Unified On-chip Multi-level Cache Management Scheme Using Processor Opcodes and Addressing Modes

By Stephen Siu-ming Wong

Supervised By

Dr. Chi-hung Chi

Submitted to the Dept. of Computer Science & Engineering

In Partial Fulfillment of the Requirements for the Degree of

Master of Philosophy

At the Chinese University of Hong Kong

June 1996

香港中文大學
大學圖書館系統
10 MAY 1997
UNIVERSITY LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Abstract

As new generations of microprocessors are to be developed, the speed gap between main memory and new processors is widened. To bridge the gap, fast but small size of cache memory was introduced. When the feature size of transistor in microprocessor is reduced, CPU designers can pack the cache memory into the same silicon die as the processor, they labeled the cache memory as first level (L1) cache. On the circuit board holding the microprocessor, system designers added another level of cache memory on the circuit board to reduce apparent memory latency, and it is labeled second level (L2) cache. The next logical step should be the integration of second level cache on the same chip as the microprocessor.

The conventional usage of cache memory is by storing frequently used data in the cache, and when later retrieval is performed, the speed will be that of the fast cache memory, instead of the slower main memory. This behavior is termed "on-demand fetch". The saving in waiting time occurs only when the same data is referenced the second and later times.

Rather than fetching on-demand, there are useful signals on the CPU which both the first and second level caches can make use of, in order to make predictions about future memory references of the processor, the cache memory can fetch those data from the main memory in advance, this introduces the notion of Cache Prefetching. Prefetching is risky, as if the prefetch is wrong, it does not only cost a waste of time for the prefetch, but, the wrong prefetch may replace a piece

of useful data that will be referenced by the processor later, because the cache memory is small in size. Therefore, an accurate prefetching is desirable. The reference pattern of the processor ultimately depends on the running program, therefore, with sufficient information from the running program, the cache system should be able to make accurate prefetching.

In this dissertation, 13 different prefetching algorithms would be presented, among them five are new prefetching algorithms proposed in this thesis, with the reminding eight were proposed by other papers. One of the new prefetching algorithms proposed in this dissertation is Source Index Register Prefetching, it is a hardware prefetching algorithm which does not depend on the support of language compiler. Source Index Register Prefetching has similar or better performance compare with one of the regarded hardware prefetching algorithms proposed by Dr. Chen Tien-Fu and others in [CB92] called RPT. Source Index Register Prefetching has a substantial improvement on hardware overhead compare with RPT. One of the major achievements from the Source Index Register Prefetching is the choice of index register in a machine instruction as index to the stride value table, instead of instruction address as used by the RPT. The selection of index register in Source Index Register Prefetching lowers the hardware overhead and guarantees the good performance of that algorithm.

The memory access time by using the new prefetching algorithms can be reduced by more than 90% of the "on-demand fetch". The average saving in memory access time for the new prefetching algorithms is around 30%.

Another major achievement in this study was the inclusion of Cache Line Concept in managing on-chip level one and level two cache. The use of line concept takes into consideration the different block sizes as commonly found on L1 and L2 cache. By using a single prefetching unit, and combined with Cache Line Concept, the prefetch behaviors for L1 and L2 cache will be different, and

they are tuned to the parameters for that particular level of cache. The single cache management unit actually ties both L1 and L2 cache together, and they work coherently to serve data that will be referenced in a short span of time.

The unified cache management is beneficial to CPU designers as the prefetching algorithm used depends on CPU signals, that electrical loading is limited. One cache management unit means there will be just a single set of CPU signals running from the CPU core to the unit, and the inclusion of second level cache will not add further electrical loading to the CPU core logic.

Acknowledgment

I must express my gratitude to my advisor, Dr. C H Chi, for his support and guidance throughout my course of research. Dr. Chi introduced me to the wonderland of Computer Architecture, and the realm of cache world in particular. Without Dr. Chi's valuable inspiration, encouragement and insight, I would not be able to complete this study. I would also thank Dr. C Lu and Dr. G Young, who are my markers, Prof. Zheng Wei-min in Tsinghua University, who is the external examiner of my thesis. They have given me helpful suggestions and constructive criticism in polishing my dissertation.

Without fellow graduate students Vincent Yiu, Siu-chung Lau, Wai-wai Chan, Xue-jie Zhang, Siu-ching Sze, and many others, my college life would not be as memorable. I would never forget the very useful brain storming sessions with Wai-wai Chan, the happy atmosphere created by Vincent, Chung, and Siu-ching. Without Xue-jie Zhang, I would not have chance to learn a bit in Mandarin.

I have to thank Dr. P C Wong in the department of Information Engineering, although we did not meet very often, Dr. Wong was the one who activated me to think about the possibility to pursue postgraduate study.

Lastly, I have to give special thanks to my parents, Mika and her parents for their love and support. They are so thoughtful to support me to go back to school. Without them, I would not be able to put down my burdens and go one step further to my goal.

Contents

1 Introduction **1**

1.1 Cache Memory 2

1.2 System Performance 3

1.3 Cache Performance 3

1.4 Cache Prefetching 5

1.5 Organization of Dissertation 7

2 Related Work **8**

2.1 Memory Hierarchy 8

2.2 Cache Memory Management 10

2.2.1 Configuration 10

2.2.2 Replacement Algorithms 13

2.2.3 Write Back Policies 15

2.2.4 Cache Miss Types 16

2.2.5 Prefetching 17

2.3 Locality 18

2.3.1 Spatial vs. Temporal 18

2.3.2 Instruction Cache vs. Data Cache 20

2.4 Why Not a Large L1 Cache? 26

2.4.1 Critical Time Path 26

2.4.2 Hardware Cost 27
2.5 Trend to have L2 Cache On Chip 28
2.5.1 Examples 29
2.5.2 Dedicated L2 Bus 31
2.6 Hardware Prefetch Algorithms 32
2.6.1 One Block Look-ahead 33
2.6.2 Chen's RPT & similar algorithms 34
2.7 Software Based Prefetch Algorithm 38
2.7.1 Prefetch Instruction 38
2.8 Hybrid Prefetch Algorithm 40
2.8.1 Stride CAM Prefetching 40

3 Simulator **43**

3.1 Multi-level Memory Hierarchy Simulator 43
3.1.1 Multi-level Memory Support 45
3.1.2 Non-blocking Cache 45
3.1.3 Cycle-by-cycle Simulation 47
3.1.4 Cache Prefetching Support 47

4 Proposed Algorithms **48**

4.1 SIRPA 48
4.1.1 Rationale 48
4.1.2 Architecture Model 50
4.2 Line Concept 56
4.2.1 Rationale 56
4.2.2 Improvement Over "Pure" Algorithm 57
4.2.3 Architectural Model 59
4.3 Combined L1-L2 Cache Management 62

4.3.1 Rationale 62

4.3.2 Feasibility 63

4.4 Combine SIRPA with Default Prefetch 66

4.4.1 Rationale 67

4.4.2 Improvement Over "Pure" Algorithm 69

4.4.3 Architectural Model 70

5 Results **73**

5.1 Benchmarks Used 73

5.1.1 SPEC92int and SPEC92fp 75

5.2 Configurations Tested 79

5.2.1 Prefetch Algorithms 79

5.2.2 Cache Sizes 80

5.2.3 Cache Block Sizes 81

5.2.4 Cache Set Associativities 81

5.2.5 Bus Width, Speed and Other Parameters 81

5.3 Validity of Results 83

5.3.1 Total Instructions and Cycles 83

5.3.2 Total Reference to Caches 84

5.4 Overall MCPI Comparison 86

5.4.1 Cache Size Effect 87

5.4.2 Cache Block Size Effect 91

5.4.3 Set Associativity Effect 101

5.4.4 Hardware Prefetch Algorithms 108

5.4.5 Software Based Prefetch Algorithms 119

5.5 L2 Cache & Main Memory MCPI Comparison 127

5.5.1 Cache Size Effect 130

5.5.2 Cache Block Size Effect 130

5.5.3 Set Associativity Effect 143

6 Conclusion **154**

7 Future Directions **157**

7.1 Prefetch Buffer 157

7.2 Dissimilar L1-L2 Management 158

7.3 Combined LRU/MRU Replacement Policy 160

7.4 N Loops Look-ahead 163

List of Figures

2.1 Probability Distribution for Spatial Locality 19

2.2 Probability Distribution for Temporal Locality 21

2.3 Constant Stride Reference Patterns 25

2.4 Content Associative Memory 41

3.1 Multi-level Memory Hierarchy Simulator Configuration 46

4.1 Block Diagram for Processor with SIRPA scheme 53

4.2 Source Register Indexed Stride Values Table 54

4.3 SVT State Field State Transition Diagram 55

4.4 CPU Access Sequence on Cache Blocks 57

4.5 CPU Access Sequence with Line Concept Prefetch 58

4.6 Processor using SIRPA and Line Concept scheme 60

4.7 CPU Access Sequence on L1, L2 cache blocks w/Line Concept . . 61

4.8 Combined Prefetch Scheme 71

5.1 Overall MCPI comparison by cache size 88

5.2 Overall MCPI comparison by cache size (cont.) 89

5.3 Overall MCPI comparison by cache size (cont.) 90

5.4 Overall MCPI Reduction comparison by cache size 92

5.5 Overall MCPI Reduction comparison by cache size (cont.) 93

5.6 Overall MCPI Reduction comparison by cache size (cont.) 94

5.7 Overall MCPI comparison by block size 95
5.8 Overall MCPI comparison by block size (cont.) 96
5.9 Overall MCPI comparison by block size (cont.) 97
5.10 Overall MCPI Reduction comparison by block size 98
5.11 Overall MCPI Reduction comparison by block size (cont.) 99
5.12 Overall MCPI Reduction comparison by block size (cont.) 100
5.13 Overall MCPI comparison by associativity 102
5.14 Overall MCPI comparison by associativity (cont.) 103
5.15 Overall MCPI comparison by associativity (cont.) 104
5.16 Overall MCPI Reduction comparison by associativity 105
5.17 Overall MCPI Reduction comparison by associativity (cont.) . . . 106
5.18 Overall MCPI Reduction comparison by associativity (cont.) . . . 107
5.19 Overall MCPI Comparsion of Line Concept 113
5.20 Overall MCPI Comparsion of Line Concept (cont.) 114
5.21 Overall MCPI Comparsion of Line Concept (cont.) 115
5.22 L2/Main Mem MCPI Comparsion of Line Concept 116
5.23 L2/Main Mem MCPI Comparsion of Line Concept (cont.) 117
5.24 L2/Main Mem MCPI Comparsion of Line Concept (cont.) 118
5.25 Overall MCPI Comparison of Default Prefetch Scheme 120
5.26 Overall MCPI Comparison of Default Prefetch Scheme (cont.) . . 121
5.27 Overall MCPI Comparison of Default Prefetch Scheme (cont.) . . 122
5.28 L2/Main Mem MCPI Comparison of Default Prefetch Scheme . . 123
5.29 L2/Main Mem MCPI Comparison of Default Prefetch Scheme (cont.) 124
5.30 L2/Main Mem MCPI Comparison of Default Prefetch Scheme (cont.) 125
5.31 Overall MCPI Comparison by Software/Hardware Prefetch Algorithms 128

5.32 Overall MCPI Comparison by Software/Hardware Prefetch Algorithms (cont.) 129
5.33 L2/Mem MCPI comparison by cache size 131
5.34 L2/Mem MCPI comparison by cache size (cont.) 132
5.35 L2/Mem MCPI comparison by cache size (cont.) 133
5.36 L2/Mem MCPI Reduction comparison by cache size 134
5.37 L2/Mem MCPI Reduction comparison by cache size (cont.) 135
5.38 L2/Mem MCPI Reduction comparison by cache size (cont.) 136
5.39 L2/Mem MCPI comparison by block size 137
5.40 L2/Mem MCPI comparison by block size (cont.) 138
5.41 L2/Mem MCPI comparison by block size (cont.) 139
5.42 L2/Mem MCPI Reduction comparison by block size 140
5.43 L2/Mem MCPI Reduction comparison by block size (cont.) 141
5.44 L2/Mem MCPI Reduction comparison by block size (cont.) 142
5.45 L2/Mem MCPI comparison by associativity 144
5.46 L2/Mem MCPI comparison by associativity (cont.) 145
5.47 L2/Mem MCPI comparison by associativity (cont.) 146
5.48 L2/Mem MCPI Reduction comparison by associativity 147
5.49 L2/Mem MCPI Reduction comparison by associativity (cont.) . . 148
5.50 L2/Mem MCPI Reduction comparison by associativity (cont.) . . 149

7.1 Processor with Prefetch Buffer 159

List of Tables

5.1 Best Prefetch Algorithm Producing Smallest Overall MCPI 109

5.2 Best Overall MCPI by using SIRPA & RPT family Algorithms . . 150

5.3 Best Overall MCPI by using PFONMISS/SIRPA/RPT Algorithms 151

5.4 Benchmarks with high reduction in MCPI by using Default Prefetch 152

5.5 Best Software Based Prefetch Algorithm 152

5.6 Overall MCPI by using SETCAM & SIRPA family Algorithms . . 153

7.1 LRU Replacement Example 161

7.2 MRU Replacement Example 162

Chapter 1

Introduction

The disparity between speed of microprocessors and main memory has been worsened in the last decade. With today's technology, typical CPUs like UltraSparc runs at 167MHz [Gwe94], MIPS R10000 processor runs at 275MHz [AC95], Digital's Alpha 21164 runs at 333MHz [Ban95]. These clock rates correspond to cycle time of 5.9ns, 3.6ns and 3ns respectively. However, the random access time for typical transistor based dynamic memory is about 60ns [Bol94]. If a datum is to be retrieved from main memory, the processor will be stalled for 10-20 cycles. That means a degradation of performance in one order of magnitude. The situation would be even more demanding when UltraSparc, MIPS R10000 and DEC Alpha 21164 can execute at most four instructions [Gwe94] [AC95] [Ban95] per cycle.

To bridge the speed gap between microprocessors and main memory, fast memory is inserted between the processor bus and the main memory. This type of memory is called cache memory [Smi82]. However, due to high cost to produce cache memory, its size is usually limited. The performance of the overall computer system can be improved if instructions and data that are to be referenced are kept in the cache memory [Prz90]. The less frequently used data are stored in the main memory.

1.1 Cache Memory

In order to feed a processor with enough instructions and data without stalling, it is typical to include cache memory in the system design. Most modern microprocessors have on-chip cache memory implemented. UltraSparc has a 16k Bytes cache memory for instruction and a 16k Bytes cache memory for data [Gwe94]. Intel's Pentium has 8k Bytes instruction cache and 8k Bytes data cache [CTR93]. MIPS R10000 has 32k Bytes instruction cache and 32k Bytes data cache [AC95]. These on-chip cache memory designs match the speed of the microprocessor [AC95] in general. If all instructions and data can be accessed from the on-chip cache, the processor can execute programs in high speed.

The working principle of cache memory depends on locality, and there are two kinds of them. Temporal locality refers to the expectation that instructions and data that are currently in use, will be referenced again in the near future. Spatial locality refers to the likelihood of adjacent access to memory in a short span of time [Prz90]. The locality feature in a program depends on the general sequential execution flow and loops inside it. When requested instructions or data can be found in the cache, the processor can retrieve from the cache instead of from the main memory. There will be a saving of time, which is equal to the difference of access timing between the cache and the main memory. If the majority of processor memory requests can be satisfied from the cache, the memory latency as experienced by the processor will be that of the access time of the cache, instead of the slower main memory.

Another layer of cache memory can be implemented on the system board level, which is called second level cache. Second level cache is usually larger in size but with a slower access time compare with the on-chip first level cache. There are newer generation processors with on-chip second level cache, such as DEC Alpha 21164 and Intel Pentium Pro, whereas other processors have on-chip second level

cache controller, such as MIPS R10000, Sun UltraSparc I, and IBM/Motorola PowerPC 620.

1.2 System Performance

The performance of a computer system depends on the length of time that the CPU spent on computation and the time spent on moving data to/from memory. The total of the above adds up to a figure in number of clock cycles that the computer system used to execute a program. This figure can be used to show the performance of a computer system [HP90]. However, in order to compare performance across different programs, another measurement, which should be independent of the running program, is necessary. Cycles Per Instruction (CPI) is such a measurement that is independent of the complexity of the running program. CPI is defined as the number of CPU clock cycles for a program to run divided by the instruction count for that program [HP90].

For non-superscalar processor, the upper limit of CPI is 1. When CPI is 1, that means the processor can finish an instruction every cycle. For superscalar processor, as there are more than one function units in the CPU, it can process a few instructions in parallel. In theory, the upper limit of CPI of superscalar processor can be smaller than one, the limit should be the inverse of the maximum number of instructions that can be issued per cycle.

1.3 Cache Performance

There are a few quantities which can be used to measure cache performance. One of them is cache hit ratio (or hit rate, if the quantity is expressed as a pure number). Hit ratio is defined as the percentage of memory requests that can be found (a hit) in the cache [Prz90]. The upper limit of hit ratio is 100%, at

that time, all memory references can be satisfied by the cache memory, and the memory latency will be the access time of the cache. However, as the cache size is small compare with the main memory, it is seldom to have 100% hit ratio. For hit ratio lower than 100%, the memory latency can be calculated by the following formula:

$$\begin{aligned} \text{latency} &= \text{hit rate} * \text{cache access time} \\ &+ (1 - \text{hit rate}) * \text{lower level memory access time} \end{aligned}$$

Hit rate can be used as a rough comparison guideline for memory hierarchy performance. The higher the hit rate, it is expected the system will have a higher performance. Hit rate is inherently easy to find by simulation, as the system has only to determine whether a memory reference is a hit or a miss in a cache.

Due to the fact that the memory latency depends not solely on the hit rate, but also the cache access time, and lower level memory access time, for different computer systems, it will be misleading to compare hit rate for performance comparison. Eventhough a system may have a higher hit rate, we cannot conclude that it will have a higher performance, because the lower level memory access time is unknown. In [BC91], another measurement called Memory CPI (MCPI) was proposed. Memory CPI is the average number of cycles that the processor spent to retrieve a word from the memory hierarchy. For the MCPI figures reported in this dissertation, they are calculated by the following formula:

$\text{MCPI} = \text{CPI} - \text{CPI with infinite fast memory}$

CPI with infinite fast memory was found by stimulating a computer system with a 100% first level cache hit ratio, and the latency for the first level cache is zero cycle. That means, whenever the processor requests a word from the memory, the requested word will be presented to the processor without delay.

The CPI with infinite fast memory shows the processing computation overhead of the program, which that figure depends on the complexity of the running program. MCPI is a fair comparison for cache performance, because it does not depend on the number of instructions in the program, and it does not depend on the time required by the processor to do arithmetic calculations also. Assumed, the clock rate of two systems are equal, their MCPIs can be compared directly, whereas, the lower the MCPI, the faster the memory response.

MCPI is harder to find by simulation compare with the hit rate, as the simulation has to be done in a cycle–by–cycle basis. Not only whether a memory access is a hit or a miss has to be determined, but the exact number of cycles that spent to fetch that memory request is required. When there are a number of memory levels in the system, and all these memory levels can initiate transfer in parallel, the demand on resolution of the simulator is great.

There was an analytical model presented in [MeM92], which used a simplified model to calculate the CPI with certain cache configuration parameters. However, that model is hard to extend to cover a system with cache prefetching algorithm. Chang and Hsu in [ChH94] had proposed a method to reduce the overhead in simulating hit ratio. Obaidat and Khalid in [ObK95] refined hit ratio into Solo Hit Ratio, Local Hit Ratio and Global Hit Ratio. Lim, Bae, Jang, etc. did an analysis on worst case timing for a RISC processor using cache memory in [LBJ95].

1.4 Cache Prefetching

The performance of cache memory can be improved by cache prefetching. Cache prefetching is cache system initiated memory transfer, which the cache system predicts that the processor will access certain memory location in the near future.

When the cache prefetches are accurate, the hit rate of the cache system will be increased, and hence the MCPI can be reduced, because, more processor memory requests can be satisfied from the cache memory, instead of from the slower main memory.

In general, the memory location to be prefetched can be found by two categories of methods. One is from cache hint instruction embedded in the running programming. These hint instructions are inserted by the language compiler. When the compiler detects a loop, the compiler can insert hint instructions to prefetch those data that will be used in the next iteration. Because, those hint instructions are inserted by compiler, this category of prefetching is called software based prefetching. The accuracy of software prefetching is very high, as the compiler can perform rigorous analysis on the original source program, and the data to be accessed can be predicted with high precision. However, software based prefetching suffers from big overhead, as hint instructions are inserted in the program, they will in themselves increase the memory traffic and the processor has to spend time to fetch those cache hint instructions, although it will perform nothing on those instructions.

The other cache prefetching category uses pattern information from historical memory accesses. The cache management system keeps track of memory reference locations, and if a regular pattern is found, the later memory accesses can be predicted, and hence prefetching can be performed on those locations. This category is called hardware prefetching. Because there is no extra hint instruction in the program stream, there is no software overhead in hardware prefetching schemes. However, the performance of hardware prefetching algorithms depend on the ability to discover memory access patterns. It is one of the main goal of this dissertation to propose hardware prefetching algorithms for high performance cache system.

1.5 Organization of Dissertation

Related studies of cache memory and multi-level cache hierarchy will be reported in the next chapter. Previous works on software based and hardware prefetching algorithms for cache will also be discussed.

A multi-level memory hierarchy simulator was built to test prefetching algorithms studied in this dissertation. A brief discussion of the features of the simulator will be presented in the third chapter.

A few hardware prefetching algorithms for cache system are proposed in the forth chapter. The rationale to have a combined level one and two cache management system will be presented also in that chapter.

Eight benchmark tests in the SPEC92 suite were used to measure the performance of cache prefetching algorithms. In a particular benchmark, NASA7, as there are seven different programs inside it, a detail study on each program was done, and results gathered. All the benchmark results are to be presented in the fifth chapter. Total cycles consumed by each benchmark programs, Overall MCPI, partial MCPI due to Second Level Cache/Main Memory and hit rate for both level 1 and level 2 cache are included in that chapter.

A conclusion on the achievement of this dissertation will be presented in the sixth chapter, and future directions are proposed in that chapter also.

In the Appendix, data from the simulations are included there, and detail graphs of each benchmark tests can be found in that chapter.

Chapter 2

Related Work

In this chapter, research works done by other people would be summarized. Firstly, an introduction to the memory hierarchy used in a computer system would be discussed. Then, the configuration parameteres affecting the performance of cache memory would be presented. One of the important properties that made cache memory works is the locality principle. The classification of it would be discussed.

When cache memory does prove to work well in a broad range of programs, the reason why processor designers do not opt for a large on chip first level cache would be discussed. The trend is instead to implement on chip second level cache. Finally, three categories of cache prefetch algorithms would be presented at the end of this chapter.

2.1 Memory Hierarchy

If the memory references made by a processor to the memory system is truly random, the gain in performance by employing a small amount of faster memory complement with a large pool of slower memory will be minimal. Because the probability that the next memory reference will fall into the area contained in

the fast memory depends solely on the ratio of size of the faster memory to the slower memory. The assumption that the faster memory is small in size, makes the above probability small, and therefore, it will not be useful to improve system performance.

From actual trace studies, we found that memory references are with patterns. We can make prediction to the forthcoming memory reference by history of previous memory references. Because of this property, if all the to-be-referenced data are stored in the faster memory, and the processor can access all the required data from that memory, there will be an appearance that the whole system is running out of the faster memory without knowing the existence of the slower memory.

The faster memory is small in size, we still need the large pool of slower memory to hold programs and data. We just have to devise an algorithm to decide when we should promote a piece of data from the slower memory to the faster memory, and in turns, which piece of data in the faster memory is to be replaced. The algorithm is simply a comparison on the probabilities that which memory address will going to be accessed, and take the content of that memory address into the faster memory.

By using similar arguments as above, we can have a hierarchy of memory layers, with the fastest memory and the smallest size at the top of the hierarchy, and each lower layer with progressively slower access time and larger in size. A hierarchy of layers is better than a two-layer architecture because system designers usually have more than two choices of memory with different cost/performance ratio and other physical characteristics. In general, the faster the memory, the higher the manufacturing cost and the converse is true for slower memory. Multi-layer hierarchy is also desirable to minimize the penalty paid when a piece of data is requested by the processor but that cannot be found in the fastest memory.

With the backup of layers of progressively slower memory, it will have a high chance that the required data can be found in an intermediate layer instead of the lowest one.

2.2 Cache Memory Management

The configuration of cache memory can be classified by a few parameters.

2.2.1 Configuration

Block Size

In order to manage a cache, it is seldom to use byte as unit to allocate memory in it, instead the whole cache is divided into *cache blocks*. The smallest unit to occupy in a cache is usually a block, although there are implementations of cache with sub-blocks. The size of a block is called *block size*, and in some literature, it is called *line size*. With a large block size, the number of blocks in a cache will be small, and vice versa. The hardware overhead to manage a cache, including the tag memory, valid bits, dirty bits, depends on the number of blocks. Therefore with a smaller block size, the overhead will be higher.

It is common to have block size in a cache selected to be integer multiples of its data bus width. With a 64-bit data bus, the block size in a cache may be 8 Bytes, 16 Bytes, 32 Bytes. The larger the block size, the higher the penalty when there is a cache miss. Because, the cache block allocated has to be fully filled from the lower level memory before the CPU can proceed to process, although there are implementations with "early available" feature, which the cache will selectively to load the portion of memory in the block that is requested by the processor.

A block in a cache will act as a small look ahead buffer, because when the

CPU access a memory location in a block, the bytes ahead will also be fetched into the cache. If the memory access is going forward, the next reference can be done in the cache. Therefore, in general, a large block size will gives a higher hit rate for the cache. However, a large block size will decrease the number of available blocks in a cache, and more conflicts in allocating blocks are expected, this will decrease the usefulness of the cache.

Set Associativity

When there is a cache miss, a cache block will be allocated from the cache to hold the data to be transferred from memory. Which blocks can be selected to use depend on the associativity of the cache. For cache with associativity of one, or called *direct mapped cache*, there is only one block that can be selected. The block number is found by a simple modulus calculation from the miss address. When the set associativity is larger than one, the cache is called a *set associative cache*, the mapping function from address to cache block number will be one–to–many.

When the processor requests a memory location, the same mapping function will be performed to find out the potential block which may hold the requested datum. The exact address of the memory reference will be compared with the tag value stored in the cache block, if there is a match, then that will be a cache hit. Because there is only one potential cache block per memory address in a direct mapped cache, the address comparison to be done is minimal.

Direct mapped cache is the simplest to be implemented in hardware [Hil88], however, because there are many addresses in the main memory that will be mapped to a single cache block, there will be conflicts in using a cache block. The situation is bad especially when two data to be used in a program loop maps to the same cache block, then, these two data will replace each other in turns and causing a lot of cache misses. The situation can be improved by allowing more

than one cache blocks to serve a memory address. The number of blocks available to be used is called the associativity of the cache. With a cache of associativity two, that means, for every memory address, there are two blocks in the cache that can hold the data. The two blocks are termed a set, in a set associative cache.

Set associative cache poses difficulties in hardware implementation. When the processor requests a memory location, the cache has to check all the blocks that are allowed to serve that location in order to determine whether it is a cache hit [Hil88]. In order to speed up the access, all the address comparisons have to be done in parallel. With a 4-way set associative cache, there should be four comparators in the cache to make address comparisons. It is a hardware difficulty to have a high number of cache blocks in a set, because each comparator will impose electrical loading on the CPU address bus, and the maximum loading allowed is usually a limited figure.

In general, for first level cache, due to it is in the time critical path, the set associativity will not be a large number. However, in second level cache, it may be possible to implement set associative cache there. Yang and Adina in [YaA94] suggested an innovative method to choose the set associativity for cache, in order to minimize the chance for conflict miss.

Cache Size

The size of the cache memory is usually small. The *cache size* is determined by the block size, number of sets and associativity of the cache. The apparent cache size that can be utilized by programs can be found by the following formula:

cache size = block size $*$ associativity $*$ number of sets

The larger the cache size, usually, it will give the better performance. However with the same cache size, the difference in block size, associativity and number

of sets will give different cache performance [PHH88]. Moreover, there is hardware overhead in implementing the cache tag memory, which is the area to hold the memory address of where the data in the cache block are come from. And additional memory has to be reserved to implement valid bits and dirty bits if the cache is a write back cache. These overheads are usually not reported in the cache size, but they do consume area in the CPU die.

When the first level cache is partitioned into two parts, namely *Instruction cache* (I-cache), and *Data cache* (D-cache), the optimal size for them are not necessarily the same. Stone, Turek, etc. in [STW92] studied the conditions to find out a good allocation of size to respective caches. Boleyn, Debardelaben, etc. in [BDT93] studied the usage of a split data cache designed for superscalar processors, in particular how the integer unit and the floating point unit in the CPU can access the data cache in parallel.

2.2.2 Replacement Algorithms

With direct mapped cache, if a piece of data is requested by the processor, and the memory location is not currently held in the cache, there is only one block suitable to hold that data. If the cache block selected is occupied by some other memory locations, the original data will be replaced. As there is no choice for which block is going to be replaced, there is no alternate replacement algorithm. However, with set associative cache, there are more than one blocks which can hold the requested data, the replacement algorithm determine which block is selected for replacement. In general, there are two types of replacement algorithm for set associative cache.

Least Recently Used

Least Recently Used replacement algorithm depends on the belief that the cache block which is least recently accessed will also have the least chance to be used again in the future. This assumption is usually true except in wield situations where the number of blocks required is larger than the set associativity of the cache.

In order to implement LRU replacement algorithm, the cache system has to keep track to the last access time of each cache block. By using a time tag for each cache block, when it is necessary to do a block replacement, the block in a set with the smallest time tag will be selected. To memorize a big time tag field is usually a waste of memory space, therefore, most processors implements pseudo LRU replacement algorithm. The size of the time tag field is small, and the cache memory will periodically clear out all time tags in the cache. Then, only a running counter with a few values is used to mark the last access time. With 3 to 4 bits of time tag fields, the performance of the cache will be very close to the true LRU replacement algorithm.

Random Replacement

Another category of replacement algorithm is *random replacement.* When a cache block has to be selected for replacement in a set associative cache, the cache system will select a random block in the set for replacement. Random replacement does not depend on the time tag field, and therefore to use it, the time tag field is not required in the cache system. With a sufficiently large cache, the performance of random replacement is very close to LRU algorithm.

There are other replacement algorithms designed for specific situations, such as *First In First Out*, or *Last In First Out.* Westerholz and Honal, etc. proposed in [WHP95] three methods to use processor runtime information to control the

cache replacement algorithms. These replacement algorithms are not covered in this thesis.

2.2.3 Write Back Policies

When a processor does only read operations on memory, the data content kept in the main memory will be the same as those kept in the cache. There is nothing to do when a cache block has to be replaced, the data content in the cache block can simply be discarded, because the main memory still has a copy of the latest value. However, when there are processor updates to certain memory locations, we have to deal with how the updated values are to be propagated to the main memory. Jouppli in [Jou93] studied the effect of different write policies including Write Through and Write Back cache.

Write Through Cache

Write Through is the simpler type of update policy. For every processor update to memory location, even when it is a cache hit, the updated value will write both to the cache memory and the main memory. The processor will only be allowed to proceed when both the cache update and main memory update are completed. In this type of update policy, the values kept in main memory are always the most update version, and the cache does not have to keep track whether there is processor update. On cache block replacement, the block selected to be replaced can simply be discarded.

The benefit of Write Through cache is a simpler hardware implementation, however it suffers from the fact that the cache will only be useful to shorten memory latency for read operations. Write operations are always treated as cache miss in terms of memory latency. From trace analysis, we found that there were a lot cases that the same memory address is repeatedly updated and only

the last value is useful to the result. The processor performance can be improved if the cache writes to the main memory only when that cache block is replaced.

Write Back Cache

Write Back cache handles memory updates from processor more gracefully than the Write Through scheme. The updated values are stored only in the cache blocks and at the same time, a flag in the cache block called *Dirty Bit* is set. When that cache block is selected to be replaced, if the block is dirty as indicated by the Dirty Bit, the content in the cache block will be transferred to the main memory. If the processor updates a particular memory location in the cache for many times, all the updates will be made in the cache memory and the latency will be the speed of the cache.

The Write Back scheme should give a superior performance compare with Write Through one. However, if there are multiple processors sharing the same main memory but they have corresponding private cache, there will be coherence problem. That is, the content in the main memory is not always the most updated version, and the private cache in each processor may have their own copy of 'updated' values.

2.2.4 Cache Miss Types

Mark D. Hill in [Hil87] classifies cache miss into three categories.

Compulsory Miss

Compulsory Miss occurs when the processor references a memory location the first time. Assumed the cache memory supports on-demand fetch only, compulsory miss cannot be avoided, as it is the time when the data are read from the main memory and stored into a cache block.

Conflict Miss

Conflict Miss occurs when two data are mapped to the same cache set, and they replace each other from the cache. With direct mapped cache, only one cache block is available to each memory address, and if two memory addresses which mapped to the same cache block are accessed, they will incur conflicts. With set associative cache, where there are more than one cache blocks in a set, the conflict miss will only occur when the number of data items mapped to the same cache set is greater than the set associativity of the cache.

Capacity Miss

Capacity Miss refers to the situation that the cache memory is not large enough to hold the working set of the program. The only way to reduce capacity misses is to increase the cache size.

2.2.5 Prefetching

Cache using on-demand fetching reduces the memory latency when the processor references a memory location the second time, when the content of that memory location is contained in the cache, then the access time for that location is of the speed of the cache. In order to further improve the memory access time, the cache system can initiate pro-active prefetching from the main memory. If the prefetched data are used by the processor in the future, the memory latency for the first access will also be the speed of the cache. Prefetching is workable as the processor does not access memory in constant pace. Usually, there are idle cycles when the processor is doing computation, and the memory bus is free. Prefetching algorithm makes use of these idle bus cycles, and transfer data from main memory to the cache.

Just the same as processor initiated memory request, cache prefetching usually

consumes more than one cycle to complete the transfer of a cache block. There will be a situation when the cache is performing prefetching but the processor would make a memory request. In order to let the processor has the highest priority to retrieve the requested data, most cache prefetching algorithms will abort the prefetch transfer in place and give way to the processor memory request.

Prefetching algorithm is one of the focus in this thesis, in later paragraphs, we will discuss some common cache prefetching algorithms. We are going to discuss the properties of memory reference pattern, including the nature of them, and the classification of patterns.

2.3 Locality

From program trace analysis, when a processor accesses a certain memory address, there is a high probability that the memory references in the near future will fall into close vicinity of the current memory access, we call this property locality. Locality can be further classified into two types, one of them is related to space and is called *Spatial Locality.* The other one is related to time and is called *Temporal Locality.*

2.3.1 Spatial vs. Temporal

For machine instruction execution, there is very high tendency that the instruction reference pattern will be linear, which the memory address following the current one will be referenced in the next cycle. The exception will be some branch instructions. For conditional branches, the backward branch has a more than 80% of chance to be a taken branch due to most backward branches are actually the last statement in program loops. Whereas for a forward branch, the chance for the branch to be taken is around 50%. These forward branches are

usually instructions used in if–then statements.

In general, with a current access to memory address a, the probability that a future memory access to address $a + \delta$, where δ is a small integer, is very high. As δ becomes larger, the probability for that address to be accessed will be diminished. The probability distribution across the memory address will have the shape as in figure 2.1.

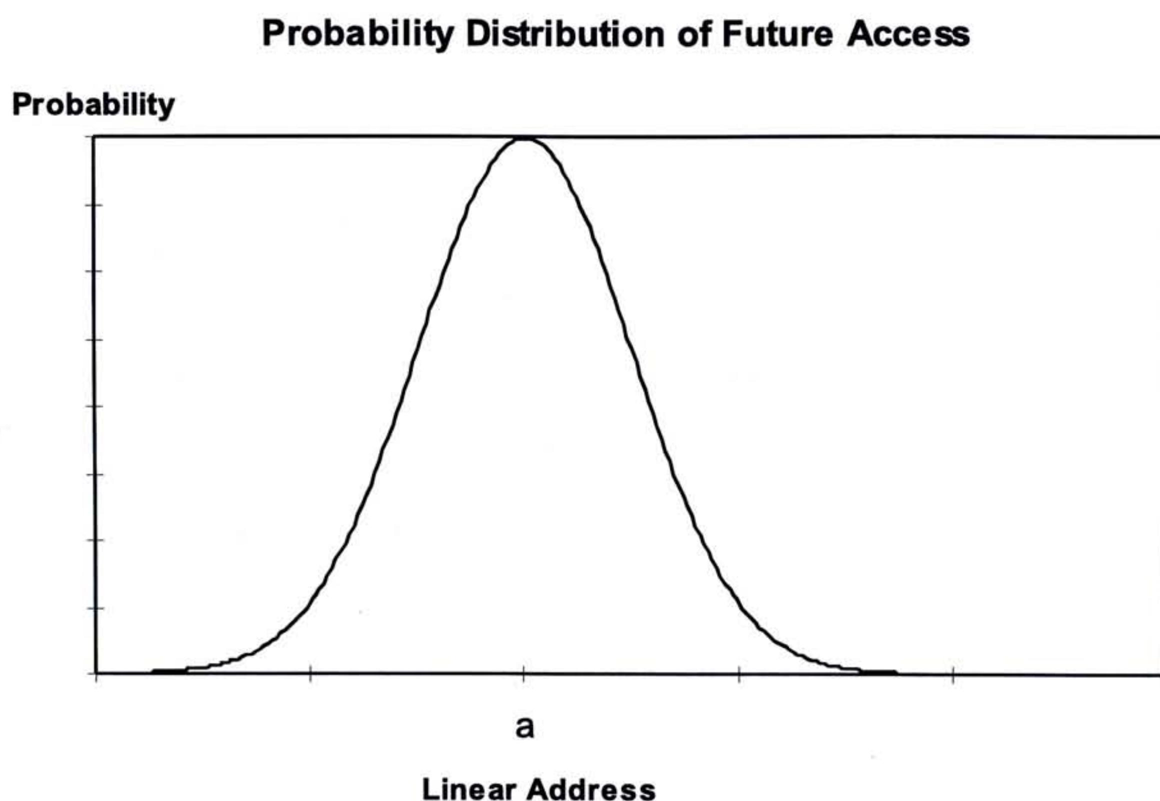


Figure 2.1: Probability Distribution for Spatial Locality

When the processor accesses memory address a, the cache memory should hold the contents from memory around a, then, the probability that the processor can find the future memory references in the cache will be high. We termed this kind of property Spatial Locality. The block size in a cache serves the spatial locality property by retrieving in advance some bytes around the referenced location.

Other than instruction reference, data reference also exhibits spatial locality. In programs using matrices or arrays, it is common to have program loops to

process the whole matrix or array. The memory access pattern for data reference will follow the prediction of spatial locality.

For instructions in a program loop, these instructions have a high chance to be used again in the next loop iteration. The probability depends on the loop iteration count. For some constants used in a program, the same data may be reused in the future. With the following code fragment:

```
for (i=0; i<100; i++)
    a[i] = b[i] + c;
```

The content in c is reused for 100 times and so do the instructions to be performed in the loop. When a memory location is accessed, the same location has a high probability to be accessed again in a short span of time. The probability for that location to be accessed will be decreased with time, that means, when a memory address is not accessed for a long time, that address will have a lower chance to be called in the near future. That property of memory reference is called *Temporal Locality*. The probability distribution of future access to the same address will have a shape of figure 2.2.

When the processor accesses memory address a, the cache memory should hold the content in address a for future reference. When there is a short of space in the cache, the cache should replace the datum with the lowest chance to be accessed in the future. In general, the selected datum to be replaced will be that of the least recently used.

2.3.2 Instruction Cache vs. Data Cache

In order to make a fast processor, it is common to have separate instruction cache and data cache. There are a few reasons to support the split cache configuration. The first one is for memory transfer bandwidth consideration, with two separate

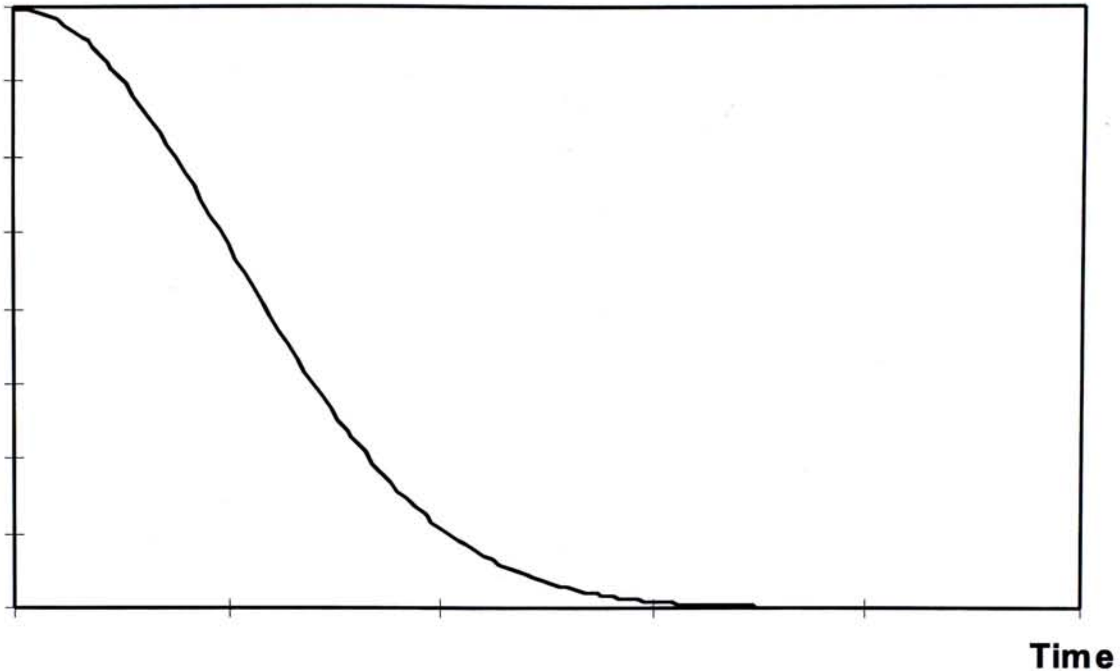


Figure 2.2: Probability Distribution for Temporal Locality

caches serving instruction and data separately, the available memory transfer bandwidth is doubled compare with a unified cache. For machine instruction with data access, the processor has to fetch the instruction code and data to be accessed from memory. If the access path to instruction code and data is separated, both accesses can be carried out in parallel. In particular, the separation of instruction and data paths is called Harvard Architecture [Goo89].

The parameters for cache configuration that will give the best performance are not the same for instruction and data cache in general.

From the above discussion, it is evident that for instruction cache, both the spatial locality and temporal locality will have dominant effect, and it will be seldom to have programs jumping around randomly in the code. The instruction cache should hold the next instructions to be needed, and the size of an instruction

is processor dependent. With today's RISC processor, the instruction size is usually 32-bit in length, the block size for instruction cache is best to be multiples of 32-bit.

For data cache, the best block size depends on the data structure used in the program, and it will be varied from program to program. When a program uses small data, such as integers and floating point numbers, the optimal block size can be smaller, just to be enough to hold these small data. On the other hand, when a program processes big data records, the data cache block cache should be correspondingly larger. Therefore, in order to achieve optimal performance, the separation of instruction and data cache will allow for different parameters to be used.

The third reason to have separate instruction and data cache is to avoid inter-displacement between instruction and data. The number of instructions in a program loop varies together with the data to be processed in that loop. In general, the size of instructions that will be frequently reused added to the size of data that will be referenced in a short span of time constitutes the *working set* for a program. If instruction and data cache is mixed together, it will have a situation that the fetching of some instructions may displace a portion of data already in the cache. By separation of instruction and data cache, these two types of memory will not be in conflict fighting for space in a cache.

As data reference in a program may not exhibit the same locality property as instruction reference, Chen in [BC91] further classified the data access pattern into the following categories:

Scalar Stride

Stride is defined as the distance in address between a previous memory access and the current one. *Scalar Stride* refers to the situation when a simple variable

is used inside a program loop. In general, scalar access will occur when a constant is used, or a running total is accumulated to a memory address. The following program fragments show a scalar access to memory location holding variable *c*.

```
for (i=0; i<100; i++)
    a[i] = b[i] + c;


for (i=0; i<100; i++)
    c += a[i];
```

Zero Stride

Zero stride refers to the situation when an element in an array is referenced repeatedly within a program loop. The difference between zero stride and scalar is that the subscript in the array of a zero stride access will not be changed in the program loop, but it may be changed outside the loop. The following program fragments show a zero stride access to memory location holding array *c*.

```
for (i=0; i<100; i++)
    a[i] = b[i] + c[j];


for (i=0; i<100; i++)
    c[j] += a[i];
```

Zero stride and scalar access show the property of temporal locality, however, with sufficient number of registers in a CPU, the repeatedly referenced datum will usually be set aside in a CPU register, and the memory system will only be updated when the final result is saved. The repeat memory reference pattern will only be visible when the CPU cannot hold all the temporary results in registers and some of them has to be written out to memory.

Constant Stride

Constant Stride means the address distance to the next memory reference is a constant compare with the previous memory access. This kind of pattern is typical for array or record access. As the current program loop may process only a single member in records, the data reference will jump in the forward direction with a constant distance between accesses. With the following code fragment, the memory access to array a and b shows the constant stride feature.

```
for (i=0; i<100; i++)
    a[i] = b[i] + c;
```

From main memory, the constant stride reference pattern can be shown in figure 2.3.

The variable d is termed as the stride of the memory reference to array a (and D is the stride for array b). If d is smaller than the block size of the cache, that means in each cache block, there are more than one elements in it, then the transfer of the cache block provides prefetching property for the memory reference. If data consumption rate for the processor is slower than a cache block filling time, the memory latency for each reference will be that of the cache access time, except when a reference is started at the head of a cache block, at that location, the memory latency will be that of a cache miss, or the main memory access time.

When the stride d is larger than the cache block size, for every memory reference, if the content of the memory location is not already in cache, there will be a cache miss, the memory latency will be very poor for this type of memory accesses. In order to improve the situation, we can make use of a larger block size, in order to encapsulate more than one elements in a cache block, however, it may be impractical, as the stride d may be a very large number, approaching

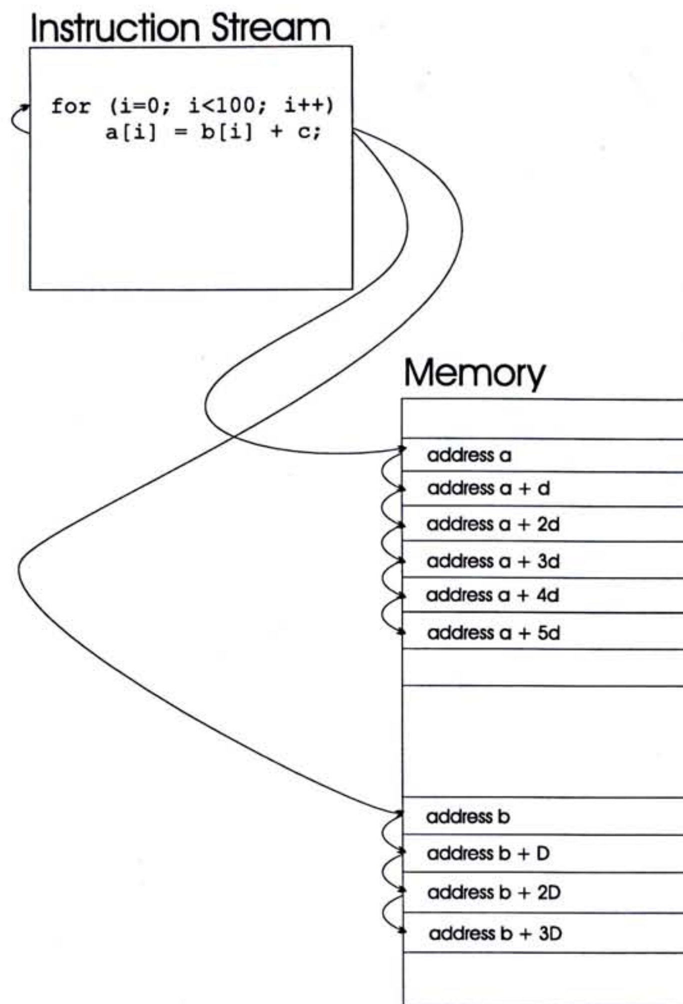


Figure 2.3: Constant Stride Reference Patterns

or even larger than the cache size, it will not be good to have a small number of blocks in a cache, as the granularity is too coarse.

Another method to improve memory latency for large stride access is through prefetching. In this thesis, most prefetching algorithms target on the constant stride access pattern, and their effect will be highly visible in large stride cases.

Irregular Stride

The third type of memory access pattern is *irregular stride*. In programs using linked list or memory pointer operation, the stride value will vary from time to time. The following code fragment shows an irregular stride access on ptr.

```
for (ptr=start; ptr!=NULL; ptr=ptr->next)
    ptr->member1 = ptr->member2 + ptr->member3;
```

As a particular record address depends on the previous record's next pointer, we cannot make prediction on where the next record is until that one is accessed by the processor. There are studies to tackle irregular stride access pattern with prefetching, but they are out of the scope of this thesis.

2.4 Why Not a Large L1 Cache?

When today's processors can have tens of millions of transistors on a single die, it will be natural to use part of the available transistors to implement on-chip cache [Mel95]. On-chip cache has a few benefits over off-chip cache. On-chip cache has a shorter physical distance to the CPU core, and the transmission time spent on electrical wires is shortened. On-chip cache usually has a wider bus width compare with off-chip one, because there is lesser constrain in wiring inside a chip, then to go off-chip. The current generation of processors already has hundreds of pins to connect with the main board, it will be very difficult to add more pins to the packaging of chips. On-chip cache can make use of dedicated bus to communicate with CPU core, but for the same reason as the above, it may not be possible to reserve a large number of pins for dedicated off-chip cache bus.

Whereas on-chip is justifiable, but why not to build a large on-chip single level cache? In fact, it is impractical to build single level cache up to certain size with some technologies (eg. GaAs) [BW88]. We will discuss two considerations in the following paragraphs.

2.4.1 Critical Time Path

From a hardware perspective, a large size cache is actually made up of smaller size memory chips. All these memory chips exert electrical loading onto the address bus, data bus, and control bus of the CPU core. However, the CPU core

is usually designed to drive a specific number of electrical loadings only. In order to drive a large number of chips, a component called bus driver can be inserted between the bus and the memory chips. Bus driver can boast the driving power of the bus to a few times of the original value. But bus driver incurs timing delay. With today's processor of running frequency around 100MHz, each cycle is merely 10ns, and the timing delay of bus driver is in the range of a few nano second. With a really large cache, it may require a two-tier or more bus drivers, which a clock cycle may already be spent in them. This factor limits the size of first level cache, because the memory latency will be significantly increased by inserting one more cycle per memory access.

A better solution is to use multi-level cache hierarchy. The first level cache is kept to be small in size, and serving the most likely requested data. Another layer of cache serves the less likely requested data. Then, the number of chips on the first level cache can be kept to a minimum, the number of bus drivers and hence the timing delay can be reduced.

2.4.2 Hardware Cost

To build memory cells with access time in the nano second range is expensive. For dynamic memory cell, only a capacitor is required to hold a binary bit of datum. But dynamic memory cell is slow in access time, with current technology, the access time is around 50-70 nano second. Static memory cell has a faster access time. With 2 transistors per binary bit, they can build a memory cell with access time in the range of 20 nano second. In order to build still faster memory cells, a 6-transistor architecture can be used. However, the number of transistor count will be tripled.

The first level cache is usually multiple-ported in current design, in order to let different units in the CPU to access the first level cache in parallel. To build

multiple-ported cache, one way is to duplicate the memory cells, then each bank of memory cells can serve a request from different units. The result is that, in order to make a very fast first level cache matching the cycle time of the CPU and to be multiple-ported, a lot of transistors are used to build memory cells. The number of transistor count to build a single memory cell for first level cache may be a few times the required number of transistor to build a second level cache. Therefore, only the most frequently requested data are justified to be placed into the first level cache. Other data will be best to be placed into slower, but cheaper to build, second level cache or main memory [Wan89].

2.5 Trend to have L2 Cache On Chip

The above discussion shows why a large on-chip level one cache is not practical, instead with enough transistor budget, CPU designers tend to build multi-level cache on the CPU die. On chip second level cache is beneficial to the processor performance, as firstly, the total cache size can be increased without adding extra electrical loading on the CPU core. Secondly, the second level cache can be made of cheaper but slower design, because the majority of memory references are expected to be served by the first level cache. Thirdly, the on chip second level cache can make use of CPU signals that will not be available off chip. These CPU signals include current program counter, instruction type, addressing mode, name of CPU registers in use for the current instruction, branch prediction from the branch unit, prefetch queue information, etc. These CPU signals will be too cumbersome to be available off chip, as the number of pins required will be too large. These signals can help in making prediction for future CPU memory access. New prefetching algorithms can be devised for the cache to prefetch memory locations to be required in the future.

There were previous studies ([SL88], [MeM95], [TaS94], [Liu94], [HaJ92], [NeA91], [Sez93], [ChK91], [BuK90], [APB92], [OyW92], [JoW94]) for performance on multi-level cache hierarchy, although they may not focus on the on chip second level cache design. For on chip second level cache, there were studies in [HuO94], they tried to find out the best configuration parameters for on chip multi-level cache, but not suggesting cache prefetching algorithm. The necessity to have more than a single level of cache to bridge the speed disparity is there for a long time.

The following are a few examples that commercially available processors have on-chip second level cache.

2.5.1 Examples

DEC Alpha 21164

Digital's Alpha 21164 CPU has an 8k Bytes on-chip first level instruction cache, with 32 Bytes blocks. The first level instruction cache is direct mapped. There is also an 8k Bytes on-chip first level data cache. The block size is 32 Bytes, and the update policy is write through. There are two read ports on the data cache, that means two memory references can be outstanding simultaneously.

The Alpha 21164 CPU also contains a second level cache of 96k Bytes. The second level cache is a mixed cache shared by instruction and data. The on chip second level cache uses write back update policy. The on-chip second level cache uses 64 Bytes blocks and it has a set associativity of 3.

The following is an extraction from the designers of Alpha 21164 CPU to justify the reason to have two level cache architecture instead of a large single level cache.

> "Two-level Data Cache. Many workloads benefit more from a

> reduced latency in the data cache than from a large data cache. We considered a single-level design for a large data cache. For circuit reasons, physically large caches are slower than small caches. To achieve a reduced latency, we chose a fast primary cache backed by a large second-level cache. As a result, the effective latency of reads is better in the Alpha 21164 CPU chip than it would have been in a single-level design.
>
> The two-level data cache has other benefits. The two-level design makes it reasonable to implement set associativity in the second-level cache. Set associativity enables power reduction by making data set access conditional on a hit in that set. The two-level design also allows the second-level cache to hold instructions, which makes a larger instruction cache unnecessary."

The Alpha 21164 CPU also contains logic to control off-chip cache. If that level of cache is implemented, it will become third level cache. The off-chip cache can have size range from 1MBytes to 64MBytes. The third level cache is direct mapped, as the number of pins available on the CPU package is limited. The third level cache uses write back update policy.

Intel Pentium Pro

The Intel Pentium Pro contains an 8k Bytes on-chip instruction cache and an 8k Bytes on-chip data cache. The two on-chip first level caches are non-blocking, means the processor can proceed to process other instructions even when there is a cache miss. Only when there is dependency on the cache miss data that the processor has to stall and wait for the cache miss to be served.

The Intel Pentium Pro processor also contains a 256k Bytes or a 512k Bytes on-package second level cache. The second level cache is actually implemented

on a separate die as the CPU core, however, all the CPU core together with the second level cache dies are housed in a single package. The 2-die approach is a tradeoff between size of second level cache and the ease of manufacturing. The 256k Bytes second level cache is made of 15.5 million transistors and the CPU core is made of 5.5 million transistors. If all these 21 million transistors are made on the same die, the yield will not be commercially feasible for current technology. On the other hand, in order to make a large second level cache, a large number of transistors is required. The same packaging technique to house the CPU core and the second level cache makes the communication between them as fast as possible. And due to wire bondings are used to connect the 2 dies, there are more signals that can be propagated from the CPU die to the second level cache, because connection pins are not required.

2.5.2 Dedicated L2 Bus

There are some current microprocessors with on-chip second level cache control logic, but there is no memory cell implemented on chip. This can be a balance point to have the benefits of on-chip CPU signals for second level cache management but does not incur the production difficulties to implement a large number of memory cells. The following are examples of this category of processors.

MIPS R10000

The MIPS R10000 processor has a 32k Bytes 2-way set associative, 2-way interleaved on-chip first level data cache with LRU replacement policy. The block size of the on-chip data cache is 32 Bytes. The update policy of the on-chip data cache is write back. The MIPS R10000 also has a 32k Bytes 2-way set associative instruction cache. The block size of the on-chip instruction cache is 64 Bytes. There is also a dedicated 128-bit second level cache bus with all required signals

to drive off-chip static memory chips. The cache management logic for the second level cache is built on chip. The cache implemented on the MIPS R10000 processor is non-blocking.

PowerPC 620

The IBM/Motorola PowerPC 620 processor has a 32k Bytes 8-way set associative, non-blocking data cache. The on-chip data cache can be configured to use write back or write through update policy. There is a separate 32k Bytes 8-way set associative on-chip instruction cache. In the CPU die, there is second level cache management logic, the external interface is a dedicated second level cache bus with 128-bit bus width, and implementing a direct mapped mixed instruction and data off-chip cache.

Nexgen Nx6x86

The Nexgen Nx6x86 is an Intel Pentium compatible processor with 32k Bytes 2-way set associative, dual-ported on-chip first level data cache. The Nx6x86 also contains a 16k Bytes 2-way set associative on-chip first level instruction cache. There is on-chip second level cache management logic, and the interface to the off-chip memory chips is a dedicated second level cache bus with bus width of 64 bits. The off-chip second level cache is a mixed instruction and data cache using write back update policy.

2.6 Hardware Prefetch Algorithms

Due to locality property, it is possible to make predictions to future processor memory references by observing the memory access pattern. *Hardware Prefetching* makes use of historical data to deduce future memory access locations. In

order to reduce the memory latency for compulsory miss, prefetching can be performed on cache memory when the memory bus is idle. If the prediction of future memory reference is accurate, the memory access time will be that of the cache memory instead of the slower main memory.

2.6.1 One Block Look-ahead

Alan Jay Smith in [Smi82] studied a general algorithm for hardware prefetching. The idea is when the processor accesses a cache block i, due to spatial locality, cache block $i+1$ will also likely to be referenced in a short time. The prefetching algorithm is to trigger the transfer of cache block $i+1$ from main memory if it is not already in cache. This type of prefetching algorithm is called *One Block Look-ahead* (OBL). There are three variations in OBL that in [Smi82] had studied. The variation depends on when the prefetch will be performed. One way is to prefetch cache block $i+1$ if reference to cache block i causes a cache miss, it is called *Prefetch On Miss.* The other way is to prefetch cache block $i+1$ if reference to cache block i causes a cache hit, it is called *Prefetch On Hit.* The third way is to prefetch cache block $i+1$, no matter reference to cache block i is a cache hit or miss, it is called *Always Prefetch.*

The effect of OBL is similar to a cache system with larger cache block size. Just the memory transfer is broken down into smaller pieces in OBL case, whereas large cache block size initiates a lengthy transfer when there is a cache miss. However OBL improves over large cache block size by maintaining the larger number of blocks in a fixed size cache. The access to cache block i is treated as a confirmation that the cache block may be accessed again in the future due to locality.

OBL and large cache block size work well in instruction cache, as the reference pattern has high sequentiality. However, in data cache, the stride value varies

from case to case. In order for large cache block size to reduce compulsory miss, the stride value should be smaller than the cache block size, or every access to a new datum will cause a cache miss. For OBL, the stride value has to be smaller than twice the block size, because whenever cache block i is referenced, cache block $i+1$ will be fetched, this effectively doubles the apparent block size. With large stride value, both OBL and large cache block size will perform very poor, as the prefetching in OBL and the large cache block size will bring in pollution to the cache, the cache hit rate will be decreased.

In [DDS95], Dahlgren, Dubois, etc. enhanced the OBL scheme to work on a shared memory multiprocessor environment. They proposed a method to adjust the look ahead level to tune the cache prefetch algorithm behavior.

With a splitted instruction and data cache design, there were studies ([YoS93], [YeP93]) of cache prefetch algorithm for the instruction cache. They focused on how the cache prefetch algorithm can continue to prefetch when there are branch instructions in the program.

2.6.2 Chen's RPT & similar algorithms

Baer and Chen in [BC91] proposed a hardware prefetching algorithm basing on a *Reference Prediction Table* (RPT), a *Branch Prediction Table* (BPT) and a *Look-ahead Program Counter* (LA-PC). The RPT keeps track of CPU instructions to issue memory accesses, the difference in memory address from previous access and a state field. The LA-PC works with the BPT to make prediction to future *Program Counter* (PC) value, if the predicted PC address has a corresponding entry in the RPT, the future memory access location can be calculated by the formula

future memory access location = previous memory access location + stride

That future memory access location will be prefetched into the cache memory,

and if the LA-PC keeps a far enough distance from the actual PC, the prefetch may have time to complete the memory transfer before the datum is actually requested by the processor.

The state field in each RPT entry is to filter out non-constant stride memory accesses and prevent the prefetching algorithm to issue erratic prefetches to pollute the cache memory. The state field has the following possible values:

- initial
- transient
- steady
- no prediction

Cache prefetch will only be issued for RPT entry in steady state.

Fu and Patel in [FP91] studied the stride values for memory access in vector processors. They measured the miss rate with no-prefetch, sequential-prefetch, and stride-prefetch algorithms. The sequential-prefetch and stride-prefetch algorithms reduced the cache miss rate over no prefetch case. Between the sequential prefetch and stride prefetch algorithms, the later one had a better performance due to that one can make accurate predictions to large stride access pattern.

Fu etc. in [FPJ92] further the above architecture to support non-vector processor by using a Stride Prediction Table (SPT). The structure of SPT is similar to the RPT proposed by Chen. in [BC91].

Issues in Chen's RPT

The Chen's RPT prefetching algorithm can tackle constant stride memory reference, no matter the stride value is smaller or larger than the cache block size. The cache pollution problem is reduced by using the state field to filter out uncertain

memory access patterns. In order to arrive the steady state, there should be two consecutive memory accesses by a CPU instruction with the same stride value.

The Chen's RPT prefetching algorithm has a good performance, however, there are still issues in the design of the algorithm that we put forward for discussion.

RPT Size One crucial component in the Chen's RPT scheme is the Reference Prediction Table. That table is used for storing the memory accessing CPU instructions, and stride values of their memory accesses. How many entries should be in the RPT? In a general RISC program, about 20% to 25% of instructions are load/store instruction with memory references. The RPT should capture the number of load/store instructions in the largest program loop in order to be effective in predicting future memory references. If the RPT is too small, an entry in it may be replaced by another instruction before the loop goes to the next iteration. In [BC91], Chen used a 512-entry RPT, and he estimated that the hardware overhead in implementing a 512 entry RPT is roughly equivalent to a 4k Bytes cache.

The hardware overhead to implement RPT is significant. If that can be reduced in size and retaining a similar performance, the saved space can be used to implement a larger cache.

The problem in the RPT implementation lies in using the instruction address as the index to the RPT. There are a lot of instruction addresses which contain load/store instructions. And they all have the potential to go into the RPT. The state field in the RPT stops the irregular stride memory access instructions from triggering prefetch, but it does not prevent the instruction address itself to pollute the precious RPT. In this thesis, we proposed a novel idea to use the source register in a load/store instruction as the index to a RPT like data

structure. Due to the limited number of registers in a processor that can be used as the source register, the data structure can greatly reduce in size, and at the same time, to retain prediction accuracy for cache prefetching.

Replacement Algorithm in RPT The Chen's RPT is in itself like an Instruction Cache without actual data fields. Just like normal cache, the associativity of the RPT affects where an entry can go into the RPT. In order to make RPT set associative, there require address comparators working in parallel to determine whether a hit is recorded in the RPT.

In [Che93], Chen reported that the performance of a direct mapped RPT is not significantly differ from a set associative RPT. However, if set associative RPT is used, what replacement algorithm is appropriate for the RPT has not been discussed. Least Recently Used (LRU) algorithm is popular in set associative cache implementation, in RPT, LRU replacement algorithm may be a good choice. Another possibility may be First In First Out (FIFO), or even random replacement.

LA-PC and BPT The LA-PC and BPT in Chen's RPT scheme are used to make prediction to instructions that the CPU will execute in the future. The LA-PC and BPT are in themselves complex hardware. The accuracy of the LA-PC will degrade if the distance between the LA-PC and the actual PC is far away, due to insufficient information contained in the BPT. How far away should the LA-PC ahead of the actual PC? Chen in [Che93] suggested that the distance should be enough to compensate for the memory latency incurred to transfer a cache block from main memory, then, the prefetch will be fully completely when the processor requests the data. The number of entries in the BPT is an issue similar to the the RPT size. The more entries in the BPT, and if set associative BPT is used, the more useful will be the BPT. But all these features add to the

transistor count required to implement them.

2.7 Software Based Prefetch Algorithm

In contrast with hardware prefetch algorithm, software based prefetch algorithms make use of the static information generated and analyzed by the language compiler. Minimal additional hardware is required using software based prefetch algorithms as the compiler will insert special instructions into the executable code. The processor will ignore all these special instructions and treat them as NO-OPERATIONs (NOPs). The cache system will monitor the instruction queue, and act according to the special instructions.

Omar in [Oma81] proposed models for software based cache management. In the Prompting model, two instructions were used, they are Prompt and Release. These instructions were used to allocate and deallocate a cache block explicitly. The model treated the cache memory similar to processor registers, as a cache block would not be loaded or replaced on demand, but according to the cache specific instructions. The language compiler had to decide when to load a cache block and when that block is not required anymore, just like the problem of register allocation.

2.7.1 Prefetch Instruction

Porterfield in [Por89] suggested to use language compiler to insert prefetch instructions into the executable code. These prefetch instructions would give the cache system a hit what memory locations are going to be accessed. It is common to insert prefetch instructions for memory locations required in the iteration $i+1$ when the processor is executing a loop in iteration i, then, the scheme is called *One Iteration Look-ahead.* In order for this scheme to work, the memory latency

for prefetching should be shorter than the time required to execute an iteration in a loop. The cache misses could be all eliminated if the above criterion is met. Porterfield reported such results on RiCEPS benchmark.

Prefetch instruction consumes an entry in the instruction queue, which usually translates to at least a cycle of execution time. In modern CPU architecture, the instruction unit may ignore the prefetch instruction, however, the size of the instruction queue is effectively shortened by the prefetch instruction. If there are a lot of prefetch instructions in a row, the instruction queue may not contain another proper instruction without dependency for the execution unit to run, then the CPU will stall.

In general, the prefetch instructions increase the instruction count in a program, and these prefetch instructions will in themselves cause timing overhead on the processor. Whether a software based prefetch algorithm can improve performance depends on the saving in memory latency for less cache miss compare with the extra time that the CPU has to spend in ignoring the prefetch instructions.

There are other studies to improve software based prefetching algorithms in [GGV90], [MLG92], [KL91], [Por89], etc. Basically, they tried to reduce those unnecessary prefetch instructions and hence to improve the overall performance of the system. They studied compiler designs, to dig out more information from the source programs, and memory hierarchy, in order to better schedule the timing for the prefetch instructions.

Through software based prefetching, cache hit rate can be improved due to the advance information that is given by the prefetch instructions. There is relatively less cache pollution problem using software based prefetching compare with hardware prefetching algorithms, because the accuracy of prefetching is very high. However, there is non-negligible overhead in the prefetch instructions. In order for the whole system to improve performance, the save in memory latency

should be greater than the overhead.

2.8 Hybrid Prefetch Algorithm

In [Ho95], Ho proposed a prefetching scheme using a mix of compiler generated special instructions and a hardware table implemented in a processor for data prefetching. The scheme was designed to significantly lower the overhead of conventional software based prefetching scheme and at the same time, to make use of compile time generated and analyzed information from the source program, hence to retain the accuracy of prefetching.

The hybrid prefetching scheme is called *Stride CAM Prefetching.*

2.8.1 Stride CAM Prefetching

The *Stride CAM Prefetching* scheme was proposed in [Ho95], a hardware structure called *Content Associative Memory* (CAM) was setup to hold the stride values that can be used to trigger cache prefetches. The scheme was designed to tackle constant stride access specifically. In this type of memory access, the stride value is an invariate inside a program loop. By using a new cache specific instruction called *setcam*, the stride value for a particular instruction is set inside the CAM.

The CAM structure is similar to the RPT used in [Che93], but without the last data reference address and the state field. The structure is shown in figure 2.4.

The usage of the fields is the same as in the RPT algorithm. Just before the beginning of a program loop, a set of setcam instructions are to be inserted there to set the stride values fixed in compile time. When the loop is running, if there is a match on the instruction address in the CAM with the current Program

instruction address	stride	store bit
0x12345678	8	1
0x27629435	4	1
0x34294927	64	1
0x48267101	4	1
0x28478291	32	1
0x28190871	4	1
⋮	⋮	⋮
⋮	⋮	⋮

Figure 2.4: Content Associative Memory

Counter, a prefetch will be trigger by adding the current memory access location (effective address) with the stride value stored in the CAM. That is why the last data reference address is not stored in the CAM as in RPT, because that address will be available during the run time.

The Stride CAM prefetching scheme implements *One Iteration Look-ahead* normally as the current PC is compared with the instruction addresses in the CAM.

The CAM is an associative memory. When the PC is compared with the entries in the CAM, all the comparisons will be done in parallel. However, for the setcam instruction, an explicit CAM entry number will be given as a parameter to set a specific entry inside the CAM.

Due to the fact that the setcam instructions are executed outside the loop, the overhead for the processor to handle these instructions compare with the conventional prefetch instructions are greatly reduced. The complexity of the new prefetching algorithm is in the order of O(1) instead of depending on the

number of iterations as with the conventional prefetch instructions.

The hardware overhead to implement Stride CAM Prefetching is considerably less than that required to implement pure hardware scheme such as Chen's RPT. There is no need to have LA-PC and BPT in the Stride CAM Prefetching model, and the CAM contains less fields than Chen's RPT. The performance of the Stride CAM Prefetching should be better than that of Chen's RPT because the accuracy of prefetch is very high, there should be very little cache pollution caused by the Stride CAM Prefetching scheme.

The Stride CAM prefetching is a great enhancement to other software based prefetching schemes, however, it still requires the compiler support to generate the setcam instructions.

Chapter 3

Simulator

A computer system simulator called *Multi-level Memory Hierarchy Simulator* (MMHS) was developed to gather performance data for the prefetching algorithms tested in this study. The MMHS is a flexible, efficient and handy tool to try new algorithms.

3.1 Multi-level Memory Hierarchy Simulator

The MMHS is a set of programs which take an address trace stream as input, and follow the memory access cycle-by-cycle in all memory hierarchy levels. Prefetching algorithm to be tested can be plugged into holders designed in the MMHS. Performance statistics are recorded. The following are major metrics reported by the MMHS

- Cycles Per Instruction
- Hit Rate
- Memory Latency
- Memory Idle Percentage

- Total Memory References

The MMHS supports the following configurable parameters.

- Cache Block Size
- Cache Set Size
- Cache Associativity
- Cache Write Policy
- Cache Replacement Policy
- Cache Memory Set-up Cycle
- Cache Memory Burst Cycle
- Bus Width
- Non-blocking Feature
- Write Buffering Feature

The priority arrangement of memory references in MMHS ensures all CPU demand fetches will have a higher priority than all prefetch requests. If memory requests with the same priority are in a queue, they will be served in a First In First Out (FIFO) manner.

When a processor demand–fetch for a memory location is issued and a prefetch operation is taking place, the prefetch will be killed and let the demand fetch to be started. This assures that prefetch operations will not compete for bus bandwidth with processor demand–fetch.

The simulation speed of the MMHS is an important factor for its usefulness. In order to study multi-layer memory hierarchy performance, a lengthy simulation, in terms of CPU instructions, has to be run. Because the upper memory

layers may already have a high hit rate, and most memory requests are served in those layers. To simulate environments with enough memory references on lower memory layers, a fast and reliable simulator is necessary to make trial runs to be finished in reasonable time.

The MMHS employs clever data structure and programming techniques to speed up simulation speed. The techniques include pre-sort scheduling queues, double link lists for fast forward/backward search, constants pre-calculated at the beginning of simulation, and use of shift operations instead of multiply/divide, strong favor to use integer arithmetic instead of floating point operations, etc.

3.1.1 Multi-level Memory Support

The Multi-level Memory Hierarchy Simulator based on a realistic model of computer system. A typical configuration is shown in figure 3.1.

Blocks in the above figure depict memory architecture layers in a computer system, and lines show bus interconnections between layers.

The MMHS does not depend on a fixed layer model, but is flexible to configure with variable number of layers in the simulated environment. The interconnections between layers in MMHS are built during run time, and no program logic change is necessary to simulate a different setting.

Write buffers can be configured to handle write operations. The number of entries in the buffer is configurable for each layer separately.

3.1.2 Non-blocking Cache

The Multi-level Memory Hierarchy Simulator supports transactional bus model with nonblocking feature. That means if a memory request is a miss in the current memory layer, another memory request can be started in the next cycle, provided that there is no data dependency on the missed datum. If the new memory

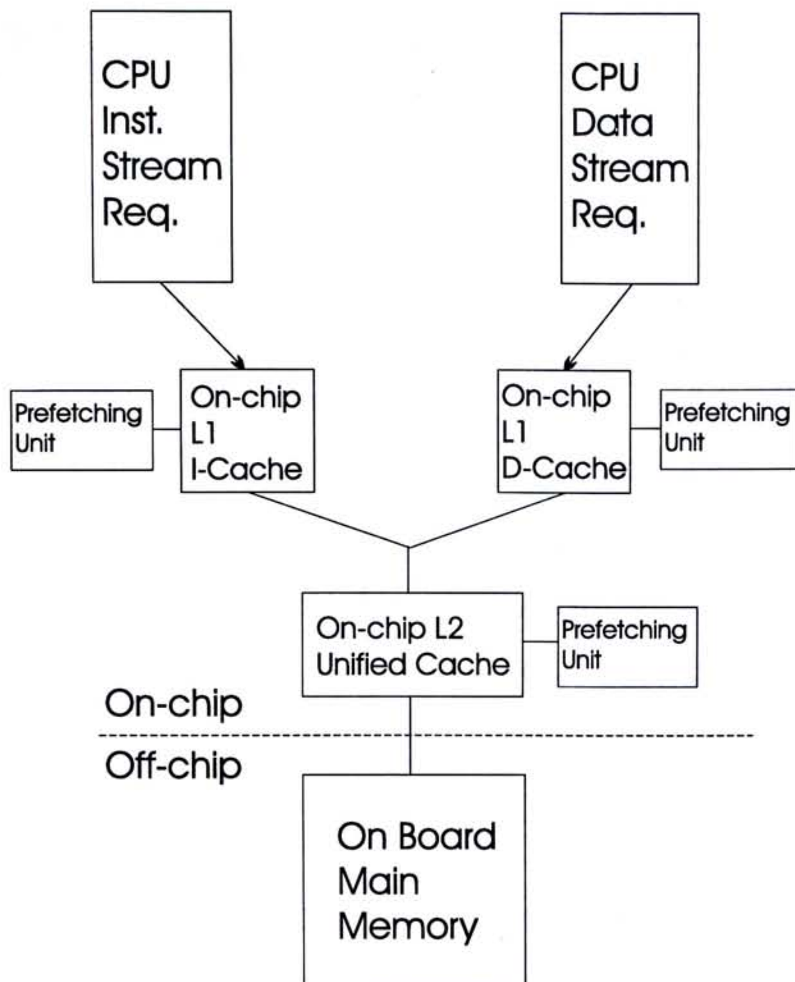


Figure 3.1: Multi-level Memory Hierarchy Simulator Configuration

request is a hit, then the later memory request may actually be completed before the earlier request.

Non-blocking cache is useful when processor(s) can issue multiple instructions in a clock cycle or when memory layers can issue their own prefetch requests. In that scenario, there may have multiple outstanding memory requests on each memory layer, and if that layer can serve multiple memory requests, the system throughput will be improved.

The non-blocking cache feature can be turned off by a programmable flag in each memory layer separately. If it is off, whenever there is a miss in a particular layer, all further memory requests will be stalled until the miss is served.

3.1.3 Cycle-by-cycle Simulation

The MMHS simulates the target system down to single CPU cycle. That means memory requests in different memory layers can be run in parallel and overlapping is allowed as long as the configuration permits.

Cycle-by-cycle simulation also makes precise Cycles Per Instruction (CPI) measurement possible as oppose to only Hit Rate measurement. Because the memory latency from each memory layer can be counted with CPU cycle precision, the total number of cycles required to satisfy an CPU instruction can be accounted for by using the MMHS. Memory CPI (MCPI) as suggested by Chen in [BC91] can be computed by subtracting the number of cycles consumed by the processor for computation.

The cycle-by-cycle simulation model makes system parameters such as memory cycle time and hit/miss penalty on each memory layer have complete flexibility. Because the lowest common time unit in the MMHS is CPU cycle, the MMHS system can support arbitrary number of cycles in the above parameters.

3.1.4 Cache Prefetching Support

The MMHS supports cache prefetching algorithms on each memory layer. The prefetching algorithm can be different for each memory layer, and in fact, there is no requirement to have prefetching in all memory layers.

Prefetching algorithm should be written as a C callable function. The prefetching function will be called each cycle after the activity in a memory layer is finished. The prefetching function can access to current statistics in all memory layers, issue prefetch memory request to any memory layer, or update statistics maintained by its own.

Chapter 4

Proposed Algorithms

In this chapter, three cache prefetch algorithms would be proposed. The justification for each algorithm would be stated together with the detail architectural model.

A section is devoted to discuss the rationale for a combined on chip first and second level cache management. The feasibility and usefulness of the idea would also be studied.

4.1 SIRPA

SIRPA stands for *Source Index Register Prefetch Algorithm.* It is a hardware cache prefetch algorithm which handles constant stride access pattern. SIRPA is a highly selective prefetch algorithm. The accuracy of it is very high, and the hardware overhead to support it is minimal.

4.1.1 Rationale

In RISC architecture, only load/store instructions are allowed to transfer data to/from main memory. This feature reduces the number of instruction types which are capable to work on the content in main memory. Due to limited

addressing modes in RISC, in order to access an array in main memory, it is common to use a CPU register to hold the current element address in the array. Whenever there is an access to the array, indirect addressing mode, or some variants of it, will be used to access the main memory. The CPU register used provides an important hint for the hardware to know which array is currently being accessed. By keep tracking the stride value of each array, it will be feasible to perform cache prefetching efficiently.

The proposed *Source Index Register Prefetching Algorithm* (SIRPA) makes use of a hardware implemented table to keep track of the stride values. When there is a memory accessing CPU instruction using a register indirect addressing mode, the source register in that instruction will be used as the index to the above table. The difference between the current effective address and the previous effective address, which is stored in the table, will be compared with the stored stride value, if there is a match, that means at least two consecutive accesses to the main memory through that CPU register has the same increment (or decrement) in effective address, that will be a strong indicator for a constant stride access pattern. Cache prefetch will be fired by adding the stride value to the current effective address, where that address is predicted to be required in the next access.

Source register in CPU instruction is a good candidate to discover stride values as it is logical to the compilation process. The SIRPA is a pure hardware oriented cache prefetching scheme. There is no need to modify the software or the instruction set to use SIRPA. That implies the very broad applicability of SIRPA, as it does not depend on a particular CPU architecture.

The structure of the Stride Values Table (SVT) used in SIRPA is similar to Chen's RPT, with one big exception that the Source Index Register is being used as the index to the SVT, whereas in Chen's RPT, an instruction address is used

as the index.

The cache prefetching accuracy of SIRPA should be very high, as there is a confirmation mechanism built in to verify the constant stride access. The cache pollution effect for the SIRPA should be very small. On the other hand, the majority of constant stride accesses should be captured by the SVT. The SVT will fail when there are a lot of arrays to be accessed in a program loop, so that the number of registers in the CPU are not enough to hold all the array addresses. However, as there are 32 or more CPU registers in today's RISC processors, it will be less likely that so large number of arrays has to be accessed in a program loop.

4.1.2 Architecture Model

SVT Size

The SVT in SIRPA serves a similar role as the RPT in Chen's RPT scheme. However, due to only a fixed number of registers are capable to be used as source register in indirect addressing mode, the number of entries in SVT is determined readily. In general, only the integer registers in a RISC processor are capable to be used as source register, as opposed to the whole bank of floating point and integer registers. The number of integer registers is around 32 to 64 in today's RISC CPUs.

The above fact compares favorably to Chen's RPT scheme. In Chen's RPT, the optimal number of entries in the RPT is not easy to be determined, as instruction address is used as index to the RPT. But the number of instructions in a program is variable. It is inevitable in RPT's case that a lot of program instructions will jam into the RPT, but they are not the candidates for constant stride access.

Less number of entries in SIRPA's SVT is crucial. By using less memory

in implementing the SVT, more transistor count can be saved to make a larger on-chip cache or with a higher set associativity. The kind of memory cells to build the SVT should be very fast, as the processor has the potential to access the main memory through indirect addressing mode in consecutive sequence. In order to avoid blocking the processor and still be able to update the entries in the SVT, the speed of the SVT should be less than the access time of the first level cache.

We have performed tests on similar configured SIRPA and RPT cache. To achieve comparable performance, the number of entries in RPT should be around four times the number of entries in SIRPA's SVT.

Intuitive Replacement Algorithm

As there are a fixed number of registers in a processor, we can build a SVT with sufficient entries to hold potential stride value for each register. The replacement algorithm being used in the SVT is intuitive, as there will be no "conflict" in the SVT itself. However, in other schemes, when the number of entries in the stride value discovery table is so much less than the number of possible candidates, there will be a design decision to use direct map or set associative organization in the table.

For direct mapped design in the stride value discovery table, the replacement algorithm is straight forward. However, there will be conflicts for two or more potential candidates to fight for the same entry in the table. If set associative organization is used, the number of conflicts can be reduced, but it will require the technologically demanding associative comparators. Replacement algorithm being used in the set associative design is another issue for consideration, because different replacement algorithms, such as LRU, FIFO, random, will lead to different behaviors.

No LA-PC and BPT

In the SIRPA scheme, there is no need to implement the LA-PC and BPT as in the Chen's RPT scheme. The triggering of cache prefetch is done when a CPU instruction is issued to the execution unit. If that instruction is a memory accessing instruction using register indirect addressing mode, the SVT will be used to check for constant stride access. If the stride calculated is the same as the value stored in the SVT, a cache prefetch for the next predicted memory address will be fired. If the CPU instruction causes a cache miss in the cache, the CPU instruction will be given priority to the cache prefetch request in using the memory bus. The SIRPA scheme is simple and easy to be implemented in hardware.

The LA-PC and BPT as used in Chen's RPT scheme are very complex hardware. The cost to make predictions to future program counter value is high. The BPT in itself is a cache to store the last branch actions performed by the processor. There are design issues to the number of entries in the BPT, organization and replacement algorithms being used in it.

Hardware Support

To implement the SIRPA scheme, the SVT has to be built between the CPU core and the cache management unit. Figure 4.1 is a block diagram showing the configuration of a SIRPA enhanced processor.

The SVT is organized by using source register as index to the table. The SVT contains several data fields as shown in figure 4.2.

The state field contains the current status for the entry and has the following values:

- initial

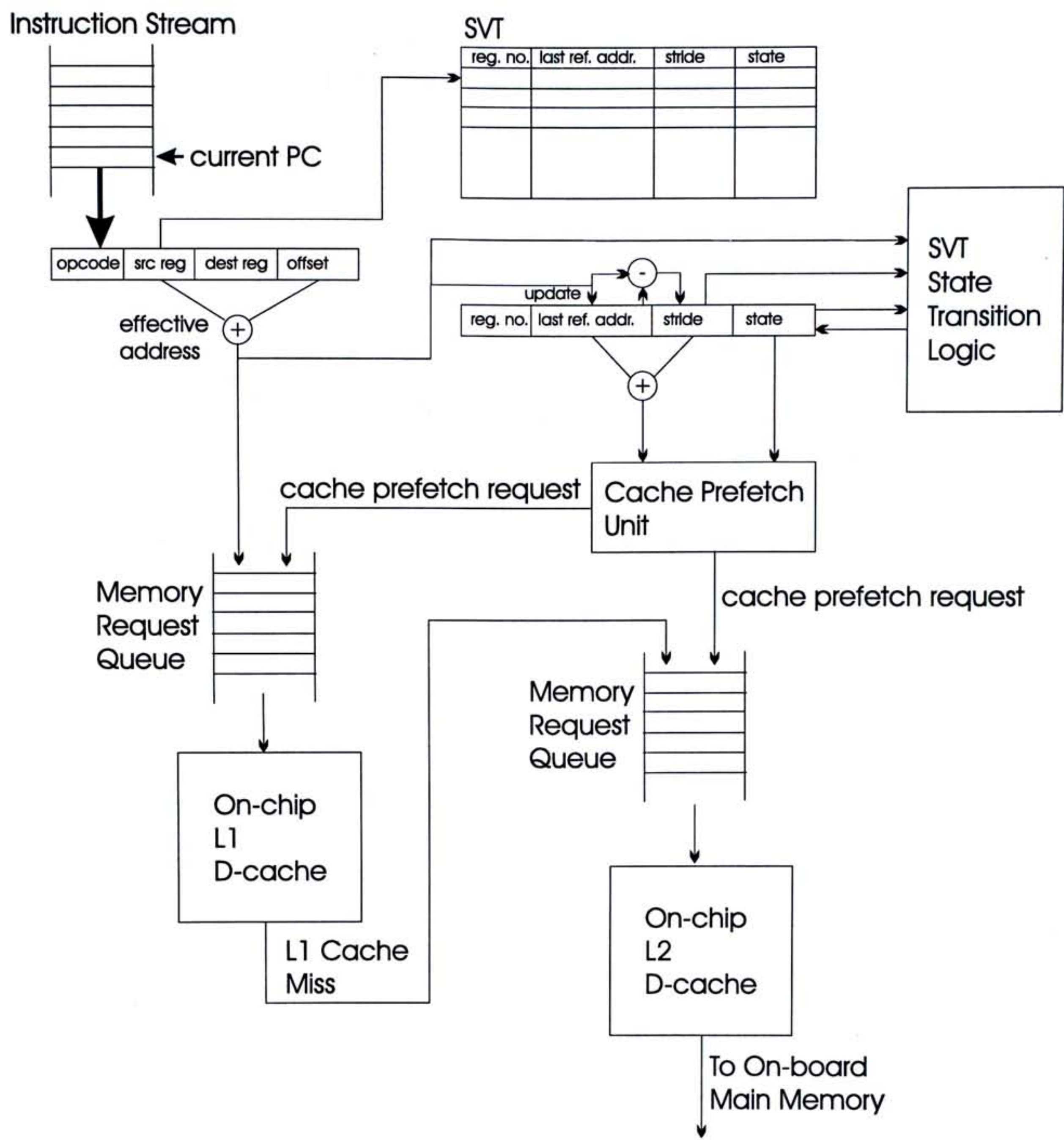


Figure 4.1: Block Diagram for Processor with SIRPA scheme

- transient
- steady
- no prediction

Details of the Algorithm

Whenever the CPU executes the following instruction:

source reg.	last ref. addr.	stride	state
r1	0x10003456	0x8	transient
r2	0x20003334	0x4	initial
r3	0x20007654	0xC	steady
r4	0x38883333	0xE	no prediction
r5	0x87654321	0xC	no prediction
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮

Figure 4.2: Source Register Indexed Stride Values Table

```
ld rm<--(rn+S)              ;S is a displacement
st rm-->(rn+S)
```

The value (rn+S), which is the effective address of the current access, will be calculated, and the entry in the SVT using rn as index, will be updated, the last reference address field will be updated with the (rn+S) value just calculated. The stride field will be the difference between the current (rn+S) value and the value before update. The state field will be updated by using the state transition diagram in figure 4.3.

After the update is finished, if the state value is "steady" for the entry just updated, a prefetch for address:

prefetch address = (last reference address + stride)

will be fired. The state value of "steady" indicated that the stride value for the entry is confirmed by at least 2 consecutive updates of that entry. When a new stride value is discovered, the state value will be "transient", and if the next data reference has the same stride value as the last update, then the state value will be promoted to "steady", otherwise, the state value remains "transient".

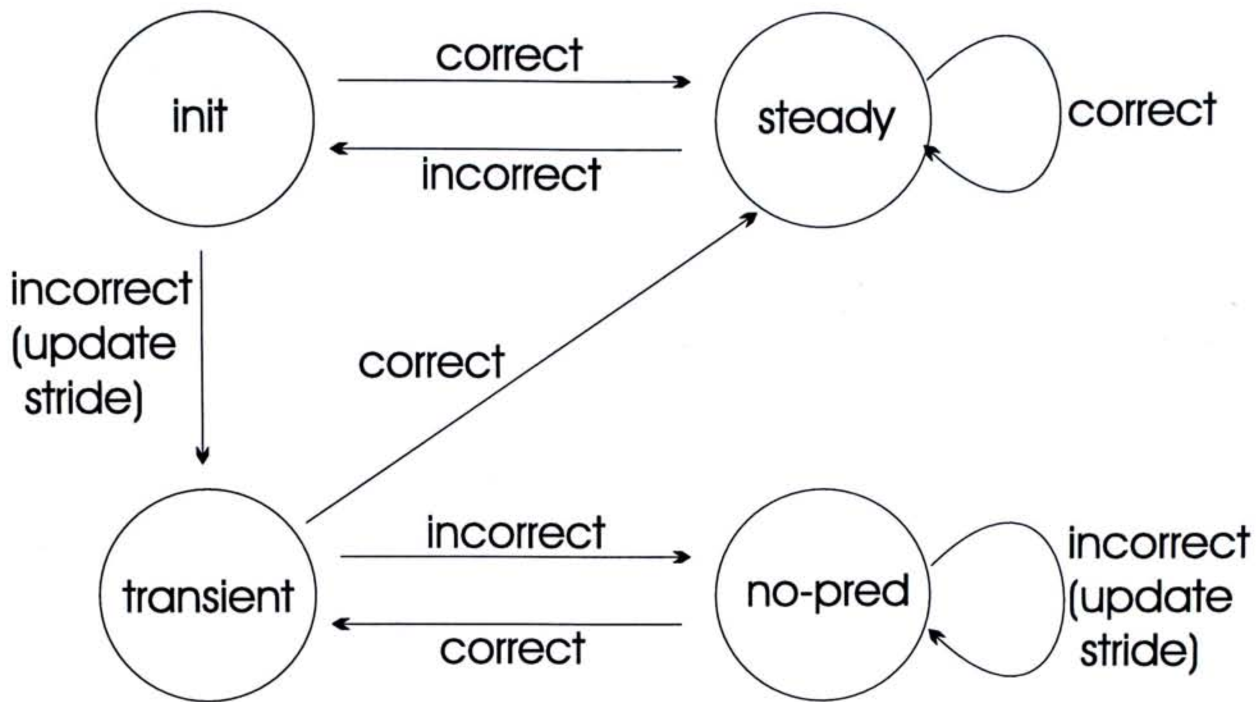


Figure 4.3: SVT State Field State Transition Diagram

The SIRPA scheme is very accurate for predicting the memory access of the following program fragment:

```
for (n = 1; n < 100; n++)
{   a[n] = b[n] + c[n];
    d[n] = e[n] + i;
}
```

The above program will likely be compiled into machine code where a CPU register (plus offset) will point to array a, and other registers pointing to array b, c, d, e respectively. The stride values for the above registers will then be the record size of the respective array. By recording the stride values, the access patterns can be predicted with high accuracy. The use of source register as the index in the stride discovery table reduces the size of the SVT, because most micro-processors nowadays have 32 registers, the maximum size of the SVT is limited by the number of registers that the processor contains.

The SIRPA scheme can be used to trigger cache prefetches for first and second level on-chip cache. Due to difference in block size in the two levels of cache, the cache block being prefetched may not be the same in them.

4.2 Line Concept

Line Concept is a technique to capture the cache block size information to fine ture a cache prefetch algorithm. By using Line Concept, the cache prefetch would have a longer time slot for the prefetch to take place. Hence, the chance to have a partial miss would be reduced.

4.2.1 Rationale

The benefit of cache prefetch can be realized when CPU computation and memory transfer for the cache block being prefetched can be overlapped. There are situations when the cache prefetch algorithm can make an accurate prefetch, but there is not enough time to transfer the cache block being prefetched onto the cache. Then, at the time when the processor requests a memory location, the cache block containing that location is still in transfer, we call it a *partial miss* (or *partial hit*). The method proposed in this section targets the partial miss and improves the timing when the cache prefetch is fired.

Line Concept prefetch is not a standalone cache prefetching algorithm, but it has to work with other cache prefetching algorithms such as SIRPA. The idea of Line Concept is to capture the cache block size information in forming a cache prefetch request. Because a cache block is the smallest unit for cache memory transfer, it imposes a minimum time for a single transfer. The Line Concept approach lengthens the time available for cache prefetch to take place before the processor makes a memory request on the prefetched cache block.

The Line Concept also captures the directional information from the memory accessing patterns. The direction of access information is to let the cache prefetch to be fired at an earlier time and hence to make the probability of computation/memory transfer overlap to be higher.

The Line Concept works by observing the current memory address being accessed by the processor and the memory location of the cache prefetch being triggered. If the above two memory addresses fall onto the same cache block, and previous memory access pattern is in the forward direction, then the memory location following the current accessed one where it will map to a different cache block number will be prefetched into the cache memory. In essence, when the cache prefetch from a cache prefetching algorithm asks for the same cache block as the current processor requested location, the next cache block will be prefetched if Line Concept is used. On the other hand, if the memory access pattern is in the backward direction, the cache block previous to the current cache block will be prefetched.

4.2.2 Improvement Over "Pure" Algorithm

Improved Prefetching Time

In the SIRPA scheme, we found that the memory access pattern with a small but constant stride access on an array is like figure 4.4.

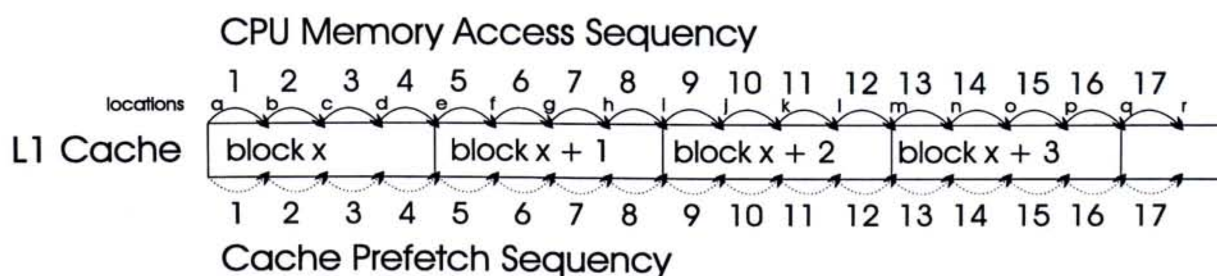


Figure 4.4: CPU Access Sequence on Cache Blocks

When CPU is accessing address at a, the prefetch calculated, assume previous

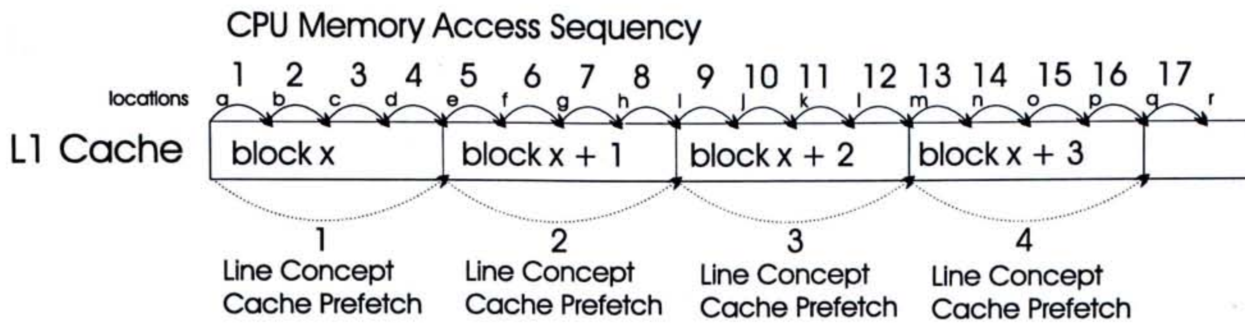


Figure 4.5: CPU Access Sequence with Line Concept Prefetch

accesses have already made the SIRPA to work, will be at address b, however, due to the organization of cache memory, address b is in the same cache block (block x) as the first CPU access. The similar situation also happens when CPU accesses address c. Only when the CPU accessed at address d, then the prefetch, which calls for address at e will actually bring in another cache block (block $x + 1$) from lower memory hierarchy. The cache system may not have sufficient time for the lower memory to send the content for cache block $x + 1$ before the CPU actually accesses it, then a (partial) cache miss occurs, and the CPU will be stalled until the cache miss is served. We attributed the problem to be lack of consideration to the size of a cache block which will affect the highly accurate prefetch.

One of the advantages of using Line Concept is to lengthen the time allowed for a cache prefetch to complete. In the above case, when the processor is accessing location a where that address is in cache block x, the cache block to be prefetched is cache block $x+1$, using Line Concept. Then the cache prefetch for block $x+1$ can be started as early as when the processor is accessing location a instead of when the processor is accessing location d. The effect of Line Concept is shown in the figure 4.5.

Higher Chance to Prefetch

As CPU demand–fetch and cache prefetch have different priorities in using the memory bus, there may be situations when the cache prefetch fired is accurate, but the memory bus is occupied by the processor for other memory transfers. The cache prefetch would not have a chance to use the memory bus. By using Line Concept in the prefetching scheme, the cache prefetch will be fired substantially earlier than without using Line Concept, the chance that a cache prefetch will get a chance to be served will be higher.

Line Concept together with the lengthened timing for cache prefetch improves the efficiency of the cache prefetching algorithm being used. We have performed experiments on SIRPA, SETCAM in [Ho95] and Chen's RPT in [Che93], with and without using Line Concept. The results are consistent that the version with Line Concept outperforms the one without. Detail results will be presented in the next chapter.

4.2.3 Architectural Model

Hardware Support

The hardware support required by the Line Concept scheme is minimal. We only have to add an extra unit called Line Concept Unit (LCU) in front of the cache prefetch unit, and the LCU has to access the CPU core for current effective address, and issue the cache prefetch to the memory request queue. Figure 4.6 is a block diagram on a processor using SIRPA and Line Concept.

Details of the Algorithm

In theory, the Line Concept scheme makes the prefetch for SIRPA or like prefetching schemes to look–ahead farer when the stride value is smaller than the cache

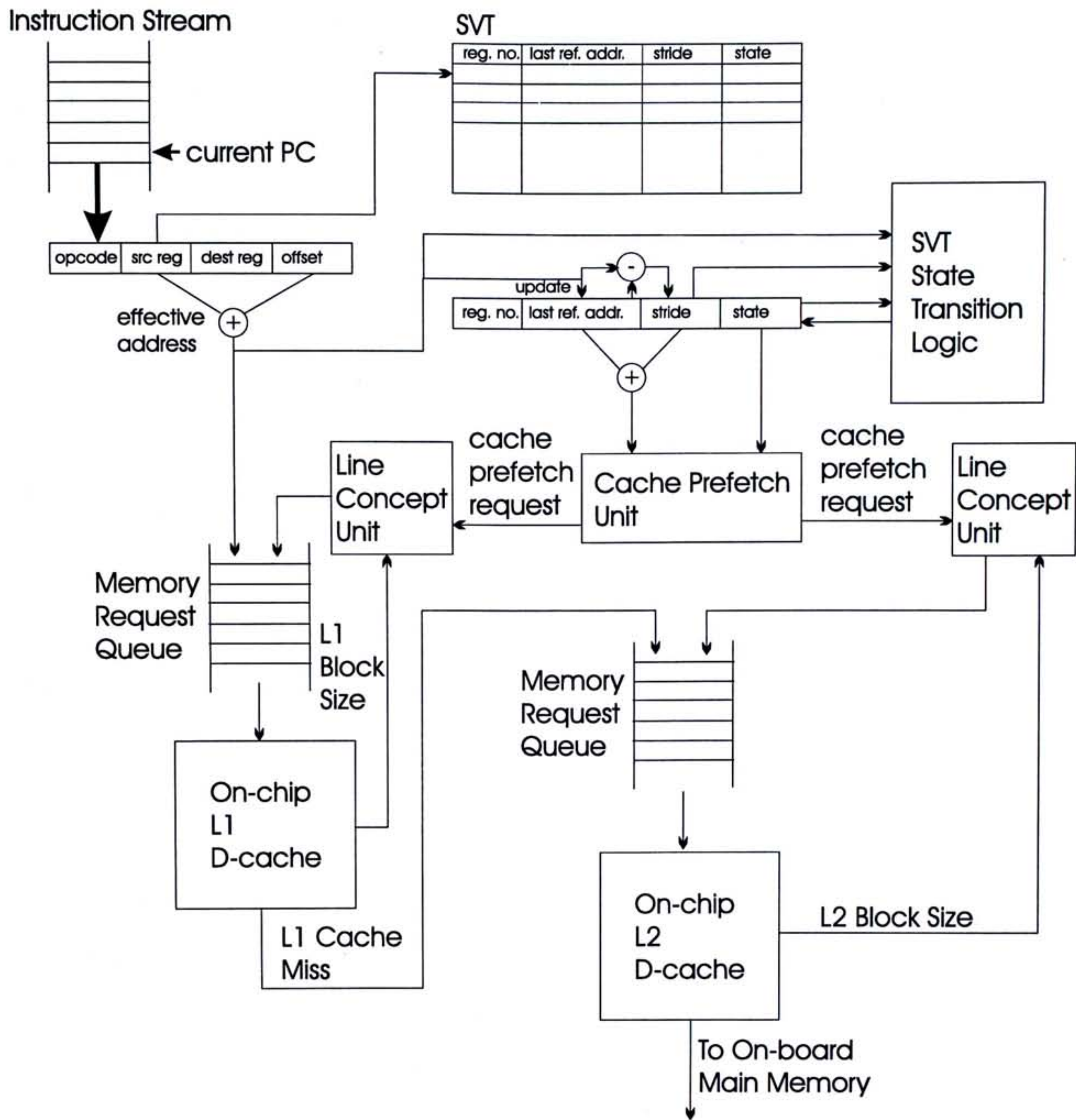


Figure 4.6: Processor using SIRPA and Line Concept scheme

block size. With conventional SIRPA, the cache prefetch address will be:

prefetch address = (last reference address + stride)

By using Line Concept, the cache prefetch address will now become:

prefetch address = (last reference address + $n *$ stride)

where n is the smallest integer which will make the cache block number containing the prefetch address to be different from the current access. With the example in figure 4.5, when CPU accesses at address a, the n chosen will be 4, as 4 is the smallest integer which will make the prefetch address falls on block

$x+1$ instead of block x which the current CPU reference falls onto.

In the new scenario, the cache will have a longer time–window to prefetch cache block $x+1$ from lower memory hierarchy, in fact the time will be around four times the original value. With small stride value, which several CPU references fall onto a single cache block, the new method can greatly improve the prefetch efficiency. The small stride value cases account for around 50 percent of all memory access patterns. For large stride values, the n chosen will be 1, which is the same as the prefetch address as calculated in the original method.

Due to different cache block size of L1 and L2 caches, which in normal case, block size in L2 will be about twice the block size in L1 cache, the same prefetch algorithm will trigger different prefetch addresses for L1 and L2 cache, because a different n will be chosen for L1 and L2 respectively. The net effect will be such that L1 cache will prefetch for cache block in the closer vicinity, which a high probability will be a hit in L2 cache. On the other hand, the L2 cache will prefetch for cache block farer away. The effect can be visualized in figure 4.7.

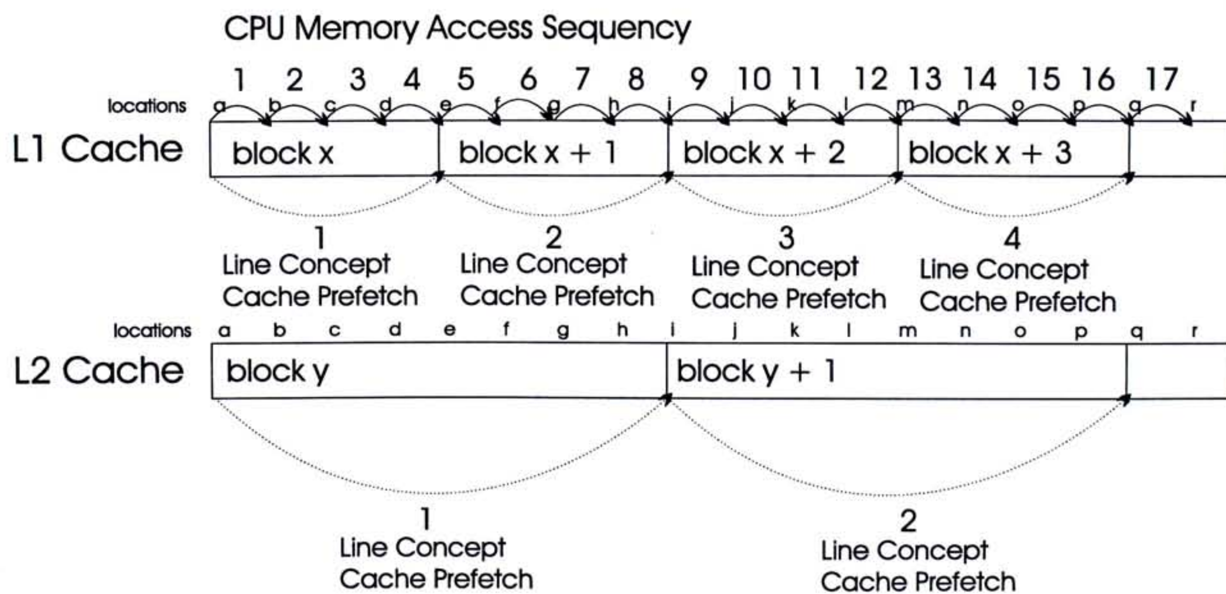


Figure 4.7: CPU Access Sequence on L1, L2 cache blocks w/Line Concept

When CPU is accessing address at a, the L1 cache will call for prefetch at address e, where L2 cache will call for prefetch at address i. Then, when the CPU

is progressively moved to work on address at e, the L1 cache will prefetch cache block $x + 2$, which now will be likely to be already in L2 cache corresponding to line $y + 1$.

Due to difference memory access time to fetch a cache block from memory hierarchy, usually, L2 cache will need a longer time to fill in a cache block than L1, by more aggressively to prefetch in L2, the longer memory access time can be compensated.

Line Concept tunes the cache block to be prefetched according to the block size as used by respective cache on the processor chip. It is a major support to the combined first and second level cache management. By using Line Concept, the cache prefetch patterns for first and second level cache can be different and take into account the cache block size effect.

With combined first and second level on-chip cache management, a single cache management unit is required to drive both levels of on-chip cache. The saving in transistor counts, and especially the saving in extra electrical loadings exerted on the CPU bus will be significant.

4.3 Combined L1-L2 Cache Management

Combined first and second level cache management would give a reduced hardware overhead on the implementation, and at the same time to make available more information for the efficent management of both cache memory levels.

4.3.1 Rationale

As can be seen in the previous chapters, on-chip second level cache should be the trend for next generation processors. In order to maximize the performance of precious on chip memory, intelligent control can be built to manage first and

second level caches. By exploring processor information, which is easily available on chip, the CPU access patterns can be learnt by the memory hierarchy to predict for future access. Due to locality property, most of the time a program will access data in the close vicinity to the current access, therefore, a small high speed memory level will speed up CPU access.

In order to build on-chip first and second level cache management logic, it will be beneficial to use a unified cache management unit to control both first and second level cache. On the one hand, the transistor count required can be reduced since the control mechanism for both levels is similar. On the other hand, by using a unified cache management unit, information can be shared between 2 level caches and more intelligence can be derived by considering cache configurations and other cache status for both levels.

4.3.2 Feasibility

In order to have a combined on-chip first and second level cache and management, we have to be convinced that firstly, it is feasible to build on-chip second level cache. Secondly, it is feasible to manage the first and second level cache through a single management unit. And it is beneficial to do so.

On Chip L2 Cache

In chapter two, the background study chapter, we have shown that the number of transistors in nowadays processor has been increased tremendously. The ultimate goal of microprocessor manufacturer is to produce fast chips. One of the methods is to use as much hardware as possible to implement software instructions, this trend is realized in RISC processors. The hardware to be implemented includes *barrel shifter*, which can make multiple of bit shifts in a cycle, *hardware multiplier*, which reduces the time required to perform multiplication through

conventional method, *hardware floating point unit*, which can calculate floating point number arithmetics without complex conversions, *branch prediction unit*, which can keep the instruction prefetcher to work through conditional branch instructions [YeP93].

The other way to improve speed on a microprocessor is through parallel execution units. In essence, the product will become a parallel computer on a chip, which in each processor cycle, the chip can process multiple number of instructions from a single program through different execution units. We have seen superscalar microprocessors in the niche market, and it will become popular in general purpose processors. A superscalar processor may contain two integer arithmetic units, one floating point unit, one unit dedicated for multimedia instruction, one load/store unit, one branch prediction unit, etc. When multiple execution units are built into a chip, the transistor counts consumed will be multiple times of a conventional microprocessor.

The third kind of way to use up the transistor count is to build a first level on-chip cache, as we explained in the previous sections, it is impractical to build a very large first level cache due to hardware constraints [BW88].

No matter how fast a microprocessor potentially can run, it still requires program instructions and data to process. If the instructions and data cannot be supplied on time, all the multiple execution units will just be idle. Therefore, in order to bridge the gap between microprocessor processing speed and on board memory speed, it is inevitable to make use of multiple layers of memory hierarchy with progressively faster speed towards the processor side. Majority of frequently used instructions and data will be kept in the fastest first level cache and it is backed by a relatively large and still speedy on-chip second level cache.

The benefit of on chip second level cache includes, it can be made of simpler and cheaper two-transistor static memory cells, rather than the more expensive

but faster memory cells as used in the first level cache. The purpose of the on chip second level cache is as a large repository for instructions and data to be used by the processor. As the second level cache is on chip, it can have a wide data bus connected with other components with the first level cache and CPU core. The on chip second level cache can be a mixed cache containing both instructions and data as contrast with the usual configuration of split instruction and data cache as implemented in the first level. The purpose of split cache in first level is to double the available bandwidth for memory transfer from the cache to the processor. As the second level cache is not as critical in speed, the mixed cache configuration will better utilize the available space in it. Different programs may have different mix of program size and data size, a mixed cache will adjust automatically for the portion of cache blocks allocated to instructions and data [STW92], however, in a split cache, the partition of instruction and data cache size has been fixed in the CPU design stage.

Second level cache can be implemented with set associative configuration. Compare with the first level cache, which speed is already the top priority concern, associative comparator is not easy to be implemented there. However, set associative cache can reduce the number of conflict miss readily [GHP93].

If second level cache is built on board, bandwidth verse number of pins running out of the chip will be a design issue. Moreover, the cache management unit as well as cache prefetcher cannot be shared between the first and second level cache. In reality, current CPU statuses, such as instruction opcode, addressing mode, register in use, instruction dependency, branch predictions, etc. cannot be available to the second level cache. Conventional on board second level cache can only run in the demand fetch mode, which limits the benefits that can be got from the cache memory.

Information Available to L1–L2

In a combined and on chip first and second level cache management unit, due to the short distance between CPU core and the unit, it can grasp current CPU statuses readily. These CPU statuses can aid prefetching algorithms to determine the processor memory access pattern.

In the *SIRPA* scheme, instruction addressing mode is used to find whether the current executing instruction is a load/store one using register indirect addressing mode. The current register in use is used in the SIRPA as index to it's SVT. Current effective address is required to update the SVT. When the SIRPA fires a cache prefetch, the priority of the request depends on whether the processor memory reference is a cache hit or a cache miss. All the above information is available on chip but hard to be obtained out of chip.

With *Line Concept* added to the SIRPA scheme, the configurations of the on chip first and second level cache has to be available to the LCU and cache prefetches will be modified according to the current processor memory request and the cache block size in the respective first and second level cache.

In *Chen's RPT* scheme, instruction addressing mode, current effective address, current program counter value and the instruction prefetch queue has to be accessible. The RPT, LA–PC and BPT in the scheme depend on the above information to make predictions to the future program counter, and in turn to fire cache prefetches in advance.

4.4 Combine SIRPA with Default Prefetch

Default Prefetch is a complement cache prefetch scheme to a highly selective prefetch algorithm, such as SIRPA. The Default Prefetch will issue cache prefetches even when the memory access patterns are not constant stride type.

4.4.1 Rationale

The SIRPA scheme as described in previous section gives highly accurate prefetch predictions for constant stride access patterns. However, for non-constant stride accesses, such as irregular stride access patterns as exhibited by data structure of linked list and hash tables, the SIRPA will not be able to trigger cache prefetch.

By the same principle of overlapping CPU computation and memory transfer from main memory to cache blocks, if we can derive a general purpose cache prefetch algorithm which can work on other kinds of stride access patterns, the memory latency as perceived by the processor can be further reduced.

The SIRPA scheme is designed with built-in confirmation feature to filter out inappropriate cache prefetch. The kind of cache prefetches given out by the scheme is highly accurate. In environments with irregular memory access patterns, the SIRPA scheme will not issue cache prefetch. This creates a void in those programs without a large amount of constant stride access patterns, the pure SIRPA scheme will degenerate to a demand–fetch one, and hence the memory latency will not be shortened.

Eventhough irregular stride access patterns cannot be tackled by the SIRPA scheme, they still show different degree of locality properties. The new scheme proposed in this section is to make use of a general purpose cache prefetch algorithm augment with the highly accurate SIRPA scheme, in order to handle all general purpose programs with moderate performance in terms of memory latency. The SIRPA scheme will be given priority when a constant stride access patterns are discovered. However, when there is no cache prefetch that can be emitted by the SIRPA scheme, the general purpose scheme will be used to make predictions on the future memory access.

The general purpose cache prefetch scheme selected should be a simple prefetching algorithm, which can handle all kinds of access pattern generally well. The

use of it is to backup the SIRPA for situations that SIRPA cannot make a prediction. The general purpose scheme should be easy to implement together with other cache prefetch schemes, as we will run two cache prefetching algorithms together in the cache management unit. The extra hardware loading on the CPU bus should not be great, as we do not intend to further lengthen the critical time path by adding extra bus drivers. Therefore, we expect the general purpose scheme should use relatively less CPU information as compared with the SIRPA scheme.

There were studies [Smi82] for general purpose prefetching algorithms. In them *One Block Look-ahead* (OBL) as introduced in the background study chapter is a common scheme. With OBL, there are three variants, they are *Prefetch on Hit*, *Prefetch on Miss* and *Always Prefetch*. Prefetch on hit and Always prefetch are not considered here as the number of cache prefetches generated will be very large as the hit ratio in the first and second level cache is expected to be over 80%. The memory bus bandwidth may not tolerate the loading exerted by the Prefetch on Hit and Always Prefetch schemes.

Prefetch on Miss (PFONMISS) is a simple cache prefetch algorithm which suits the requirement that it gives acceptable results and the number of prefetches generated is not too many. PFONMISS works on the principal that whenever there is a cache miss, the cache block following the miss will be prefetched.

When instructions and data exhibit spatial locality, the PFONMISS scheme can call in the neighborhood cache block which may be requested by the processor in the future. PFONMISS should give reasonable performance on random access, as long as the program still shows locality property.

4.4.2 Improvement Over "Pure" Algorithm

PFONMISS for Small Stride Access

In constant small stride application, where the stride values are smaller than the cache block size, PFONMISS gives similar performance as the SIRPA scheme plus Line Concept, because they will both call the next cache block into the cache when a given item is accessed. PFONMISS has the merit of simplicity.

With scalar/zero stride and irregular stride accesses, the PFONMISS scheme still works if the stride values are still smaller than the block size of the cache. Therefore, for PFONMISS to work, a larger cache block size is favorable.

On the down side, PFONMISS cannot be well adapted to constant large stride application and memory access patterns with large number of irregular accesses. Because PFONMISS does not distinguish the memory access patterns and hence there is no confirmation mechanism built in the scheme to filter out inappropriate cache prefetches. Cache pollution will be a problem in system using PFONMISS, as there may be cache blocks being replaced by the call–in blocks, where the prior one will be reused by the processor. Therefore, in a cache system using pure PFONMISS scheme, the performance of the system varies as the number of small stride access contained in the program. In general, if more than half of the memory access patterns are with small stride values, the PFONMISS scheme wins.

SIRPA for Constant Stride Access

SIRPA scheme works well in constant stride access patterns. The magnitude of stride values will not affect the accuracy of the SIRPA scheme, as they are recorded in the SVT and being used to calculate cache prefetch addresses. This property compliment with that of the PFONMISS scheme. PFONMISS scheme is sensitive to the magnitude of the stride value and works well with smaller ones.

On the other hand, SIRPA scheme is specifically designed to the constant stride access pattern with high accuracy, whereas the PFONMISS scheme is a more general purpose scheme, which works on a broad category of programs. The PFONMISS scheme will not be as accurate as the SIRPA scheme.

With SIRPA complemented with PFONMISS as default prefetch scheme, the cache prefetches generated will cover a broader range of applications well and at the same time to retain the high accuracy of cache prefetches in the SIRPA scheme.

4.4.3 Architectural Model

Hardware Support

The position of PFONMISS unit together with SIRPA supporting logic that can be used to control both the on chip first and second level cache is shown in figure 4.8.

The PFONMISS unit takes in the current cache access information including the cache block being accessed, and whether the access is a cache hit/cache miss, and fires a cache prefetch for the next cache block if the last access is a cache miss.

The cache prefetch request from the PFONMISS unit will be fired after the current cache miss is served. If at the same cycle, another cache prefetch from the SIRPA scheme is present, the PFONMISS cache prefetch request will be discarded.

Details of the Algorithm

The PFONMISS scheme works on the spatial locality property in general memory access patterns. When a cache block i mapped to real address a is accessed, if the memory access causes a cache miss, the real address $a + \delta$, where δ is the

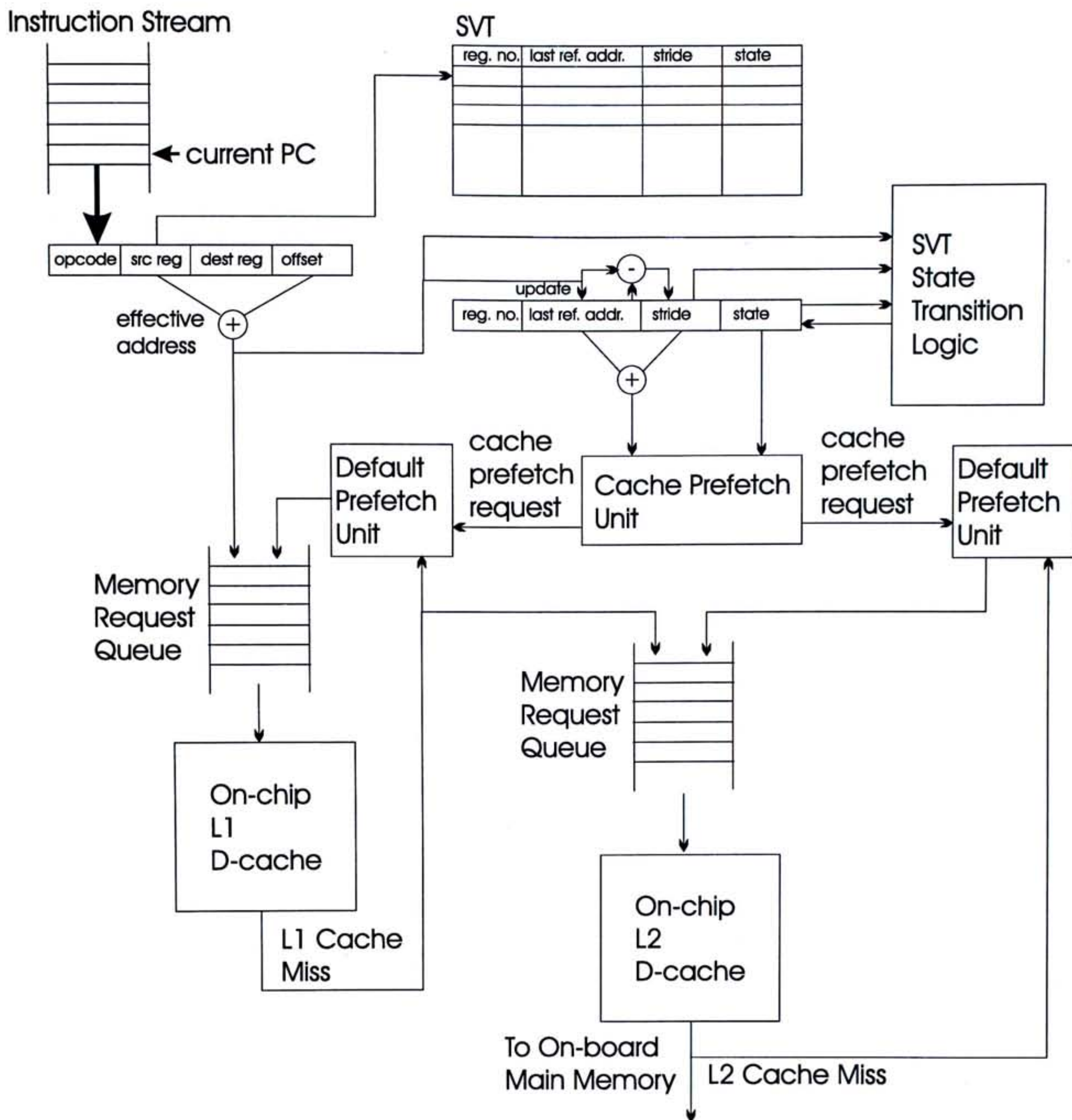


Figure 4.8: Combined Prefetch Scheme

smallest integer which can make the mapping of the resultant address to cache block $i + 1$ will be prefetched.

The PFONMISS scheme brings in the next cache block when a cache miss is happened, the goal is to prevent further cache miss if the processor moves forward in the memory locations to be accessed. If the rate of processor consumption of data is slower than the time taken to transfer a cache block of memory from lower layer to the cache, a cache miss can then be avoided.

The PFONMISS scheme assumes the direction of memory access is in the

forward direction, which is correct in the majority programs. When there is a program processing an array in the reverse order, the PFONMISS scheme will call in a lot of cache blocks which are already used in the last iteration. This will cause cache pollution. The cache pollution case will also happen when the next few items to be accessed are not in the current cache block nor in the next cache block, or the stride value is larger than the cache block size.

With the combined prefetching scheme using SIRPA and PFONMISS, we expect the cache system performance will not fluctuate greatly among different variety of programs. For scientific oriented programs using lots of arrays, the SIRPA scheme will give superior performance. For other application types, the PFONMISS algorithm will still give a general performance advantage over the demand fetch scheme.

The combined SIRPA scheme and the default prefetch of PFONMISS can also works with the Line Concept model. Only the SIRPA algorithm is modified by the Line Concept as the PFONMISS scheme will always choose the cache block different from the currently accessed one. In the result chapter, we will present benchmark tests by using pure PFONMISS scheme, SIRPA scheme with default prefetch, and SIRPA/Line Concept scheme with default prefetch.

Chapter 5

Results

In this chapter, we would state the benchmark programs that were used to test the proposed cache prefetch algorithms. The configurations of cache memory that we tested would be discussed in a section. Then, we would justify the validty of the results obtained.

We made two major kinds of comparison, one is by the Overall MCPI values obtained, and the other is by the Second Level Cache & Main Memory MCPI values. The Overall MCPI comparison reflects the performance of a computer system with respect to the memory system, whereas the L2 Cache & Main Memory MCPI sheds the effect of an efficient and effective lower memory hierarchy design.

In each type of comparison, we would present the results of the cache size effect, block size effect, and set associativity effect on both the hardware prefetch algorithms and software based prefetch algorithms.

5.1 Benchmarks Used

In order to test the proposed cache prefetch algorithms in the previous chapter, and to get comparisons with other cache prefetch algorithms in other publications,

extensive simulations had been done on the following benchmarks. The selected benchmark tests were come from the SPEC92 suite.

SPEC92 benchmark suite is a complex set of programs intended to test the performance of computer systems. The SPEC92 benchmark suite was designed by the Standard Performance Evaluation Corporation (SPEC). It is an independent body specialized in creating fair and objective benchmarks to measure computer performance.

SPEC provides continue effort to update her benchmark suite for the computer industry to make accurate and adequate performance measurements of current generation computer systems. The first version of SPEC benchmark suite was released in 1989 and it was referred to as SPEC89. The version of SPEC benchmark suite used in this thesis is the version released in 1992, and therefore it is called SPEC92. In SPEC92, there are a number of real application-based programs for measuring and comparing computer system performance. There are in general two groups of programs in SPEC92 that are tailored to test the performance of integer arithmetic and floating point arithmetic in a computer system, they are called SPEC CINT92 and SPEC CFP92 respectively.

SPEC92 is chosen because it is designed to be computationally intensive application, and does not assess the ability of a system under test to handle intensive I/O operations. The cache system design is best measured by such application as only when the processor is busily doing computation, that the memory latency from the cache system will make a performance difference. For I/O bound programs, usually the determination criterion for performance is the I/O bandwidth which is not the research interest in this thesis.

SPEC92 is widely adopted by computer manufacturers as well as academic field to be used for computer performance comparison. Although SPEC has released a new version of benchmark suite in 1995 (SPEC95), not many bench-

mark results are published on that version. SPEC95 is designed to cope with the ever growing trend in computation speed with current generation of processors. Therefore, to run SPEC95 is a much lengthier task than with SPEC92. As SPEC benchmark suite in general is to test real computer systems, if software simulator is used to run SPEC benchmarks, which the software simulator may run at a reduced speed of $1/1000th$ of the hardware speed, it will not be possible to complete the SPEC95 in reasonable time. The above are the reasons that we still chose SPEC92 benchmark suite for the work in this thesis.

5.1.1 SPEC92int and SPEC92fp

SPEC92 contains 14 floating point intensive benchmark programs in CFP92 and 6 integer intensive benchmark programs in CINT92. We have chosen randomly 3 integer programs from CINT92 and 5 floating programs to test the proposed cache prefetch algorithms by using our simulator. For each benchmark programs, we compiled the program using default compilation flags on an IBM RS/6000 workstation. The compilers used are the supplied C compiler and FORTRAN compiler by the vendor.

All the benchmark programs after compilation are then traced with an address trace generator program. The first 100 million instructions or the number of CPU instructions that required to make a complete run in each program are stored in respective trace files. One hundred million instructions are chosen because we have to make sufficient number of accesses to the on-chip second level cache to show the true performance of it. In the appendix, the average number of accesses to the first and second level cache is included for each configuration tested.

NASA7 is one of the floating point benchmark programs we used in the CFP92. It in itself consists of 7 different portions of code (called kernels) to perform different styles of floating point arithmetic operations. We found that

the first 100 million instructions cannot cover all the 7 kernels, so we take addition traces for each kernel. That set of traces include at least a complete execution of one iteration of the outer loop for each kernel. The number of instructions in some of the kernel traces exceeds three million.

The following is a brief description of the benchmark programs used. Source of information was obtained from SPEC92 document files.

COMPRESS

COMPRESS is an integer benchmark program in CINT92. The purpose of COMPRESS is to compress an input file using Lempel-Ziv coding. The program corresponds to a UNIX utility with the same name. The compress ratio by using the LZ coding for English based text file should be around 50–60%.

This benchmark has a very high code locality and the instruction cache hit rate is very high. For a 32k Bytes direct mapped instruction cache, the hit rate is around 99.5%. However, the COMPRESS benchmark exercises extensively the data cache. The static code size for COMPRESS is around 50k Bytes, and the data size is round 500k Bytes. COMPRESS was written in C language.

ESPRESSO

ESPRESSO is an integer benchmark program in CINT92. The purpose of ESPRESSO is to perform set operations such as union, intersect and difference. It tries to minimize boolean functions by producing a logically equivalent function to the input but with fewer terms. As arrays of unsigned integers are used to implement sets, this benchmark contains a large amount of constant stride memory accesses. The array sizes used in ESPRESSO are typically less than 200 members. ESPRESSO was written in C language.

NASA7

NASA7 is a floating point benchmark program in CFP92. NASA7 was written in FORTRAN language and contains 7 different sub-programs called kernels. NASA7 consumed around 8 megabytes of data space and around 100k Bytes of code.

The functions performed by each kernel in NASA7 are the following.

- btrix - Block tridiagonal matrix solution along one dimension of a four dimensional array
- cholsky - Cholesky decomposition in parallel on a set of input matrices
- emit - Creates new vortices according to certain boundary conditions
- c2fft2d - Complex radix 2 FFT on 2D array
- gmtry - Sets up arrays for a vortex method solution and performs Gaussian elimination on the resulting arrays
- mxm - Matrix multiply
- vpenta - Inverts 3 matrix pentadiagonals in a highly parallel fashion

SPICE

SPICE is a floating point benchmark in CFP92, however, SPICE performs both integer and floating point arithmetics. SPICE is a hardware circuit simulator to test circuits made by resistors, capacitors, inductors, mutual inductors, voltage and current sources and semiconductors, such as diodes, BJTs, JFETs and MOSFETs. The properties measured by SPICE are nonlinear dc, nonlinear transient and linear ac. SPICE is a moderate size program with code size of around 200k Bytes and data size of around 500k Bytes.

When SPICE is running, a program space of 300k Bytes and virtual data space of around eight megabytes are consumed. SPICE was written in FORTRAN language.

SU2COR

SU2COR is a floating point benchmark in CFP92. The majority of floating point arithmetic is carried out in double precision. SU2COR was written in FORTRAN language and being vectorizable. SU2COR computes the masses of elementary particles in quantum physics by using the framework of the Quark-Gluon theory.

SU2COR contains a large number of DO loops which should be good candidates for testing constant stride memory access. SU2COR consumes around 4 megabytes of data space, whereas the code size is around 3 megabytes.

TOMCATV

TOMCATV is a floating point benchmark in CFP92. The majority of floating point arithmetic is carried out in double precision. TOMCATV was written in FORTRAN language and being highly vectorizable. The code size of TOMCATV is around one megabytes and the array size in use in the program is around 3.7 megabytes. TOMCATV is a mesh generation program.

WAVE5

WAVE5 is a floating point benchmark in CFP92. The majority of floating point arithmetic is carried out in single precision. The purpose of WAVE5 is to study various plasma phenomena on a two-dimensional, relativistic, electro-magnetic particle-in-cell environment. WAVE5 works on a 500,000-particle problem on a 50,000 grid points. The code solves Maxwell's equations and particle equations of motion on a Cartesian mesh with field and particle boundary conditions.

It is estimated that 16 megabytes of working space is required by the WAVE5 program.

XLISP

XLISP is an integer benchmark program in CINT92. XLISP is a LISP interpreter written in C language. The purpose of the XLISP program in CINT92 is to solve the 9-queen problem.

5.2 Configurations Tested

5.2.1 Prefetch Algorithms

The following thirteen prefetch algorithms were tested. The tokens in parenthesis are used as headers in the results reported in Appendix.

- (base) Baseline - demand–fetch only
- (pomiss) PFONMISS - Prefetch On Miss
- (wash) Chen's RPT scheme
- (wash.li) Chen's RPT scheme with Line Concept
- (wash.fa) Chen's RPT scheme with Fully Associative RPT entries
- (indxrg) SIRPA - Source Index Register Prefetch Algorithm
- (indxrg.li) SIRPA with Line Concept
- (idx.pf) SIRPA with Default Prefetch
- (idx.pf.li) SIRPA with Line Concept and Default Prefetch
- (prefch) Software Prefetch Instruction

- (prefch.li) Software Prefetch Instruction with Line Concept
- (setcam) Software SETCAM Instruction
- (setcam.li) Software SETCAM Instruction with Line Concept

Due to the Software Based Prefetch Algorithms (prefch, prefch.li, setcam, setcam.li) depend on specialized instructions inserted into the program code, only the 7 kernels in the NASA7 benchmark were performed on these algorithms. The trace files for the NASA7 kernels used are the same as those used in [Ho95].

All the cache prefetch algorithms are applied on both first and second level on chip cache.

5.2.2 Cache Sizes

All the tests performed assume a split on chip first level cache. The first level instruction cache is of infinite size, as this study concentrates on data access patterns. The first level data cache is of 16k Bytes, direct mapped, block size is 16 Bytes, and using a write back write allocate policy. Non-blocking feature is enabled in the first level data cache.

Two sizes of on chip second level cache were tested. They were 64k Bytes and 128k Bytes. The sizes were chosen basing on several factors. Firstly, the size of on chip second level cache on current generation of processors, eg. DEC Alpha 21164 contains a 96k Bytes on chip second level cache. This shows the approximate number of transistor counts available for on chip second level cache. Secondly, the size of the on chip second level cache should be at least four times the on chip first level cache, because if the size difference is not large enough, the contents in both cache layers will be substantially similar, the effect of the second level cache will not be significant. Thirdly, the number of instructions which can make a reasonable number of access to the second level cache. As

the trace files used are with around 100 million instructions, they may not be sufficient to test a much larger second level cache. The time required to take a dependable performance figure was also considered.

Other than the above two sizes of on chip second level cache, another set of testes assuming an infinitely large second level cache was also performed. This set of data is to record the approximate overhead spent in the first level cache when the second level cache has a 100% hit rate. There would be no miss penalty on the second level cache. The MCPI attributed to second level cache is computed by the following formula.

MCPI due to L2 cache = MCPI due to memory − MCPI due to L1 cache

5.2.3 Cache Block Sizes

Two cache block sizes on the on chip second level cache were chosen to be tested. They were 32 bytes and 64 bytes cache blocks. The chosen values reflects the general design principal to have larger block size in the second level cache than on the first level cache. But the cache block size cannot be substantially larger than the data bus width, because a cache block is the smallest transfer unit from the main memory.

5.2.4 Cache Set Associativities

Cache set associativities of 1, 2 and 4 were tested on the on chip second level cache. These are the common values used in implementing second level cache.

5.2.5 Bus Width, Speed and Other Parameters

The data bus width between the on chip first level cache and the second level cache is assumed to be 128 bits wide. This figure is four times the register width

of today's 32-bit processor and reflects the internal data path width of current top performing processors.

Speed of the first level cache is assumed to match the processing speed of the CPU. That is, the instruction and data will be available to the processor on the same cycle if the memory request is a cache hit in first level cache when the processor is running that instruction. The penalty for a first level cache miss but a second level cache hit is 6 CPU cycles for the first word of data and 1 CPU cycle thereafter. The figures chosen approximate to a real computer system with a CPU running at 200MHz, and the second level cache is made up of memory cells with accessing speed of 30ns.

When there is a second level cache miss and the data have to be transferred from the main memory, the timing requirement is assumed to be 10 CPU cycles to access the first word in the data and 6 CPU cycles thereafter. The figures correspond to a real computer system using 50ns dynamic memory chips to implement the main memory.

The on chip second level cache supports non-blocking feature as the same as on chip first level cache. The write policy of the on chip second level cache is write back write allocate and a write buffer of 8 entries are assumed to be available on the second level cache.

The processor in the tests is assumed to be a scalar processor which can process at most one instruction in a CPU cycle. If all the instruction code and data are available to the CPU, the instruction is assumed to be finished at the next CPU cycle. This assumption does not correspond to many real life processors. However, as this thesis is targeted for the performance of on chip cache design, the computation unit in the CPU is not a concern. With the assumption of a very high performance computation unit, the consumption of instruction and data exerts a high demand on the cache system. If a cache design with prefetch

algorithm performs well in the test, it should perform even better in a CPU with slower computation power, as there will be more time for the computation and memory transfer overlap.

For Chen's RPT scheme used in this study, there were 128 entries in the RPT, and use direct mapped scheme in the RPT, with the exception of the fully associative test on Chen's RPT, which in that case, the RPT had fully associative RPT entries, using LRU replacement algorithm in the RPT. The LA-PC and BPT were set to run ahead of the CPU program counter for one loop iteration.

For the SIRPA scheme used in this study, there were 32 entries in the SVT, and the number corresponds to the number of integer registers in the target CPU. Only integer registers in the target CPU can be used in register indirect addressing mode.

5.3 Validity of Results

With thirteen cache prefetch algorithms, fifteen benchmark programs, and combinations of cache sizes, cache block sizes and set associativities, a total of 167 simulations were run. The total running time for the MMHS to simulate all these testes on a 50MHz SUN SuperSparc processor is more than 1.5 CPU years. We believe that we had made an extensive and fair study on the performance of multi-layer hierarchy of cache system.

5.3.1 Total Instructions and Cycles

In each trace file, there are a number of CPU instructions captured in it. The smallest trace file used in this study is the mxm kernel in NASA7, which contains 10.8 million instructions. The largest trace file used in this study is the gmtry kernel in NASA7, which contains 344 million instructions. The total number of

instructions in all the benchmark programs exceeds 1.3 billion.

As the above 1.3 billion instructions were tested with different configurations, a total of 14.2 billion instructions had been run through the MMHS.

The total number of CPU cycles run in the simulations was around 36 billion.

Borg, Kessler, etc. in [BKW90] supported the notion to use a very long address trace to measure the real performance of memory system.

5.3.2 Total Reference to Caches

When a computer system with cache memory is freshly started, the cache memory contains no valid cache block, therefore, in a period, almost all memory accesses from the processor cause cache miss. We call the phenomenon, *cache cold start.* The performance of the cache memory will be stabilized when the cache cold start period is over, and the usually accessed data are already in the cache. In order to study the real behavior of a cache prefetching algorithm, we would measure the performance data when the cache memory is warmed up. To minimize the effect of cache cold start, we have to make sure each cache block will have chance to be exercised for sufficient number of memory transfers. The following discusses the number of memory requests presented to different level of cache memory in the simulations performed.

First Level Cache

The number of memory requests presented to the first level cache is independent of the cache configuration. Because all memory requests on first level cache is from the processor running a particular program, which the processor configuration is an invariant in this study.

The number of memory requests per simulation on the first level cache ranges from 4 million in the vpenta kernel of NASA7 to 207 million in the gmtry kernel

of NASA7. The average number of memory requests per simulation on first level cache is 39 million.

Compare with the number of cache blocks in the first level cache, which in all our simulations are fixed at 1024, the average number of references to each cache block is 38 thousand times. We would not doubt the first level cache in each of the simulations had been fully warmed up and the performance measured should well be stabilized to reflect the real behavior of the particular configuration for the benchmark programs.

Second Level Cache

The number of memory requests presented to the second level cache depends on the cache configurations, the efficiency of the first level cache and the program behavior. The number of cache prefetches on the first level cache will also affect the number of memory requests on the second level cache, as a L1 cache prefetch is considered as a normal memory request in the L2 cache. If the first level cache has a high hit rate, the number of memory requests for the second level cache will be lower, and vice versa.

The number of memory requests per simulation on the second level cache ranges from 755 thousand times in mxm kernel of NASA7 to 117 million times in gmtry kernel of NASA7. The maximum number of memory requests on the second level cache for a particular benchmark program is usually at least 2 times the minimum number. It showed that the first level cache configurations and the cache prefetch algorithms used do make a big difference in reducing the memory requests on the second level cache.

The second level cache size, block size, set associativity and prefetch algorithm used change in each configuration. The average number of access to the second level cache blocks varies from 200 times to 451 thousand times. We believe that

all the performed simulations did warm up the second level cache in test and the performance figures obtained should reflect a stabilized cache memory.

5.4 Overall MCPI Comparison

Overall Memory Cycles Per Instruction (MCPI) is the portion of time an CPU instruction had spent in waiting the memory system for a data. The computation time for that instruction has been removed, therefore, the figure should be a system independent value.

For the thirteen cache prefetch algorithms that we proposed in previous chapters, we had grouped them into a few categories:

1. Base, without cache prefetch
2. PFONMISS
3. Chen's RPT family
4. SIRPA family
5. Software Based Cache Prefetch Algorithms

We compared the results on overall MCPI for Baseline case (in category 1), PFONMISS (in category 2), Chen's RPT scheme (in category 3), and SIRPA scheme with Line Concept (in category 4) in the following section. The Cache Size effect, Cache Block Size effect and Cache Set Associativity effect would be analyzed. Then, the effect of Line Concept and Default Prefetch schemes would be studied. Finally, the Hardware Prefetch Algorithms would be compared with Software based Prefetch Algorithms.

5.4.1 Cache Size Effect

For the two cache size configurations used in second level cache, which are 64k Bytes and 128k Bytes, we plotted a chart for each benchmark programs. Because we run twelve combinations of cache configuration for each benchmark programs, other than the variation in cache size, the other two parameters are cache block size and set associativity, in all the charts shown, we kept these two parameters constant. The selected values are 32 Bytes cache blocks and 4-way set associative configuration. These values gave the best performance in all configurations tested in general.

The Overall MCPI values are plotted against the L2 cache sizes for each benchmark programs in figures 5.1 – 5.3. The lower the MCPI implies the better performance of a configuration. In figures 5.4 – 5.6, the percentage reduction in Overall MCPI using the Baseline case as 100shown, the higher the percentage reduction, the better a cache prefetch algorithm to shorten memory lattency.

From the Overall MCPI charts, we observed that almost all cache prefetch algorithms work better in 128k Bytes second level cache. However, the improvement in memory latency varies from different cache prefetch algorithms. The bigger improvements in overall MCPI occur in compress, and spice benchmark programs. This is not surprise as both of them work with large amount of data, and the working set size was expected to be large. For other benchmarks, the difference in overall MCPI between 64k Bytes and 128k Bytes second level cache was minimal.

We can find a general trend from the charts that, there are consistent improvements in MCPI by using SIRPA with Line Concept or Chen's RPT scheme for various benchmarks. The performance of SIRPA scheme and the Chen's RPT scheme is quite similar. However, for the PFONMISS scheme, the performance fluctuates greatly.

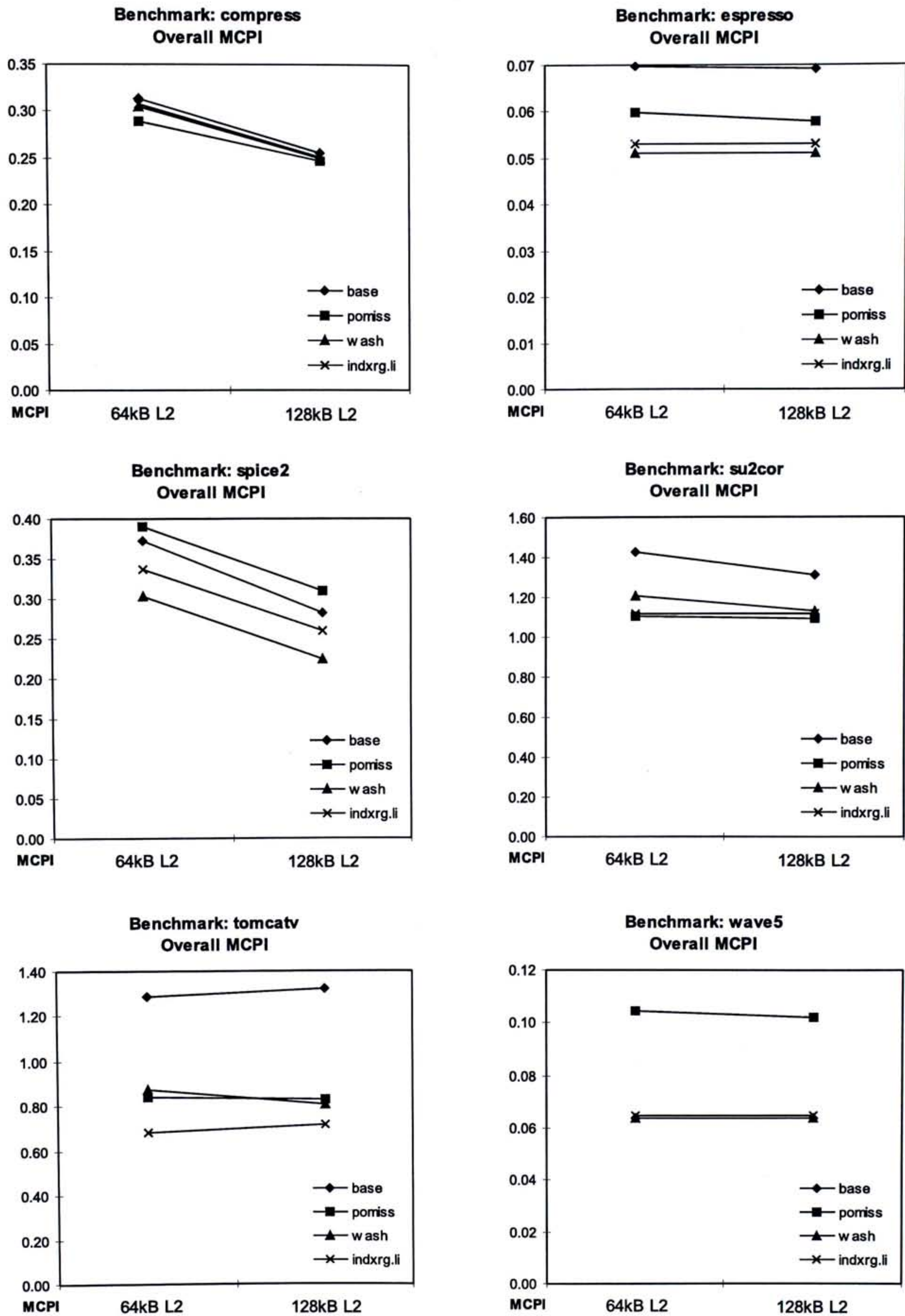
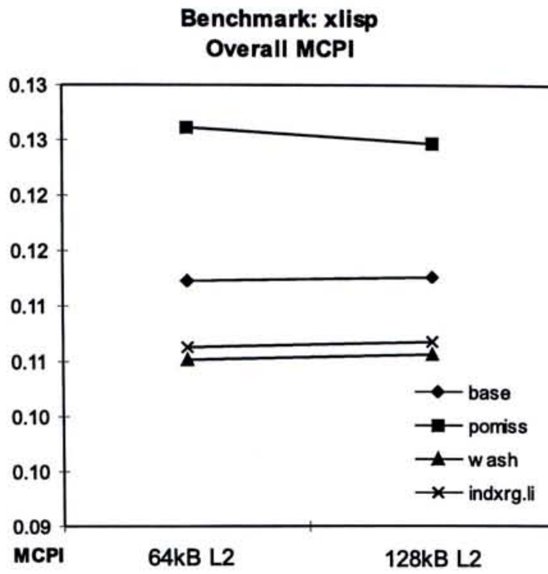
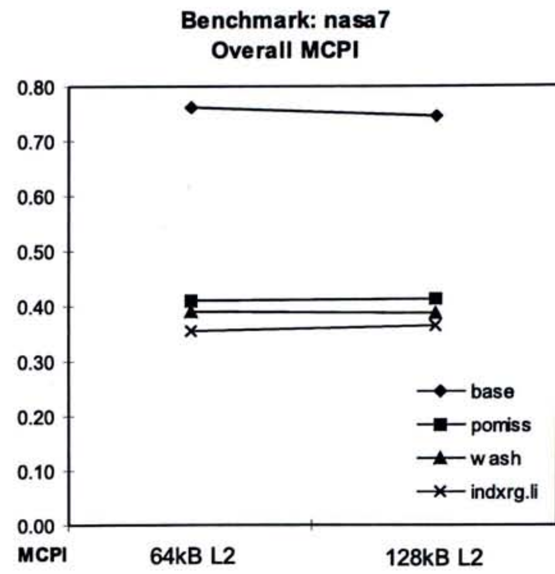
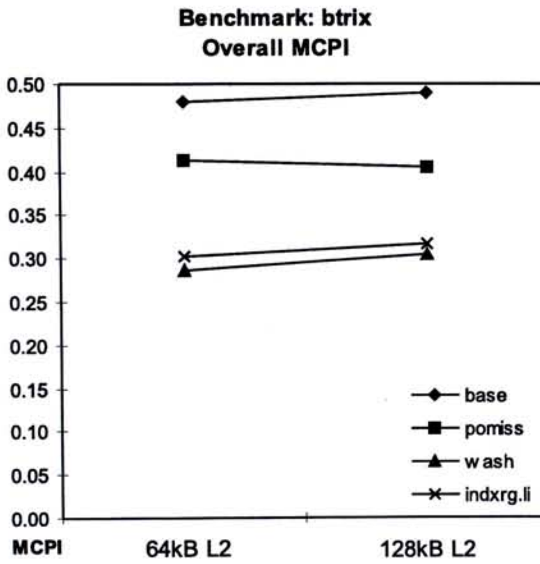
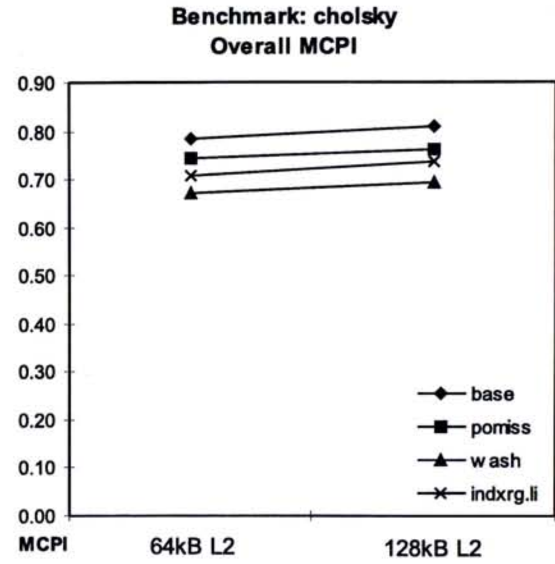


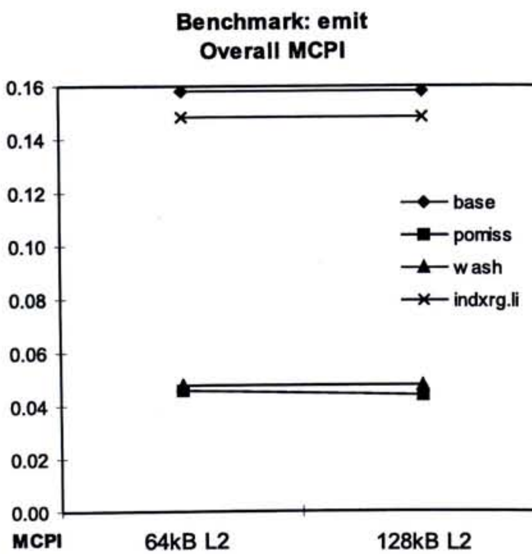
Figure 5.1: Overall MCPI comparison by cache size











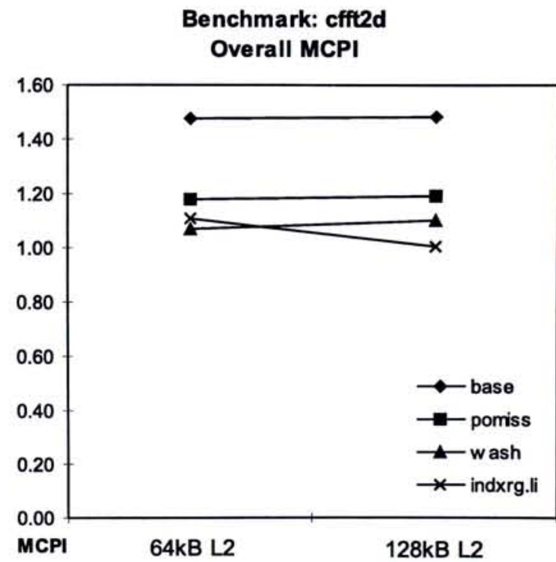
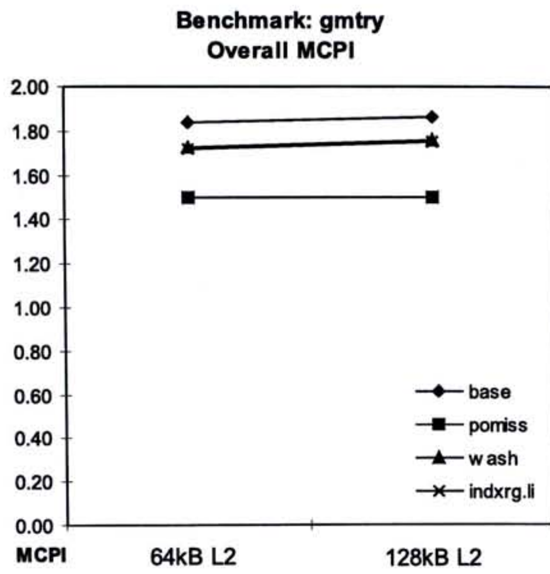
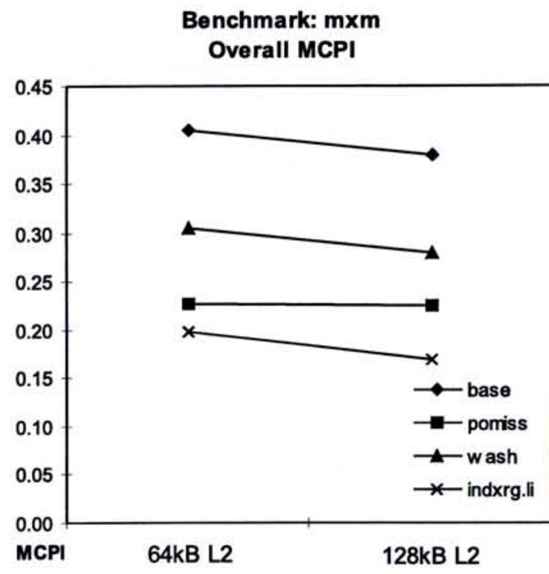


Figure 5.2: Overall MCPI comparison by cache size (cont.)





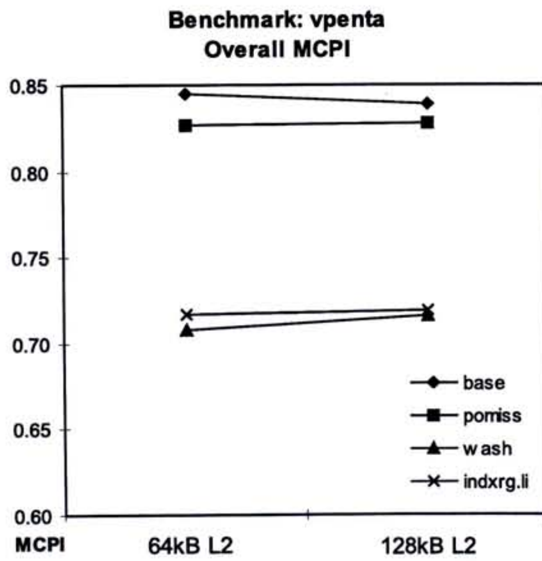


Figure 5.3: Overall MCPI comparison by cache size (cont.)

The PFONMISS scheme improves the overall MCPI in compress, espresso, tomcatv, su2cor, nasa7, btrix, cholsky, cfft2d, emit, gmtry, mxm and vpenta benchmarks. But it decreases the memory system performance in spice2, wave5, and xlisp benchmarks. The fluctuation is due to the imprecise mechanism used in PFONMISS, which for large stride access, the scheme will cause cache pollution.

Even in benchmarks where PFONMISS shows positive performance, the improvement is consistently not as good as those using SIRPA scheme or Chen's RPT scheme. The reason may be similar to the above that PFONMISS nonetheless will introduce cache pollution.

There are three cases that increase the second level cache size did result in a increase in overall MCPI, they are tomcatv, btrix kernel in NASA7, and cholsky. We attribute the anomaly to the mapping between the cache blocks in the first level cache to the cache blocks in the second level cache. The anomaly is not significant, as for all the three benchmarks, the increase in overall MCPI is very minimal.

5.4.2 Cache Block Size Effect

We had plotted the two different cache block sizes of 32 Bytes block and 64 Bytes block used in the second level cache against overall MCPI for all benchmark programs in figures 5.7 – 5.9. The reduction in Overall MCPI by using Baseline case as 100figures 5.10 – 5.12.

There is a consistent observation that the increase in cache block size from 32 Bytes to 64 Bytes gives a lengthier overall MCPI, with the exception in the mxm kernel in NASA7 benchmark.

The poor performance of a large cache block size may due to three reasons.

1. The transfer time per cache block is longer for a larger cache block size. In the simulations performed, from main memory to the second level cache,

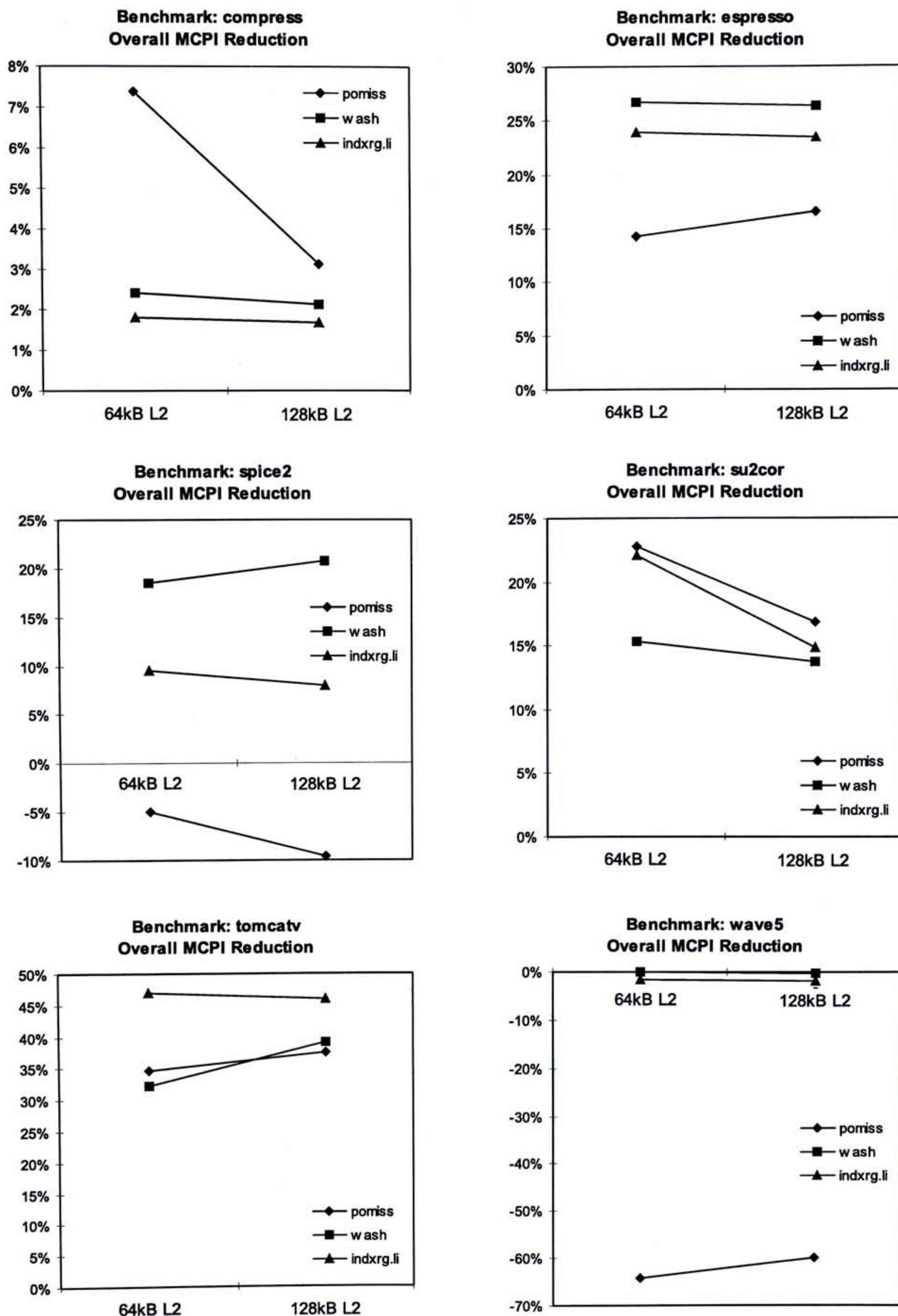
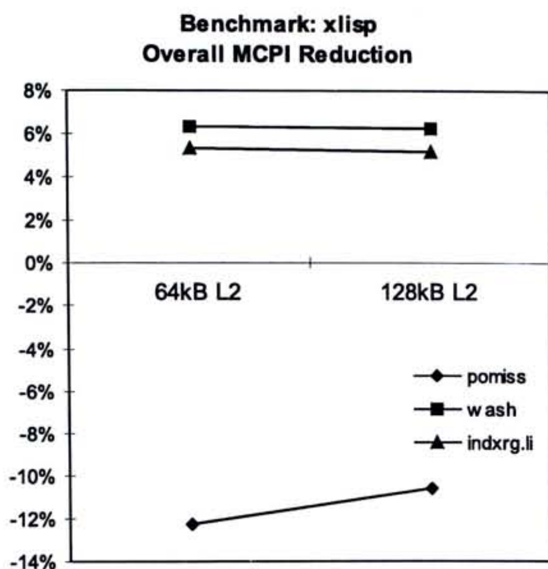
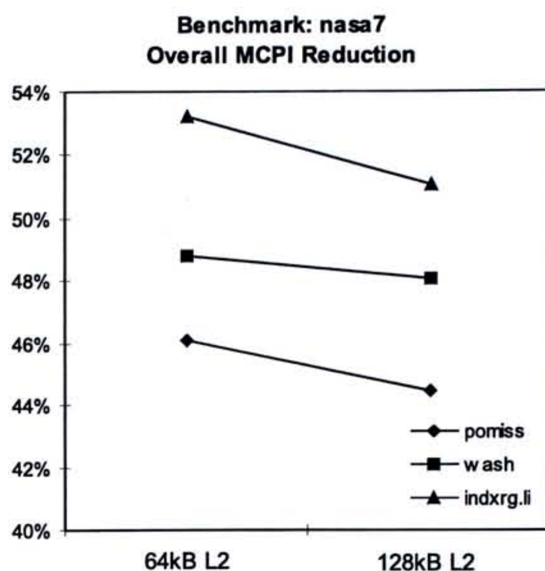
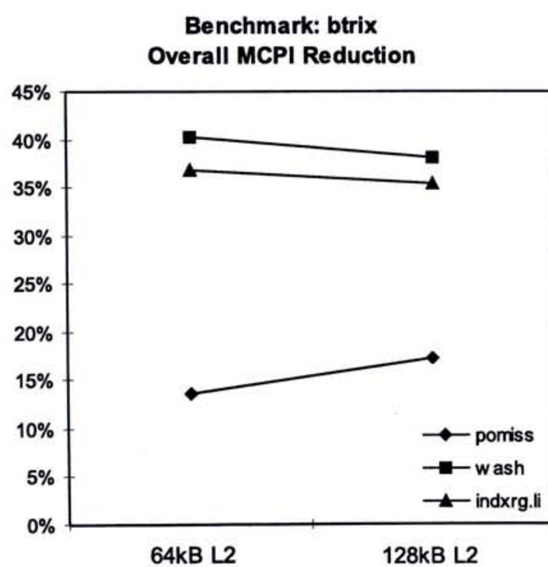
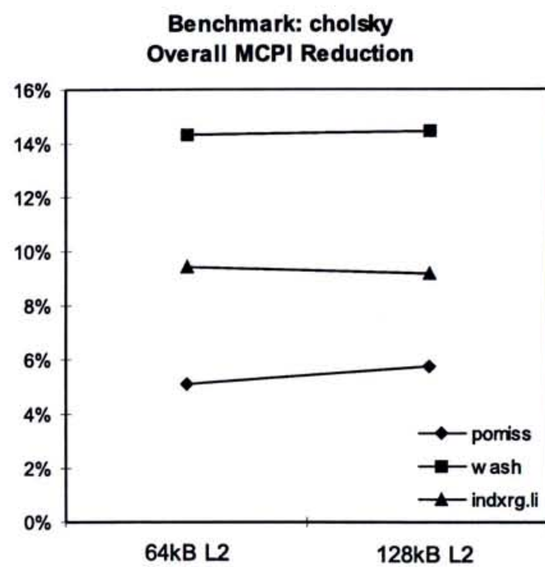


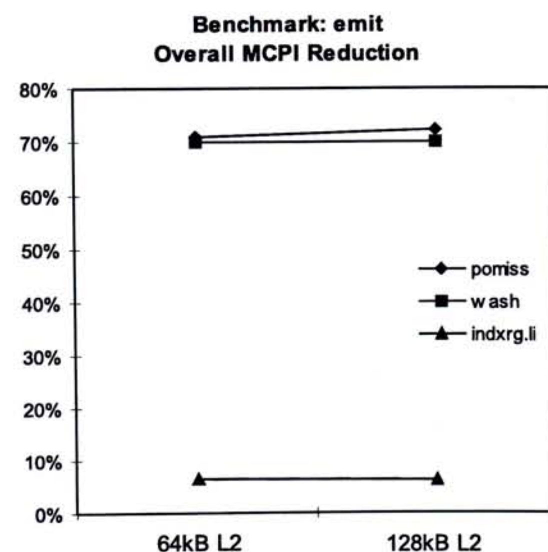
Figure 5.4: Overall MCPI Reduction comparison by cache size











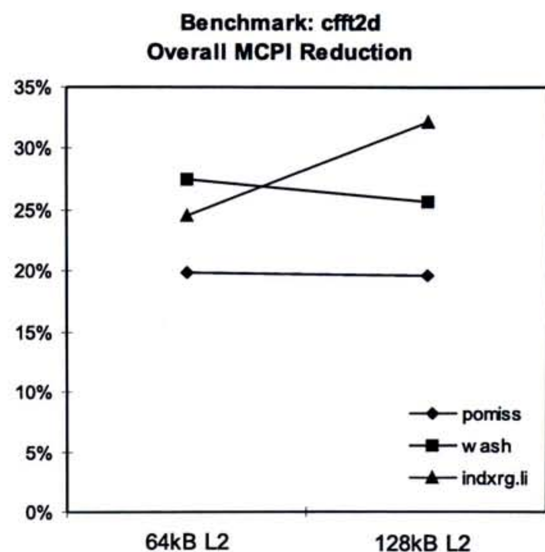
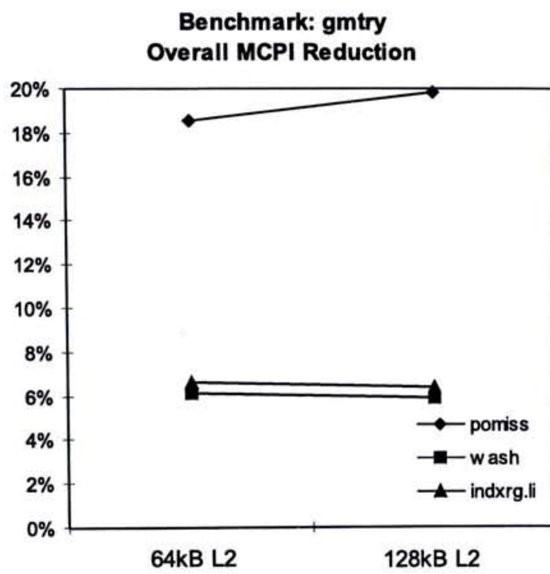
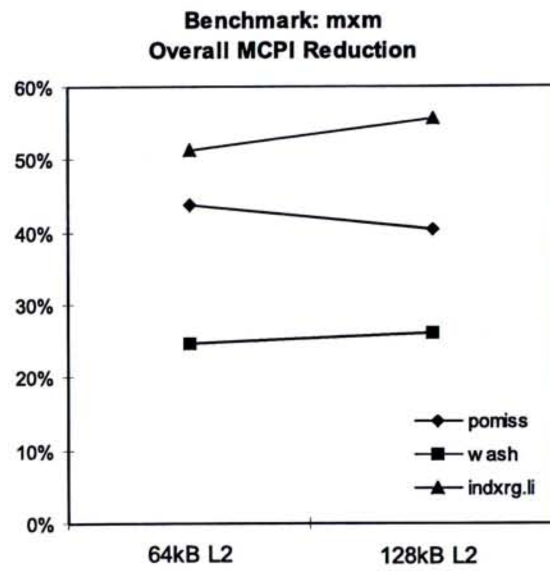


Figure 5.5: Overall MCPI Reduction comparison by cache size (cont.)





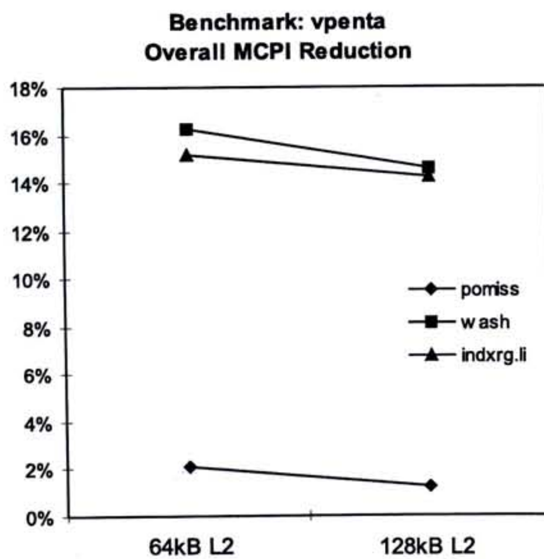
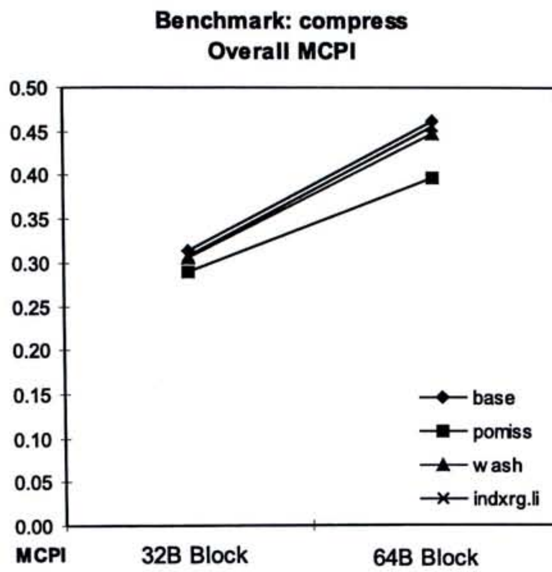
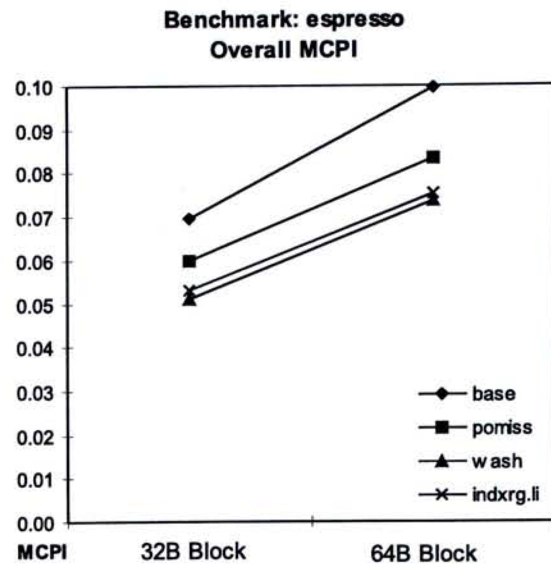
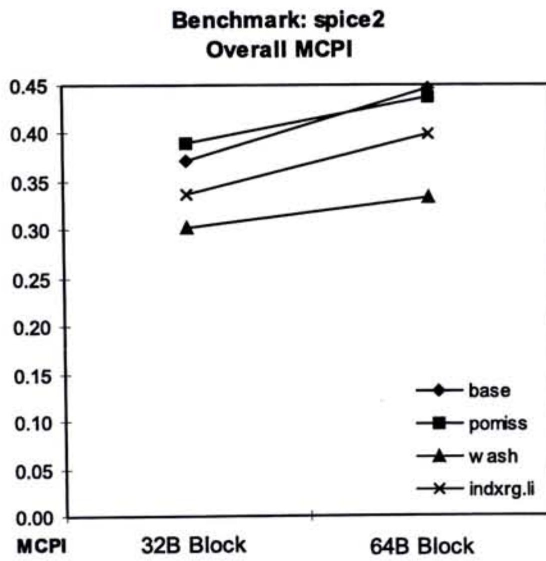
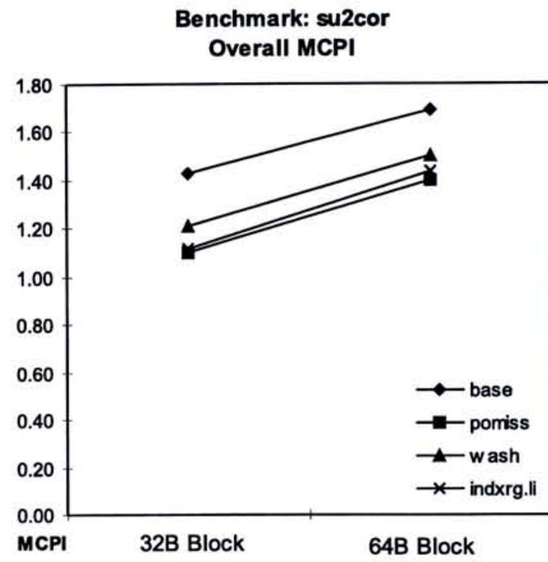


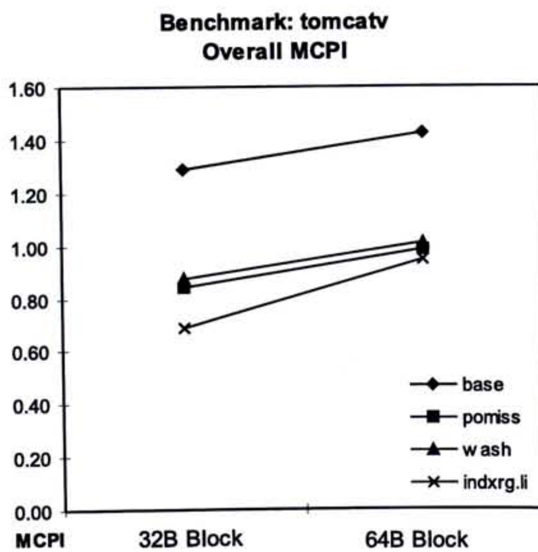
Figure 5.6: Overall MCPI Reduction comparison by cache size (cont.)











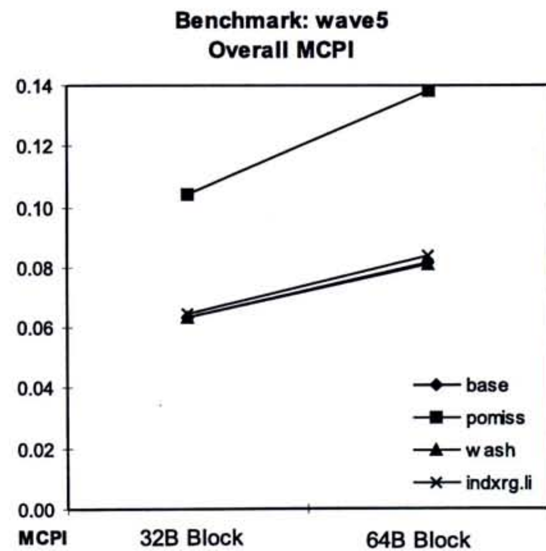
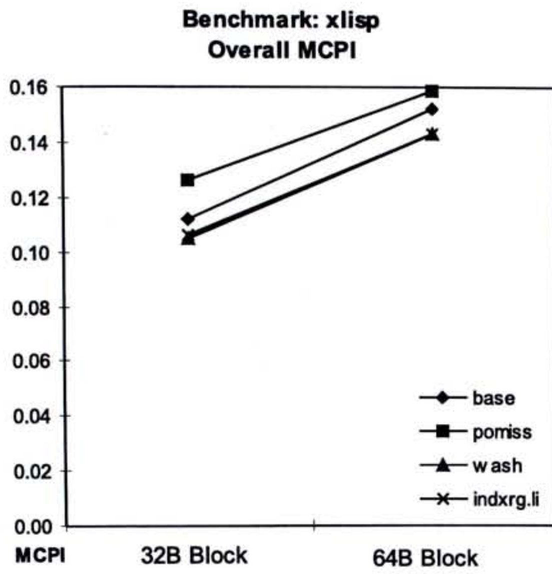
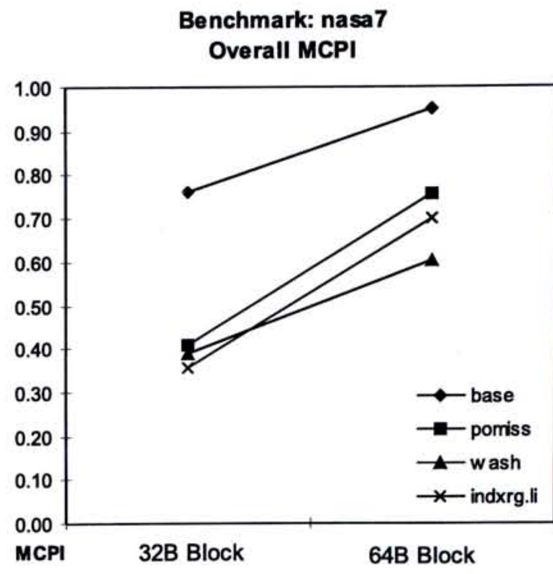
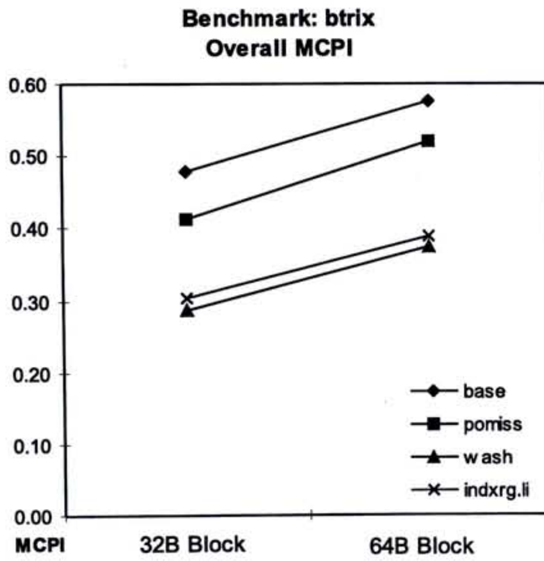
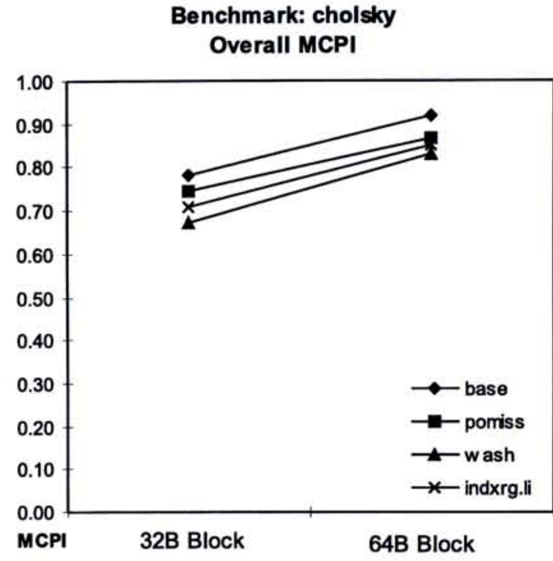


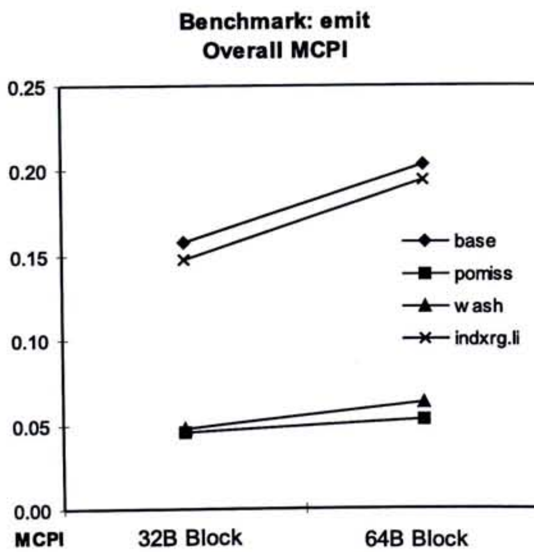
Figure 5.7: Overall MCPI comparison by block size











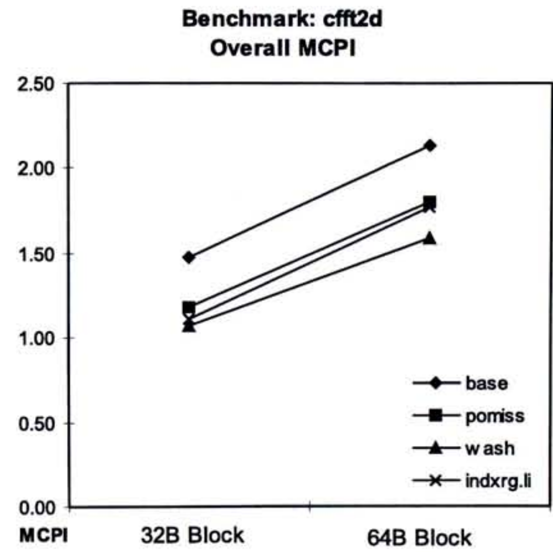
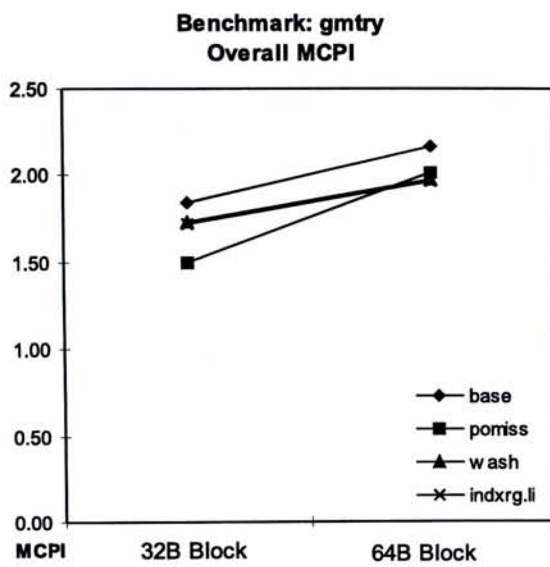
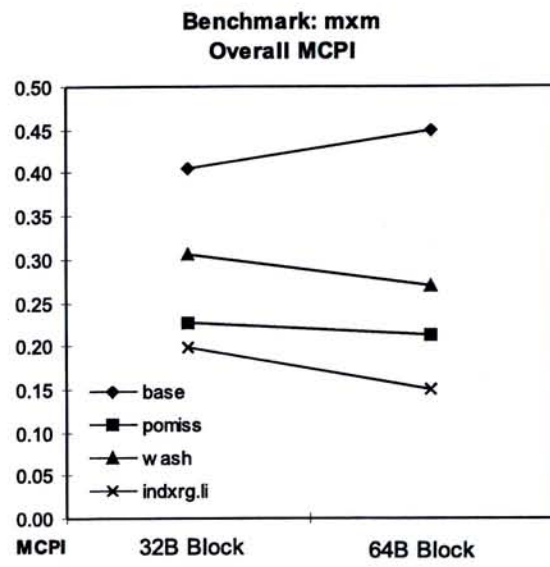


Figure 5.8: Overall MCPI comparison by block size (cont.)





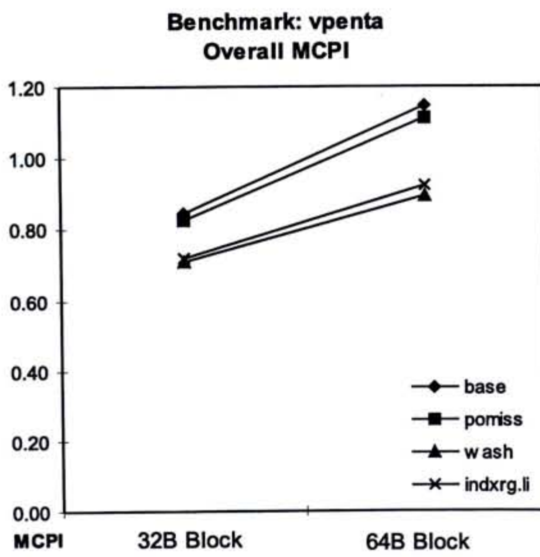
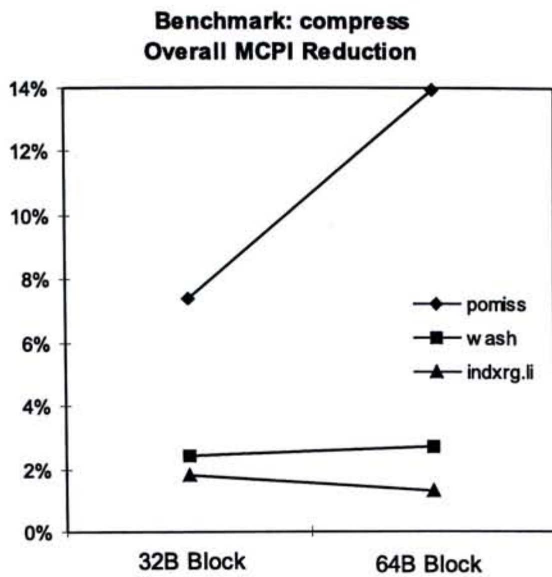
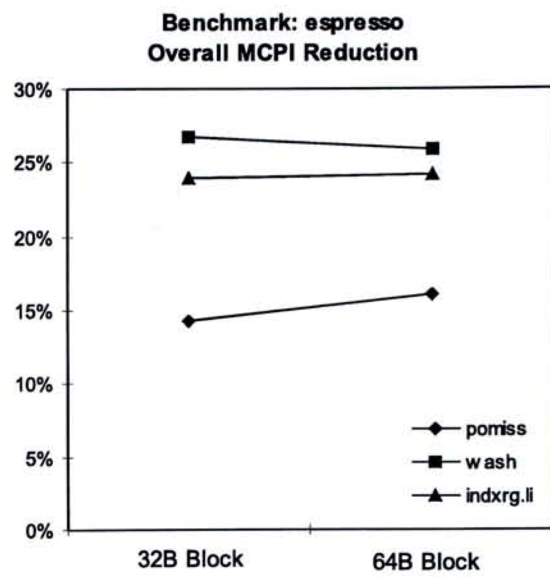
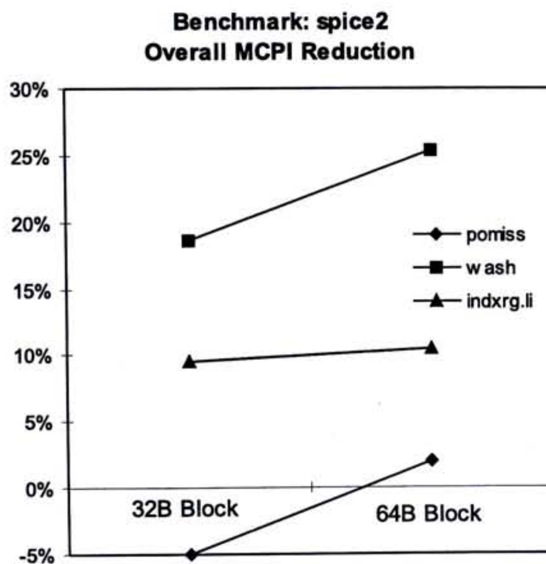
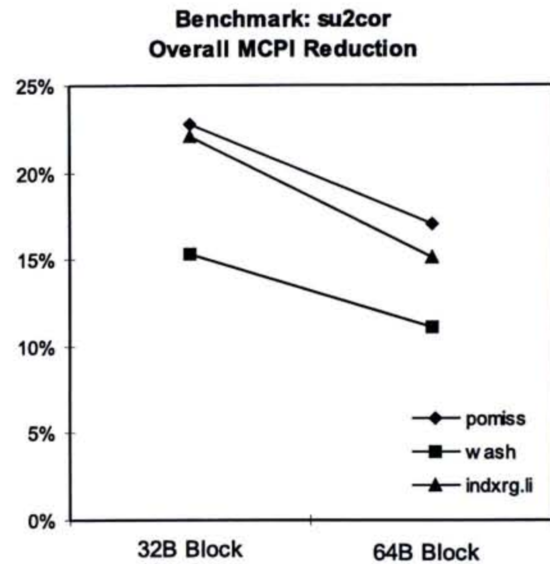


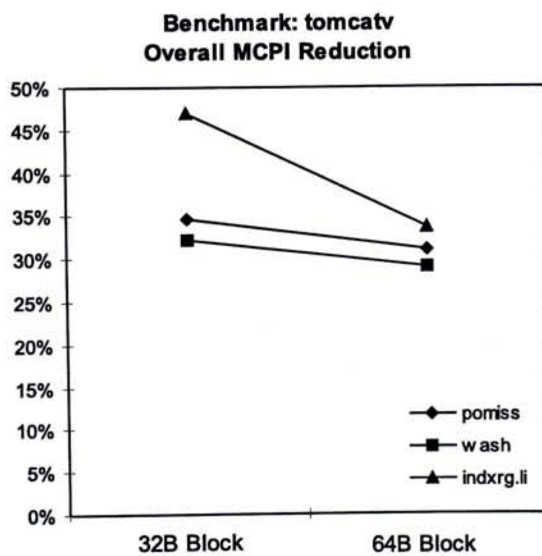
Figure 5.9: Overall MCPI comparison by block size (cont.)











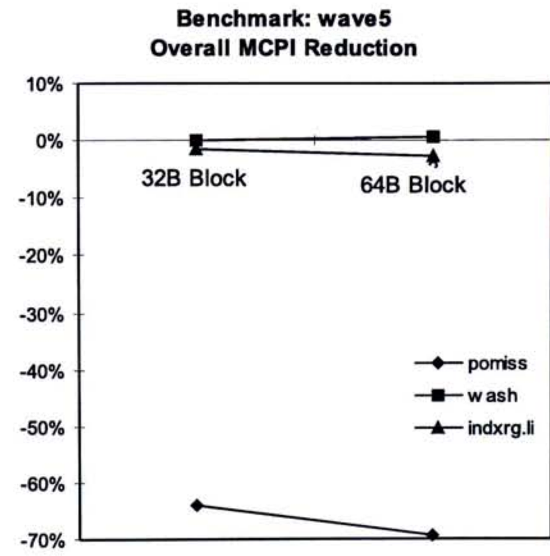
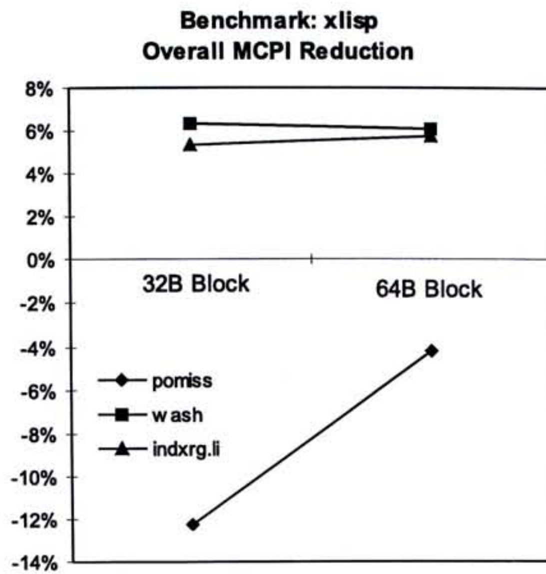
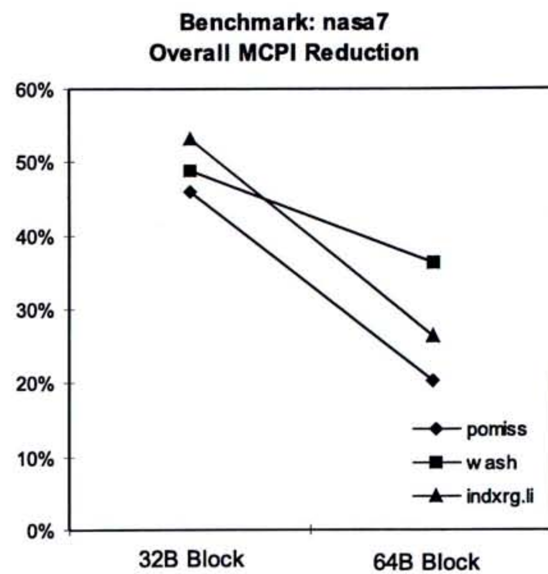
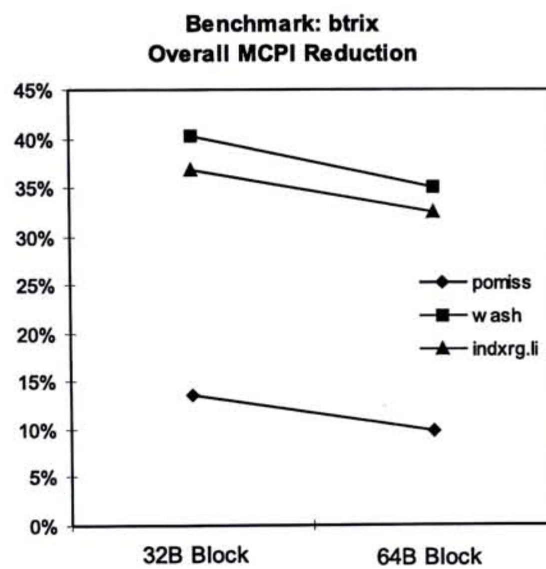
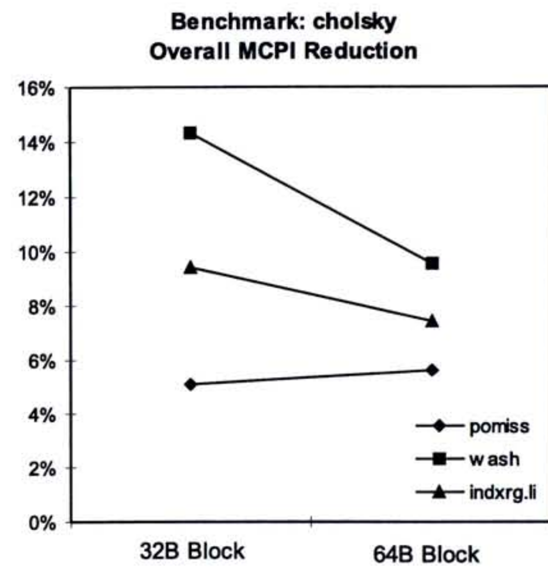


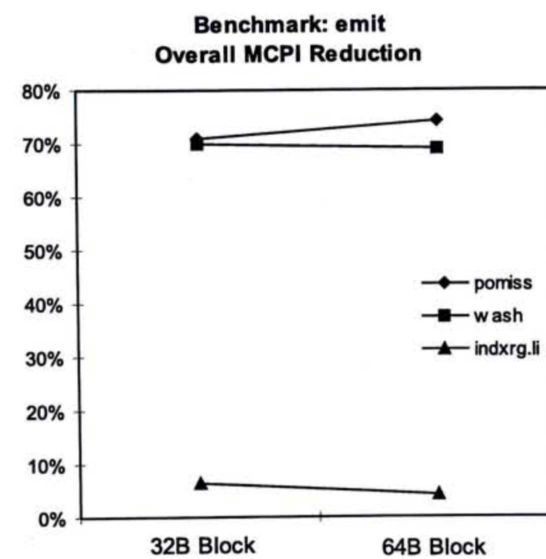
Figure 5.10: Overall MCPI Reduction comparison by block size











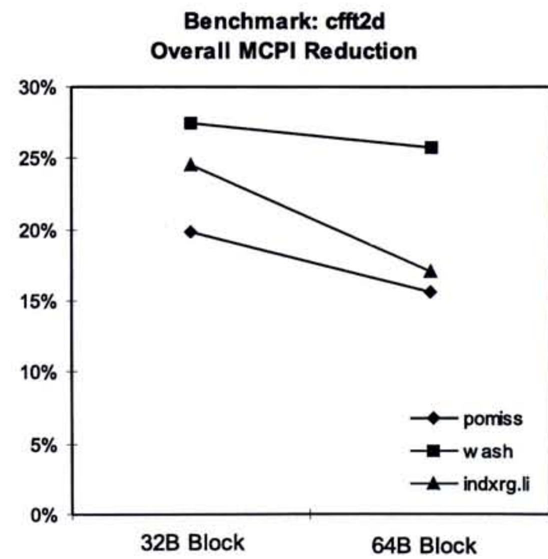
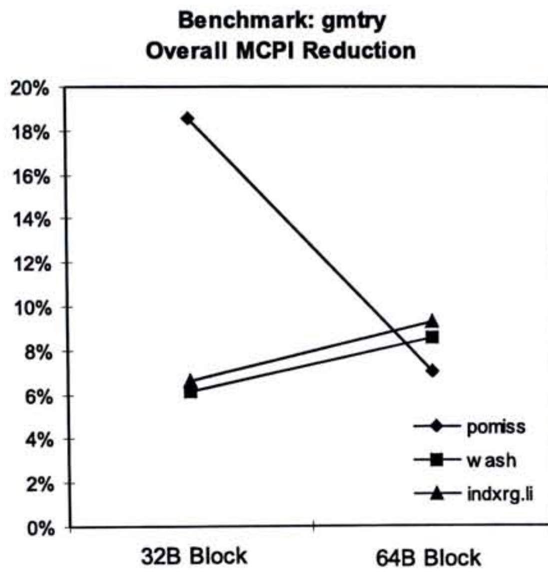
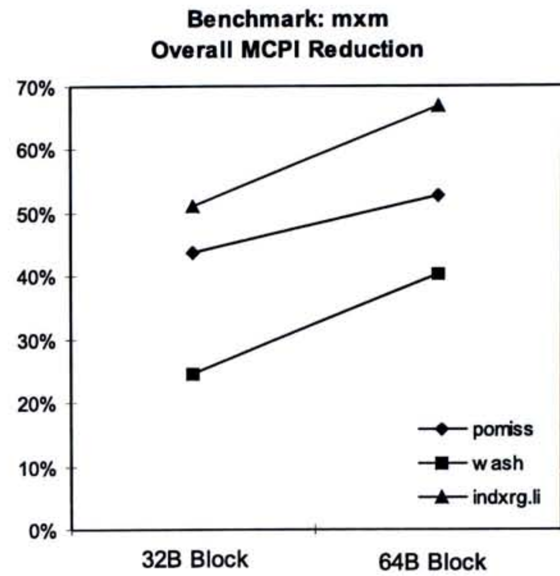


Figure 5.11: Overall MCPI Reduction comparison by block size (cont.)





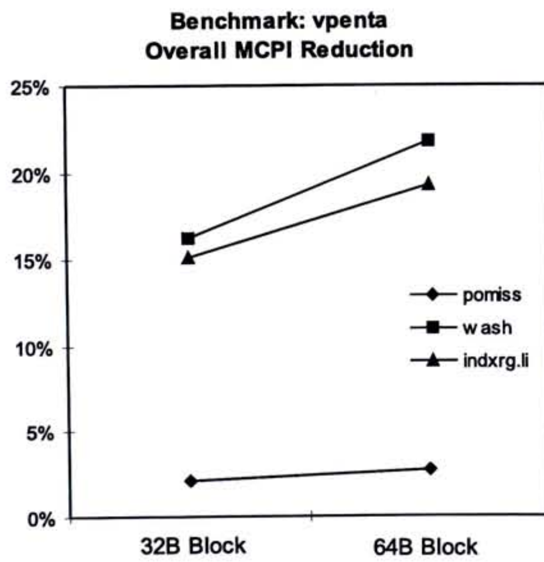


Figure 5.12: Overall MCPI Reduction comparison by block size (cont.)

the startup time for the first bus width word is 10 CPU cycles, where the following word will be available at a rate of 6 CPU cycles. That equates to a transfer time of 28 CPU cycles per a 32-Byte second level cache block, as a 128-bit bus from the main memory to the second level cache was simulated. With a 64-Byte second level cache block, the transfer time required from the main memory to the second level cache is 52 CPU cycles. That means with a second level cache miss using 64 Bytes cache block, the processor may have to stall for 52 cycles before it can continue. Therefore it is no doubt that a smaller cache block size is preferred.

2. The spatial locality gained by using a larger cache block size is minimal. With a 32-Byte cache block, it may already contain an element or two in an array for processing, the gain in further calling in more elements by using a larger cache block is very less. Or there were significant amount of large stride access that the stride value is far larger than 32 Bytes or 64 Bytes.

3. The larger cache block size used in a fixed amount of cache memory means the number of sets in the cache will be decreased. With a 128k Bytes cache, using 2-way set associative organization, there are 2048 sets if 32 Bytes cache block is used. However, with the same cache parameters, but using 64 Bytes block, there are 1024 sets available. The fewer number of sets means more memory addresses will be mapped to the same set, which the chance for conflict miss will become higher.

5.4.3 Set Associativity Effect

The three set associativity values, which are 1, 2 and 4, for the second level cache are plotted against the overall MCPI for all the benchmark programs in figures 5.13 – 5.15. The reduction in Overall MCPI charts are shown in figures 5.16 –

5.18.

There is a consistent trend that increasing the set associativity value improves the overall MCPI.

The decrease in overall MCPI is sharp when the set associativity is increased from 1 (direct mapped cache) to 2 (2-way set associative). It showed that there were some amount of conflict misses in all benchmarks when a direct mapped cache was used. However, the decrease in overall MCPI is leveled off when the set associativity is further increased from 2 to 4. The slope changes are different for different benchmarks. It showed that for some benchmarks, a higher set associativity is beneficial, even with a 4-way set associative organization, there is still room for overall MCPI improvement by reducing cache conflict misses. However, for the majority of benchmarks, the number of conflict miss had been reduced sharply with a 2-way set associative organization for the second level cache.

From all the tests we have performed, we can group the results on Overall MCPI into the following categories.

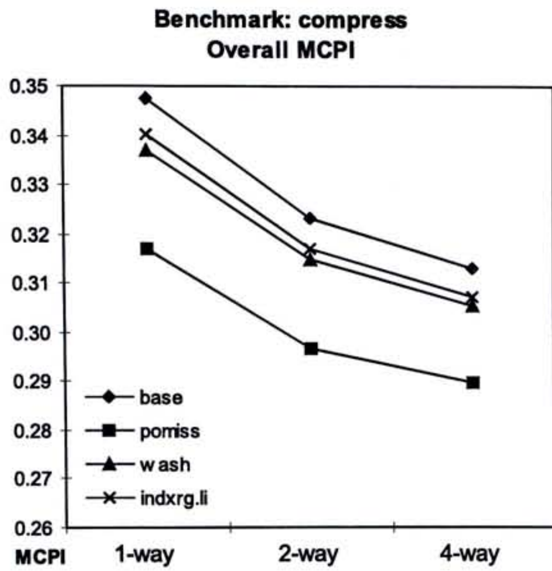
- Hardware Prefetch Algorithms
- Software Based/Hybrid Prefetch Algorithms

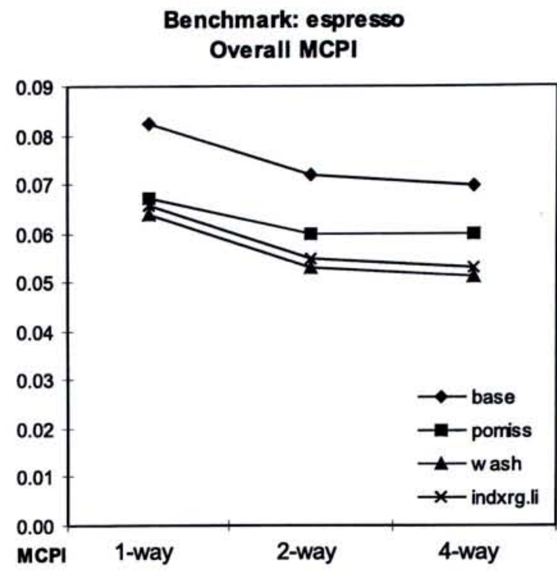
5.4.4 Hardware Prefetch Algorithms

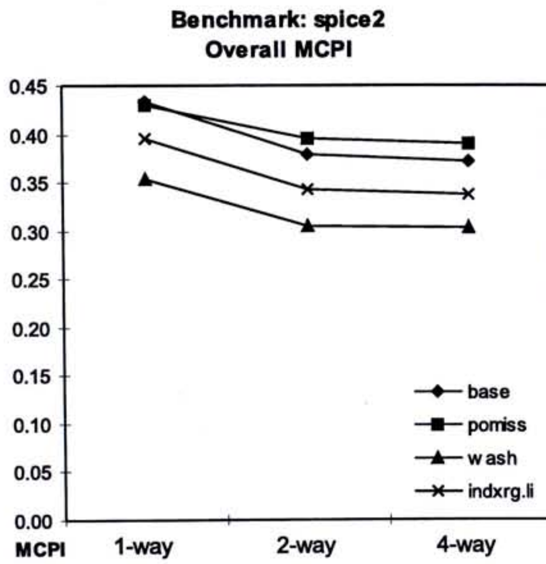
SIRPA family and similar schemes

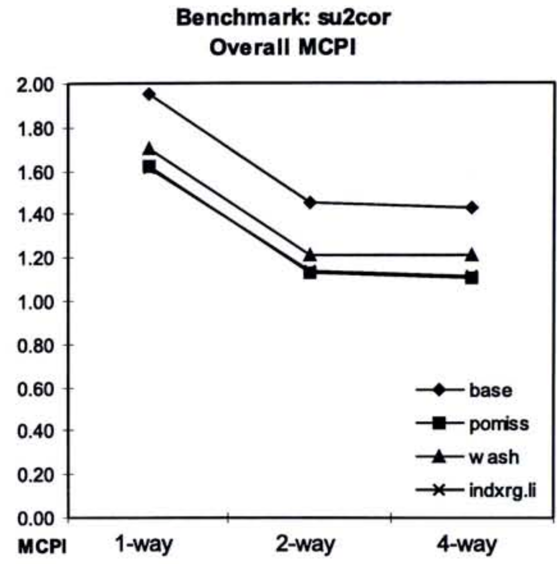
In the Hardware Prefetch Algorithms, we had performed tests on fifteen different benchmarks. The best performers in terms of MCPI in the corresponding benchmark test are:

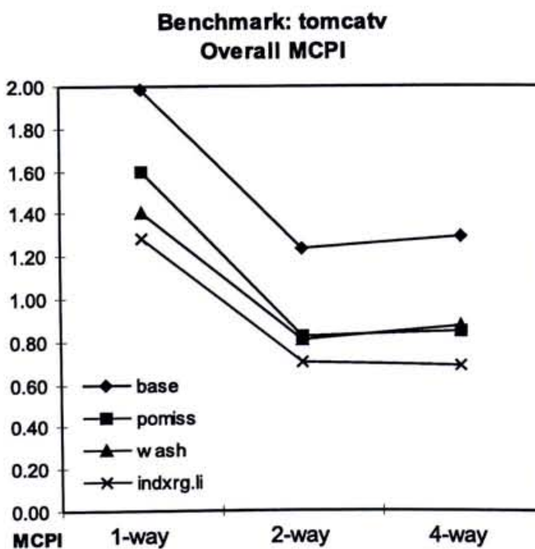
SIRPA and its enhanced versions won 8 out of 15 benchmark tests. The remaining winners were enhanced versions of Chen's RPT scheme.











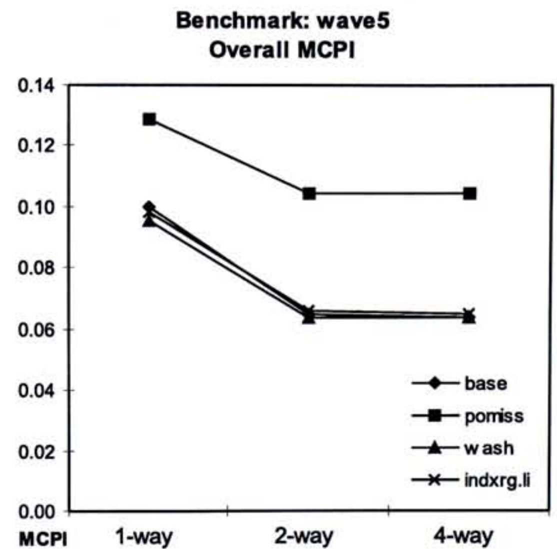


Figure 5.13: Overall MCPI comparison by associativity

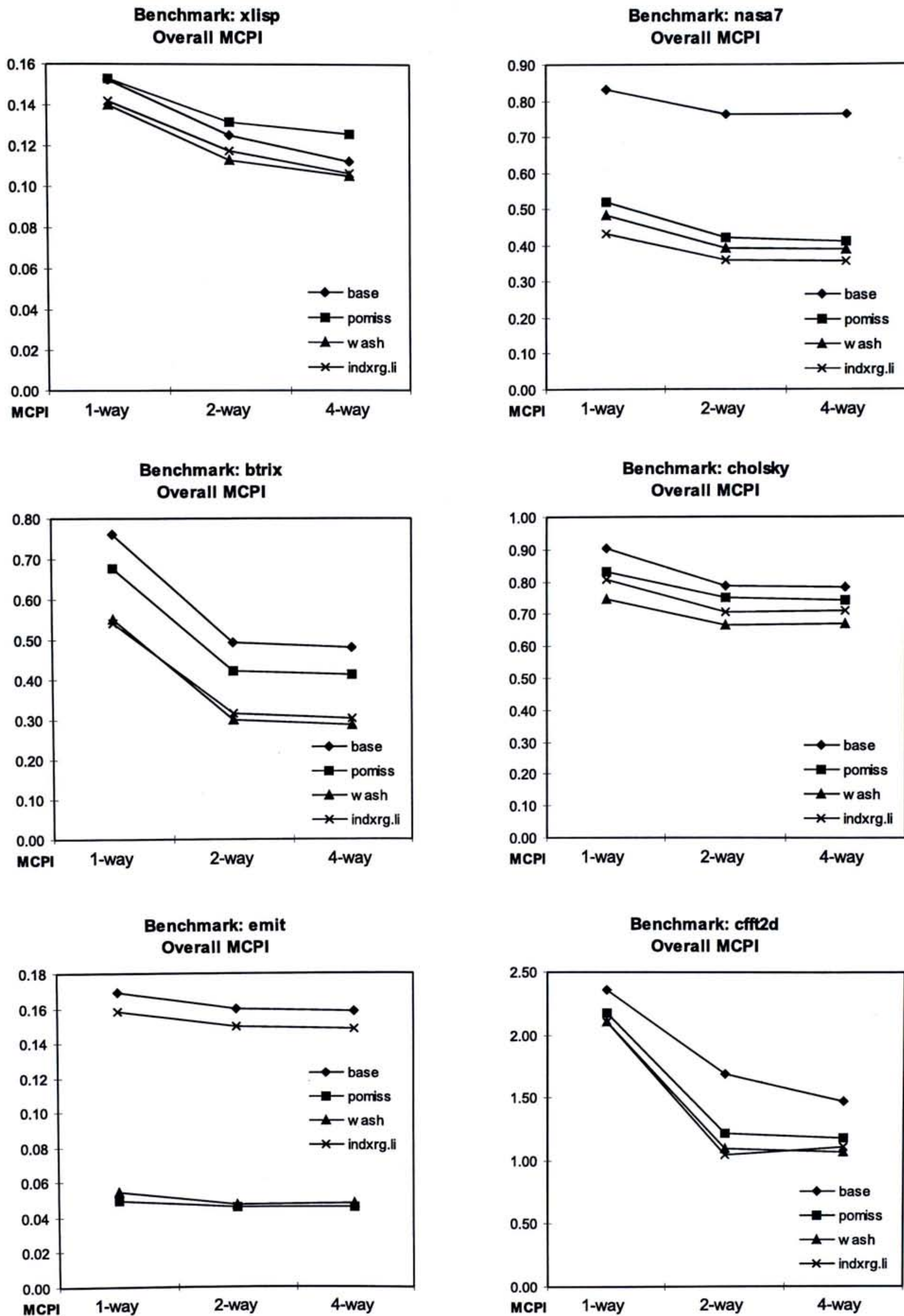
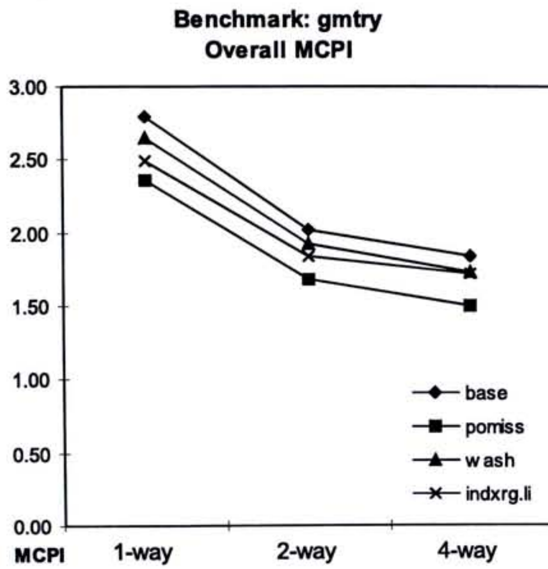
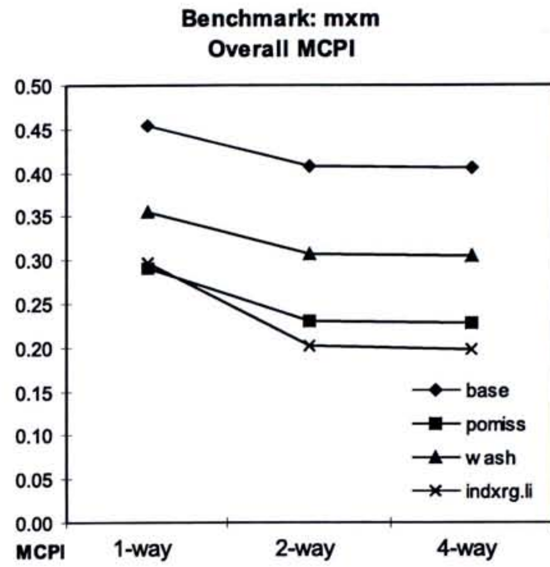


Figure 5.14: Overall MCPI comparison by associativity (cont.)





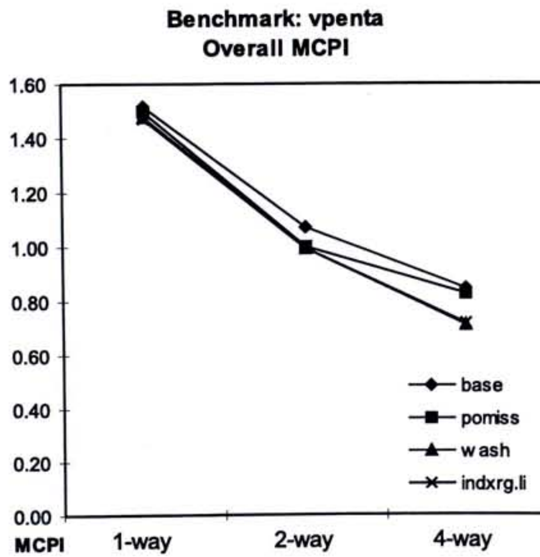
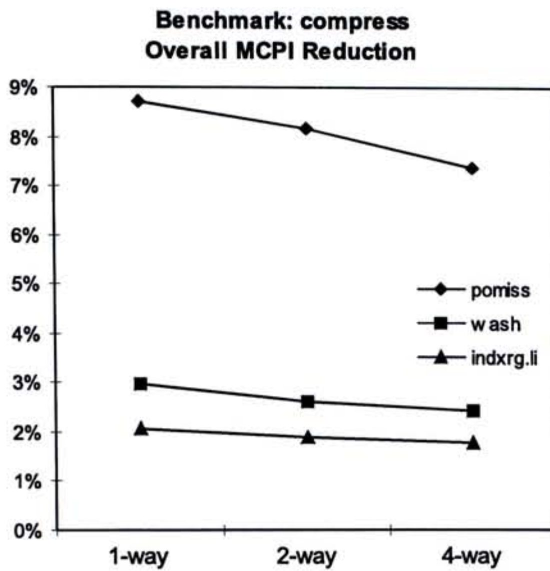
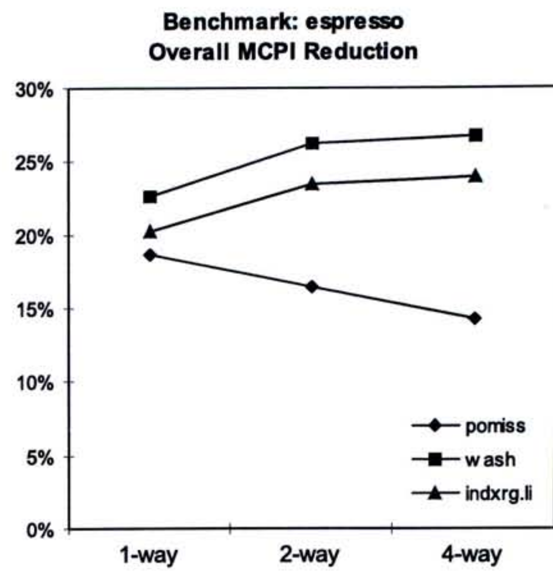
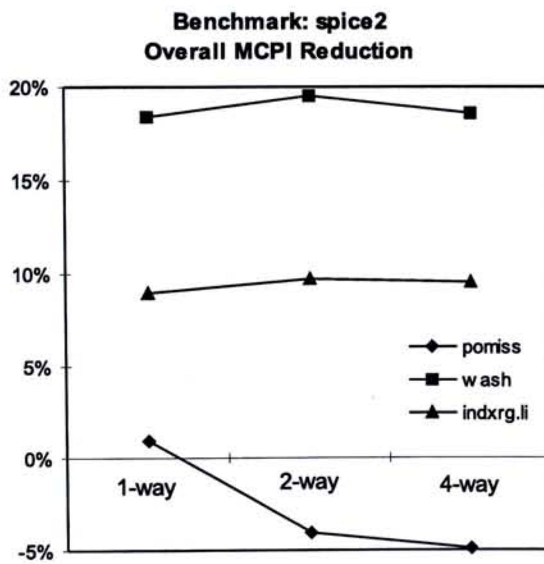
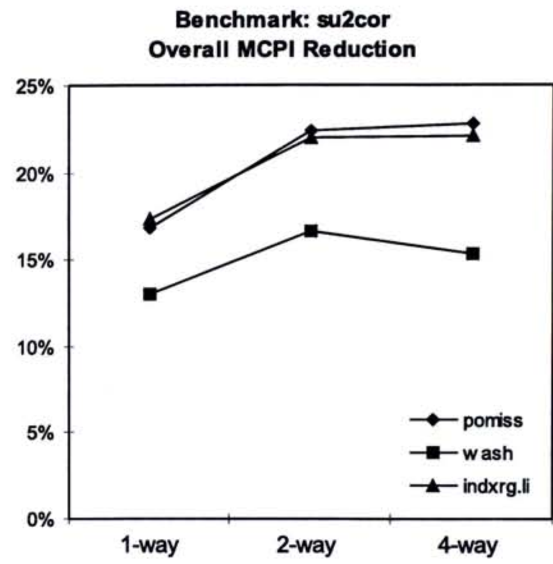


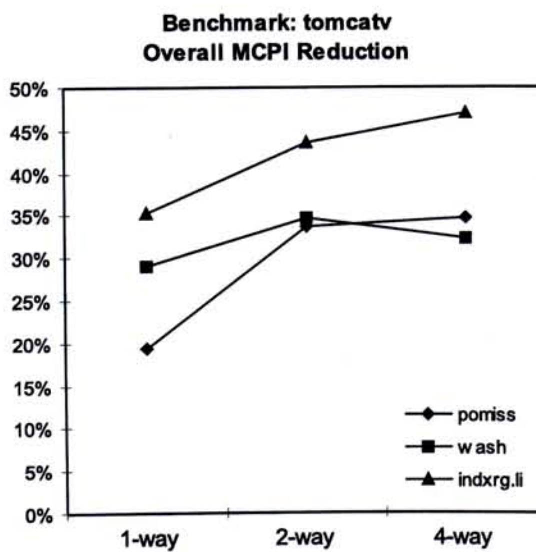
Figure 5.15: Overall MCPI comparison by associativity (cont.)











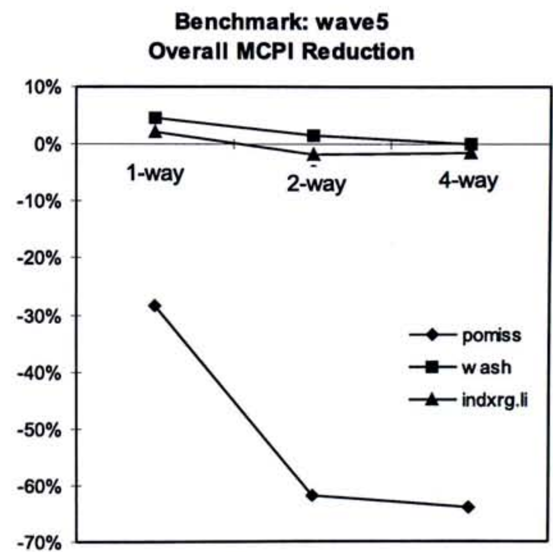
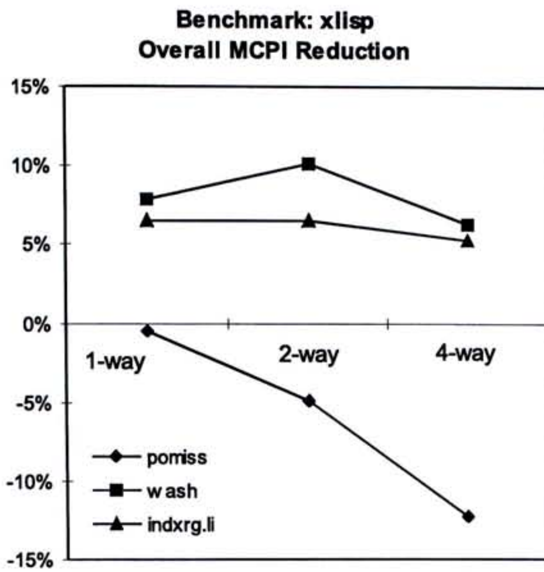
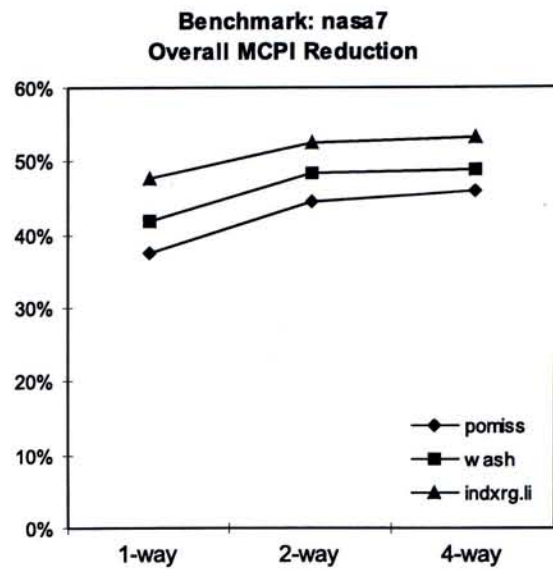
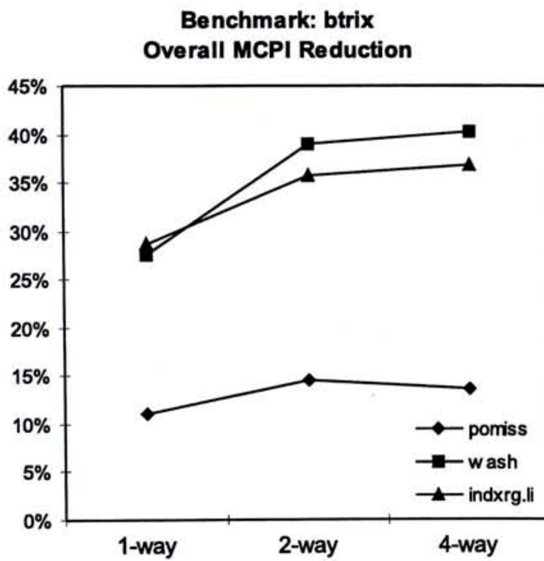
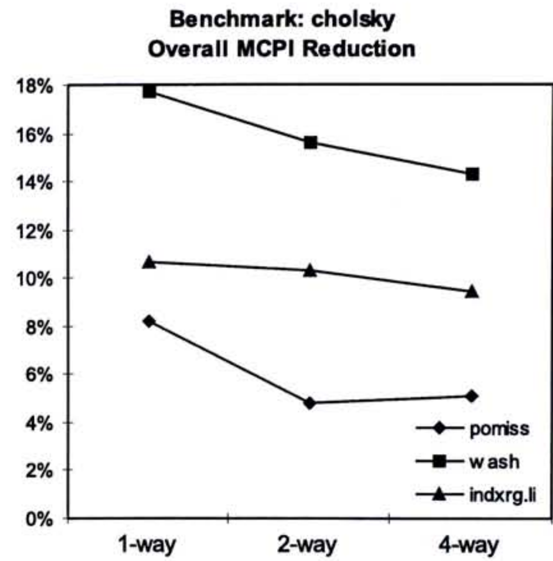


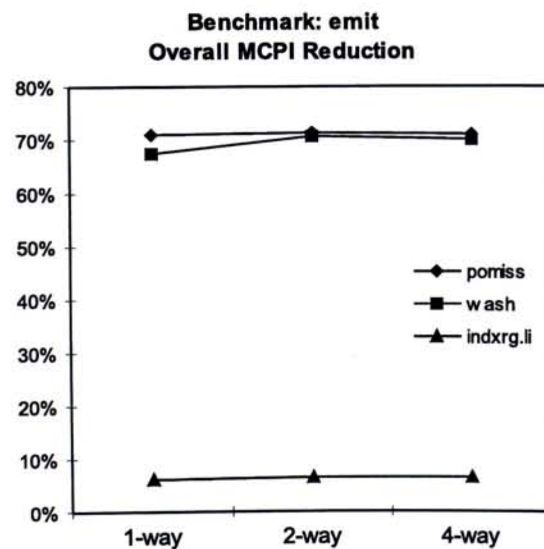
Figure 5.16: Overall MCPI Reduction comparison by associativity











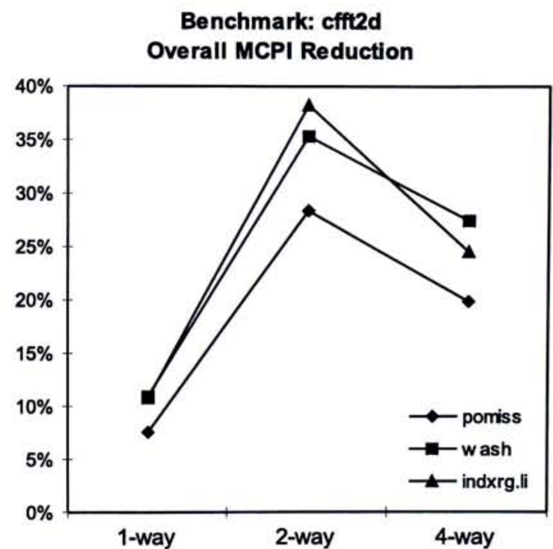
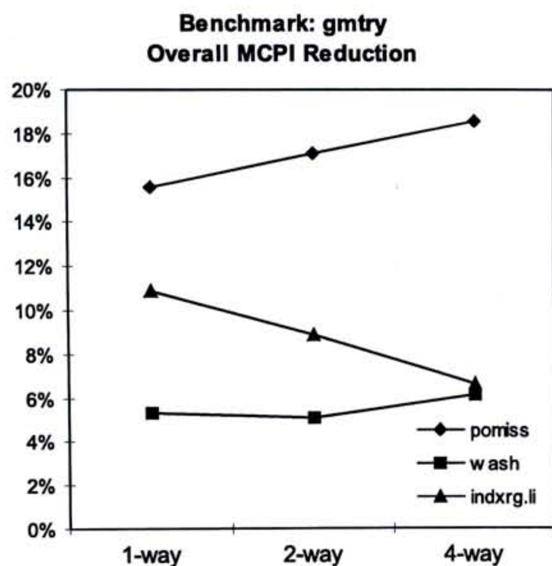
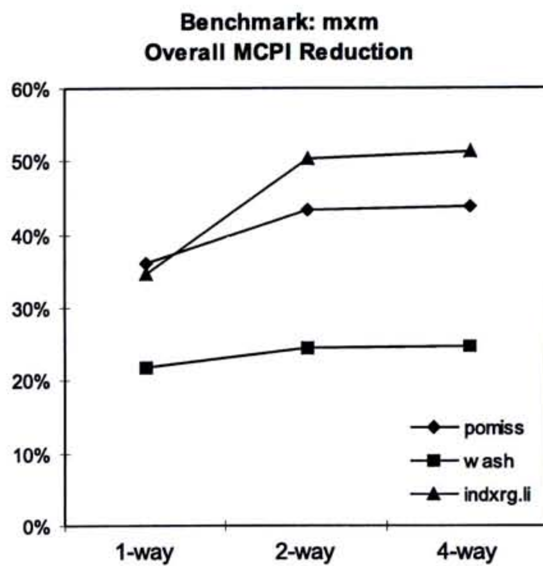


Figure 5.17: Overall MCPI Reduction comparison by associativity (cont.)





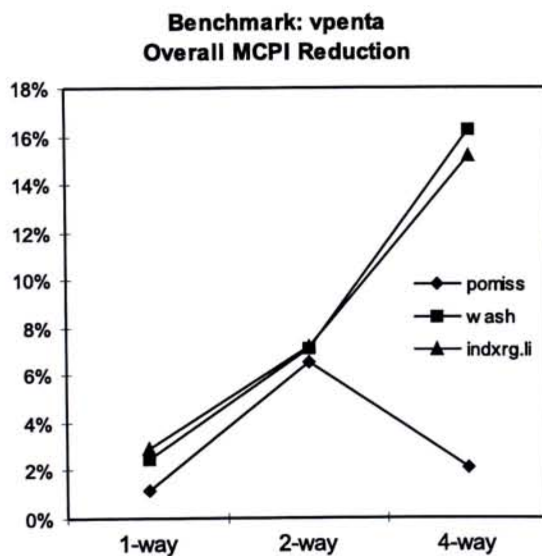


Figure 5.18: Overall MCPI Reduction comparison by associativity (cont.)

Table 5.1: Best Prefetch Algorithm Producing Smallest Overall MCPI

Benchmark	**Best Prefetch Algorithm (Smallest MCPI)**
compress	SIRPA w/ Line Concept w/ Default Prefetch
espresso	Chen's RPT w/ Fully Associative Entries
spice2	Chen's RPT w/ Fully Associative Entries
su2cor	SIRPA w/ Line Concept w/ Default Prefetch
tomcatv	SIRPA w/ Line Concept
wave5	Chen's RPT w/ Line Concept
xlisp	Chen's RPT w/ Fully Associative Entries
nasa7	SIRPA w/ Default Prefetch
btrix	Chen's RPT w/ Fully Associative Entries
cholsky	Chen's RPT w/ Fully Associative Entries
emit	SIRPA w/ Line Concept w/ Default Prefetch
cfft2d	SIRPA w/ Line Concept
gmtry	SIRPA w/ Line Concept w/ Default Prefetch
mxm	SIRPA w/ Line Concept
vpenta	Chen's RPT w/ Fully Associative Entries

In numerical sense, we listed out the top performers for respective benchmark tests by using SIRPA family and Chen's RPT family of cache prefetch algorithms. The values listed are the MCPI for the respective benchmark programs. The ratios listed are the result by dividing the two numbers. The values without bracket mean the SIRPA family performed better, and the ratios listed with brackets mean the Chen's RPT's family performed better.

The results showed that SIRPA family of prefetching algorithms performed slightly better than the Chen's RPT family. However, the difference on the top

performers from both families is quite small. It is not surprised as both the SIRPA and Chen's RPT cache prefetch algorithms tackle the constant stride access patterns in programs.

The difference in performance between SIRPA and Chen's RPT algorithms can be attributed to the following reasons.

1. When the entries in Chen's RPT are frequently replaced, some of the instruction addresses that should have a constant stride access pattern may be flushed out from the RPT by other instruction address. This point can be deduced from the above table that most top performing cache prefetch algorithms using Chen's RPT have to use fully associative RPT. When a direct mapped RPT is used, the conflict in RPT entries would be more likely to happen.

2. When the number of arrays used in a program loop has exceeded the number of available registers to hold the array addresses, register reuse may make the SIRPA's SVT unable to detect the constant stride patterns.

The SIRPA scheme has much less hardware overhead compare with the Chen's RPT scheme, because the SVT size in SIRPA can be significantly smaller. In all the tests performed, the number of entries in SVT is 32 verses the number of entries in RPT is 128. There is no LA-PC and BPT in the SIRPA scheme. As shown in the above table, the cache performance of SIRPA is on par with Chen's RPT, but with the same number of transistors available on chip, the chip using SIRPA can make a much bigger cache which the same area is occupied by the larger RPT, LA-PC and BPT in Chen's RPT scheme.

Compare with the PFONMISS scheme, either SIRPA or Chen's RPT consistently performs better. The following in a table showing the best MCPI by using PFONMISS, SIRPA and RPT. The values in bracket show the percentage using

the value in the PFONMISS column as a base.

We must aware that even the simple PFONMISS scheme may perform as well as the more sophisticated SIRPA and Chen's RPT schemes in some benchmarks (compress, su2cor, emit, gmtry). However, the built-in stride value checking mechanism in the SIRPA and Chen's RPT schemes assures the above two schemes would not degrade the system performance when the majority of memory access is not constant stride type. The above two schemes would just fire no cache prefetch at all when the memory access pattern is not certain, and the cache system become to use demand fetch only.

On the other hand, the PFONMISS scheme may cause cache pollution if the majority of memory access is with large stride values. As can be seen from the wave5 benchmark (in graph containing in Appendix), the best performing cache configuration using PFONMISS still cause the MCPI to degrade by 19.58comparing with the base without using any cache prefetch algorithm. The worst case using PFONMISS on wave5 made the memory latency almost 70than without cache prefetch at all.

Although pure SIRPA and Chen's RPT schemes may still degrade memory latency, the effect is much less severe than the PFONMISS scheme. The degradation experienced may due to the limited size of cache memory, which the prefetched cache block replaced another block that is called by the processor earlier than the prefetched one. The most severe degradation in MCPI is a mere 2.8benchmark.

Line Concept

We can observe from the top performing algorithms' table that, both the Line Concept and the Default Prefetch did help in the cache prefetch algorithms. 8 out of 15 top performing algorithms required the Line Concept to improve the MCPI.

From the data in Appendix, we found that almost all tests show that the Line Concept is useful in improving performance. The Line Concept is equally well performed with the SIRPA scheme, Chen's RPT scheme and even the software prefetch instruction scheme.

The Line Concept is a new enhancement with Chen's RPT scheme. The further reduction in MCPI could be 6Line Concept with other cache parameters the same. The Line Concept gave an even higher performance boast when using with SIRPA scheme. The further reduction in MCPI could be as high as 91using Line Concept with other cache parameters the same.

The Line Concept is useful when the stride value is small, and a few data items have been mapped to the same cache block. The Line Concept will call in the next cache block earlier than without. The overlapping in computation and memory transfer can then be enhanced.

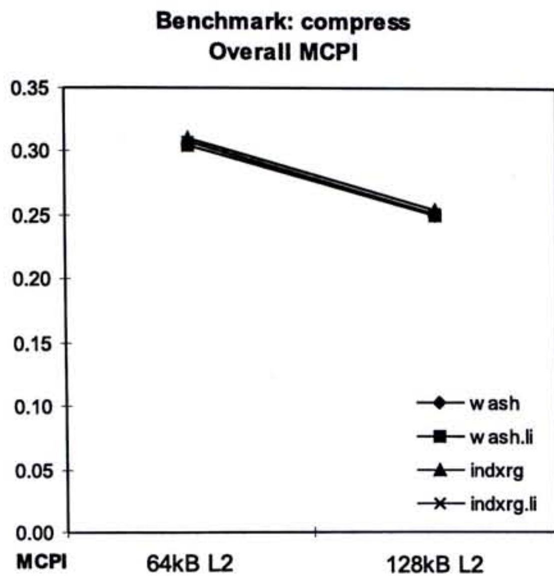
In figures 5.19 – 5.21, the Overall MCPI values of using Line Concept with SIRPA scheme and Chen's RPT scheme are shown. The corresponding MCPI values due to L2 and Main Memory are plotted in figures 5.22 – 5.24.

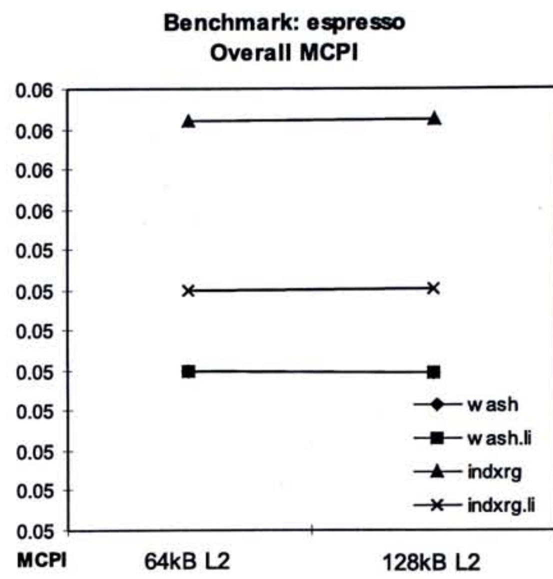
Default Prefetch

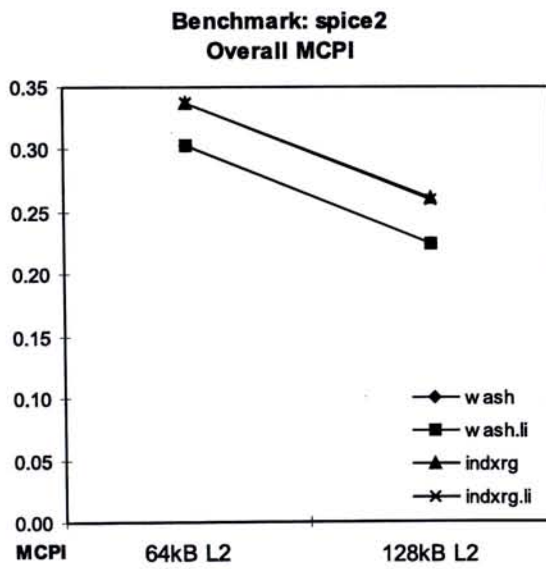
5 out of 9 SIRPA family algorithms in the top performing table required the Default Prefetch scheme. The further reduction in MCPI could be 285emit kernel of NASA7 by using Default Prefetch with other cache parameters the same.

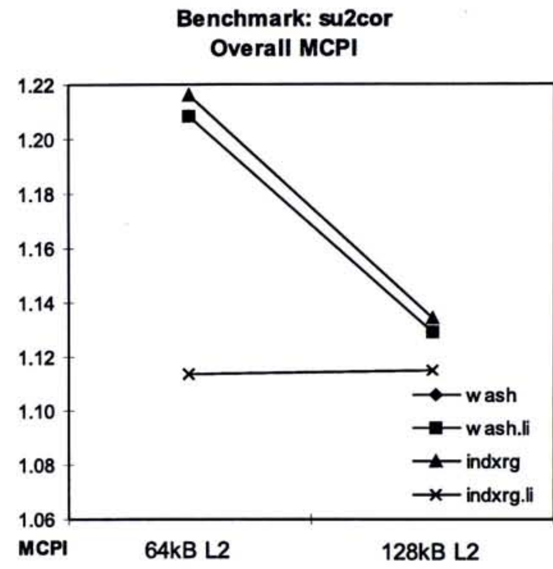
The Default Prefetch scheme is useful when the constant stride accesses are not too many, but the program still exhibits a majority of spatial locality. There are benchmark programs with a very large reduction in MCPI by using PFON-MISS. By using Default Prefetch with SIRPA, the reduction in MCPI will also be very high in these benchmark programs.

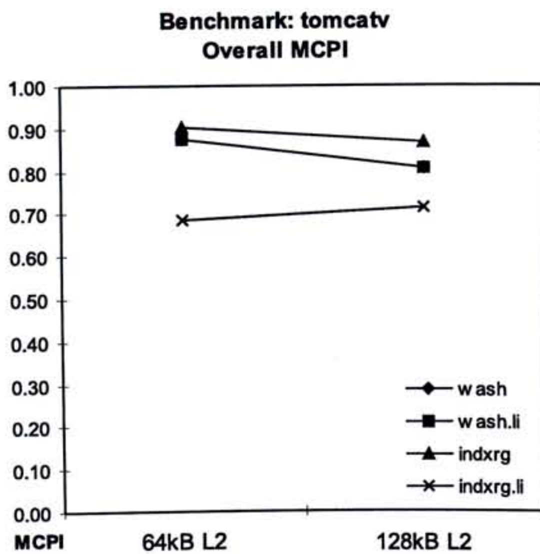
When Default Prefetch is added on the pure SIRPA scheme, in the bench-











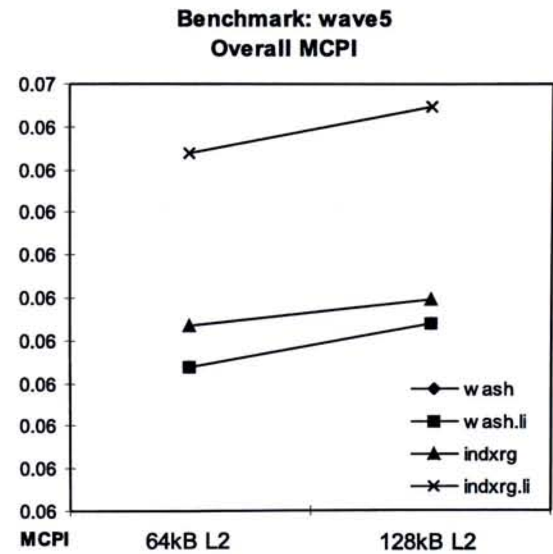
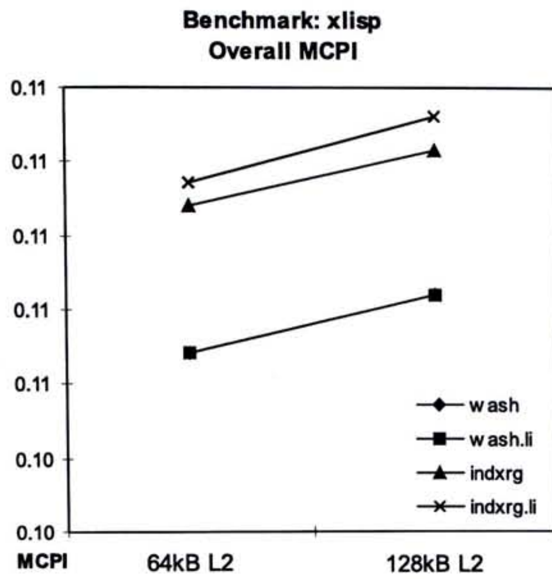
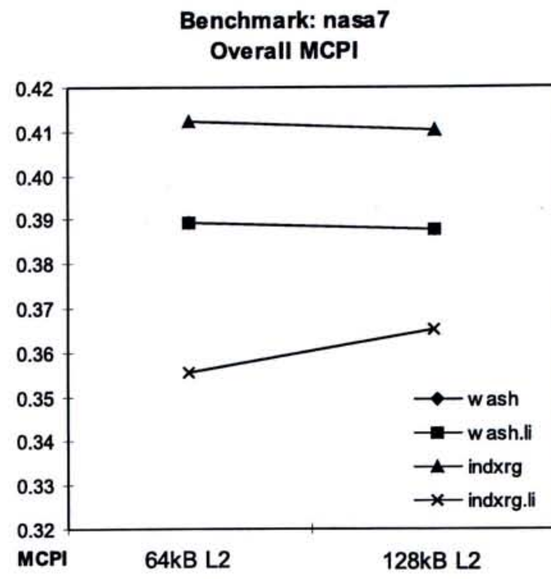
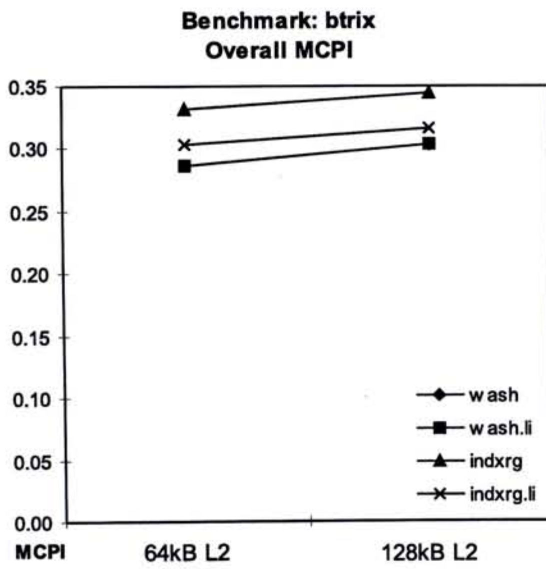
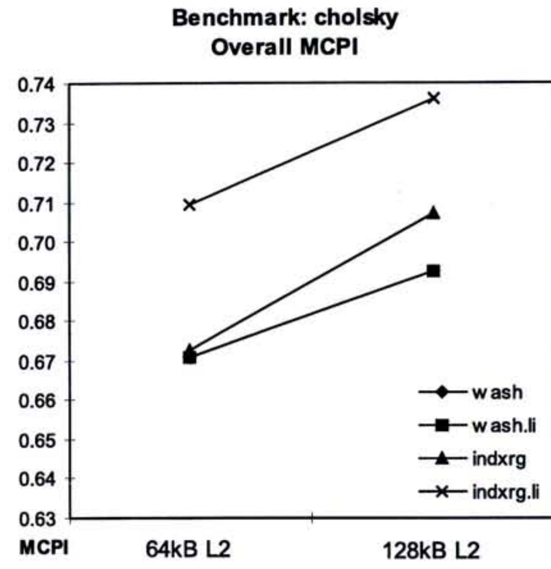


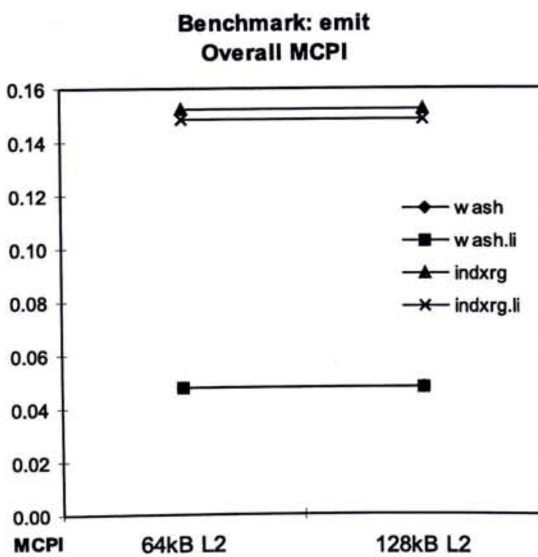
Figure 5.19: Overall MCPI Comparsion of Line Concept











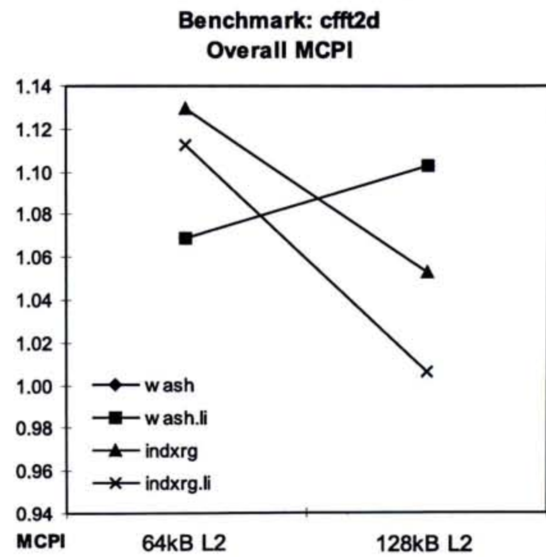
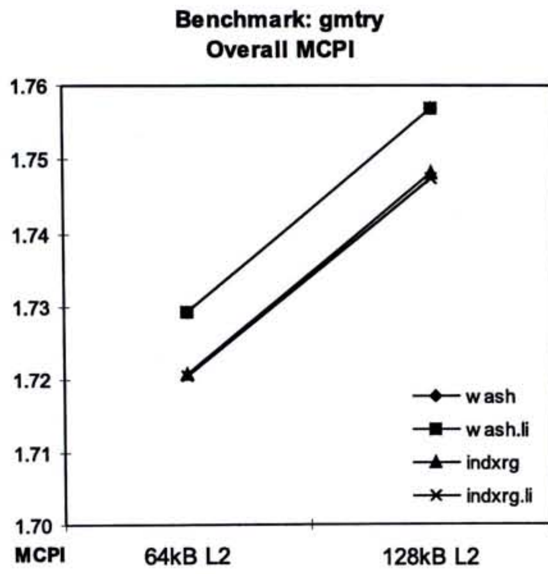
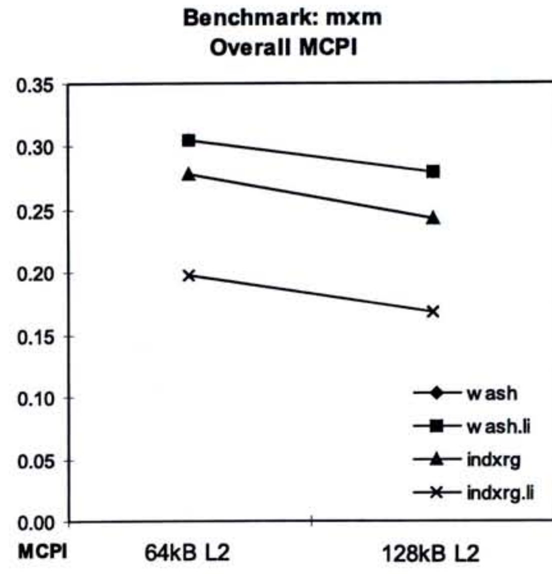


Figure 5.20: Overall MCPI Comparsion of Line Concept (cont.)





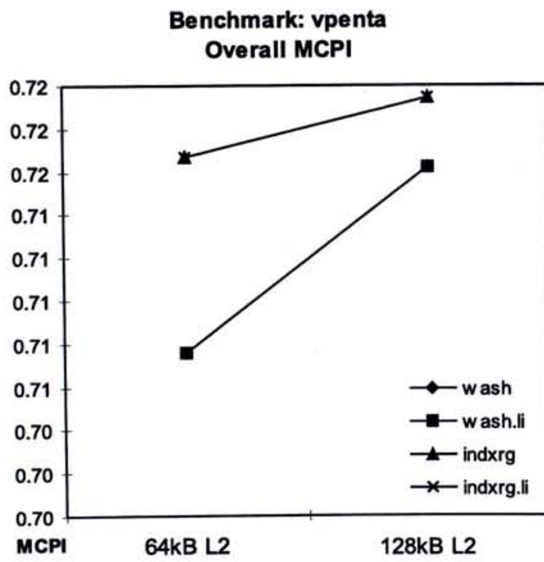
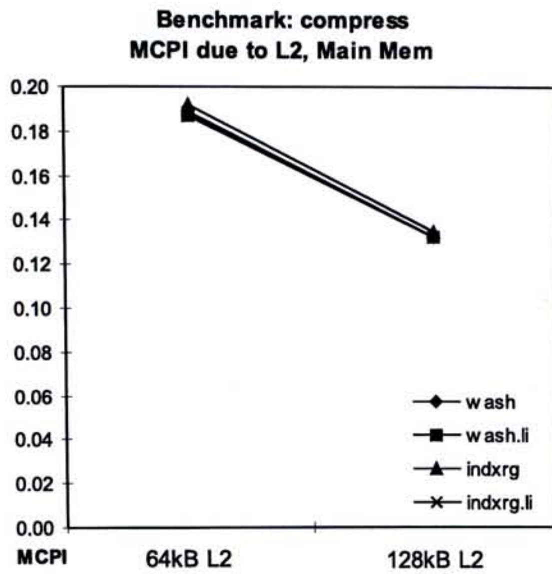
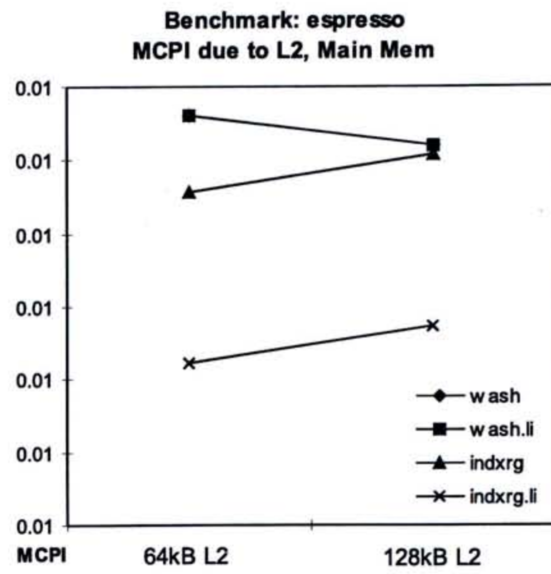
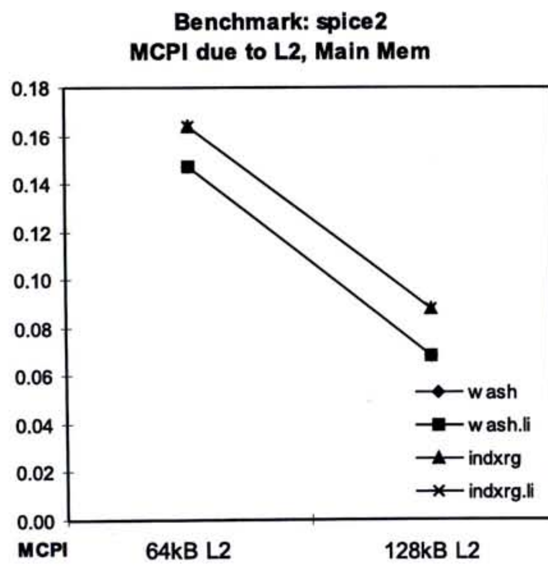
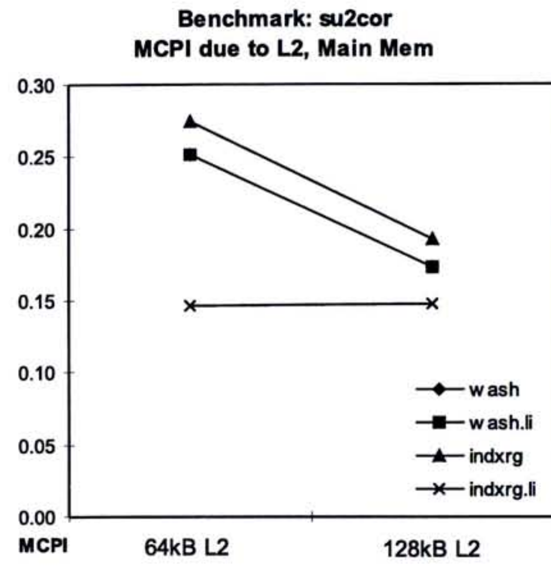


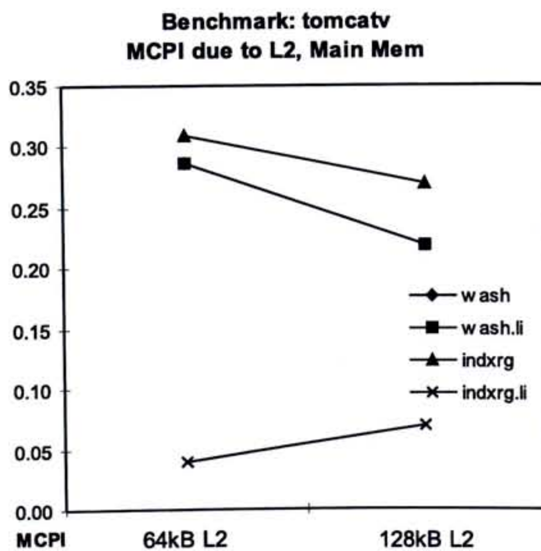
Figure 5.21: Overall MCPI Comparsion of Line Concept (cont.)











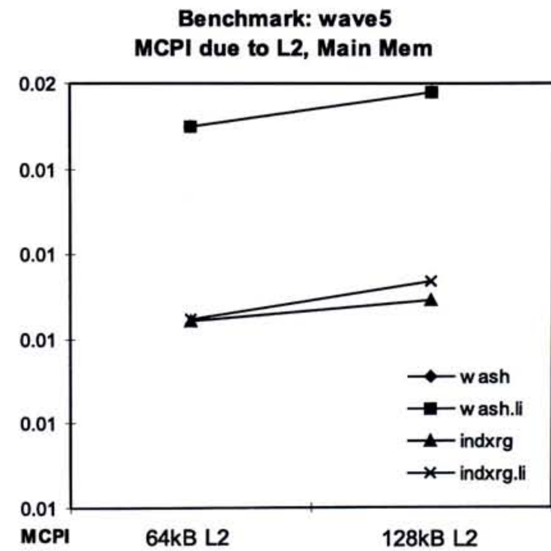
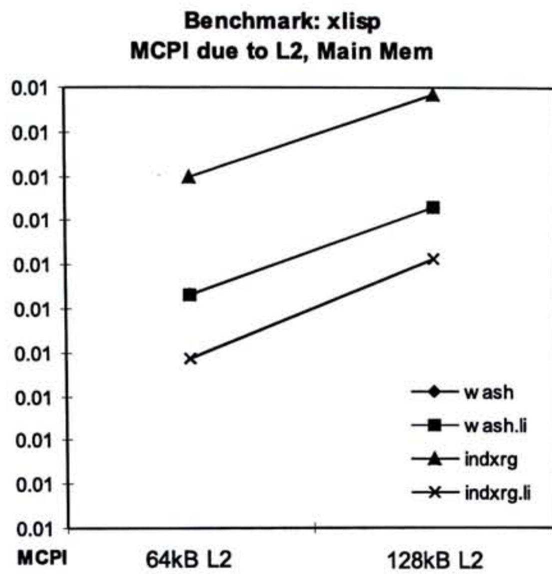
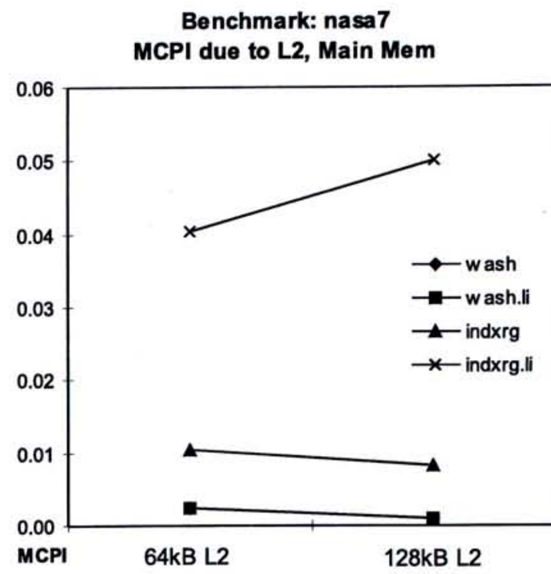
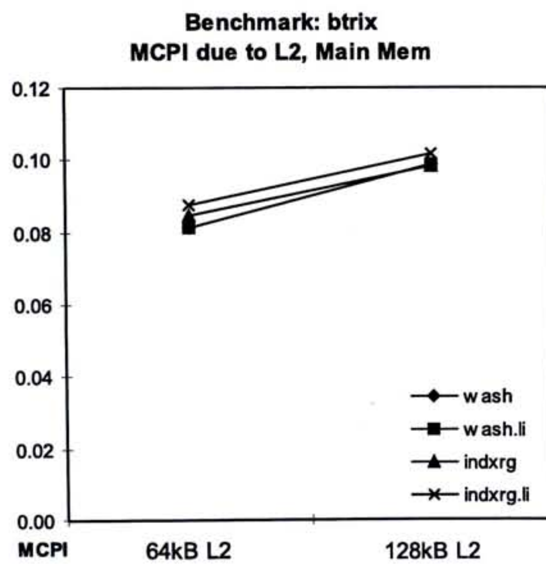
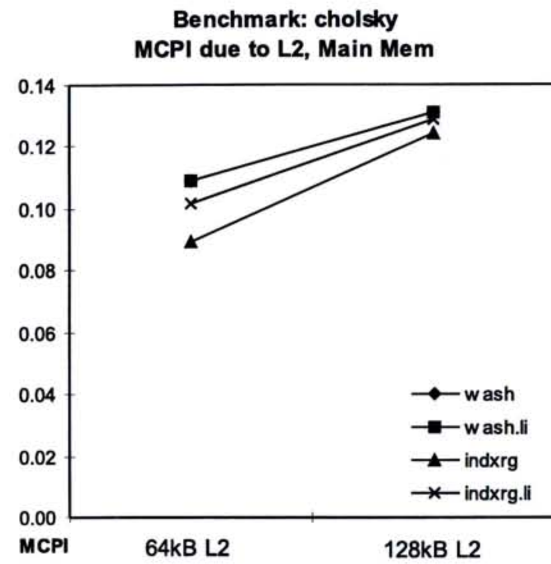


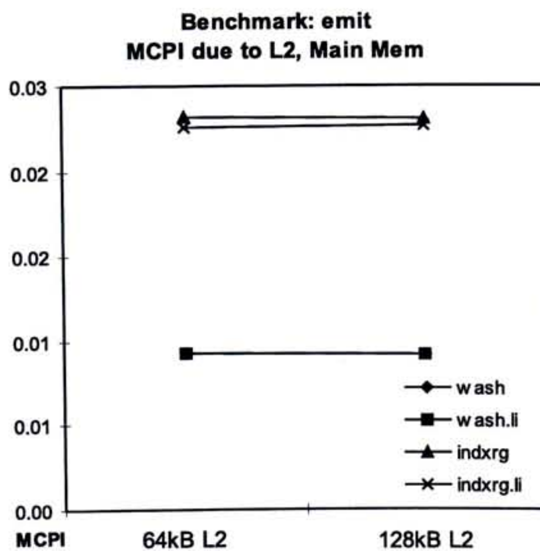
Figure 5.22: L2/Main Mem MCPI Comparsion of Line Concept











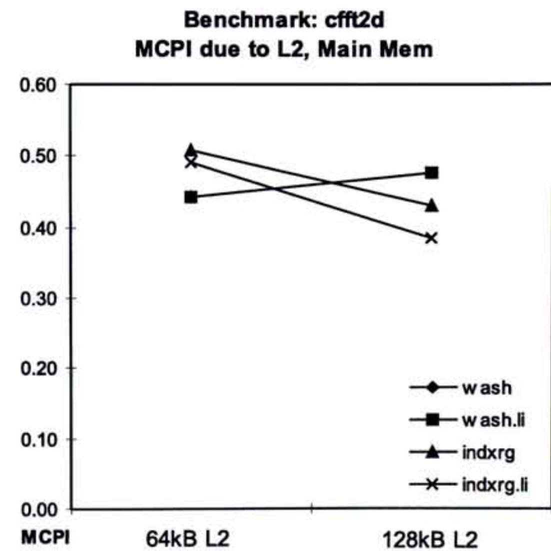
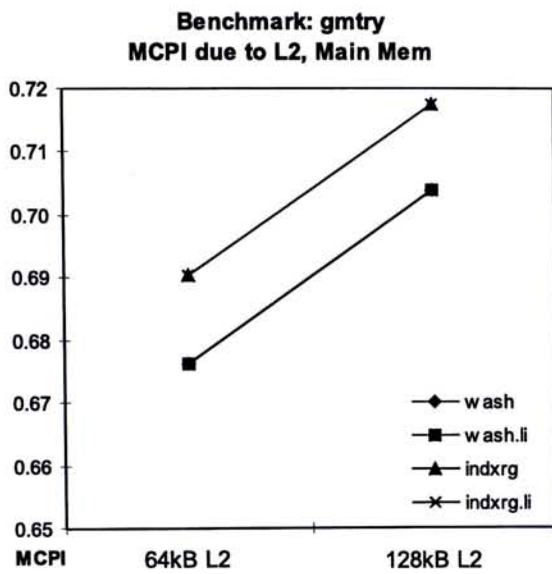
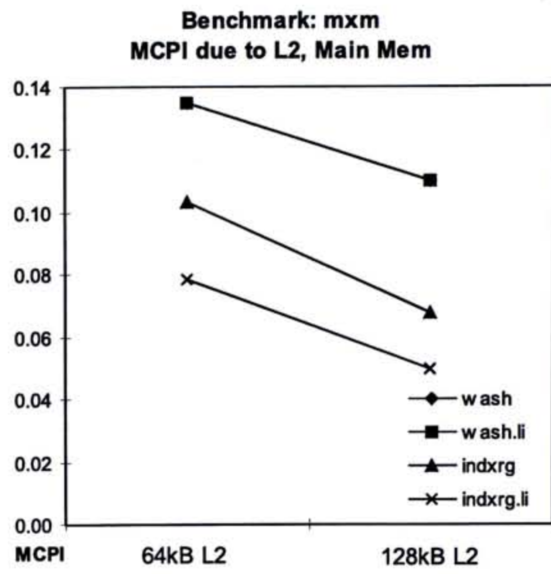


Figure 5.23: L2/Main Mem MCPI Comparsion of Line Concept (cont.)





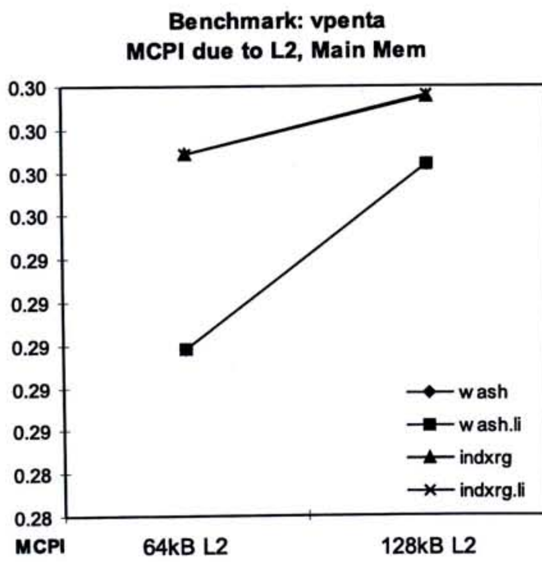


Figure 5.24: L2/Main Mem MCPI Comparsion of Line Concept (cont.)

marks that PFONMISS performed poorly, the MCPI generated by using SIRPA with Default Prefetch will also be degraded. The same problem of cache pollution will be added into the SIRPA with Default Prefetch scheme.

The Overall MCPI values for SIRPA scheme with and without using Default Prefetch enhancement are shown in figures 5.25 – 5.27. The corresponding MCPI values due to L2 and Main Memory are shown in figures 5.28 – 5.30.

5.4.5 Software Based Prefetch Algorithms

In this study, two different software based prefetch algorithms were tested. They are Software Prefetch Instruction (PREFETCH) and SETCAM scheme [Ho95]. The Line Concept was applied to both of the two software based prefetch algorithms to test any possible benefit.

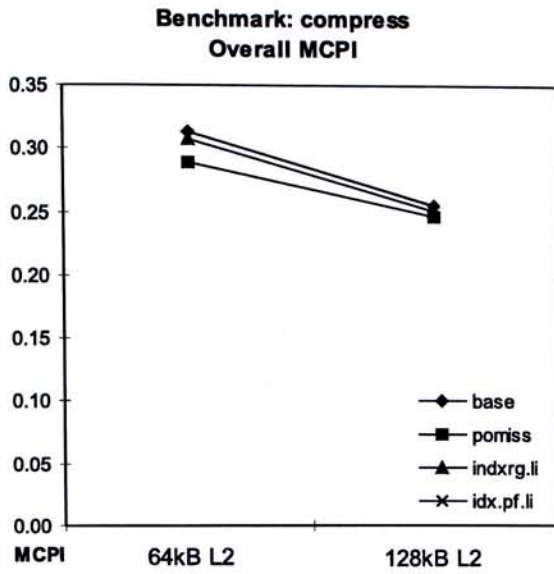
Only the 7 kernels in NASA7 has the appropriate special instructions embedded into the trace files, and the software based prefetch algorithms can be tested on these files.

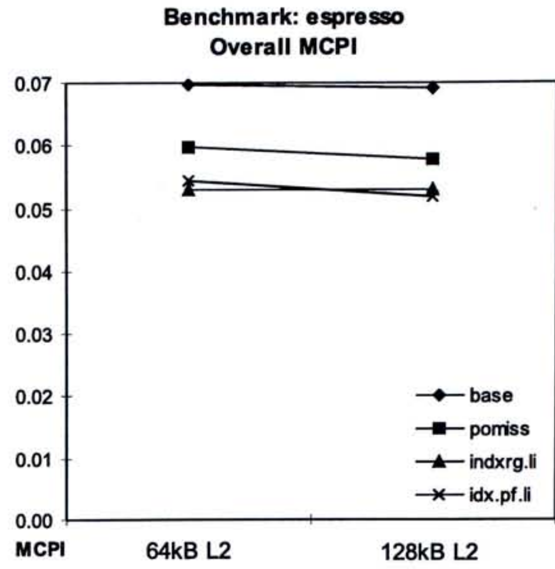
The best reduction in MCPI by using software based prefetch algorithms is summarized in the following table.

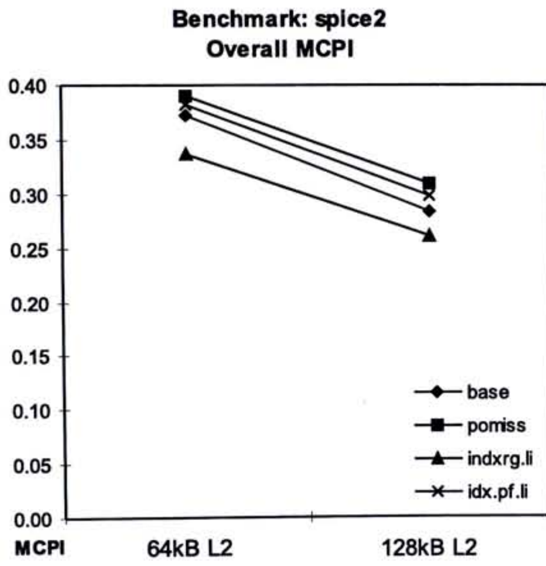
The clear cut winner in software based prefetch algorithm is without doubt SETCAM w/ Line Concept scheme. The only benchmark that PREFETCH won is the cfft2d, but the margin was too small to be significant. The reduction in MCPI by using PREFETCH with Line Concept is 0.75MCPI by using SETCAM with Line Concept is 0.74

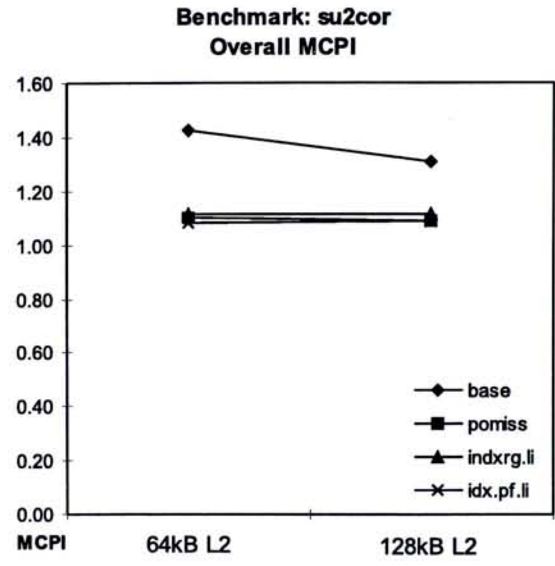
The results were consistent with those presented in [Ho95]. However, in the tests performed in this study involves 2 level cache hierarchy, but in [Ho95], the results were simulated by using 1 level cache memory.

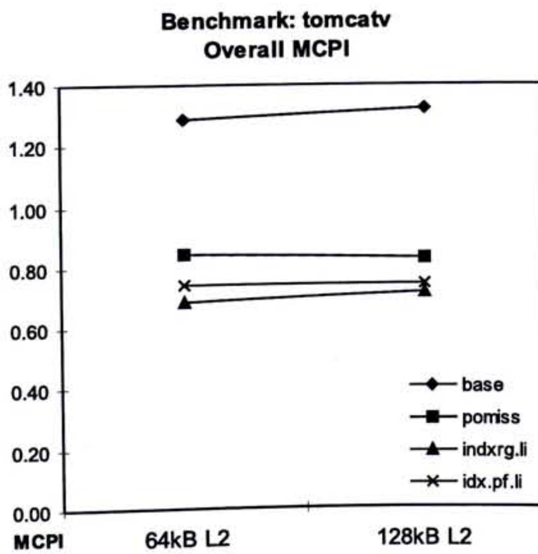
Line Concept is again proved to be useful in improving the tested software based prefetch algorithms. The only exception is the cholsky benchmark, which











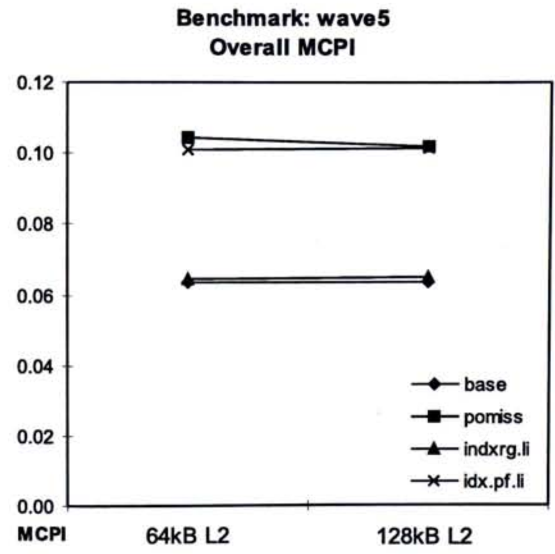
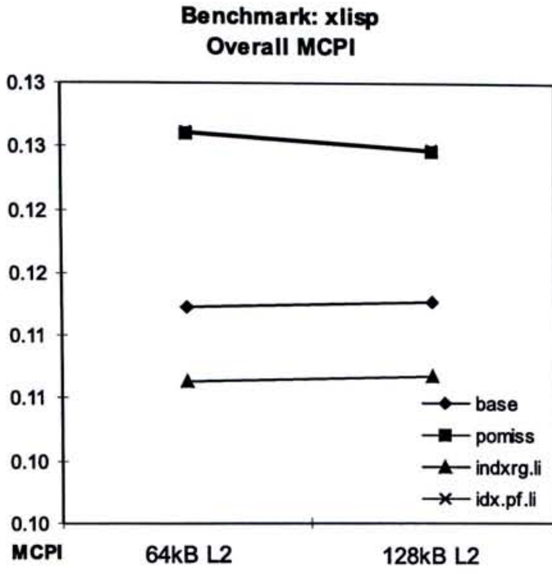
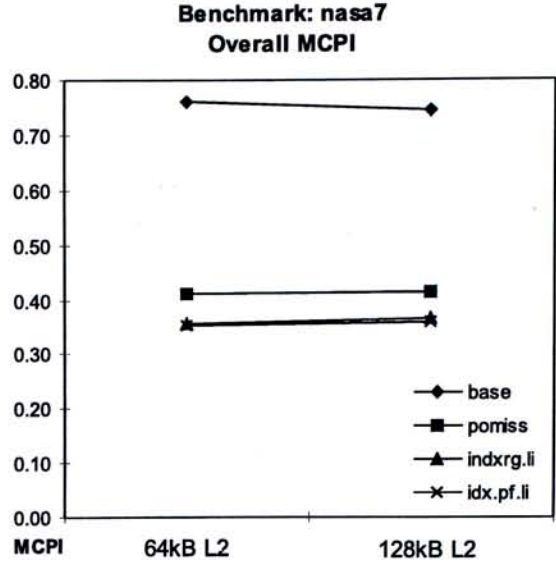
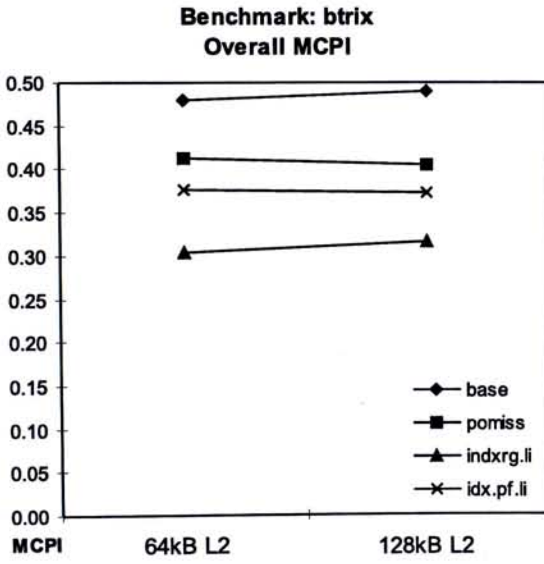
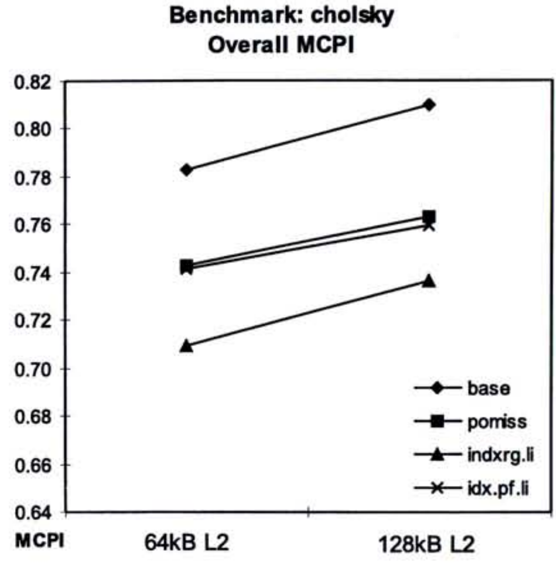


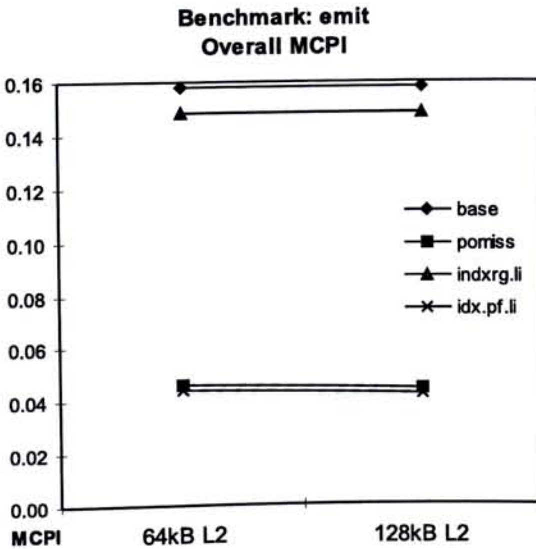
Figure 5.25: Overall MCPI Comparison of Default Prefetch Scheme











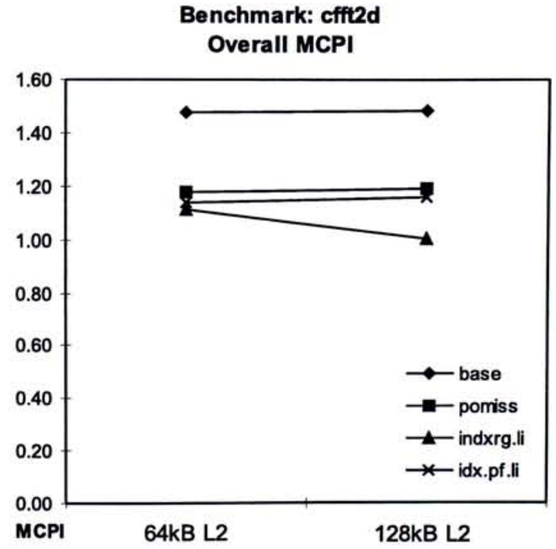
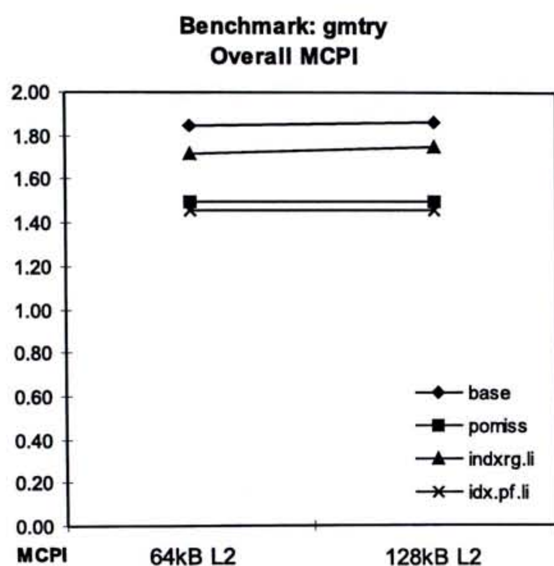
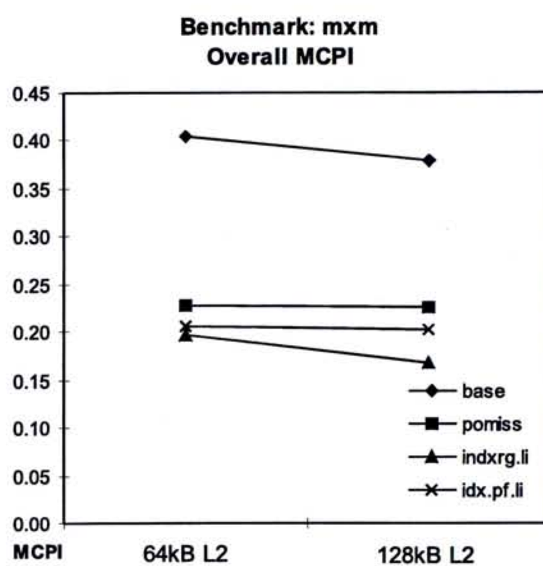


Figure 5.26: Overall MCPI Comparison of Default Prefetch Scheme (cont.)





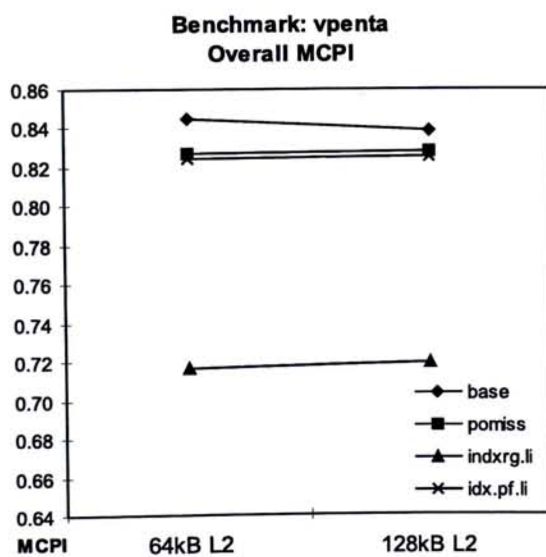


Figure 5.27: Overall MCPI Comparison of Default Prefetch Scheme (cont.)

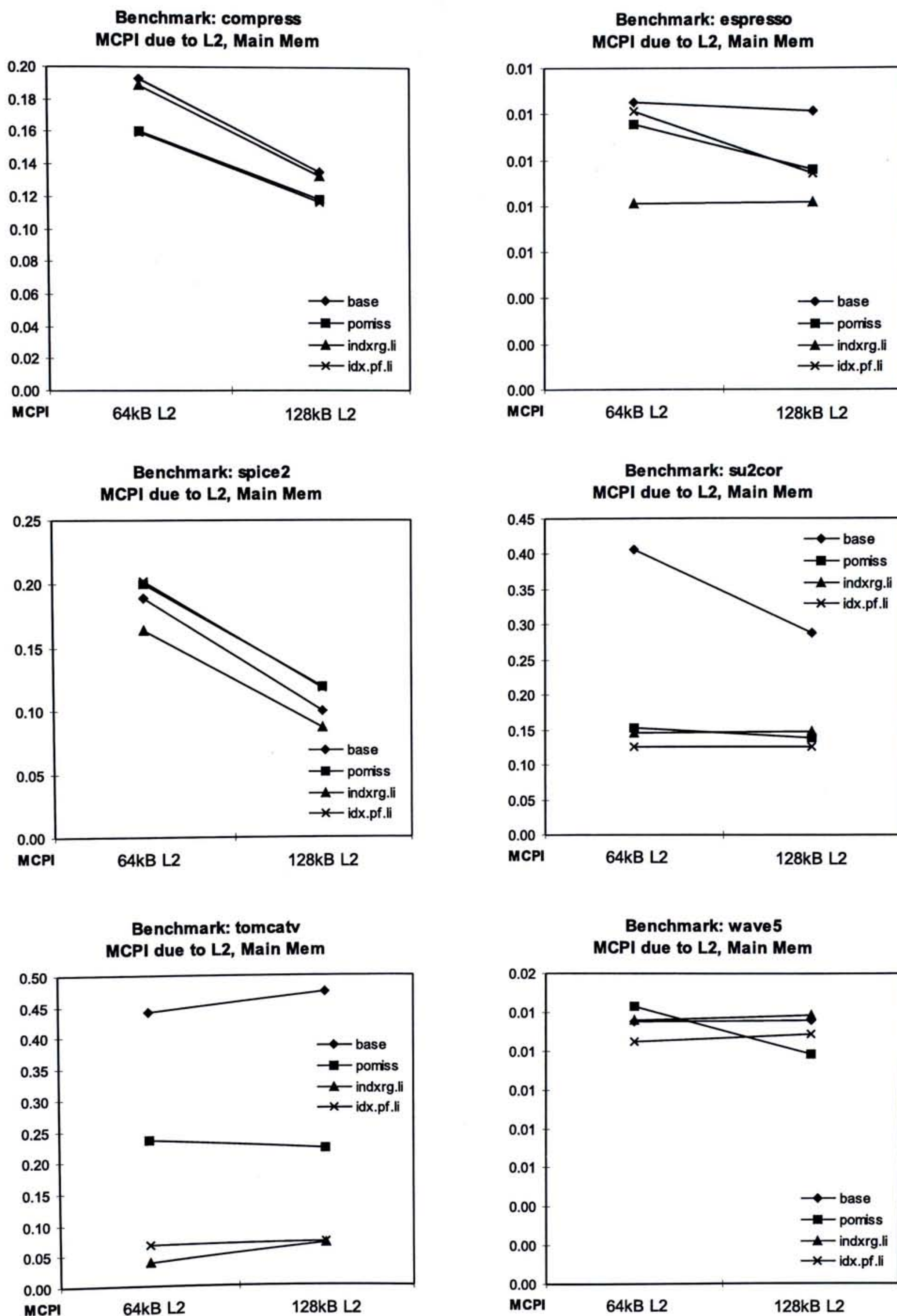
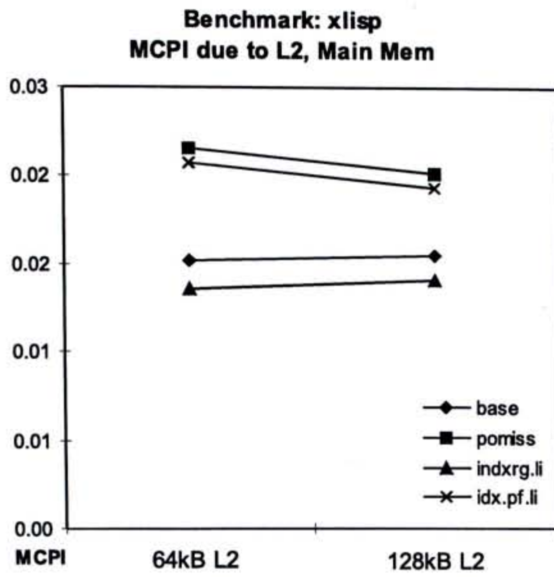
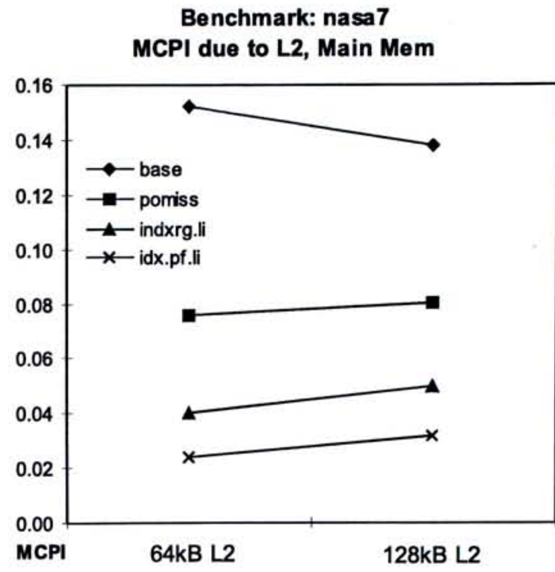
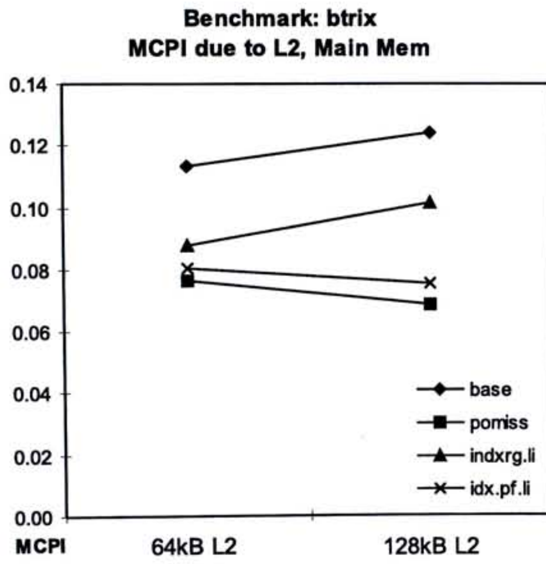
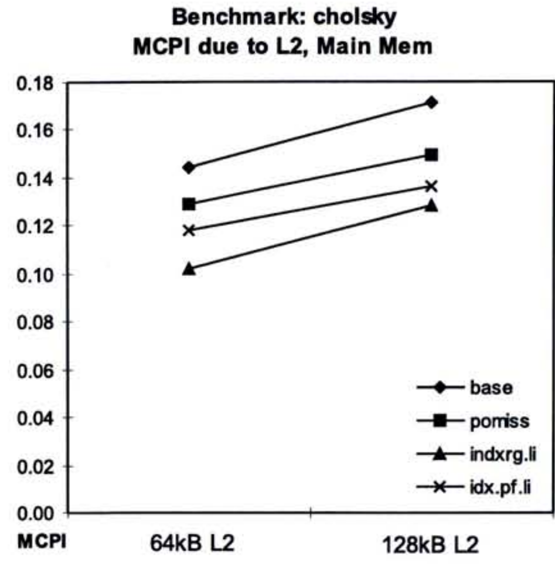


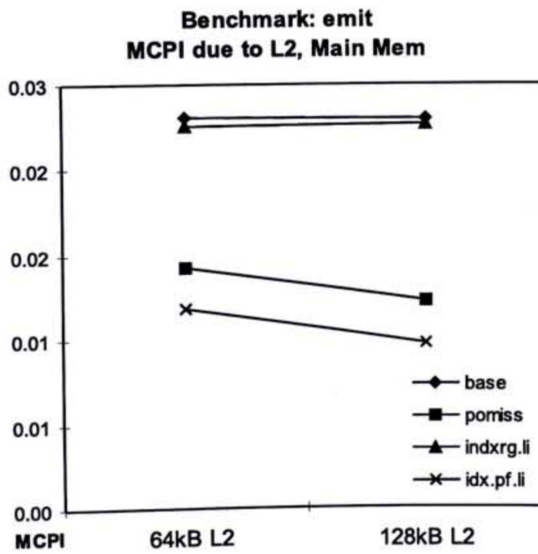
Figure 5.28: L2/Main Mem MCPI Comparison of Default Prefetch Scheme











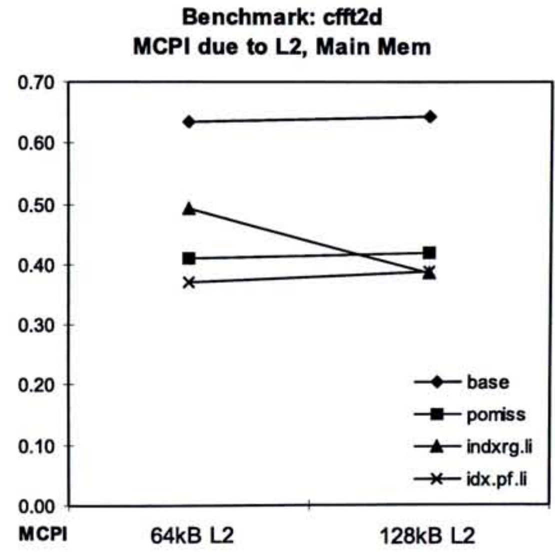
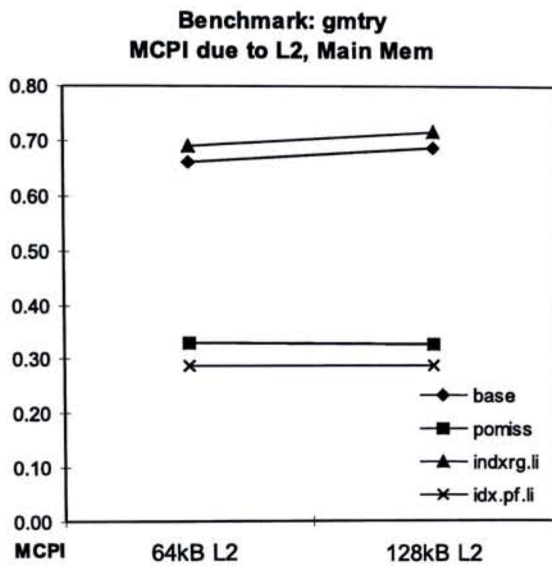
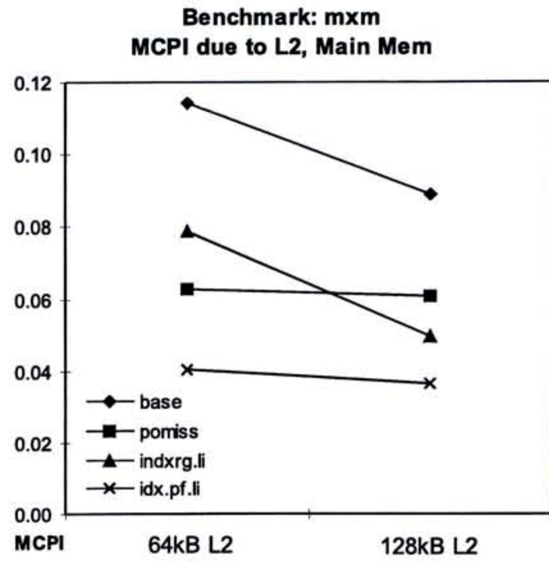


Figure 5.29: L2/Main Mem MCPI Comparison of Default Prefetch Scheme (cont.)





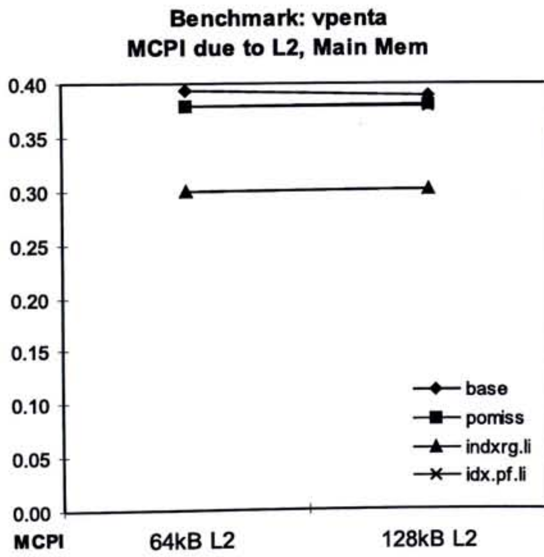


Figure 5.30: L2/Main Mem MCPI Comparison of Default Prefetch Scheme (cont.)

by using Line Concept, the reduction in MCPI is 1.6using Line Concept.

The main difference in PREFETCH and SETCAM is the instruction overhead to make a cache prefetch. The results show that when the instruction overhead can be reduced greatly, the performance of the software based prefetch algorithm will be much improved.

When comparing with hardware prefetch algorithms, we got the following table.

SIRPA family of hardware prefetch algorithms won all but the btrix and mxm benchmarks. The difference in reduction of MCPI in btrix benchmark is a mere 1.5%, and in mxm benchmark, 0.04%. However, in the cfft2d and gmtry benchmarks, the difference in performance of the SIRPA scheme and the SETCAM scheme is very sharp.

It is generally believe that software based prefetch algorithms will perform better than hardware prefetch algorithms, as the language compiler can take extensive code analysis on the executable code, and insert special instructions for cache prefetch accordingly. All cache prefetches will be accessed and there will be no cache pollution problem. If the instruction overhead can be minimized, software prefetch algorithm should be preferred to hardware ones, because hardware prefetch algorithms can only work with the executable code and data access pattern, which a lot of useful information contained in the source program may be lost. In the above table, we can observe that, with a highly accurate and efficient hardware prefetch algorithm, such as SIRPA, the performance can actually be better than a software based prefetch algorithm.

The even more important impact is that, software based prefetch algorithms always depend on the support of a specialized language compiler and hardware. Both the compiler and the processor has to support the special instructions inserted in the executable code before any benefit of the prefetch algorithm can be

seen. There will be no software compatibility with previous code. However, with SIRPA, there is no requirement in the compiler nor the processor support. The SIRPA scheme can be implemented as a separate function unit on the same chip as the processor, but the CPU has the freedom to use instruction set compatible with previous processors without SIRPA. Software compatibility can be assured but the memory latency can be improved.

Overall MCPI values for SIRPA with Line Concept scheme (a hardware cache prefetch algorithm) and other software based cache prefetch algorithms are shown in figures 5.31 and 5.32

For the seven kernels in NASA7, which software based cache prefetch algorithms can be tested together with other hardware prefetch algorithms, we found that the performance of SETCAM scheme is consistently superior than the PREFETCH instruction scheme. The overall MCPI produced by using SETCAM scheme is similar to the hardware prefetch algorithm like SIRPA, with the exception of cfft2d and gmtry kernels, which the SETCAM scheme was not so effective.

5.5 L2 Cache & Main Memory MCPI Comparison

MCPI due to second level cache and main memory is a calculated value. A set of simulations assuming an infinitely large second level cache is run. The results showed the timing information when all second level cache access resulted in cache hit. The results were deducted from the overall MCPI, and the reminders should be attributed to the timing requirement for the second level cache to main memory transfer.

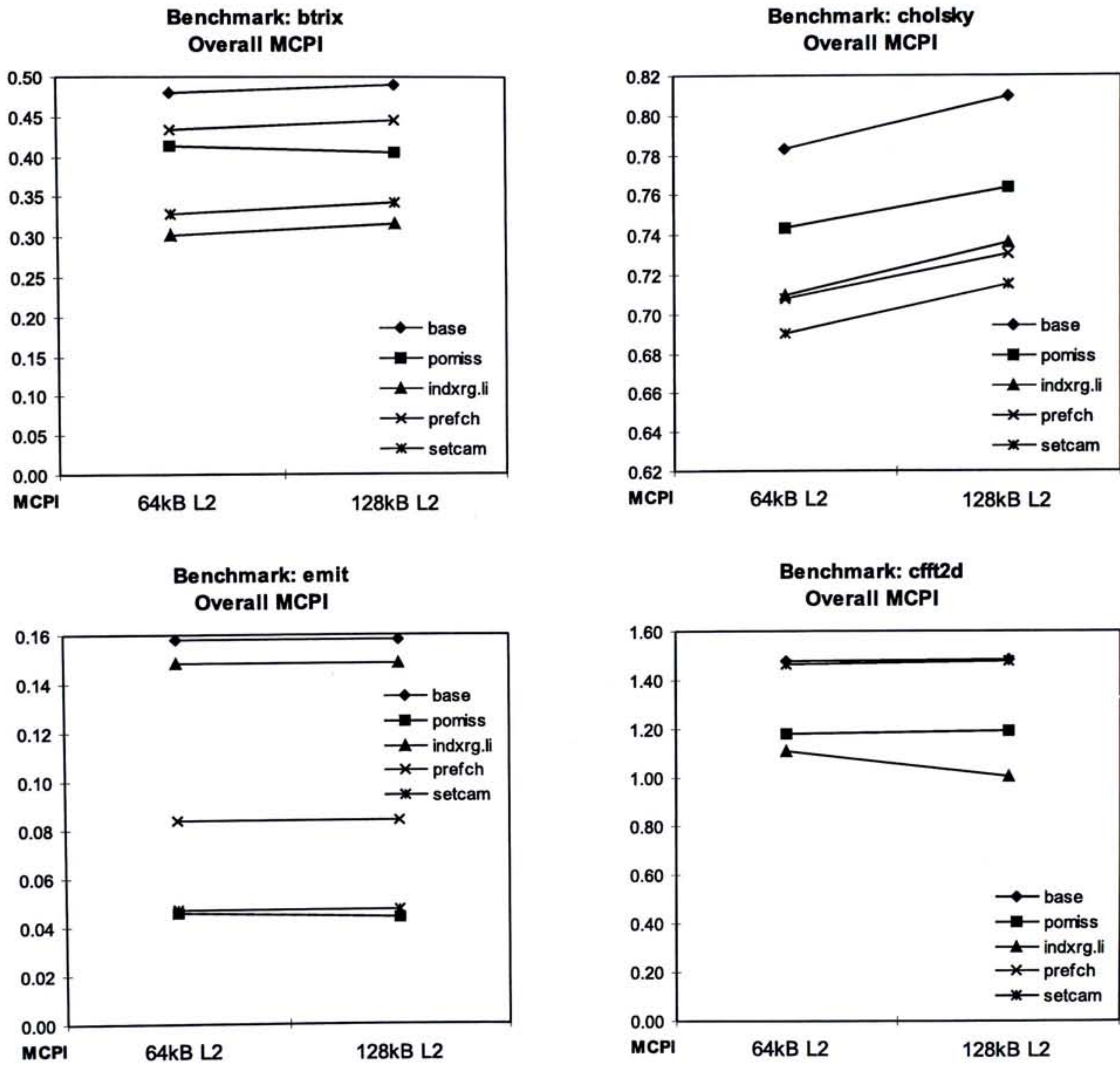


Figure 5.31: Overall MCPI Comparison by Software/Hardware Prefetch Algorithms

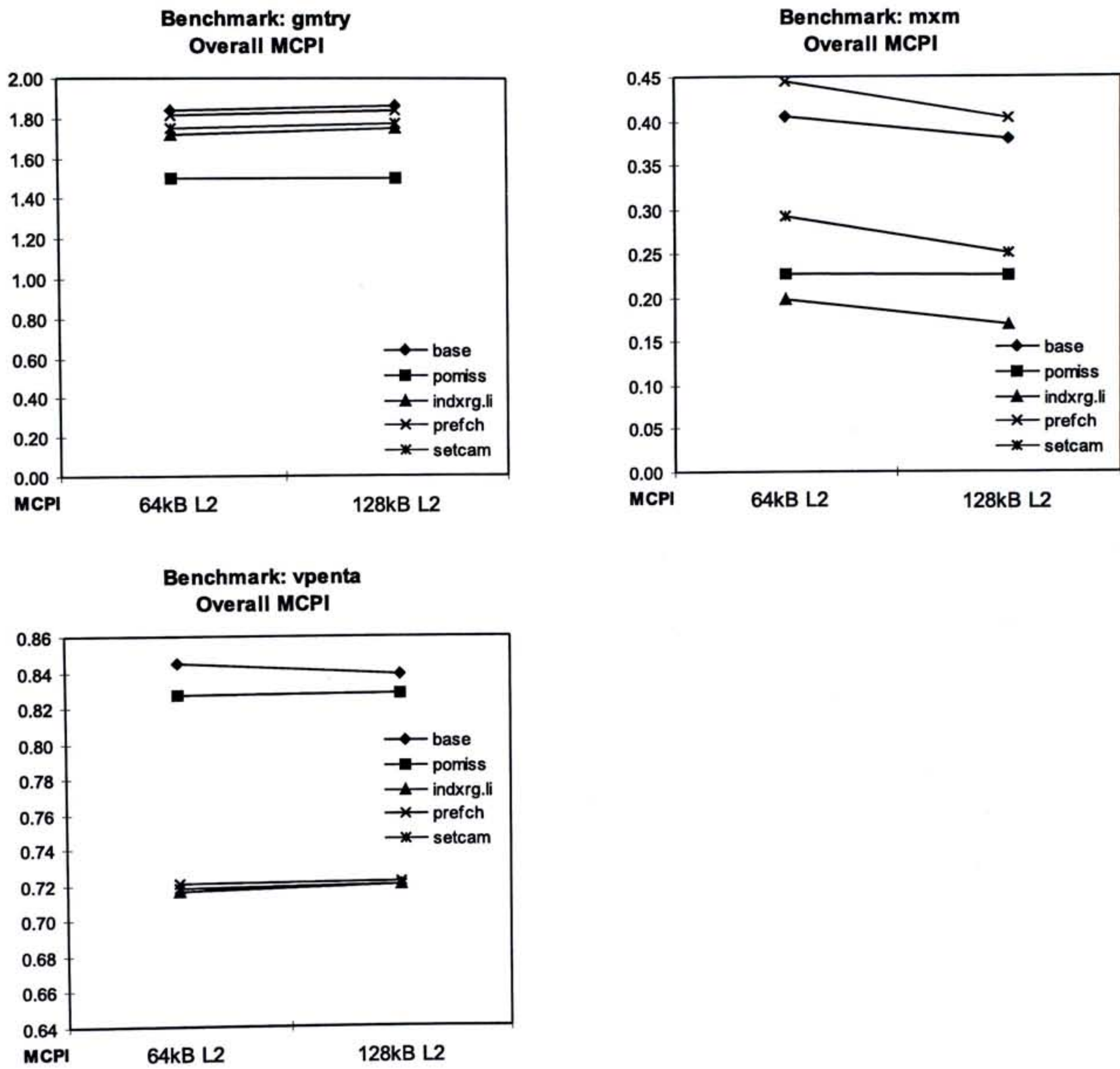


Figure 5.32: Overall MCPI Comparison by Software/Hardware Prefetch Algorithms (cont.)

5.5.1 Cache Size Effect

The charts in figures 5.33 – 5.35 are those plotted the second level cache size against the MCPI due to second level cache and main memory for each benchmark program. In figures 5.36 – 5.38, the reduction in MCPI due to second level cache and main memory for benchmark programs are shown. The results were similar to the overall MCPI comparison.

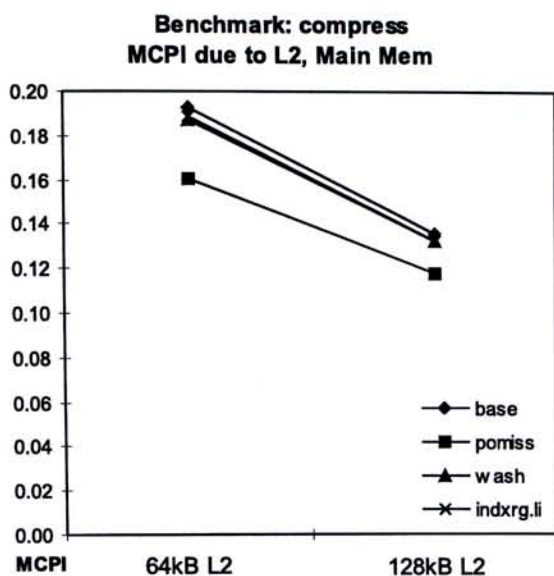
We observed large improvement in L2, Main Memory MCPI in compress, spice2, su2cor and mxm benchmarks. In espresso, wave5, xlisp, nasa7, cfft2d, emit, gmtry, and vpenta benchmarks, the performance in L2, Main Memory MCPI is leveled off by increasing the L2 cache size.

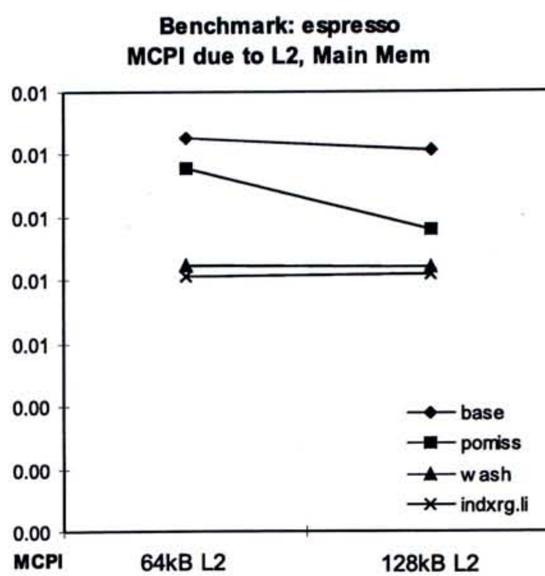
There were anomalies in the cache size effect, in the benchmark of tomcatv, wave5, btrix and cholsky, the L2 cache, Main Memory MCPI actually increases as the cache size increases. The reasons for the anomalies are similar to those presented in the section discussing overall MCPI. One more suggestion may due to the fact that with a larger second level cache, the number of transfer from the main memory to the cache is smaller, compare with a smaller second level cache. The L2, Main Memory MCPI recorded may have a higher portion of cache cold start transfer in the 128k Bytes second level cache.

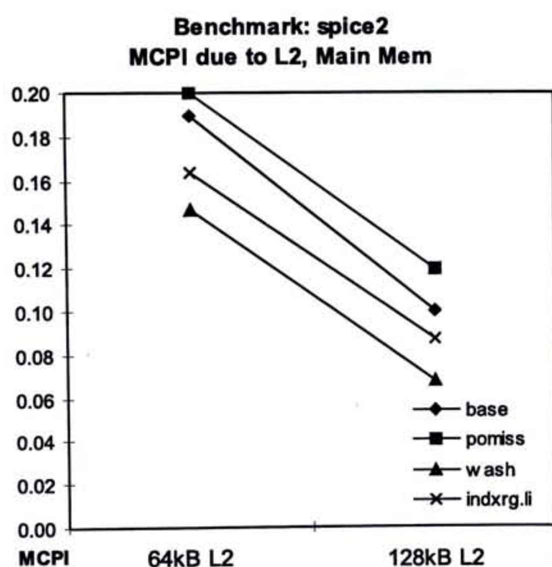
5.5.2 Cache Block Size Effect

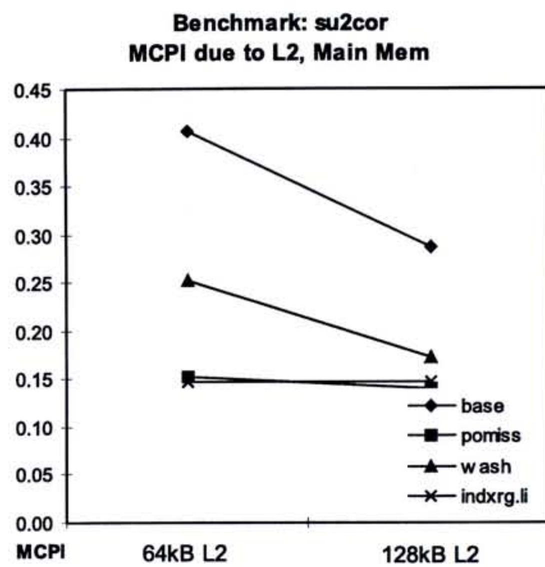
The charts in figures 5.39 – 5.41 are those plotted the second level cache block size against the L2, Main Memory MCPI for each benchmark program. In figures **??** – **??**, the reduction min MCPI due to L2 and Main Memory are plotted against the change in cache block size.

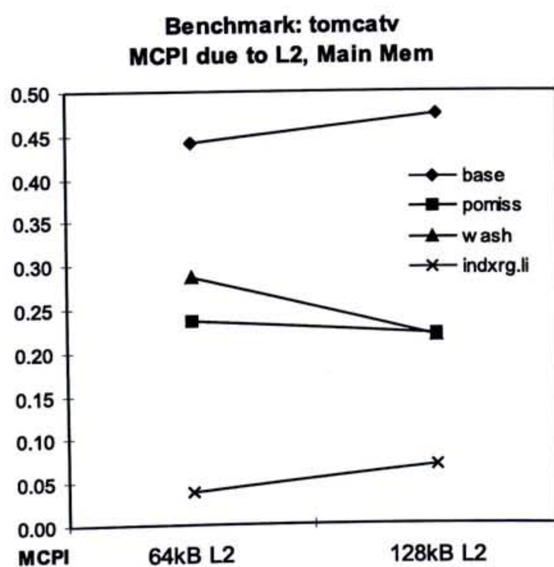
The results were consistent with the overall MCPI charts. All benchmark programs showed an increase in L2, Main Memory MCPI when the second level cache block increases from 32 Bytes block to 64 Bytes block, with the only exception











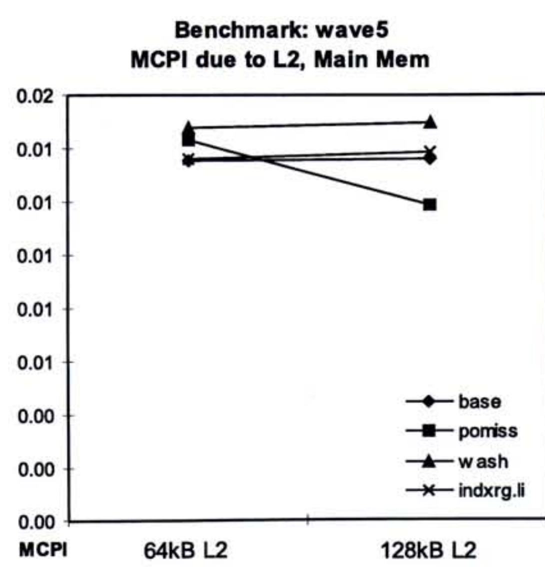
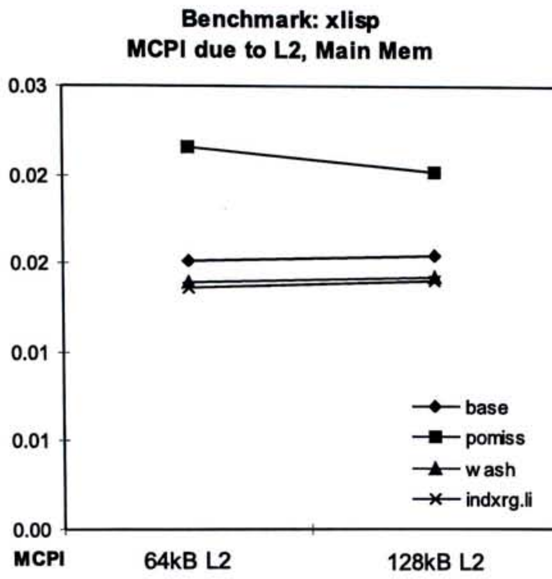
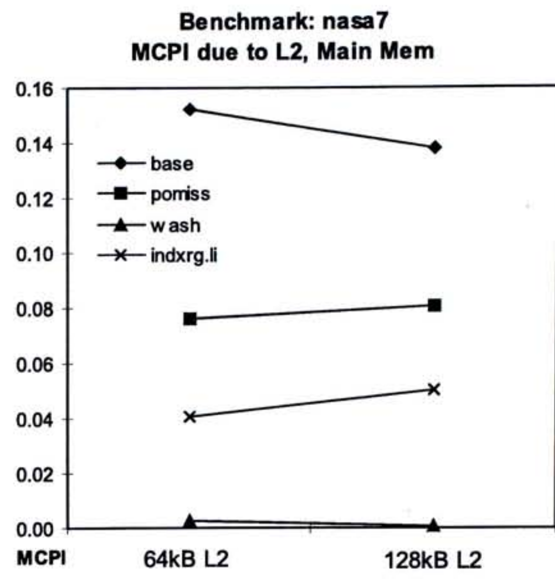
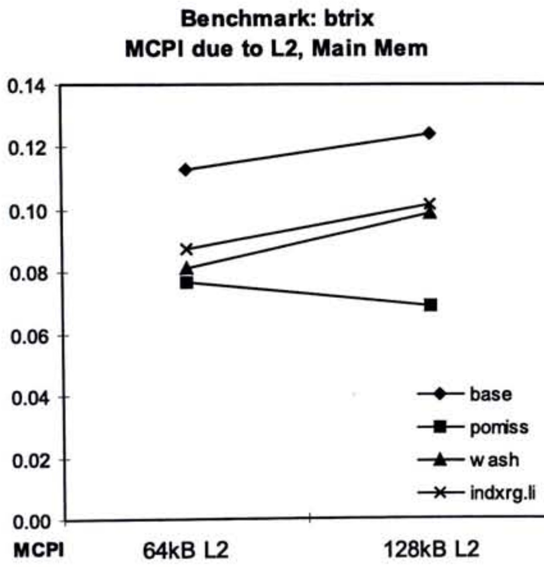
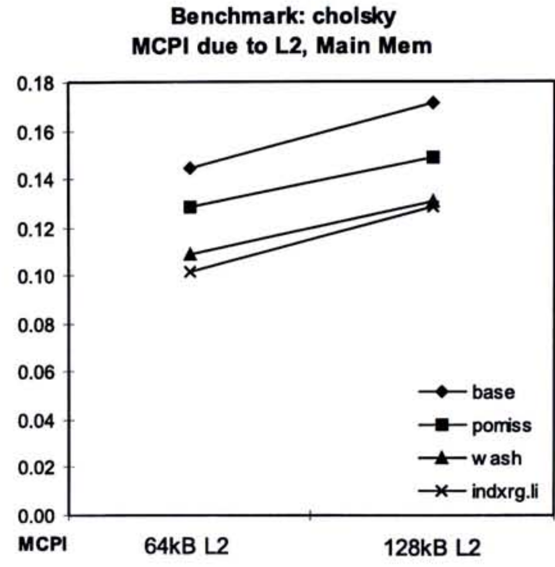


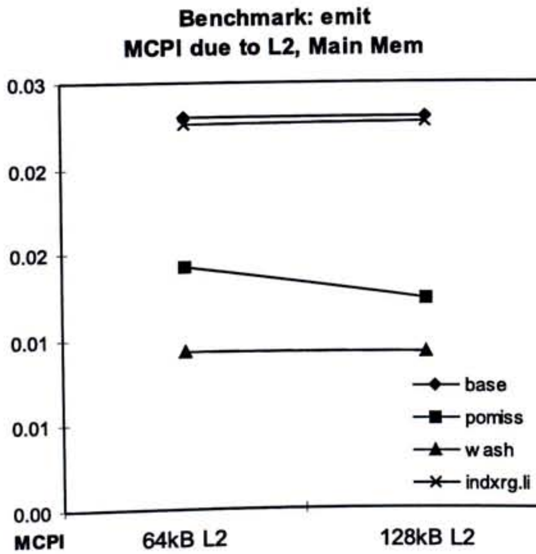
Figure 5.33: L2/Mem MCPI comparison by cache size











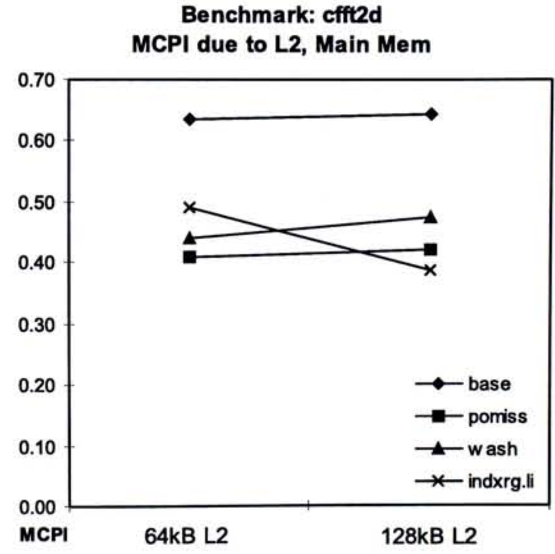
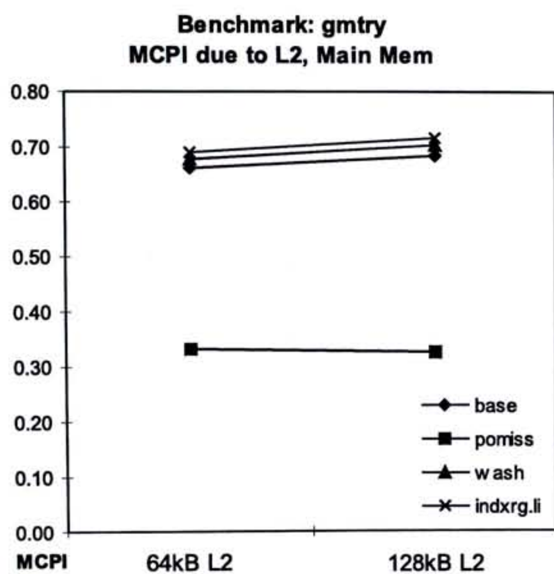
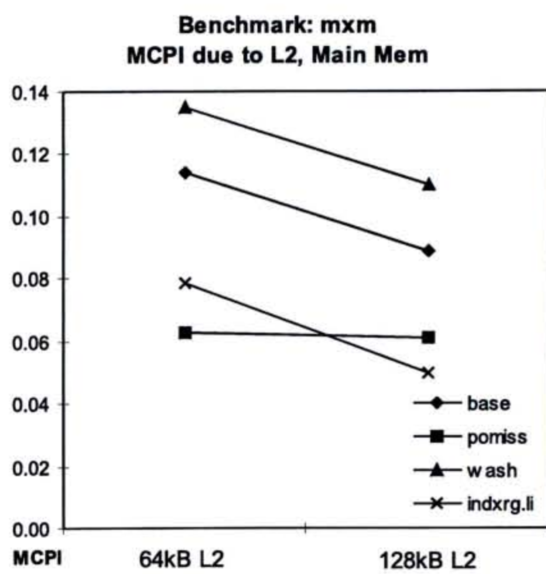


Figure 5.34: L2/Mem MCPI comparison by cache size (cont.)





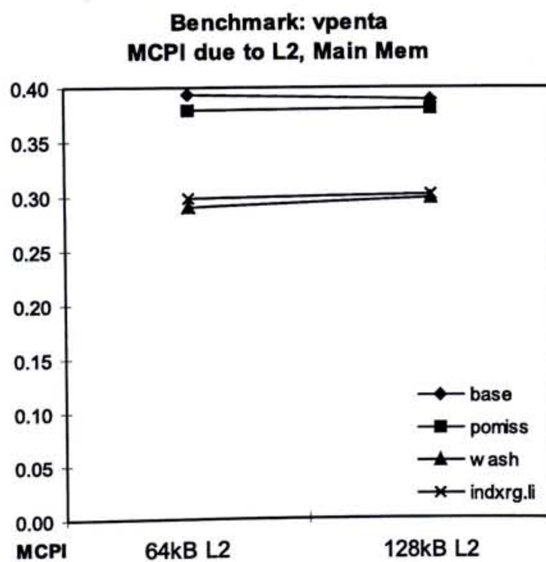
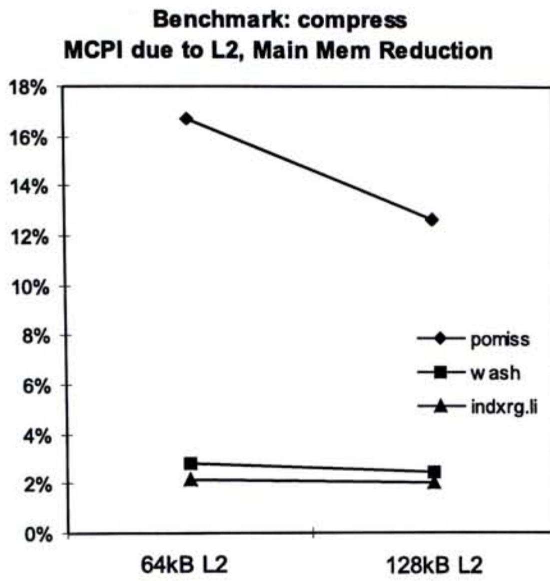
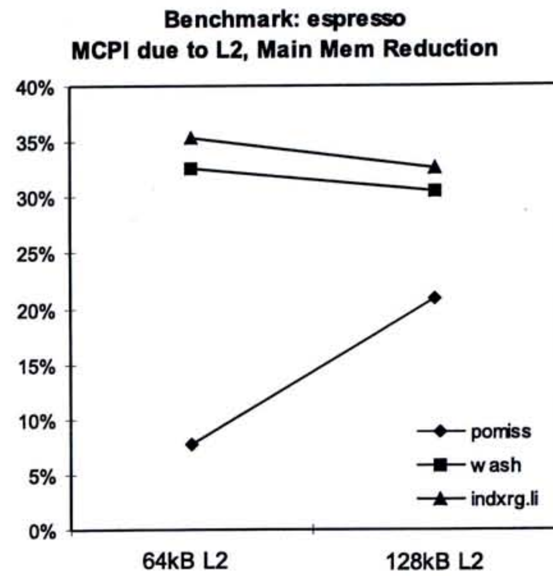
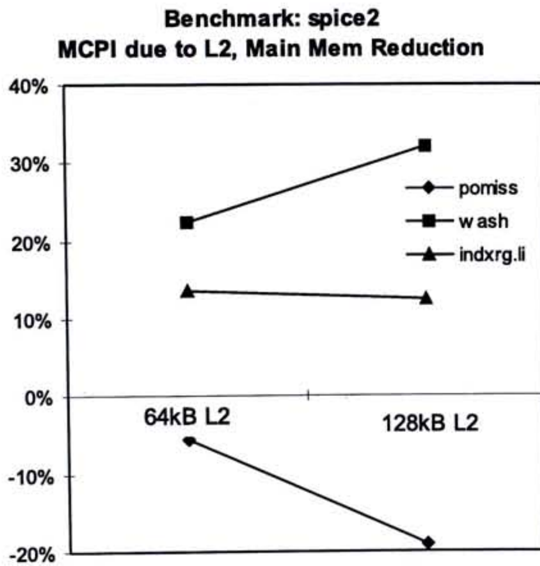
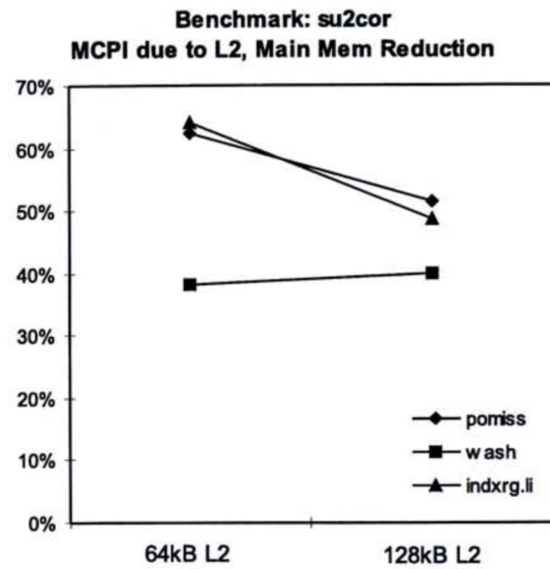


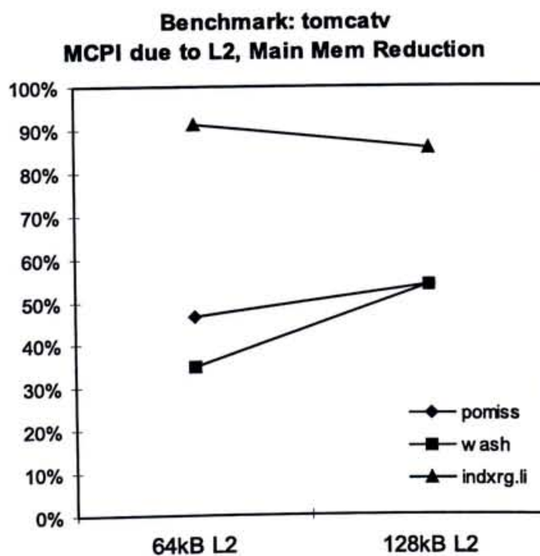
Figure 5.35: L2/Mem MCPI comparison by cache size (cont.)











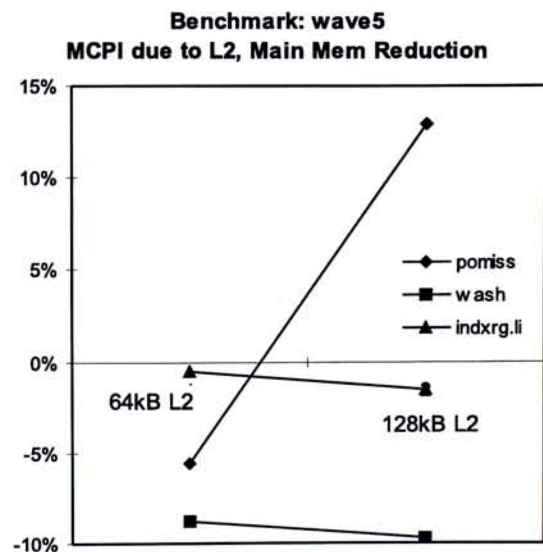
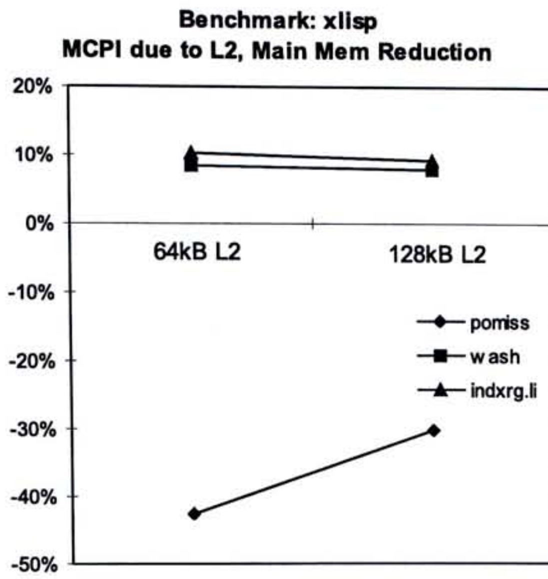
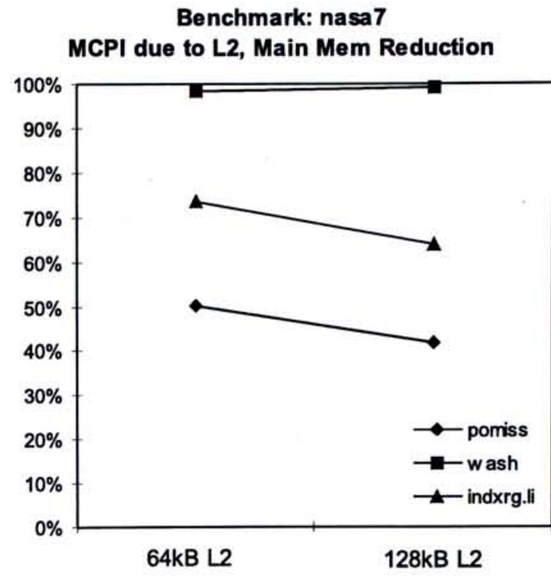
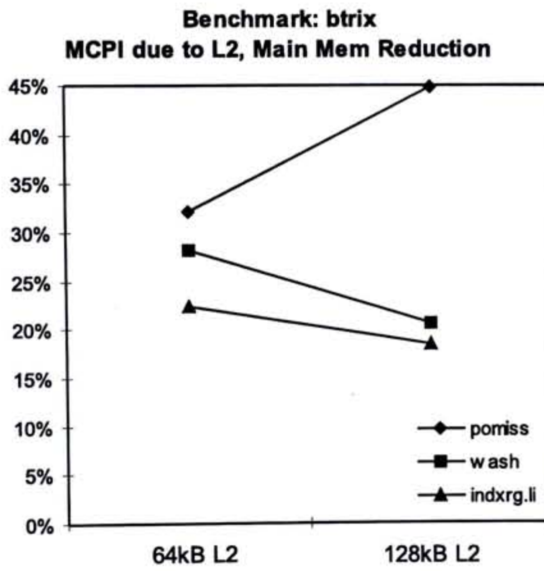
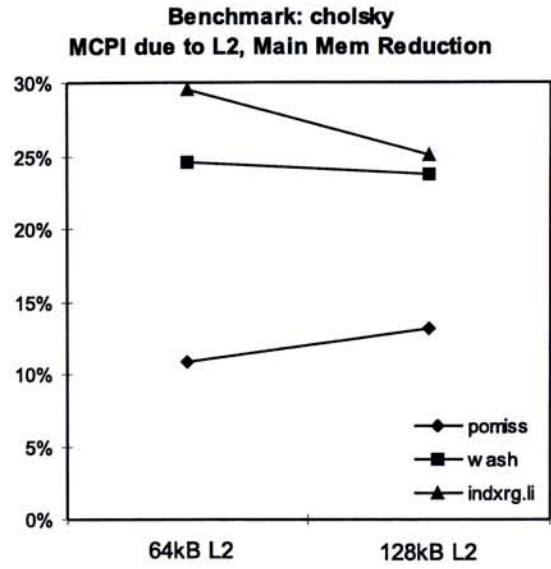


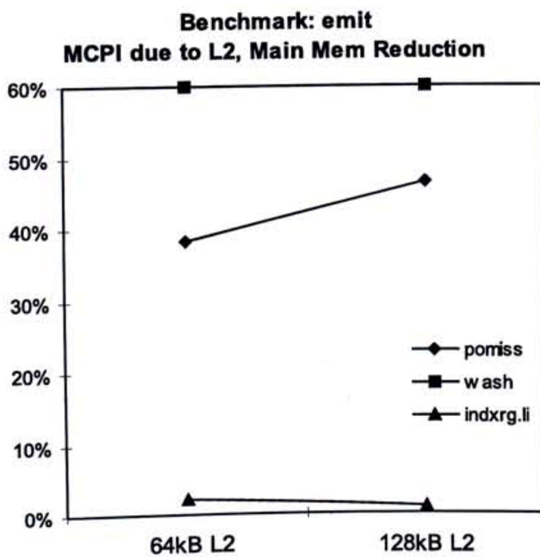
Figure 5.36: L2/Mem MCPI Reduction comparison by cache size











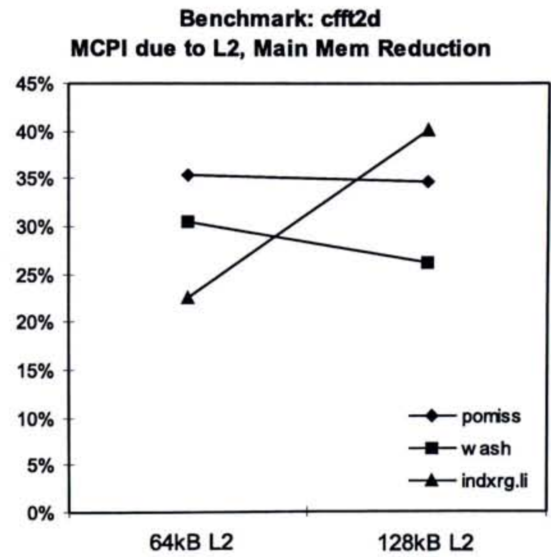
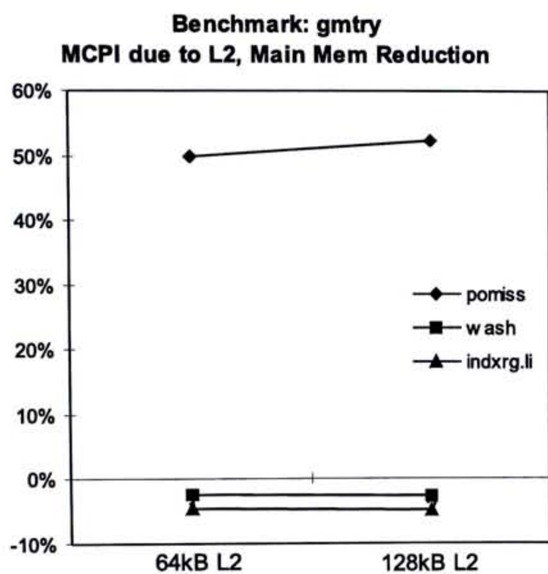
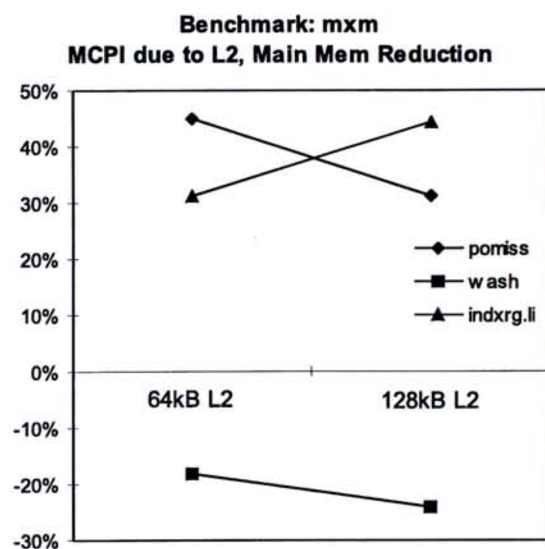


Figure 5.37: L2/Mem MCPI Reduction comparison by cache size (cont.)





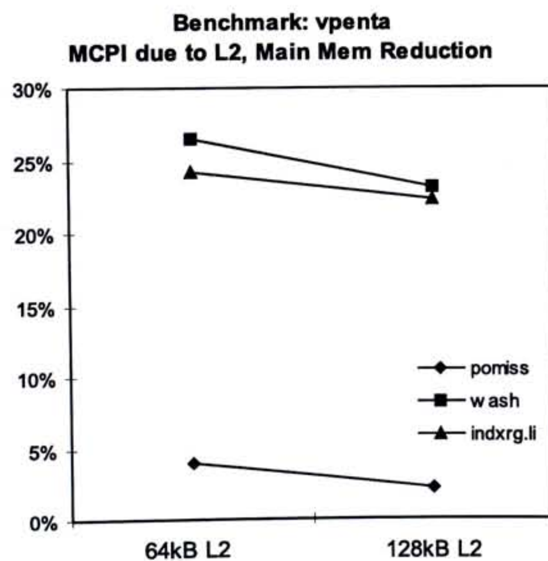
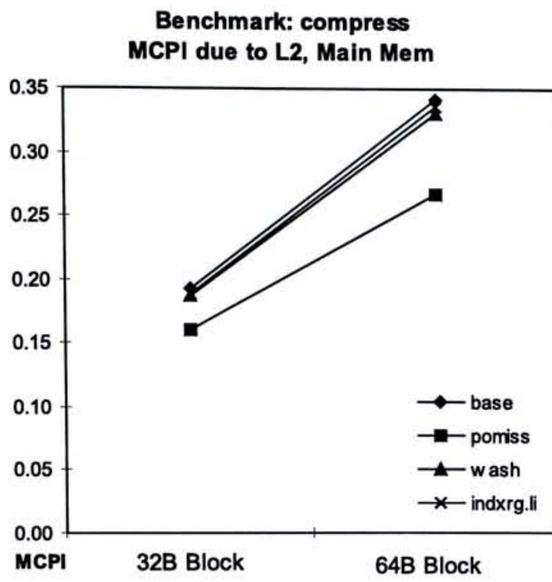
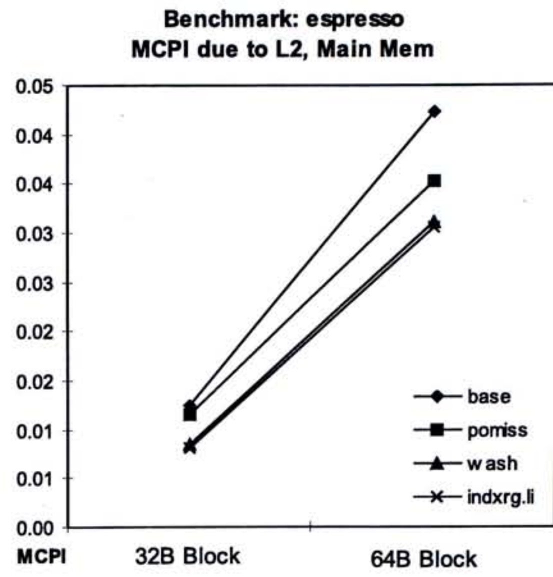
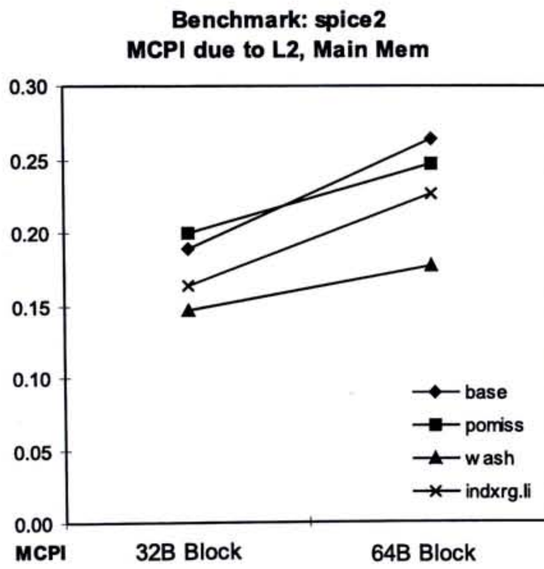
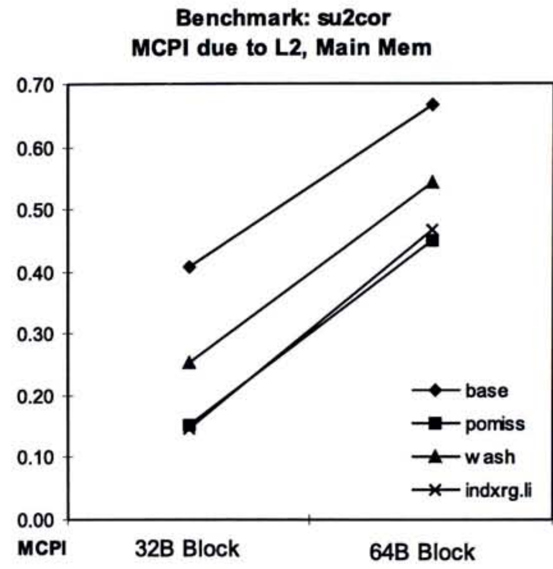


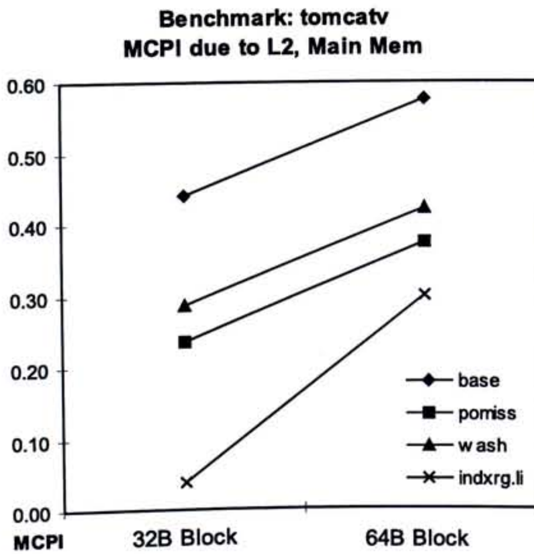
Figure 5.38: L2/Mem MCPI Reduction comparison by cache size (cont.)











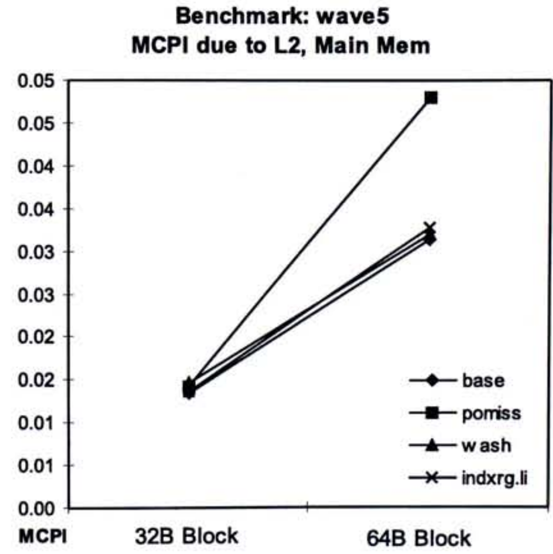
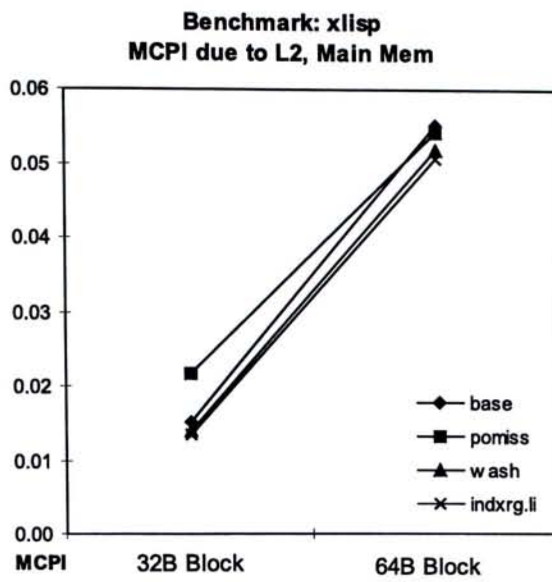
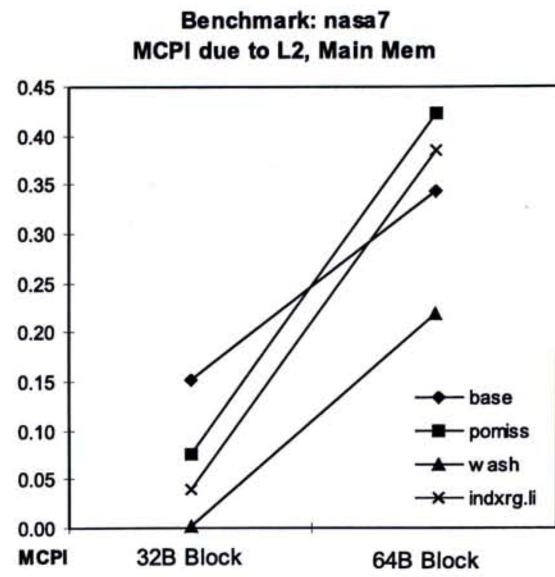
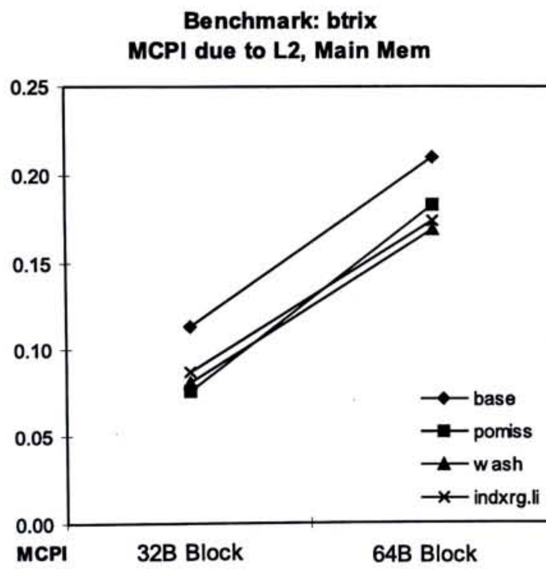
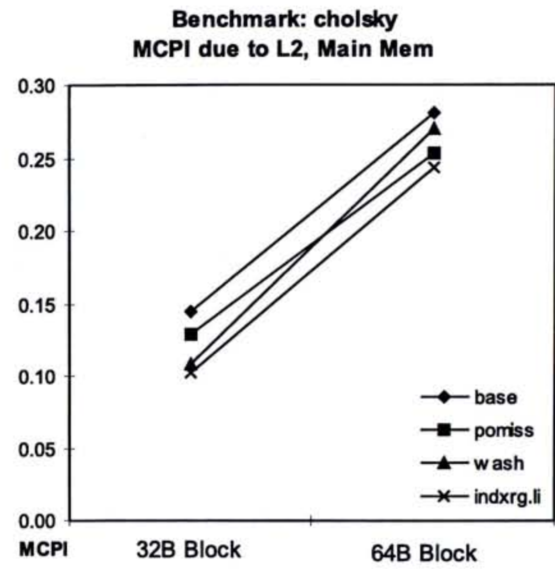


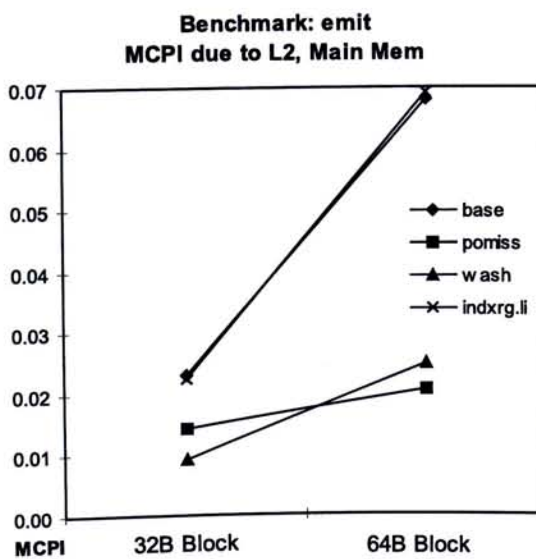
Figure 5.39: L2/Mem MCPI comparison by block size











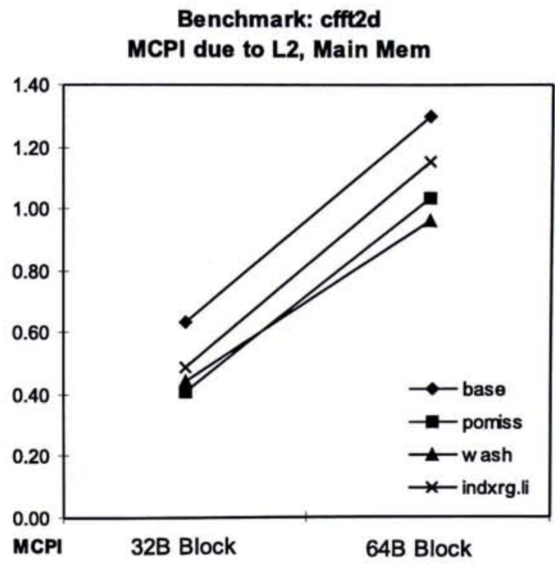
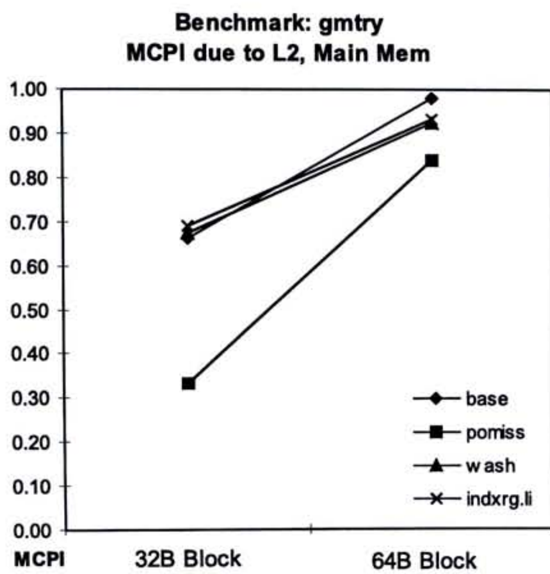
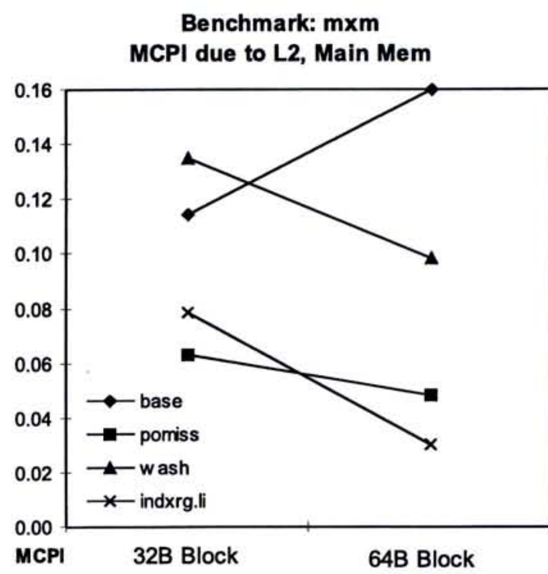


Figure 5.40: L2/Mem MCPI comparison by block size (cont.)





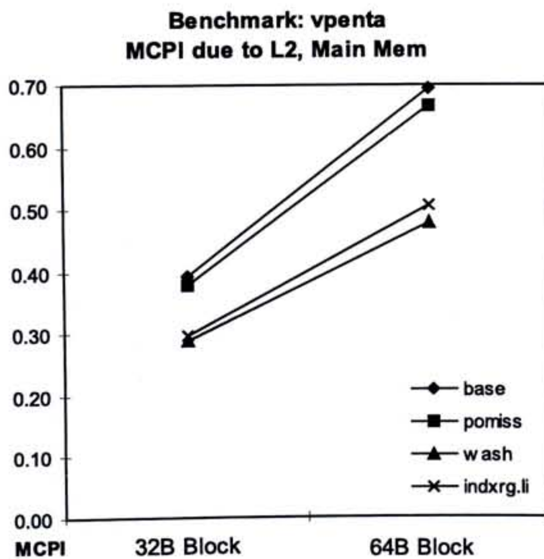
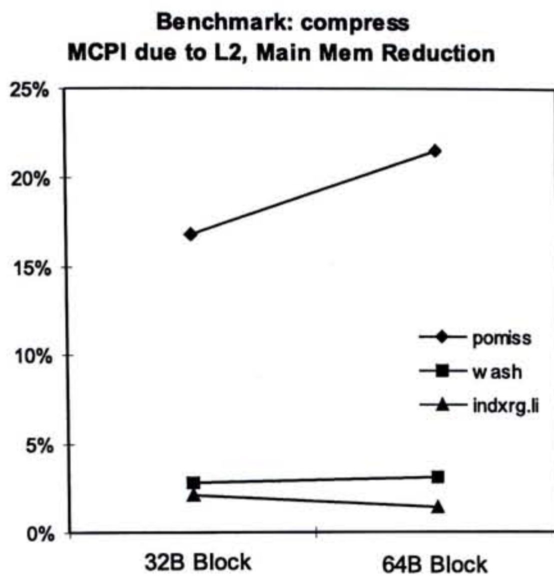
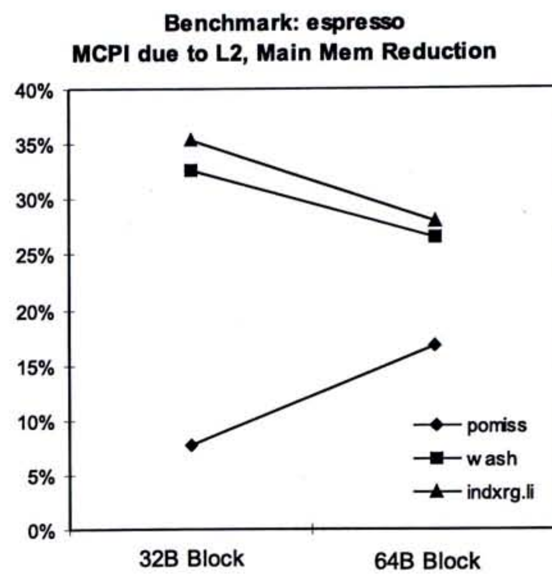
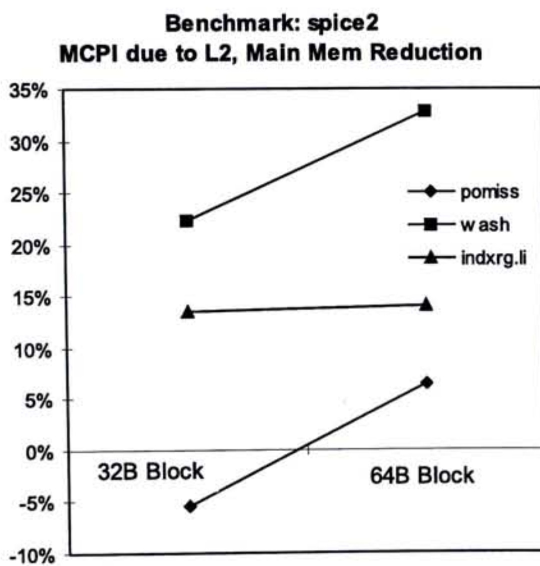
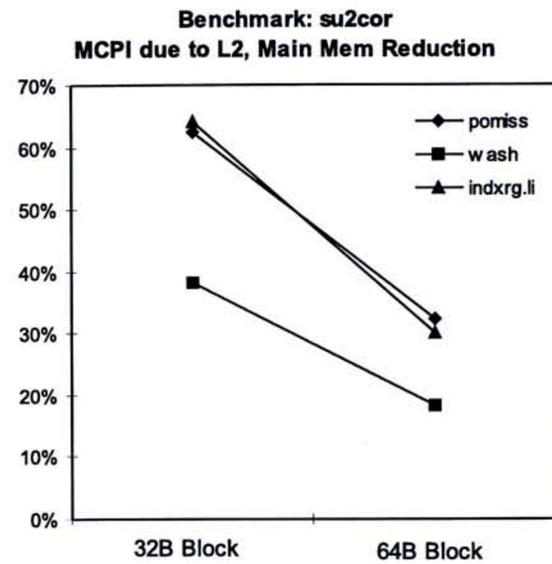


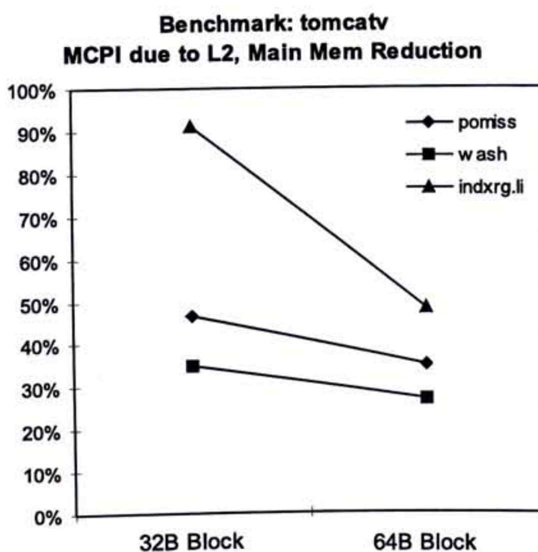
Figure 5.41: L2/Mem MCPI comparison by block size (cont.)











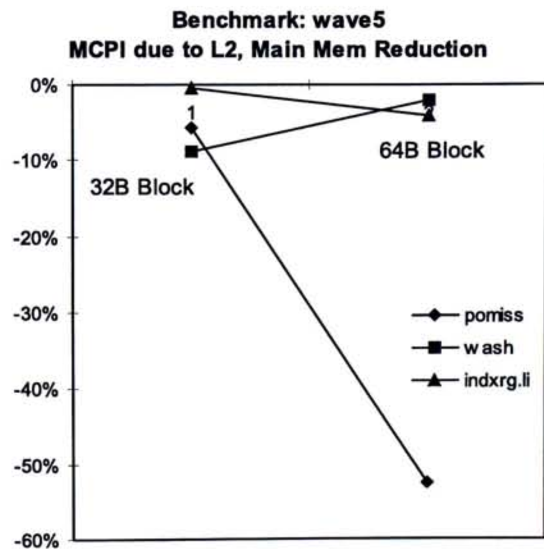
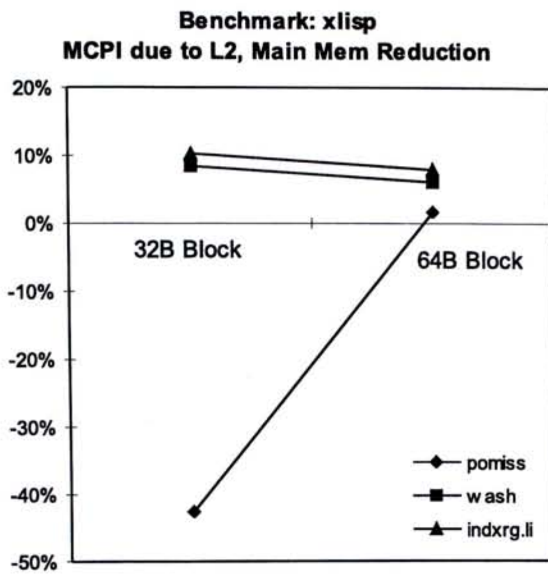
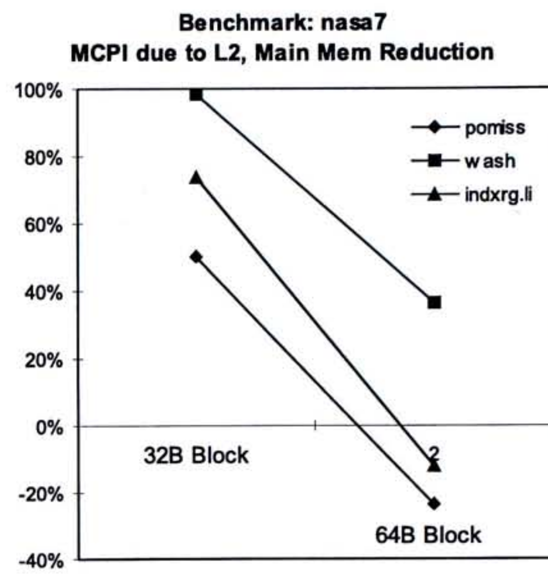
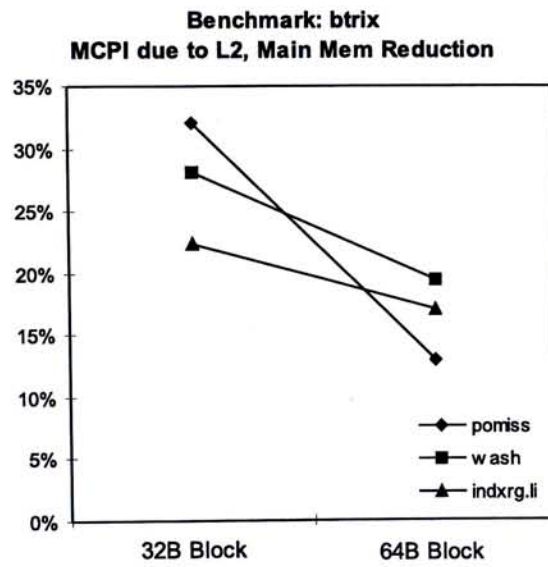
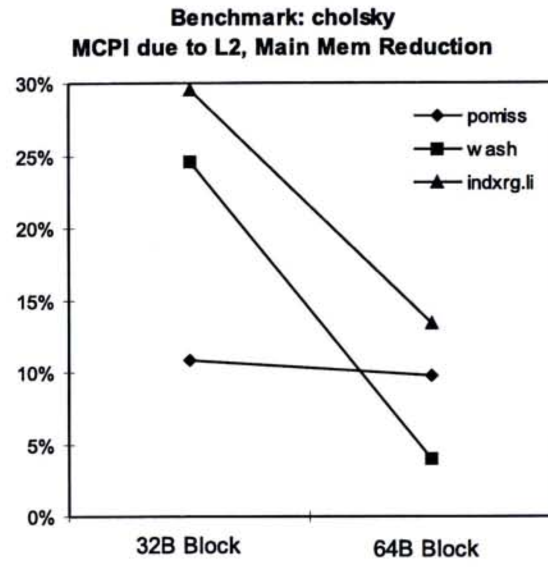


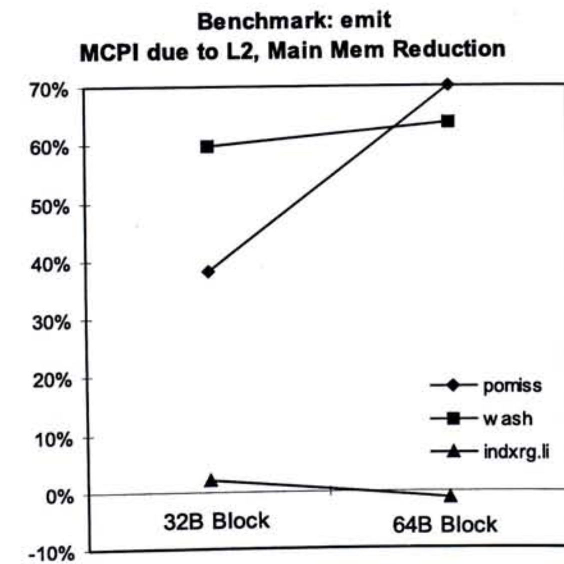
Figure 5.42: L2/Mem MCPI Reduction comparison by block size











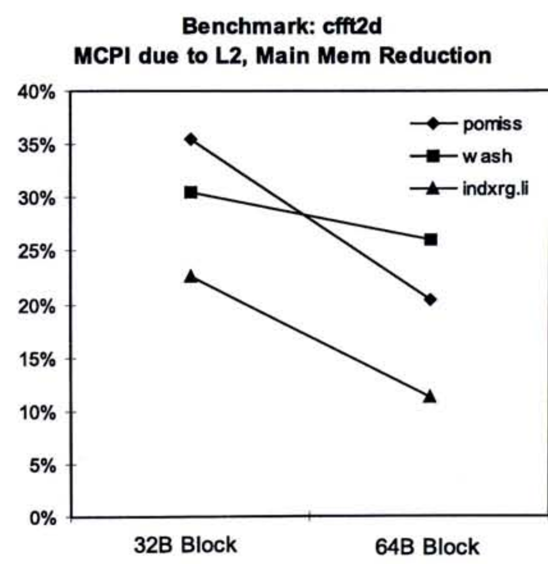
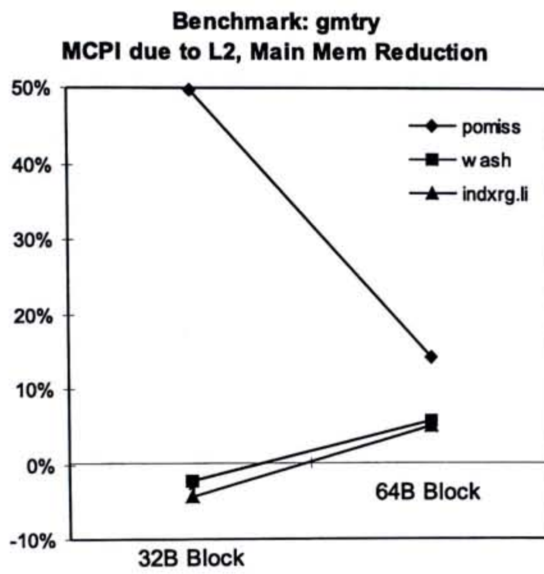
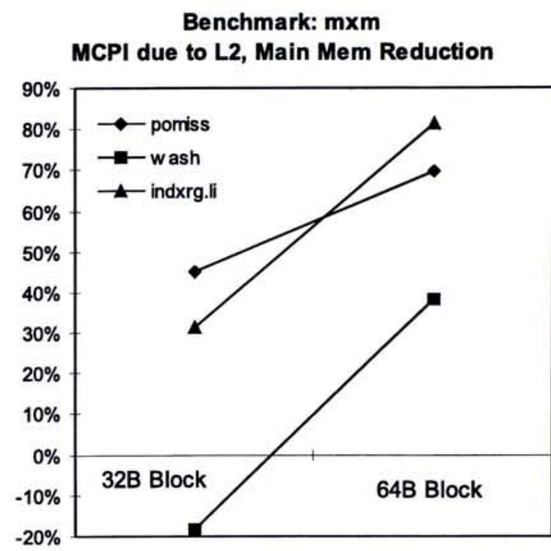


Figure 5.43: L2/Mem MCPI Reduction comparison by block size (cont.)





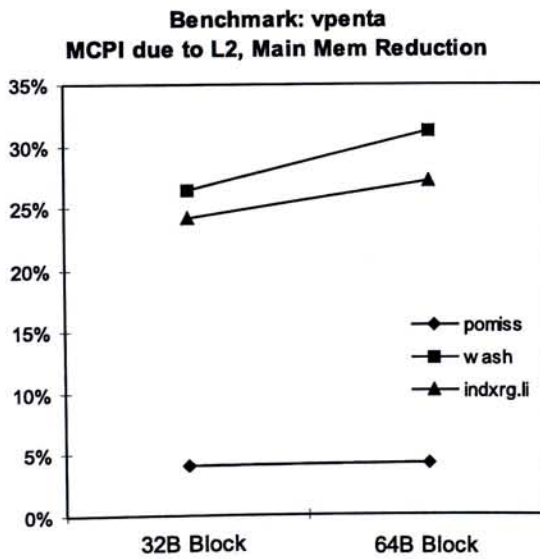


Figure 5.44: L2/Mem MCPI Reduction comparison by block size (cont.)

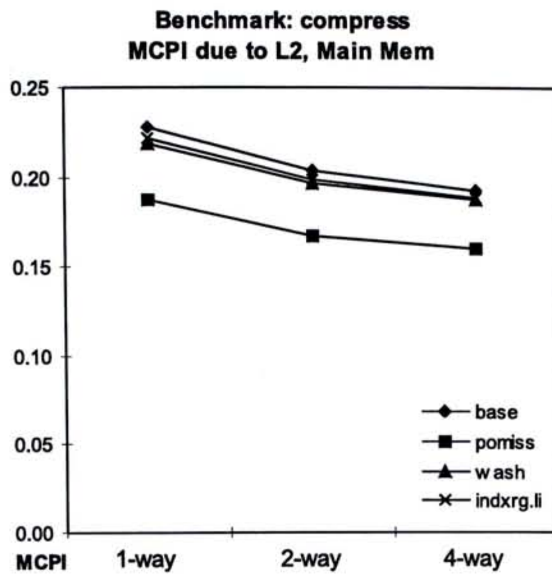
of mxm kernel in NASA7.

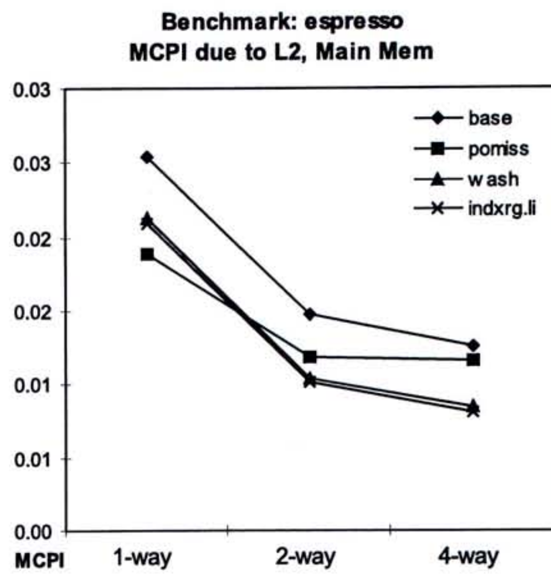
The reasons for the decrease in performance are similar to those presented in the previous section discussing overall MCPI. The increase in L2, Main Memory MCPI was even sharper when compare with the overall MCPI charts. It showed that the first level cache, which was an invariant part in the simulations, already shielded off the majority of memory accesses, and the bad effect of using a larger cache block size was contributed at large by the second level cache to main memory transfers.

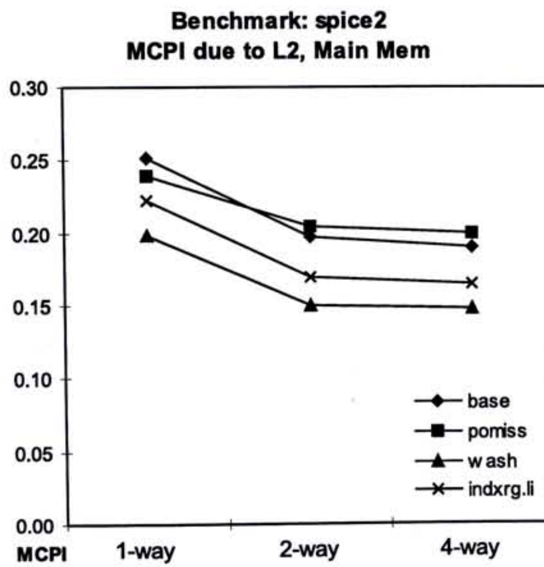
5.5.3 Set Associativity Effect

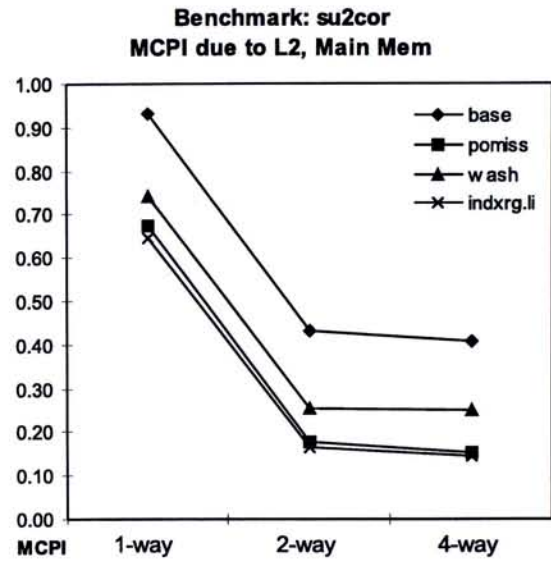
Charts with the set associativity values used in the second level cache plotted against the L2, Main Memory MCPI for each benchmark program are shown in the figures 5.45 – 5.47. L2, Main Memory MCPI reduction against the change in L2 cache set associativity is plotted in figures ??l2:reduction:set:1 – 5.50.

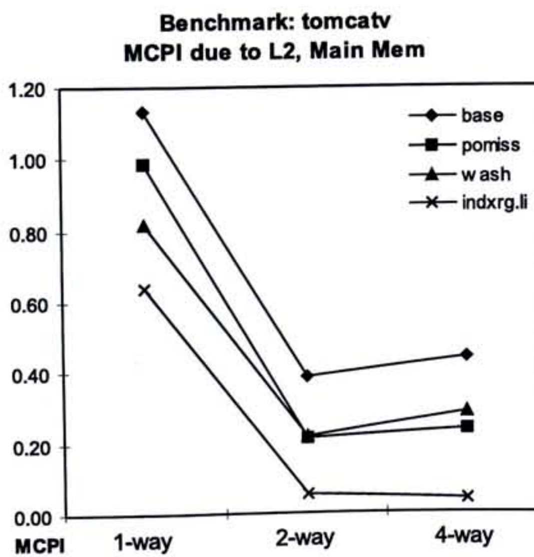
The results are in general an amplified version of the charts that presented in the overall MCPI section. The effect of using a higher set associativity is more important in improving the L2, Main Memory MCPI. It is a proof that the simulations done in this thesis gave a more complete view on a real computer system with a multiple hierarchy of memory levels. A set associative second level cache is inevitable in the design. For most of the benchmarks tested, there were a sharp improvement in L2, Main Memory MCPI when the set associativity was increased from 1 to 2. The further improvement by increasing the set associativity from 2 to 4 depends on different programs.











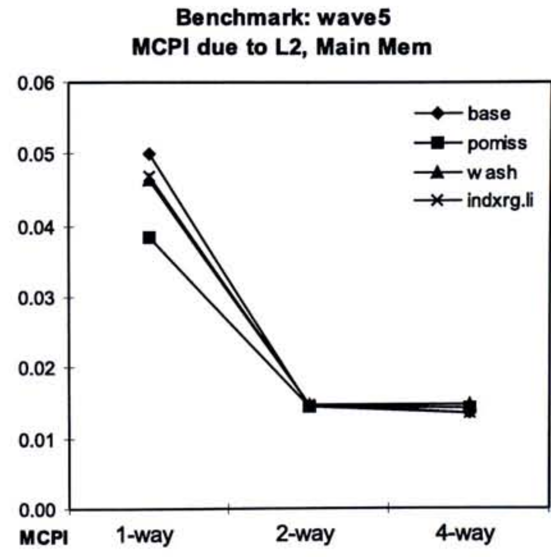
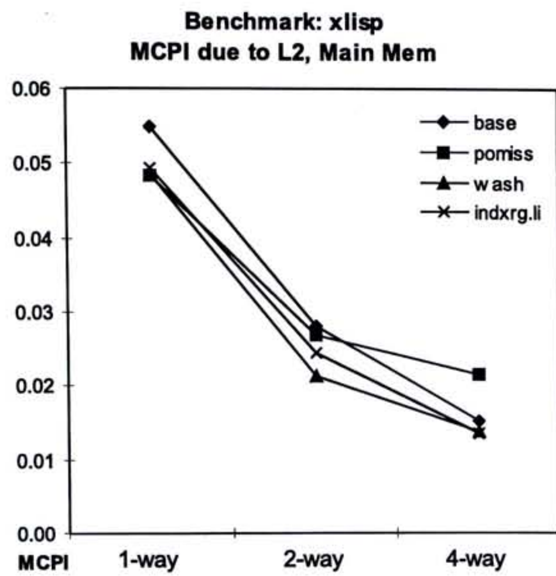
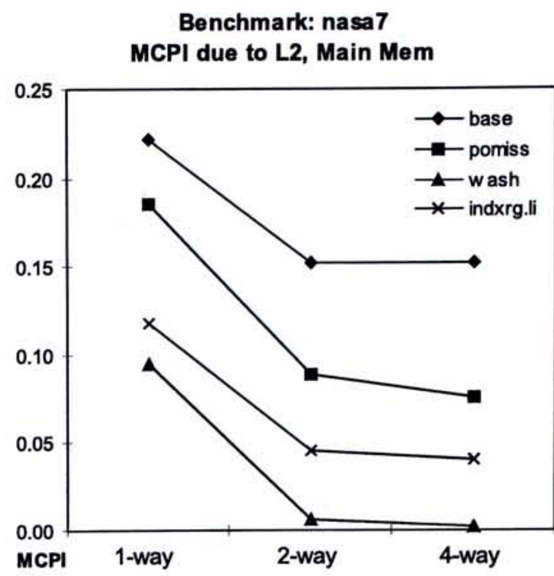
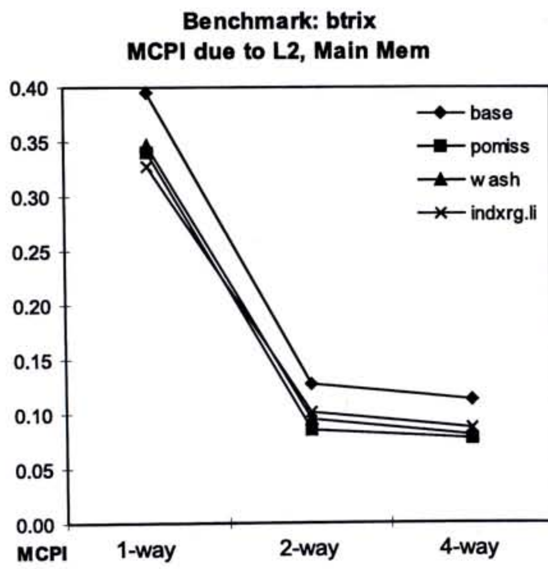
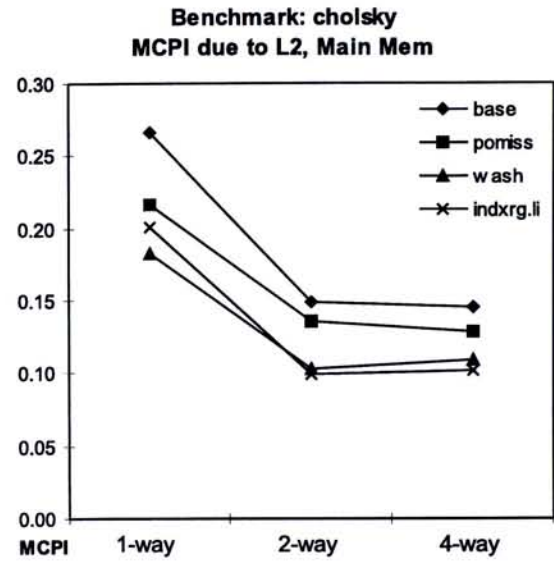


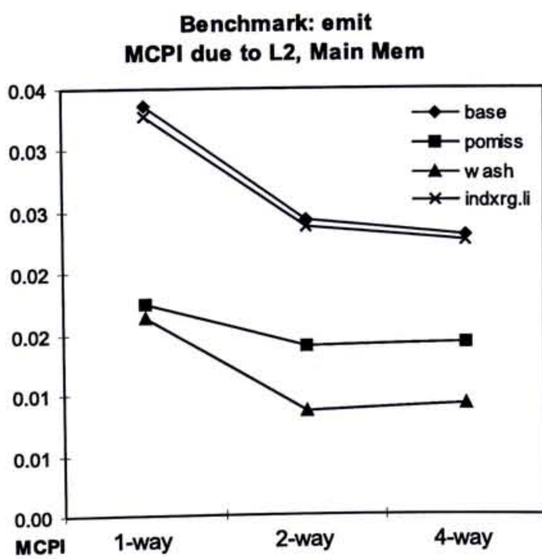
Figure 5.45: L2/Mem MCPI comparison by associativity











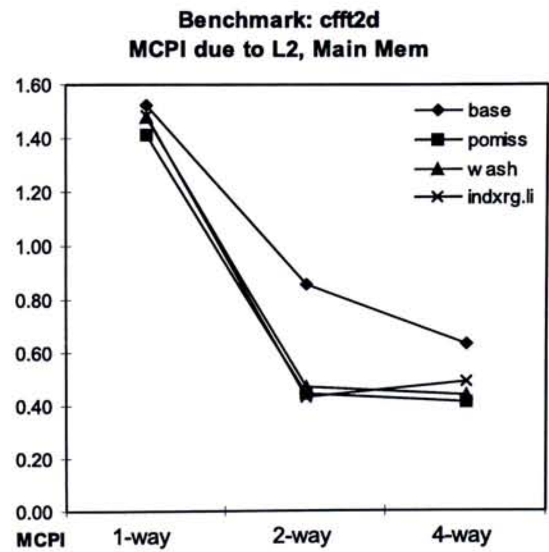
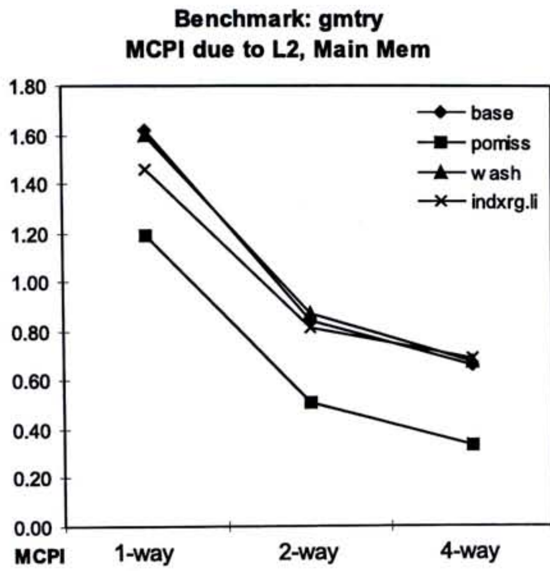
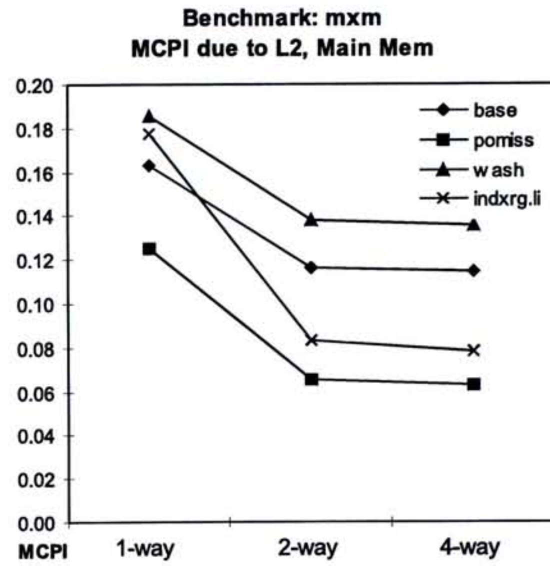


Figure 5.46: L2/Mem MCPI comparison by associativity (cont.)





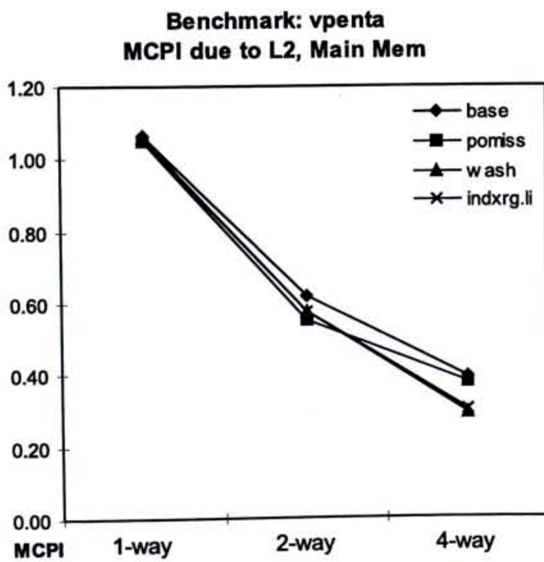
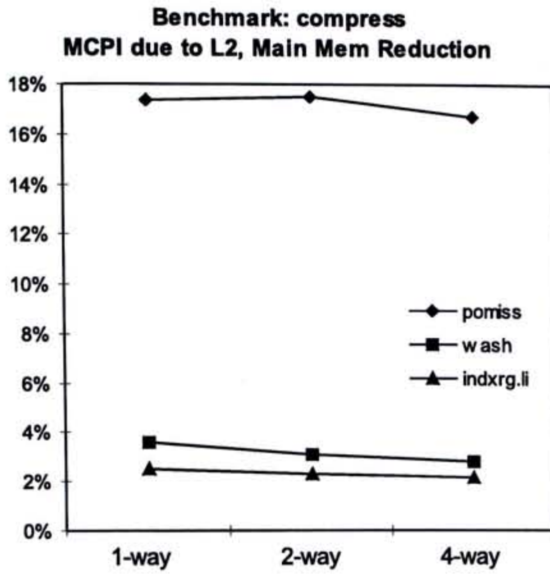
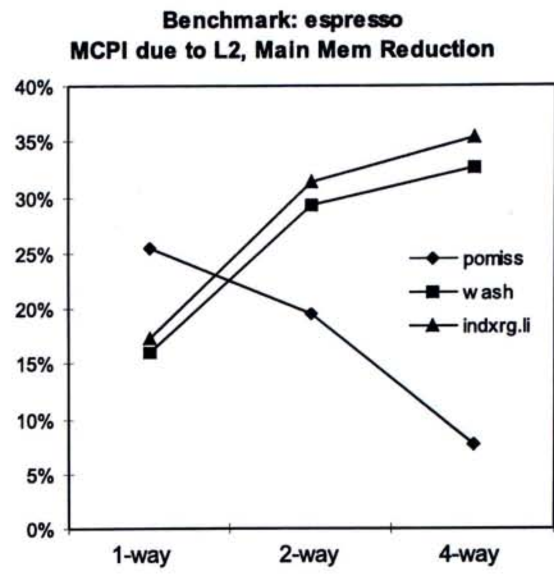
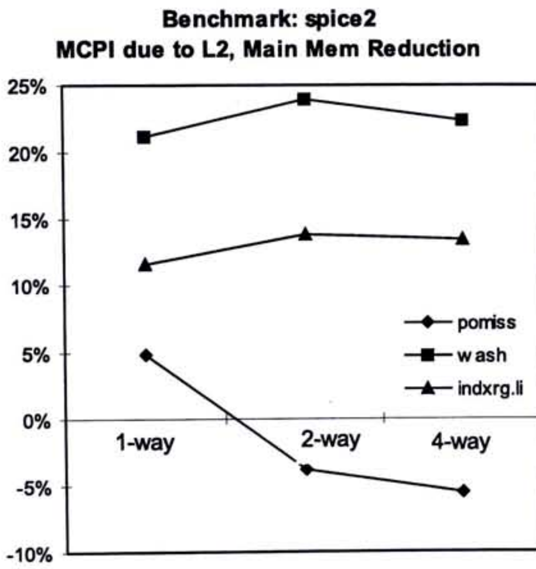
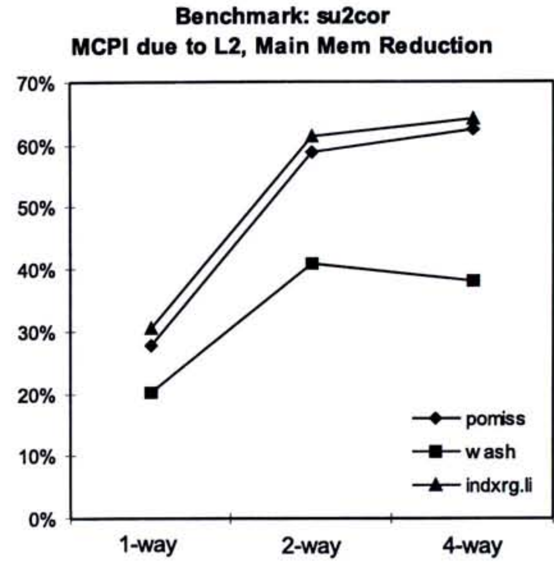


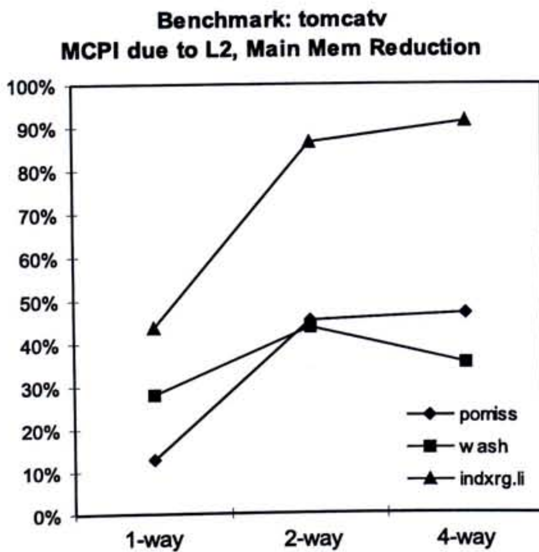
Figure 5.47: L2/Mem MCPI comparison by associativity (cont.)











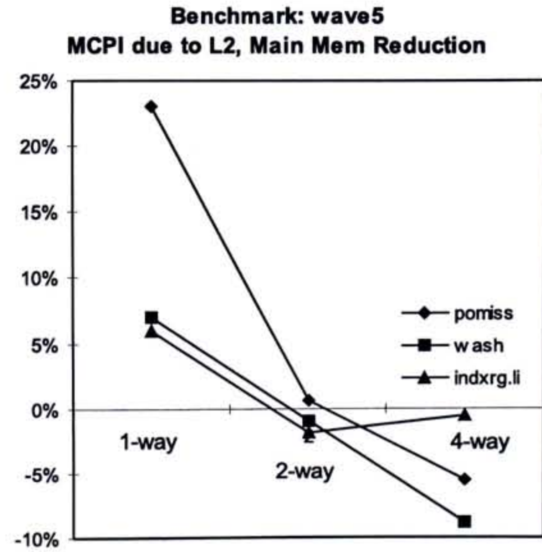
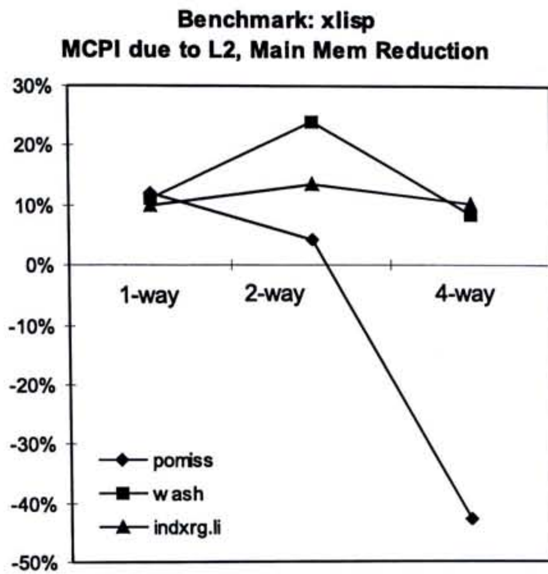
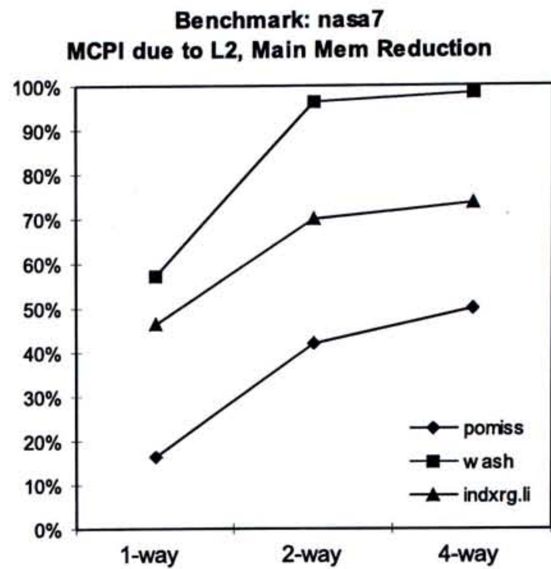
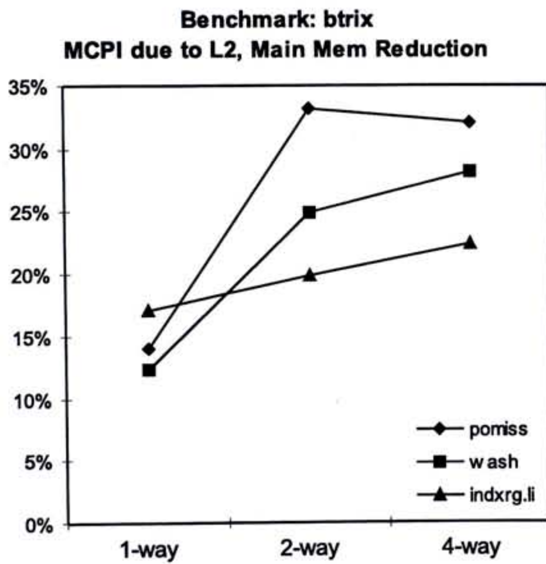
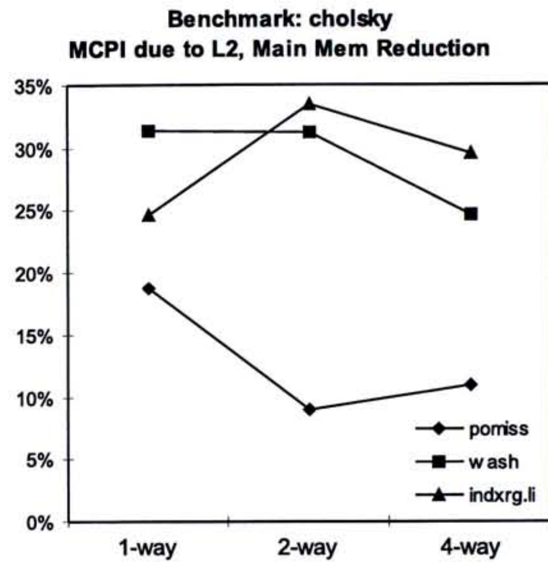


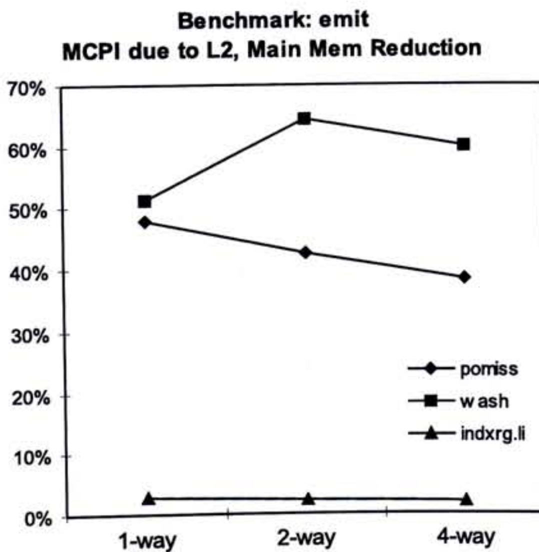
Figure 5.48: L2/Mem MCPI Reduction comparison by associativity











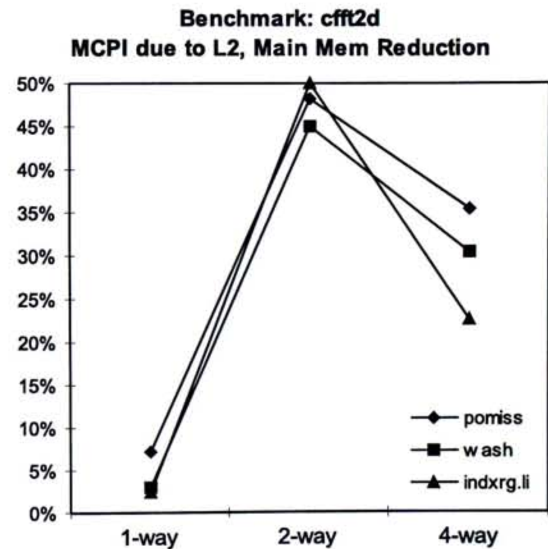
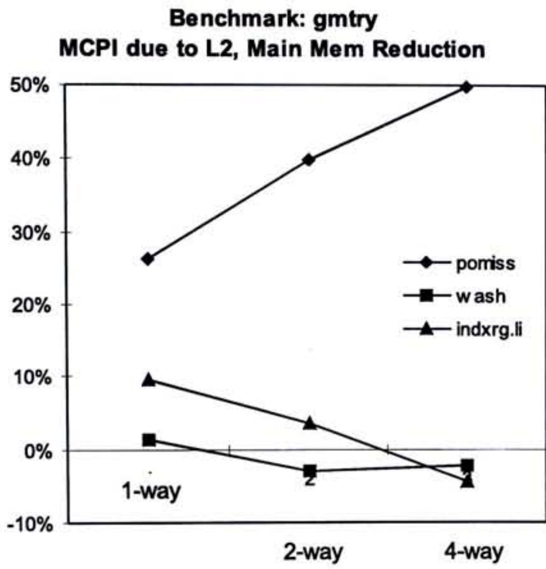
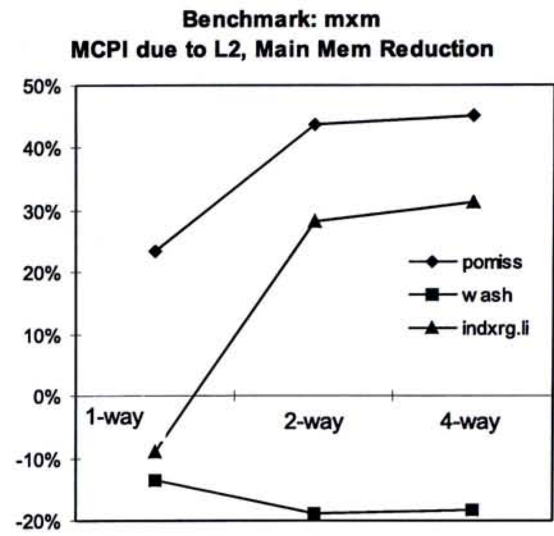


Figure 5.49: L2/Mem MCPI Reduction comparison by associativity (cont.)





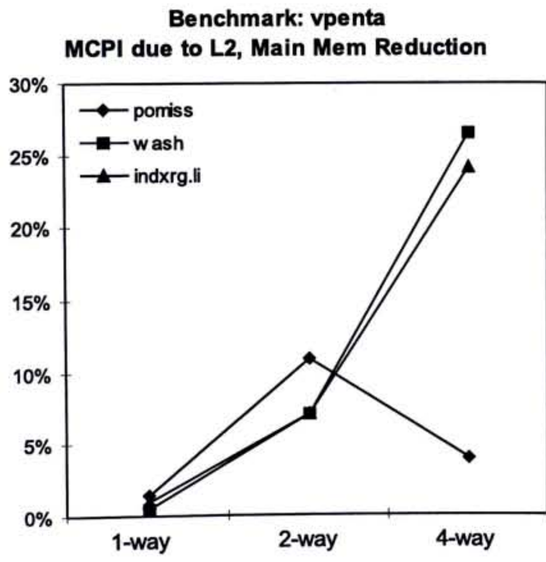


Figure 5.50: L2/Mem MCPI Reduction comparison by associativity (cont.)

Table 5.2: Best Overall MCPI by using SIRPA & RPT family Algorithms

Benchmark	Best MCPI by SIRPA family	Best MCPI by Wash. RPT family	Ratio
compress	0.24612	0.24979	1.0
espresso	0.05145	0.05064	(1.0)
spice2	0.26036	0.22352	(1.2)
su2cor	1.08390	1.11053	1.0
tomcatv	0.68474	0.79914	1.2
wave5	0.06387	0.06362	(1.0)
xlisp	0.10620	0.10495	(1.0)
nasa7	0.35351	0.38674	1.1
btrix	0.30295	0.28570	(1.1)
cholsky	0.67255	0.66393	(1.1)
emit	0.04198	0.04521	1.1
cfft2d	1.00582	1.06920	1.1
gmtry	1.45718	1.72924	1.2
mxm	0.14724	0.24979	1.7
vpenta	0.71672	0.70720	(1.0)

Table 5.3: Best Overall MCPI by using PFONMISS/SIRPA/RPT Algorithms

Benchmark	Best MCPI by PFONMISS	Best MCPI by SIRPA family	Best MCPI by RPT family
compress	0.24723	0.24612 (100%)	0.24979 (101%)
espresso	0.05777	0.05145 (89%)	0.05064 (88%)
spice2	0.30970	0.26036 (84%)	0.22352 (72%)
su2cor	1.08854	1.08390 (100%)	1.11053 (102%)
tomcatv	0.82086	0.68474 (83%)	0.79914 (97%)
wave5	0.10083	0.06383 (63%)	0.06362 (63%)
xlisp	0.12358	0.10620 (86%)	0.10495 (85%)
nasa7	0.40998	0.35351 (86%)	0.38674 (94%)
btrix	0.40565	0.30295 (75%)	0.28570 (70%)
cholsky	0.74303	0.67255 (91%)	0.66393 (89%)
emit	0.04417	0.04198 (95%)	0.04521 (102%)
cfft2d	1.18129	1.00582 (85%)	1.06920 (91%)
gmtry	1.49627	1.45718 (97%)	1.72924 (116%)
mxm	0.20791	0.14724 (71%)	0.24979 (120%)
vpenta	0.82674	0.71672 (87%)	0.70720 (86%)

Table 5.4: Benchmarks with high reduction in MCPI by using Default Prefetch

Benchmark	Best MCPI by PFONMISS	Best MCPI by Pure SIRPA	Best MCPI by SIRPA w/ Def. Prefetch
emit	0.04417	0.15239	0.04403
compress	0.24723	0.25373	0.24688
su2cor	1.08854	1.13396	1.08465
tomcatv	0.82086	0.85102	0.75393

Table 5.5: Best Software Based Prefetch Algorithm

Benchmark	Software Based Prefetch Algorithm
btrix	SETCAM w/ Line Concept
cholsky	SETCAM
emit	SETCAM w/ Line Concept
cfft2d	PREFETCH w/ Line Concept
gmtry	SETCAM w/ Line Concept
mxm	SETCAM w/ Line Concept
vpenta	SETCAM w/ Line Concept

Table 5.6: Overall MCPI by using SETCAM & SIRPA family Algorithms

Benchmark	**Best MCPI by SETCAM family**	**Best MCPI by SIRPA family**
btrix	0.29597	0.30295
cholsky	0.68265	0.67255
cfft2d	1.46480	1.00582
emit	0.04641	0.04198
gmtry	1.74663	1.45718
mxm	0.14717	0.14724
vpenta	0.71777	0.71672

Chapter 6

Conclusion

In this thesis, we proposed a new cache prefetch algorithm based on the register usage and addressing mode of CPU instructions. The SIRPA scheme has a superior performance compared with conventional OBL schemes such as PFONMISS. The accuracy of SIRPA scheme is very high, and the scheme performs consistently well across a wide range of benchmark programs. The SIRPA scheme tracks the common constant stride access patterns and fires cache prefetch for the next memory address where the process will access in the next loop iteration. The scheme compares well in performance with more sophisticated scheme like the Chen's RPT scheme, however, the hardware complexity is greatly reduced due to lack of large size RPT, LA–PC and BPT in SIRPA.

A major achievement by using the SIRPA scheme is the selection of source register as the target for use as index in the SVT. As the number of registers in a processor is fixed, the number of entries in the SVT corresponding with each register is readily determined. The number of targets for checking constant stride access patterns is small due to the fact that the addresses to arrays in memory are usually stored in registers in RISC processors. The SVT will not be polluted by other non-constant stride memory accesses compare with other scheme using instruction address the index to stride discovery table.

By using a good hardware cache prefetch algorithm, like the SIRPA, the improvement in memory latency can match that of the software or hybrid cache prefetch scheme, such as the SETCAM. The benefit of hardware cache prefetch scheme is software compatibility with already available software, no instruction set modification and no need to have language compiler support.

Two enhancements to current cache prefetch scheme were studied. One is Line Concept scheme, which makes a cache prefetch to fire earlier than the scheme without using it. The improvement includes a higher chance that the cache prefetch will be carried out, because there is a longer period that the cache prefetch may find an idle memory bus. Another improvement is to increase the available cache prefetch time and hence improve the computation and memory transfer overlap. The end result is a reduced memory latency for the whole system. We had applied the Line Concept scheme to the SIRPA scheme, the Chen's RPT scheme, the SETCAM scheme and the software PREFETCH instruction scheme. We found that the Line Concept scheme is effective to a broad range of cache prefetching algorithms especially for the hardware cache prefetching algorithms.

The other enhancement to cache prefetch schemes proposed in this thesis is the Default Prefetch scheme. The Default Prefetch scheme helps when the memory access pattern is not predominantly constant stride, but still with high spatial locality. The scheme effectively brings the benefit of PFONMISS and SIRPA together.

In this thesis, we had studied a variety of on chip first and second level cache configurations with unified cache management system. We had shown that a unified system is desirable in the on chip environment due to reduced hardware overhead and electrical loadings on the CPU bus. The information on the CPU chip can aid the cache prefetch algorithm to make accurate predictions to future

memory access. The performance of both levels of cache is good on a range of benchmark programs.

In conclusion, unified on chip cache management schemes with cache prefetch algorithms for the first and second level cache were studied, the impact to the memory latency was simulated and measured by a set of well accepted benchmark programs. The results showed that the on chip cache system can bridge the disparity in speed between the processor and the on board main memory. The trend to include on chip second level cache is undeniable. A coherent and pro-active method to manage on chip cache memory is necessary to realize the highest yield that the small size memory can give out. The data presented in this thesis is a firm foundation to support a good hardware design of on chip cache system.

Chapter 7

Future Directions

The cache prefetch algorithms and cache management schemes proposed in this thesis can be further enhanced. The following is a few suggestions that future study can be based on with elementary justifications. The author intends to publish results in the future by using the methods suggested in the next paragraphs.

7.1 Prefetch Buffer

Cache prefetches may increase memory latency, and hence decrease system performance, if the prefetched cache block replaces another block which is required by the processor. This creates a conflict miss. In cache prefetch scheme like SIRPA, the prefetches are highly accurate, the prefetching action can overlap computation with memory transfer time, that it will be unwise to stop the cache prefetch in order to minimize the cache for conflict miss. How can we circumvent the paradox?

One suggestion is to make use of a small buffer which is designed to hold cache prefetch blocks in it [Jou90]. When the prefetched blocks and the normal cache blocks are separated, there will be no conflict miss possible between them. The prefetch buffer should be small in size, 16–32 blocks in the buffer should be

enough as those prefetches are for look ahead memory accesses and the majority of programs may not have a program loop using more than 32 arrays inside it. The replacement management of the prefetch buffer should be very simple, we do not expect there will be high temporal locality in the prefetch buffer, because most of the prefetches detected by the SVT in SIRPA is due to spatial locality. One simple replacement algorithm that can be used in the prefetch buffer is *First In First Out* (FIFO).

When a block in the prefetch buffer is accessed by the processor, we may have two possible schemes to handle the case. One is to put the block in the prefetch buffer back to the normal cache. Then, the replacement management thereafter will be that of the normal cache, which may be LRU. Another method is to let the prefetch block remains in the prefetch buffer. There are merits and demerits in both schemes. The prefetch buffer for the first and second level cache should be separated.

The configuration of a system with prefetch buffer is shown in figure 7.1.

7.2 Dissimilar L1-L2 Management

In the experiments performed in this study, the cache management scheme in the first and second cache is the same. The benefit of using a unified scheme to handle both level of cache memory is reduced hardware overhead and electrical loading on the CPU bus. However, the details of the cache prefetch scheme and other parameters used in the cache management can be different in the first and second level caches, in order to fine tune the cache behavior of respective cache levels.

One method may use a reduced strength cache prefetch algorithm on the first level cache, in order to shorten the critical time path. But use a more

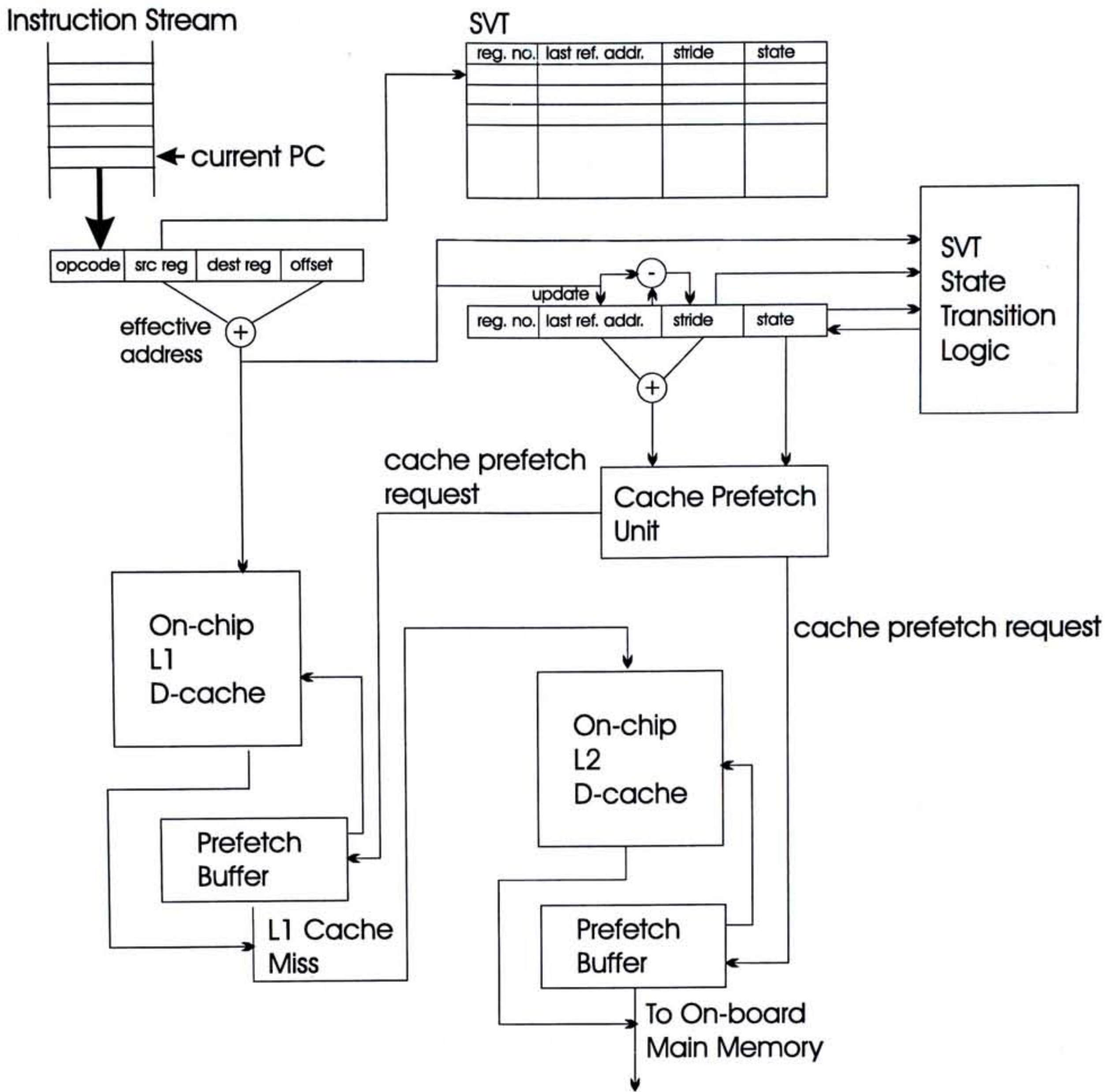


Figure 7.1: Processor with Prefetch Buffer

sophisticated cache prefetch algorithm in the second level cache. Certain cache prefetching schemes may be more suitable for a larger cache, whereas others may perform better with smaller cache block size. In the previous chapters, the Line Concept scheme and the Default Prefetch scheme were shown to be effective in general benchmarks, but there are cases where the above schemes may not perform as good.

The Line Concept is good for small cache block size, as the scheme will compensate for the less cache prefetch time in small size cache blocks. A small block size cache will make the number of cache blocks to be larger, which will help in

reducing conflict miss.

The Default Prefetch scheme will perform better with a large cache memory. The accuracy of the Default Prefetch (PFONMISS) is not as high as the SIRPA scheme. However, the Default Prefetch can fill in the void when the SIRPA cannot fire a cache prefetch. We expect the Default Prefetch will introduce a small amount of cache pollution, the pollution effect will be less destructing for a larger cache. With a very small size cache like those used in the first level cache, the tolerance for cache pollution will be very low.

Different combinations of cache prefetch schemes for the first and second level caches may make the overall memory latency to be smaller. The effect of different cache prefetch schemes should be studied with respect to different cache parameters. The unified on chip cache management can be modified to apply tuned algorithms for the first and second level cache.

7.3 Combined LRU/MRU Replacement Policy

In a cache, due to limited space to hold data, an algorithm has to be devised to choose a cache block to be discarded when a new block has to be called in. The best solution is to choose the cache block that will not be accessed for the longest period of time in future. However, without a pre-run of the program, one cannot know in advance which cache block will not be accessed for the longest time. A pre-run is simply not a solution for the majority computer systems in use, therefore, approximation algorithms have to be devised. A well accepted algorithm is *Least Recently Used* (LRU) algorithm. Due to temporal locality, for the cache block which was not accessed for the longest time is assumed to be not useful in the future, and that cache block will be chosen for replacement. The

other side of the algorithm implies that, for whatever cache block just accessed, it will have low chance to be chosen for replacement.

The LRU algorithm works for the majority of cases, which temporal locality exhibits, however, there are situations where the opposite to LRU, ie. *Most Recently Used* (MRU) replacement will be the optimal solution. The following simplified example shows the phenomenon:

Assumed the following cache blocks accessed in a program loop in sequence:

1, 2, 3, 1, 2, 3

Cache slots to be available are 2

If LRU replacement is used, the cache replacement activities are shown in table 7.1.

Table 7.1: LRU Replacement Example

Step	**Slot 1**	**Slot 2**
1	block 1	empty
2	block 1	block 2
3	block 3 (replacement)	block 2
4	block 3	block 1 (replacement)
5	block 2 (replacement)	block 1
6	block 2	block 3 (replacement)

Total replacement: 4

If MRU replacement is used, the cache replacement activities are shown in table 7.2

The above shows that LRU may not be the best algorithm in all cases, however we do not want to pay the price where non-LRU replacement algorithm gives a terrible prediction.

Table 7.2: MRU Replacement Example

Step	**Slot 1**	**Slot 2**
1	block 1	empty
2	block 1	block 2
3	block 1	block 3 (replacement)
4	block 1	block 3 (hit, no replacement)
5	block 1	block 2 (replacement)
6	block 1	block 3 (replacement)

`Total replacement: 3`

With on chip first and second level cache using SIRPA, we can improve the LRU replacement algorithm by extracting data from the SVT as described in the following.

In the prefetch address calculation, which is:

prefetch address = current effective address + stride

If stride value is found to be positive and large, that means the next access will fall on another cache block. It is a strong hint that the current accessed cache block will *not* be accessed soon, as the stride value confirms that the access pattern is going forward. We can place such cache block on a MRU queue, which the head of the queue will be chosen for replacement if another cache block has to be called in.

For other memory accesses, the conventional LRU algorithm will apply, which whenever a cache block is accessed, the block will be placed onto the tail of the LRU queue, which that block will have the least chance to be replaced.

The combined LRU/MRU replacement policy should improve the efficiency of the cache system, as less conflicts will be expected. The net result of the

cache system will be similar to a cache design with larger cache size. The new replacement algorithm can be implemented separately in on-chip first and second level cache.

7.4 N Loops Look-ahead

In the SIRPA scheme proposed in the previous chapter, the cache prefetches issued are those look–aheaded memory accesses by the processor for one loop. The scheme checks the current executing instruction, if the addressing mode is register indirect, and the entry in the SVT corresponds with a constant stride access pattern, a cache prefetch will be fired with address:

prefetch address = current effective address + stride

The prefetch address should correspond to the next element in the same array where the processor will reference in the next loop iteration. It is a *One Loop Look-ahead* scheme. It will be easy to modify the scheme to make *N Loops Look-ahead*, the new formula to compute the prefetch address will become:

prefetch address = current effective address + $N *$ stride

The determination of the value N will be a complex task, as the farer the cache prefetches, the more chance that the prefetched blocks will cause conflict miss in the cache. Moreover, at the end of the loop iterations, the extra predictions to the outside of an array will cause cache pollution.

One method is to adjust the value of N dynamically, and N is stored in each entry in the SVT. The other method is through the use of prefetch buffer as mentioned in the above paragraphs.

Bibliography

[AC95] Ali Ahi, Yung-chin Chen, etc., R10000 Superscalar Microprocessor, in Hot Chips VII (95), pages 7.4.x, 1995.

[APB92] Mani Azimi, Bindi Prasad, Ketan Bhat, Two Level Cache Architectures, IEEE, 1992.

[BC91] Jean-Lou Baer and Tien-Fu Chen, An effective on-chip preloading scheme to reduce data access penalty, in Proceedings, 1991 International Conference on Supercomputing, pages 176–186, 1991.

[BW88] Jean-Loup Baer, Wen-Hann Wang, On the Inclusion Properties for Multi-level Cache Hierarchies, in IEEE 1988, pages 73–80, 1988.

[Ban95] Peter Bannon, The Alpha 21164A: Continued Performance Leadership, in Microprocessor Forum, 1995.

[Bol94] Keith Boland, Predicting and Precluding Problems with Memory Latency, in IEEE Micro, pages 59-67, August 1994.

[BDT93] Rodney Boleyn, James Debardelaben, Vivek Tiwari, Andrew Wolfe, A Split Data Cache for Superscalar Processors, IEEE, 1993.

[BKW90] Anita Borg, R. E. Kessler, David W. Wall, Generation and Analysis of Very Long Address Traces, IEEE, 1990.

[BuK90] Hakon O. Bugge, Ernst H. Kristiansen, Bjorn O. Bakka, Trace-driven Simulations for a Two-level Cache Design in Open Bus Systems, IEEE, 1990.

[ChH94] Si-en Chang, Chia-chang Hsu, Efficient Simulation Methods for Multi-level cache Memory Hierarchies, IEEE, 1994.

[Che93] Tien-Fu Chen, Data Prefetching for High-Preformance Processors, in Technical Report 93–07–01, Department of Computer Science and Engineering, University of Washington, July 1993.

[CB92] Ten-Fu Chen and Jean-Loup baer, Reducing Memory Latency via Non-blocking and Prefetching Cache, in Proceedings, ASPLOS IV, pages 51–61, 1992.

[ChK91] Alok Choudhary, Senthil Krishnamoorthy, Experimental Evaluation of Multilevel Caches for Shared Memory, IEEE, 1991.

[CTR93] Computer Technology Research Corporation, Pentium: Intel's 64-bit Superscalar Architecture, Computer Technology Research Corporation, 1993.

[DDS95] Fredrik Dahlgren, Michel Dubois, Per Stenstrom, Sequential Hardware Prefetching in Shared-memory Multiprocessors, in IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 7, July 1995.

[FP91] John W. C. Fu and Janak H. Patel, Data Prefetching in Multiprocessor Vector Cache Memories, in Conference Proceedings, The 18th International Symposium on Computer Architecture, pages 54–63, 1991.

[FPJ92] John W. C. Fu, Janak H. Patel and Bob L. Janssens, Stride Directed Prefetching in Scalar Processors, in Conference Proceedings, The 25th International Symposium on Microarchitecture, pages 102–110, 1992.

[GHP93] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith, Cache Performance of the SPEC92 Benchmark Suite, in Technical report, Computer Science Division, University of California at Berkeley, 1993, Technical Report UCB/CSD 91/648.

[Goo89] A. J. van de Goor, Computer Architecture and Design, Addison-Wesley Publishing Company, 1989.

[GGV90] Edwared H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum, Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies, in Proceedings, 1990 International Conference on Supercomputing, pages 354–368, 1990.

[Gwe94] Linley Gwennap, UltraSparc Unleashes SPARC Performance, in Microprocessor Report, October 3, 1994, pages 1-10, 1994.

[HaJ92] L. P. Happel, A. P. Jayasumana, Performance of a RISC machine with two level caches, in IEE Proceedings-E, Vol. 139, No. 3, May 1992.

[HP90] John L. Hennessy, David A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1990.

[Hil87] Mark D. Hill, Aspects of Cache Memory and Instruction Buffer Performance, Technical report, Computer Science Division, University of California at Berkeley, November 1987, Technical Report UCB/CSD 87/381.

[Hil88] Mark D. Hill, A Case for Direct Mapped Caches, Computer, IEEE, December 1988.

[Ho95] Chi-Sum Ho, Software-Assisted Data Prefetching Algorithms, in his Master Thesis, Department of Computer Science, Chinese University of Hong Kong, June 1995.

[HuO94] Rupinder Hundal, Vojin G. Oklobdzija, Determination of Optimal Sizes for a First and Second Level SRAM-DRAM On-Chip Cache Combination, IEEE, 1994.

[Jou90] Norman P. Jouppi, Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers, in the 17th Annual International Symposium on Computer Architecture, Conference Proceedings, May 28–31, 1990.

[Jou93] Norman P. Jouppi, Cache Write Policies and Performance, in Conference Proceedings, The 20th International Symposium on Computer Architecture, pages 191–201, 1993.

[JoW94] Norman P. Jouppi, Steven J. E. Wilton, Tradeoffs in Two-level On-chip Caching, IEEE, 1994.

[KL91] Alexander C. Klaiber and Henry M. Levy, An Architecture for Software-controlled Data Prefetching, in Conference Proceedings, The 18th International Conference on Computer Architecture, pages 43–53, 1991.

[LBJ95] Sung-soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-do Rhee, etc., An Accurate Worst Case Timing Analysis for RISC Processors, in IEEE Transactions on Software Engineering, Vol. 21, No. 7, July 1995.

[Liu94] Lishing Liu, Issues in Multi-level Cache Design, IEEE, 1994.

[MeM92] N. Mekhiel, Daniel C. McCrackin, Performance Analysis for a Two Level Cache System, in IEEE, 1992.

[MeM95] N. Mekhiel, D. C. McCrackin, Simplified Performance Modelling of Hierarchical Memory Systems, IEEE, 1995.

[Mel95] Charles Melear, Integrated Memory elements on Microcontroller Devices, IEEE, 1995.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, in Proceedings, ASPLOS IV, pages 62–73, 1992.

[NeA91] B. Nelson, J. Archibald, K. Flanagan, Performance Analysis of Inclusion Effects in Multi-Level Multiprocessor Caches, IEEE, 1991.

[ObK95] M. S. Obaidat, Humayun Khalid, A Performance Evaluation Methodology for Computer Systems, IEEE, 1995.

[Oma81] Thabit Khalid Omar, Cache Management by the Compiler, in his PhD thesis, Department of Computer Science, Rice University, May 1981.

[OyW92] Yen-jen Oyang, Le-chun Wu, Optimal Design of Megabyte Second-level Caches for Minimizing Bus Traffic in Shared-memory Shared-bus Multiprocessors, IEEE, 1992.

[Por89] Alan K. Porterfield, Software Methods for Improvement of Cache Performance on Supercomputer Applications, in his PhD thesis, Department of Computer Science, Rice University, May 1989.

[Prz90] Steven A. Przybylski, Cache and memory hierarchy design: a performance-directed approach, Morgan Kaufmann Publishers, Inc., 1990.

[PHH88] Steven Przybylski, Mark Horowitz, John Hennessy, Performance Tradeoffs in Cache Design, IEEE, 1988.

[Sez93] Andre Seznec, About Set and Skewed Associativity on Second Level Cache, IEEE, 1993.

[SL88] Robert T. Short, Henry M. Levy, A Simulation Study of Two-Level Caches, in IEEE 1988, pages 81–88, 1988.

[Smi82] Alan Jay Smith, Cache Memories, Computing Survey 14, 3:473–530, 1982.

[STW92] Harold S. Stone, John Turek, Joel L. Wolf, Optimal Partitioning of Cache Memory, in IEEE Transactions on Computer, Vol. 41, No. 9, 1992.

[TaS94] Ju-Ho Tang, Kimming So, Performance and Design Choices of Level-two Caches, IEEE, 1994.

[Wan89] Wen-hann Wang, Multilevel Cache Hierarchies, PhD Thesis, Department of Computer Science and Engineering, University of Washington, 1989.

[WHP95] Karl Westerholz, Stephan Honal, Josef Plankl, etc., Improving Performance by Cache Driven Memory Management, IEEE, 1995.

[YaA94] Qing Yang, Sridhar Adina, A One's Complement Cache Memory, in Proceedings of the 1994 International Conference on Parallel Processing, Aug. 15–19, 1994.

[YeP93] Tse-yu Yeh, Yale N. Patt, Branch History Table Indexing to Prevent Pipeline Bubbles in Wide-issue Superscalar Processors, IEEE, 1993.

[YoS93] Honesty C. Young, Eugene J. Shekita, An Intelligent I-Cache Prefetch Mechanism, IEEE, 1993.

CUHK Libraries
003510956