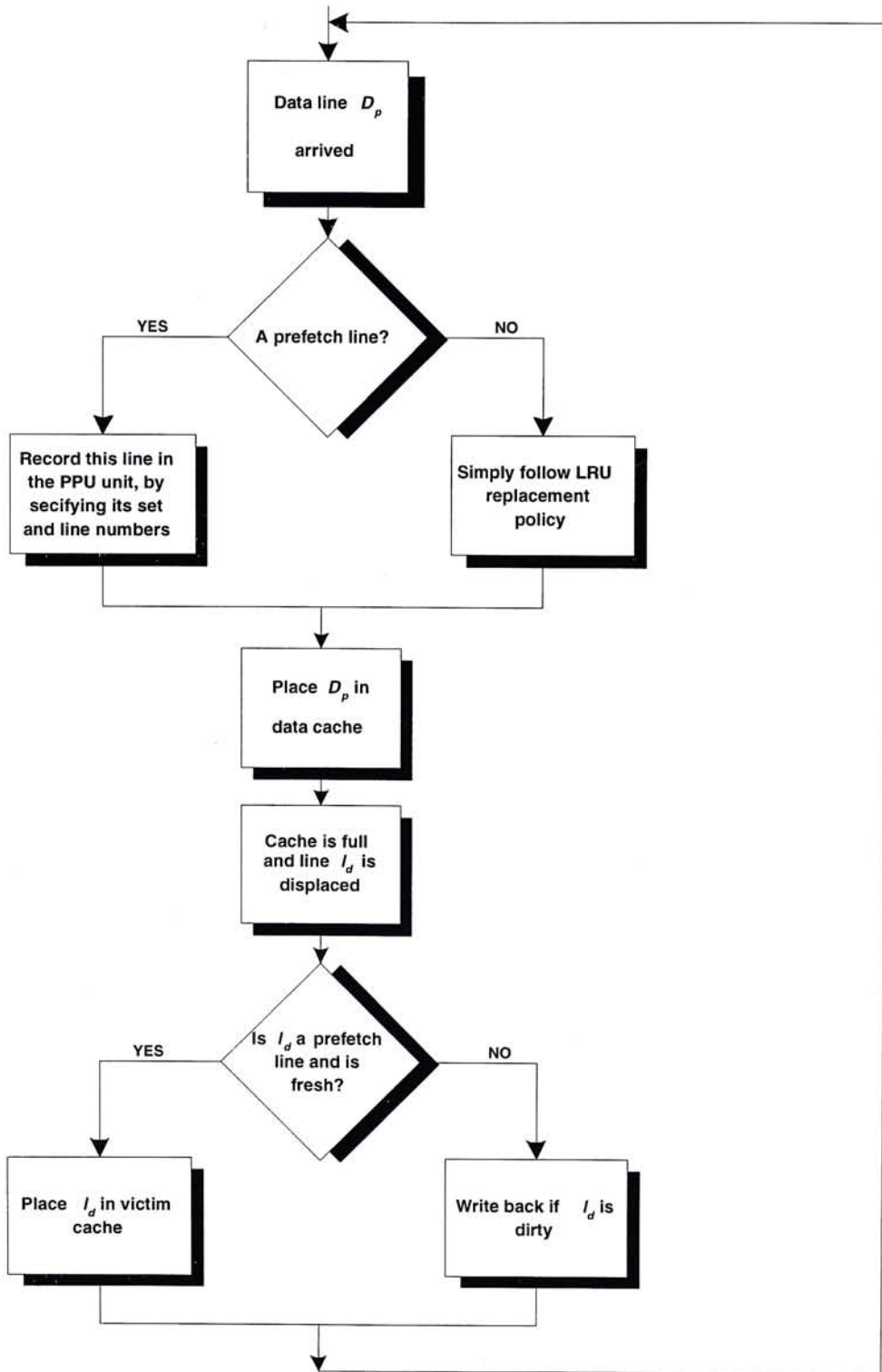


Figure 5.7: Control Flow of PPUVC

When a data line arrives, it will first check if it is a prefetched line. If so, then a record will be kept in the PPU unit. When there is a reference to any cache line, then priorities of the lines will be checked and updated. If there is any conflict occurs in the cache and a line has to be replaced, then the one with lowest priority will be discarded. If the line to be discarded from cache is an unreferenced prefetched line, then it will be placed to the victim cache. On the other hand, the lines that have been referenced will not be placed in the victim cache. These lines will be written back to lower level memory if they have been modified. Otherwise, they will be simply discarded from cache.

5.3 Prefetch Cache for IAP Lines

IAP lines have quite different property as compared to demand-fetch lines and prefetch-on-miss lines, since they are likely to reference only once during the entire program execution time. In order to localize their effects and minimize their influences to normal cache lines, a small on-chip prefetch cache is added [YoS98]. All IAP lines will be placed in the prefetch cache instead of the on-chip cache. A prefetch cache has the same structure as a cache, which consists of a series of entries with a tag, an valid bit, a dirty bit, a hot bit and a data line. Prefetch cache functions independently from the data cache. When there is a reference, both data cache and prefetch cache will be checked in parallel for any potential hits. Figure 5.8 gives the general picture of cache support in IAP architecture.

Through the on-chip cache controller, the processor attempts to access the data in the primary cache. If the data is there, then the processor will retrieve it. If a primary cache miss occurs, i.e., the data is not found in the primary cache, the cache controller checks to see if the data is in the secondary cache. If the data is found in the secondary cache, it is fetched into the primary cache. If the data is not present in the secondary cache, it is fetched as a cache line from main memory and is written into both the secondary cache and the primary cache before the processor retrieves it. For the sake of simplicity, we assumed there is no secondary

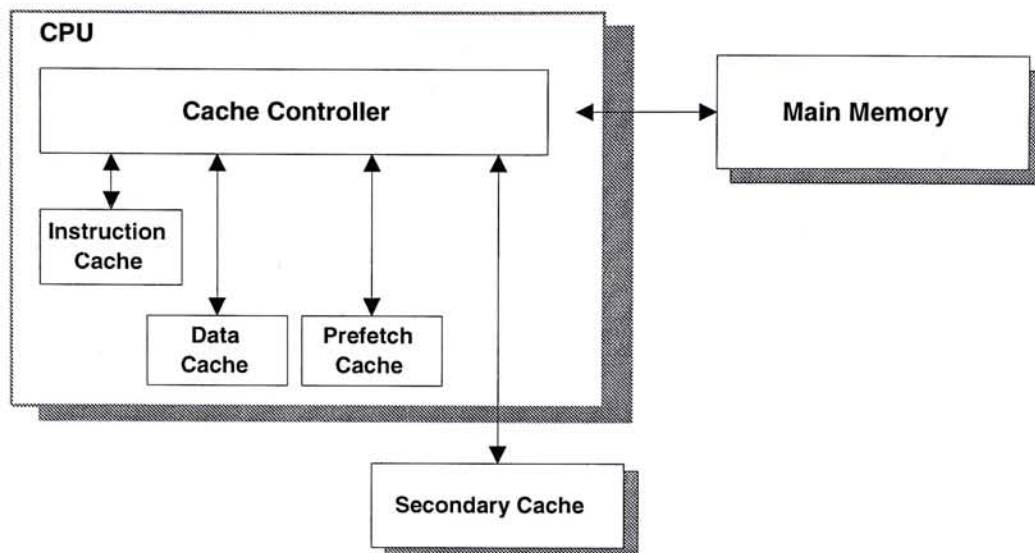


Figure 5.8: Cache Support in the IAP architecture

cache misses. In other words, the secondary cache is assumed to be infinitely large to hold all data. It is possible for the data to appear in different levels of memory. The data is kept consistent through the use of write back methodology, in which modified data is not written back to memory until the cache line is replaced. For any data prefetched by the prefetch unit, if this prefetch is generated by the IAP scheme, then it will be placed in the prefetch cache instead of the normal data cache. The data in prefetch cache is also kept consistent through write back methodology.

In addition to implementing LRU, the conventional cache replacement policy, in the prefetch cache, we also try using the IZ and FIFO replacement policies in it. We have just implemented one non-usage-based algorithm, FIFO, since non-usage-based algorithms are found to have similar performance as mentioned by Smith [Smi82].

Similar to the victim cache, the prefetch cache itself does not prefetch data but only keep prefetched data available for use. Figure 5.9 gives a summary of the control flow of the prefetch cache.

For each data line arrived in cache, it will be checked if it is a prefetched line. If the data line is a demand fetch line, then place it in the normal data cache. Otherwise, it will be further checked to see if it is prefetched by the IAP scheme. If such a case is detected, then it will be placed in the prefetch cache, and all

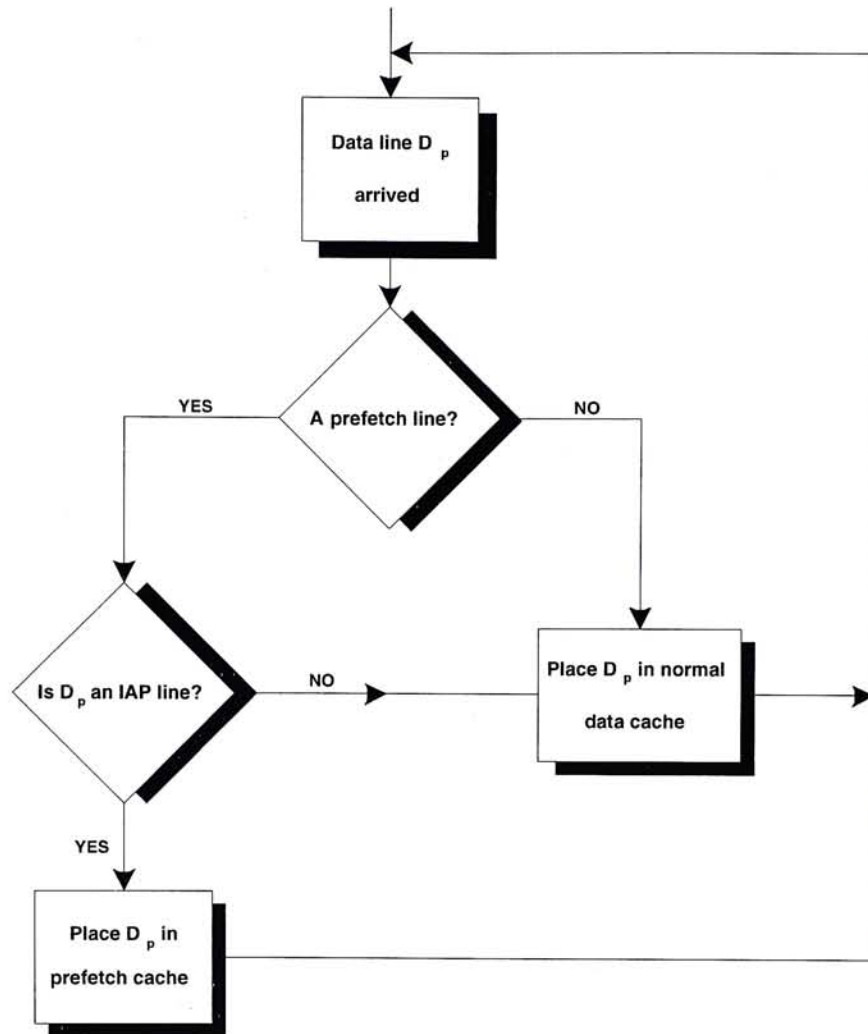


Figure 5.9: Control Flow of the Prefetch Cache Scheme

other types of prefetched lines will be placed in the normal data cache.

5.4 Chapter Summary

In this chapter, two designs for hardware controlled replacement schemes and one replacement policy are proposed. By careful selection of data line to be replaced when a cache miss occurs, our replacement policies, IZ and PPUVC, will be able to reduce the cache pollution problem and retain the benefit of localities posed by normal cache lines. With the reference-once property of IAP lines, the placement design, prefetch cache, is able to localize the influence of IAP lines. Again, it helps to alleviate the cache pollution problem.

Chapter 6

Performance Evaluation

6.1 Methodology and metrics

In order to have a deeper understanding of the algorithms proposed, and to show their potentials, we evaluate our proposed architecture using detail trace-driven cache simulation. We use eight SPEC92 ¹ benchmarks to generate traces for our study. They include three integer-intensive programs: compress, espresso and li, and five floating-point intensive programs: nasa7, spice2g6, su2cor, tomcatv and wave5. Table 6.1 gives a brief description of the benchmarks used.

SPEC Benchmark Suite		
Program	Language	Description
compress	C	Adaptive Lempel-Ziv compression
espresso	C	Boolean function minimization
li	C	Lisp interpreter solving the nine queens problem
nasa7	Fortran	Seven floating-point synthetic kernels
spice2g6	Fortran	Analog circuit simulator
su2cor	Fortran	Quantum physics mass computation
tomcatv	Fortran	Mesh generation program
wave5	Fortran	Maxwell's equation solver

Table 6.1: SPEC Benchmark Applications used

¹SPEC is a trademark of the Standard Performance Evaluation Corporation

6.1.1 Trace Driven Simulation

Clearly, the type of jobs (i.e., the *job mix* or *instruction mix*) will be important to the cache simulation, since the cache performance can be highly data and code dependent.

For example, given a cache size of 8K lines with an anticipated miss rate of 10%, (1 miss in 10 memory references) we would require about 80K lines to be fetched from memory before it could reasonably be expected that each line in cache was replaced. To determine reasonable estimations of actual cache miss rates, each cache line should be replaced a number of times (the *accuracy* of the determination depends on the number of such replacements.) This net effect is a memory trace of some factor larger is required, say another factor of 10, or about 800K lines. That is, the trace length would be at least 100 times the size of the cache. Therefore, we chose to collect 100 millions instructions for each benchmark programs.

With the help of *xtrace* facilities, each of the chosen benchmark programs in the SPEC92 suite was traced on the IBM RS/6000 workstation and 100 million instructions for each benchmark were collected. The process of trace-driven simulation is summarized in Figure 6.1.

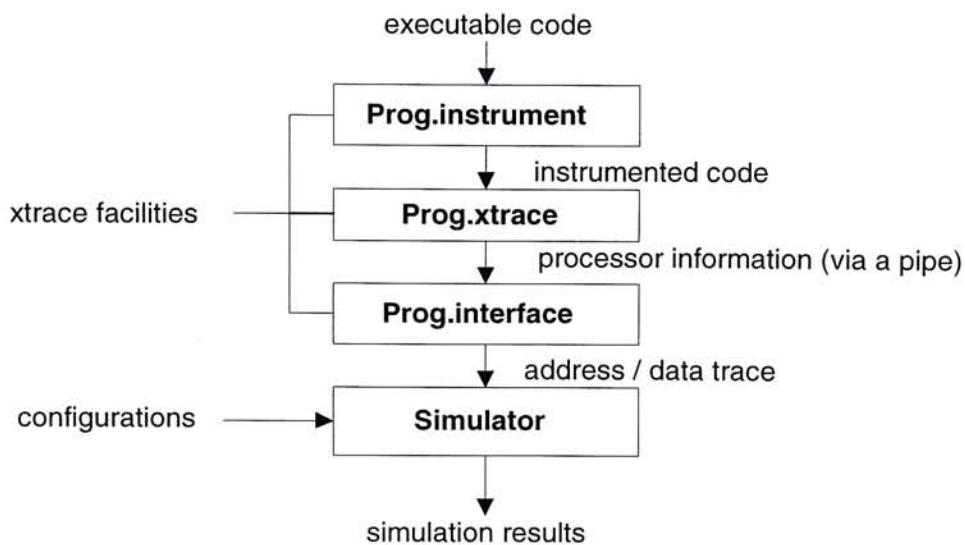


Figure 6.1: Trace-driven simulator using *xtrace*

The benchmarks were firstly compiled on the RS/6000 workstation. The executable codes were then instrumented by the program *instrument*, which inserted extra codes into the executable codes in order to extract the processor information during the program execution. The instrumented codes were then handled by *xtrace* and executed on the RS/6000 machine. The extracted processor information was then passed to the program *interface* via an explicit pipe. The program *interface* could be defined by the users to produce the desired trace format. The following information was recorded for each instruction that was traced:

Inst_Address, Inst_Content, < Data_Ref_Address if any >, < No_of_Bytes_Ref if any >

The simulator is designed such that it can read in the configuration descriptions such as cache size, set-associativity, line size etc., and simulation objects such as the CPU and the memory system are created based on these parameters. The trace data were stored in hard disk, such that a number of simulators could be run in parallel on different machines to speed up the simulations. The simulator read in the trace recorded one by one. Then the content of each instruction was decoded and the opcode, the addressing mode together with the register(s) used in the address calculation were found.

6.1.2 Caching Models

The baseline cache and the proposed caches use a write-back, write-allocate policy, and an 8-entry prefetch queue. We assumed that the processor has an ideal instruction cache with no instruction cache miss incurred. An elementary architectural model, which consists of a processor with perfect pipelined and a 4-way associative data cache with a line size of 32 bytes and a total size of 16K bytes, is defined for the simulations and the replacement algorithm is assumed to be LRU (Least Recently Used). For comparison, each dimension of the cache (cache size, line size and set associativity) is varied respectively for different simulations (cache sizes range from 8K bytes to 32K bytes, line size from 16 bytes to 64 bytes, and

set associativities simulated from 1 to 4) while the other two are kept constant.

The memory model of the second level cache in the simulations is assumed to be interleaved and its design and timing characteristics are shown in Figure 6.2. The memory is organized into a number of banks (or modules) to handle multiple words at one time rather than a single word. Each bank is one word wide which is the same as the first level cache and the bus. A cache line usually consists of a number of words (for example, a 32-bytes line consists of eight 4-byte words). Whenever a cache miss occurs in the first level cache and a fetching request is sent to the second level cache, the banks will work simultaneously – bank 0 will start reading for the first word in the block, bank 1, the second word, bank 2, the third word,... etc. However, since there is only one memory bus between the first and second level cache, the transfers of the words must be processed sequentially. As a result, the time for a demand fetch request to transfer a cache line between the first level cache and the second level cache memory can generally be summarized by the equation $C_1 + C_2 \times (block_size - 1)$, where C_1 is the delay time for the first word to arrive after a cache miss (that is, *startup_overhead + transfer_time_for_a_word*) and C_2 as a parameter that indicates the bus bandwidth between the first level cache and the second level cache (that is, transfer time for a word). In our experiments, C_1 was assumed to be 6 and C_2 to be 1.

For a given cache line size, the time for a demand request (due to the first level cache miss) to finish is assumed to be equal to the time for a prefetch (to the second level cache) to finish. The prefetch requests are reside in the prefetch queue and the request R_p at the beginning of the queue will be sent to the second level cache if the queue is not empty and the bus to the second level cache is free. If the address of the pending prefetch request immediately follows that of the current request R_c (demand fetch or prefetch) being processed in the second level cache (i.e. *address_of_Rp = address_of_Rc + 1*), the interleaved memory, that is, the second level cache, does not need to wait until the request R_c is completely finished. It can continue to process R_p when some memory banks are free, although the memory bus may be still transferring the data of R_c . In this

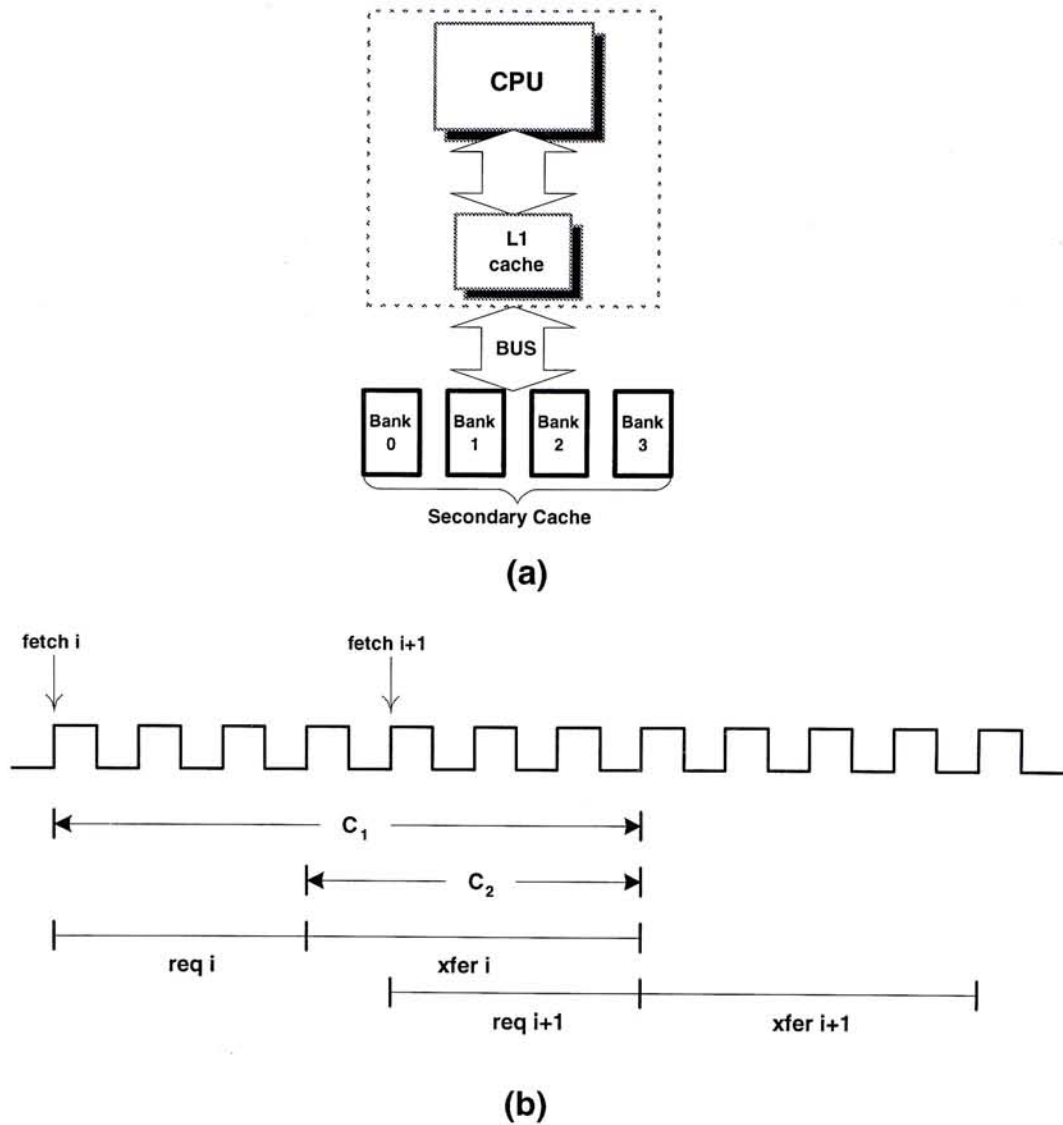


Figure 6.2: Memory Model of the simulator: (a) Interleaved memory (b) Timing of data access

case, the startup overhead of R_p can be hidden and the time for completing the prefetch request will be equal to $C_2 \times \text{block_size}$.

The second level cache can only handle one request at a time, no matter it is a demand fetch or prefetch request. When a demand fetch miss occurs in the first level cache, it will try to fetch the data from the second level cache. However, it may be in a situation that the second cache is serving a prefetch request. In case of such conflict, the priority will be given to the demand fetch miss and the prefetch request will be aborted and the demand fetch request will be started next cycle.

For simplicity, the second level cache (memory) is assumed to be infinitely large. That is, there is no cache miss in the second level cache.

Each instruction is assumed to be executed in one cycle and no superscalar architecture is simulated and cache access upon a cache hit is assumed to be one cycle.

6.1.3 Simulation Models and Performance Metrics

In Table 6.2, the percentages of *LOAD/STORE-UPDATE* instructions in the instruction mix of SPEC92 benchmarks programs are shown. As it can be seen from the percentages of the table, *LOAD/STORE-UPDATE* is fully utilized by current compiler technology. Ranging from a few percent to over 95 percent of the *LOAD/STORE* instructions belongs to *LOAD/STORE-UPDATE* category.

Benchmark	Percentage of instructions Executed					
	Total <i>LOAD</i>	<i>LOAD-UPDATE</i>	Total <i>STORE</i>	<i>STORE-UPDATE</i>	Total <i>LOAD/STORE</i>	Total <i>LOAD/STORE-UPDATE</i>
compress	21.5	0.0	9.2	0.3	30.7	0.3
espresso	22.0	11.1	3.9	1.3	25.9	12.4
li	25.4	0.8	15.2	2.3	40.6	3.1
nasa7	42.8	42.0	1.7	1.4	44.5	43.4
spice2g6	18.3	1.4	9.9	1.1	28.2	2.5
su2cor	26.4	8.5	14.1	6.1	40.5	14.6
tomcatv	29.6	18.3	11.1	10.1	40.7	28.4
wave5	26.5	1.4	9.7	2.2	36.2	3.6

Table 6.2: Percentages of *LOAD/STORE-UPDATE*s in SPEC92 Benchmark Suite

Numerous experiments on various caching models were simulated using the collected SPEC92 traces as the input.

- The simulated cache size ranged from 8K bytes to 32K bytes.
- The simulated line size ranged from 8 bytes to 64 bytes.
- The simulated set associativities ranged from 1 to 4.
- The Time for a demand fetch request to transfer a cache line between the first level cache and the second level cache memory was equal to $(C1 + C2 \times (Line_Size - 1))$, where $C1$ is the delay time for the first word to arrive

after a cache miss, and $C2$ is a parameter that indicates the bus bandwidth between the first level and the second level cache.

- The time for a demand fetch request to finish was assumed to be the same as that for a prefetch request to finish for a given cache line size.
- The second level cache/memory was assumed to be infinitely large, and there was no cache miss in the second level cache.
- When there existed a memory request R_a trying to kill another request R_b that was currently being served, there would be one cycle time delay before the new request R_a could start.
- The size of the prefetch queue was assumed to be eight entries.
- Each instruction was assumed to be executed in one cycle and no superscalar architecture was simulated.
- One cycle access time was needed for a cache hit.
- Seven cache prefetch models were simulated :
 1. Data cache without any prefetching.
 2. Data cache with prefetch-on-miss.
 3. Data cache with the combined IAP scheme – using the IAP scheme for *LOAD/STORE-UPDATE* instructions and the default prefetch-on-miss for non-*LOAD/STORE-UPDATE* instructions.
 4. Data cache as mentioned in 3, but with a mixed replacement policy consisted of LRU and IZ.
 5. Data cache as mentioned in 2, but with priority pre-updating and victim cache to tackle with the prefetched lines. The size of the victim cache is assumed to be 4 entries, with one entry equal to the size of one line in data cache.

6. Data cache as mentioned in 3, but with priority pre-updating and victim cache to tackle with the prefetched lines. The size of the victim cache is assumed to be 4 entries, with one entry equal to the size of one line in data cache.
 7. Data cache as mentioned in 3, but with a prefetch cache added to hold the prefetched lines by the IAP scheme.
- All the enhancements that mentioned in Section 2 were implemented in the IAP scheme simulated. And only *LOAD/STORE-UPDATE* instructions using *index-displacement* addressing mode was handled in the simulated IAP. While for any *LOAD/STORE-UPDATE* instructions using *index-based register* addressing mode, the IAP scheme did not issue any prefetch request.

In the past, cache design had been frequently taken as a back seat to CPU design: the cache subsystem was often designed to fit the constraints imposed by the CPU implementation. The execution time of a program fundamentally depended on how well the two units worked together to execute instructions. The execution time was most effectively minimized when the realities of cache design influenced the CPU design and vice versa. Furthermore, caches had traditionally been evaluated solely on the basis of hit (or miss) ratios - a metric that can often be deceiving.

In order to reflect the actual performance of the algorithms proposed, two main metrics were used here to evaluate the performances of different schemes.

The first performance metric is *cycle per instruction* due to *memory* (data cache) misses. This performance parameter, MCPI, measures the average additional processor stall time due to the first level cache misses. It also helps to show the degree of degradation of CPU performance due to the data cache misses in terms of memory cycle stalls per instruction. Generally, it can be calculated with the execution CPI and baseline CPI by the equations:

$$MCPI = CPI_{execution} - CPI_{baseline}$$

$$\text{where } CPI_{\text{execution}} = \frac{\text{total_number_of_cycles_executed}}{\text{total_number_of_instructions}}$$

$$\text{and } CPI_{\text{baseline}} = \frac{\text{total_number_of_cycles_executed_for_no_cache_miss}}{\text{total_number_of_instructions}}$$

This is a better measurement parameter than the cache hit (or miss) ratio, as the penalty of a cache miss depends on the cache line size. Moreover, partial cache hit (or miss) situation, which is the situation that when a block i is being prefetched, the cache line i is actually referenced, always occurs. Partial cache hit arises due to the limited bandwidth between the first level cache and the second level cache (or memory). When a partial hit occurs, the data prefetching will be allowed to finished and the requested data are sent to the CPU. Thus the penalty of partial cache hits is not a constant, and it ranges from 1 to (*maximum cache miss penalty* - 1) (i.e. $(C1 + C2 \times (Block_Size - 1) - 1)$). Under this situation, the cache hit ratio is much difficult to reflect the actual cache performance, as some part of the data is overlapped with the processor execution while other part of the data fetching time is visible to the processor.

Since we assume the processor can execute each instruction in one cycle and there is an ideal instruction cache in the system, one may intuitively deduce that the fact that the memory bus between the processor and the first level cache is only 32-bit (4-byte) wide. It also means that at most 4 bytes can be transferred between the processor and the first level cache in one cycle. If the data needed to be loaded or stored by an instruction is longer than 4 bytes the instruction can only be finished after all required data are loaded in the processor and the execution time is sure to be longer than one cycle even when there is no cache miss. For example, a *LOAD/STORE-DOUBLEWORD* instruction will be executed for two cycles even when the data needed is found in the cache. As a result, the baseline CPI will probably be greater than one and this effect is more significant in the double precision floating point benchmarks such as *nasa7* and *tomcatv*, in which most of the data are double words of 8 bytes long. Table 6.3 shows the

baseline CPIs found for the eight benchmarks programs used in our simulations.

Benchmark	baseline CPI
compress	1.032
espresso	1.011
li	1.075
nasa7	1.444
spice2g6	1.084
su2cor	1.286
tomcatv	1.407
wave5	1.124

Table 6.3: Baseline CPIs of SPEC92 Benchmark Suite

Another one was the additional processor stall time due to the first level cache misses.

The percentage of delay time reduction over no prefetch was defined as

$$\%DelayTimeReduction = \frac{MemoryDelayTime_{NoPrefetchCache} - MemoryDelayTime_{PrefetchCache}}{MemoryDelayTime_{NoPrefetchCache}}$$

The metric can be used to show the extent of memory stall time reduces due to data cache miss with respect to an elementary cache using no prefetch scheme.

6.2 Simulation Results

In the following sections, the experimental results are presented to show the benefits of the mixed replacement scheme in IAP, the impact of PPUVC in cache performance improvement and also the effect of cache performance with the use of a small prefetch cache. The architecture with the elementary caching model using no prefetch scheme is compared with the same architecture augmented by each of these schemes.

6.2.1 General Results

- All the replacement and placement schemes that deal with IAP lines seemed to have no effect on the benchmark compress. It can be found out in Table 6.2, only 0.3% of the total instructions (less than 1% of *LOAD/STORE* instructions) is of the type *LOAD/STORE-UPDATE*. As the prefetching actions of the IAP scheme is triggered by the *LOAD/STORE-UPDATE* instructions, only a few prefetched requests will be generated for the IAP scheme and their effects will be negligible. Therefore, IZ replacement policy, PPUVC and also the prefetch cache, which work upon IAP lines, had insignificant effect on the cache performance. Moreover, the combined IAP scheme would revert back to a simple prefetch-on-miss scheme and the two schemes suffered a slight performance degradation with respect to the no prefetch cache, which is probably due to the lack of constant stride references in the program (as reflected by the lack of *LOAD/STORE-UPDATE* instructions).
- Prefetch-on-miss, the traditional hardware prefetching scheme, generally has some improvements over most of caching models tested except for the benchmark compress. The combined IAP scheme showed performance improvement over all caching models for almost all benchmarks used (except compress).

Varying Cache Size

Figures A.1, A.2, A.3 and A.4 in Appendix A shows the simulation results for the eight benchmark programs using cache size varies from 8K to 32K bytes. All experiments were done with caching models of 32-byte line size and 4-way associativity.

As expected, one can find that the MCPI decreased as the cache size increased. However, for some benchmarks, such as *su2cor* and *tomcatv*, the pure IAP schemes

showed little improvement when the cache size was small, i.e., 8K bytes, but exhibited substantial improvement when the cache was increased to 16K and 32K bytes. For *tomcatv*, when the cache size is very small (8K bytes), the pure IAP scheme actually degraded the performance instead of improving it (can be observed in Figure A.1 (g)). This is probably due to the small cache size and the aggressive cache prefetching scheme. Even though the prefetching can be very accurate, those accurately prefetched data will displace each other away from the data cache before they have the chance to be used. However, as the cache size increased from 8K bytes to 16K bytes, this cache conflict problem was minimized and the IAP scheme started to have substantial cache performance improvement.

However, in the three proposed schemes – IZ, PPUVC and prefetch cache, the performance improvement was generally more significant for small cache size in *su2cor*, *tomcatv* and *wave5* (Figures A.1, A.2, A.3 and A.4). When the cache size increased, the performance improvement was comparable to the pure IAP scheme.

The significant improvements for small cache size in IZ and PPUVC schemes are due to the careful selection of lines to be replaced. By using the specific displacing criteria in these two schemes, lines, which are likely to be useless in the future, are displaced in cases of conflicts. When the cache size was small, conflict misses occurred more frequently. If a replacement algorithm can accurately predict which line should be discarded, then many future memory accesses can be eliminated. IZ scheme makes use of the reference-once property of IAP lines for deciding the replacement criteria, and it can accurately predict which line should be discarded for most of the benchmark programs. For the PPUVC, it helps to shorten the life time of possibly mispredicted lines as well as maintaining the properties of locality.

Varying Cache Line Size

Figures A.5, A.6, A.7 and A.8 in Appendix A shows the simulation results for the eight benchmark programs using different prefetching schemes. The experiments were done with caching models of 16K-byte cache size, 4-way associativity and

varying cache line size from 4 to 32 bytes.

The MCPI curves for IAP schemes generally had U-like shapes. That is, the MCPIs of the programs first declined from small line size to the optimal line size. Then, the directions of the curve reversed and the MCPIs kept rising after the optimal line sizes. These observations are common to be found in most of the cache simulations. As the line size increases, more data will be fetched one time and the spatial locality between these data may be beneficial to processor execution. Moreover, it is also more economical on average to fetch a larger line one time than to fetch a smaller line several times separately because the time to fetch a line from memory is equal to $C_1 + C_2 \times (\text{line_size} - 1)$. As the line size increases from the smallest size to the optimal one, these effects are dominant and the MCPI continues to drop in this range. However, as the line size keeps increasing after that point, using larger cache line size for sequential prefetching seems to be not so effective. As the line size further increases, greater portions in the lines will contain data that will not be referenced in the near future and the lines will be kicked out without these data being touched. Moreover, increasing the cache line size elongates the time for fetching a line from the memory. This means the CPU must wait longer for the same amount of data needed (for example, the CPU is stalled longer for a 4-byte datum in a 64-byte line than a 4-byte datum in a 32-byte line). At the same time, this also increases the risks of killing the prefetches by demand fetches caused by real cache misses. Finally, the larger line size reduces the total number of distinct lines that can be put into the data cache and increases the conflicts between lines in the cache which may cause some useful data to be kicked out before it is referenced by the CPU. When these adverse effects of larger line size outweigh the benefits brought, increasing line size will mean higher miss rate, more processor idle time and lower CPU performance. These explain why the MCPI curves rose after passing the optimal line sizes.

For some programs (compress, espresso, spice2g6 and su2cor), the MCPI curves showed that the caches worked better when the line size was small (4 bytes). It is probably because the data of consecutive references are separated

far apart and do not reside in the same line. As a result, only small portions of the large lines fetched from the secondary memory will be referenced in the near future and the locality introduced by the large line size does not help much. On the other hand, with the smaller line size, the cache with the same size can contain more lines and it gives more flexibility for the IAP schemes to do accurate prefetching. As a conclusion, smaller cache line size is preferred in these situations. This also agrees with what Lee [Lee87] found about smaller line sizes for data cache.

However, for espresso (Figures A.5, A.6, A.7 and A.8 (b)), the increasing MCPI curves of the schemes worked on IAP lines turn around and began to drop when the line size increases from 32 bytes to 64 bytes (similar phenomena were also observed for the cache only and prefetch-on-miss curves when the line size increased from 16 bytes to 32 bytes). Although the explanation for this phenomenon is not very clear, we suspect that this is related to the data accesses with large stride values of 32 to 64 bytes in the program. From 4 bytes to 32 bytes cache line size, the number of lines that can be stored in the cache is reduced by half each time when the line size is doubled. However, if the stride size of the data accesses is large, a small increase in the line size does not capture more useful data. Consequently, increasing the cache line size below 32 bytes line size only causes cache pollution and results in poor cache performance. When the cache line size was increased from 32 bytes to 64 bytes, sequential data prefetching using large line size starts to have some effect and the cache performance is improved.

It seems that the IAP lines in the program *nasa7* had high temporal locality. For the PPU scheme, it could not obtain significant improvement in prefetch-on-miss only cache where only prefetch-on-miss lines are involved. However, the PPU could obtain a quite significant decrease in MCPI, especially when the line size was small, in cache with IAP scheme. IAP lines in *nasa7* has high temporal locality, which could be further confirmed by its performance degradation in IZ scheme when comparing with the combined IAP (Figure A.5 (d)). In IZ scheme, an IAP line would be kicked out very soon after it has been referenced, due to the

underlying assumption of low temporal locality in IZ scheme. However, it seems that this assumption is not true in *nasa7*.

Varying Cache Set Associative

Figures A.9, A.11, A.10 and A.11 in Appendix A shows the effect of increasing the cache set associativity. As it is expected, from the set associativity of 1 to 2, the performance was generally improved (except the benchmark *su2cor*). With a one-way associativity (direct mapping) cache, every line could be placed at only one position. If it happens that two sequences of data accesses, for example, two arrays inside the same loop, are mapped to similar sets, they will continually displace each other's data line in the cache, although the displaced line may contain data that will be referenced in the near future. As a consequence, miss rate will be increased and the cache performance will be degraded. It accounts for the large improvement from one-way to two-way set associative cache. This effect was more obvious for the benchmark *nasa7* (Figures A.9, A.11, A.10 and A.11 (d)). For the scheme PPUVC in the prefetch-on-miss-only cache model, the performance difference between direct-mapped and 2-way set associative was less significant than other schemes which involve IAP lines. For the direct-mapped case, IZ was actually reverted to combined IAP scheme. However, for prefetch cache and PPUVC in IAP scheme, the program *nasa7* had a lower MCPI in direct-mapped situation than in others. The reason is that IAP has high accuracy of prefetching, and this aggressive prefetching prefetches data into the cache before their actual references. From Figure 6.2, we can find that almost all (over 90%) of the data references belong to the LOAD/STORE-UPDATE (constant stride) type and are mainly chains of array or pointer references. With this large amount of constant stride references, the chance of conflicts induced by the address mapping will probably be very high. Due to the relative high conflict misses in direct-mapped cache, some of these useful IAP lines were trashed out before their references. Therefore, there were wastes of clock cycles, as these lines had to fetch into the cache again in the future. With the use of victim cache, to hold this lines, in IAP

scheme, it could extend the life time of these lines in the *cache* and avoid many potential memory accesses. Similar performance improvement could be observed in prefetch cache for *nasa7*, in which the fully-associative prefetch cache provided a place to hold these IAP lines before their references. In both cases, the IAP lines could stay somewhere near the cache before their references, and fetching from these buffer space was much faster than fetching the data from lower level memory.

However, the cache performance was more or less the same for 2-way and 4-way set associativity. Although increasing the associativity gives more flexibility for cache line placement, at the same time, the number of sets in the cache will be halved and the performance will be less dependent on the associativity in these situations.

6.3 Simulation Results of IZ Replacement Policy

From the figures in Appendix B, most of the experiments showed some improvements when the IAP scheme together with the new replacement policy IZ was implemented. More importantly, the implementation of the IAP and IZ in current architecture is not difficult, as no processor architecture changes are necessary, but only some simple additional on-chip cache hardware is required.

- The effects of the IZ scheme could be classified into two main streams. The first group can achieve significant performance improvement, the second group shows slightly improvement or nearly coincident with the original scheme prefetch-on-miss scheme.

For some of the benchmarks such as *spice2g6*, *espresso*, *li* and *nasa7*, the default prefetching scheme seemed to have no impact to the cache performance. The curve for the IZ scheme almost overlapped with each other. However, for *su2cor*, *tomcatv* and *wave5*, the IZ scheme helped to reduce

the memory stall time further. This can be explained as follows.

In the *spice2g6*, *espresso*, *li* and *nasa7* programs, the reason may either [1] the spatial and temporal localities are weak in these programs, or [2] most of data references with strong temporal locality are referenced by *LOAD/STORE-UPDATE* instructions. It seems that the former one is more likely, since performance degradation will surely be observed if case [2] is true. As a result, *IZ* just had no further significant improvement when comparing with combined *IAP* in these programs. On the other hand, for *su2cor*, *tomcatv* and *wave5*, a significant portion of the data references with strong localities are referenced by non- *LOAD/STORE-UPDATE* instructions and the numerous amount of prefetched lines generate conflicts and misses in the cache. The *IZ* scheme helps to resolve these conflicts and improve the cache performance.

- Referring to Figures B.1 to B.9. The memory stall time reduction that could be achieved by the *IZ* scheme ranged from about a few percent to over 90%, with an average of about 50%. These figures really show the potentials of the *IZ* schemes. This kind of improvement in cache performance over a wide range of benchmark programs (instead of some small routines or kernels such as Livermore Kernels) is really substantial. Furthermore, this performance improvement can be obtained by just modifying the on-chip cache hardware and no change to the processor architecture (such as the instruction set) is required.

6.3.1 Analysis To *IZ* Cache Line Replacement Policy

With reference to the figures in Appendix B, some factors affecting the performance of *IZ* could be observed.

- **Instruction Mix of Benchmark Programs**

When a benchmark program, such as *compress*, contains only few *LOAD/STORE-UPDATE* instructions, there was no improvement brought by the *IZ* policy.

This is up to our expectation, as IZ policy is mainly applicable to the lines prefetched by the IAP scheme. When there are only few of IAP lines, the effect brought by IZ will naturally be small. On the other hand, for benchmark programs with lots of *LOAD/STORE-UPDATE* instructions, IZ contributed significant improvement on the cache performance when the cache was not large enough to hold all the demand fetched and prefetched lines. This can be reflected by the simulation results of *su2cor* and *tomcatv*.

As a result, the number of *LOAD/STORE-UPDATE* instructions affects the performance of cache with IZ policy.

- **Cache Size**

With reference to the two benchmark programs, *su2cor* and *tomcatv*, IZ had obtained the greatest improvement among the eight programs, it could be easily observed that there was larger improvement on cache performance for smaller cache size than large cache size. This scenario could be easily observed in the benchmark program *wave5*. The reason is that for larger cache size, the cache has enough space to hold the lines fetched or prefetched from the main memory. Thus the replacement of lines occurs less frequently, the effect of IZ cannot be observed then. However, as small cache size is not enough to hold all the data that needed, and thus replacement of lines occurs frequently. As a result, IZ can improve the utilization of cache space by displacing the referenced IAP lines out instead of other lines that may possess strong spatial or temporal localities.

Generally, the number of prefetched lines actually referenced increased for cache with IZ scheme together with the combined IAP. This can be explained by the fact that some prefetched lines may have to wait a long time before their actual references, when there is not enough space in cache, some of these lines have to be displaced out by LRU scheme before referencing. It will certainly be a waste of cycle time, as these lines have high probabilities for being referenced later and they may have to load into cache again. However,

by using IZ scheme, these lines could stay in the cache for a longer time. As those referenced IAP lines will be displaced instead of other prefetched lines which have higher potential to be referenced in the future.

IZ did show its worthiness on cache performance for small cache size and when the cache could not accommodate all the lines that contain data to be referenced.

6.4 Simulation Results for Priority Pre-Updating with Victim Cache

For any data reference, sequential checking of data cache and victim cache was performed. The data cache will be checked first to see if the requested data is there. If the data is not found in data cache, then a miss occurs, the cache controller will be informed to search the victim cache. If the data is found in the victim cache, then the processor use one more cycle to fetch the data from the victim cache. Otherwise, lower level memory should be involved.

Referring to the figures in Appendix C, the performance improvement in the eight benchmark programs can be classified into three categories:

1. the performance improvement was large (over 20%),
2. the performance improvement was slight to moderate (1% to 20%),
3. the improvement was nil or caused a minor degradation.

6.4.1 PPUVC in Cache with IAP Scheme

The metric on comparison were based on the comparison with combined IAP. The delay time reduction figures that quoted, was based on a 16K bytes cache size, 32 bytes line size and 4-way set associative.

IAP scheme has high accuracy in performing prefetching, the number of mis-predicted lines is few. Due to its accuracy, cache misses are highly reduced. As a result, prefetch-on-miss lines are few, Figure 6.3 gives a comparison of number of

prefetch-on-miss lines in cache with IAP and prefetch-on-miss only cache. Moreover, the references to prefetch-on-miss lines is few in the cache model with IAP scheme. Figure 6.4 shows the actual number of prefetch-on-miss lines referenced when comparing with the total number of prefetched lines. These explain the insignificant improvement for PPU in IAP schemes.

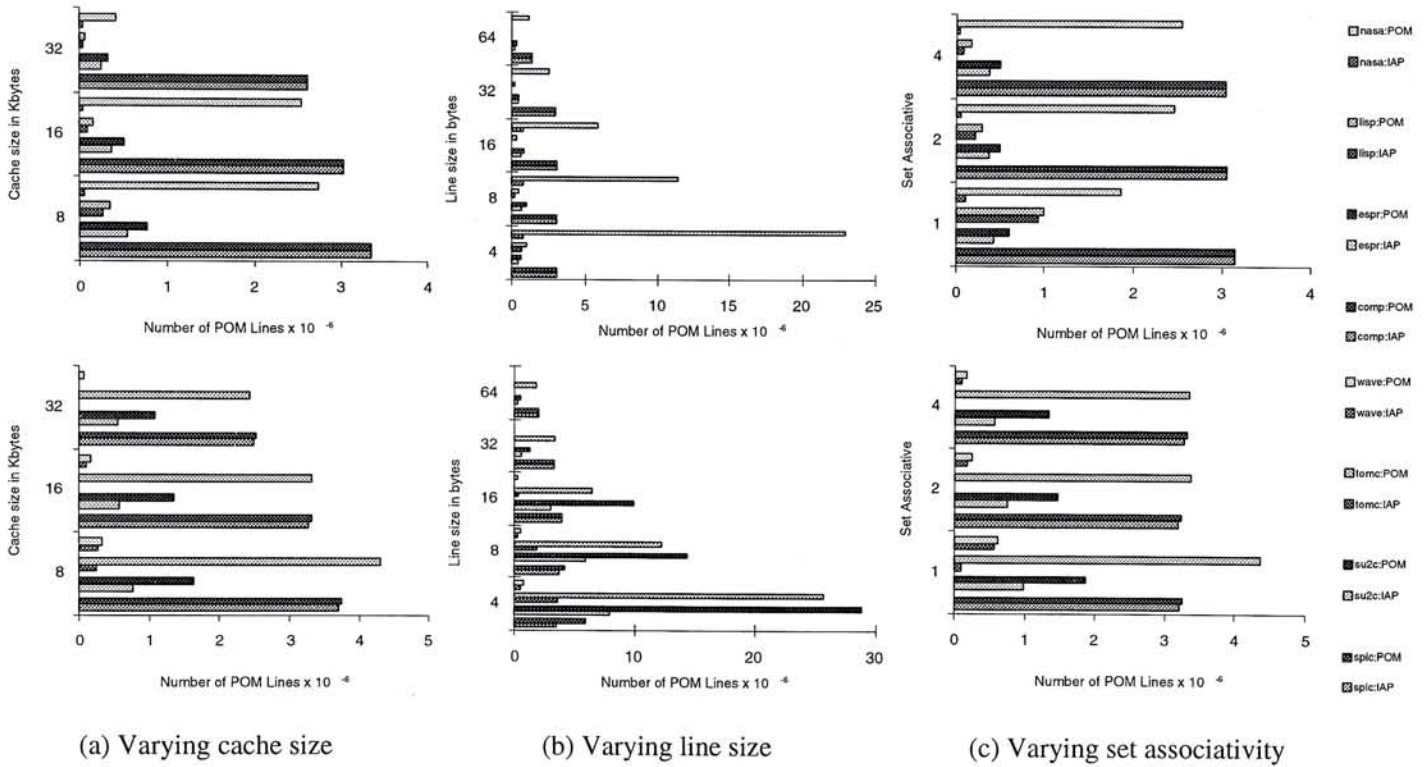


Figure 6.3: Comparison of number of prefetch-on-miss lines in IAP cache and prefetch-on-miss-only cache

The programs can be divided into three groups, the first group consists of su2cor and tomcatv, which could achieve over 20% of memory delay time reduction for the default cache parameters ².

The second group consists of espresso, li and nasa7, in which each could obtain few percent of performance improvement.

The third group consists of compress and wave5, as a little performance degradation was observed. The classification is based on the default cache parameters. Though wave5 could obtain quite large performance improvement when the cache size was small, it is classified as in the third group for the sake of consistency.

²16K bytes cache size, 32 bytes line size and 4-way set-associative

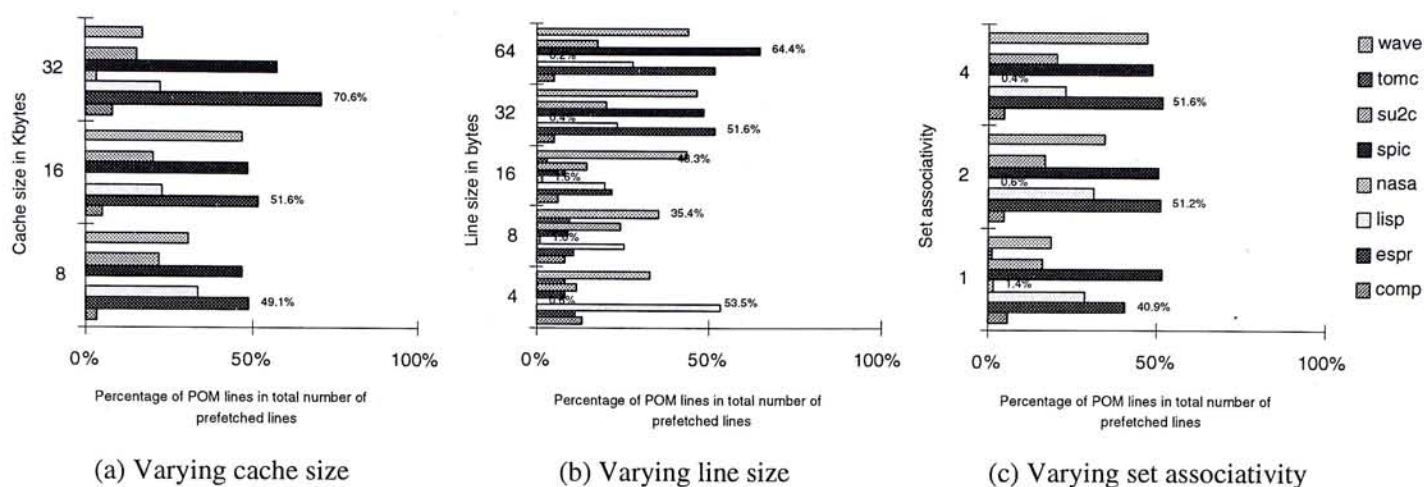


Figure 6.4: Percentage of prefetch-on-miss lines referenced in total number of prefetched lines

Actually, the effect of PPUVC for IAP lines is not great. The reason is that IAP scheme has very high accuracy of prefetching, and thus mispredicted lines are few. On the other hand, it may degrade the performance when the time between two successive references to IAP lines is long. Thus these IAP lines, which are waiting for being referenced, may be determined as *useless* and have their priorities pre-updating by the mechanism. They bear the risks to be discarded before their actual reference. This situation will cause more potential cache misses and increases the number of memory accesses.

For the first group of programs, the performance improvement for the two programs was actually less than that of PPUVC in prefetch-on-miss-only cache when the cache size was small. At which PPUVC in IAP could obtain only 28% of memory delay time reduction, while PPUVC in prefetch-on-miss-only cache could obtain 35%. The reason maybe possibly be due to relative small number of prefetch-on-miss lines in the IAP cache, and thus pre-updating of mispredicted prefetched lines has little effect.

6.4.2 PPUVC in prefetch-on-miss Cache

From the simulation results, one can easily see that PPU combined with victim cache resulted in better cache performance on average. All programs except

spice2g6, in which with two curves nearly coincident, could achieve a further reduction in memory delay when compared with prefetch-on-miss-only cache.

For the program compress, it had a little performance degradation when comparing with no prefetch cache, however, the degradation was small in this case when comparing with IZ. As compress possesses low spatial locality property and there are few constant stride references, and therefore, the prefetched data by prefetch-on-miss are most likely to be useless. These caused a waste of clock cycle as well as pollution of the cache. For the PPU scheme, unlike IZ which only work on IAP lines, it helps to shorten the life time of mispredicted lines, and thus a better result could be observed when comparing with IZ.

The programs can be divided into three groups. The first group could achieve up to 49% of memory delay time reduction, which was about 46% further reduction than prefetch-on-miss. This group of programs consists of su2cor and tomcatv. For the default parameters: 16K bytes cache size, 32 bytes line size and 4-way set associative, su2cor could achieve 35.6% of Memory delay time reduction in PPUVC scheme, while prefetch-on-miss could obtain only 2.4%. Tomcatv could achieve 43.0% of memory delay time reduction while prefetch-on-miss only obtained 15.3%. i.e., there were about 33% and 28% further reduction in memory delay in su2cor and tomcatv respectively.

The second group consists of espresso, li and wave5, in which they could achieve 1% to 3% of memory delay time reduction. Note that when implementing PPUVC in IAP scheme, wave5 had a little performance degradation (about 1%) for the cache with default parameters.

The third group consists of compress, nasa7 and spice2g6, in which the curves of nasa7 and spice2g6 were nearly coincident with that of prefetch-on-miss. However, compress showed a little performance degradation when comparing with no prefetch cache.

The Effect of On-chip Cache Size on Victim Cache Performance

When the cache size is small, the number of available lines in cache is few. As a result, thrashing of cache lines occurs frequently. Many prefetched lines are displaced from cache before their references. Therefore, memory cycles have to be spent to fetch these lines back into cache during their actual references. With the addition of priority pre-updating and victim cache, those prefetched lines that have not been referenced for a long time after their fetching will be discarded first. The miss penalty for erroneously displacing useful lines is reduced by using the small victim cache. This situation easily follows from the fact that some benchmark programs could obtain significant performance improvement when the cache size was small, while less promising results when the cache size became larger. The programs that highly illustrated this including tomcatv and wave5. Tomcatv could achieve a 50% further reduction in 8K bytes cache, when it is compared with no improvement when the cache size increased to 32K bytes. Wave5 obtained 27% further reduction in memory delay comparing with no improvement for 32K bytes cache. Figures C.10, C.11 and C.12 show the memory delay time reduction of victim cache by varying cache size.

The Effect of Line Size of On-chip Cache on Victim Cache Performance

When line size is large, the number of lines in the cache is reduced and thrashing of lines occurs more frequently. With PPUVC, the penalty due to discarding useful data is minimized. Victim cache is able to hold few lines that are likely useful in the future. At the time these lines are actually needed, one cycle is needed to fetch them to use by the processor comparing with the long miss penalty when fetching from lower level memories. The program su2cor obtained 47% further memory delay time reduction with 64 bytes line size while there is no significant improvement for 4 bytes line size. With line size 4 bytes, tomcatv had no significant further improvement compared with prefetch-on-miss-only cache, however, the deviation between memory delay time reduction in of prefetch-on-miss-only

cache and the cache with PPUVC became greater. In which tomcatv had a 59% of further reduction in memory delay for 64 bytes line size. The same situation was observed in wave5, which could obtain a 14% of further memory delay time reduction. Figures C.13 and C.14 shows the results of these two benchmarks.

The Effect of Set Associativity of On-chip Cache on Victim Cache Performance

Comparing with prefetch-on-miss, most programs could obtain better performance improvement with victim cache when the degree of associativity was low. When the associativity was low, say, direct-mapped, thrashing of lines occurs frequently. For direct-mapped cache, memory blocks would map to the same cache line due to the mapping algorithm such as bit selection. This may result in the situation that numerous blocks compete for the same cache line, while many cache lines are remained unused. Consequently, utilization of cache is low, and also many data are discarded before being referenced. The use of victim cache helps alleviate this problem. Figures C.16 to C.18 show the results of varying set associativity. Programs such as espresso, lisp, nasa7, su2cor, tomcatv and wave5 could achieve a further memory delay reduction ranged from 8% to 46%. While compress and spice2g6 had similar performance as the original one in direct-mapped cache.

The results for victim cache show that the addition of a small amount of hardware can dramatically improve the system performance. It is difficult, mostly impossible, to derive algorithms for caches that can optimize the system performance of every programs and applications. An algorithm that can satisfy most of the programs seems to be more applicable and practical.

6.5 Prefetch Cache

The default size for the prefetch cache was 1K bytes. Simulations were done on cache model with prefetch cache sizes ranged from 256 bytes to 4K bytes and results are shown in Figure 6.5. Prefetch cache of size 1K bytes is a reasonable

choice, though larger size seems to have better performance for certain benchmarks. However, larger size means larger delay. Different set associativities were also performed in the prefetch cache, the associativity ranged from direct-mapped to fully associative.

For any data references, parallel checking of prefetch cache and data cache was performed. It is assumed that there was no extra cycle delay for checking the prefetch cache. We chose 1K bytes prefetch cache size, 32 bytes line size, 4-way set associative and fully-associative, as the default parameters.

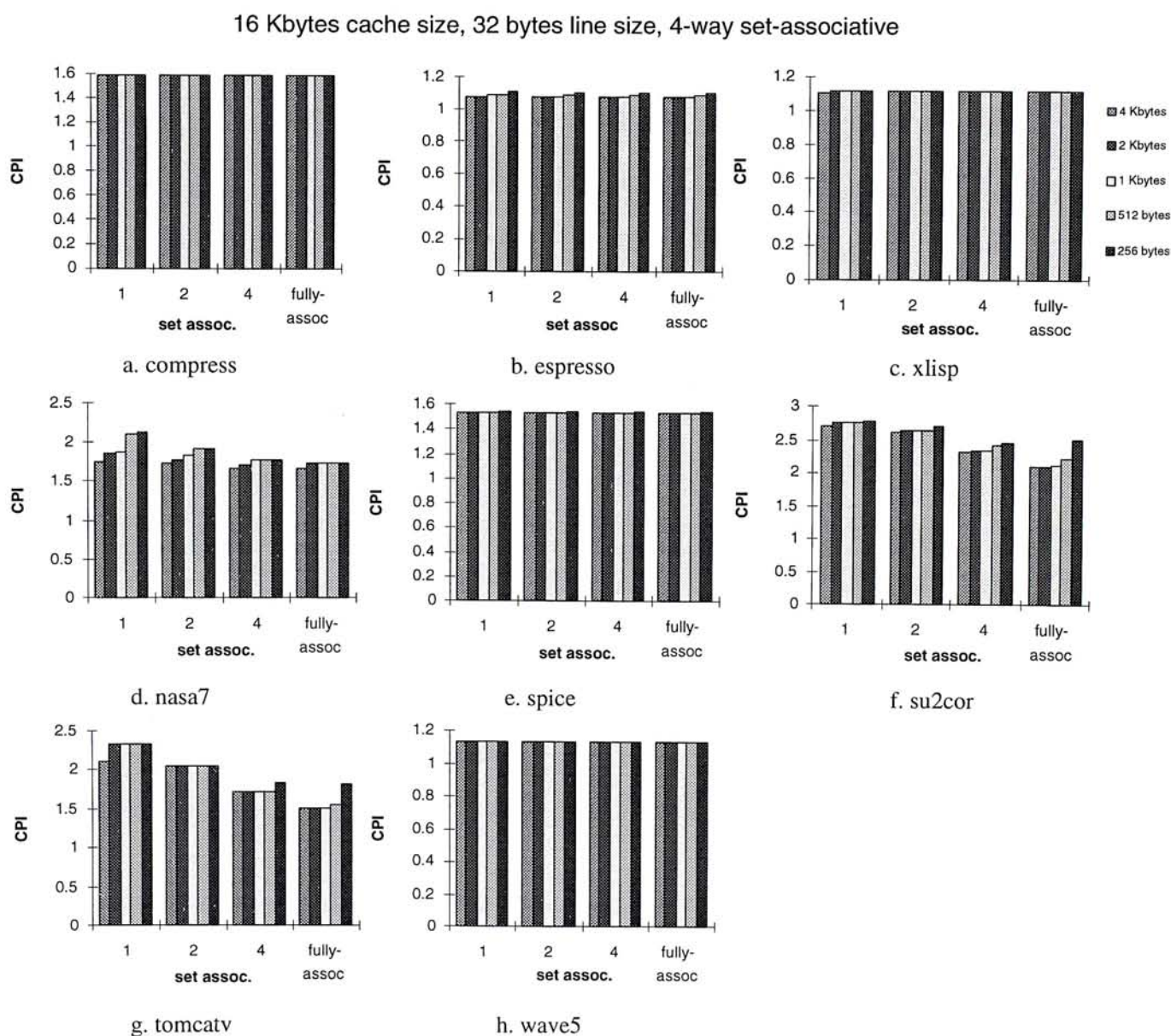


Figure 6.5: The effect of Prefetch Cache size on cache performance

We examined further the performance curves by dividing the eight benchmarks

into four groups: [1] performed extremely well, [2] performance improvement was moderate, [3] yielded a slight improvement to the performance, and [4] contribution to the reduction in data access penalty was nil.

Prefetch Cache with FIFO Replacement Policy

The following observations were obtained in the fully-associative prefetch cache. All programs except nasa7 obtained better performance than the basic IAP, and all obtained better results than prefetch-on-miss-only cache. The memory delay time reduction was up to 99%³ as found in tomcatv.

The first group is formed by su2cor and tomcatv with memory delay time reduction ranged from 51% to 99% in fully-associative prefetch cache. There was 44% to 55% further reduction than basic IAP. In Figure D.1 the delay time reduction of these two programs were clearly shown.

The second group contains only wave5, which achieved 81% memory delay reduction over no prefetch, and was 21% further performance improvement than basic IAP. Figure D.2 show the simulation results of wave5 in terms of its the delay time reduction.

The third group includes espresso, li and spice2g6. The memory delay time reduction was 33% up to 83%, which was 1% to 4% further reduction. Refer to Figure D.3 for the results of this group of programs.

The fourth group contains only compress and nasa7, which had got no performance improvement, and nasa7 had a slightly performance degradation when comparing with basic IAP. Figure D.4 shows the results.

Prefetch Cache with LRU Replacement Policy

In the fully-associative prefetch cache, the programs espresso, su2cor, tomcatv and wave5 showed better system improvement. The performance improvement of compress, li and spice2g6 was slight. Besides, nasa7 had a slightly performance

³All percentages quoted are obtained by using the cache parameters: 16K bytes cache size, 32 bytes line size and 4-way set associative

degradation. In fully-associative prefetch cache, all benchmark programs showed same pattern and similar degree of performance improvement as that of FIFO prefetch cache. Refer to Figures D.13 to D.16 for the results of the four groups of programs in LRU prefetch cache.

Referring to Table 6.2, only 0.3% of the instructions executed belonged to *LOAD/STORE-UPDATE* in compress, thus IAP lines were few in this benchmark program. As a result, the influence and usage of prefetch cache is insignificant. However, a different situation was observed in *nasa7*, which contained up to 43.4% of instructions executed belonged to *LOAD/STORE-UPDATE*, resulting in numerous number of IAP lines. The prefetch cache may not be large enough to hold all the prefetched IAP lines, and some unreferenced IAP lines are thrashed by new prefetched IAP lines. The thrashing misses caused the degradation of performance in *nasa7*. As a result, *nasa7* got a performance drop in both prefetch cache with FIFO and LRU replacement policy.

Prefetch Cache with IZ Replacement Policy

Using IZ replacement policy in fully-associative prefetch cache showed degradation in the performance. However, the degradation was of a much lesser extent in the 4-way prefetch cache. In some programs, IZ in 4-way prefetch cache could even achieve similar performance as that of FIFO.

The Effect of Set Associative on Prefetch Cache Performance

Basically, prefetch cache with 4-way set-associative exhibited similar performance as that of fully-associative prefetch cache. However, in the programs *su2cor* and *tomcatv*, the performance improvement was of a lesser extent. Programs such as *tomcatv* could only achieve a maximum of 70% and 81% memory delay time reduction when using LRU and FIFO replacement policy respectively. Another benchmark *su2cor* also had smaller performance improvement. The smaller promising performance improvement in 4-way set-associative prefetch cache maybe due to the reason that IAP lines in *tomcatv* and *su2cor* were mapped to some particular

sets, leaving other sets unused and thus thrashing of lines in the prefetch cache occurs. Utilization of prefetch cache is thus lower than that of fully-associative prefetch cache, in which the thrashing misses is minimized or eliminated.

Comparison of the Three Simulated Replacement Policies

In general, LRU replacement policy is a better choice for data cache when ignoring its costs [Smi82]. However, among all replacement policies that simulated, FIFO seems to be the most suitable one in prefetch cache. Figure 6.6 illustrates the reason why there is performance difference between LRU and FIFO replacement policies.

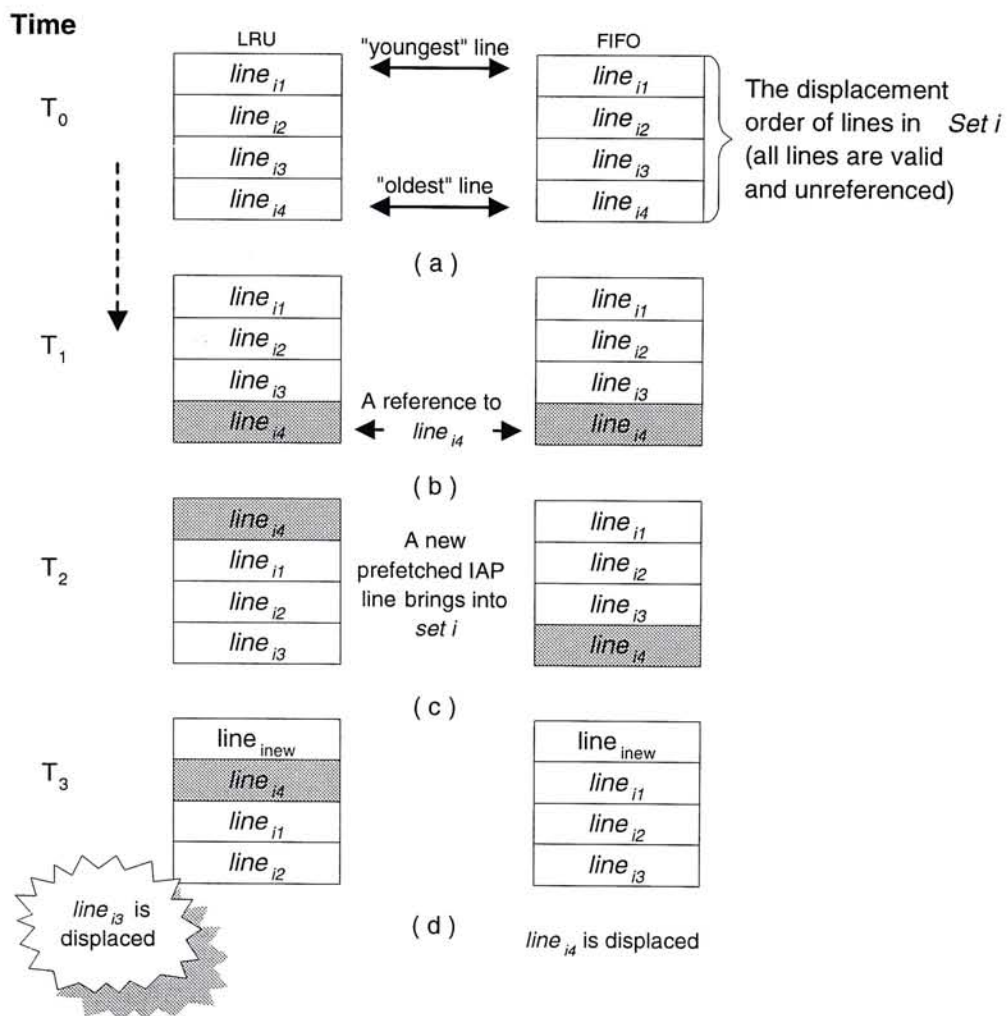


Figure 6.6: An illustration of the performance difference between LRU and FIFO

In Figure 6.6 ⁴, we consider a set i which is full. All cache lines ($line_{i1}$, $line_{i2}$,

⁴Referenced lines are shaded

$line_{i3}$, $line_{i4}$) in set i are valid and unreferenced originally at T_0 . Please be reminded that the cache lines are drawn according to the order of their displacement (i.e., their priorities), but not representing their actual placement in cache set i . That is, the one with lowest priority will place at the bottom, while the one with highest priority at the top. If there is a reference to $line_{i4}$, which is the one with lowest priority, at time T_1 in Figure 6.6b. Then $line_{i4}$ will become the highest priority one in the set when following LRU replacement policy. However, it is obvious that its priority will be the same when FIFO replacement policy is used. At T_2 , there is a new prefetched line brought into set i , thus a line has to be discarded to accommodate the new line. As a result, we can see from Figure 6.6d that $line_{i3}$ is discarded if follows LRU replacement policy, but $line_{i4}$ is displaced when follows FIFO policy. As mentioned before, IAP scheme poses very high accuracy, and thus $line_{i3}$ is very likely to be referenced in the future. Besides, most IAP lines pose referenced-once properties, $line_{i4}$ is most possibly useless in the future. Thus discarding $line_{i4}$ instead of $line_{i3}$ may improve the system performance. These illustrate why LRU replacement policy performed worse than FIFO replacement policy.

Using IZ replacement policy prefetch cache yielded the worst performance improvement. When applying to a data cache which mixed with IAP lines and normal cache lines, IZ replacement policy yielded good results. However, when comparing with other simulated replacement policies, it is not strange for IZ to obtain poor performance improvement in prefetch cache which contains only IAP lines. IAP lines pose reference-once properties, and thus once they are referenced, they are considered useless and can be discarded. In prefetch cache, all are IAP lines, and thus there exists cases that some referenced IAP lines remained in the prefetch cache for the entire executing time of the program. As an example, consider the situation in Figure 6.7. In which the oldest line $line_{i4}$ may remain in the prefetch cache for the entire execution time of program.

Moreover, FIFO is a good choice as FIFO replacement policy requires the simplest hardware complexity among the above three replacement policy.

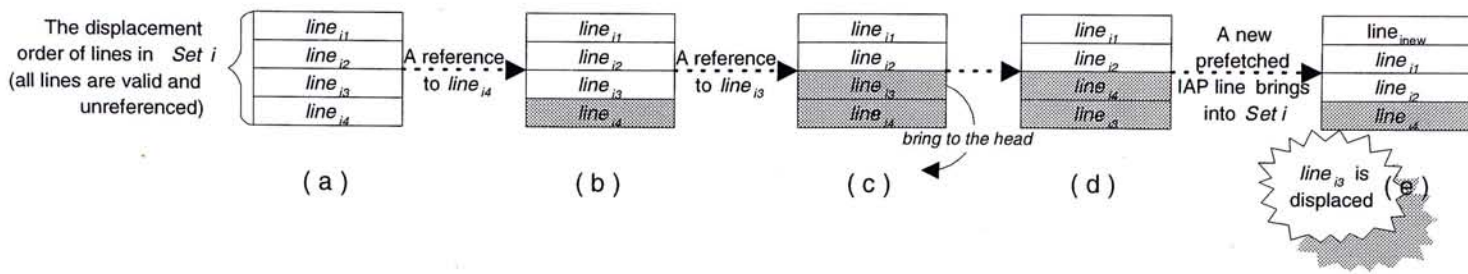


Figure 6.7: An instance of line activities in IZ prefetch cache

Figures D.13 to D.16 show the results for the four groups of programs in FIFO and LRU, all the three simulated policies in 4-way set associative and fully associative prefetch cache.

The results for prefetch cache show that the addition of a small amount of hardware can dramatically improve the system performance.

6.6 Chapter Summary

In this chapter, the performance of IZ, PPUVC and prefetch cache is evaluated using cycle by cycle simulations of the eight SPEC92 benchmarks. For comparison, the performance of a traditional hardware prefetching scheme, prefetch-on-miss, is also included. Besides, to show the potential of the proposed schemes, we also include the performance of the combined IAP scheme. Cache models with varying cache size, line size and associativity are simulated. Except the slight performance degradation for the benchmark program compress, the results show that the three schemes are generally effective in reducing the data access penalty in almost all the other benchmark programs tested.

It is observed that IZ, PPUVC and prefetch cache outperform the combined IAP for most of the eight benchmark programs. Ranging from a few percent up to over 50% of further memory delay time reduction when comparing with the combined IAP scheme.

Chapter 7

Architecture Without

LOAD-AND-STORE Instructions

The IAP that has been proposed so far requires the definition of *LOAD/STORE-UPDATE* compound instructions in the architecture. Power series such as the IBM/MOTOROLA/Apple PowerPC and the IBM RS/6000 contain such kind of instructions. For those machines without *LOAD/STORE-UPDATE* instructions, the IAP scheme still can be easily extended.

First, some architectures have compound instructions that are functionally equivalent to the *LOAD/STORE-UPDATE* instructions defined in IBM PowerPC or RS/6000. One of these instructions is the *LOAD/STORE-MODIFY* in the HP's Precision Architecture (PA RISC) 1.1. As a result, the IAP scheme can be easily extended to this type of machines without any difficulties. Second, for those machines without similar kind of compound instructions, if an update-counter per register is available, then the IAP scheme can still be implemented. The function of the update-counter UC is to book-keep its corresponding register R(UC), if the register is an index register used by some *LOAD/STORE* instructions using index-displacement addressing mode in a loop, the value of the stride used by the index displacement *LOAD/STORE* instructions will be learnt during the first iteration of the loop and will be stored into the update-counter UC. Consequently, very accurate data prefetching comparable to the IAP scheme can be carried out

in the remaining iterations of the loop to improve the cache performance. As long as the IAP scheme can be implemented, the IZ scheme can also be used.

Moreover, to show the potential of Priority Pre-Updating scheme, it is possible to import pre-updating into architectures which employ different prefetching algorithm, no matter it is prefetch-on-miss or one-block-lookahead.

Chapter 8

Conclusion

In this dissertation, we propose two cache line replacement policies – IZ and PPUVC, and one placement policy – prefetch cache. These three policies are specially designed for prefetched lines.

The first replacement policy, called IZ, is to be implemented with the IAP scheme. This is used to improve data cache performance. From our simulation study, we found that with this replacement policy built into the combined IAP scheme, the processor idle time due to memory access can easily be reduced by over 20%. In some programs, this replacement can even achieve a 99% of memory delay time reduction.

In fact, the IZ replacement policy with IAP scheme has very good potential to be imported into current cache designs. The reasons are: [1] no change in the architecture is required, [2] no new compiler optimization technique is required, [3] the IZ scheme can work with different kinds of replacement policies, [4] the IZ with IAP has high potential to improve system performance. Though the controller design for the IZ replacement strategy is similar to LRU, it is worthy to implement due to the low cost of hardware.

The second replacement policy, Priority Pre-Updating, helps to determine which data should be replaced during cache misses by shortening the life time of those suspected erroneously prefetched lines. However, using PPU solely may not be able to obtain significant improvement. The program data set is usually

much larger than the cache size, and thus thrashing of unreferenced useful lines occurs frequently. Therefore, a small fully-associative victim cache is added to hold those unreferenced prefetched lines, which have been displaced from data cache due to capacity or conflict misses. By using combined PPU and victim cache, it is possible to achieve up to 50% memory delay time reduction in some of the SPEC92 benchmark suite in the cache model with prefetch-on-miss only. PPUVC can achieve up to 100% of memory delay time reduction in cache model with IAP scheme.

Victim cache consists of only four entries, each entry is of the same size as a cache line, and is insignificant when comparing with the size of data cache. Therefore, it is worthy to implement it in current architecture. A more important concern, perhaps, is the extra hardware logic necessary to search the PPU and updating the cache. With the even-increasing amount of hardware logic available on a chip, the quantity of it involved is not really a serious problem.

The placement policy – prefetch cache is used to hold prefetched IAP lines. With the assumption that the data cache and prefetch cache will be checked for a data reference, it is possible to achieve up to 90% of memory delay time reduction.

Hardware costs are now low enough to permit extra hardware for prefetch cache, i.e., the amount of cache available for data is not necessarily reduced as a prefetch cache is added. As a result, it is worthy to implement prefetch cache in current architectures. This technique helps to reduce cache pollution and increases system performance.

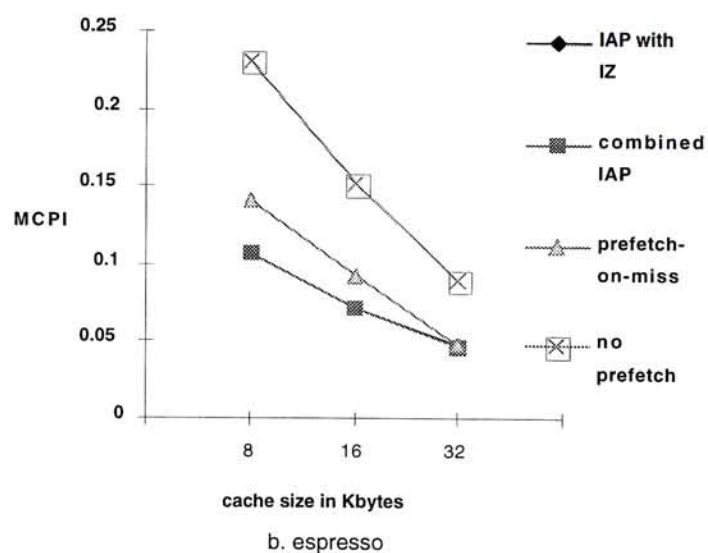
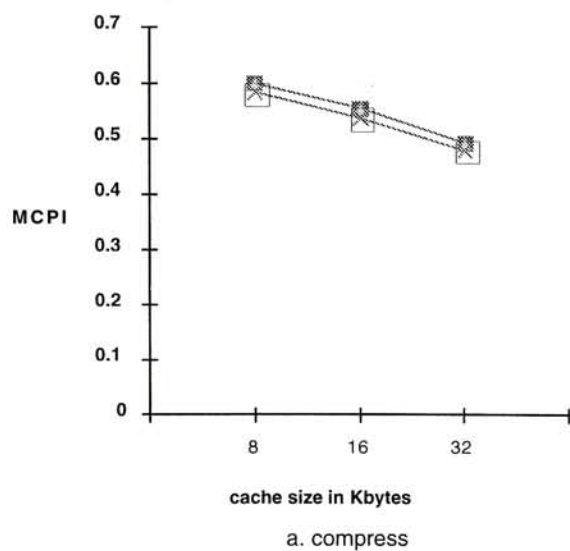
Whether or not the proposed mechanisms would be practical to implement in hardware is not really addressed in our experimentation, but the indications are that the difficulties would be the minor. Having some of the more complicated heuristics proven to be extremely useful, then their introduction into a hardware scheme could have proven challenging.

Appendix A

CPI Due to Cache Misses

A.1 Varying Cache Size

A.1.1 Instant Zero Replacement Policy



Appendix A CPI Due to Cache Misses

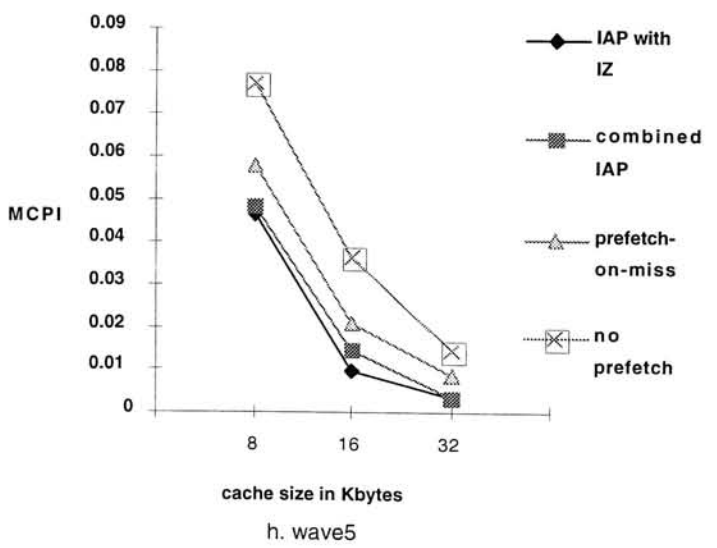
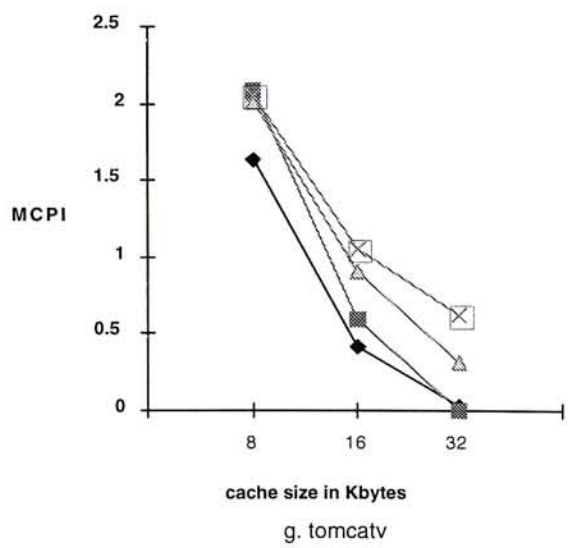
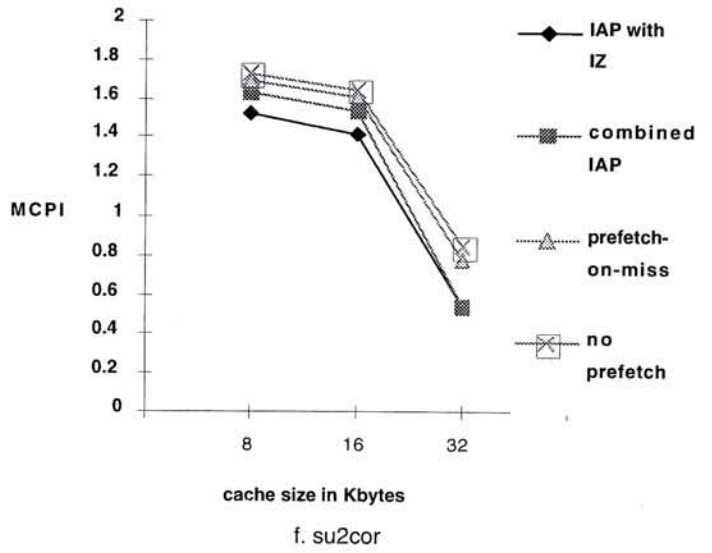
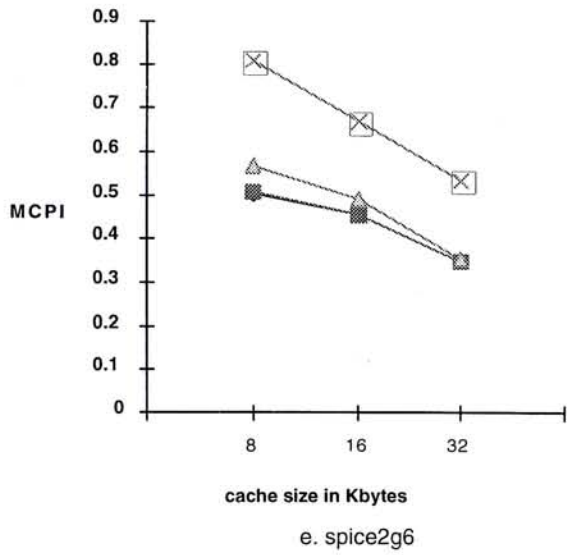
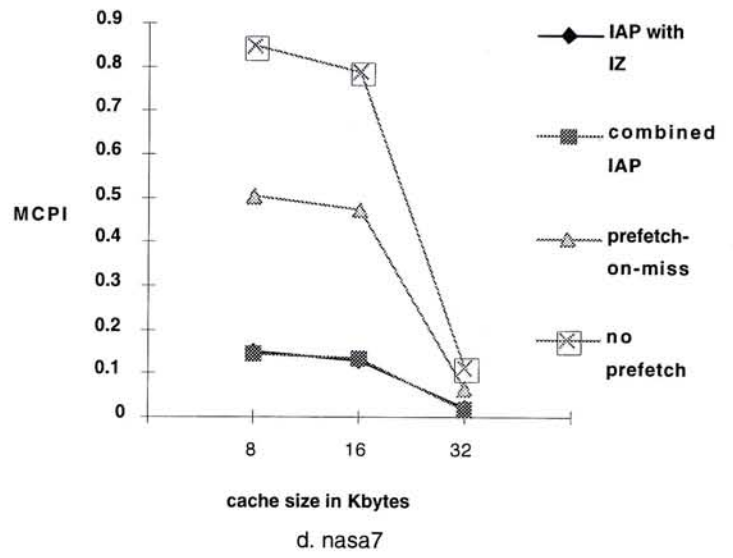
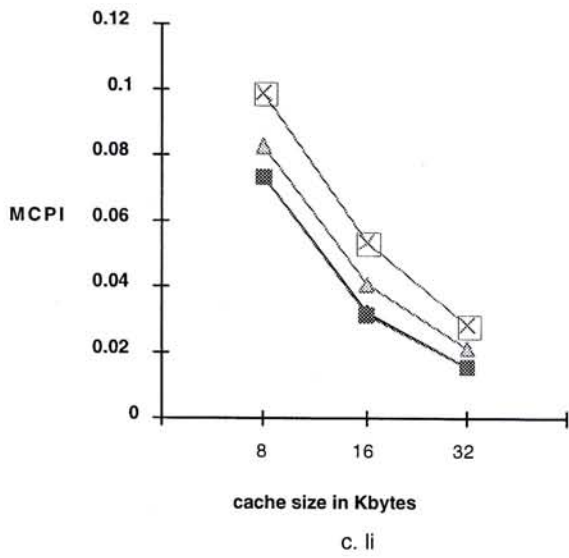
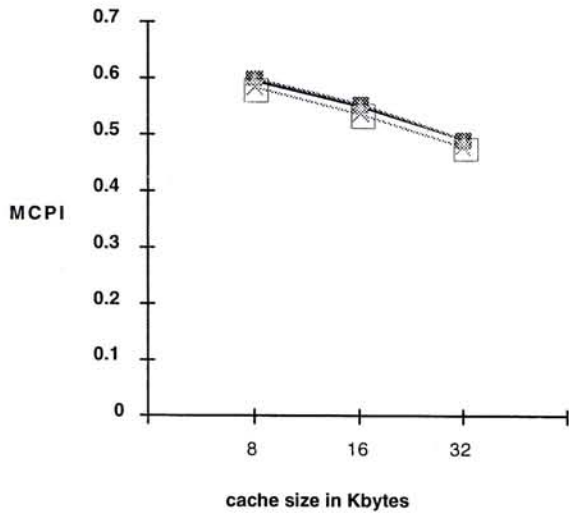
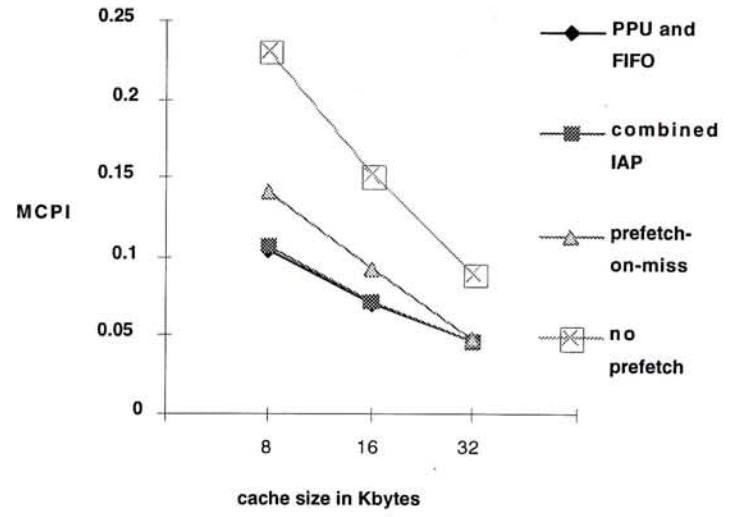


Figure A.1: MCPI by varying cache size in IZ scheme

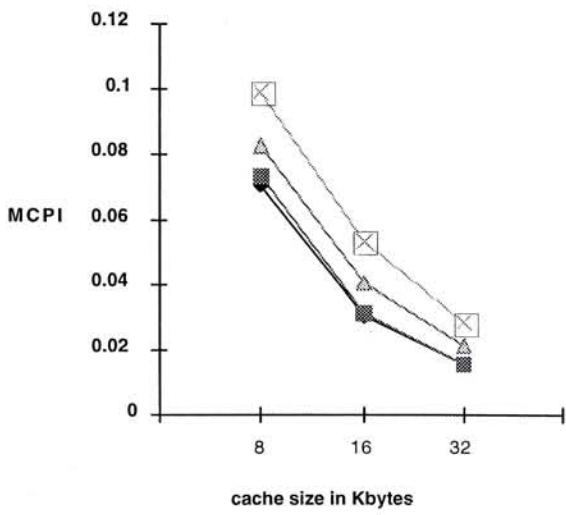
A.1.2 Priority Pre-Updating with Victim Cache



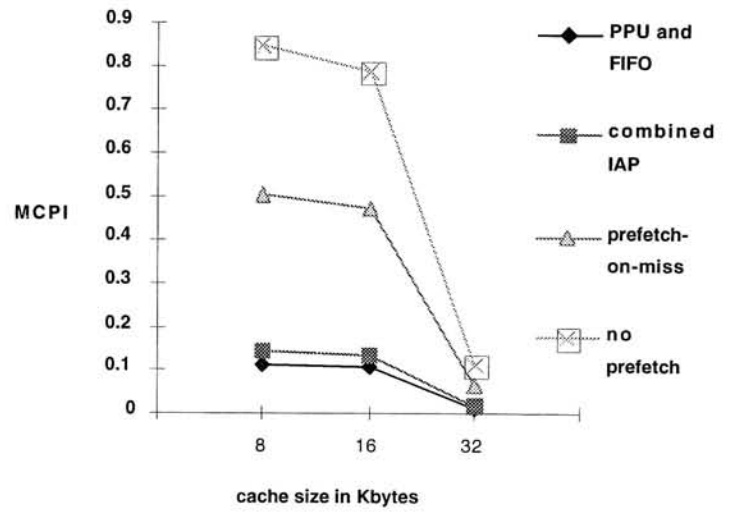
a. compress



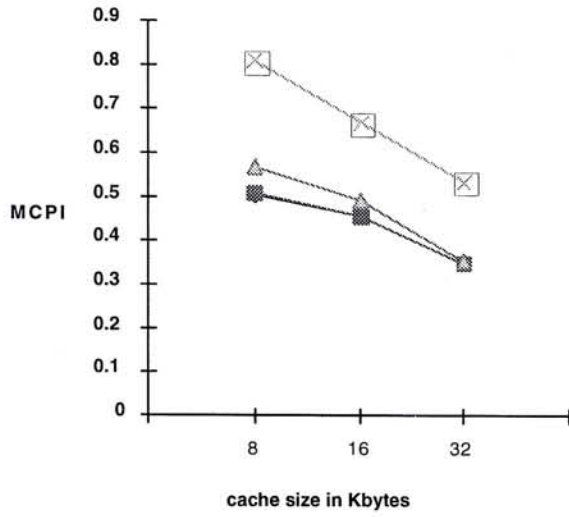
b. espresso



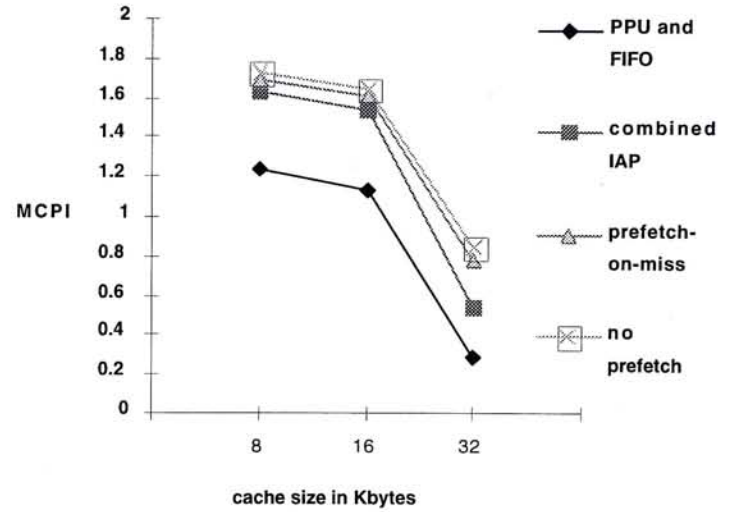
c. li



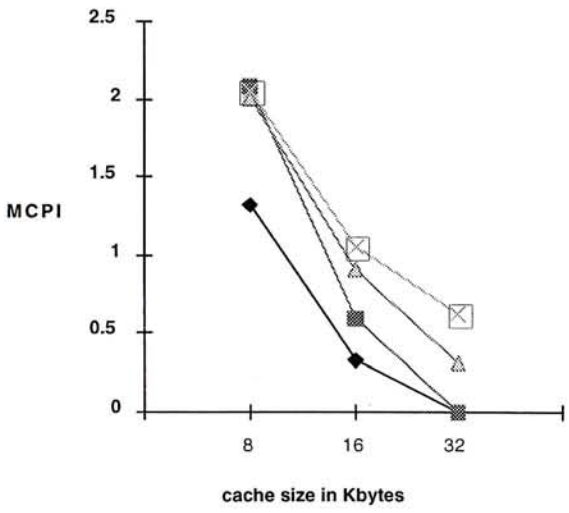
d. nasa7



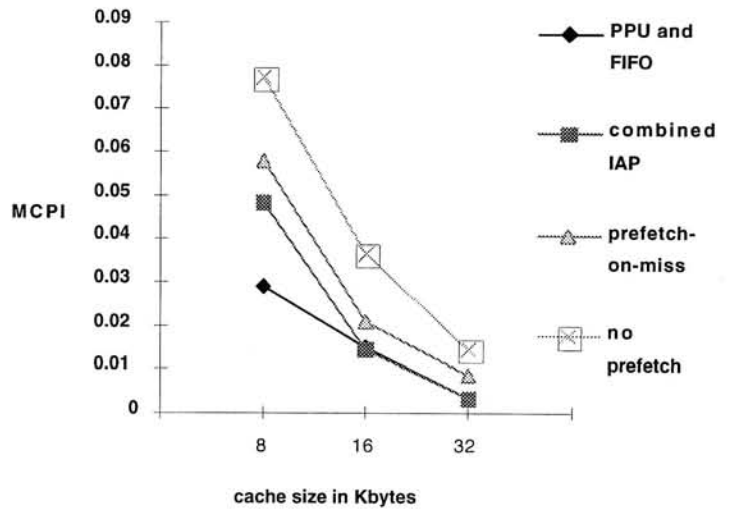
e. spice2g6



f. su2cor

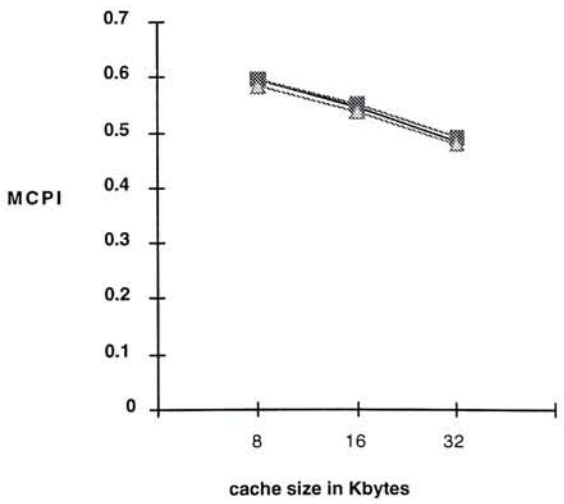


g. tomcatv

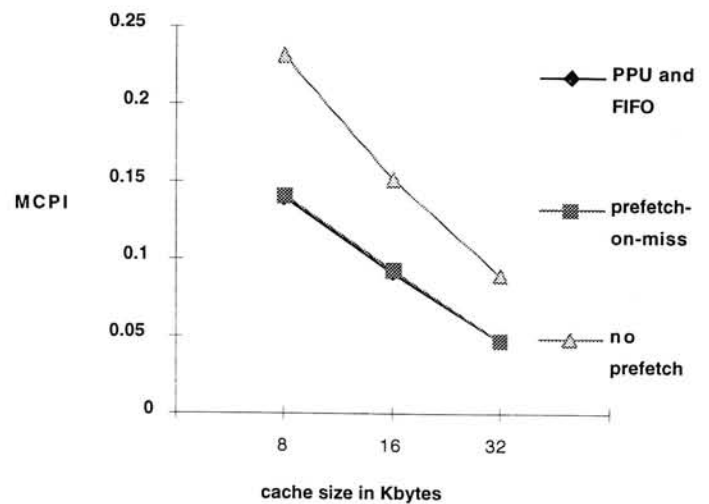


h. wave5

Figure A.2: MCPI by varying cache size in PPUVC with IAP scheme

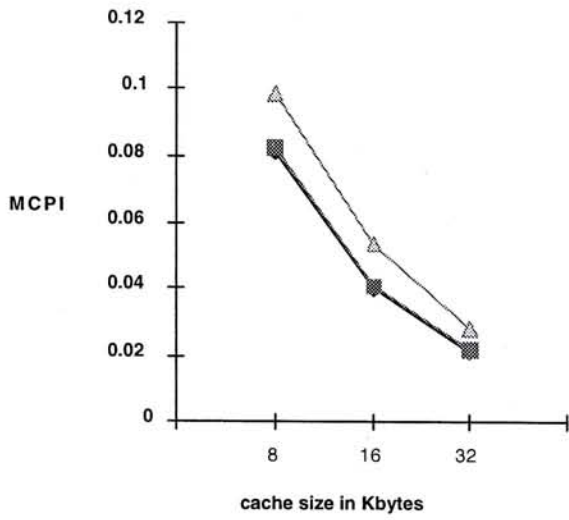


a. compress

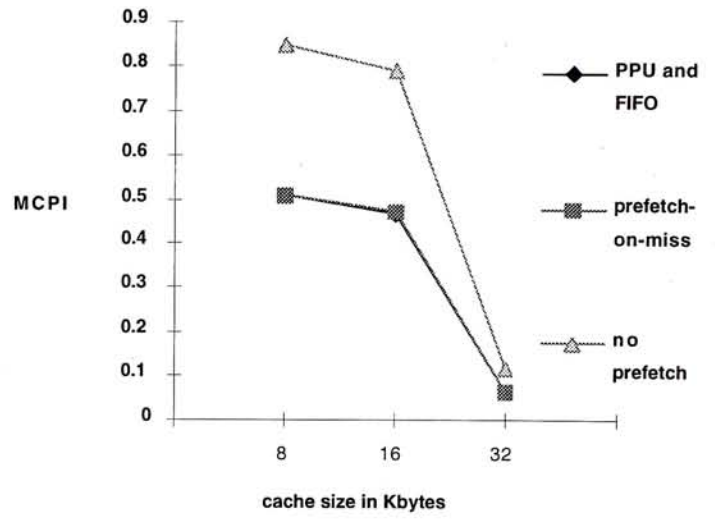


b. espresso

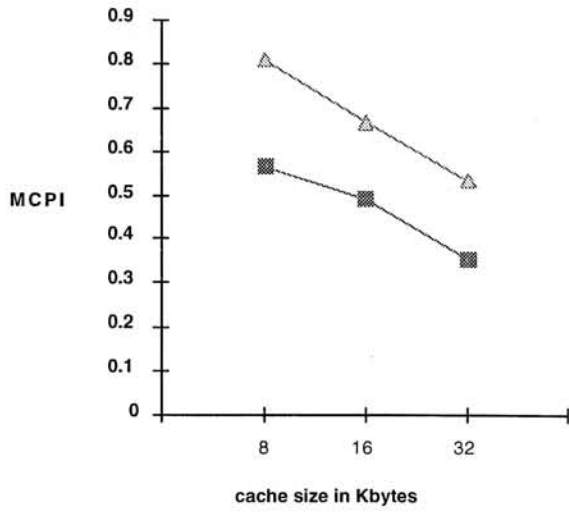
Appendix A CPI Due to Cache Misses



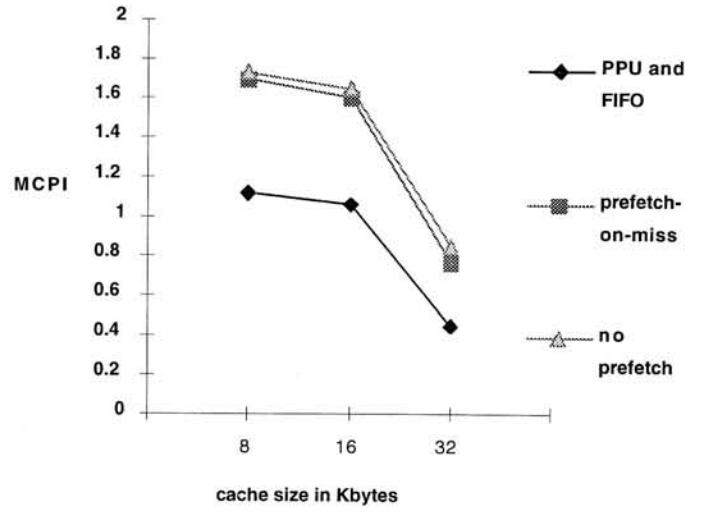
c. li



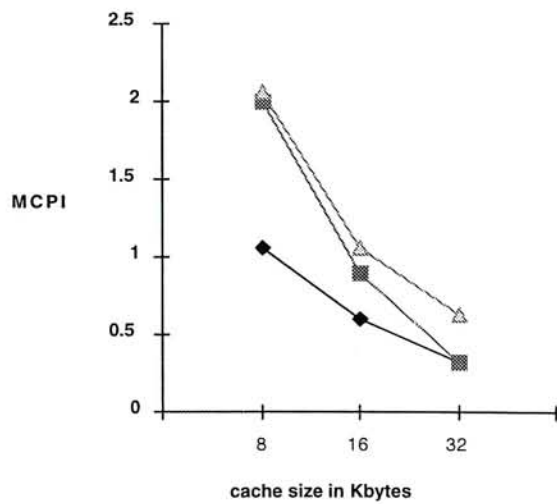
d. nasa7



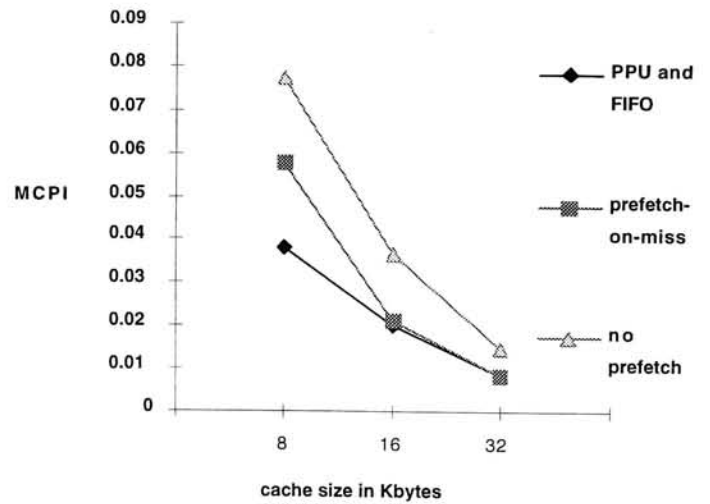
e. spice2g6



f. su2cor



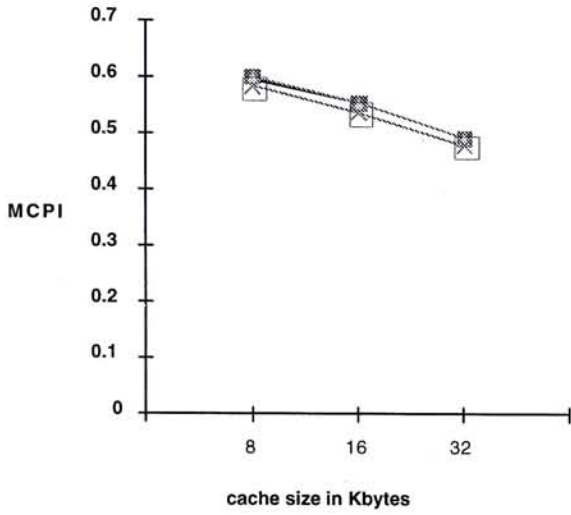
g. tomcatv



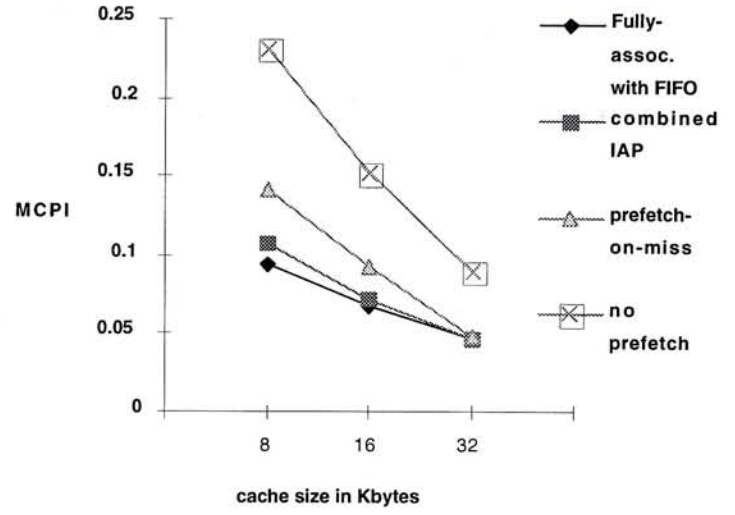
h. wave5

Figure A.3: MCPI by varying cache size in PPUVC with prefetch-on-miss scheme

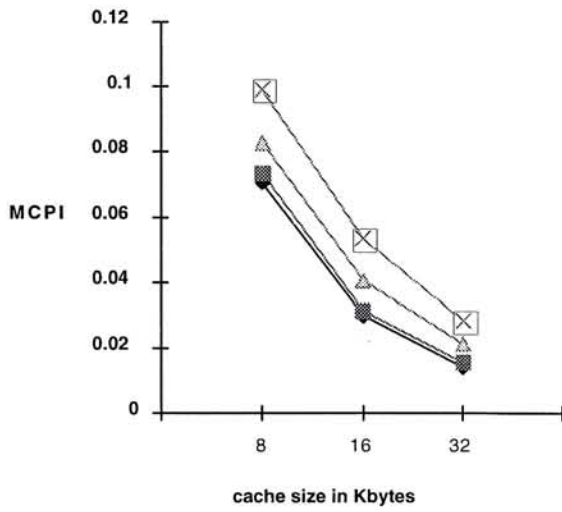
A.1.3 Prefetch Cache



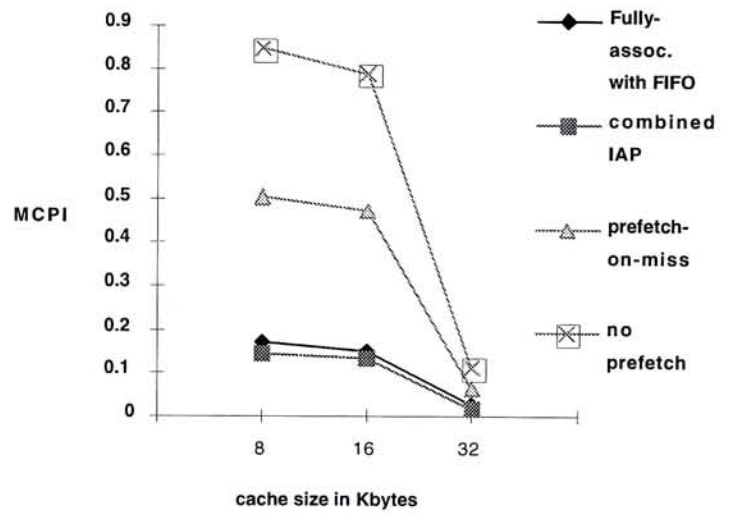
a. compress



b. espresso



c. li



d. nasa7

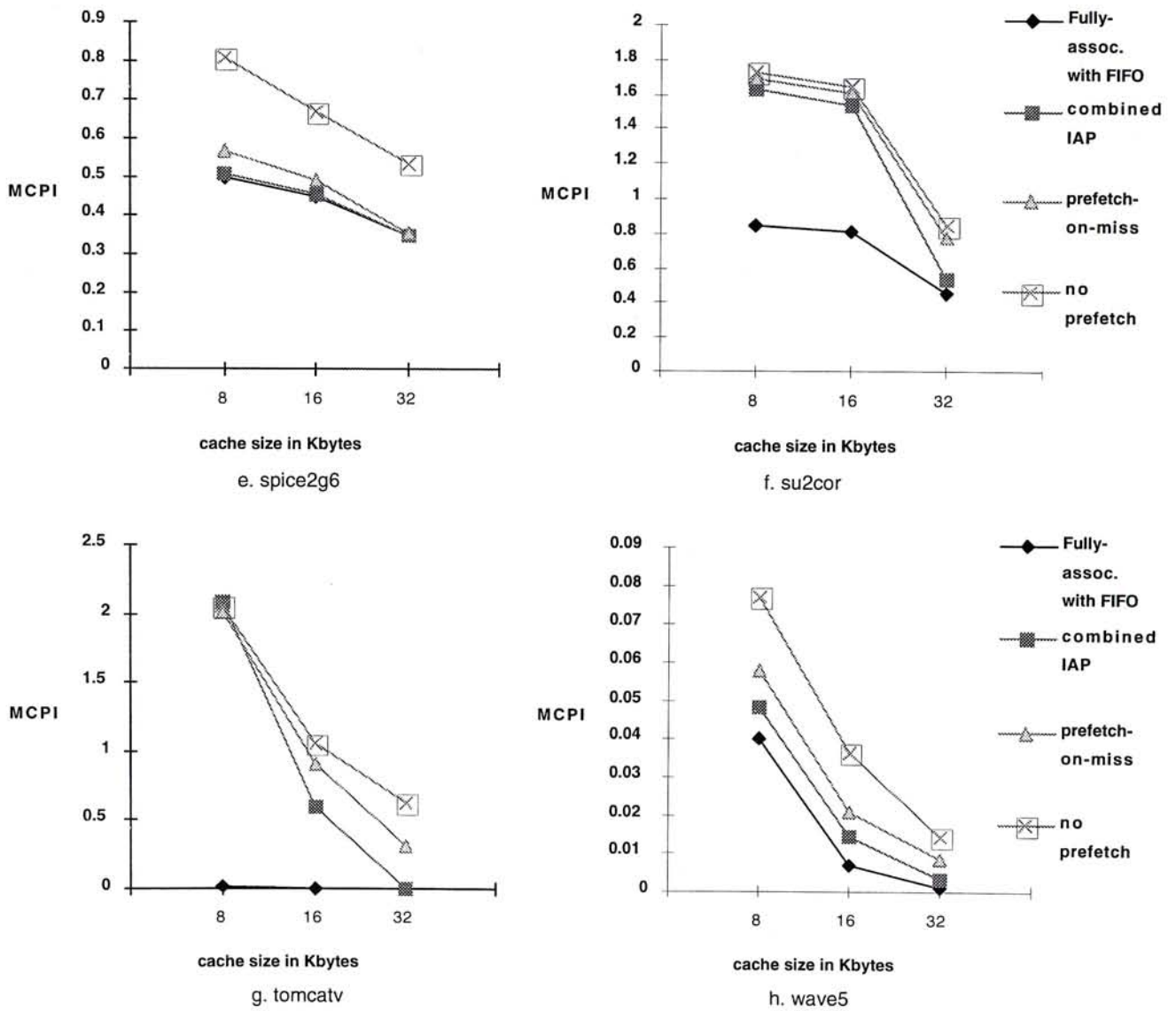
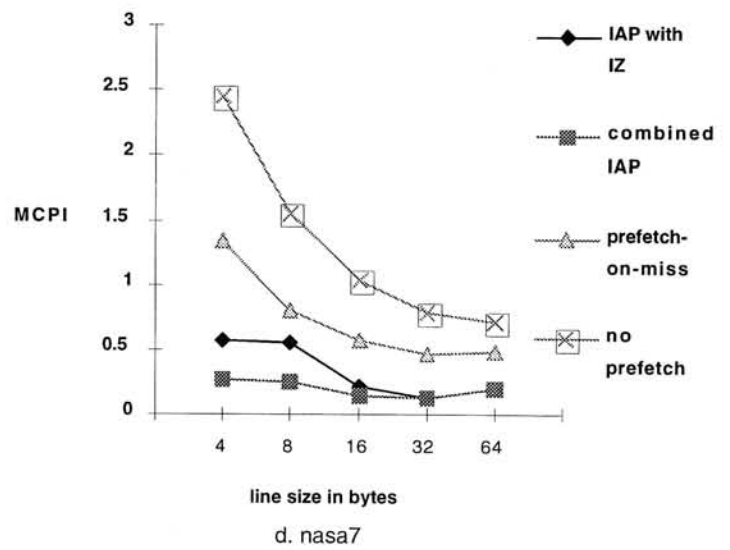
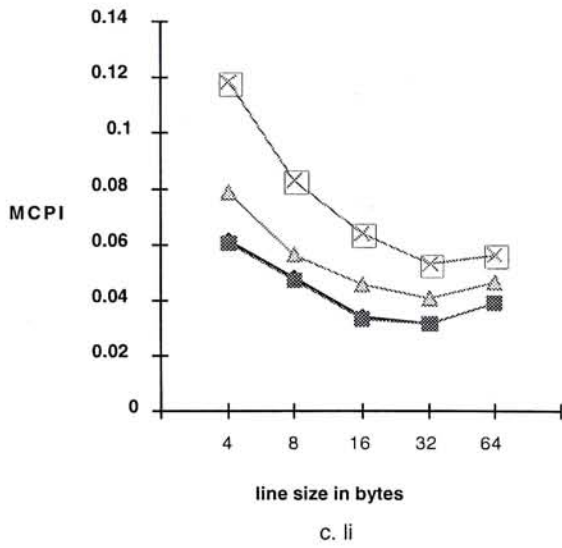
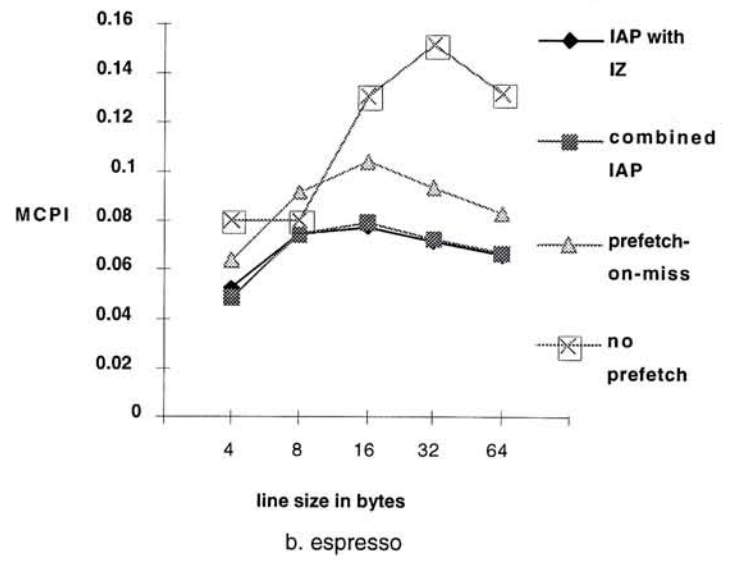
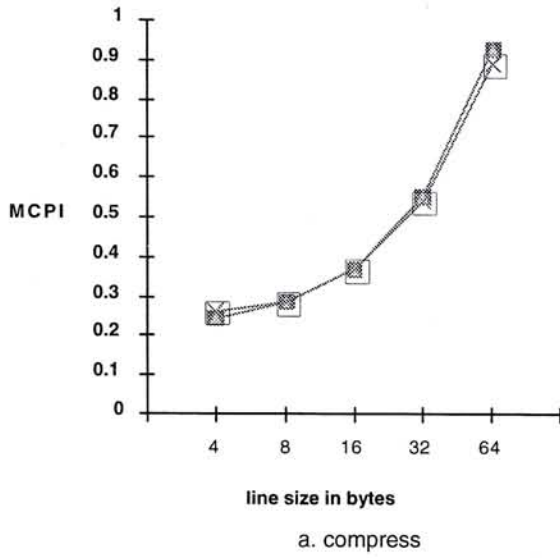
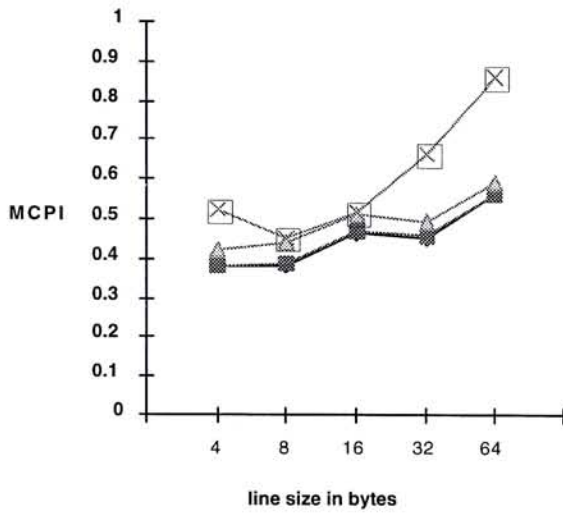


Figure A.4: MCPI by varying cache size in prefetch cache scheme

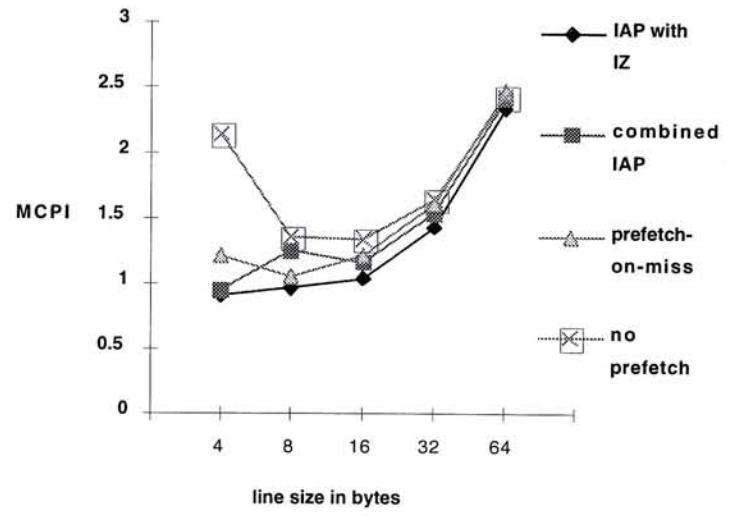
A.2 Varying Cache Line Size

A.2.1 Instant Zero Replacement Policy





e. spice2g6



f. su2cor

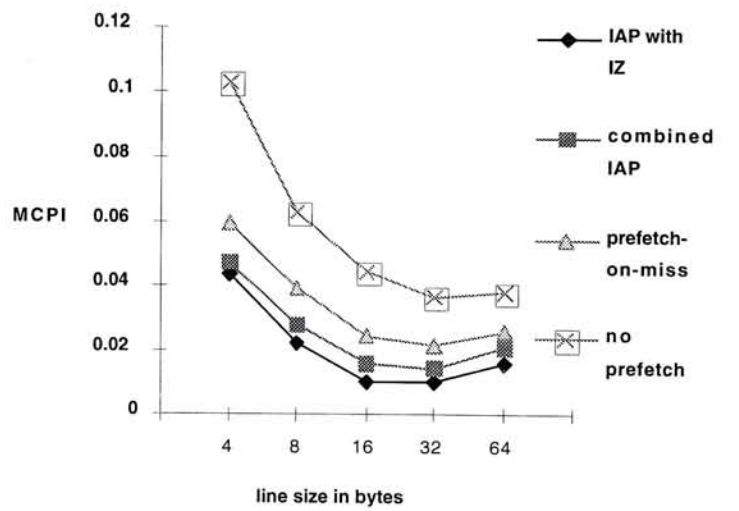
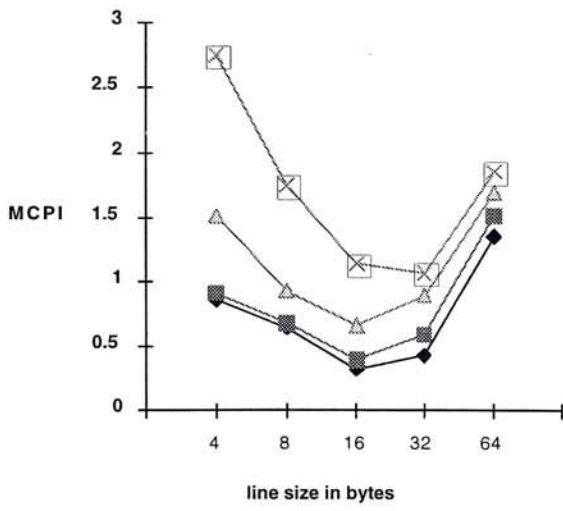
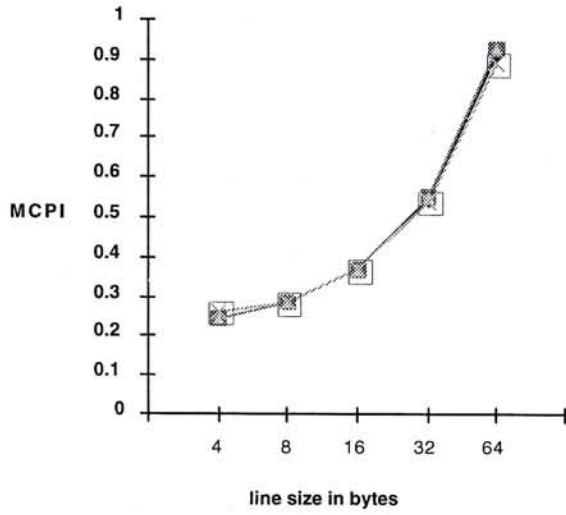
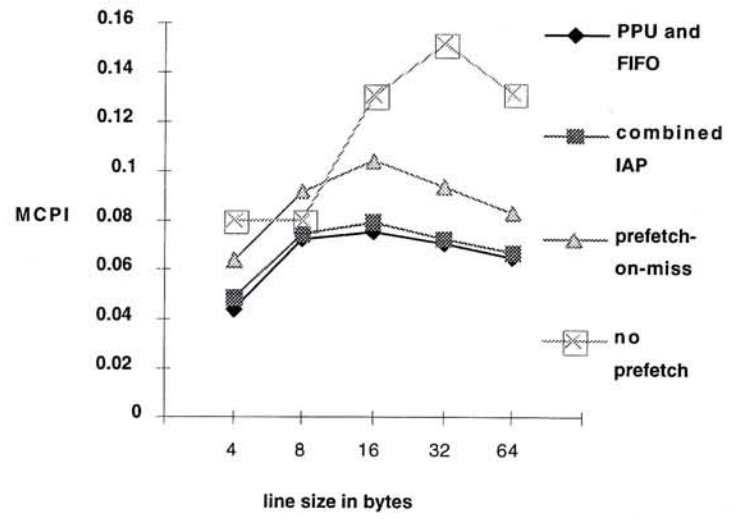


Figure A.5: MCPI by varying cache line size in IZ scheme

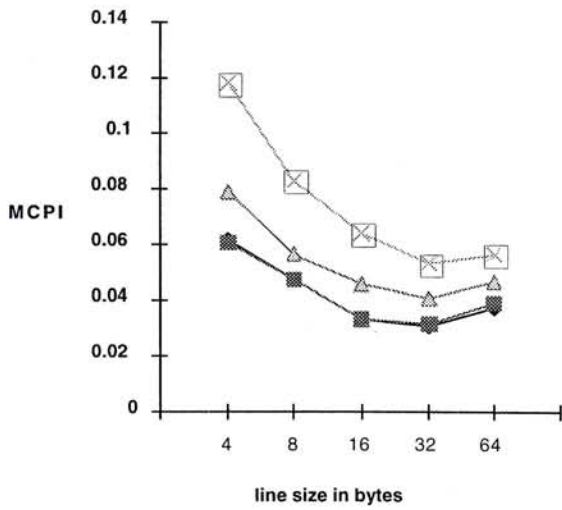
A.2.2 Priority Pre-Updating with Victim Cache



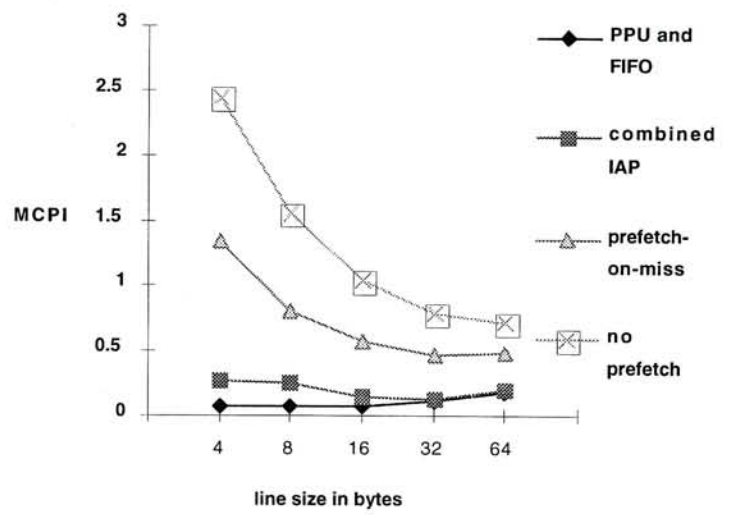
a. compress



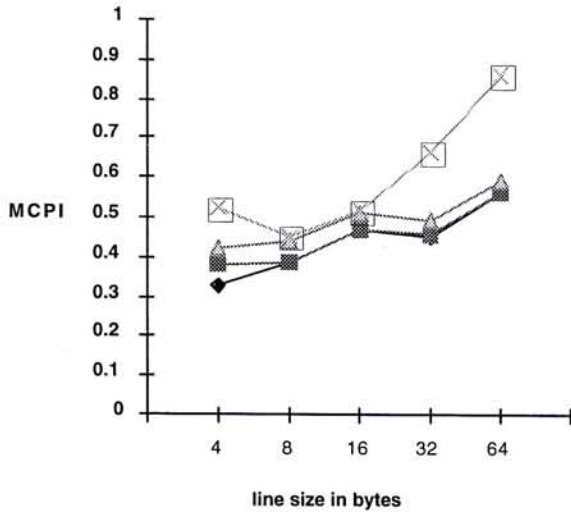
b. espresso



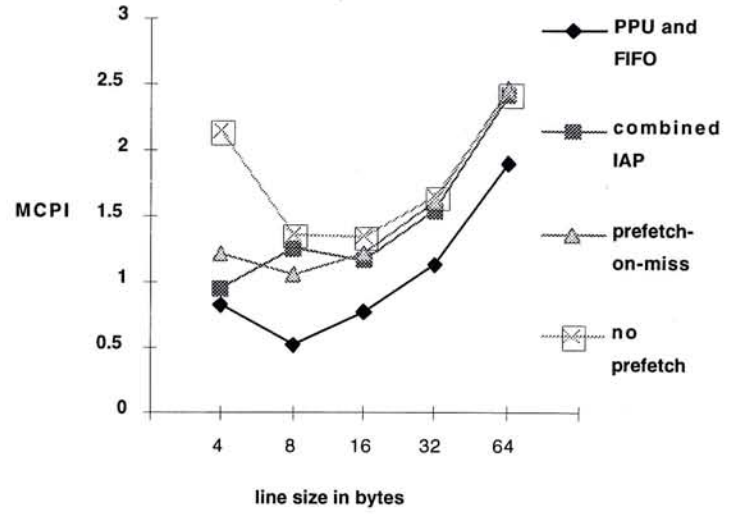
c. li



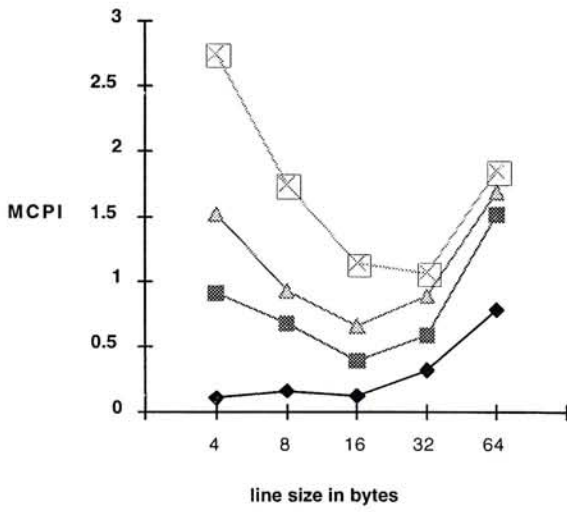
d. nasa7



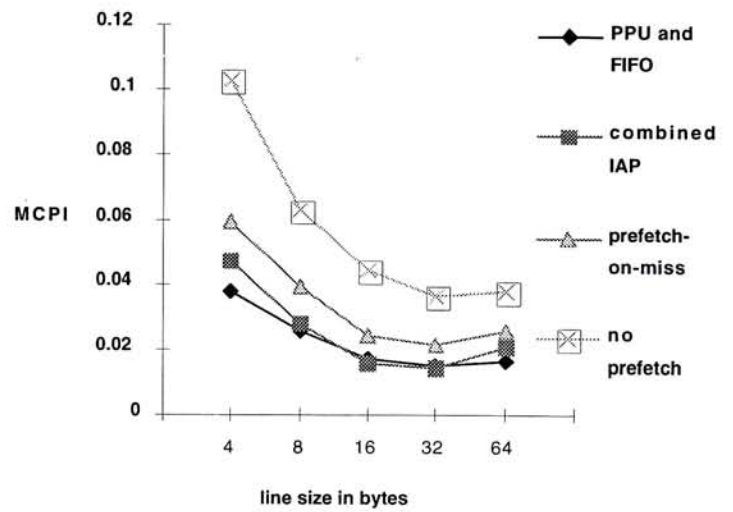
e. spice2g6



f. su2cor

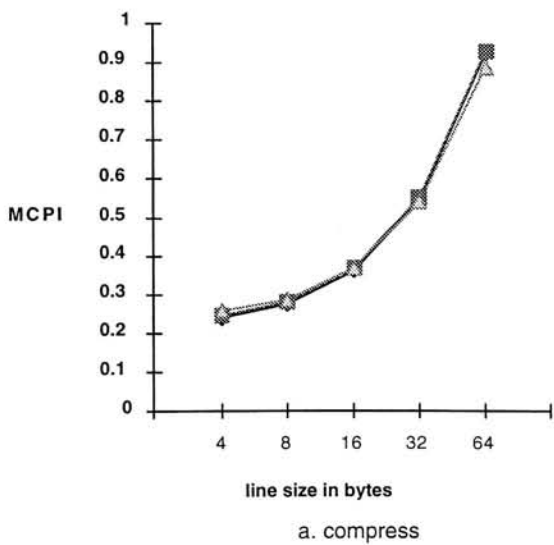


g. tomcatv

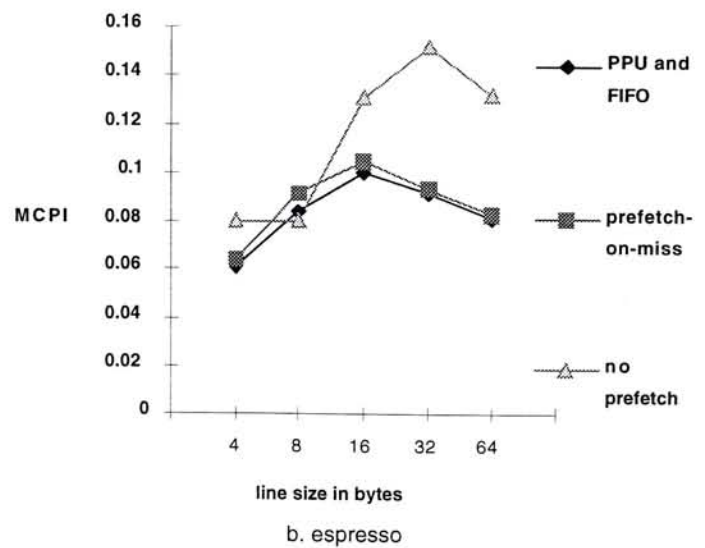


h. wave5

Figure A.6: MCPI by varying cache line size in PPUVC with IAP scheme

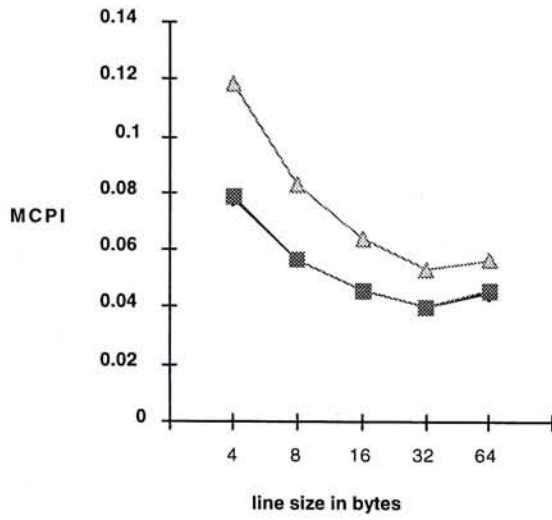


a. compress

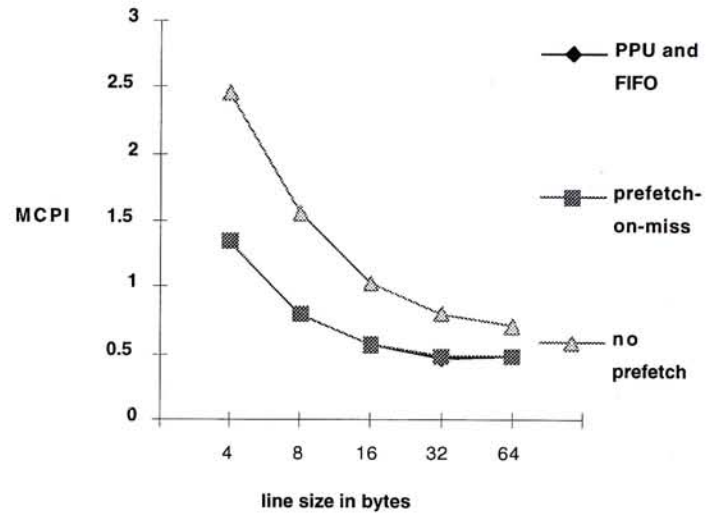


b. espresso

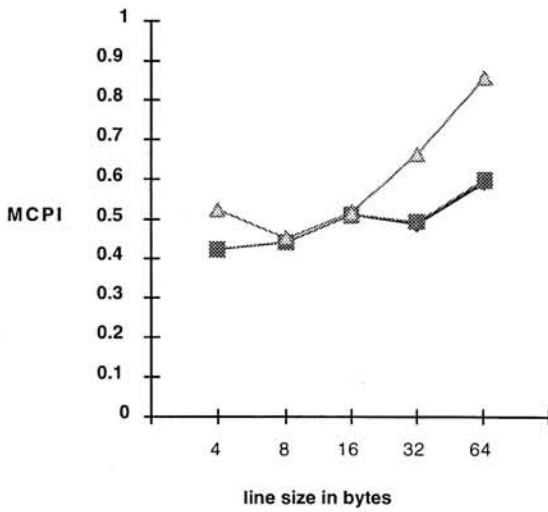
Appendix A CPI Due to Cache Misses



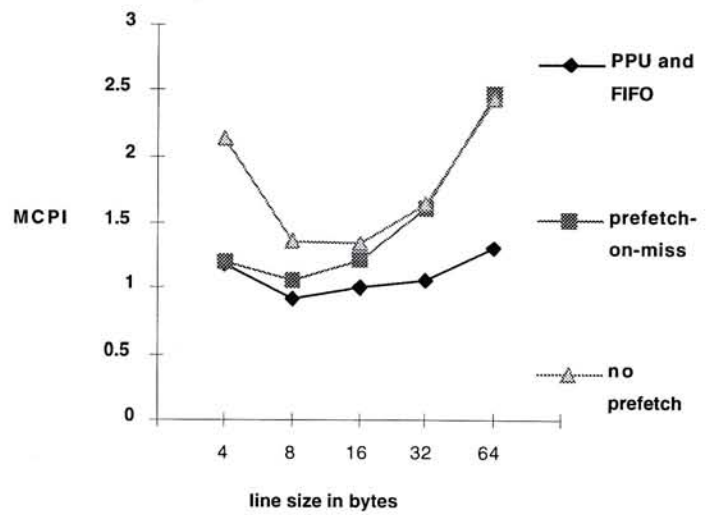
c. li



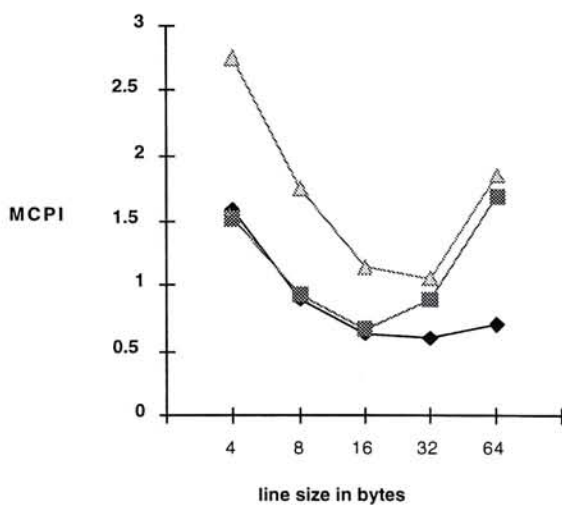
d. nasa7



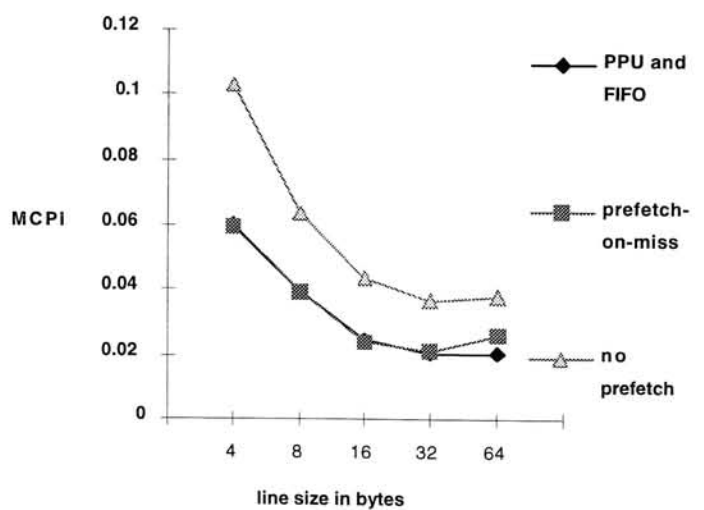
e. spice2g6



f. su2cor



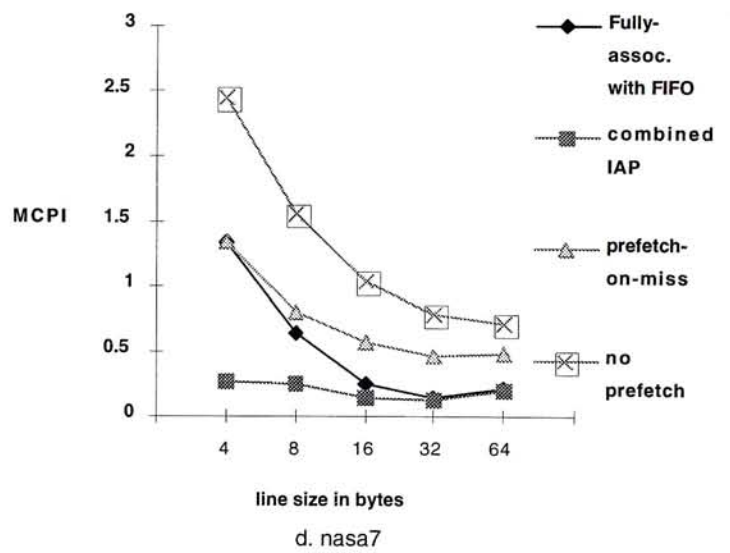
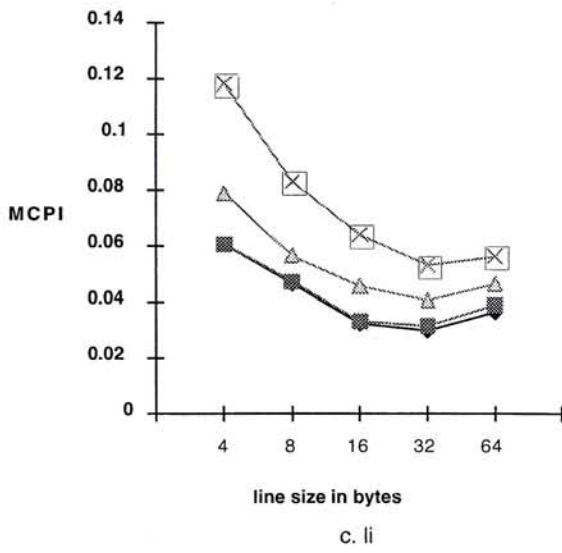
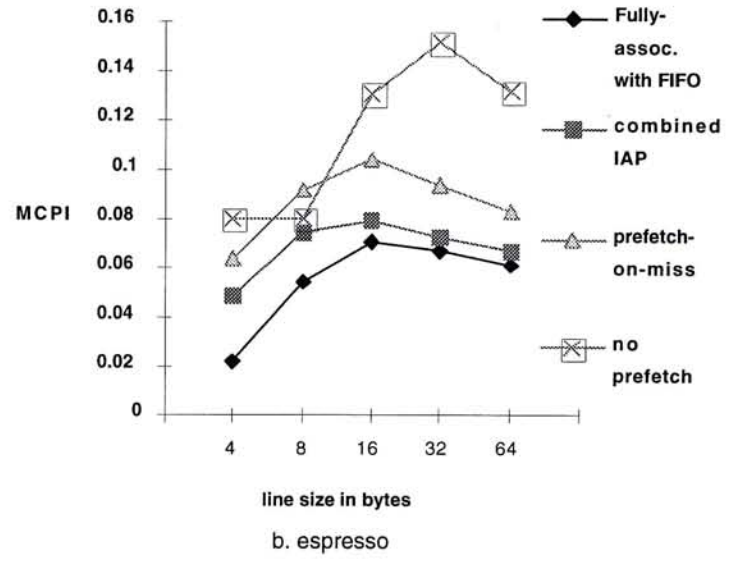
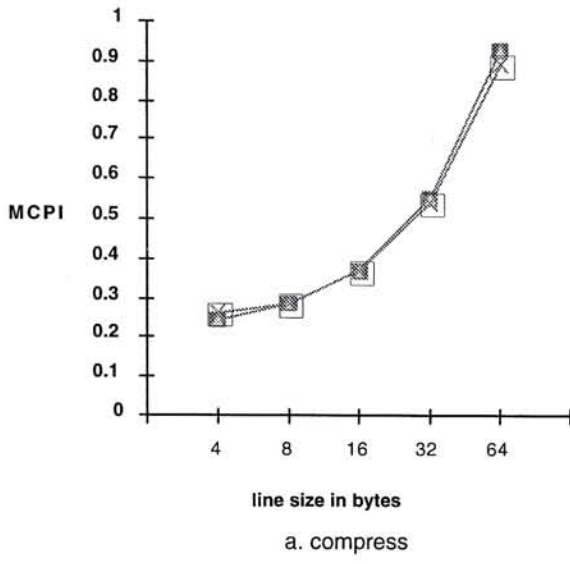
g. tomcatv



h. wave5

Figure A.7: MCPI by varying cache line size in PPUVC with prefetch-on-miss scheme

A.2.3 Prefetch Cache



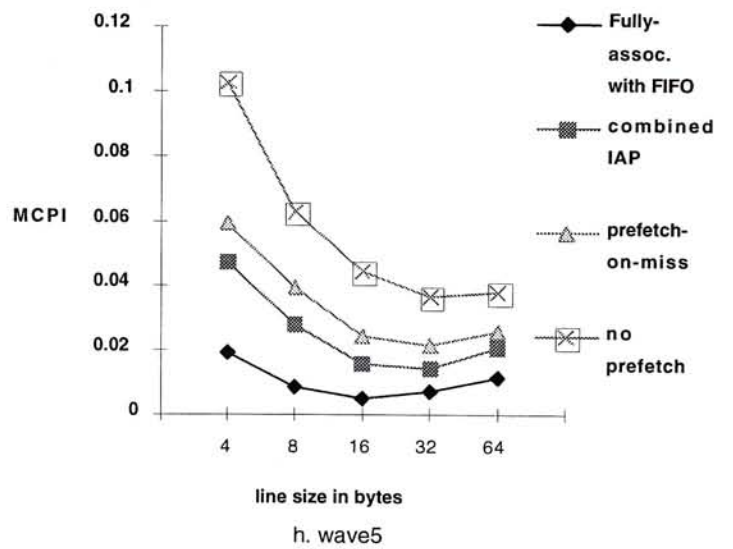
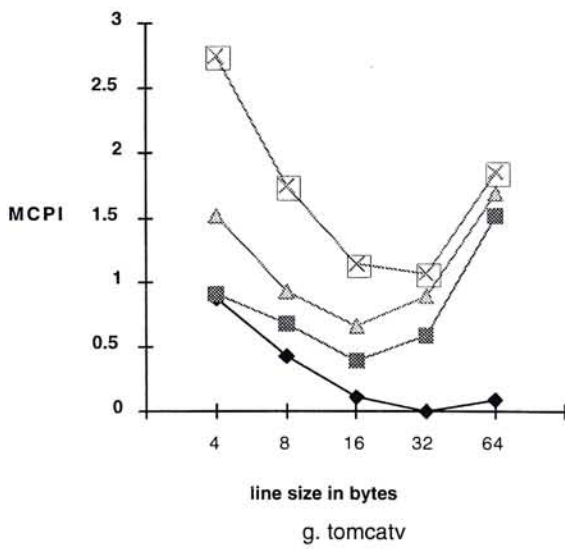
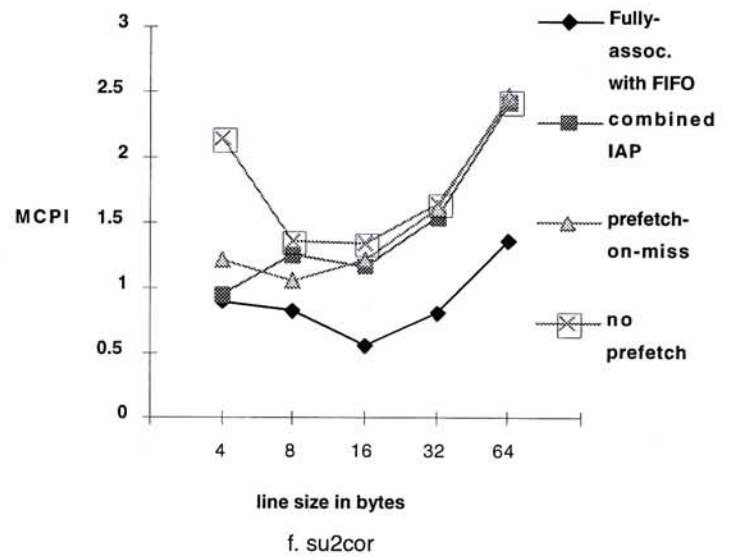
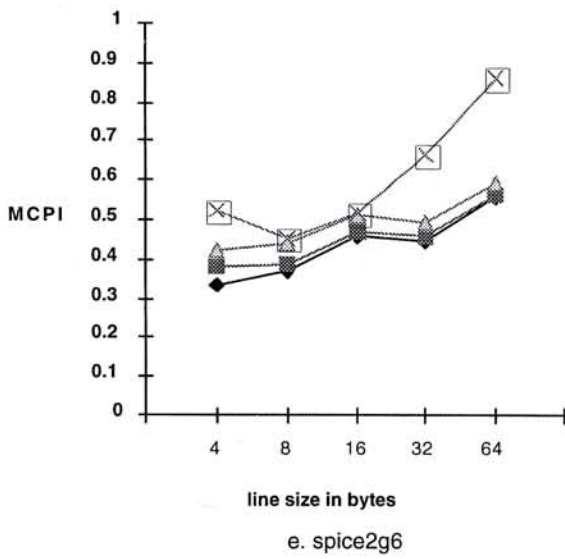
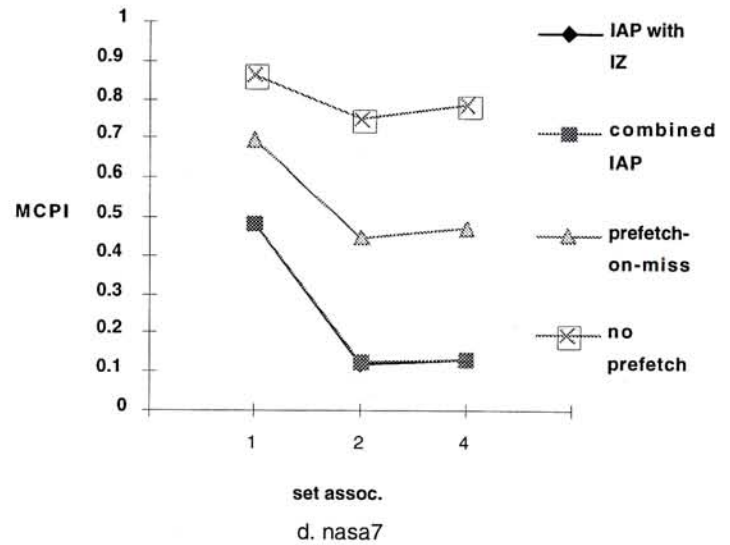
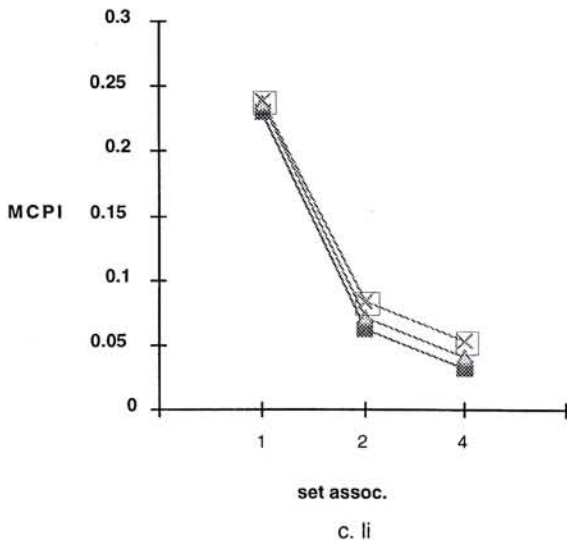
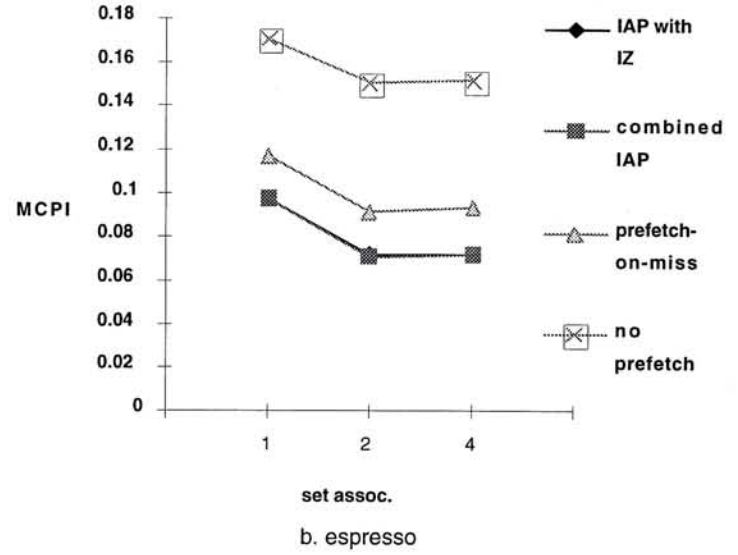
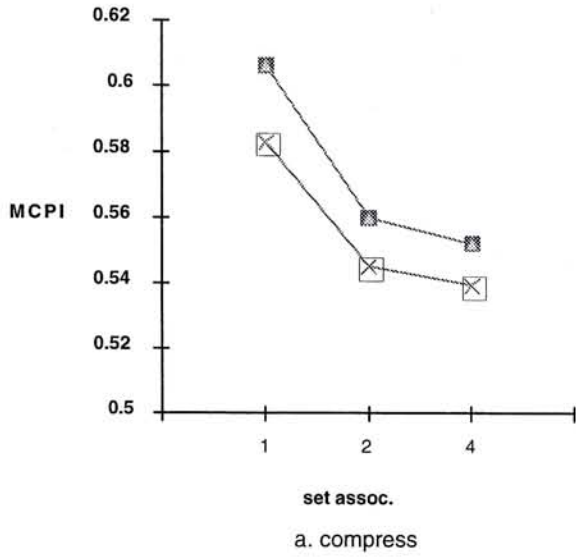


Figure A.8: MCPI by varying cache line size in prefetch cache scheme

A.3 Varying Cache Set Associative

A.3.1 Instant Zero Replacement Policy



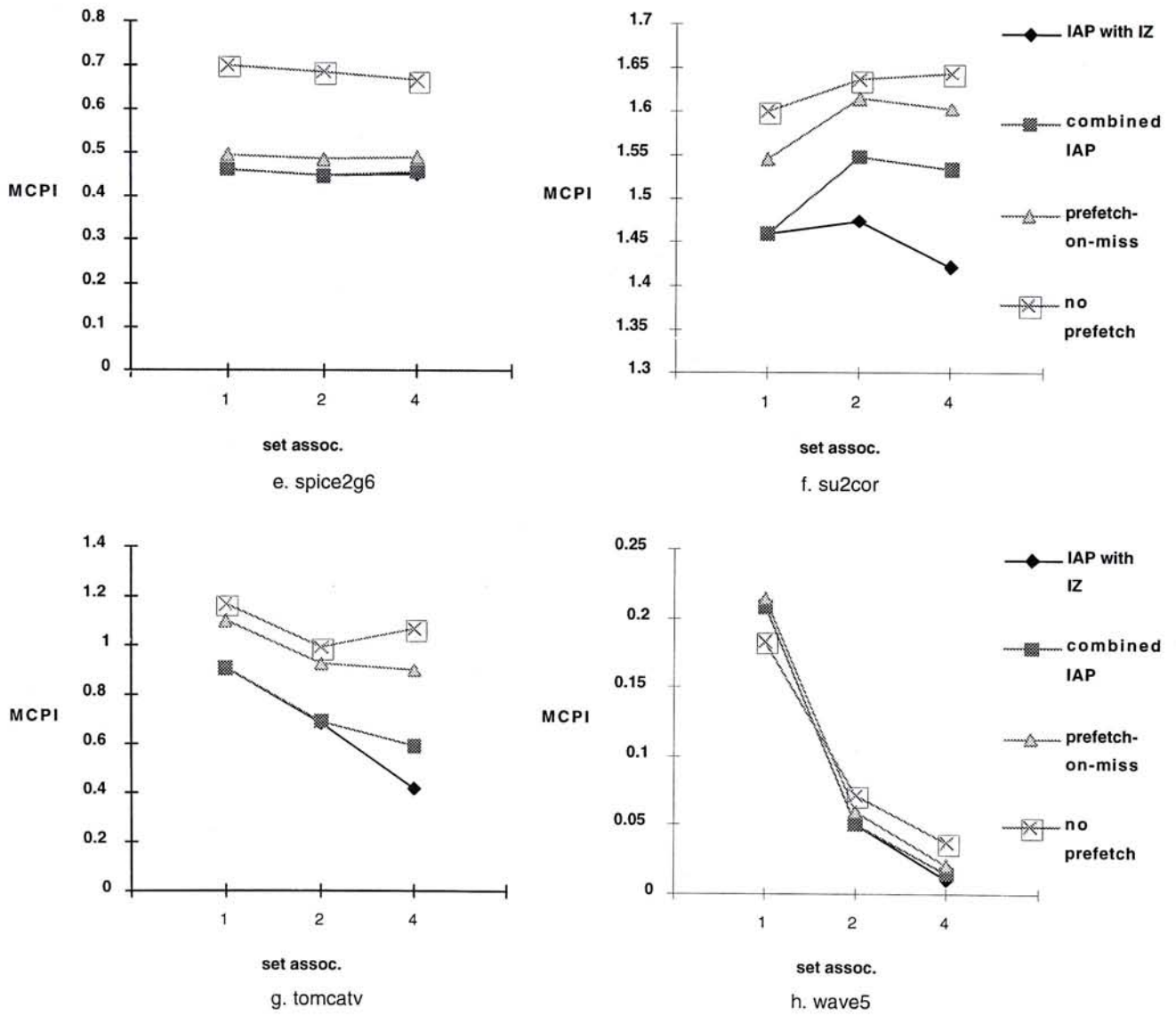
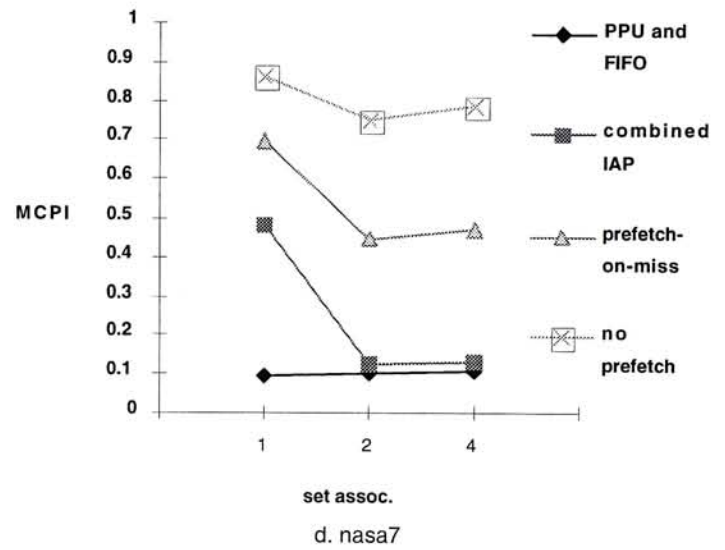
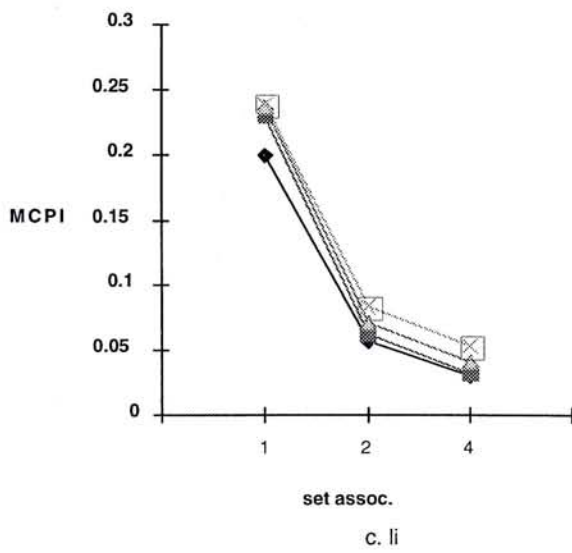
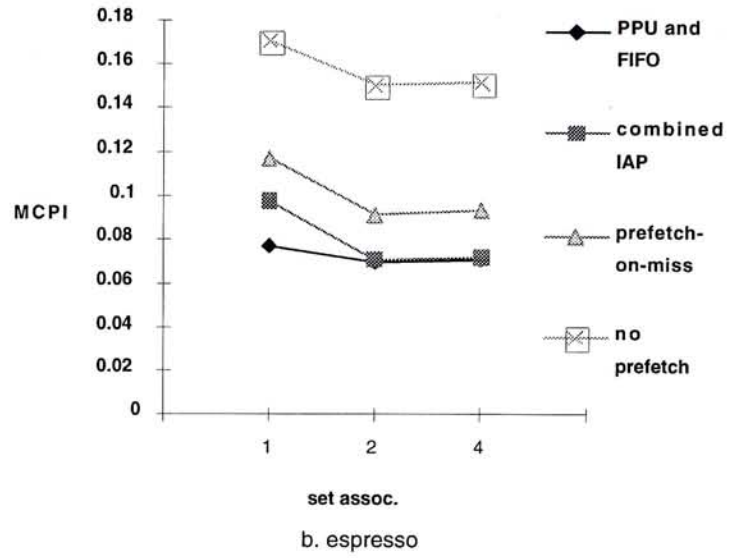
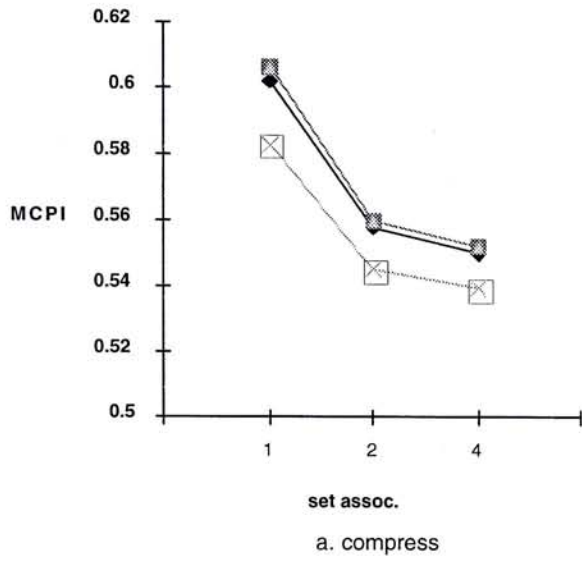


Figure A.9: MCPI by varying set associative in IZ scheme

A.3.2 Priority Pre-Updating with Victim Cache



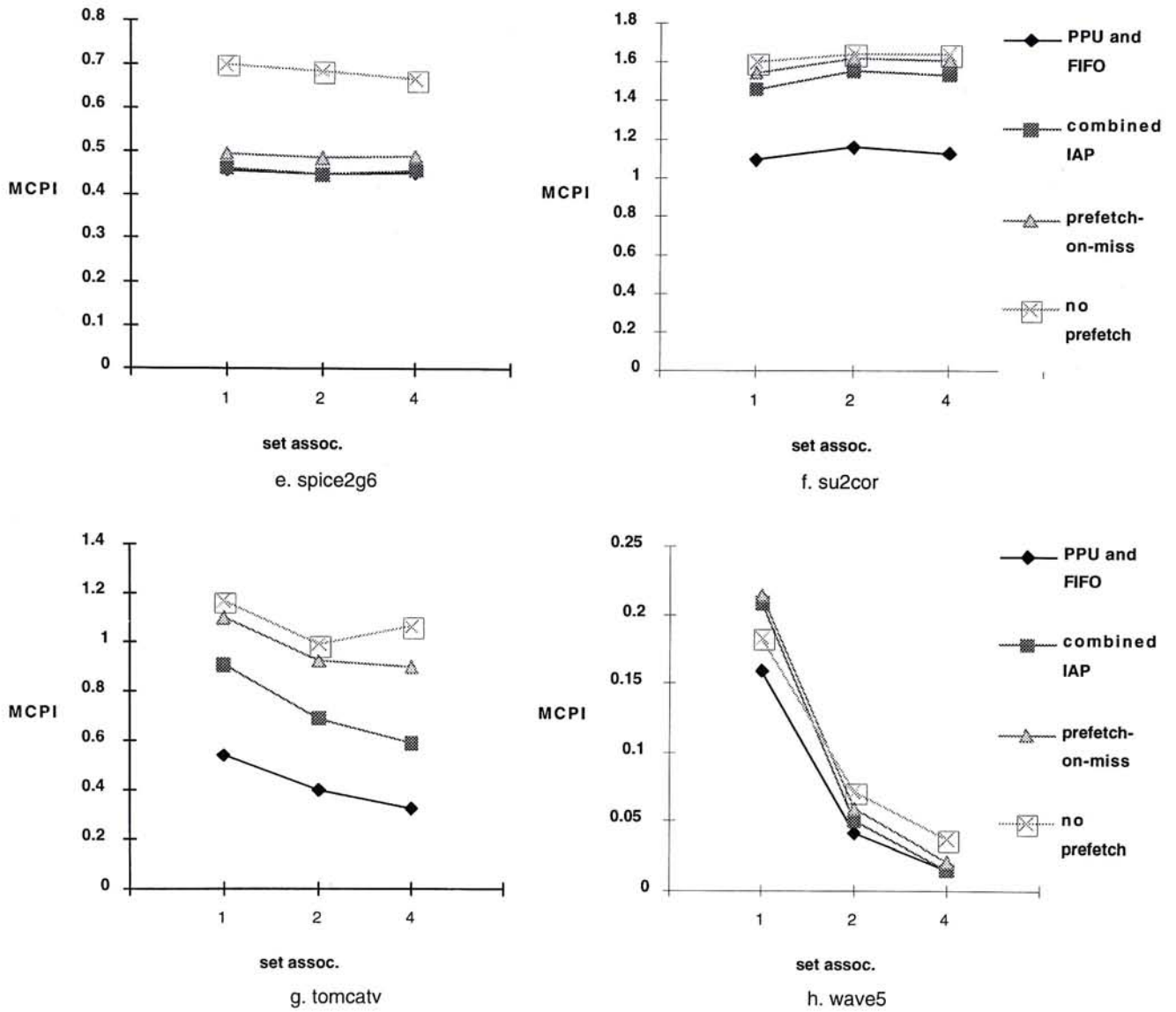
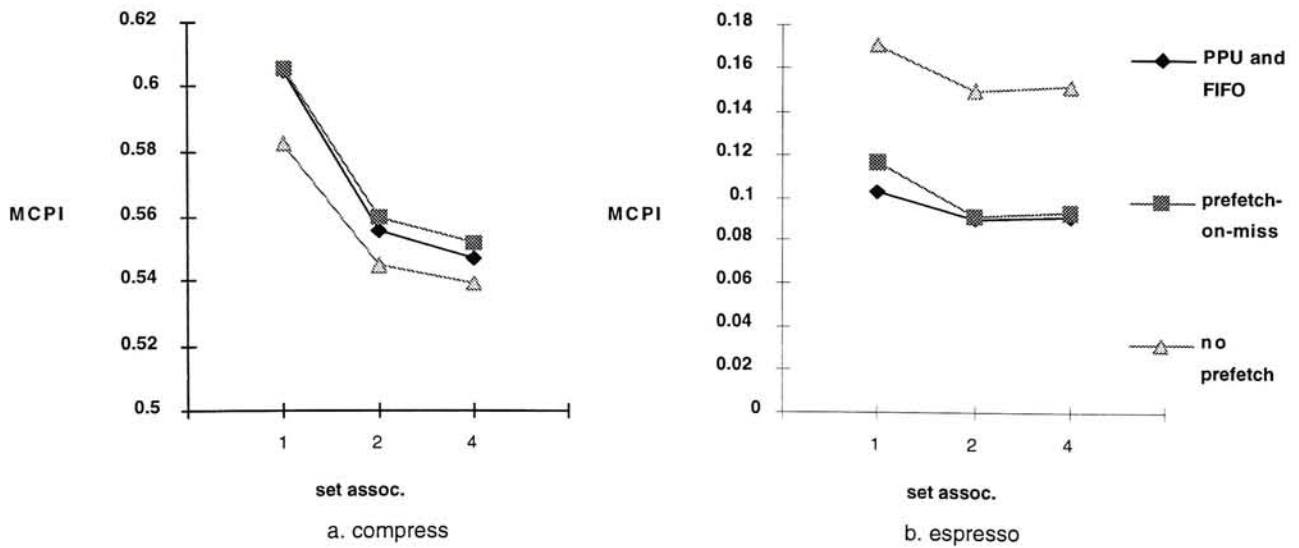


Figure A.10: MCPI by varying set associative in PPUVC with IAP scheme



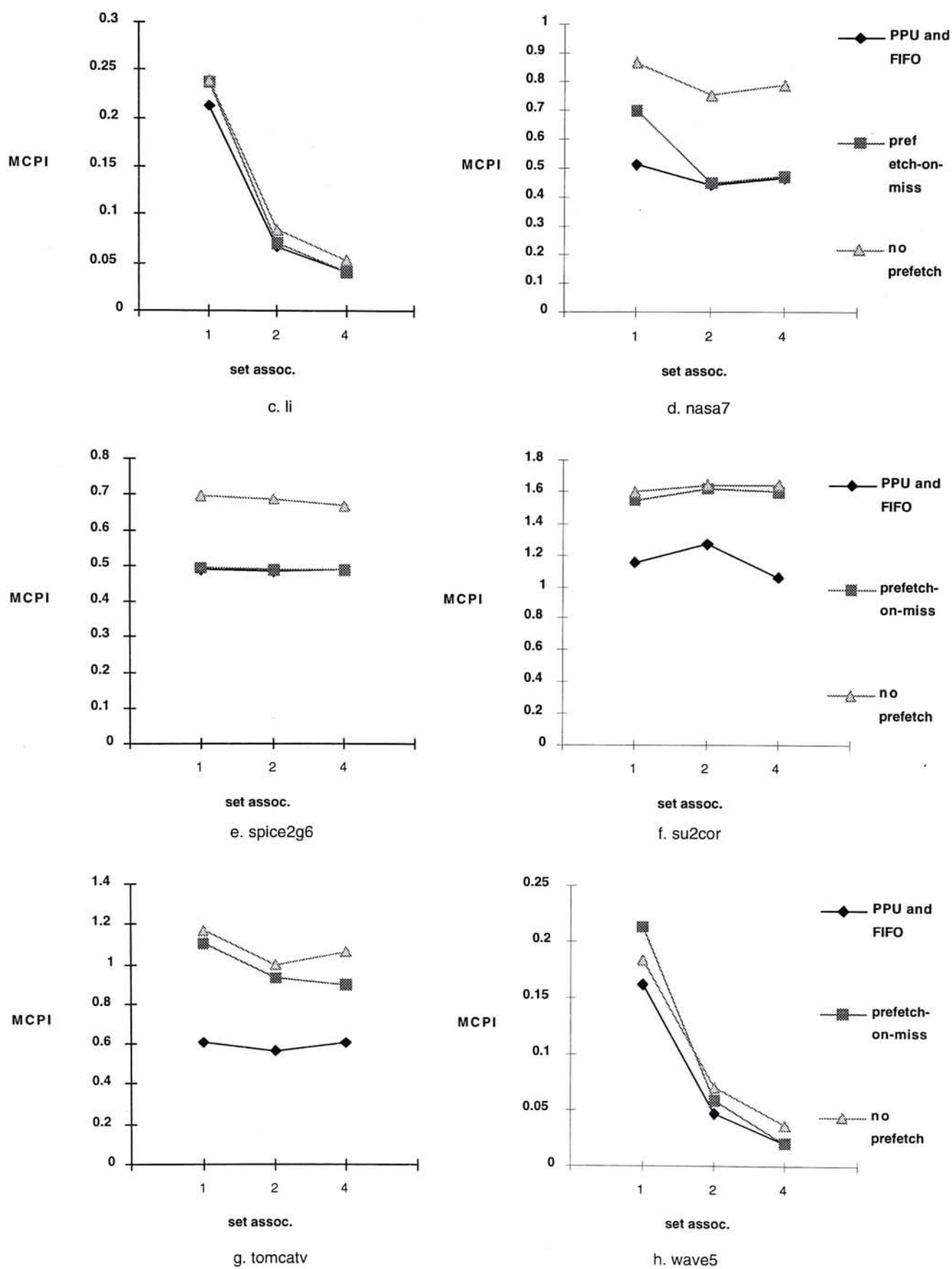
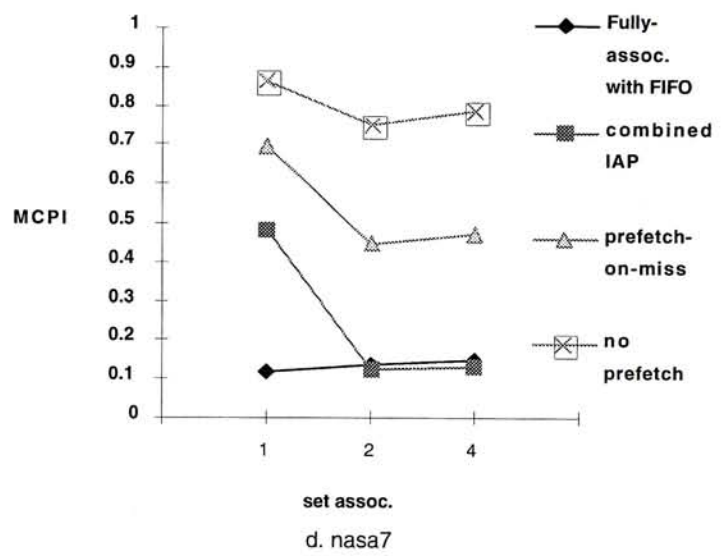
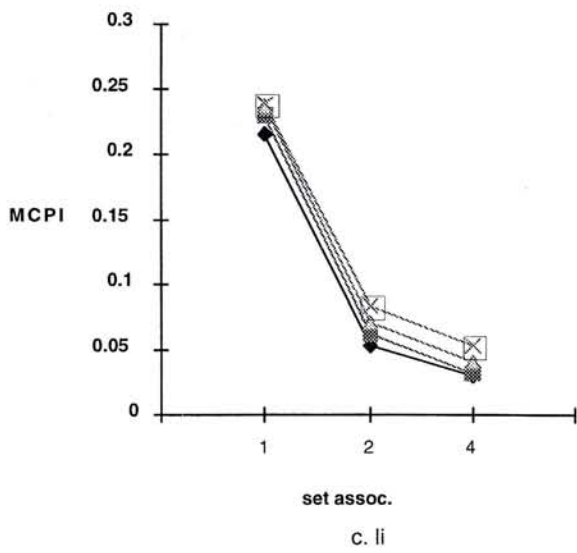
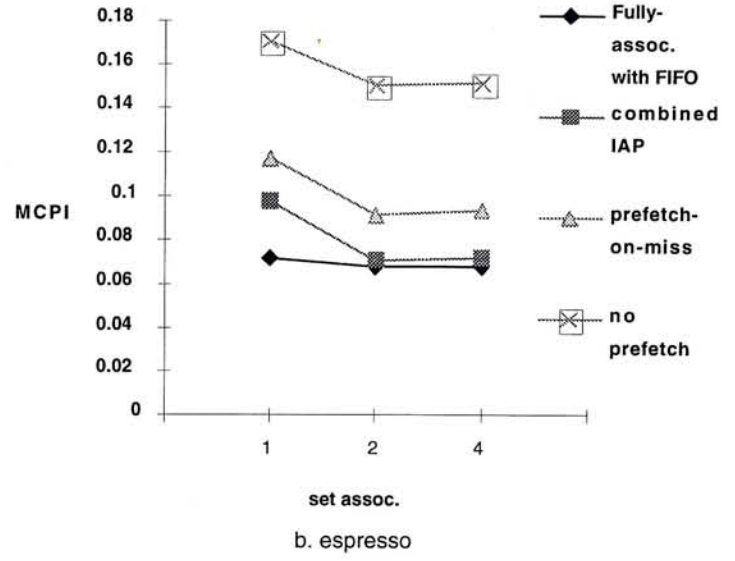
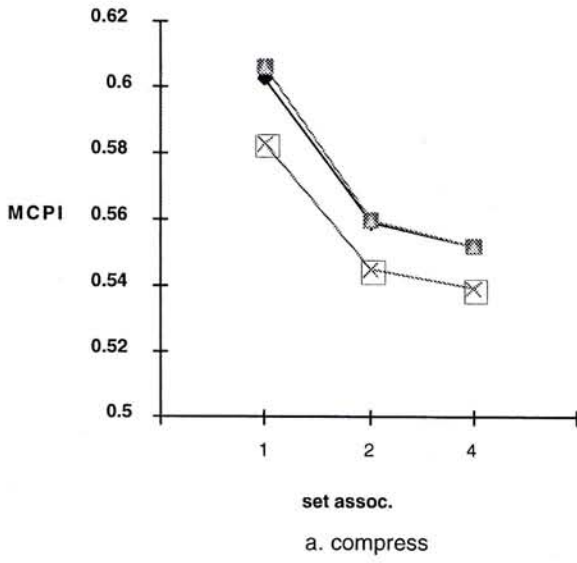


Figure A.11: MCPI by varying set associative in PPUVC with prefetch-on-miss scheme

A.3.3 Prefetch Cache



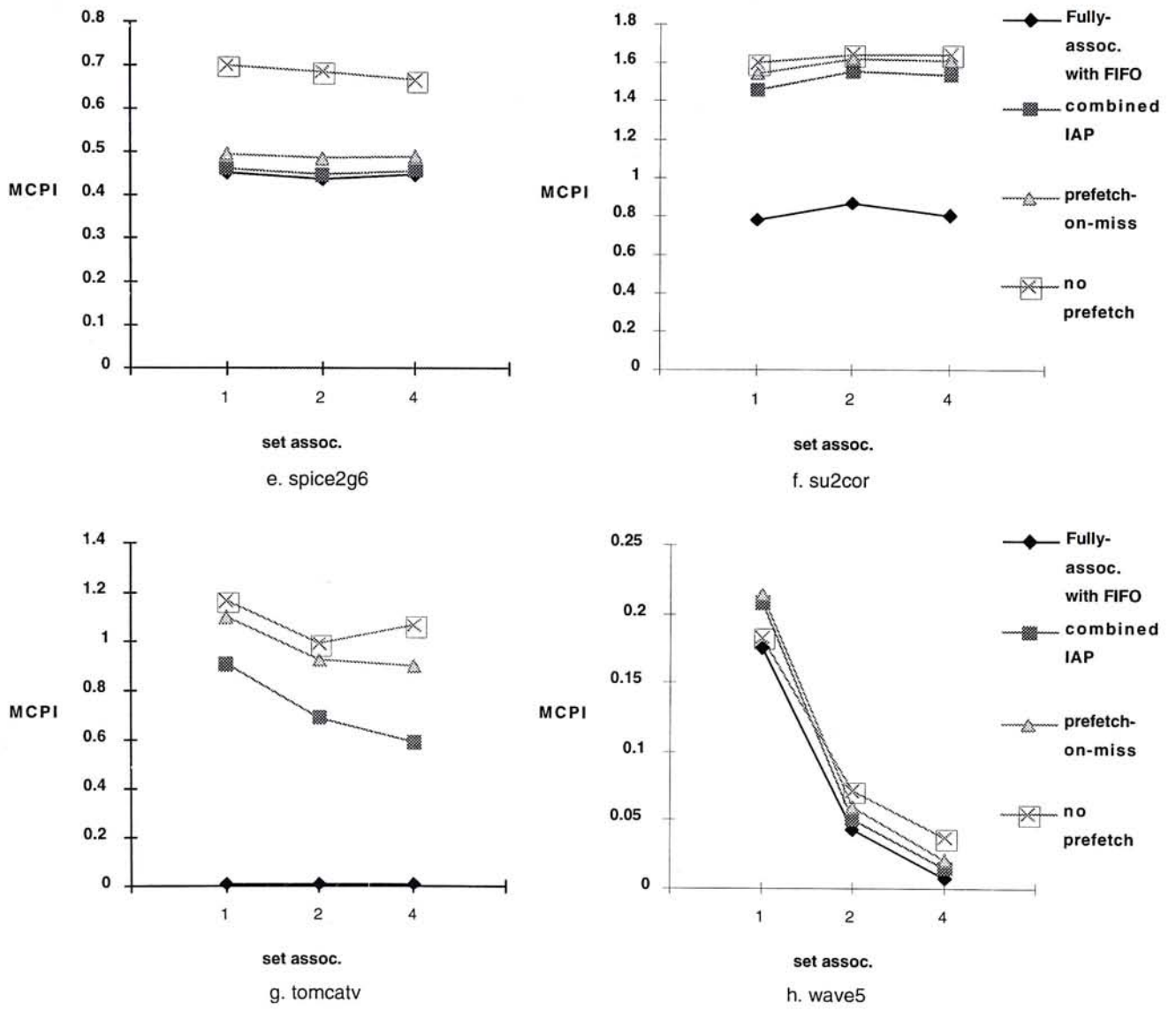


Figure A.12: MCPI by varying set associative in prefetch cache scheme

Appendix B

Simulation Results of IZ Replacement Policy

B.1 Memory Delay Time Reduction

B.1.1 Varying Cache Size

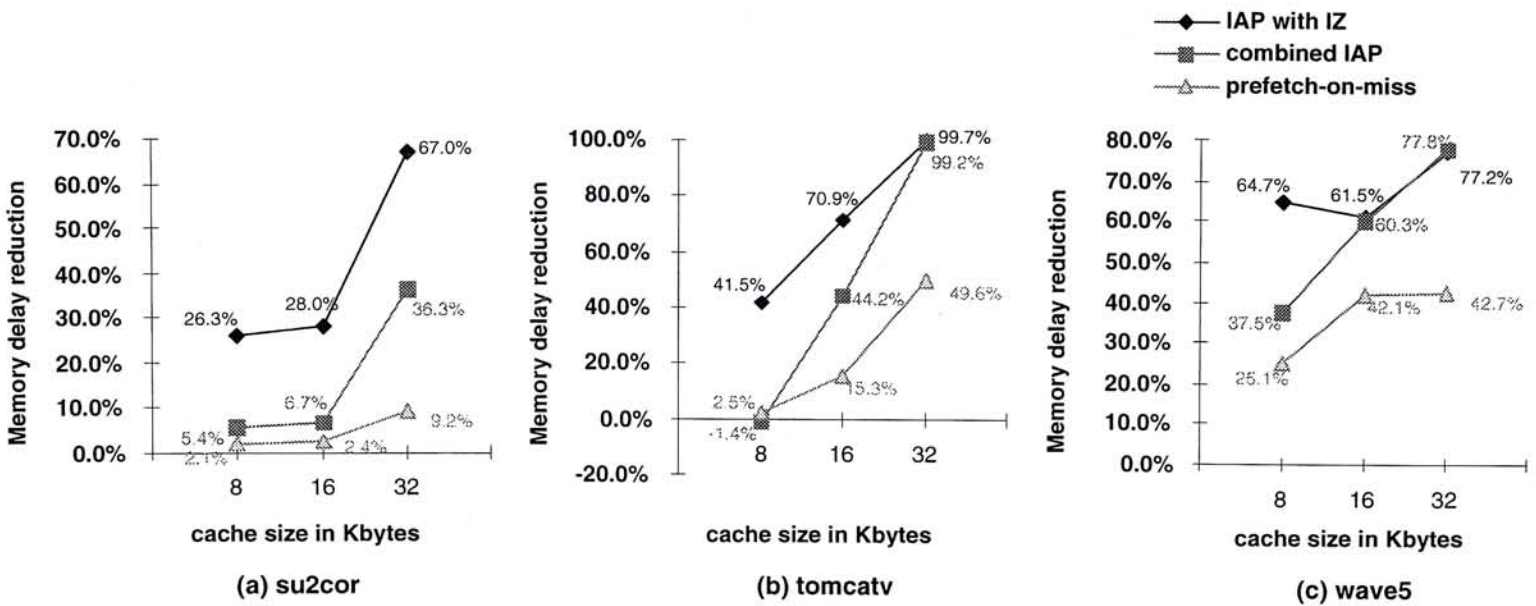


Figure B.1: Results of the first group programs in IZ

Appendix B Simulation Results of IZ Replacement Policy

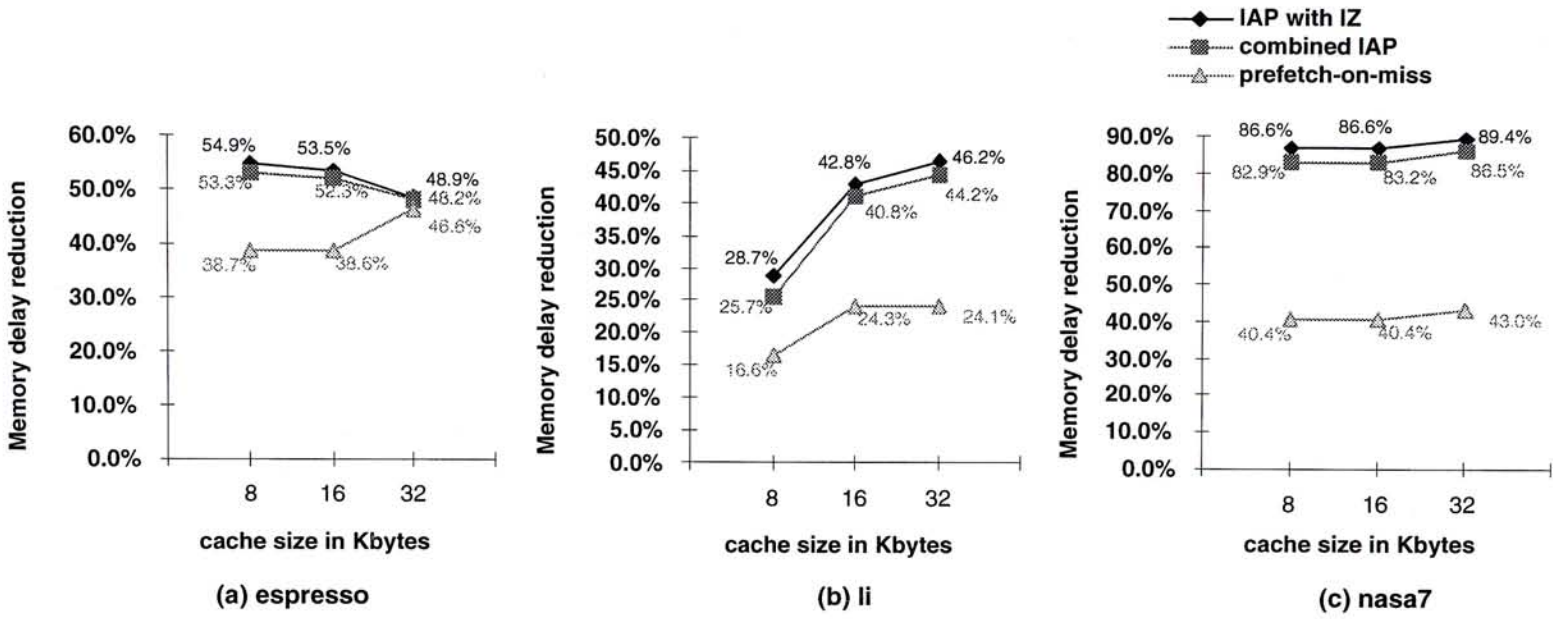


Figure B.2: Results of the second group programs in IZ

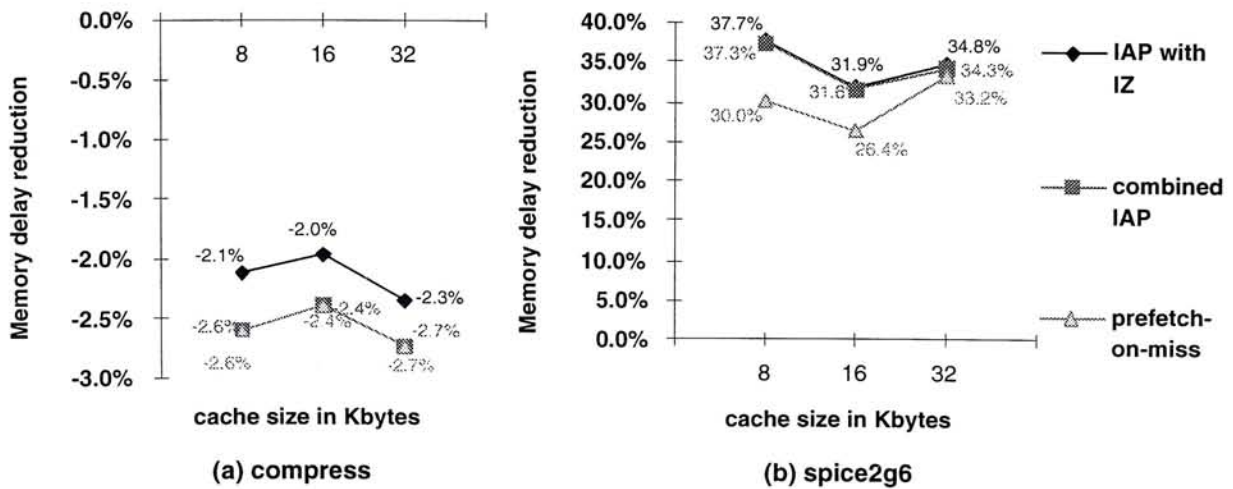


Figure B.3: Results of the third group programs in IZ

B.1.2 Varying Cache Line Size

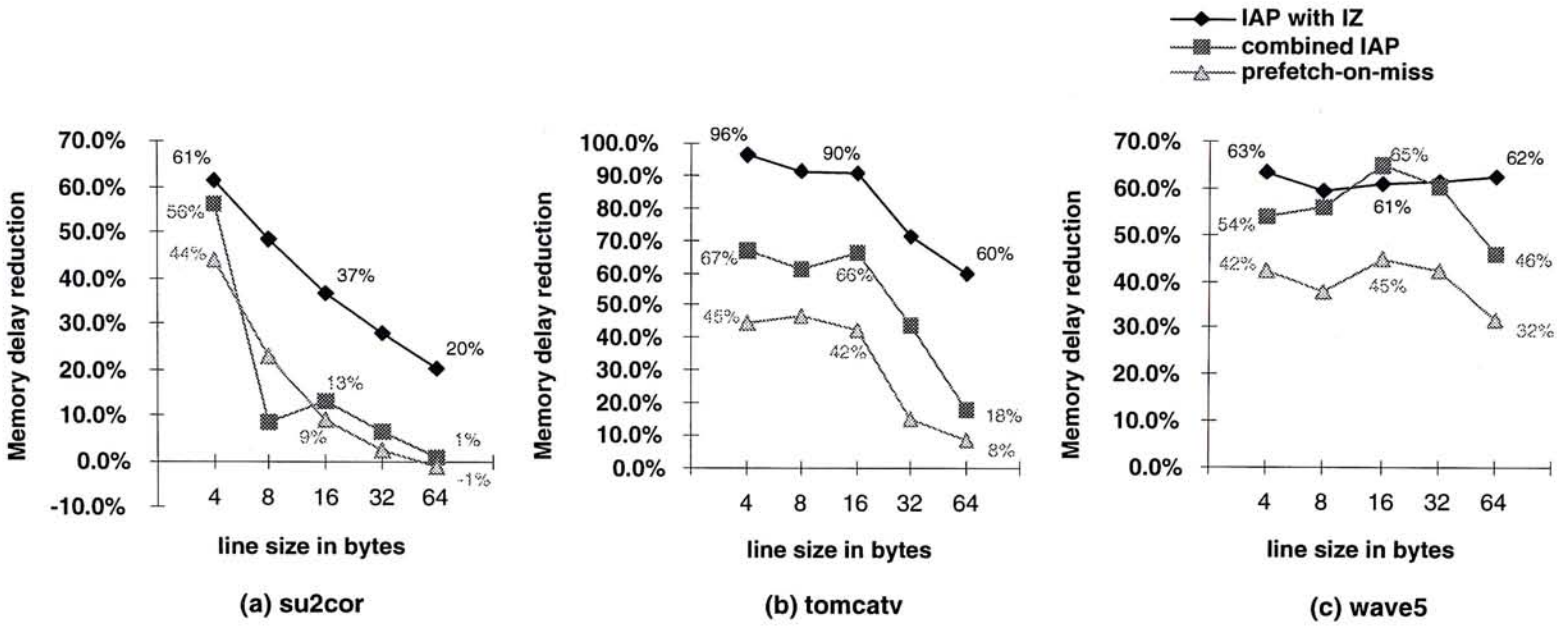


Figure B.4: Results of the first group programs in IZ

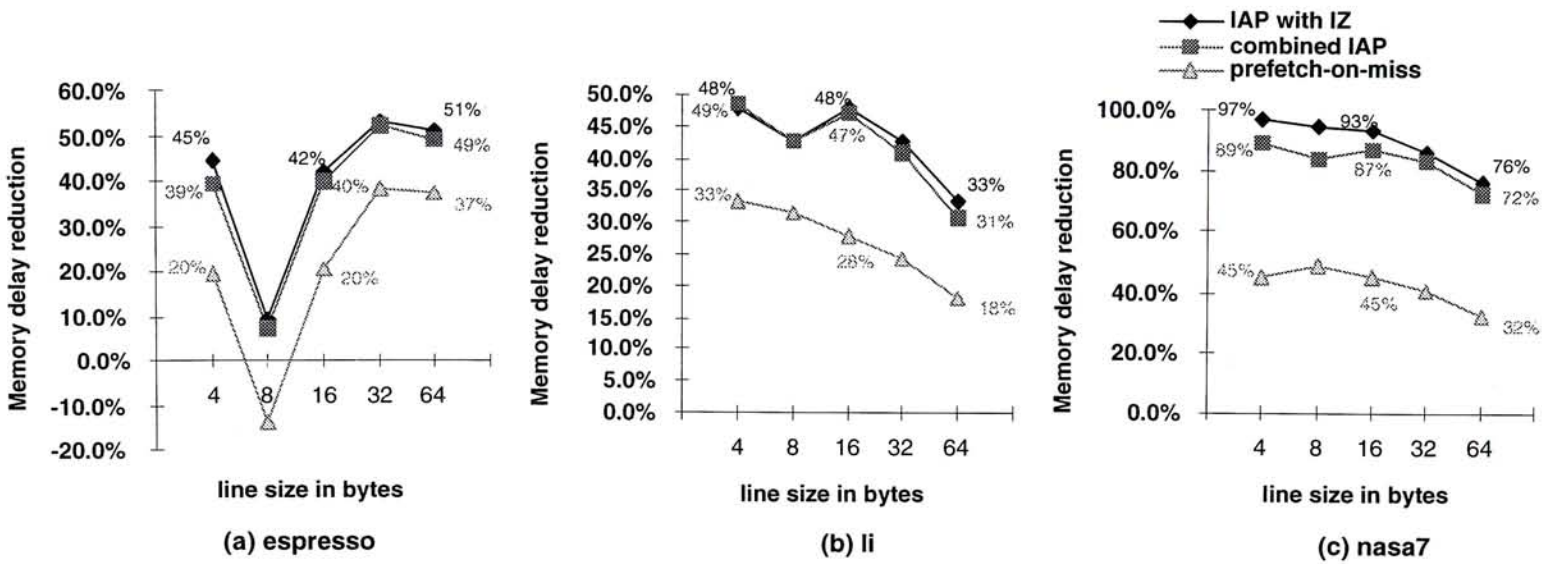


Figure B.5: Results of the second group programs in IZ

Appendix B Simulation Results of IZ Replacement Policy

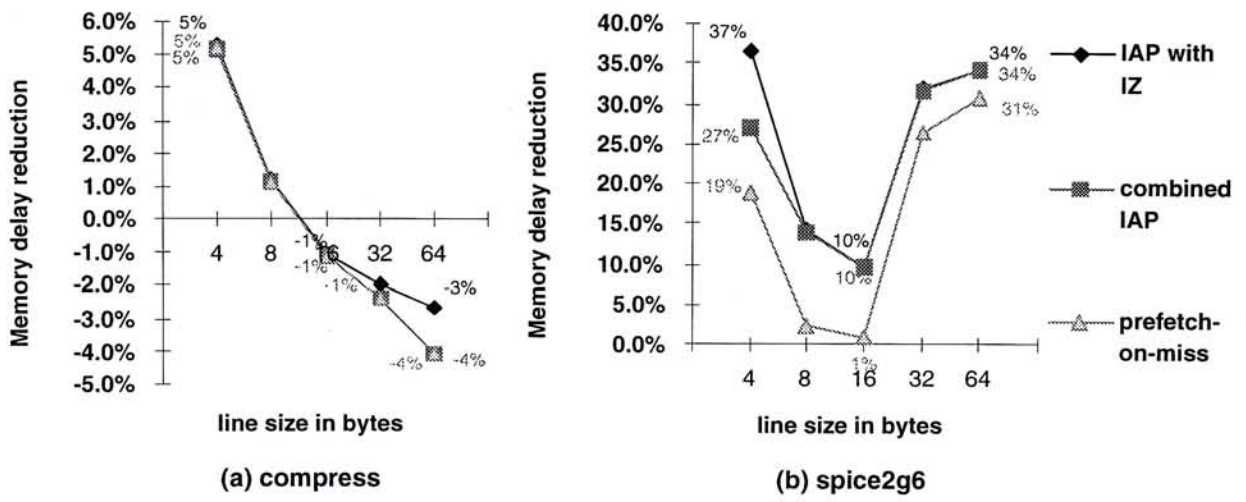


Figure B.6: Results of the third group programs in IZ

B.1.3 Varying Cache Set Associative

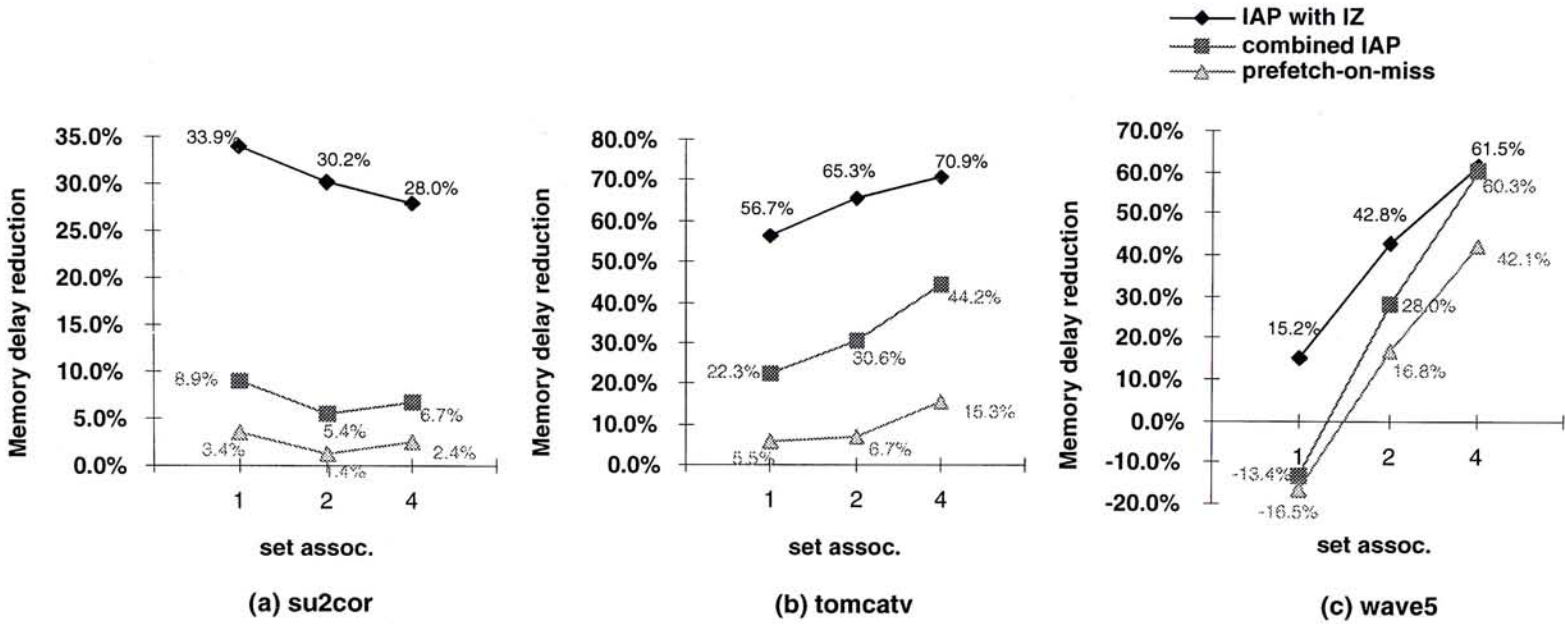


Figure B.7: Results of the first group programs in IZ

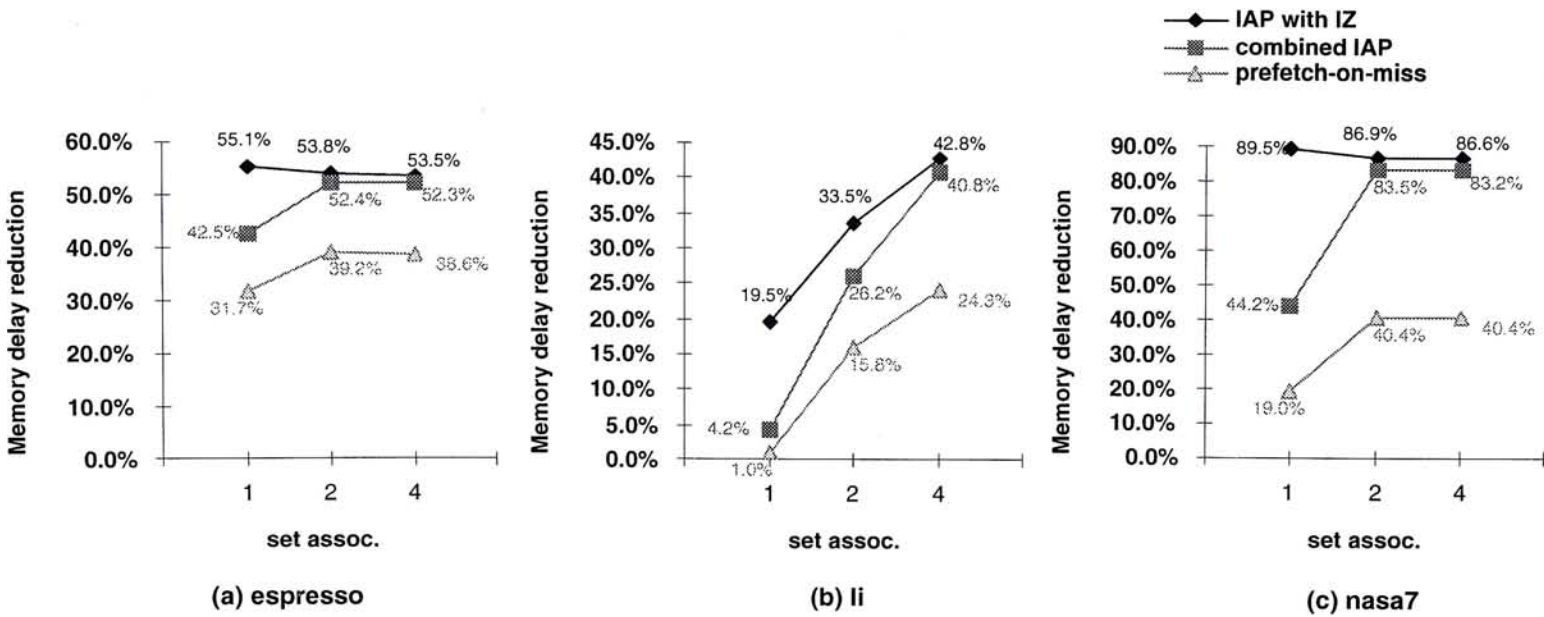
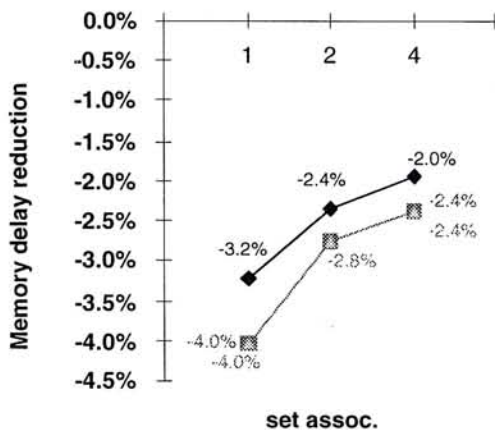
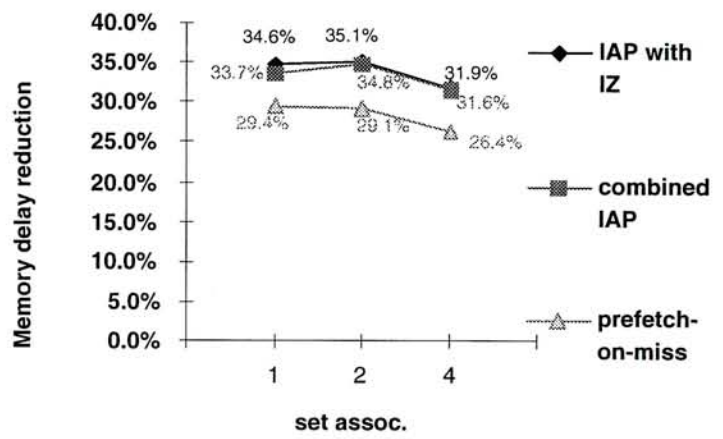


Figure B.8: Results of the second group programs in IZ

Appendix B Simulation Results of IZ Replacement Policy



(a) compress



(b) spice2g6

Figure B.9: Results of the third group programs in IZ

Appendix C

Simulation Results of Priority Pre-Updating with Victim Cache

C.1 PPUVC in IAP Scheme

C.1.1 Memory Delay Time Reduction

Varying Cache Size

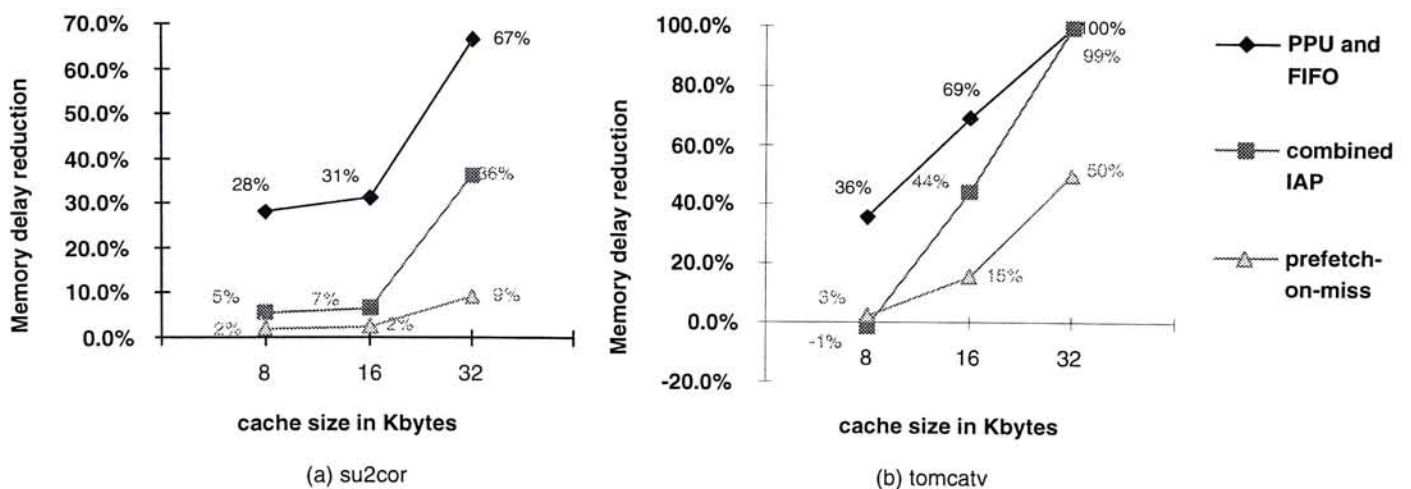


Figure C.1: Results of the first group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

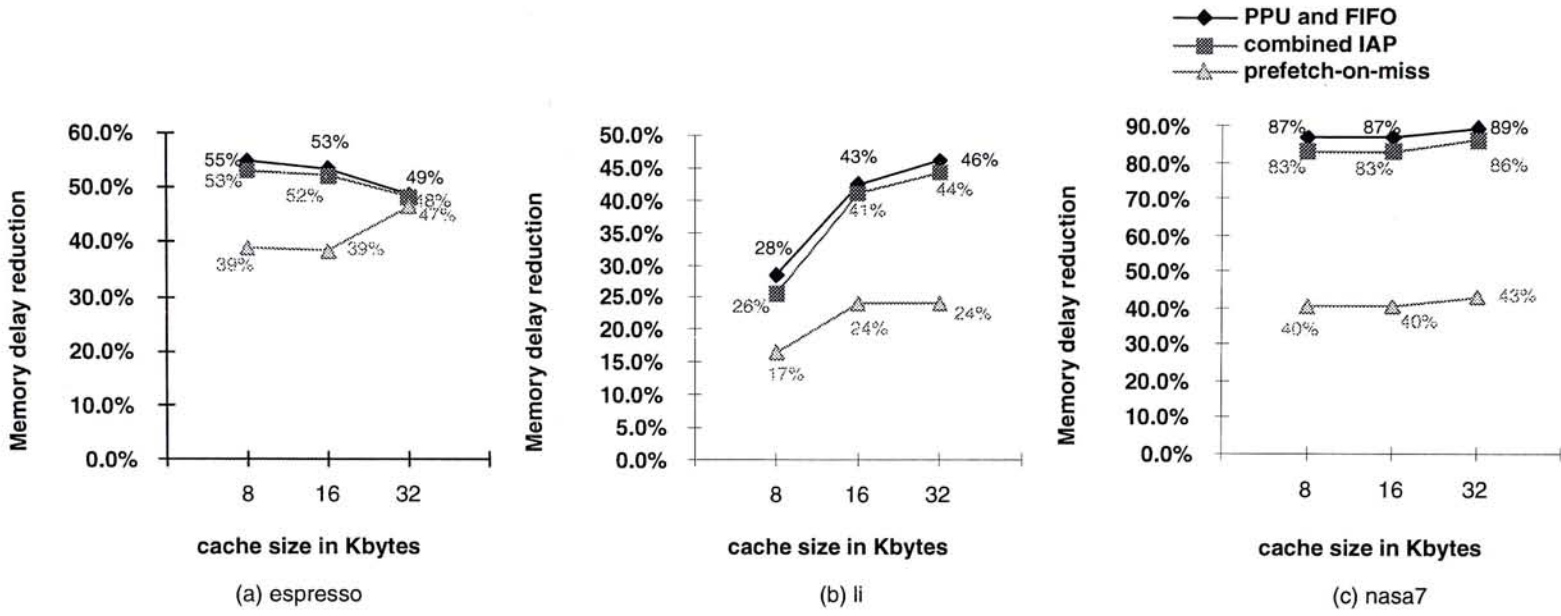


Figure C.2: Results of the second group programs in PPUVC

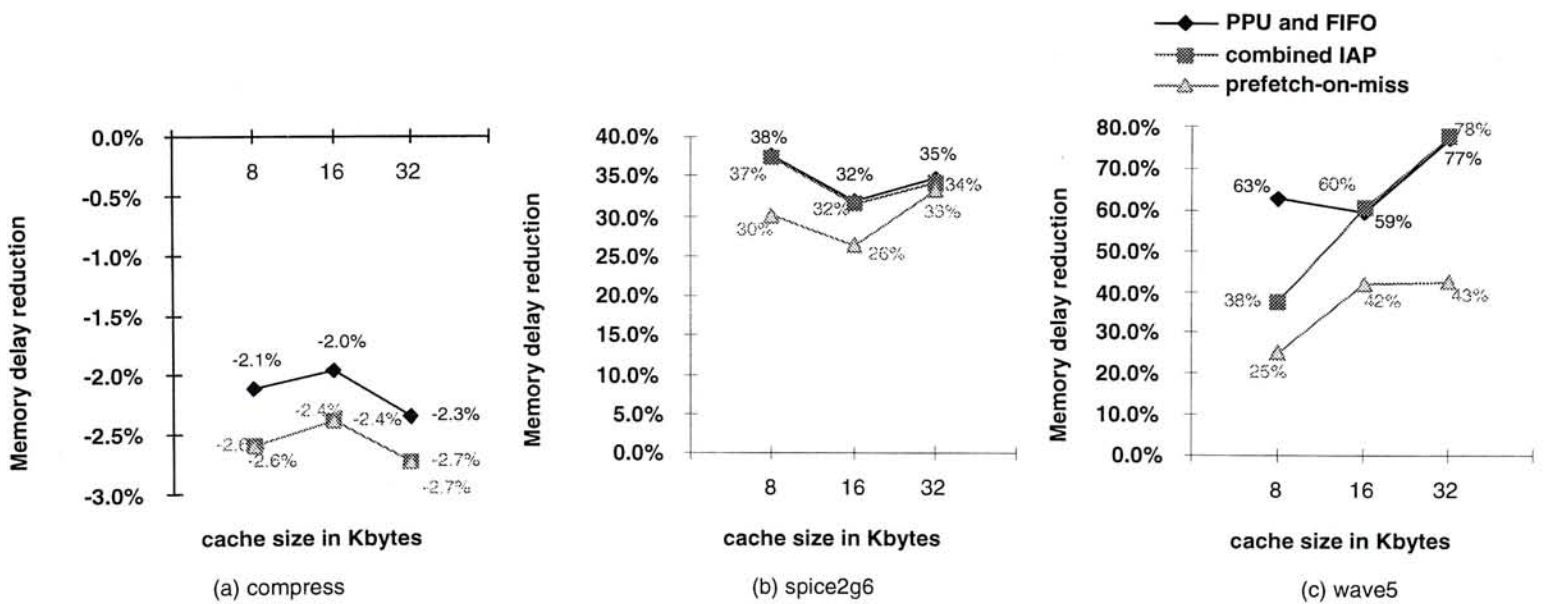


Figure C.3: Results of the third group programs in PPUVC

Varying Cache Line Size

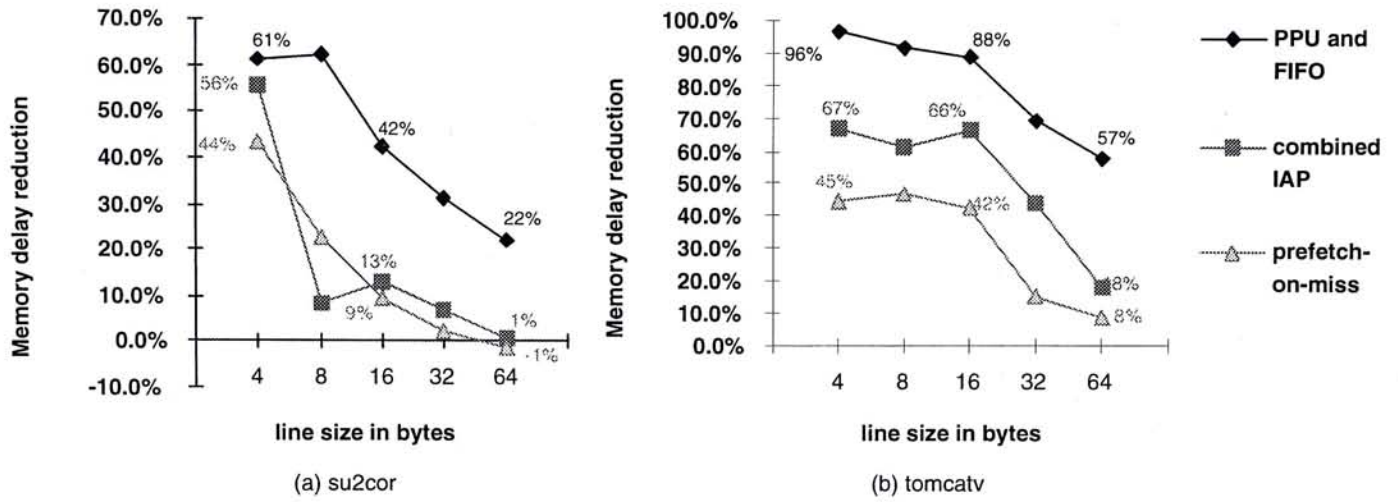


Figure C.4: Varying line size

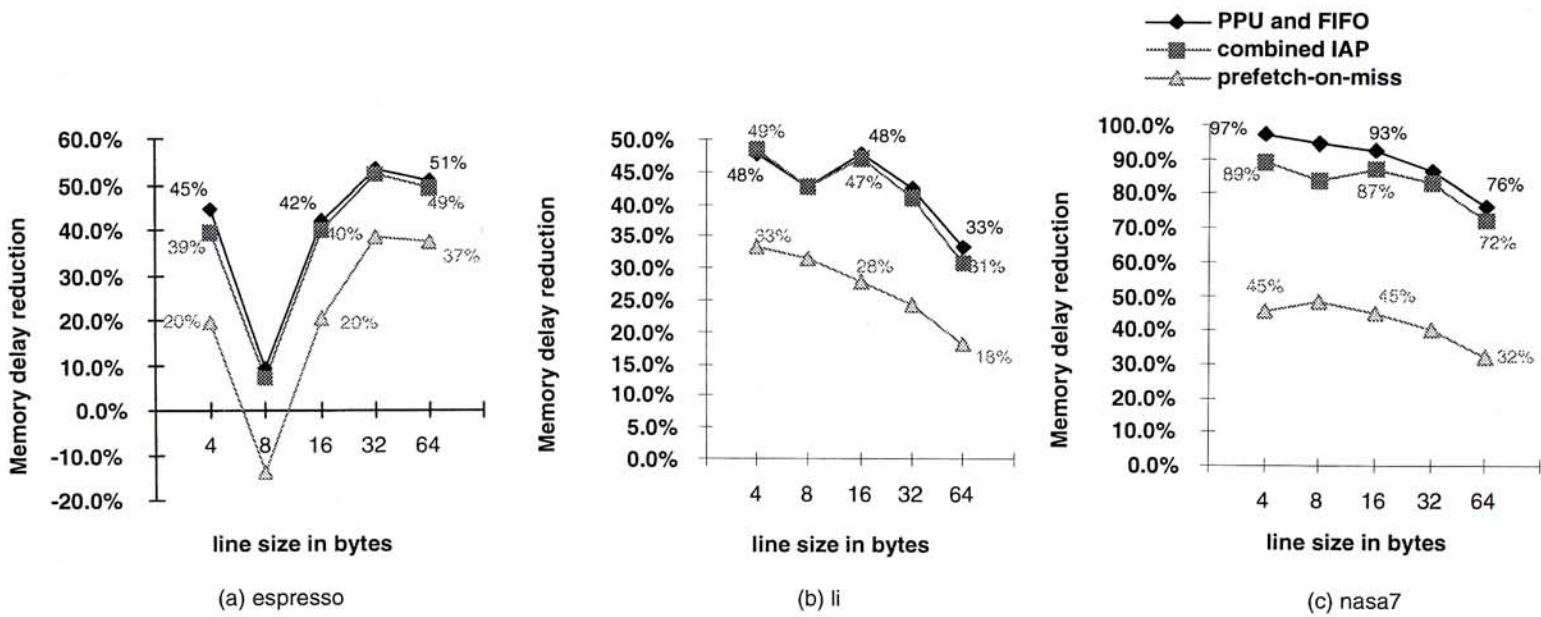


Figure C.5: Results of the second group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

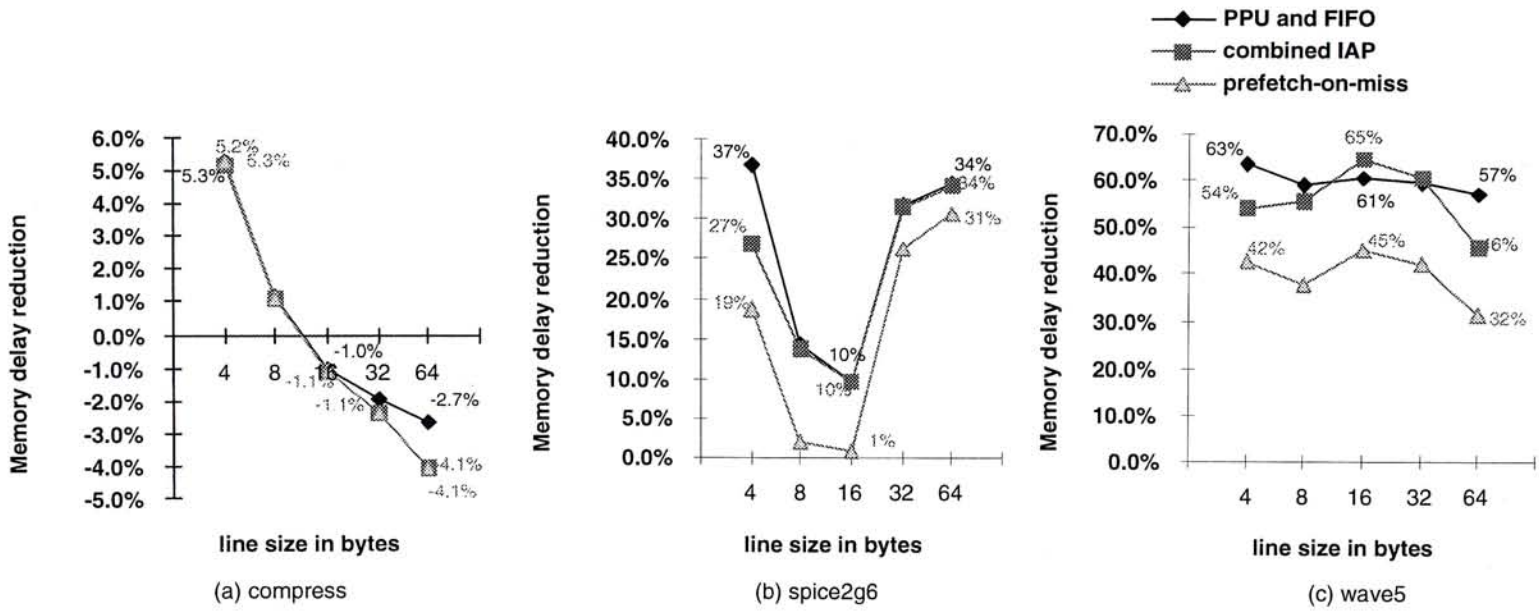


Figure C.6: Results of the third group programs in PPUVC

Varying Set Associative

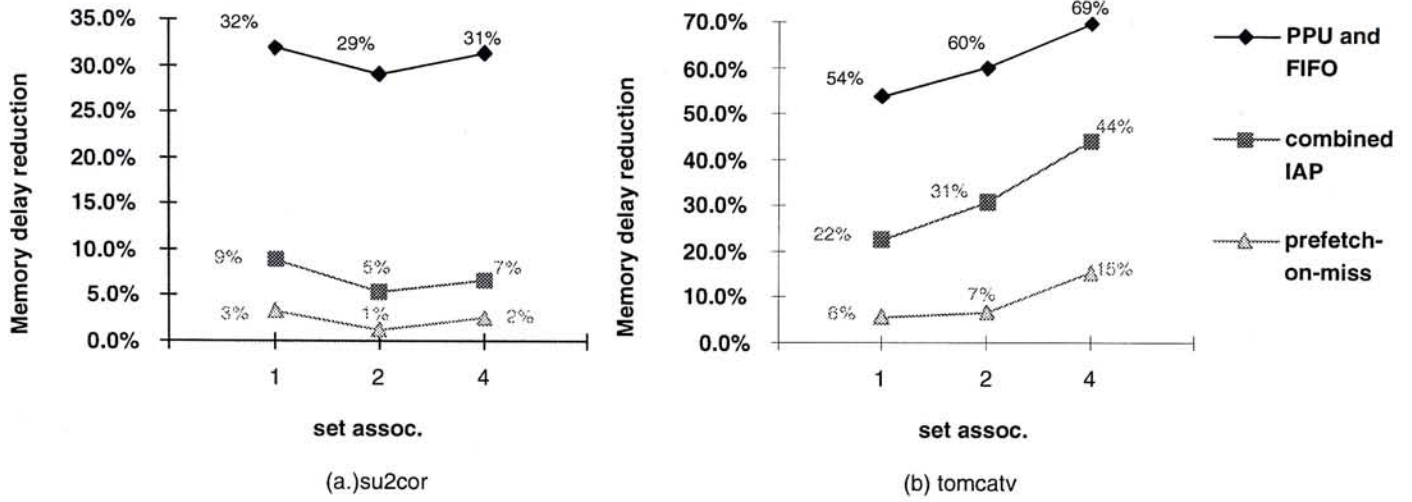


Figure C.7: Varying set associative

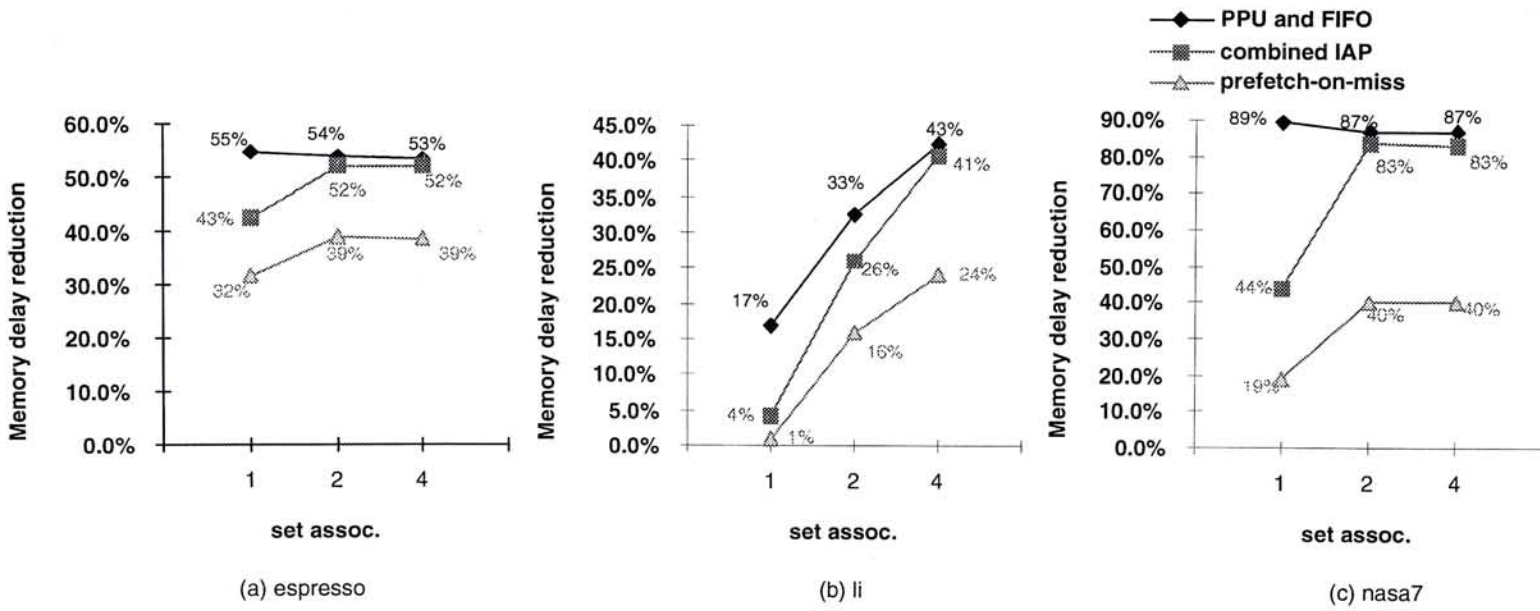


Figure C.8: Results of the second group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

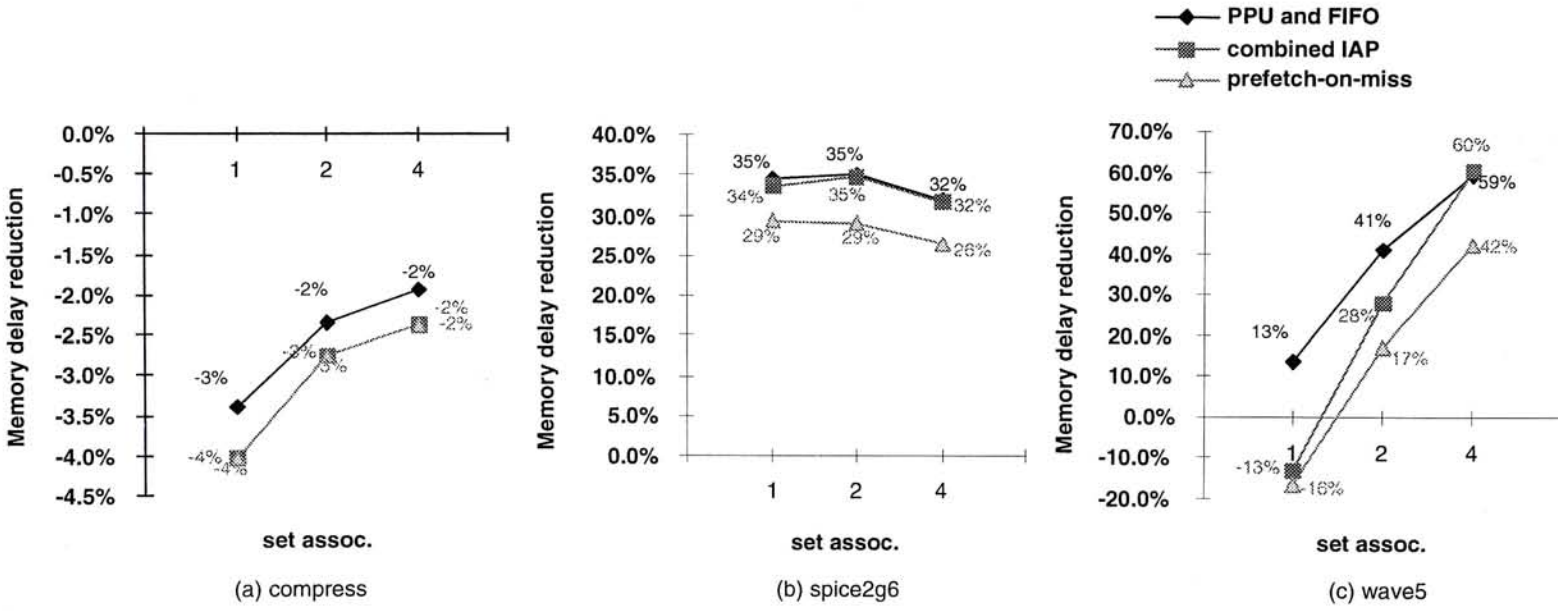


Figure C.9: Results of the third group programs in PPUVC

C.2 PPUVC in Cache with Prefetch-On-Miss Only

C.2.1 Memory Delay Time Reduction

Varying Cache Size

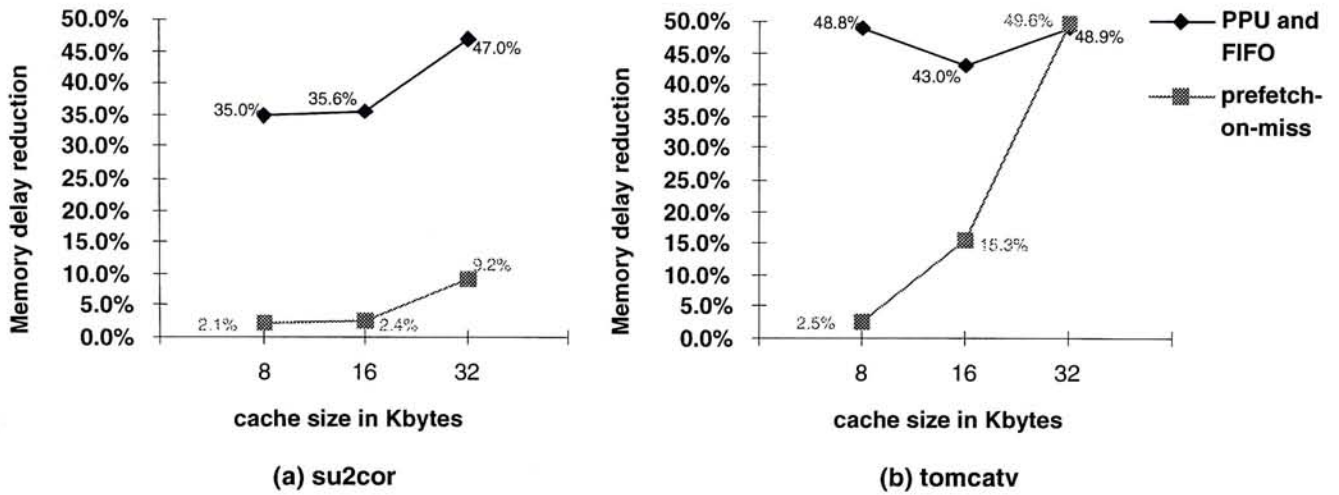


Figure C.10: Results of the first group programs in PPUVC

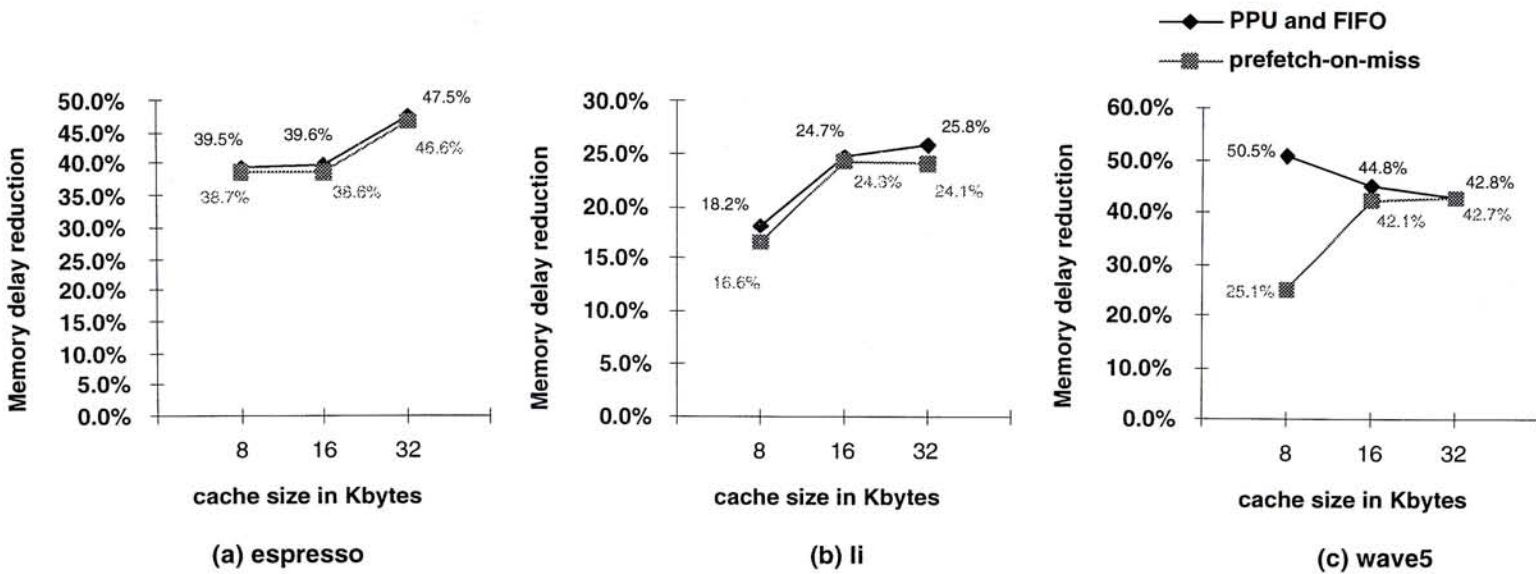


Figure C.11: Results of the second group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

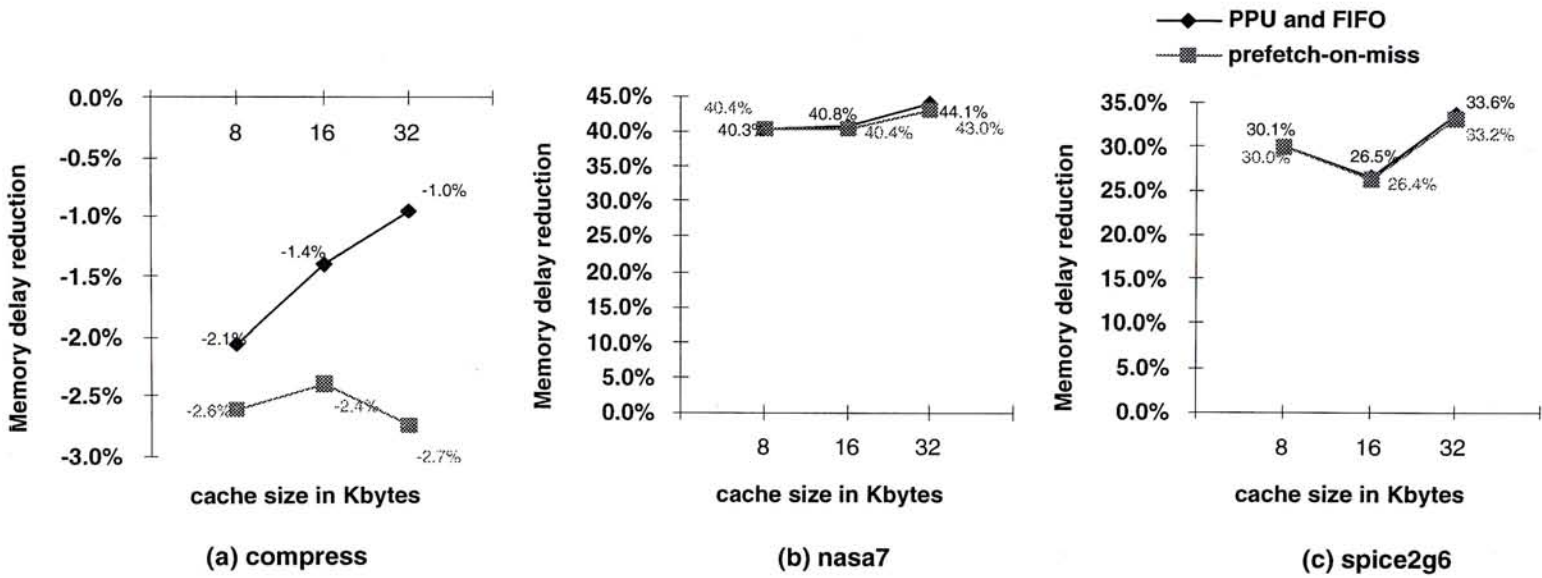


Figure C.12: Results of the third group programs in PPUVC

Varying Cache Line Size

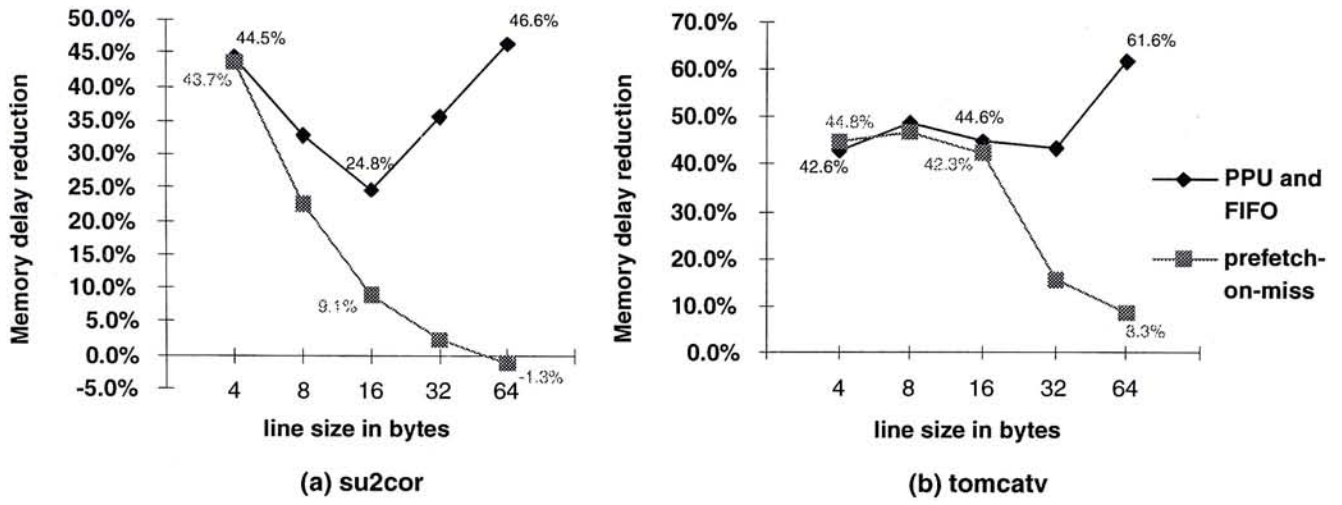


Figure C.13: Results of the first group programs in PPUVC

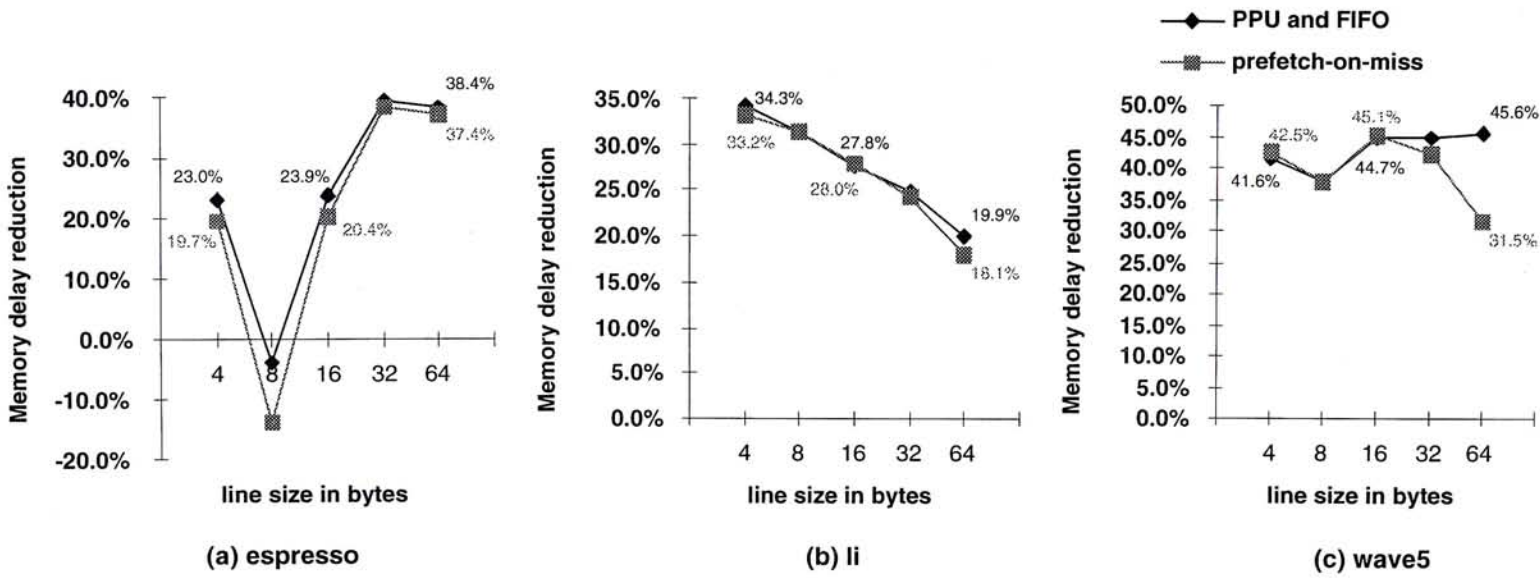


Figure C.14: Results of the second group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

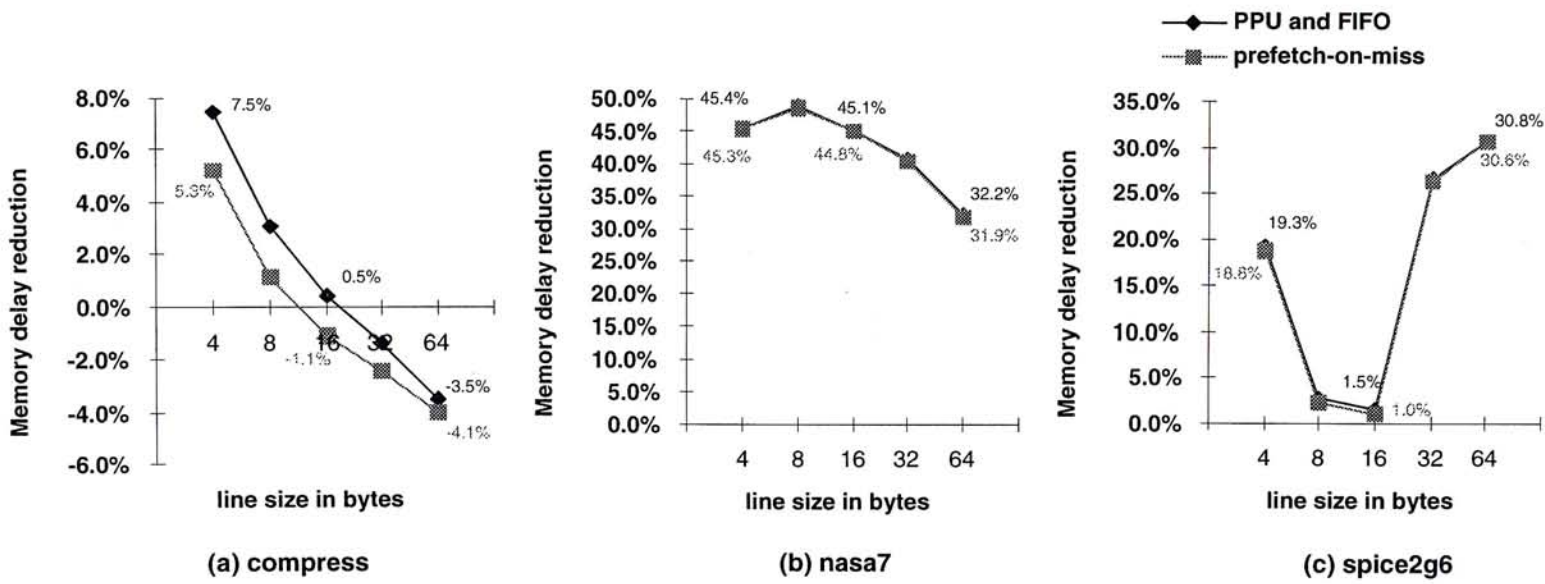


Figure C.15: Results of the third group programs in PPUVC

Varying Set Associative

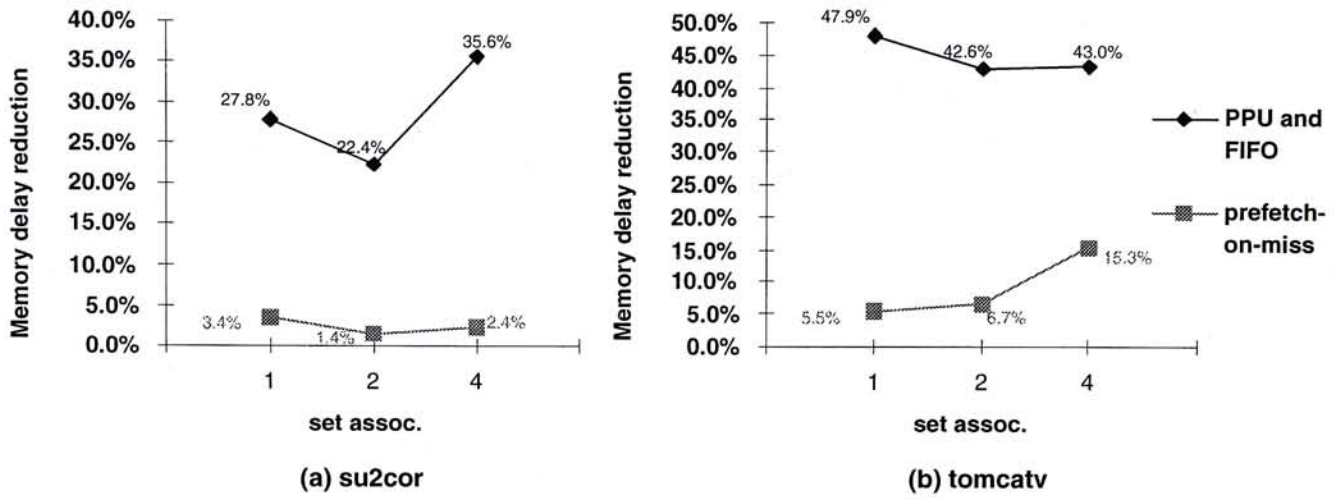


Figure C.16: Results of the first group programs in PPUVC

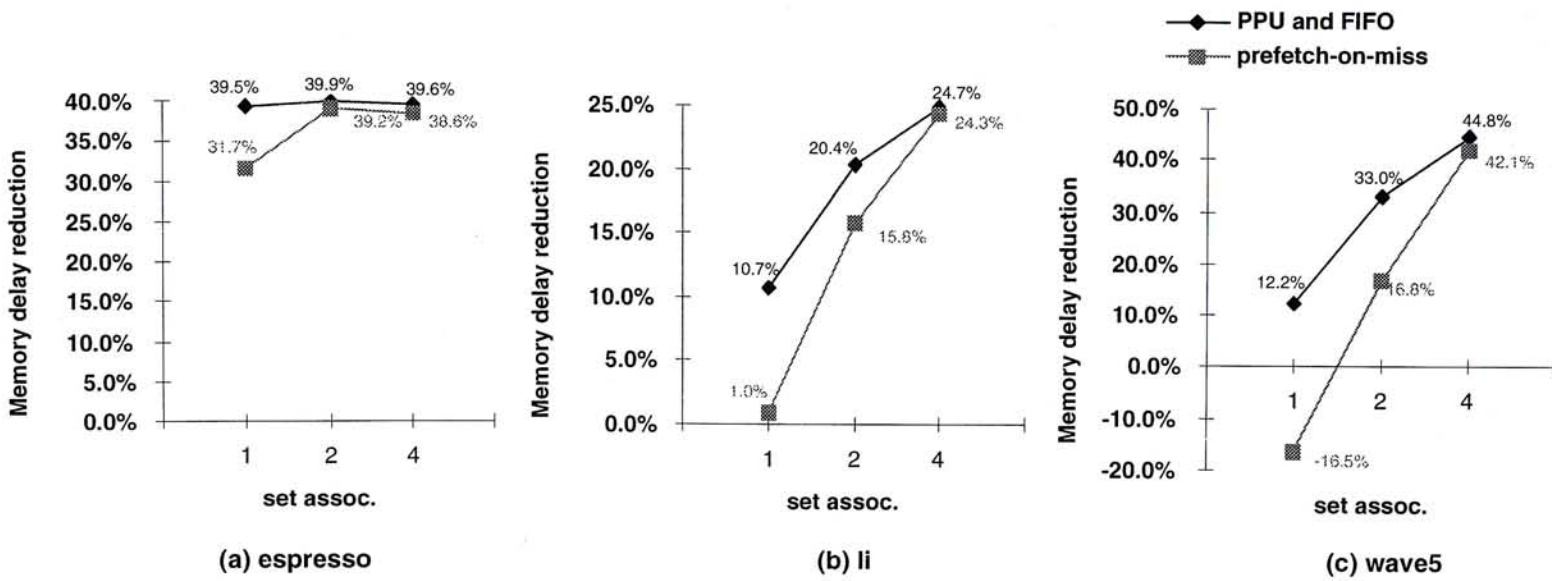


Figure C.17: Results of the second group programs in PPUVC

Appendix C Simulation Results of Priority Pre-Updating with Victim Cache

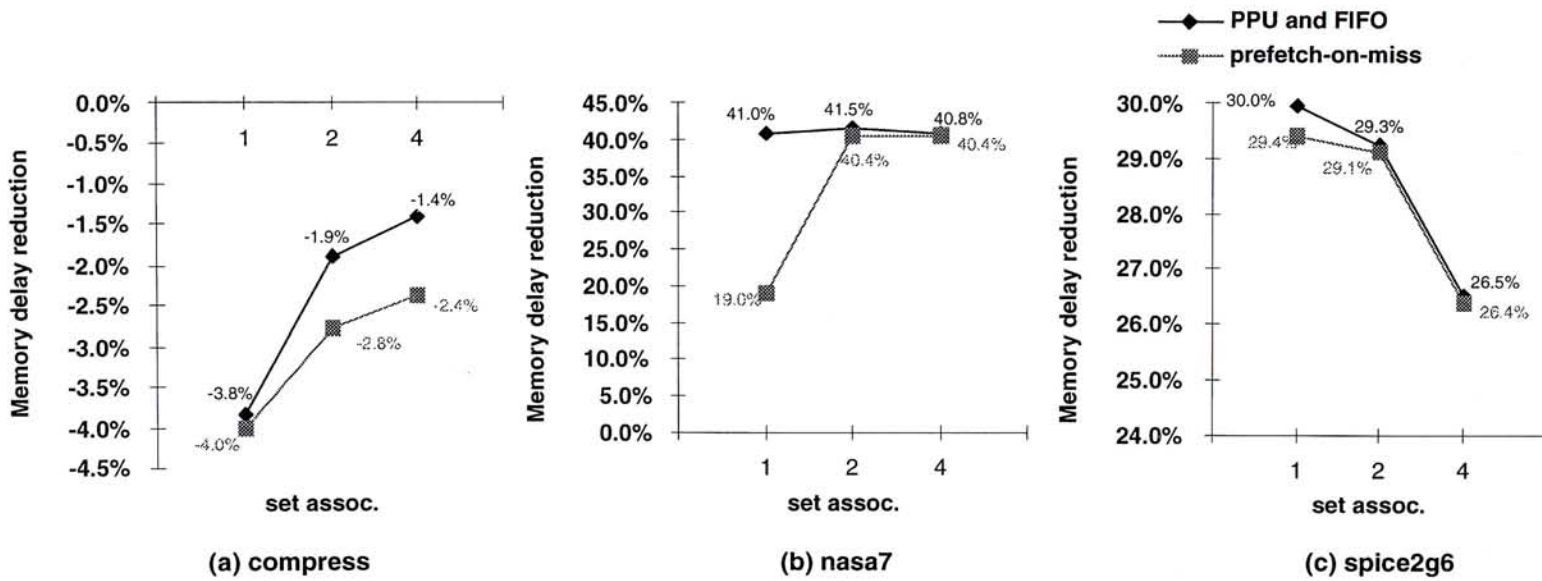


Figure C.18: Results of the third group programs in PPUVC

Appendix D

Simulation Results of Prefetch Cache

D.1 Memory Delay Time Reduction

D.1.1 Varying Cache Size

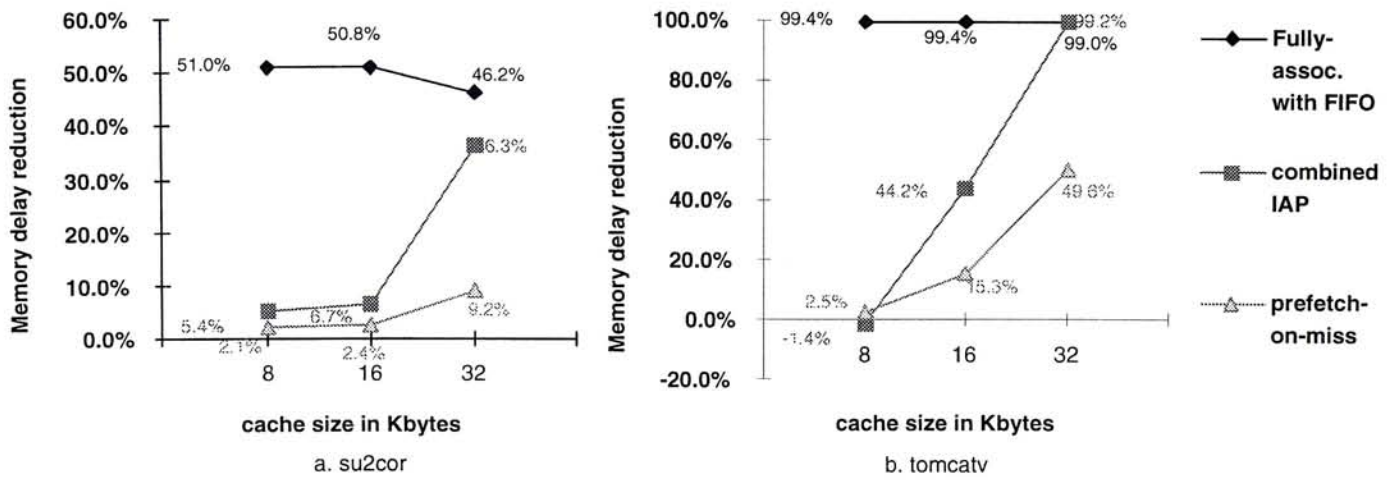


Figure D.1: Results of the first group programs

Appendix D Simulation Results of Prefetch Cache

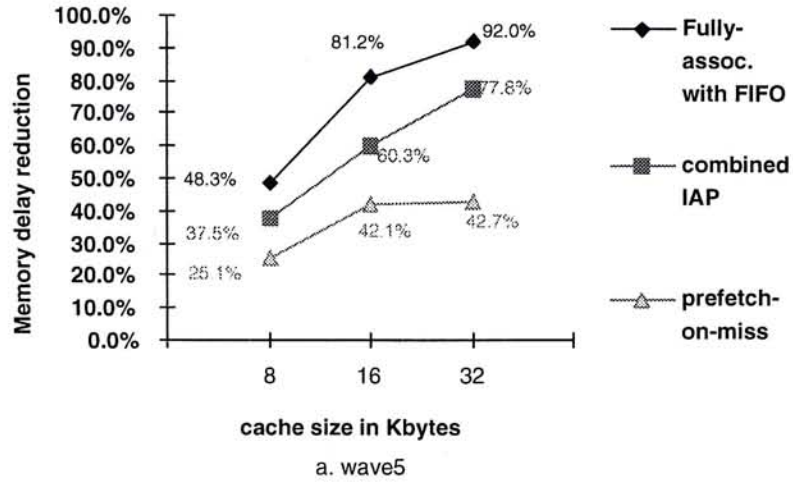


Figure D.2: Results of the second group programs

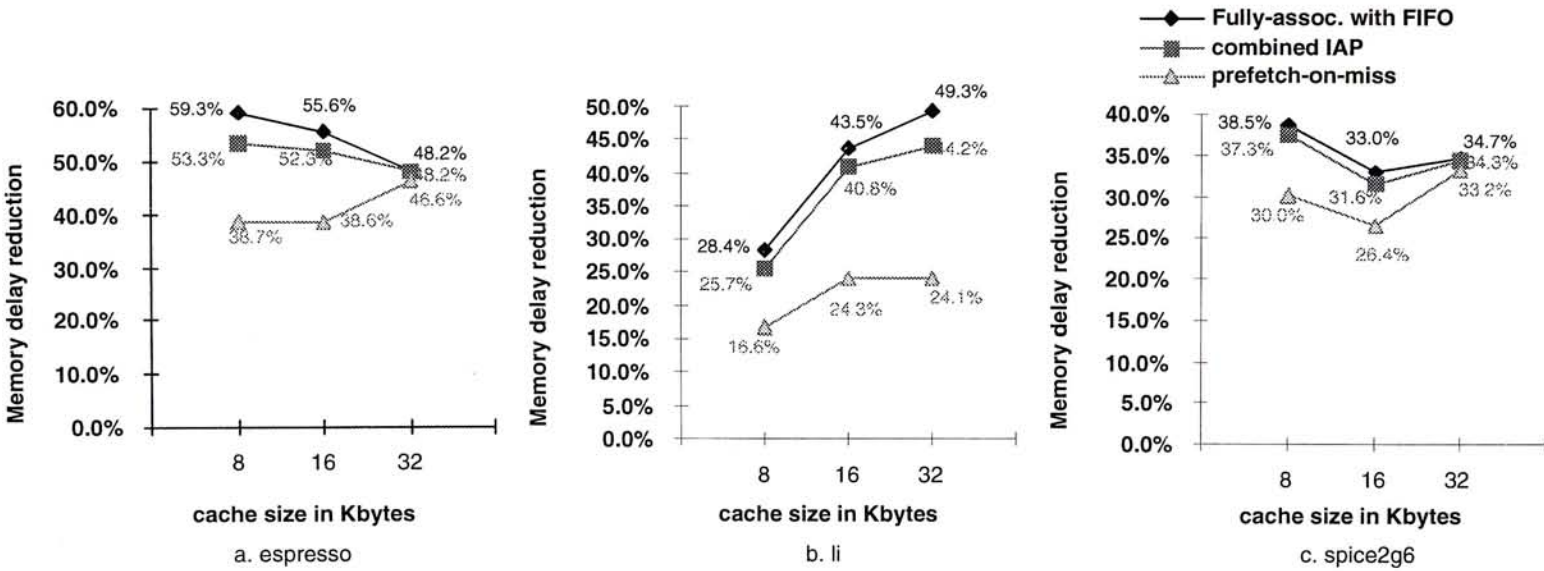


Figure D.3: Results of the third group programs

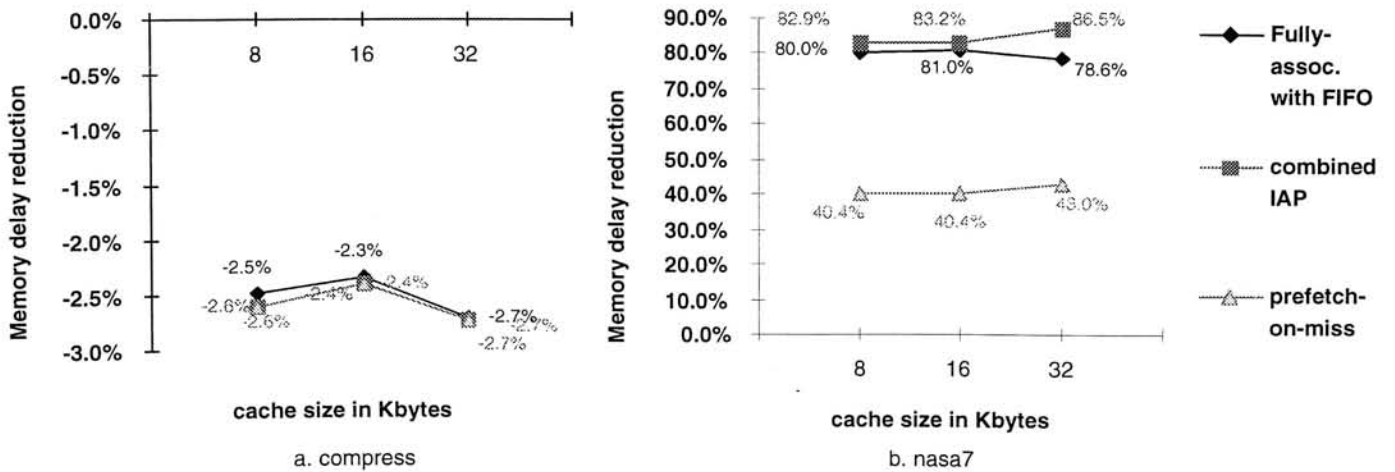


Figure D.4: Results of the fourth group programs

D.1.2 Varying Cache Line Size

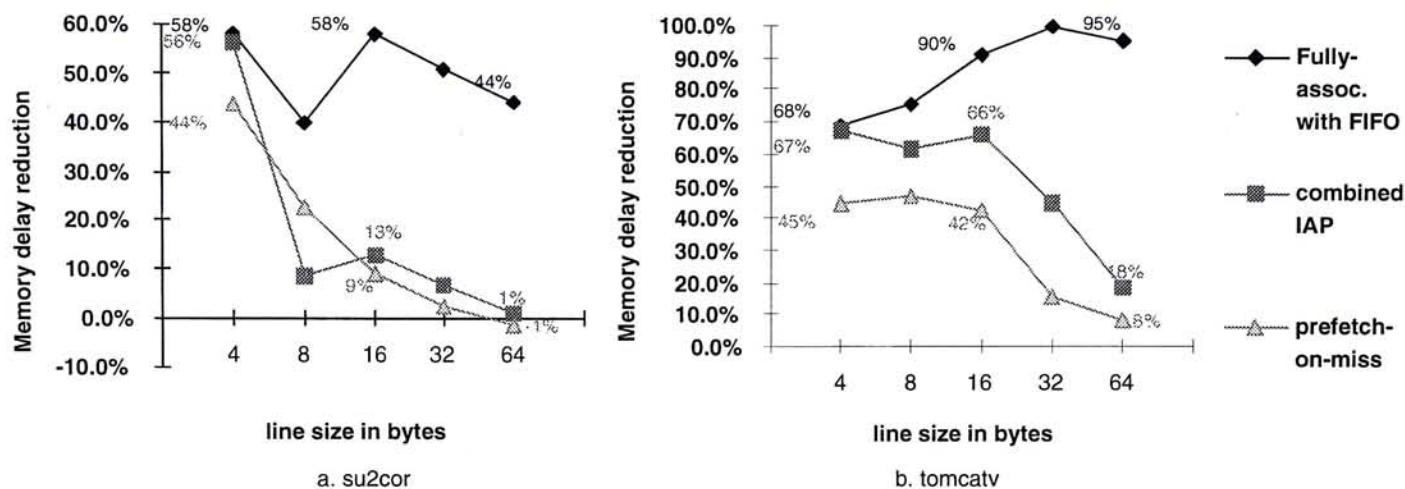


Figure D.5: Results of the first group programs

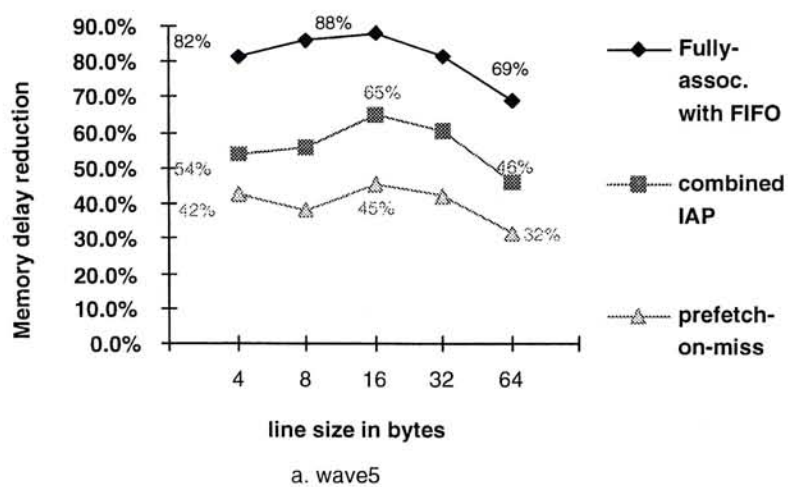


Figure D.6: Results of the second group programs

Appendix D Simulation Results of Prefetch Cache

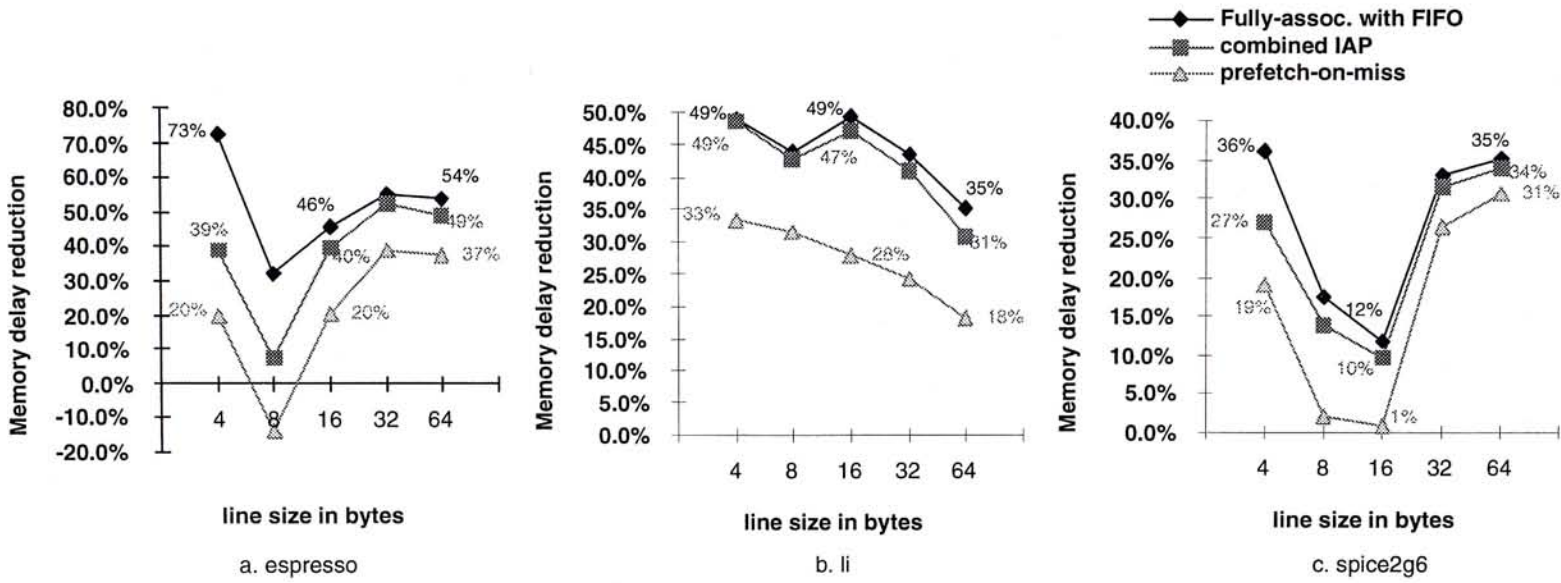


Figure D.7: Results of the third group programs

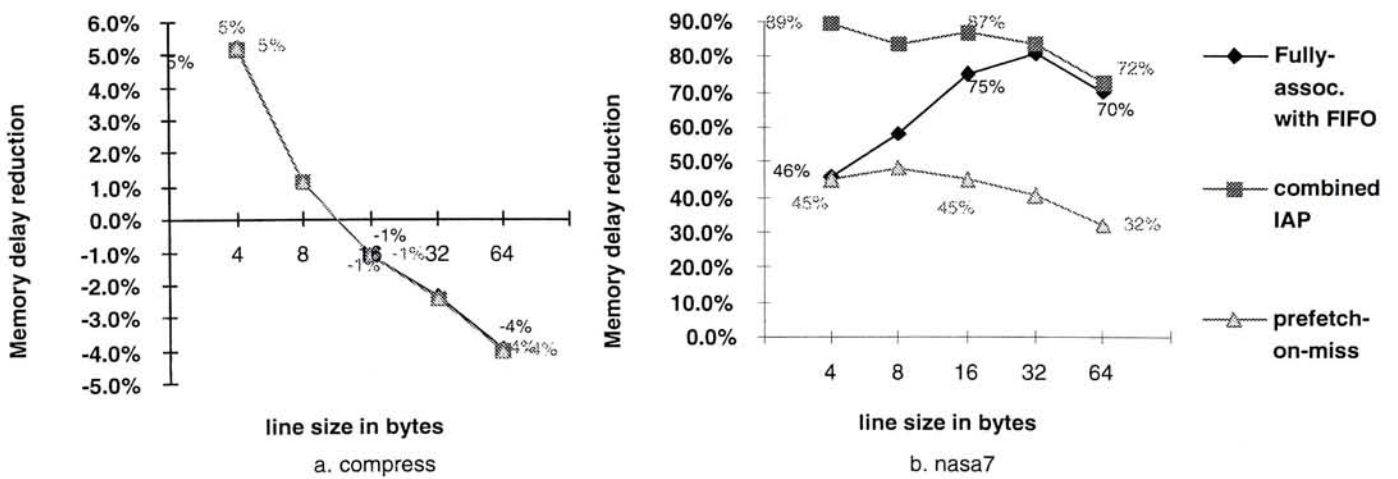


Figure D.8: Results of the fourth group programs

D.1.3 Varying Cache Set Associative

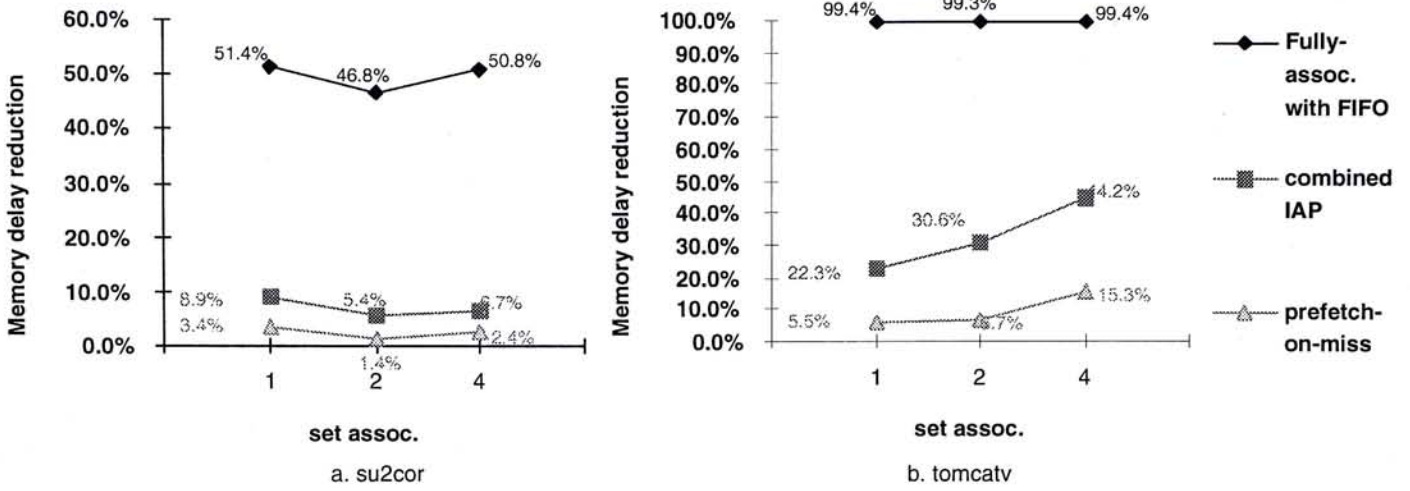


Figure D.9: Results of the first group programs

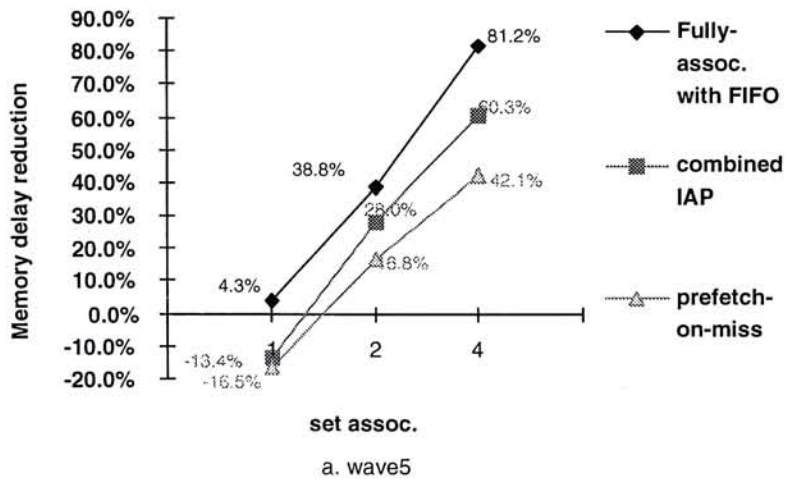


Figure D.10: Results of the second group programs

Appendix D Simulation Results of Prefetch Cache

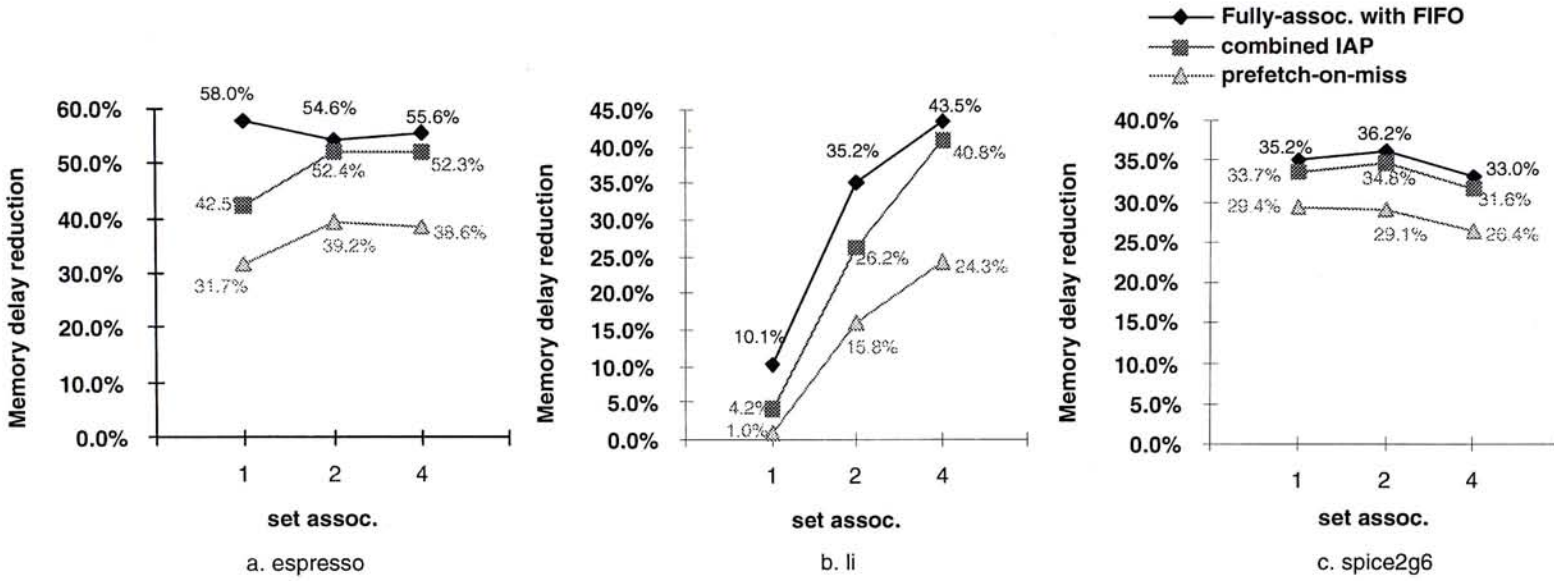


Figure D.11: Results of the third group programs

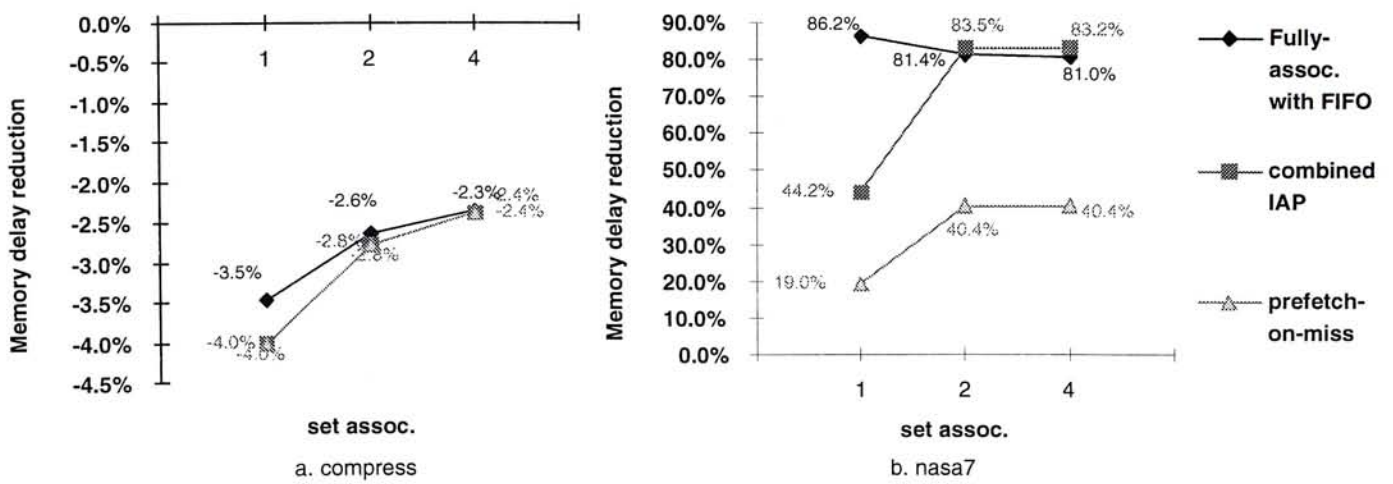


Figure D.12: Results of the fourth group programs

D.2 Results of the Three Replacement Policies

D.2.1 Varying Cache Size

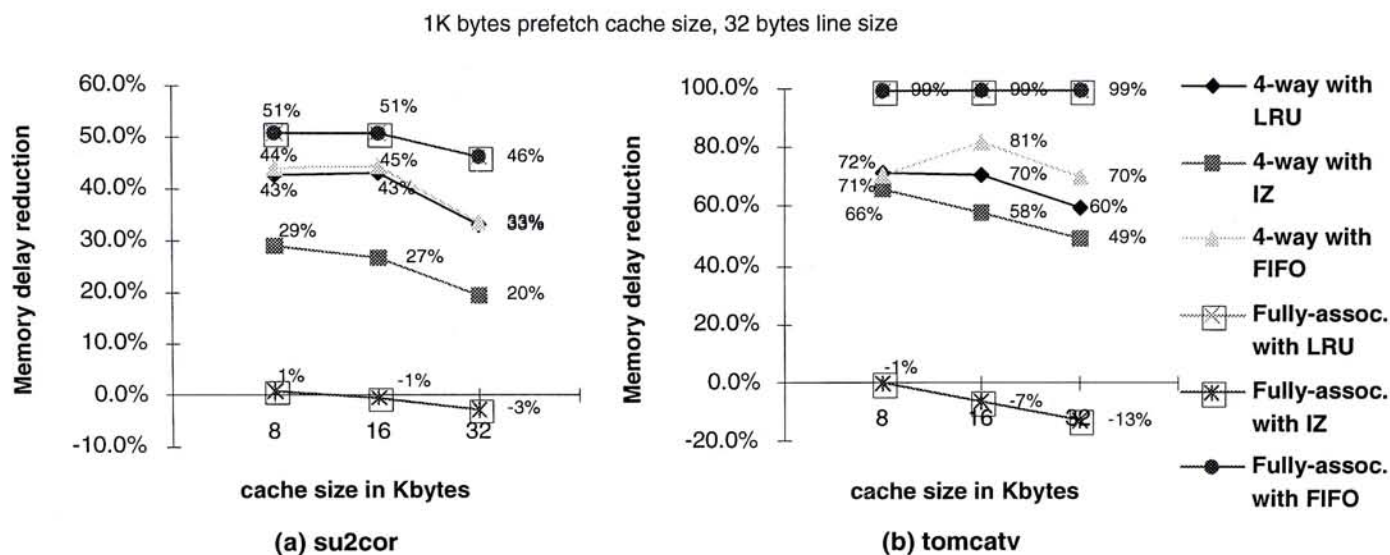


Figure D.13: Results of the first group programs

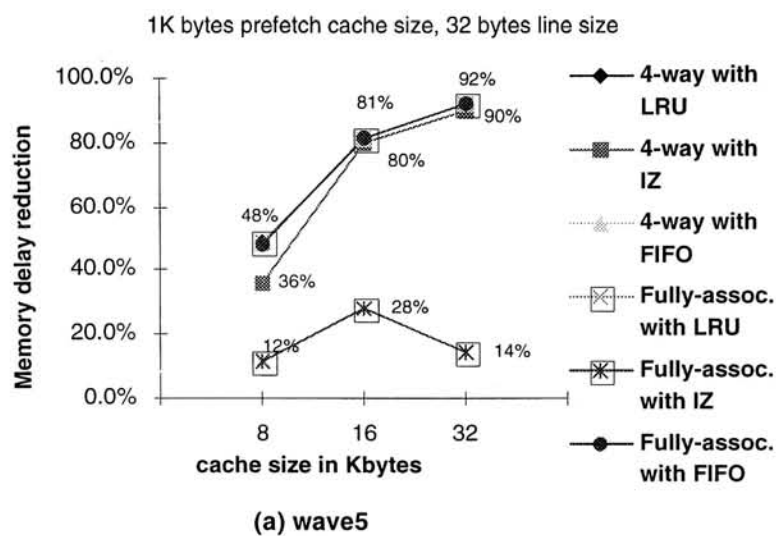


Figure D.14: Results of the second group programs

Appendix D Simulation Results of Prefetch Cache

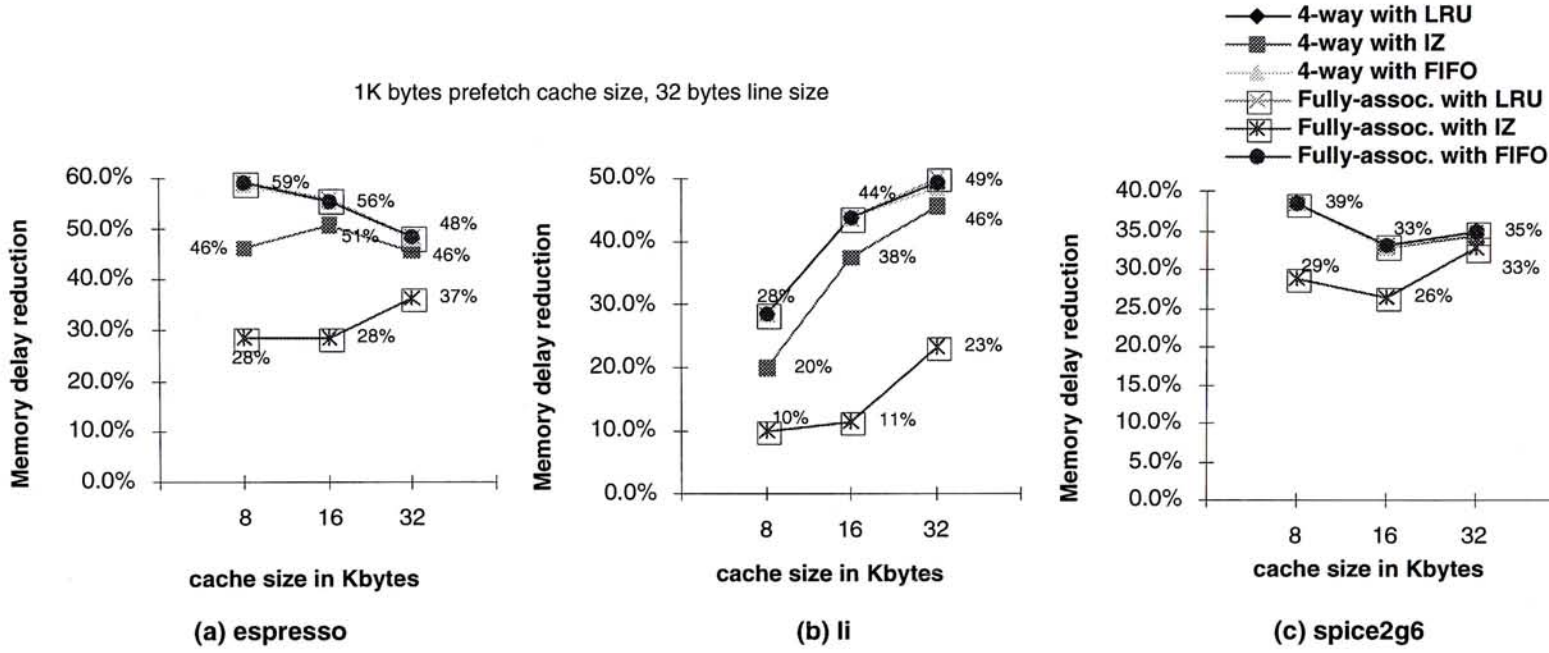


Figure D.15: Results of the third group programs

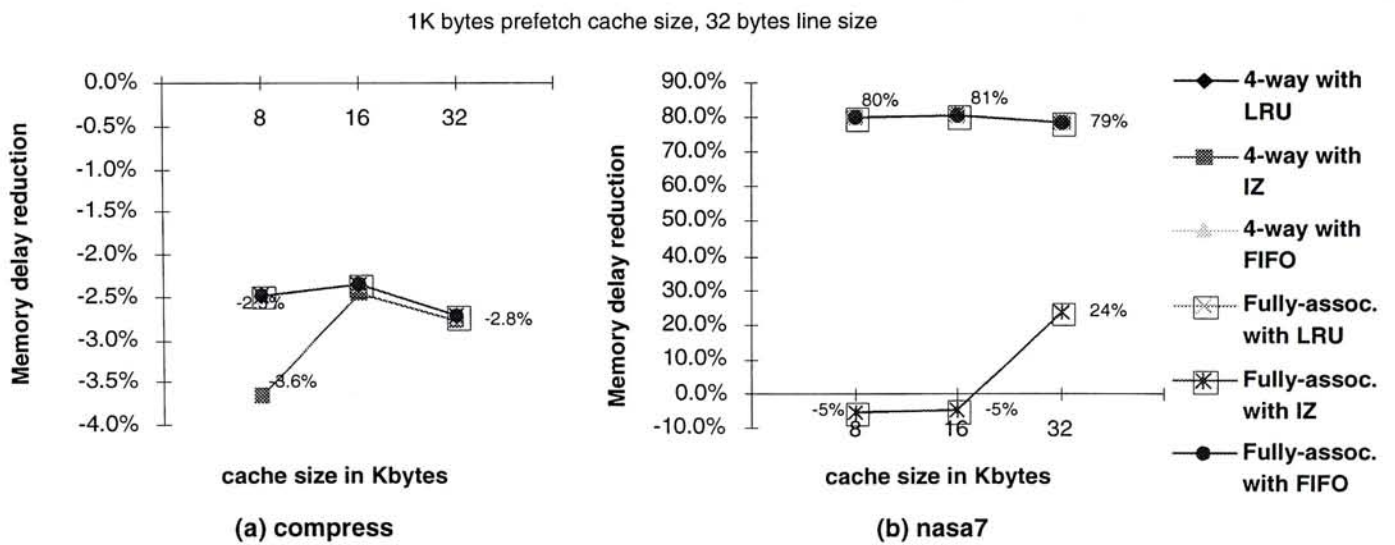


Figure D.16: Results of the fourth group programs

D.2.2 Varying Cache Line Size

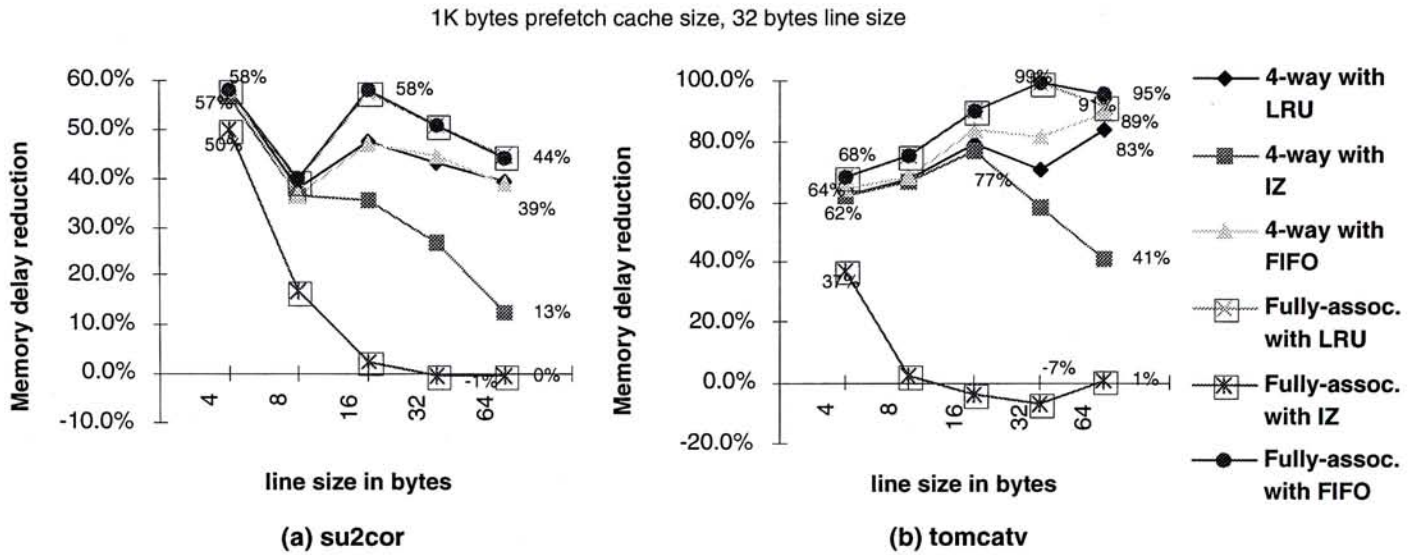


Figure D.17: Results of the first group programs

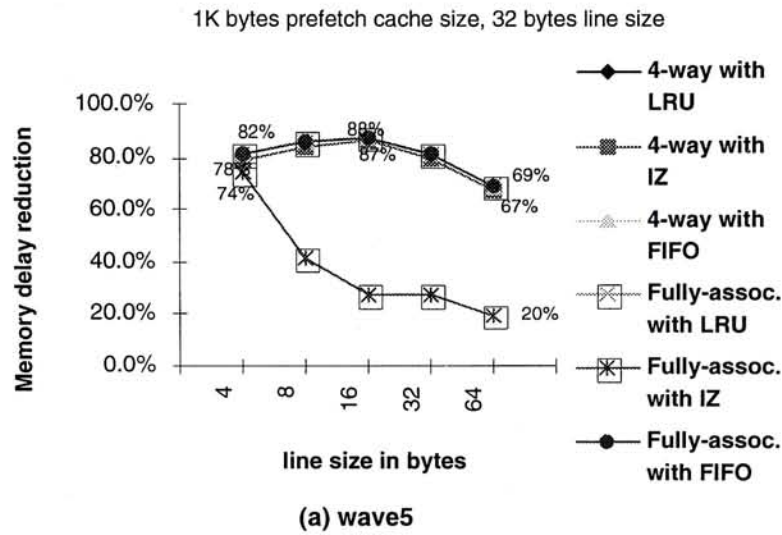


Figure D.18: Results of the second group programs

Appendix D Simulation Results of Prefetch Cache

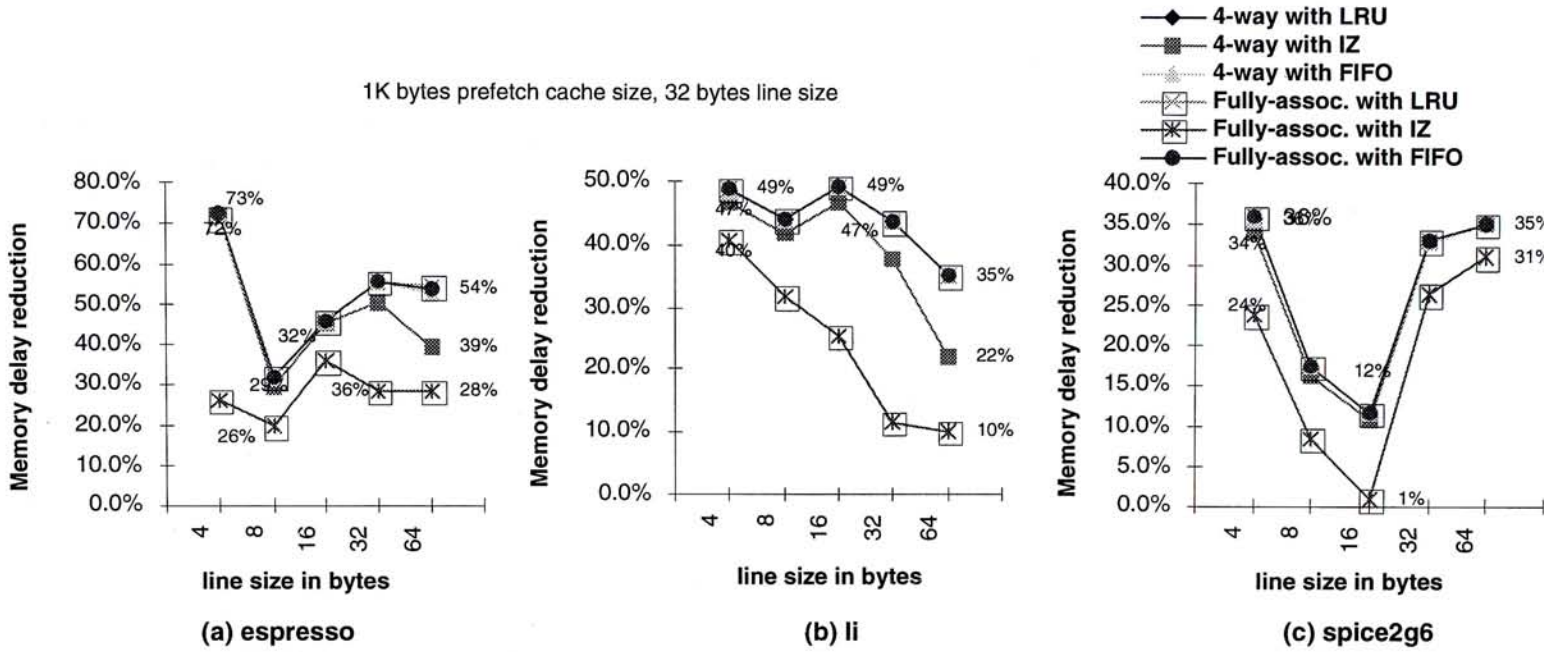


Figure D.19: Results of the third group programs

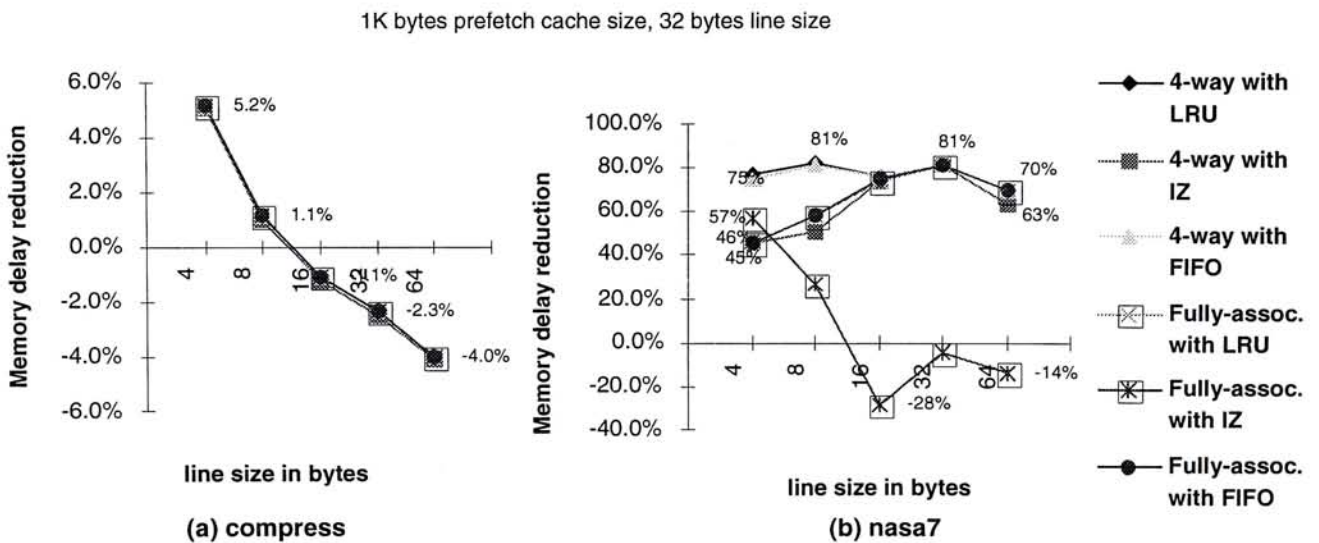


Figure D.20: Results of the fourth group programs

D.2.3 Varying Cache Set Associative

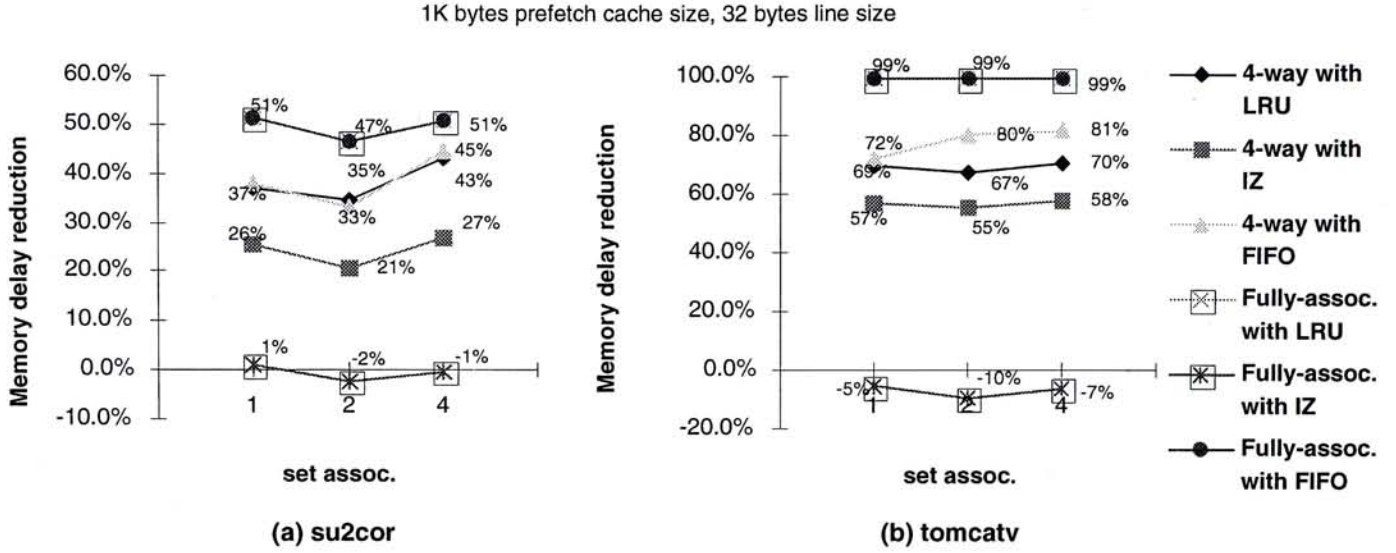


Figure D.21: Results of the first group programs

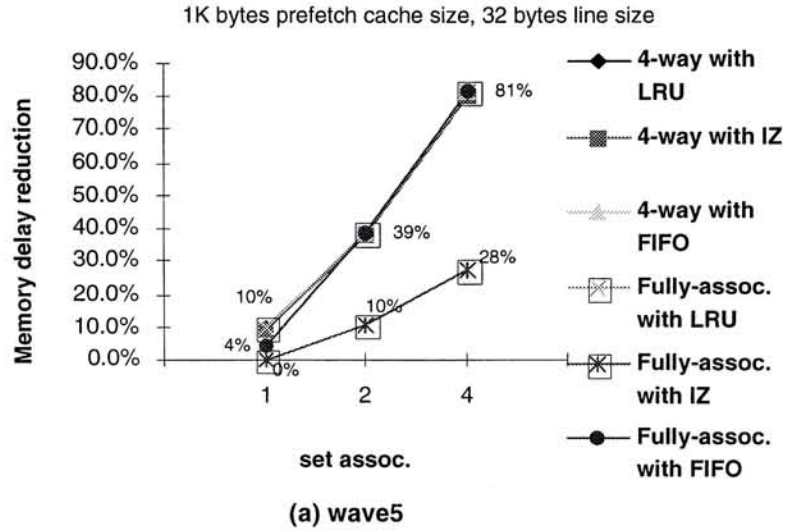


Figure D.22: Results of the second group programs

Appendix D Simulation Results of Prefetch Cache

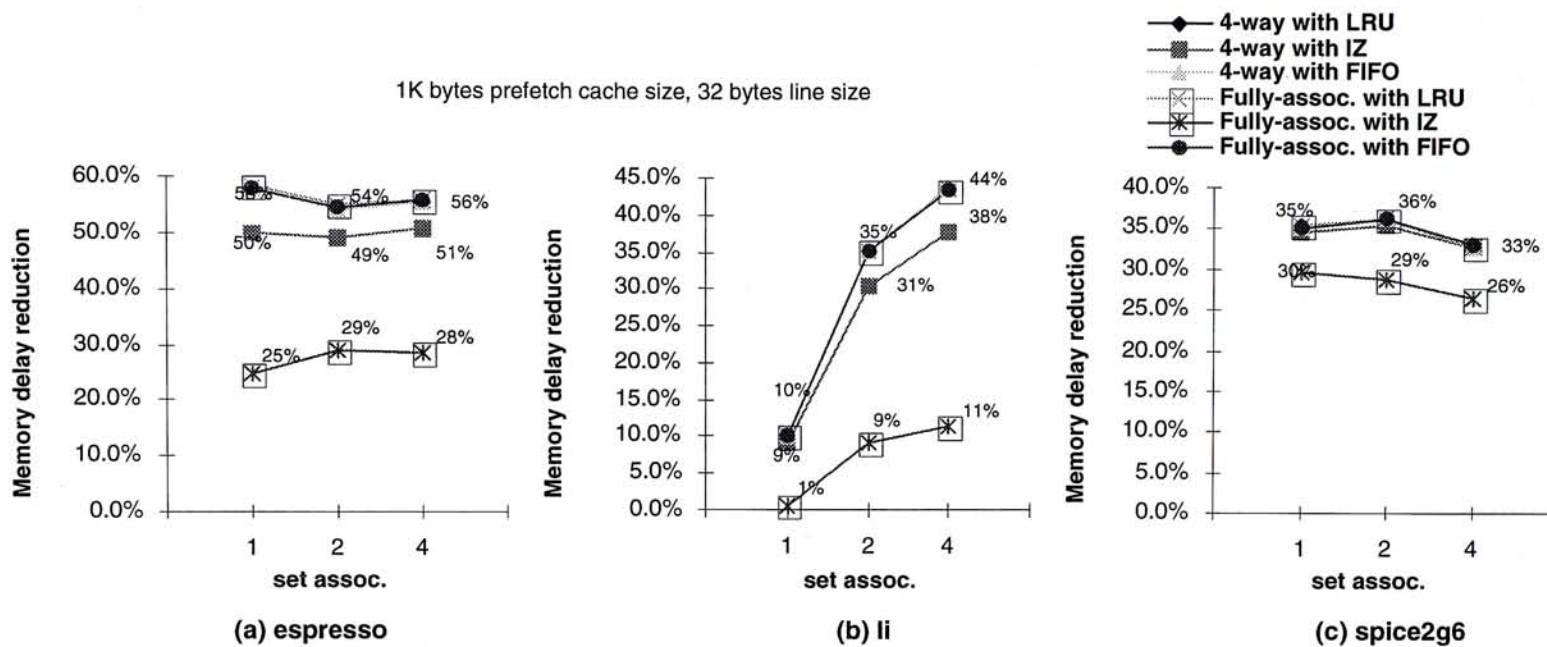


Figure D.23: Results of the third group programs

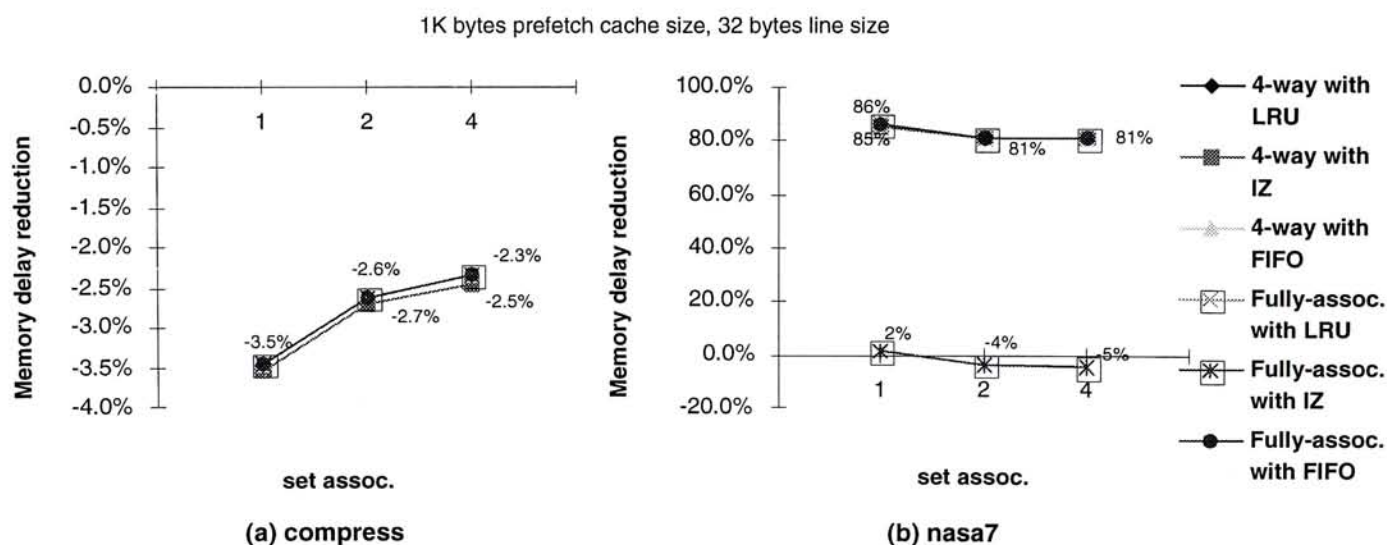


Figure D.24: Results of the fourth group programs

Bibliography

- [BaW89] Baer, J.L., Wang, W.H., "Multi-level cache hierarchies: Organizations, protocols and performance," *Journal of Parallel and Distributed Computing*, Volume 6, Number 3, 1989, pp.451–476.
- [BaC91] Baer, J.L., Chen, T.F., "An effective on-chip preloading scheme to reduce data access penalty," *Proceedings of the 1991 International Conference on Supercomputing*, 1991, pp.176–186.
- [Bre87] Brent, G.A., "Using program structure to achieve prefetching for cache Memories," *Ph.D Thesis, University of Illinois at Urbana-Champaign*, January 1987.
- [CaK91] Callahan, D., Kennedy, K., Porterfield, A., "Software prefetching," *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp.40–52.
- [ChB92] Chen, T.F., Baer, J.L., "Reducing memory latency via non-lining and prefetching caches," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA*, October 1992, pp.51–61.
- [ChB94] Chen, T.F., Baer, J.L., "A performance study of software and hardware data prefetching schemes," *21st Annual International Symposium on Computer Architecture*, 1994, pp.223–232

- [ChM91] Chen, W.Y., Mahlke, S.A., Chang, P.P., Hwu, W.W., "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," *Proceedings of Microcomputing 24*, 1991.
- [Chi94] Chiueh, T.C., "Sunder: A programmable hardware prefetch architecture for numerical loops," *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, May 1994, pp.128–137.
- [FuP91] Fu, W.C., Patel, J.H., "Data prefetching in multiprocessor vector cache memories," *Proceedings of the 18th Annual Symposium on Computer Architecture*, May 1991, pp.54–63.
- [FuP92] Fu, W.C., Patel, J.H., "Stride directed prefetching in scalar processors," *Proceedings of the 25th International Symposium on Microarchitecture*, 1992, pp.102–110.
- [GoG90] Gornish, E., Granston, E., Veidenbaum, A., "Compiler-directed data prefetching in multiprocessor with memory hierarchies," *Proceedings of the 1990 International Conference on SuperComputing*, 1990, pp.354–368.
- [HeP95] Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1995.
- [HP94] Hewlett-Packard, Inc., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, HP Part Number 09740-90039, third Edition, February 1994.
- [IBM89] IBM, *AIX V3.2 for RISC Systems/6000: Assembler Language Reference*, SC23-2197-01, 1989.
- [IBM94] IBM, *The PowerPC Architecture*, edited by May, C., Silha, E., Simpson, R., Warren, H., Morgan Kaufmann, 1994.

- [Jou90] Jouppi, N.P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 18th Annual Symposium on Computer Architecture*, May 1990, pp.364–373.
- [KIL91] A.C.Klaiber and H.M.Levy., "An architecture for software-controlled data prefetching," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp.43–53.
- [Kro81] Kroft, D., "Lockup-free instruction fetch/prefetch cache organization," *8th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, 1981, pp.81–87.
- [Lau96] Lau, S.C., "Improving on-chip data cache performance using instruction register information," *Master Thesis, Department of Computer Science and Engineering, the Chinese University of Hong Kong*, June 1996.
- [Lee87] Lee, R.L., "The effectiveness of caches and data prefetch buffers in large-scale memory multiprocessors," *Ph.D Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, May 1987.
- [MoG91] Mowry, T.C., Gupta A., "Tolerating latency through software-controlled prefetching in shared-memory multiprocessor," *Journal of Parallel and Distributed Computing*, Volume 1, Number 2, June 1991, pp.87–106.
- [MoL92] Mowry, T.C., Lam, M.S., Gupta, A., "Design and evaluation of a compiler algorithm for prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating System, Boston, M.A.*, October 1992, pp.62–73.
- [Mot92] Motorola Inc., *PowerPC 601 RISC Microprocessor User's Manual*, Publication Number MPC601UM/AD, 1992.
- [Por89] Porterfield, A.K., "Software methods for improvement of cache performance on supercomputer applications," *Technical Report COMP TR 89-93, Rice University*, May 1989.

- [Smi78a] Smith, A.J., "Sequentially and prefetching in database systems," *ACM Transactions on Database Systems*, Volume 3, Number 3, 1978, pp.223–247.
- [Smi78b] Smith, A.J., "Sequential program prefetching in memory hierarchies," *IEEE Computer*, Volume 11, Number 12, December 1978, pp.7–21.
- [Smi82] Smith, A.J., "Cache memories," *ACM Computing Surveys*, Volume 14, Number 3, September 1982, pp.473–530,
- [SzY97] Sze, S.C., Young, G.H., "Accurate data prefetching with intelligent replacement policy," *Proceedings of the International Conference on Imaging Science, Systems, and Technology, Las Vegas, Nevada, USA*, July 1997, pp.74–77.
- [Tha81] Thabit, K.D., "Cache management by the computer," *Ph.D Thesis, Rice University*, November 1981.
- [WeS94] Weiss, S., Smith, J.E., *POWER and PowerPC*, Morgan Kauffmann, 1994.
- [YoS98] Young G.H., Sze, S.C., Lau, S.C., "An effective placement policy in cache for prefetched lines," to appear in *ISORA '98, Kunming, China*, August 1998.

CUHK Libraries



003703772