Data Prefetching for High-Performance Processors

Tien-Fu Chen

Technical Report 93-07-01

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

July 1993

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with ‘‘fair use’’ as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted ‘‘the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.’’

Signature______________________________

Date_________________________________

University of Washington

Abstract

Data Prefetching for High-Performance Processors

by Tien-Fu Chen

Chairperson of Supervisory Committee: Professor Jean-Loup Baer
Department of Computer Science
and Engineering

Recent technological advances are such that the gap between processor cycle times and memory cycle times is growing. Techniques to reduce or tolerate large memory latencies become essential for achieving high processor utilization. In this dissertation, we propose and evaluate data prefetching techniques that address the data access penalty problems.

First, we propose a hardware-based data prefetching approach for reducing memory latency. The basic idea of the prefetching scheme is to keep track of data access patterns in a reference prediction table (RPT) organized as an instruction cache. It includes three variations of the design of the RPT and associated logic: generic design, a lookahead mechanism, and a correlated scheme. They differ mostly on the timing of the prefetching. We evaluate the three schemes by simulating them in a uniprocessor environment using the ten SPEC benchmarks. The results show that the prefetching scheme effectively eliminates a major portion of data access penalty and is particularly suitable to an on-chip design and a primary-secondary cache hierarchy.

Next, we study and compare the substantive performance gains that could be achieved with hardware-controlled and software-directed prefetching on shared-memory multiprocessors. Simulation results indicate that both hardware and software schemes can handle programs with regular access patterns. The hardware scheme is good at manipulating dynamic information, whereas software prefetching has the flexibility of prefetching larger blocks of data and of dealing with complex data access patterns. The execution overhead of the additional prefetching instructions may decrease the software prefetching performance gains. An approach that combines software and hardware schemes is shown to be very

promising for reducing the memory latency with the least overhead.

Finally, we study non-blocking caches that can tolerate read and write miss penalties by exploiting the overlap between post-miss computations and data accesses. We show that hardware data prefetching caches generally outperform non-blocking caches. We derive a static instruction scheduling algorithm to order instructions at compile time. The algorithm is shown to be effective in exploiting instruction parallelism available in a basic block for non-blocking loads.

Table of Contents

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 Overlapping computations with memory accesses . . . 2
1.2 Organization of the Dissertation . . . 5

Chapter 2: Related Work 7

2.1 Data Prefetching . . . 7
2.1.1 Hardware-Controlled Prefetching . . . 7
2.1.2 Software-directed Prefetching . . . 10
2.2 Non-blocking Caches . . . 12
2.2.1 Lockup-free Caches . . . 12
2.2.2 Non-blocking Reads/Writes . . . 13

Chapter 3: Data Prefetching Caches 15

3.1 Motivation . . . 15
3.2 Generic Reference Prediction . . . 17
3.2.1 Reference Prediction Table - RPT . . . 18
3.2.2 RPT mechanism . . . 18
3.2.3 Example and Discussion . . . 21
3.3 Lookahead Reference Prediction . . . 22
3.3.1 Lookahead Program Counter (LA-PC) and RPT . . . 24
3.3.2 Lookahead Distance and Limit . . . 24
3.3.3 Handling Cache Misses . . . 26
3.4 Correlated Reference Prediction . . . 26
3.4.1 Implementation of Correlated RPT . . . 27

3.5 Summary 30

Chapter 4: Performance Evaluation of Hardware Prefetching Schemes 31

4.1 Evaluation Methodology for Uniprocessors 31
4.1.1 Trace driven Simulation 31
4.1.2 Architectural Models 32
4.1.3 Benchmarks and Metrics 35
4.2 General Results 36
4.3 Effect of Design Variations 40
4.3.1 Effect of Memory Models and Latencies 40
4.3.2 Effect of Block Size 43
4.3.3 Organizing the Reference Prediction Table 44
4.3.4 Varying the Lookahead Limit 46
4.3.5 Alternatives for the Placement of the Prefetched Data 47
4.4 Summary 52

Chapter 5: Comparative Evaluation of Software and Hardware Prefetching Schemes 54

5.1 Overview 54
5.2 Software Prefetching 55
5.3 Qualitative Comparison 56
5.3.1 High-level Comparison 57
5.3.2 Identifying Cache Misses 57
5.3.3 Prefetch Instruction and Predicate 59
5.3.4 Scheduling Prefetches 60
5.3.5 Prefetching in Multiprocessors 62
5.3.6 Other Aspects and Final Words 63
5.4 Quantitative Evaluation Methodology 64
5.4.1 Architectural Models 65
5.4.2 Simulation Environment and Benchmarks 66
5.4.3 Model Implementations 69
5.5 Simulation Results 71

5.5.1 General results 72
5.5.2 Detailed Analysis 75
5.5.3 Negative Effect of Prefetching 77
5.5.4 Effect of Memory Latency 80
5.5.5 Impact of Consistency Models 83
5.6 Combining Hardware and Software Prefetching 84
5.7 Summary 88

Chapter 6: Non-blocking Caches **90**
6.1 Overview 90
6.2 Background and Performance Issues 91
6.2.1 Non-blocking Caches 91
6.2.2 Performances Issues 92
6.3 Architectural Models and Evaluation Methodology 94
6.3.1 Processor-cache Models 94
6.3.2 Simulation Method 96
6.4 Simulation Results 98
6.4.1 Effect of Architectural Variations 98
6.4.2 Effect of Large Latency 103
6.5 Compiler Assistance for Non-blocking Loads 104
6.5.1 Instruction Scheduling and Register Renaming 104
6.5.2 Algorithm for Non-blocking Loads 105
6.5.3 Effect of Instruction Scheduling 111
6.6 A Hybrid Design 113
6.7 Summary 115

Chapter 7: Conclusion **116**
7.1 Summary of Results 116
7.2 Future Research 118

Bibliography **121**

Appendix A: Supplemental Data **130**
A.1 Evaluation of Data Prefetching 130
A.2 Evaluation of Non-blocking Caches 137

List of Figures

3.1 Example of Matrix Multiplication 17
3.2 Reference Prediction 19
3.3 Example: Filling RPT entries 21
3.4 Block diagram of data prefetching 23
3.5 RPT with Lookahead mechanism 25
3.6 Kernel 6 27
3.7 Correlated RPT 27
3.8 Example: Correlated RPT entries 29

4.1 Trace-driven simulator using *pixie* 32
4.2 Three memory models 34
4.3 Simulation Results for $\delta = 30$ -- *Overlapped* 37
4.4 Effect of memory models and latencies 41
4.5 MCPI vs. block size for 32K cache (*Overlapped*) 44
4.6 Hit ratio and attempted prefetch of RPT 45
4.7 MCPI vs. LA-limit (d) for $\delta = 30$ (*Overlapped*) 47
4.8 Variations in prefetching placement 50

5.1 Example of instrumented loop 58
5.2 Scheduling prefetches 61
5.3 Model Architecture 65
5.4 Direction Execution Simulator 67
5.5 Simulation results 73
5.6 Network traffic 78
5.7 Effect of memory latency 81
5.8 Prefetching based on weak and sequential consistency 84
5.9 Effectiveness of combining HW-pf and SW-pf 86

6.1 Prefetching vs. Lockup-free for $\delta = 30$ (*Pipelined*) 99
6.2 Effect of a larger latency (for $\delta = 30$ vs. $\delta = 100$ *Pipelined*) 103
6.3 Building the DAG for a basic block 106
6.4 Instruction Scheduler on the DAG 107
6.5 DAG of an example 108
6.6 Scheduling the example 108
6.7 Instruction scheduling and register renaming 109
6.8 Effect of instruction scheduling on NBC for $\delta = 30$ (*Non-overlapped*) . . . 112
6.9 Effect of instruction scheduling on NBC for $\delta = 30$ (*Pipelined*) 113
6.10 Hybrid design on varying cache size $\delta = 30$ (*Pipelined*) 114

A.1 Effect of memory models and latencies (continued from Figure 4.4) 130
A.2 MCPI vs. block size for 32K cache (continued from Figure 4.5) 133
A.3 MCPI vs. LA-limit (d) for $\delta = 30$ (continued from Figure 4.7) 134
A.4 Variations in prefetching placement (continued from Figure 4.8) 135
A.5 Prefetching vs. Lockup-free for $\delta = 30$ (continued from Figure 6.1) 137
A.6 Hybrid design on varying cache size $\delta = 30$ (continued from Figure 6.7) . . 139
A.7 Effect of memory models: Prefetching vs. Lockup-free 140

List of Tables

3.1 Classification of memory access patterns 16

4.1 Characteristics of benchmarks 36

5.1 Benchmarks characteristics - average numbers for a single processor in the 16 processor simulation 68

5.2 Proportions of Conflicts in the direct-mapped sets 79

6.1 Architectural Choices 96

6.2 Statistics of benchmarks (for first 100 million instructions on 32K baseline cache) 97

6.3 Average of basic block size, non-blocking distance, and extra registers needed 112

Acknowledgments

I would like to express sincere appreciation to my advisor, Professor Jean-Loup Baer. I am indebted to him for his inspiration, patience, and encouragement. Without his valuable guidance and insight, I would never have completed this study.

I would like to thank Professor Susan Eggers for her helpful comments and constructive criticism in my study. I thank Professor Arun Somani for his effort on the reading committee and for his comments and suggestions in this dissertation. I also thank the other members of my supervisory committee, Professor Carl Ebeling and Professor George Prater.

I wish to thank friends Wen-Hann Wang, who gave me unfailing encouragement and support, Yi-Bing Lin for his valuable technical assistance, and Shu-Yuen Hwang for his suggestions. I thank Michael Smith and Stephen Goldschmidt at Stanford for their helping me to use simulation tools *pixie* and tango. Without their help, the experimental study in this dissertation would not have been possible. I also appreciate fellow graduate students Craig Anderson, Claudia Chiang, Calvin Lin, Ton Anh Ngo, and Richard Zucker during my study in the department. I also thank my friends Yu-Jung Chang, Yeh-Chung Din, and Po-Fat Yuen for giving me a wonderful stay in Seattle.

Lastly, I wish to give special thanks to my wife Meei-Shin, my daughter Janice, and my family in Taiwan for their love and support.

Chapter 1

Introduction

Processor performance has increased dramatically over the last few years and has now surpassed the 200 MIPS level. Memory latency and bandwidth have also progressed but at a much slower pace. It is therefore essential that we investigate techniques to reduce the effects of the imbalance between processor and memory cycle times. The introduction of caches between the processor and memory modules has been shown to be an effective way to bridge this gap. However, with a cache miss penalty that is becoming relatively larger, a system may still experience low processor utilization even with a high hit rate. Hence, efficient mechanisms for optimizing accesses to the memory hierarchy are mandatory for the realization of high performance systems.

The principle behind an effective cache implementation is to take advantage of locality without a performance loss. With current VLSI developments, several functional units, instruction and data caches, and hardware assists can be included on the processor chip. Therefore, a first obvious method for reducing the average memory access time is to implement multi-level cache hierarchies [Baer & Wang 89] with an on-chip first level cache. However, under the usual caching mechanism, the processor will still stall on a first-level cache miss and of course also on misses on any of the next levels of the memory hierarchy with an even larger penalty time, until the miss is resolved. Since usually a processor must stall on a cache miss, caches do not actually hide memory latency but, instead, they eliminate many memory accesses. In order to make further progress towards the reduction in memory latency, memory accesses due to cache misses must proceed in parallel with processor execution. As a result, a number of different solutions have been proposed to allow computations to overlap with memory accesses for hiding memory latency. They basically provide efficient mechanisms to allow buffering and pipelining of memory references.

1.1 Overlapping computations with memory accesses

Many solutions have been proposed to reduce memory accesses and/or hide memory latency. They can be classified according to several dimensions such as the type of latency (read, write, consistency), the architectural component (processor, cache, or network), the target system (uniprocessor or multiprocessor), and the basic technology (hardware, software, or hybrid). In this introductory chapter, we briefly list some of them; those most germane to our work will be described in more detail in the next chapter.

The simplest technique is a write buffer, a FIFO queue which is used to hide the write latency [Smith 82a]. Buffering is a performance enhancement for caches both with write-back and write-through strategies. Under write-back, the write buffer is used to hold replaced dirty blocks while ''normal'' execution proceeds. The system uses spare interconnect and memory cycles to write buffered values to the next level of the memory hierarchy. Under write-through, the buffering of write requests releases the CPU from waiting for the writes to complete. An extension to the write buffer is a write cache [Bray & Flynn 91], organized like a small regular cache, that uses an allocate strategy on write misses and write backs to reduce the number of writes. Unlike a write buffer, the write cache allows writes which access the same cache line to be combined and thus, reduces the write miss penalty.

The next solution is a design that allows the processor to continue execution on unsatisfied memory references through the use of non-blocking caches (also called lockup-free caches [Kroft 81]). The non-blocking caches (see Section 2.2 for more detail) hide the latency of data misses through the overlap of data accesses and computations to the extent allowed by the data dependencies and consistency requirements. While the entire write latency can be hidden by a sufficiently large write buffer, dependence restrictions must be observed by the read requests in the processor and it is therefore likely that only a portion of the read latency can be hidden. Consequently, extra hardware complexity in both caches and processors is required to record the information pertaining to the outstanding requests, such as the cache line where the return value is to be stored and which function units are waiting for what data.

To achieve additional overlap, the compiler may perform instruction scheduling to avoid unnecessary stalls by keeping the CPU busy [Gibbons & Muchnick 86, Krishnamurthy 90, Kerns & Eggers 92]. The freedom of instruction scheduling in most compiler algorithms is limited by the data and control dependencies in the programs. Better performance is

generally achieved by moving loads far enough ahead of their uses. Like ‘‘regular’’ non-blocking loads, speculative loads [Chen *et al.* 92, Rogers & Li 92] fetch memory values into registers directly bypassing the cache if need be, i.e., without blocking the pipeline on a miss, and allow other instructions to be executed simultaneously. Because these loads are performed based on speculative execution, they will be scheduled under fewer constraints. For instance, speculative loads can be moved around across the basic block boundary regardless of control dependence restrictions. The basic requirement is that they should be issued speculatively without introducing unnecessary faults into the system. Speculative loads require both software and hardware supports, including instruction scheduling and extra state bits and transitions in register management.

An important approach is prefetching (the main topic of this dissertation), that is, the action of bringing data in the cache before it is actually needed. Prefetching is similar to speculative loads in the sense that it is non-blocking and behaves like a hint without incurring semantic faults. The main difference between prefetching and speculative loads is that the data are loaded into the cache rather than registers, and thus the restriction due to the limited number of registers does not limit the flexibility of prefetching. Depending on how prefetches are determined and initiated, prefetching can be either hardware-controlled [Baer & Chen 91, Fu & Patel 92] or software-directed [Porterfield 89, Klaiber & Levy 91, Mowry *et al.* 92]. The hardware approach detects accesses with regular patterns and issues prefetches at run time, whereas the software approach relies on the compiler to analyze programs and to insert prefetch instructions. In a shared-memory multiprocessor environment, where consistency requirements are taken into account, prefetching can be either binding (at compile time) [Lee *et al.* 87b, Gornish *et al.* 90] or non-binding (supported by hardware coherence) [Mowry & Gupta 91, Tullsen & Eggers 93]. In the former case, a data block location is bound to the value of prefetched data in caches at the time that the prefetch completes, whereas in the latter case, data is kept coherent by the cache coherency mechanism.

Yet another technique is the use of a processor with multiple hardware contexts [Kowalik 85, Weber & Gupta 89, Agarwal *et al.* 90, Alverson *et al.* 90, Nikhi *et al.* 91, Kurihara *et al.* 91]. If several threads are assigned to a processor, memory latencies can be masked by rapidly context switching to a different thread rather than waiting for a memory reference to complete. The two key issues for implementing multiple-context processors are: when is context switching performed [Boothe & Ranade 92, Laudon *et al.* 92],

and what defines a context [Hum & Gao 91]. Variations on these issues include conditional-switch, switch-on-cache-miss, and switch-every-cycle.

The cache coherence, or cache consistency, problem [Archibald & Baer 86] arises in shared-memory multiprocessors where several copies of the same block can be present in the local caches of the individual processors. The presence of write buffers and non-blocking caches makes the consistency problem more complicated, not only because yet another location for a copy of the data is possible, but also due to the fact that the results of writes may not be immediately observed (e.g., by the I/O system or other processors). One of the techniques to alleviate the processor's stalls on observing writes is load bypassing, that is, a memory load can bypass memory stores that are buffered and thus overlap between accesses can be exploited. Load bypassing is essentially required for processors with dynamic scheduling [Hennessy & Patterson 90]. A second technique is to relax the memory consistency model. A consistency model is an agreement between the parallel programs and the multiprocessor architecture on the ordering that shared references must observe. The most intuitive model is *sequential consistency*, that is, the serialization of the interleaving of the execution of the parallel processes like on a sequential machine. However, it imposes the strictest restrictions on the buffering of memory accesses. In order to further weaken the constraints imposed by serializability, several relaxed models of memory consistency have been proposed, including weak consistency and release consistency models[Dubois *et al.* 88, Adve & Hill 90, Gharachorloo *et al.* 91a, Dubois *et al.* 91, Zucker 92]. They allow memory accesses between synchronizations to be executed out-of-order and therefore exploit an overlap between memory accesses. Under these consistency models, the synchronizations should be explicitly specified.

In summary, many solutions have been proposed and shown to be effective in tolerating memory latencies. The success of these solutions relies on sufficient memory bandwidth for parallel memory accesses. Most of them require extra hardware support in processors and caches. A danger in the additional complexity is that it will increase the critical path time in the processor and thus offset the performance gains. The architecture proposed in this dissertation is an attempt to improve the technique of data prefetching for reducing memory latencies without increasing the critical cycle time.

1.2 Organization of the Dissertation

In this dissertation, we focus on one of the techniques mentioned above, namely hardware-based prefetching through a hardware unit whose function is to generate prefetches for the data cache. The goal of the prefetching is to reduce the processor stall time by bringing data into the cache just before its use. Ideally, the latency time would be totally masked; practically it can only be reduced since there are many impediments that prevent a perfect prediction of both the instruction stream, e.g., imperfect branch prediction, and of the data stream, e.g., data dependent addresses. The basic idea of the hardware-based prefetching scheme is to keep track of data access patterns in a reference prediction table (RPT) organized as an instruction cache.

In Chapter 3, we describe three variations of the design of the RPT and associated logic. They differ mostly on the timing of the prefetching. In the simplest scheme, called *generic*, prefetches can be generated one iteration ahead of actual use. The *lookahead* variation takes advantage of a look-ahead program counter that ideally stays one memory latency time, i.e., potentially several loop iterations, ahead of the real program counter and that is used as the control mechanism to generate the prefetches. Finally the *correlated* scheme uses a more sophisticated design to detect patterns across loop levels. As mentioned previously, a review of related work is presented in Chapter 2.

Chapter 4 presents the results of performance evaluation. The three designs are evaluated by simulating the ten SPEC benchmarks cycle-by-cycle in a uniprocessor environment. The results show that (1) the three hardware prefetching schemes all yield significant reductions in the data access penalty when compared to regular caches, (2) the *look-ahead* scheme is the preferred one in terms of cost-performance, and (3) the benefits are greater when the hardware assist augments small on-chip caches.

In Chapter 5, first we qualitatively compare the substantive performance gains that can be achieved with hardware-controlled and software-directed prefetching. Then we evaluate these two schemes for the prefetching of shared data through direct-execution simulation in a shared-memory multiprocessor environment. Results indicate that hardware prefetching is good at handling programs with regular access patterns and at manipulating dynamic information. In addition to the capability of handling regular accesses, software prefetching has the flexibility of prefetching larger blocks of data (rather than cache lines) and of dealing with some load dependent references. However, the execution overhead of the additional prefetching instructions decreases the software prefetching performance gains.

A combination of software and hardware approaches is promising in taking advantage of both schemes without incurring much overhead.

In Chapter 6, we discuss and evaluate the effectiveness of non-blocking caches and compare it with that of the proposed prefetching scheme. We consider compiler-based optimizations to enhance the effectiveness of non-blocking caches and propose a hybrid design based on the combination of prefetching and non-blocking schemes. Results from instruction level simulations show that the proposed hardware prefetching caches generally outperform non-blocking caches. Also, the relative effectiveness of non-blocking caches is more adversely affected by an increase in memory latency than that of prefetching caches. However, the performance of non-blocking caches can be improved substantially by compiler optimizations such as instruction scheduling and register renaming. The hybrid design can be very effective in reducing the memory latency penalty for many applications.

Finally, Chapter 7 summarizes the dissertation and suggests directions for future research.

Chapter 2

Related Work

This chapter gives a survey of previous work which is directly relevant to the main theme of this dissertation. First, we specifically review previous data prefetching techniques, including hardware-controlled and software-directed approaches. We then briefly review non-blocking techniques that are used to hide memory latency and discuss associated compiler optimization algorithms.

2.1 Data Prefetching

Data prefetching in the context of this work is defined as the asynchronous action of bringing data in the data (or mixed instruction-data) cache before it is directly accessed by a memory instruction (such as a load or a store). In fact, data that will never be used might be erroneously prefetched. The prefetching might be triggered either by a hardware mechanism, or by a software instruction, or by a combination of both.

2.1.1 Hardware-Controlled Prefetching

Hardware-based approaches can be classified into two categories: *spatial* schemes, when the decision to prefetch is based on the access to the current cache block, and *temporal* schemes, which implies lookahead decoding of the instruction stream. In the spatial approaches, prefetches occur when there is a miss on a cache block, and the address of the prefetched block is dependent on the current data access, while prefetches in the temporal category occur at times ahead of actual use and are not related to cache misses.

Spatial schemes

Smith [82a] studies variations on the one block look-ahead (OBL) policy, i.e., upon referencing block i, the only potential prefetch is to block $i+1$. Upon referencing block i, three options are: prefetch block $i+1$ unconditionally, prefetch block $i+1$ only on a miss to block i, or prefetch block $i+1$ if block i is referenced for the first time after prefetching

(a one-bit encoding is required). Smith's experiments show a decrease in miss ratios when prefetching is used but with a concomitant increase in memory traffic due to potentially prefetching unused blocks. Memory latencies are now (relatively to the processor speed) much larger than when Smith performed his experiments. Therefore, the risks of the processor stalling, because the memory bus is busy servicing a yet unneeded prefetched block rather than a current miss, has become greater. Based on a similar observation, Przybylski [90] argues against complex (pre)fetch strategies because either there is not enough memory bandwidth or because misses are too temporally clustered.

An extension to OBL where several consecutive data streams are prefetched in FIFO *stream buffers* is proposed by Jouppi [90]. The FIFO queues are filled sequentially starting from the missing block address. He finds that a stream buffer of four 8-byte blocks can remove up to 85% of the misses for a 4K I-cache but will remove only about 35% of the misses of a 4K D-cache. As could be expected, OBL, or extensions based on (sequential) spatial locality, would work better for I-caches than for D-caches and its effectiveness decreases with increased block sizes. Miss rates can be reduced, mostly for direct-mapped caches, at the expense of some increase in memory traffic. These OBL-based schemes take advantage of limited (sequential) spatial locality but are not able to deal with large strides.

The use of stride information carried by vector instructions leads Fu and Patel [91] to propose prefetch strategies for vector processors. They define the *cache load size* l as the number of bytes loaded into the cache on a miss, $l = (p + 1) \times b$, where b is the block size and p is the number of blocks prefetched. A *sequential-prefetch* strategy is such that, on a cache miss, the cache prefetches p **consecutive** blocks for a reference which is a scalar or a short stride ($< b$) vector access. The *stride-prefetch* strategy states that in addition to sequential-prefetching, the cache prefetches p blocks for long stride ($\geq b$) vector accesses on a cache miss, where those b-byte blocks are separated by the stride. By simulating four numerical programs on a 2-way 64K cache with a block size of 32 bytes and a 128-byte load size, they find that sequential-prefetch can increase performance by 30%-50% by loading multiple small blocks to capture spatial locality. A stride-prefetch cache shows some performance improvement, but not significantly better than a sequential-prefetch cache.

Later on, Fu and Patel [92] derive a similar approach for scalar processors. The primary mechanism is to record the previous memory address in a history table and to generate prefetch requests by calculating a stride between the current address and the previous

address if the stride is non-zero. Their results show that a history table with a small number of entries can significantly reduce, with low overhead, the number of cache misses for programs that can be highly vectorized. However, a significant overhead for non-vectorized programs may occur. The main problem is that the approach lacks the control of preventing unnecessary prefetches on irregular accesses or unneeded blocks. The same general idea of a hardware assist is presented by Sklenar [92], but without any performance or cost evaluation. These two schemes post-date our earlier study of a hardware-assist function for prefetching [Baer & Chen 91]. Our approach will be described in Chapter 3.

Temporal schemes

By lookahead decoding of the instruction stream, temporal mechanisms attempt to have data be in the cache ‘‘just in time’’ to be used. Lee *et al.* [87a] propose a data prefetching scheme based on instruction lookahead for CISC-like machines. The processor includes an instruction prefetch buffer and a data prefetch buffer (FIFO queue), which is used to hold the operand addresses of the decoded instructions. As each new instruction is decoded, an entry for the data buffer is created and the data prefetch is generated for the operand simultaneously. Hence, in ideal situations (far enough ahead) the data can be available by the time the instruction is actually executed. When a conditional branch is encountered, the system will randomly select a path, and instruction and data prefetch buffers continue the decoding and prefetching until the condition is evaluated. When an incorrect branch prediction is detected, the execution will stall waiting for the buffer to be flushed.

Implicit prefetching is present in decoupled architectures [Smith 82b], where two instruction streams operate concurrently, communicate via queues and drive two execution units: one for data access and one for functional operations. The data access stream can be ‘‘ahead’’ of the functional stream and hence can prefetch operands most likely needed in the near future.

The main problem with temporal prefetching is that the time window where the prefetching can occur is limited by the instruction decoding buffer size and is not wide enough for large memory latencies.

In summary, in spatial schemes, the opportune time at which prefetch should be initiated is not linked very closely to the time of next use, while in temporal schemes, the address of the data to be prefetched is based on the values of the speculated operands and is not

related to the current locality in the cache.

2.1.2 Software-directed Prefetching

A totally different approach to prefetching is to use software-directed techniques that rely on data access patterns detected by static program analysis. An intelligent compiler inserts data prefetch instructions several cycles before their corresponding memory instructions. A processor has to explicitly execute the prefetch instruction to initiate a prefetch request. Such prefetch instructions, which are just *hints* to the memory subsystem for reducing memory latency, are found in contemporary processors, such as ALPHA [DEC 92].

Porterfield [89] examines the effect of prefetching all array references in the most inner loops of programs by inserting prefetches one iteration ahead. Based on a simulation study of scientific programs, he finds that with good compile-time analysis, software prefetching is more effective than a simple prefetching through one cache block lookahead (OBL) or the use of a large cache line size. His results show that if the software prefetching were free, it could decrease the execution time of programs by up to 50%, with a decrease of over 20% on the average. However, he recognizes that the original ‘‘prefetch all’’ scheme may lead to too much overhead. The overhead can be reduced by selectively prefetching references that will be misses and by keeping the effective address in a register between the prefetch and the actual load.

Klaiber and Levy [91] show that the time to prefetch should depend on the memory latency and loop execution time. They propose a prefetching scheme which brings data into a separate fetch buffer instead of a unified cache. Chen *et al.* [91] examine compiler-assisted data prefetching in superscalar processors. Their results, based on fairly small caches, show that a prefetch buffer is more effective than increasing the cache dimension in solving the data pollution problem. Their study of software prefetching for non-scientific codes indicates that it is difficult to generate prefetch addresses early when the access patterns are irregular.

Gornish *et al.* [90] propose an algorithm, meant for parallel programs in multiprocessors, that finds the earliest point, before a loop that an entire subarray can be prefetched. The determination is based on the control and data dependencies in the program. The approach mainly focuses on asynchronous block transfers of data from global memory to local memory before the data are actually used, rather than on a single cache line at a time. Because they want the algorithm to be generic under any architecture and do not have a

specific coherence mechanism in mind, their approach is a binding prefetching, that is, a data block location is bound to the value of prefetched data in caches at the time that the prefetch completes. Based on a simulation of the execution of Fortran Kernels on a multiprocessor, they show that even a simple algorithm can reduce the processor stall time by as much as 32% to 97%. However, as prefetching is binding and subject to control and data dependence constraints at compile time, the restriction on the binding time may suppress significant prefetching candidates and limit the flexibility of prefetching.

Mowry and Gupta [91] propose a nonbinding software-controlled prefetching scheme. Prefetching is nonbinding in the sense that prefetched data are still kept coherent (by hardware) as if the data were fetched by normal operations. Unlike the algorithm by Gornish *et al.* [90], nonbinding prefetching provides the flexibility that prefetching can be done with fewer restrictions, but requires hardware-based coherence because the hardware must be dynamically aware of which data have been prefetched. They show (by manually inserting prefetch instructions) that prefetching can increase the performance by 83% to 86% for programs with regular data access patterns and by only 14% for a program with extensive use of linked lists. In addition to the study of a prefetch strategy, they consider both prefetch instructions for read and write accesses, and also data prefetches of size determined by the semantic data objects.

While software prefetching schemes discussed in the above are shown to be effective, most of the studies do not address the cost: issuing prefetching incurs instruction overhead and may increase the traffic in the memory system. Mowry and Gupta [92] further developed a compiler algorithm to perform the prefetch insertion. The new advance is to identify useful references for prefetching without introducing too much unnecessary overhead. Based on locality analysis and on loop transformations with proper prefetch predicates, the algorithm selectively inserts prefetches only for those references which are likely to cause cache misses. As the algorithm demonstrates locality analysis only for simple codes that operate on dense matrices, it is not clear that the compiler can automatically add useful prefetching for the programs with more complicated access patterns.

In summary, software-directed prefetching aims at compiler techniques to insert prefetches instructions without requiring too much hardware complexity. The two main issues are how the prefetch instructions are inserted and how their overhead can be reduced.

2.2 Non-blocking Caches

Non-blocking caches allow execution to proceed concurrently with one or more cache misses until necessary dependence on the missing data mandates stalling. In contrast to the overlap of computations *prior to* an actual cache miss by prefetching techniques, the non-blocking scheme exploits the overlap of memory accesses with *post*-miss computations. Although the design requires more complex hardware support in the processor, it can exploit instruction level parallelism without incurring extra overhead to the memory system.

Non-blocking caches (also called lockup-free caches) were originally proposed by Kroft [81] for uniprocessors. He introduced the concept of Miss Information/Status Holding Registers (MSHR) to keep track of multiple pending requests. Interestingly, the terms ‘‘lockup-free cache’’ and ‘‘non-blocking cache’’ are used interchangeably in the literature. We clarify the features by dividing them into two categories: (1) a cache supporting multiple outstanding memory requests, but blocking the processor on read misses (blocking loads), and (2) the processor supporting non-blocking loads and writes. In the following discussion, we refer to the first category as lockup-free caches and specifically refer to the latter as non-blocking reads/writes.

2.2.1 Lockup-free Caches

A cache design in the first category is generally used or studied for supporting advance architecture features in high-performance computers. The lockup-free cache in RP3 [Brantley *et al.* 85] supports non-blocking prefetches and multiple outstanding stores. However, reads are blocking until the missed datum returns. The lockup-free cache appearing in the studies of the DASH multiprocessor [Gharachorloo *et al.* 91a, Mowry & Gupta 91] allows multiple outstanding write and prefetch requests, while the processor still stalls on read misses. In principle, the capability of handling multiple pending requests is essential for prefetching or any other ‘‘buffering’’ and ‘‘pipelining’’ techniques that we have reviewed. Most studies on prefetching [Lee *et al.* 87b, Porterfield 89, Jouppi 90, Fu & Patel 91] and relaxed memory consistency models [Gharachorloo *et al.* 91b] simply assume pipelined caches, which also can be thought of as lockup-free caches.

2.2.2 Non-blocking Reads/Writes

In the second category, where out-of-order execution is allowed, the processor essentially requires extra hardware complexity to perform dynamic scheduling and support non-blocking loads. The memory requirements include a write buffer allowing load bypassing and a cache capable of servicing multiple requests. The SIMP architecture [Murakami *et al.* 89], which provides dynamic dependency resolution with speculative branch prediction, can take more advantage of lockup-free caches than a processor with MSHR's. Gharachorloo *et al.* [92] explore the advantages of relaxed consistency models in dynamically scheduled processors. With lockup-free caches associated with each processor, shared-memory multiprocessors have concurrently pending misses in the various processors. Scheurich and Dubois [91] discuss variations of cache coherence protocols for lockup-free caches of multiprocessors in weakly ordered systems. In their study, Sohi and Franklin [91] show that a multi-ported non-blocking cache at the first level can support the data bandwidth demands of an advance instruction issue mechanism. Stenstrom *et al.* [91] formulate access order information from programs, so that a lockup-free cache can exploit this information to achieve performance improvements. They present an implementation which can support and control pipelining among multiple accesses. In the IBM RS/6000 processor [Oehler & Groves 90], register tagging is implemented to allow data cache accesses to overlap with the execution of subsequent independent register-to-register instructions.

The schemes discussed in the above require fairly complex hardware. Performance can be improved by compiler assistance. Compiler optimizations for non-blocking loads mainly consist of instruction reordering and the insertion of independent instructions after non-blocking loads to keep the processor as busy as possible. Traditional instruction list schedulers [Gibbons & Muchnick 86, Krishnamurthy 90] can be employed to perform the code scheduling. More recently, Kerns and Eggers [92] proposed balanced scheduling, which is particularly suitable to non-blocking loads since the latency of a load is unknown until run time. Their key idea is to distribute loads according to the amount of instruction level parallelism that is available.

With a combination of hardware and software solutions, speculative loads fetch memory values into registers directly. Unlike normal non-blocking loads, they are speculatively executed so that unnecessary semantic page faults or dependencies will be ignored without introducing extra overhead. The advantage is that the compiler has more freedom to move

around the speculative loads, but they require extra detection and correction mechanism. Chen et. al. [92] study a design of preload register update, which allows the compiler to move load instructions even in the presence of data dependence. They derived the scheduling support for register preloads based on the superblock structure. Rogers and Li [90, 92] describe a hardware mechanism for non-fault speculative loads and compiler techniques to move across basic block boundaries.

In short, we have reviewed two related techniques for tolerating memory latency: data prefetching caches and non-blocking caches. The next chapter will present a new hardware-based data prefetching scheme.

Chapter 3

Data Prefetching Caches

Prefetching based on sequentiality has been shown be successful for the optimization of I-caches, but much less so for D-caches. In this chapter, we propose a hardware-based data prefetching scheme that overcomes the drawbacks of the previous approaches discussed in Section 2.1.1. The key idea of our scheme is to detect dynamically the strides and access patterns in a reference prediction table (RPT). Based on the timing of prefetching determination and issue, we propose three variations, in increasing order of complexity *generic*, *lookahead*, and *correlated*. The common basis of these schemes is to predict, based on data access patterns, the data stream far enough in advance, so that the required data can be *prefetched* and be in the cache when the ''real'' memory access instruction is executed. Our approach takes advantages of both spatial and temporal schemes.

In this chapter, we first give the basic motivation in Section 3.1. Then we describe the three schemes: generic, lookahead, and correlated in sections 3.2, 3.3, and 3.4 respectively. Finally, we summarize and briefly compare our scheme with other relevant approaches.

3.1 Motivation

The hardware supporting schemes generate prefetches by taking advantage of the regularity of memory access patterns when they exist and prevent prefetching when the access patterns are unpredictable.

Consider a program segment with m-nested loops indexed by $I_1, I_2, \cdots, I_m$. Let LP_{I_i} be the set of statements with data references in the loop at level i. Given a data reference r, we can divide the memory access patterns into four categories: *scalar*, *zero stride*, *constant stride*, and *irregular* as shown in Table 3.1.

The difference between *scalar* and *zero stride* is that the latter is a reference to a subscripted array element with the subscript being an invariant at the inner loop level but modifiable at an outer level. Obviously, standard caches work well for *scalar* and *zero stride* references. Caches with large block sizes and simple prefetch strategies (cf. Section 3.4)

Table 3.1: Classification of memory access patterns

Pattern	Description	examples
scalar	simple variable reference	index, count
zero stride	$r \in LP_{I_i}$ with subscript expression unchanged w.r.t I_i	A[I_1,I_2] in LP_{I_3} TAB[I_1].off in LP_{I_2}
constant stride	$r \in LP_{I_i}$ with subscript expression linear w.r.t I_i	A[I_1] in LP_{I_1} A[I_1,I_2],A[I_2,I_1] in LP_{I_2}
irregular	none of the above	A[B[I]] in LP_I A[I,I] in LP_I Linked List

can improve the performance for the *constant stride* category if the stride is small but will be of no help if the stride is large. Our goal is to generate prefetches in advance for uncached blocks in the *scalar*, *zero stride*, and *constant stride* access categories independently of the size of the stride. At the same time, we will avoid unnecessary prefetching for the *irregular* accesses. Our scheme will be most appropriate for high-performance processors with relatively small first-level caches, i.e., those such that they cannot hold the working set of the application, with a small block size, and running programs where the data access patterns are regular but not necessarily of stride 1.

The basis of our design is a Reference Prediction Table (RPT) that holds data access patterns of load/store instructions. To illustrate the concept, we consider the usual matrix multiplication loop (for more detail, see Section 3.2.3) and the pseudo-assembly RISC-like code version of the computational part of the inner loop shown in Figure 3.1. In the code we assume that the subscripts are kept in registers. At steady state, the RPT will contain entries for the three load *lw* and the store *sw* instructions. Since each iteration of the inner loop accesses the same location of A[i,j] (*zero stride*), no prefetch will be requested for it. Depending on the block size, references to B[i,k] (*constant stride*) will either be prefetched at every iteration (block size = 4), or every other iteration (block size = 8), and so on. Load references to C[k,j] (*constant stride* with a stride larger than the block size) will generate a prefetch instruction every iteration.

```
int A[100,100],B[100,100],C[100,100]
for i = 1 to 100
    for j = 1 to 100
        for k = 1 to 100
            A[i,j] += B[i,k] × C[k,j]
```

(a) A Matrix Multiplication

addr	instruction		comment	
500	lw	r4, 0(r2)	; load B[i,k]	stride 4 B
504	lw	r5, 0(r3)	; load C[k,j]	stride 400 B
508	mul	r6, r5, r4	; B[i,k] × C[k,j]	
512	lw	r7, 0(r1)	; load A[i,j]	stride 0
516	addu	r7, r7, r6	; +=	
520	sw	r7, 0(r1)	; store A[i,j]	stride 0
524	addu	r2, r2, 4	; ref B[i,k]	
528	addu	r3, r3, 400	; ref C[k,j]	
532	addu	r11, r11, 1	; increase k	
536	bne	r11, r13, 500	; loop	

(b) assembly code

Figure 3.1: Example of Matrix Multiplication

3.2 Generic Reference Prediction

The most intuitive prediction scheme is to have prefetches for the $(i + 1)^{st}$ iteration be generated when the ith iteration is executed. Thus, when the program counter (PC) decodes a load/store instruction, a check is made to see if there is an entry corresponding to the instruction in the RPT. If not, it is entered. If it is there and if the reference for the next iteration is predictable (as defined below), a prefetch is issued. This *generic* scheme involves only the PC and the RPT. As shown in Figure 3.4, the hardware requirement for the *generic* design is a subset of the more complex *lookahead* variation that will be

described in Section 3.3. We now introduce the design and use of the RPT under the *generic* scheme.

3.2.1 Reference Prediction Table - RPT

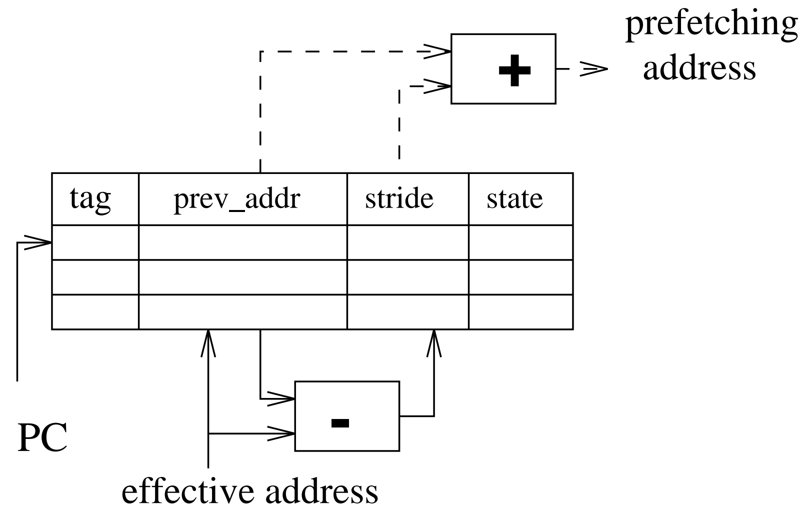
The Reference Prediction Table (RPT) is used to keep track of previous reference addresses and associated strides for load and store instructions. The RPT is organized as a cache. Each RPT entry has the following format (see Figure 3.2):

- *tag:* corresponds to the address of the Load/Store instruction
- *prev_addr:* the last (operand) address that was referenced when the PC reached that instruction.
- *stride:* the difference between the last two addresses that were generated.
- *state:* a two-bit encoding (4 states) of the past history; it indicates how further prefetching should be generated. The four states are:
  - *initial:* set at first entry in the RPT or after the entry experienced an incorrect prediction from *steady* state.
  - *transient:* corresponds to the case when the system is not sure whether the previous prediction was good or not. The new stride will be obtained by subtracting the previous address from the currently referenced address.
  - *steady:* indicates that the prediction should be stable for a while.
  - *no prediction:* disables the prefetching for this entry for the time being.

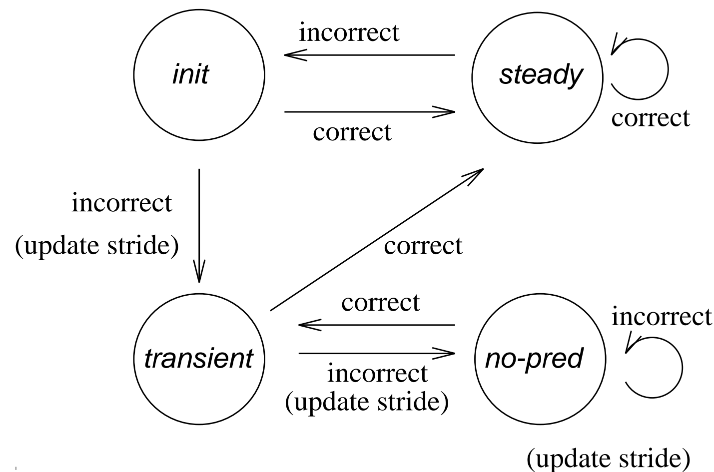
3.2.2 RPT mechanism

When the PC encounters a load/store instruction with effective operand address *addr*, the RPT is updated as follows: (To make it clear, we denote *correct* by the condition: $addr = (prev_addr + stride)$ and *incorrect* by the condition: $addr \neq (prev_addr + stride)$.)

- A.1. There is no corresponding entry. The instruction is entered in the RPT, the *prev_addr* field is set to *addr*, the *stride* to 0, and the *state* to *initial*.



(a) reference prediction table



(b) state transition by PC

Figure 3.2: Reference Prediction

- A.2. There is a corresponding entry. Then:

  (a) Transition --

  When *incorrect* and *state = initial*:

  Set *prev_addr* to *addr*, *stride* to *(addr - prev_addr)*, and *state* to *transient*.

  (b) Moving to/being in steady state --

When *correct* and (*state* = *initial, transient,* or *steady*):

Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *steady*.

(c) Steady state is over; back to initialization --

When *incorrect* and *state* = *steady*:

Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *initial*.

(d) Detection of irregular pattern --

When *incorrect* and *state* = *transient*:

Set *prev_addr* to *addr*, *stride* to *(addr - prev_addr)*, and *state* to *no prediction*.

(e) No prediction state is over; back to transient

When *correct* and *state* = *no prediction*:

Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *transient*.

(f) Irregular pattern --

When *incorrect* and *state* = *no prediction*:

Set *prev_addr* to *addr*, *stride* to *(addr - prev_addr)*, and leave *state* unchanged.

Following the update, a prefetch request can be generated based on the presence and state of the entry. Note that the generation of a prefetch does not block the execution of the instruction stream and, in particular, the increment of the PC. There are two mutually exclusive possibilities:

- B.1. No action.

  There is no existing entry or the entry is in state *no prediction*.

- B.2. Potential prefetch.

  There is an entry in *init*, *transient*, or *steady* state. A data block address *(prev_addr + stride)* is generated. If the block is uncached and the address is not found in an Outstanding Request List (ORL) (see Section 3.3), a prefetch is initiated. This implies sending a request to the next level of the memory hierarchy, or buffering it if the communication channel is busy. The address of the request is entered in the ORL.

3.2.3 Example and Discussion

Figure 3.3 illustrates how the Reference Prediction Table is filled and used when the inner loop of the matrix multiplication code shown previously is executed. We restrict our example to the handling of the 3 load instructions at addresses 500, 504, and 512. We assume that the base addresses of matrices A, B and C are respectively at locations 10,000, 50,000, and 90,000.

tag	prev_addr	stride	state

Initially empty

(a)

tag	prev_addr	stride	state
500	50,000	0	*init*
504	90,000	0	*init*
512	10,000	0	*init*

After iteration 1

(b)

tag	prev_addr	stride	state
500	50,004	4	*transient*
504	90,400	400	*transient*
512	10,000	0	*steady*

After iteration 2

(c)

tag	prev_addr	stride	state
500	50,008	4	*steady*
504	90,800	400	*steady*
512	10,000	0	*steady*

After iteration 3

(d)

Figure 3.3: Example: Filling RPT entries

Before the start of the first iteration, the RPT can be considered empty since there won't be any entry corresponding to addresses 500, 504, and 512 (cf. Figure 3.3.a). Let us assume also that no element of A, B, or C has been cached.

When the PC executes for the first time the load instruction at address 500, there is no corresponding entry. Therefore, the instruction is entered in RPT with its *tag* (500), the *prev_addr* field set to the address of the operand, i.e., 50,000, the *stride* set to 0, and the *state* to *initial* (cf. A.1 above). Similar actions are taken for the other two load instructions (cf. Figure 3.3.b). In all three cases, there will be cache misses and no prefetches.

When the PC executes the load instruction at address 500 at the beginning of the second

iteration, we are in the situation described as ‘‘transition’’ (cf. A.2.a). The following three actions are taken:

1. Normal reference access to address 50,004. This results in a cache hit if the block size is larger than 4 and in a miss otherwise.

2. Update of the entry in the RPT. The *prev_addr* field becomes 50,004, the *stride* is set to 4, and the *state* to *transient*.

3. Potential prefetch of the block at address (50,004 + 4) = 50,008. A prefetch occurs if the block size is less than 8.

Similar actions take place for the load at address 504 with, in this case, the certainty that a prefetch will be generated (cf. Figure 3.3.c). For the load at instruction 512, we are in the situation ‘‘moving to steady state’’ (cf. A.2.b). The *prev_addr* and *stride* fields are unchanged and the *state* becomes *steady*. Of course, we have a cache hit and no prefetch.

During the third iteration, all three loads should result in cache hits, or in indications that prefetches for the referenced items are in progress. The RPT entries are updated as shown in Figure 3.3.d (note the *transient* to *steady* transitions); prefetches are generated for blocks at addresses 50,012 (if needed) and 91,200. Subsequent iterations follow the same pattern.

As can be observed in Figure 3.2, *scalar* and *zero stride* references will pass from *initial* to *steady* state in one transition (instruction 512). The *constant stride* references will pass through the *transient* state to ‘‘obtain’’ the stride and then stay in *steady* state (instructions 500 and 504). References with two wrong predictions in a row (not shown in the example) will be prevented from being prefetched by passing to the *no prediction* state; they could re-enter the *transient* state, provided that the reference addresses become predictable. For instance, accesses to elements of a triangular matrix may follow such a pattern. Note that the *stride* field is not updated in the transition from *steady* to *initial* when there is an incorrect prediction.

3.3 Lookahead Reference Prediction

The *generic* scheme has a potential weakness associated with the timing of the prefetch. If the loop body is too small, the prefetched data may arrive too late for the next access,

and if the loop body is too large, an early arrival of prefetched data may replace (or be replaced by) other useful blocks before the data is actually used. The *lookahead* reference prediction scheme seeks to remedy this drawback.

An ideal time to issue a prefetch request is to perform the prefetch δ cycles ahead of the actual use, where δ is the latency to access the next level in the memory hierarchy. The *lookahead* prediction will approximate this ideal prefetch time with the help of a pseudo program counter, called the Look-Ahead Program Counter (LA-PC), that will remain as much as possible δ cycles ahead of the regular PC and that will access the RPT in order to generate prefetches. The LA-PC is incremented as the regular PC. It is used in conjunction with a Branch Prediction Table (BPT) to take full advantage of the *lookahead* feature.

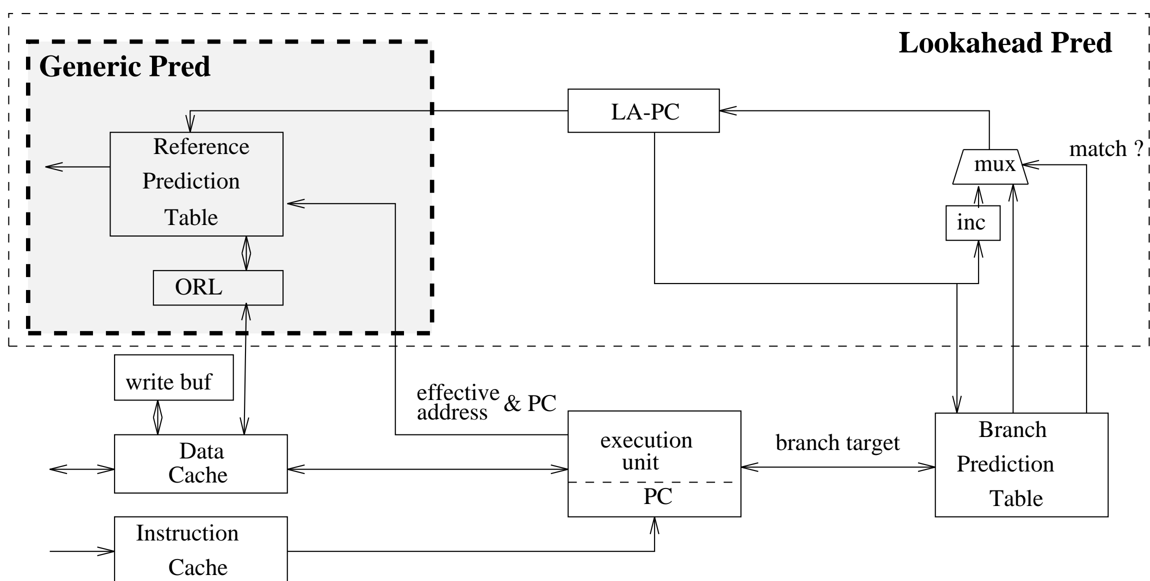


Figure 3.4: Block diagram of data prefetching

An overall block diagram of the target processor is shown in Figure 3.4. The bottom part of the figure abstracts a common high-performance processor with on-chip data and instruction caches. The upper-left part shows the Reference Prediction Table (RPT) and the Outstanding Request List (ORL) that keeps track of the addresses in progress or outstanding requests. Under the *generic* scheme, the RPT is accessed by the PC. In order to implement the *lookahead* mechanism, a Look-Ahead Program Counter (LA-PC) and its associated logic are added to the top part (on the right in the figure). The LA-PC is a secondary PC used to predict the execution stream. In addition, we assume that a Branch Prediction Table (BPT) such as branch target buffer (BTB), a branch prediction mechanism for the PC in a

high-performance processor, is used for modifying the LA-PC.

Entries in the RPT and BPT, are initialized and updated when the PC encounters the corresponding instruction. In the *lookahead* scheme, in contrast to the *generic* prediction, it is the LA-PC rather than the PC that is used to generate potential prefetches according to rules B.1 and B.2 of the previous section. At each cycle, the LA-PC is simply incremented by one. When the LA-PC finds an entry in the BPT, it indicates that the LA-PC points to a branch instruction. In that case, the prediction result of the branch entry in the BPT is provided to modify the LA-PC. Note that, unlike the instruction prefetch structure in [Lee *et al.* 87a] or decoupled architectures[Smith 82b], the system does not need to decode the predicted instruction stream. Instead, the lookahead mechanism is based on the history information of the execution stream.

3.3.1 Lookahead Program Counter (LA-PC) and RPT

In the *generic* prediction scheme, prefetching can occur only one iteration ahead and thus, as mentioned earlier, the prefetched data might not yet be in the cache there when the real access takes place. This situation will occur when the loop iteration time is smaller than the memory latency. With the help of the *lookahead* mechanism, the LA-PC may wrap around the loop and revisit the same data instruction when the execution time of a loop iteration is smaller than the memory latency. In this way, we may have multiple iterations lookahead.

An extra field (*times*) in the entries of the RPT will keep track of how many iterations the LA-PC is ahead of the PC (cf. Figure 3.5). Another difference in the design of the RPT with respect to the *generic* RPT (cf. Figure 3.2) is that it is the LA-PC that is used to detect and to generate prefetch requests while the PC is still used to access the RPT when an effective address is obtained. Now, when the LA-PC hits an instruction with a corresponding entry in the RPT, the address of a potential prefetch is determined by computing $(prev_addr + stride \times times)$. The *times* field is incremented whenever the LA-PC hits the entry, while it is decremented when the PC catches up with the entry. The *times* field is reset when it is found that the reference prediction of the corresponding entry is incorrect.

3.3.2 Lookahead Distance and Limit

The ideal Look Ahead distance (LA-distance), i.e., the time between the execution of the instruction pointed to by the PC and that of the instruction pointed to by the LA-PC, is

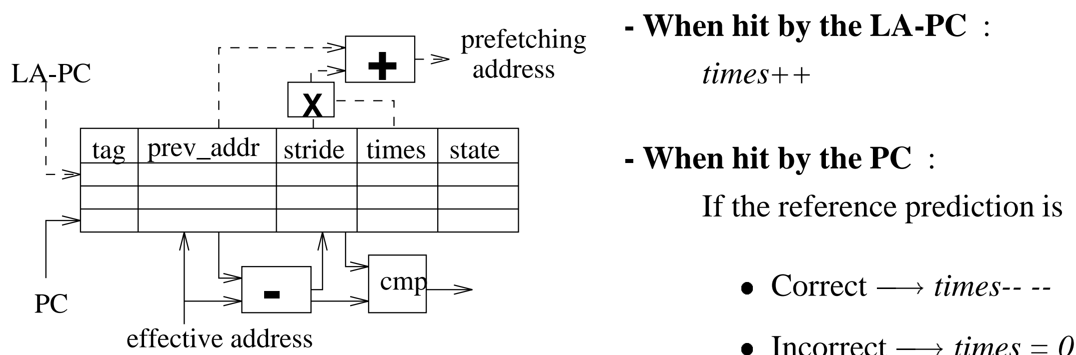


Figure 3.5: RPT with Lookahead mechanism

equal to the latency δ of the next level in the memory hierarchy. Clearly this can only be approximated, since the LA-distance is variable. Initially, and after each wrong branch prediction, the LA-distance will be set to one, i.e., the LA-PC points to the instruction following the current PC. When a real cache miss occurs or when a prefetch is not completed by the time the data is actually needed, the current execution is stalled, i.e., the value of PC does not change, while the LA-PC can still move ahead and generate new requests (recall the role of the ORL).

As shown in Figure 3.4, the LA-PC is maintained with the help of a branch prediction mechanism BPT. BPT designs have been thoroughly investigated [Lee & Smith 84, Perleberg & Smith 89] and we will not repeat these studies here. In our experiments we use the Branch Target Buffer (BTB) with two-bit state transition design described in [Lee & Smith 84] and we assume that the BTB has been implemented in the core processor for other purposes.

As the LA-distance increases, the data prefetch can be issued early enough so that the memory latency can be completely hidden. However, the further PC and LA-PC are apart, the more likely the prediction of the execution stream will be incorrect because the LA-distance is likely to cross over more than one basic block. Moreover, we don't want some of the prefetched data to be cached too early and displace other needed data. Therefore, we introduce a system parameter called Look Ahead Limit (LA-limit d) to specify the maximum distance between PC and LA-PC. Thus, the LA-PC is stalled (until the normal execution is resumed) in the following situations: (1) The LA-distance reaches the specific limit d, or (2) the ORL is full.

3.3.3 Handling Cache Misses

On a cache read miss, the cache controller checks the ORL. If the block has already been requested, a ‘‘normal’’ (but less lengthy) stall occurs. (We call *hit-wait cycles* those cycles during which the CPU waits for the prefetched block to be in the cache.) Otherwise, a regular load is issued with priority over the buffered prefetch requests.

Since we are using a write-back, write-allocate strategy, a write miss in the data cache will cause the system to fetch the data block and then update the desired word. If the block size is larger than a single word, we can initiate prefetching as for a read miss. When the block size is one word, no prefetch needs to be issued but a check of the ORL is needed for consistency purposes. In case of a match, the entry in the ORL must be tagged with a discard status so that the data will be ignored when it arrives.

When the LA-PC has to be reset because of an incorrect branch prediction, the buffered prefetch requests are flushed. Finally, when a prefetch raises an exception (e.g., page fault, out-of-range violation) we ignore the prefetch. The drawbacks of a wrong page fault prediction would far outweigh the small benefits of a correct prefetch.

3.4 Correlated Reference Prediction

In the previous two designs, reference prediction was based on the regularity between *adjacent* data accesses. In general, the schemes work well for predicting references in inner loops. However, the results are less significant for those execution segments with small inner-loop bodies or triangle-shaped loop patterns because of the frequent stride change in the outer iterations. For example, let us look at Livermore Kernel Loop 6 in Figure 3.6.

While executing the inner loop, accesses to the B matrix have regular strides (e.g., B[3,0], B[3,1], B[3,2] and B[3,3] have a *stride* of 4). This pattern will be picked up by the two schemes presented above. However, an incorrect prediction will occur each time the *k* loop is finished, e.g., when accessing B[4,0] after B[3,3]. We can observe though that there is a correlation between the accesses due to the termination of the inner loop (i.e., B[1,0], B[2,0], B[3,0] etc. have a *stride* of 400). Correlation has led to the design of more accurate branch prediction [Pan *et al.* 92, Yeh & Patt 92] and can be equally applied to data reference prediction.

The key idea behind *correlated* reference prediction thus is to keep track not only of those *adjacent* accesses in inner loops (as in the above two schemes) but also of those

```
        int B[100,100], W[100]
        DO 6 i=1,n
        DO 6 k=0,i
              W(i) = W(i) + B(i,k)*W(i-k)
6       CONTINUE
```

1,0	1,1			
2,0	2,1	2,2		
3,0	3,1	3,2	3,3	
4,0	4,1	4,2	4,3	4,4

(a) Code (b) Access pattern of Matrix B

Figure 3.6: Kernel 6

correlated by changes in the loop level. Since branches in the inner loop are taken until the last iteration, a non-taken branch will trigger the correlation to the next level up.

3.4.1 Implementation of Correlated RPT

The implementation of a *correlated* scheme would bring two additions to the *lookahead* mechanism: a shift register to record the outcome of the last branches and an extended RPT with separate fields for computing the strides of the various correlated accesses. In the most general case, an *N-bit* shift register can be used to keep track of the results of the last *N* branches and to serve as a mask to address the various fields in the extended RPT.

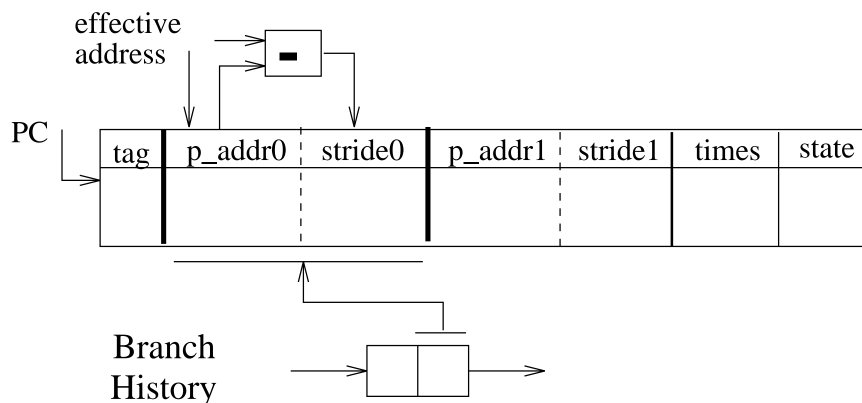


Figure 3.7: Correlated RPT

Since prefetching too far in advance might be detrimental, we restrict ourselves to the

correlation in two-level nested loops. The RPT is extended (cf. Figure 3.7), with a second pair of *prev_addr* and *stride* fields for recording the access patterns of the outer loop. (Note that at the outer loop level the *times* and *state* fields are no longer relevant.) We also have now a two-bit shift register for recording the outcome of loop-only branches. Assuming that a bit ‘1’ encodes a taken branch, the steady state encoding while executing the inner loop will be ‘11’. In that case prefetching will be based on the entry in the RPT corresponding to the inner loop (the ‘‘right’’ part in the figure). When the branch is not taken, the shift register will contain ‘10’ (due to a non-taken inner-loop branch followed by a taken outer-loop branch) and prefetching will be based on the part of the entry corresponding to the outer loop (the ‘‘left’’ part). Updating of the left *prev_addr*0 and *stride*0, as well as the right *prev_addr*1, fields will take place at the beginning of an outer iteration (when the shift register contains ‘10’ or ‘00’) while all the right fields will be updated for consecutive inner iterations (when the register contains ‘11’ or ‘01’).

Figure 3.8 shows how the RPT entry for B[i,k] would be filled and updated during execution of the first three iterations of the outer loop of Kernel Loop 6. We have left out the *times* field for ease of explanation. Without loss of generality, we can assume that the initial content of the shift register is ‘10’ and that the entry in the RPT is empty. At the initial access of B[1,0], all fields are filled as in the previous schemes (first row of top table in Figure 3.8). At the second access (first row of bottom table) only the right fields are modified as they would be in the previous schemes (the shift register contains ‘11’). At the beginning of the second outer iteration, i.e., first access to B[2,0], the shift register will again contain ‘10’ (branch non-taken). Thus, prefetching of B[3,0] and B[2,1], if needed, will be done for accesses in both levels of the loop, and updating of the first pair of fields and of *prev_addr*1 will be performed (second row of top table). On subsequent accesses to B[2,i], prefetching and updating will be based on the right fields (second row of bottom table). At the beginning of the third outer iteration, we are in steady state (cf. last row of the table). By that time B[3,0] should have been prefetched (at the end of the second iteration). A prefetch to B[3,1] would be generated and most likely not activated if the line size is large enough, i.e., if B[3,0] and B[3,1] are in the same line.

Three issues regarding the implementation of a correlated reference prediction scheme need to be addressed. In all three cases, we make reasonable assumptions to keep a design as simple as possible. The first assumption is that since it is easy for a compiler to distinguish between end of loop branches and other branches, the former will be flagged,

1st inner iteration

outer iteration	prev addr0	stride0	prev addr1	stride1	state
1	B[1,0]	0	B[1,0]	0	*init*
2	B[2,0]	400	B[2,0]	4	*transient*
3	B[3,0]	400	B[3,0]	4	*steady*

2nd inner iteration

outer iteration	prev addr0	stride0	prev addr1	stride1	state
1	B[1,0]	0	B[1,1]	4	*transient*
2	B[2,0]	400	B[2,1]	4	*steady*
3	B[3,0]	400	B[3,1]	4	*steady*

Figure 3.8: Example: Correlated RPT entries

e.g., in the branch prediction table, and the shift register will be modified only when they are encountered. This assumption could be removed by letting the shift register be modified on every branch as in [Pan *et al.* 92]. While some predictable patterns might emerge, it is not evident that the complexity of implementation is warranted. Second, we assume that the loop iterations are controlled by backward branches[1]. Third, we assume that prefetches for the correlated references (across outer iterations) are issued as in the *generic* case, since the LA-PC will be the same as the PC on the incorrect branch prediction. Those prefetches are generated without any attempt to control the prefetch issue δ cycles ahead of the actual use of the data. The assumption is quite reasonable since accesses in the outer loop are separated by the execution of all the iterations of the inner loop which, in most likelihood, will take longer than δ cycles.

---

[1] A backward branch always passes execution control to a location which is before the address of the branch instruction. The compiler could easily perform a transformation resulting in backward branches for these programs in which forwards conditional branches are used to control the loop iterations [Ball & Larus 93].

3.5 Summary

In this chapter we have proposed a design for a hardware-based prefetching scheme. The goal of this support unit is to reduce the CPI contribution associated with data cache misses. The basic idea of data prefetching is to predict future data addresses by keeping track of past data access patterns in a Reference Prediction Table. Based on the various times when a prefetch is issued, we have investigated three variations: generic, lookahead, and correlated prefetching schemes.

The basic mechanism is that access instructions are recorded in the RPT associated with a finite state mechanism to prevent unnecessary prefetches. The generic scheme generates prefetches for the $(i+1)^{st}$ iteration when the ith iteration is executed. The lookahead scheme controls prefetches one memory latency time ahead of actual use by a lookahead program counter. It can have prefetches issued multiple iterations in advance. Finally, the correlated scheme uses a more sophisticated design to detect patterns across loop levels.

The prefetching hardware support unit that we advocate is designed to be close to the processor without introducing extra gate delays to the critical path. The performance evaluation in the next chapter will show that this design is effective and applicable to a chip with limited area. For example, it can be a support unit for an on-chip cache.

It is worthwhile to close this chapter with a brief qualitative comparison of our design with the two closest approaches [Fu & Patel 92, Sklenar 92] that were reviewed in Section 2.1.1. Fu and Patel [92] present a mechanism which will generate a prefetch request based on two consecutive accesses by adding the current effective address with the difference between the current and previous addresses. They do not provide a mechanism to prevent unnecessary prefetches. Severe data pollution and memory saturation problems could occur when memory bandwidth is limited. Since there is no state transition mechanism, this scheme corresponds to a degenerated version of our *generic* scheme. Sklenar [92] gives a design for a prefetch unit, which behaves like a co-processor. The prefetch processor calculates the effective addresses and generates prefetch requests. Although these two approaches were proposed later than our original scheme [Baer & Chen 91], they apparently lack two key ingredients: (1) the prefetch for the next iteration is generated on current access, and therefore cannot prefetch more than one iteration ahead, and (2) they do not have a lookahead mechanism to approximately control the arrival time of prefetched blocks.

Chapter 4

Performance Evaluation of Hardware Prefetching Schemes

In this chapter, we evaluate the three proposed hardware schemes and we examine several design issues. Our study is performed at the level of instruction simulation in the context of a RISC uniprocessor model. First, in Section 4.1, we describe the methodology for evaluating the proposed schemes in an uniprocessor environment. Since memory bandwidth is one of the vital resources if prefetching is to succeed, we consider three memory models with varying capability for handling concurrent memory requests. We use ‘‘cycle per instruction contributed by memory access’’ (MCPI) as the main metric instead of miss ratio in order to reflect results in a more realistic manner. Then, we present and discuss the general results of the simulation on ten SPEC benchmarks in Section 4.2. To better understand the characteristics of the schemes, we examine several design issues in Section 4.3, including the effect of memory latencies, block sizes, and the organization of the reference prediction table. We also consider various placements for the prefetched data.

4.1 Evaluation Methodology for Uniprocessors

4.1.1 Trace driven Simulation

We evaluate our proposed architectures using cycle-by-cycle trace generation combined with on-the-fly simulation. To avoid the overhead of re-running the trace generation for every configuration, we simulate several configurations simultaneously. As illustrated in Figure 4.1, the simulator, written in C++, first reads in a variety of configuration descriptions and creates simulation objects (which models most aspects of the CPU and the memory system) initialized with various configurations and parameters (such as cache size and latency time).

Benchmarks are instrumented on a DECstation 5000 (R3000 MIPS CPU) using the *pixie* facility. As shown in Figure 4.1, the simulator runs the *pixified* benchmark programs, which will generate address traces at the same time. The simulator reads the trace through

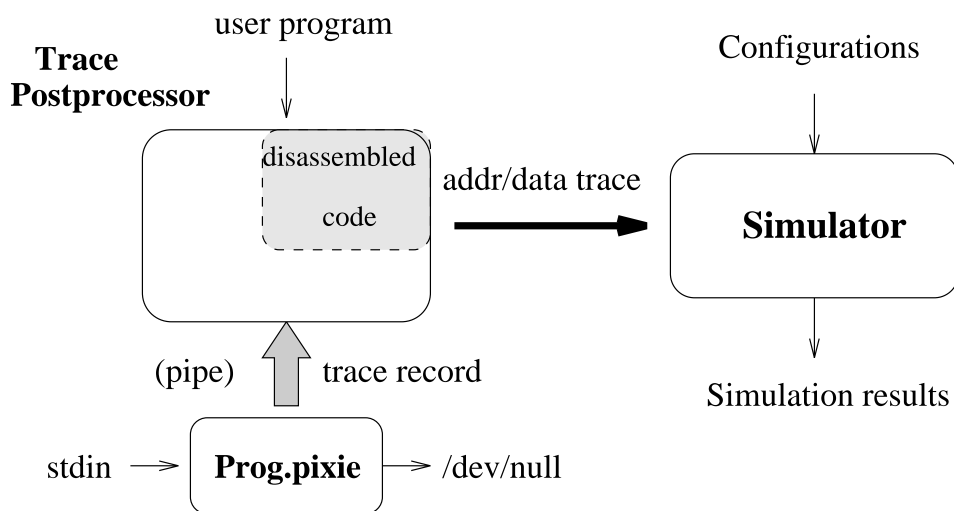


Figure 4.1: Trace-driven simulator using *pixie*

a *pipe* facility and feeds the trace records to each simulation object. This on-the-fly trace simulation method is advantageous in the sense that large traces do not need to be saved and furthermore it provides the flexibility of simulating rescheduled code. Traces include data and instruction references so that the simulator can emulate the detail behavior of overlapping computation with data access. The experiment results are collected at the clock cycle level from the individual configurations.

We use ten SPEC [1] benchmarks (see Section 4.1.3) to generate traces for our study. The traces captured at the beginning of the execution phase of the benchmarks are discarded because they are traces of initial routines that generate the test data for the benchmarks. No statistical data are recorded while the system simulates the first 500,000 data accesses. However, these references are used to fill up the cache, the Branch Prediction Table, and the Reference Prediction Table in order to simulate a warm start. After the initialization phase and the warm-start period, simulations results are collected for the first 100 million instructions for all programs.

4.1.2 Architectural Models

For comparison purposes, a baseline architecture consisting of a processor with perfect pipelining and direct-mapped D-cache with 32K bytes and a block size of 32 bytes,

[1] SPEC is a trademark of the Standard Performance Evaluation Corporation.

unless otherwise specified, is also simulated. The baseline and prefetching caches use a write-back, write-allocate policy, and an 8-entry write-back buffer for replaced dirty data lines. We assume that the processor has an ideal instruction cache with no instruction cache miss incurring. The RPT we use is a 512-entry table organized as shown in Figure 3.2. When the schemes with lookahead are evaluated, branch predictions are performed by a Branch Target Buffer with a two-bit state transition design [Lee & Smith 84]. Both baseline and prefetching caches will cause the processor to stall on a cache miss.

The interface between the processor and the cache can handle one data access at each cycle and, in case of a hit, the load latency is one cycle (i.e., delayed load with one delay slot). In the case of a write hit, an extra cycle is required to modify the data block in the cache. The refilling of a prefetched line will be delayed when it competes with real data accesses for the cache. Also, real cache misses could conflict with prefetch or outstanding write requests in the cache interface. We will assume, conservatively, that a fetch in progress cannot be aborted. However, a real read miss will be given priority over buffered prefetch requests or writes.

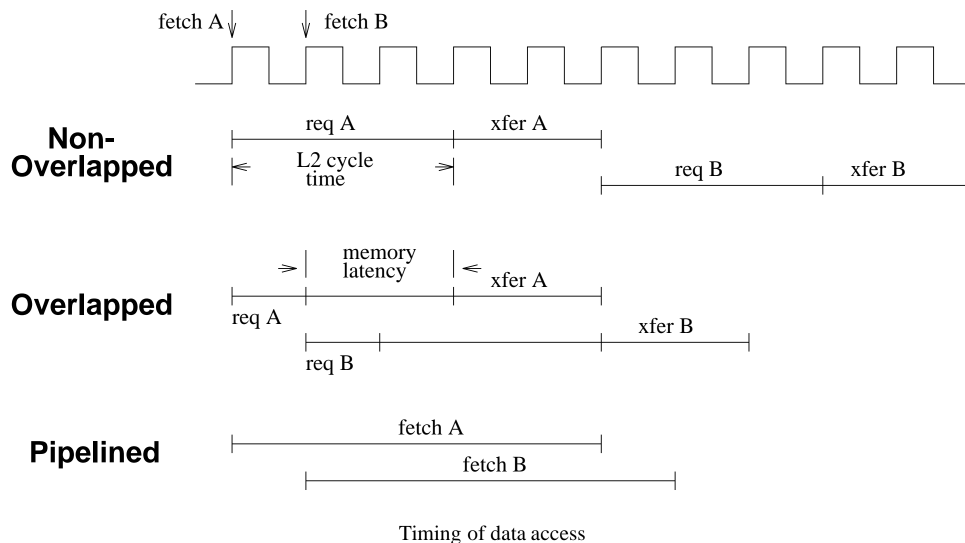
Data bandwidth is an important consideration in the design of an architecture that allows overlap of computation and data accesses since several memory requests (e.g., cache misses, prefetching requests) can be present simultaneously. In this study, we present three memory model interfaces with increasing capabilities of concurrency between caches and the next level in the memory hierarchy. Since several requests can be present, either in process or waiting to be processed, we have associated an Outstanding Request List (ORL) with the prefetching caches. A requirement for this list is that it can be searched associatively.

The three memory interfaces are as follows (cf. Figure 4.2 for timing charts and block diagrams).

- **Non-overlapped(1)** : As soon as a request is sent to the next level, no other request can be initiated until the (sole) request in progress is completed. This model is typical of an on-chip cache backed up by a second level cache.

  This interface supports only one cache request at a time.

- **Overlapped(C,N)** : The access time for a memory request can be decomposed into three parts: request issue cycle, memory latency, and transfer cycles. We assume that during the period of memory latency other data requests can be in their request



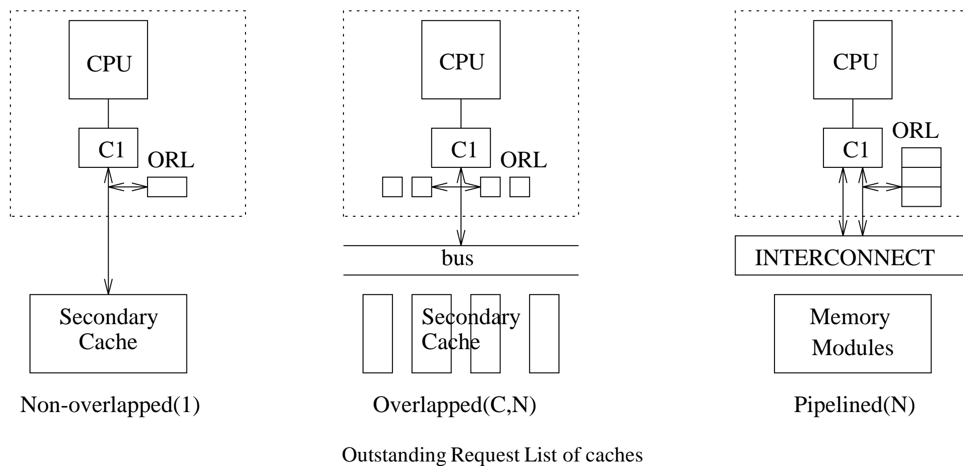


Figure 4.2: Three memory models

issue or transfer phases. However, no more than one request issue or transfer can take place at the same time.

This model represents split busses and a bank of C interleaved memory modules or secondary caches. An ORL with N entries is associated with each module.

- **Pipelined(N)** : A request can be issued at every cycle. This model is representative of processor-cache pairs being linked to memory modules through a pipelined packet-switched interconnection network. We assume a *load through*

mechanism[Smith 82a], i.e., the desired word is available as soon as the first data response arrives. An *N*-entry ORL is associated with the cache.

The configurations of the ORLs used in our experiment are **Non-overlapped(1)**, **Overlapped(8,2)**, and **Pipelined(8)** respectively. The memory latency δ is usually equal to 30. The *Overlapped* model is used as the default model to show general results since it is the most likely implementation for future high performance processors. The cycle times of the three phases (requesting, accessing memory, and transferring) are 2, 20, and 8 cycles respectively.

4.1.3 Benchmarks and Metrics

As mentioned earlier, we use ten applications from the SPEC benchmarks, which are compiled by the MIPS C compiler and the MIPS F77 compiler, both with optimization options. Table 4.1 shows the dynamic characteristics of the workload. The columns below *data references* show the proportions of data references (weighted by their frequency) that belong to the memory access categories mentioned in Section 3.1. They are one indication of the reference predictability of the ten programs. Scalar or zero stride references are beneficial to the data cache and, in addition, the prefetching schemes can be useful in bringing back in advance blocks that were displaced because of a small cache size (capacity misses) or a small associativity (conflict misses). Constant stride references, which may substantially contribute to cache misses, should be helped by the RPT schemes. Prefetching should be avoided for unpredictable irregular references. The column *branch prediction miss ratio* shows the outcome of branch predictions with a 512-entry BPT, which functions like a 2-state-bit Branch Target Buffer[Lee & Smith 84]. This is a second indication of the reference predictability, illustrating the possible benefits exploited by the lookahead approach.

We experimented with the three architectural choices, and varying architectural parameters, described previously. The results of the experiments are presented in terms of ‘‘cycle per instruction contributed by memory accesses’’ (MCPI) as the main metric. Since we assume that the processor can execute each instruction in one cycle (perfect pipelining) and that we have an ideal instruction cache, the only extra contribution of CPI is due to the data access penalty. Hence, the MCPI due to data access penalty is obtained as:

$$MCPI_{data\ access} = \frac{total\ data\ access\ penalty}{number\ of\ instructions\ executed}$$

Table 4.1: Characteristics of benchmarks

Name	data references			branch pred. miss ratio
	scalar, zero stride	constant stride	irregular	
Tomcatv	0.312	0.682	0.006	0.005
Fpppp	0.981	0.006	0.014	0.110
Matrix	0.059	0.921	0.021	0.073
Spice	0.581	0.239	0.180	0.060
Doduc	0.692	0.154	0.154	0.120
Nasa	0.006	0.989	0.003	0.008
Eqntott	0.338	0.574	0.088	0.069
Espresso	0.460	0.424	0.116	0.055
Gcc	0.516	0.120	0.365	0.204
Xlisp	0.440	0.078	0.482	0.156

The reason for choosing MCPI as a metric instead of the miss rate or the average effective access time is that MCPI can reflect the actual stall time observed by the processor, taking both processor execution and cache behavior into account. In the figures, we also give the percentage of the data access penalty reduced by the prefetching scheme. This percentage number is computed as:

$$\%\text{ of penalty reduced} = \frac{\text{data penalty}_{cache} - \text{data penalty}_{prefetch}}{\text{data penalty}_{cache}} \times 100$$

4.2 General Results

In this section, we present experimental results that show the benefits of the prefetching schemes. We compare an architecture with a baseline cache with the same architecture augmented by each of the three prefetching schemes. These comparisons are performed on all ten SPEC benchmarks.

Figure 4.3(a)(b)(c) shows the results of the simulation of the four architectures with the data access penalty MCPI as a function of the cache size. The *Overlapped* memory model is used, the block size is 32 bytes, and the RPT and the BPT used in the prefetching schemes have 512 entries. The results show that the prefetching organizations always perform better

than the pure cache scheme since they have the same amount of cache and, in addition, the prefetching component. When the cache is too small to contain the working set of the application, the best prefetching scheme can reduce the data access penalty from 16% up to 97%. The additional cost paid for prefetching is justified by the significant performance improvement. This additional cost (RPT and logic) is approximately equivalent to a 4K-byte D-cache (cf. Section 4.3.3).

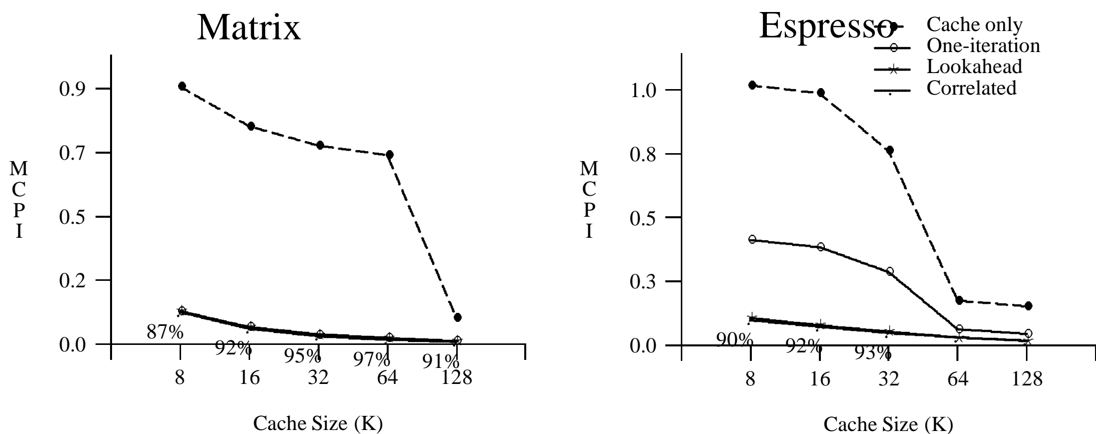


Figure 4.3: Simulation Results for $\delta = 30$ -- *Overlapped*

We examine further the performance curves by dividing the ten benchmarks into three groups: 1) prefetching performs extremely well, 2) prefetching yields a good or moderate improvement to the performance, and 3) prefetching's contribution to the reduction in data access penalty is slight.

The first group is formed by the Matrix and Espresso benchmarks, in which the data access penalty has been reduced by over 90%. For all practical purposes, the CPI due to data access is almost completely eliminated. With good reference predictability in these two programs, the flat performance curves of the prefetching illustrate that a cache of small size is sufficient to capture most of the locality when compulsory cache misses have been eliminated by the prefetching.

The second group includes Tomcatv, Nasa, Eqntott, and Xlisp in which the prefetching yields a good performance improvement, a reduction in data access penalty in the range of 35% to 70%. In the case of Eqntott and Xlisp, the MCPI penalty is about a quarter of a cycle even for a very small cache. Seeking further improvement is not worthwhile. In Tomcatv and Nasa, as the cache size increases, the miss penalty of both the pure and prefetching caches is minimized until the working set is captured (e.g., 32 K for Tomcatv). Notice

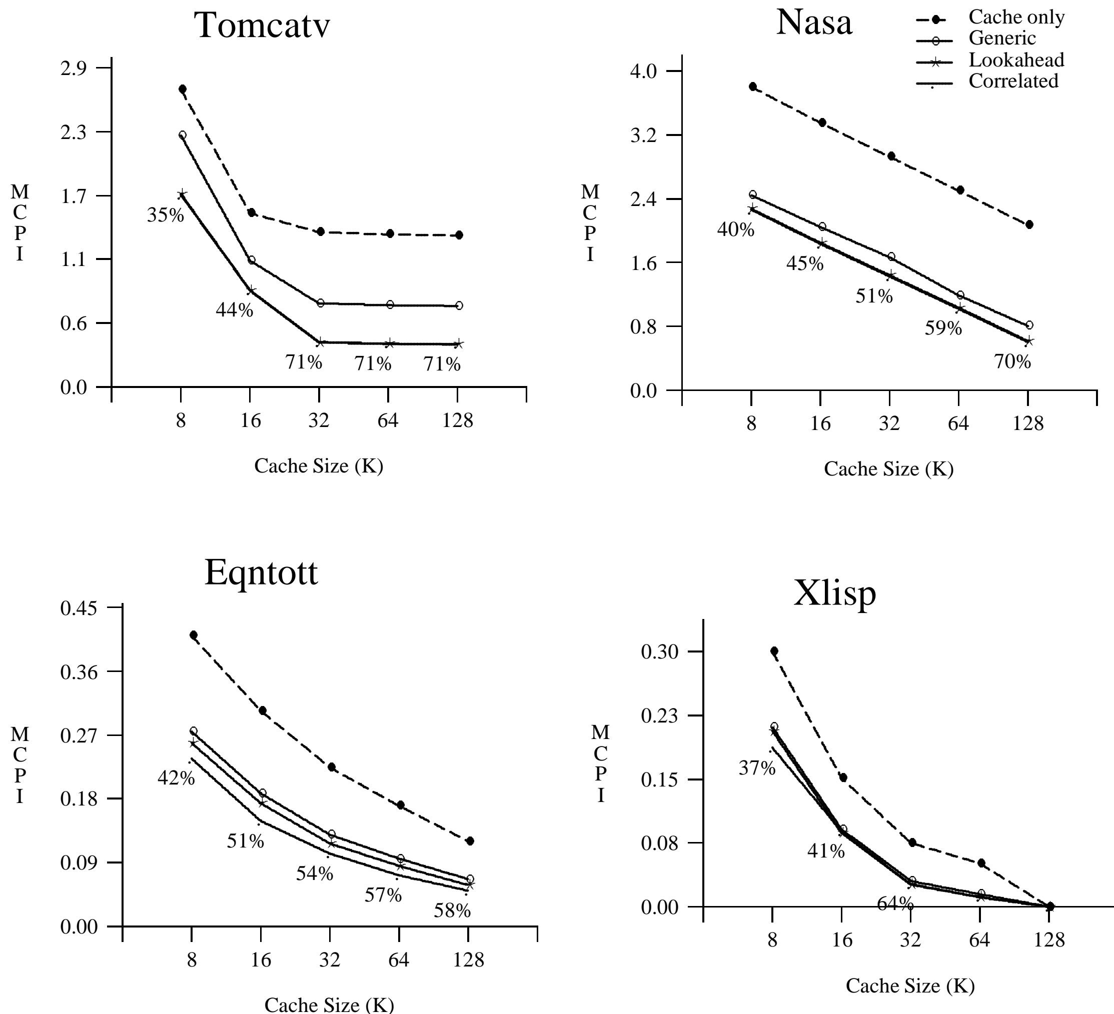


Figure 4.3b: Simulation Results for δ = 30 -- *Overlapped*

that the absolute reduction in the MCPI is significant, over one cycle in both cases, and independent of the cache size. Based on the results of the first two groups, it can be seen that the performance data at moderate cache size (e.g., 16K and 32K) argues forcefully for spending some cache real estate on the RPT and BPT rather than increasing the cache size.

The third group consists of Spice, Doduc, Gcc, and Fpppp. For Spice and Doduc prefetching is still valuable: the data access penalty is reduced by about 30%. For Fpppp, and to a lesser extent Gcc, a pure cache of 16K has almost captured the working set: prefetching cannot help much. There are several factors that lead to the small advantage brought upon by prefetching. First, because the fraction of references in scalar or zero

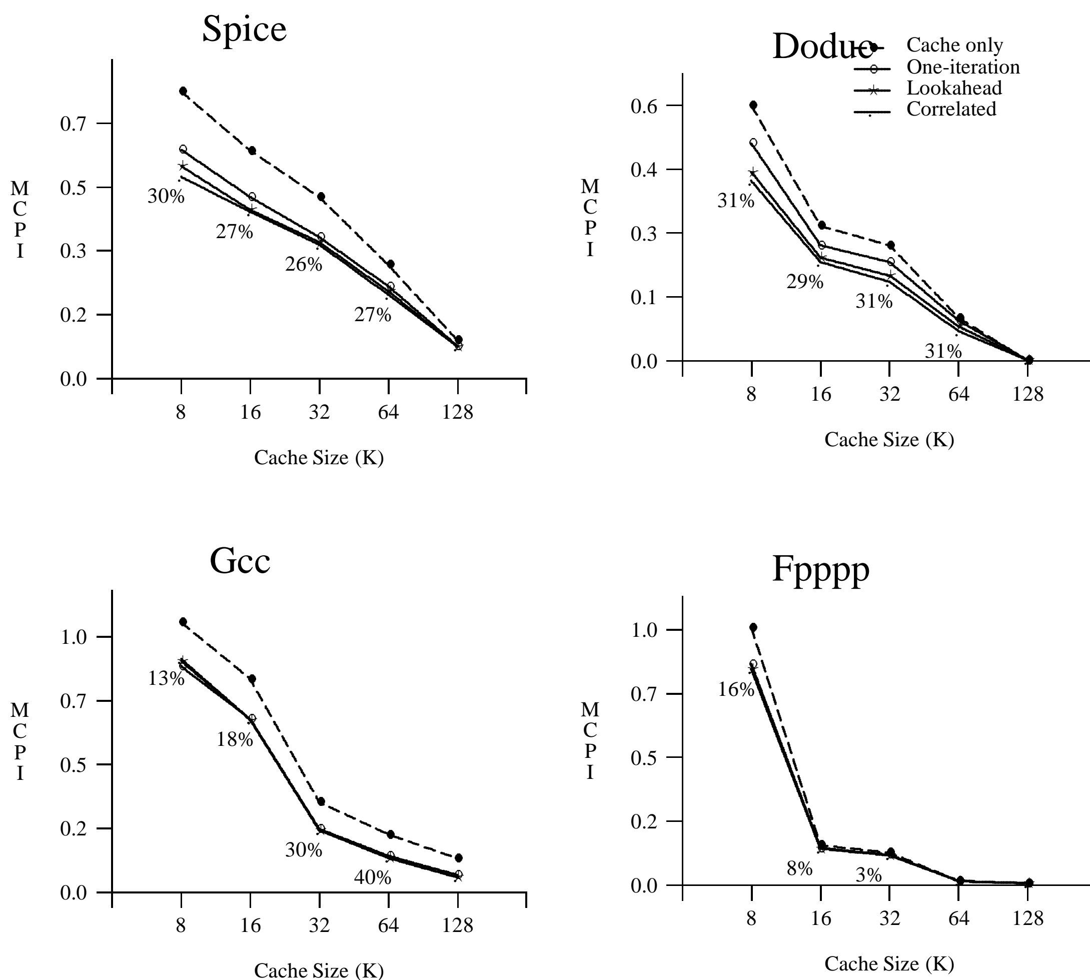


Figure 4.3c: Simulation Results for δ = 30 -- *Overlapped*

stride categories dominates (98% in Fpppp and over 50% in the other 3 benchmarks, cf. Table 4.1), the performance contribution by prefetching accesses with non-zero strides become less significant. Second, the significant branch prediction miss ratio (e.g., 20% in Gcc) precludes successful prefetching. And, third, the RPT may not be capable to hold all active memory instructions at the same time because of either its limited associativity or its small number of entries. We examine this last issue later in this section.

Finally we compare the relative performances of the three reference prediction schemes. As could be expected, the increased level in hardware complexity pays off. However, the difference between the *lookahead* and *correlated* variations is always small, less than 2%, with Eqntott being the only exception with one data point showing a 10% improvement. The difference between *lookahead* and *generic* is more significant. It is most notable,

differences of over 40%, in the benchmarks Tomcatv (large loop body so in the *generic* scheme the prefetched data will arrive too early displacing other useful data or being replaced before its use) and Espresso (small basic block so in the *generic* scheme the data will arrive too late, generating *hit-wait* cycles). These results show that the *lookahead* logic is worth implementing since it allows the flexibility to prefetch at the correct time while the complexity required to help data accesses in outer loops as in the *correlated* scheme plays a much less significant role.

In summary, the prefetching schemes are effective in reducing the data access penalty. A prefetching hardware unit is particularly worthwhile when the chip area is limited and a choice has to be made between the added unit and slightly increasing the on-chip cache capacity.

4.3 Effect of Design Variations

In this section, we examine the impact of several architectural issues on the performance of prefetching. The issues include memory latencies, memory models, changes in block size, the organization of the RPT, the lookahead-limit, and various placements of prefetched blocks. In the remainder of the section, for brevity sake, we restrict ourselves to reporting on benchmarks with the most salient features (cf. see Appendix A.1 for complete results). All performance evaluations of prefetching are based on the lookahead scheme.

4.3.1 Effect of Memory Models and Latencies

Figure 4.4 presents the data access penalties of the baseline cache and the *lookahead* scheme with respect to the three memory models and memory latency varying from 10 to 50 cycles for four of the benchmarks (Tomcatv, Espresso, Eqntott, and Xlisp). Each bar corresponds to one architecture and one memory latency, with the MCPI due to a *Pipelined* access and the overhead coming from the *Overlapped* and *Non-overlapped* models stacked on top of each other. The two numbers inside the bars of the lookahead prefetching give the percentages of the penalties reduced by the prefetching for the *Non-overlapped* model (worst) and the *Pipelined* model (best) respectively. The overhead in the case of the baseline cache comes from the waiting time incurred by a cache miss when a write back is in progress since we assume that a request in progress cannot be aborted. Similarly, the overhead in the prefetching scheme includes the stall time of ‘‘real’’ demand cache

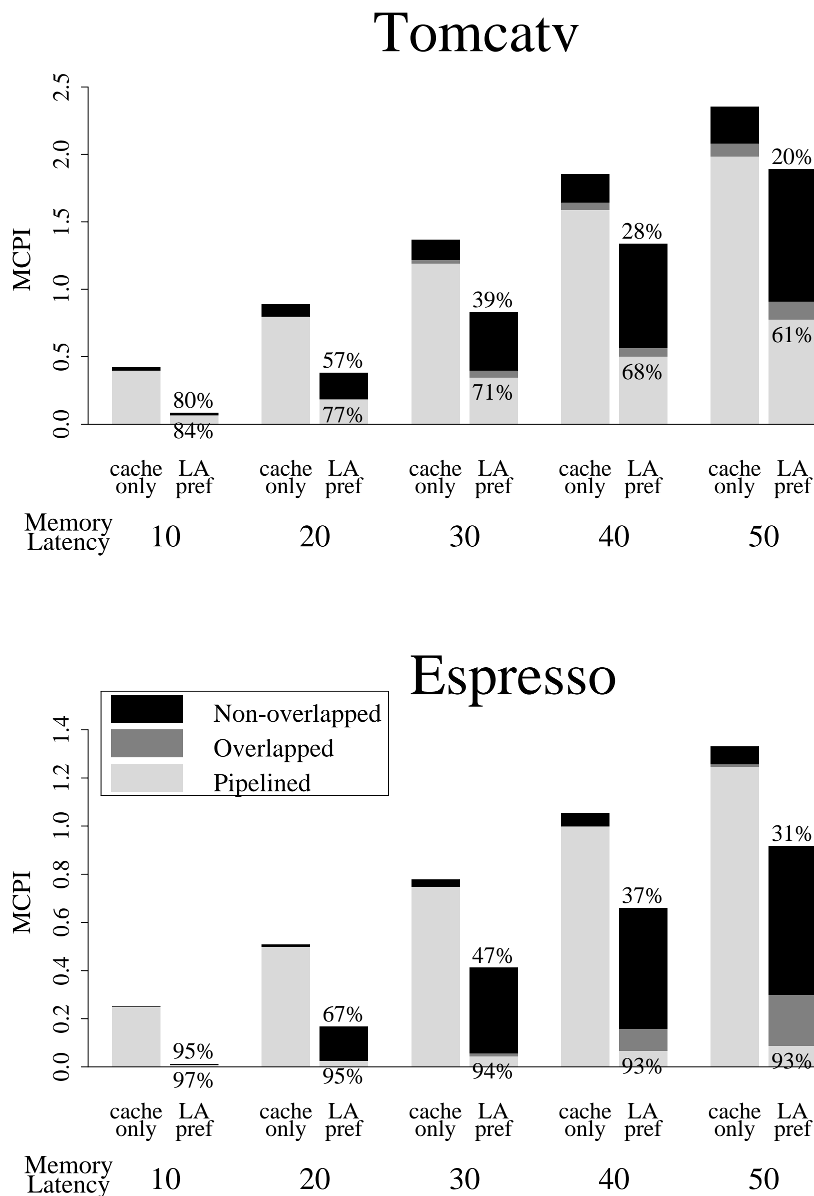


Figure 4.4a: Effect of memory models and latencies

miss waiting for prefetching or write-back requests in progress. Note that it is not very meaningful to have a large access time (say 50 cycles) for the *Non-overlapped* model and a small latency of 10 cycles for the *Pipelined* model. We simply intend to show the effect of the stall penalty when a large spectrum of memory bandwidth is presented.

As could be expected, a memory interface with restricted bandwidth like that of the *Non-overlapped* model will result in poorer relative performance improvements with longer

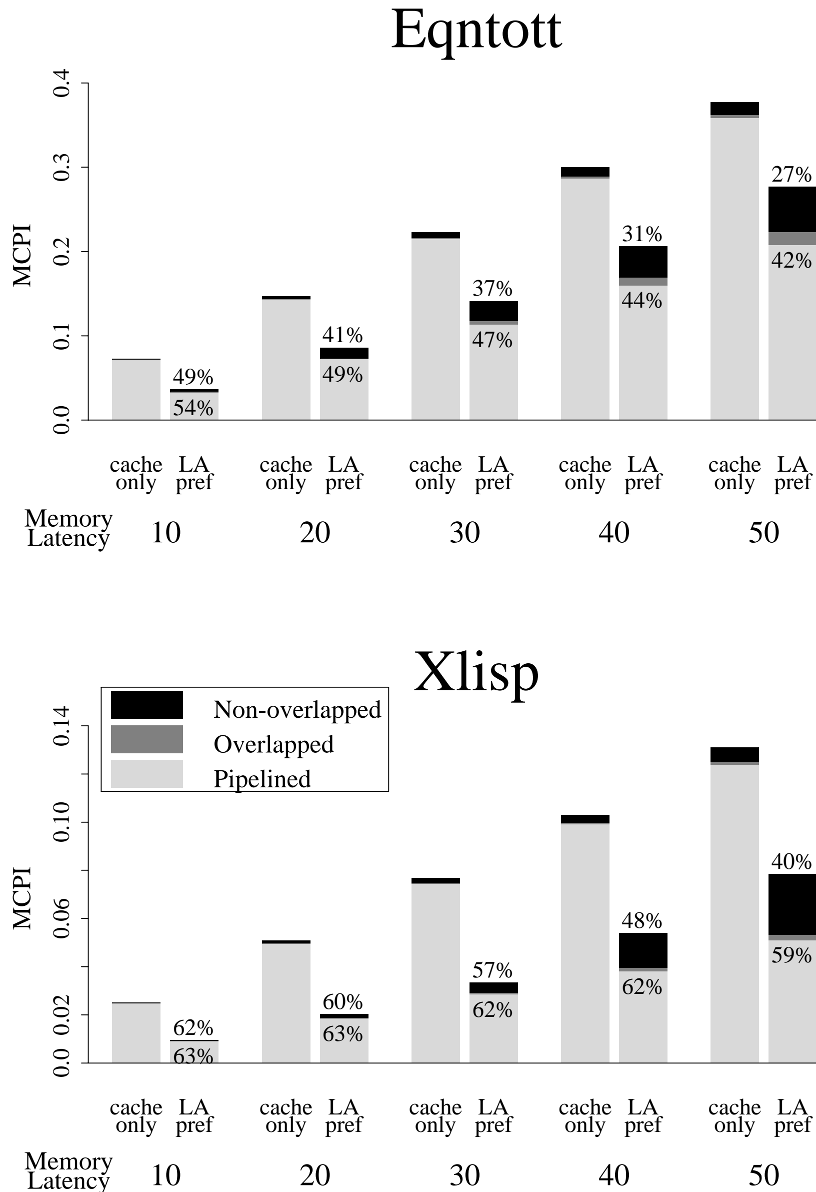


Figure 4.4b: Effect of memory models and latencies (Continued)

memory latencies. This is quite noticeable in the benchmark Espresso and even more in Tomcatv, where the MCPI's are larger than others. A large portion of busy time is eliminated when passing from the *Non-overlapped* model to the *Overlapped* model and then even more with the *Pipelined* model. For all four benchmarks, the difference between the latter two models is less significant than that between the first two models. This is because much of the required parallelism can be exploited by the *Overlapped* model. The

results shown in Figure 4.4 indicate that an adequate interface is necessary to meet the memory bandwidth demand of prefetching techniques that exploit the parallelism among several memory requests. Cache miss reduction by itself is not sufficient to assess the value of a prefetching scheme.

As the memory latency increases, the relative access penalty of the prefetching scheme in all three models also increase. In the case of the *Non-overlapped* model, the main reason is the lack of concurrency in the requests, resulting both in *hit-wait* cycles and in the ORL being full more often. Another reason, common to all three models, is that the *lookahead* scheme relies on the branch prediction for the LA-PC. The correctness of the prediction is sensitive to a large latency (see also Section 4.3.4) and therefore wrong prefetches using the interface can occur more often with larger latencies. The better results obtained with small memory latencies reinforce our previous claim that the *lookahead* scheme is beneficial to high-performance processors with a limited on-chip cache. Such benefits do not degrade too much even with an interface to a secondary cache with limited concurrency such as the *Non-overlapped* model.

4.3.2 Effect of Block Size

It is well known that for a cache of given capacity and associativity, the block size that leads to the best hit ratio is a compromise between very large sizes to increase the spatial locality and small sizes to reduce conflict misses[Przybylski 90]. Given that a prefetching scheme will increase the spatial locality, we can predict that the best block size for a prefetching scheme should be smaller than or equal to that of the pure cache.

Figure 4.5 presents the performance of the various architectures as a function of the block size. The baseline is a 32K-byte direct-mapped cache. The prefetched blocks are of the same size as the blocks fetched on real misses. Our experiments are based on the *Overlapped* model with a transfer rate of 8 bytes per cycle, and request and memory latency of 2 and 20 cycles respectively. As can be seen in the figure, the best block size for the baseline architecture is either 32 or 64 bytes and the choice can lead to significant improvements, for example a reduction in MCPI by a factor of 3 in Matrix and a factor of 2 in Eqntott when passing from a block size of 8 to a block size of 32. By contrast, the prefetching scheme is much less sensitive to the block size and the best results are obtained for a block size of 32 or less. This result one more time argues for the hardware prefetching being associated with an on-chip cache since limited bandwidth (small number of pins, i.e.,

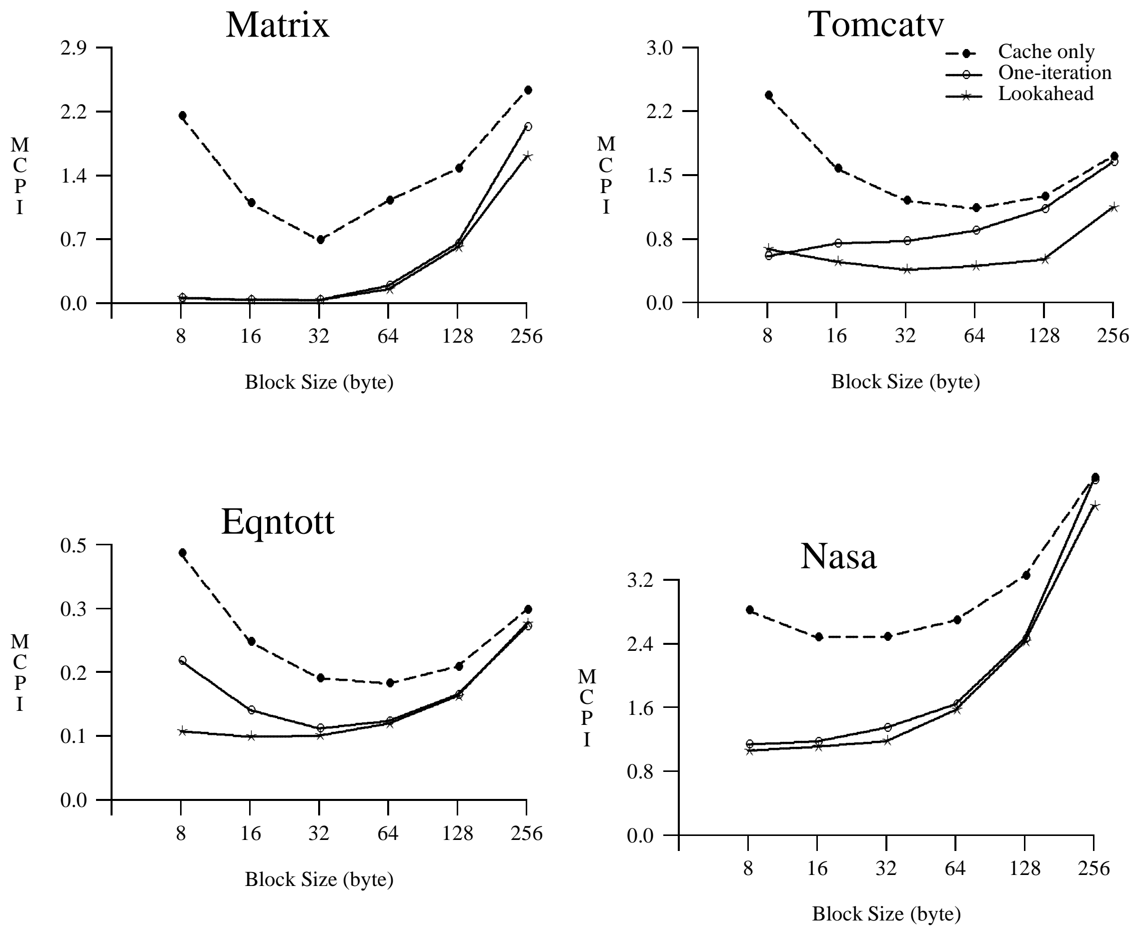


Figure 4.5: MCPI vs. block size for 32K cache (*Overlapped*)

small block size) is not an impediment to its performance.

4.3.3 Organizing the Reference Prediction Table

As discussed in Section 4.2, the benefits incurred by the prefetching schemes depend primarily on program behavior, more specifically on the amount of predictable references. A second factor is the organization of the Reference Prediction Table, i.e., its size and its associativity.

Let us look at the cost of implementing a direct-mapped 512-entry RPT. In each entry of the generic scheme, the *prev_addr* and *stride* fields need 4 bytes each, and the tag and state bits are similar to the tag directory in a cache. For a lookahead scheme, we have to add a few bits per entry for the *times* field. The correlated scheme requires significantly more space (maybe 50% more). Therefore, the cost of a 512-entry RPT for the lookahead

scheme is roughly equivalent to that of a 4K-byte data cache with block size of 8 bytes.

The hit ratio of instructions referencing the RPT is over 90% in seven out of the ten SPEC benchmarks. In an eighth benchmark, Fpppp, the hit ratio is very low (as low as 10%) even when we double the size of the RPT. This is primarily because the program has a very large loop body due to a long sequence of scalar accesses. Most of the references recorded in the RPT have been replaced when the loop starts its next iteration. For the two remaining benchmarks, Gcc and Doduc, Figure 4.6 shows the fractions of instructions that hit in the RPT as a function of the size and associativity of the RPT. In addition, the line entitled ‘‘prefetch attempt %’’ (below the number of RPT entries) shows the percentage of accesses hitting entries not in the *no-prediction* state and with a non-zero stride for which prefetching was attempted.

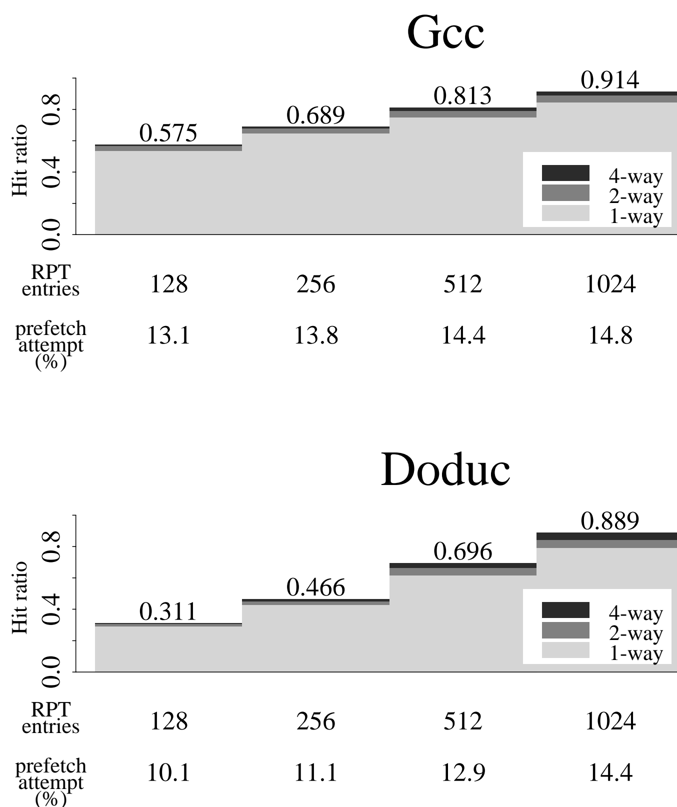


Figure 4.6: Hit ratio and attempted prefetch of RPT

As shown in the figure, increasing the associativity of the RPT has minimal effect. The sequential nature of instructions is the reason for this lack of improvement. On the other hand, increasing the size of the RPT improves the hit ratio since a small RPT cannot hold

the referencing instructions of the most frequently executed loops in these two benchmarks. Note also, that the ‘‘prefetch attempt %’’ increases with the larger hit ratio. It is because it takes two or three accesses to regain the necessary stride information for those instructions that have been replaced. When a fairly good hit ratio is obtained, the percentage of accesses with prefetches is roughly equal to that of data accesses in the *constant stride* category (cf. Table 4.1).

A question that might arise is given extra chip capacity, should it be devoted to a larger or more complex D-cache or a larger or more complex RPT. On one hand, a good hit ratio in the RPT may not be directly translated into a smaller miss ratio in the data cache (depending on the fraction of non-zero stride accesses). On the other hand, adding complexity to the data cache may yield a better performance, but care should be taken not to increase the basic cycle time of the cache (e.g., because of extra gate delays due to comparators and multiplexors). However, on the basis of our experiments we would not argue for a larger or more sophisticated RPT. A possible solution for improving the RPT hit ratio without enlarging the table is to not replace those entries with non-zero strides. While useful patterns might be preserved, there is a problem, namely we are locking in the RPT entries corresponding to instructions that will be never executed again. A better approach is to enter in the RPT only those instructions that may have a non-zero stride. A compiler can easily provide this information. In an experiment on the DECstation 5000, we simply excluded memory instructions that use the stack pointer or a general register (*sp* or *gp*) from being entered in the RPT, since in general a non-scalar reference does not use these two registers as base register. The hit ratios for Fpppp and Doduc were increased up to 91% for an RPT of 512 entries.

In summary, in most cases, a moderate sized RPT (e.g., 512 entries, roughly equivalent to a 4K-byte cache) is sufficient to capture the access patterns for the most frequently executed instructions. A possible optimization that would be useful for programs with very large basic blocks is to be selective in storing entries in the RPT.

4.3.4 Varying the Lookahead Limit

In the *lookahead* and *correlated* schemes, the LA-PC is used to control the timing of the prefetches. Its forward progress is bounded by the Lookahead limit *d*, i.e., the maximum number of cycles allowed between LA-PC and PC. Setting *d* must take into account two opposite effects. We should certainly issue prefetches early enough and therefore *d* must

be greater than δ so that the number of *hit-wait* cycles is reduced. This is even more crucial when the data misses are clustered and the memory model is restrictive like in the *Non-overlapped* and to a lesser extent the *Overlapped* model. On the other hand, d should not be too large and cross over too many basic blocks because the branch prediction mechanism loses some of its reliability with increased d. Also, we don't want prefetched data to replace (or to be replaced by) other useful blocks.

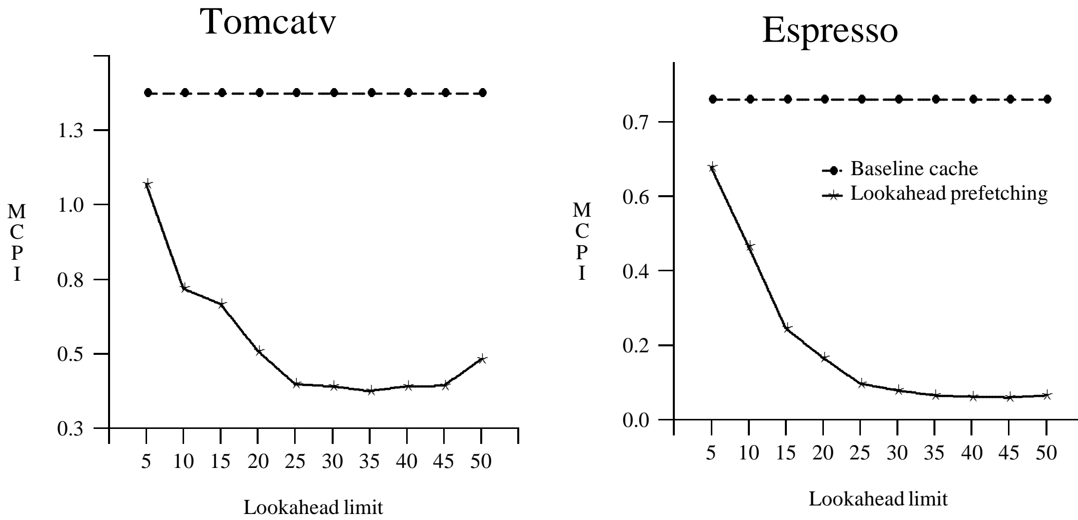


Figure 4.7: MCPI vs. LA-limit (d) for $\delta = 30$ (*Overlapped*)

Figure 4.7 shows the performance of the *lookahead* scheme under the *Overlapped* model as a function of the Lookahead limit d for two representative programs. When d is less than the memory cycle time δ (30 cycles in the figure), each access to the prefetched block in progress will be a *hit-wait* access and thus contributes *hit-wait* cycles to the total access penalty. The contributed *hit-wait* cycles are decreasing as d approaches δ. A local minimum for the MCPI happens around $d = 35$. A further increase in d will result in a slight MCPI increase because of the two aforementioned factors (incorrect branch prediction and data replacement). For the *Overlapped* model, it appears that setting d to a value slightly above δ will give the best results.

4.3.5 Alternatives for the Placement of the Prefetched Data

Up until now, our model architectures have placed the prefetched data directly in the first-level direct-mapped, 32K-byte cache. This strategy imposes additional complexity to the design of the primary cache. For example, that cache must be able to serve multiple

requests (non-blocking on prefetches) and a priority scheme for ''real'' misses must be implemented. Performance factors also come into play: interference between prefetch and real miss requests as well as data pollution. This potential performance loss could be alleviated by increasing either the cache size or the associativity. However, this would further add to the cost and complexity of the first-level cache.

In case of a cache hierarchy, a possibility is to prefetch only in the secondary level off-chip cache. Placing the prefetched data in the secondary cache simplifies the design of the primary cache since the extra features mentioned above are moved to the secondary cache - off the critical path; and the secondary cache is generally large so that the effect of pollution is decreased. Such a design could be advantageous when the latency to the secondary cache is not too large. However, when the latency of the secondary cache is one order of magnitude larger than the hit time in the primary cache, the small reductions in cache interference and the detrimental effects of data pollution do not balance out with the increase in the average cache access time due to bringing the prefetched data from the secondary cache. Mowry and Gupta [Mowry & Gupta 91] study software prefetching in a secondary-level remote access cache (RAC) in the context of the DASH cluster architecture. Prefetching in the RAC is the default but their results show that prefetching in the primary cache would have been more effective. A simulation study of prefetching into a second level cache is performed by Smith *et al.* [91]. Several hardware-based schemes (basically only using OBL), are simulated showing up to a 38% reduction in the penalty of memory accesses for scientific programs.

In this section, we investigate an alternative to prefetching in the primary cache, namely prefetching in a separate prefetch buffer. We also contrast the prefetch buffer solution with a *victim cache* [Jouppi 90]. The separate prefetch buffer has a block size the same as that of the primary data cache [Klaiber & Levy 91, Chen *et al.* 91]. The rationale is to nullify data pollution effects. However, the hardware complexity of this solution is non-negligible: requests for data must be sent -- and checked -- simultaneously in the cache and the buffer; and the buffer itself takes some space and should be fully associative. There is therefore a danger that accesses to the cache will take longer and be on the critical path for the determination of the cycle time. While the overall hit access time will usually be equal to that of the regular data cache (provided the mechanism can bypass the outcome of the prefetch buffer), the determination of a miss will be delayed until the comparisons in the buffer complete. This will take longer since the prefetch buffer has a larger associativity

than the data cache. At an equivalent cost of hardware complexity, we can trade the prefetch buffer for a *victim cache*, a small fully-associative cache holding the most recently replaced data lines. The intent here is that instead of using the buffer for the sole use of storing prefetched data, the presence of the victim cache will reduce the number of conflict misses among useful and prefetched blocks. At the other extreme of the usage spectrum of the victim cache is a variation with a 32-entry buffer for nonzero stride data only. The intuition for this solution is that accesses to the non-scalar data will gradually change and sweep a large portion of the data area. The ''reuse lifetime'' of non-scalar data should be shorter than that of scalar data. To put them in a FIFO buffer may be more advantageous than to place them in a unified cache where they may conflict with other scalar data which are more likely to be reused. Finally, we also consider an even cheaper, in terms of the number of comparators, a two-way set associative cache. For cache hits, the hit access time in the two-way cache is larger than that of a direct-mapped cache with an extra buffer, whereas the determination of a cache miss takes longer in the prefetch buffer. Overall, the average access time in a two-way cache is larger because most references should be cache hits.

Figure 4.8 presents the results of our simulations using the various alternatives. For ease in simulation, we assume that the cycle time is the same in all solutions. This favors the buffer and two-way set associative schemes. We show the MCPI with decomposition of the read penalty into real *read miss* and *hit-wait* cycles. We show from left to right: a baseline cache (direct-mapped), a baseline (direct-mapped) with 32-entry victim cache, a two-way set associative cache, and then several lookahead prefetching caches: a unified cache without extra buffer, a unified cache with a 32-entry victim cache, a cache with a 32-entry prefetch buffer, a cache with a 32-entry buffer for nonzero stride data, and finally a cache with prefetching based on a two-way set associative unified cache.

First we look at the effect of an extra buffer and of the two-way set associativity without prefetching. As can be seen, both the victim cache and the two-way cache show a slight performance improvement over the direct-mapped baseline cache. The benchmark Nasa shows remarkable benefits from the extra hardware because there are severe conflict misses in the program. The two options, victim cache and set-associativity, make similar contributions to performance improvements.

Then we examine the impact of the extra hardware components on prefetching. As can be observed in the figure, the cache with a victim cache as well as with prefetch buffers

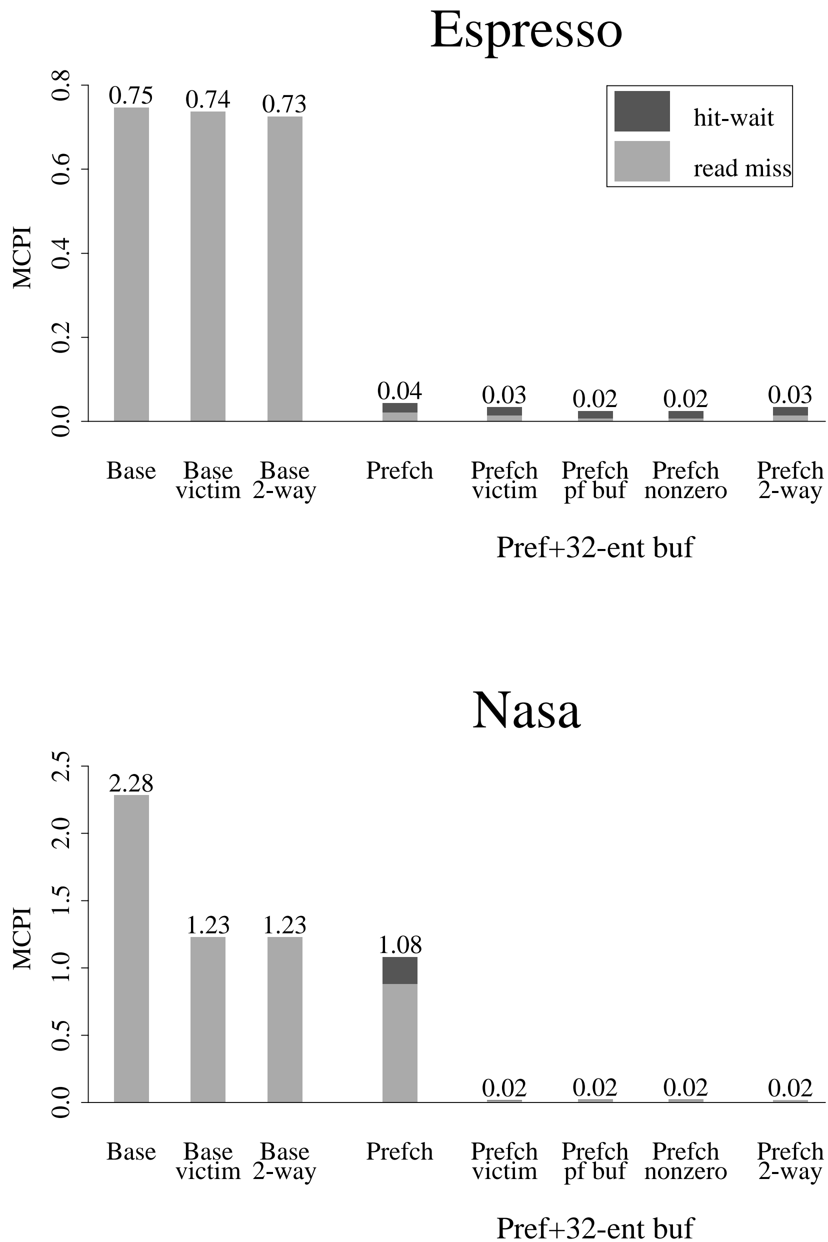


Figure 4.8a: Variations in prefetching placement

does have a better reduction in read penalty than a unified prefetching cache without any extra hardware. However, the performance improvement in all benchmarks except NASA is not significant when compared to the overall reduction of the read penalty by effective prefetching schemes. The benefits of these choices are roughly equivalent to the gains

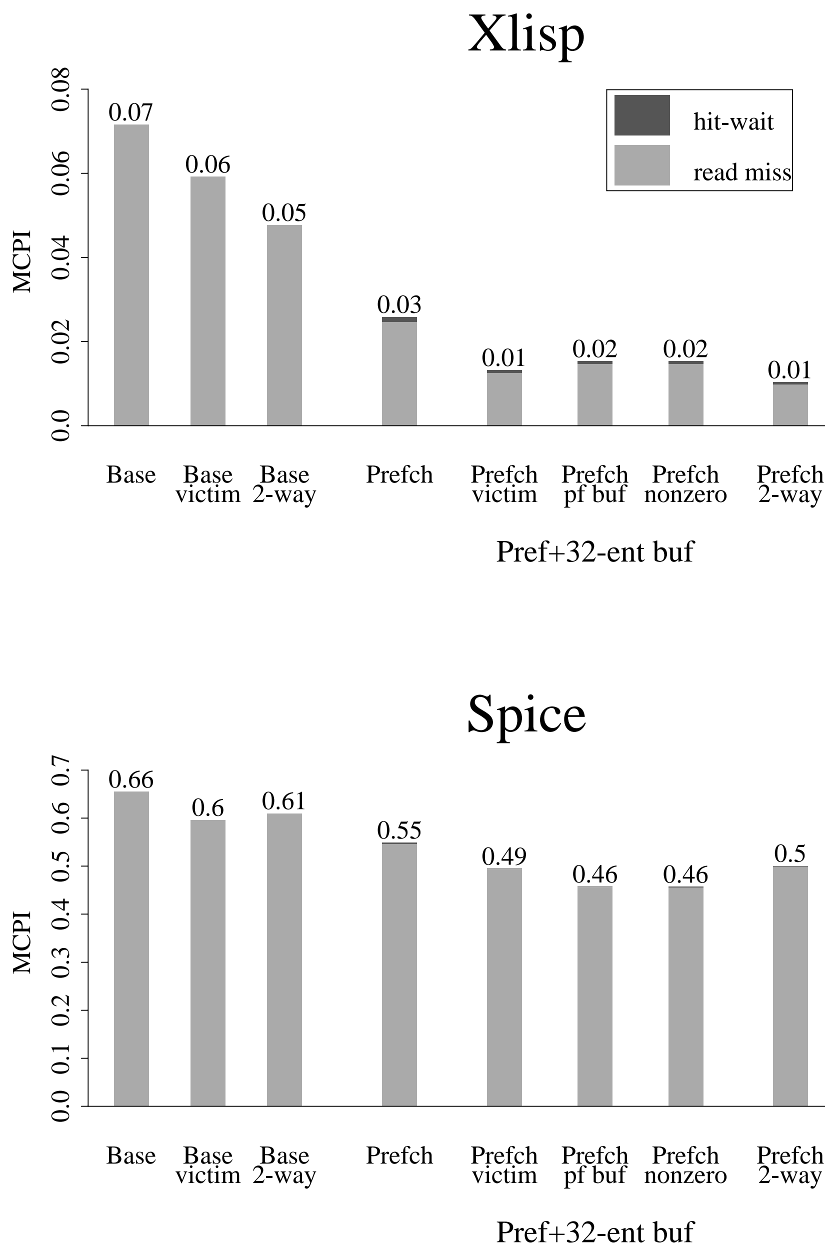


Figure 4.8b: Variations in prefetching placement

brought upon in the baseline caches when the same features are added. In case of Nasa the costs of adding an extra buffer eliminates most of the data access penalty. The magnitude of the performance gain is similar to that in the baseline caches.

When we compare a victim cache, a prefetch buffer or nonzero-stride buffer, we can

see that the difference among them is negligible. Basically since the buffer is a temporary FIFO queue, the lifetime of a block in the buffer does not exceed the length of that buffer. It does not make much difference regarding what blocks are placed in the buffer instead of the data cache. As we use a lookahead mechanism to control the arrival time of prefetched data and a state detection mechanism to avoid unnecessary prefetches, conflicts between non-scalar and scalar data are not significant.

Last, we see that a two-way set associative cache performs about the same as the direct-mapped cache with the 32-entry buffer. Nevertheless, it requires less complexity in terms of the number of comparators, when compared with the fully-associative buffer.

Overall, our experiments suggest that the cache interference and data pollution problem in prefetching are not critical for those benchmarks with a primary cache of moderate size. An extra prefetch buffer appears unnecessary.

4.4 Summary

In this chapter we have evaluated the three prefetching schemes presented in Chapter 3 by comparing them with a pure cache design at various cache sizes. These comparisons were performed using cycle by cycle simulations of the ten SPEC benchmarks. The results show that the prefetching schemes are generally effective in reducing the data access penalty. The cost of the hardware unit is not prohibitive; a moderately sized RPT (roughly equivalent to a 4K cache) is generally sufficient to capture the access patterns for the most frequently executed instructions. We observed that the lookahead scheme has a moderate win over the generic scheme, while the performance difference between the lookahead and correlated schemes is fairly small.

We have also examined the performance of the prefetching scheme when we vary architectural parameters such as block size, memory latency, and memory bandwidth. The main results are that the performance of the lookahead prefetching is best for small blocks (8 or 16 bytes) and that its effectiveness is quite significant with a small memory latency even when assuming a restricted bandwidth interface to the next level of the memory hierarchy. These observations lead us to argue that a hardware-based prefetching scheme would be valuable and cost-effective as an assist to an on-chip data cache backed-up by a second-level cache with an access time an order of magnitude larger.

Finally, we examined several alternatives for target storage devices for the prefetched data. Comparisons among unified caches, prefetch buffers, and victim caches suggest that

the unified cache (direct-mapped or two-way) is the most cost-effective choice since the cache interference and data pollution due to prefetching are minimal.

We therefore advocate an effective prefetching hardware support unit as an assist to an on-chip cache. However, hardware prefetching schemes may not be as effective in higher levels of the memory hierarchy. In that case, when the latencies are two orders of magnitude larger than the processor cycle time, prefetching data by software-directed techniques may be more beneficial. The software approach might also lend itself much better to multiprocessor environments. The next chapter will give a comparative evaluation of software and hardware prefetching and will look at possible ways to combine hardware and software prefetching.

Chapter 5

Comparative Evaluation of Software and Hardware Prefetching Schemes

5.1 Overview

The previous two chapters have shown that hardware prefetching can be an effective mechanism for tolerating memory latency. However, as mentioned in Section 2.1, prefetching techniques can be in the software as well as in the hardware domain. The choice of whether to apply hardware or software solutions to prefetching is an interesting question for the architecture community. In this chapter, we will discuss the advantages and disadvantages of both schemes, and try to see how a possible combination of the two can be achieved.

As seen previously, hardware-based prefetching requires some support unit connected to the cache but no modification to the processor. Its main advantage is that prefetches are handled dynamically at run-time without compiler intervention. The drawbacks are that extra hardware resources are needed and that memory references for complex access patterns are difficult to deal with. In contrast, software-directed approaches rely on compiler technology to perform static program analysis and to insert prefetch instructions. The CPU explicitly executes prefetch instructions to initiate data fetches for caches. These schemes may perform prefetching selectively and effectively. The drawbacks are that they cannot dynamically uncover some useful prefetching (e.g., conflict misses and invalidation misses) and that there is some non-negligible execution overhead due to the extra prefetch instructions.

In this chapter, we compare our proposed hardware scheme with the software-directed prefetching approach in both qualitative and quantitative ways. The qualitative comparison is performed by contrasting our hardware scheme with the software scheme (mainly Mowry *et al.*'s approach [92]) focusing on aspects such as how accesses are identified for generating prefetches and how prefetches are scheduled within loops. The quantitative evaluation is performed by a direct-execution simulation of three SPLASH benchmarks

and of the Matmat kernel in a shared-memory multiprocessor environment. We emulate software prefetching by manually inserting prefetches in the codes. The metrics of interest include the effectiveness of the prefetching schemes in reducing execution time, the side effect of prefetching schemes such as the increase in network traffic, the performance sensitivity to a range of memory latencies, and the impact of the memory consistency model. We also discuss means of combining both approaches.

Our qualitative comparisons indicate that in the domain of linear array references both hardware and software schemes are able to generate prefetches to reduce cache misses. When complex data access patterns are considered, the software approach may have more compile-time information to perform sophisticated prefetching, whereas the hardware scheme has the advantage of manipulating dynamic information (such as conflict misses or input data dependence). While the software scheme may have a code expansion problem, the predictability that the prefetched data will be used is not as great in the the hardware scheme. Our performance results from the simulation of the four benchmarks confirm these observations. Our results also show that hardware prefetching introduces more memory traffic into the network than software prefetching and that the performance gains of both approaches degrade slightly when the memory latency is getting larger. Our simulations indicate that an approach combining software and hardware schemes is very promising in reducing the memory latency with least overhead.

The rest of the chapter is organized as follows: Section 5.2 gives some background information on software prefetching. In Section 5.3, we compare the two schemes in a qualitative fashion. Section 5.4 describes the evaluation methodology as well as the model implementations of the prefetching schemes that we study. Section 5.5 presents simulation results and explores the impact of varying memory latencies, of the memory consistency model, and the side effect that prefetching can bring up. Section 5.6 shows an architecture for combining the software and hardware schemes.

5.2 Software Prefetching

In this section we give an implementation background of software-directed prefetching schemes in more detail than what has been discussed in Chapter 2. Most software approaches proposed in the past mainly focus on the loop domain for uniprocessors and most of them study prefetching based on codes with manually inserted prefetches [Porterfield 89, Klaiber & Levy 91, Mowry & Gupta 91]. Because Mowry et. al.'s [92]

scheme is the only one, to our knowledge, that has been automated in an experimental compiler, we will basically use their framework as the basis of our comparison.

Software-directed prefetching requires support from hardware and software. On the hardware side, the processor must provide a special instruction to initiate prefetches, and the cache should be able to support servicing multiple memory requests concurrently (as we have discussed in Section 2.2.1). Also, the system needs a prefetch issue buffer to hold pending prefetch requests. When the processor executes a prefetch instruction, the address of the block to be prefetched (as specified in the instruction) will be inserted into the prefetch issue buffer. Prefetch requests in the buffer will be issued whenever the memory interface allows it. Once the buffer is full, the processor may either stall until an entry is available or the prefetch request is simply discarded.

In Mowry et. al's approach, a compiler algorithm identifies those data references that are likely to be cache misses, and prefetches are inserted only for them. Specifically, the algorithm focuses on array accesses whose indices are linear functions of the loop indices in scientific programs. The compiler algorithm uses locality analysis to perform data reuse analysis, and then derives, based on given cache parameters (e.g., cache size and block size), a set of accesses that belong to a (so called) localized iteration space in which locality is preserved among accesses. Once the locality is known, a prefetch predicate for each reference that would lead to a cache miss is introduced in the loop for determining if the prefetch should be executed in a particular iteration. However, the cost of the prefetch predicate can be removed by loop splitting, that is, decomposing the loops into different sections in which all predicates will be evaluated to the same value. Then, prefetches are scheduled within the loop by taking into account the memory latency and estimated loop execution time. At this last stage, the concept of software pipelining is used to schedule prefetches several iterations ahead of their corresponding references.

Since the compiler algorithm is aimed at the domain of linear array references, which is similar to where our hardware scheme obtained its motivation, it is interesting to compare and contrast the perspective benefits and implementation costs of Mowry et. al.'s approach and of our hardware scheme.

5.3 Qualitative Comparison

In this section, we first give a high-level comparison between the software and hardware schemes from a general point of view, and then we specifically contrast Mowry et. al.'s

approach [92] with our hardware scheme in more detail. Lastly, we focus on design issues in a multiprocessor environment.

5.3.1 High-level Comparison

When compared to hardware-based schemes, software-directed approaches have some advantages. First, their hardware cost is minimal. In addition to the common requirement of all prefetching schemes (i.e., a lockup-free cache), the only requirement of the software prefetching in the processor is the extra prefetch instruction. Unlike the hardware scheme, there is no need for a complex hardware mechanism to detect and perform prefetching. Second, prefetches for accesses with simple and even with complex patterns, primarily for loop-domain references, can be identified at compile time. Moreover, more user information can be exploited so that prefetched data are most likely to be used. And third, in a multiprocessor environment, more factors such as data coherence, task scheduling, and task migration can be taken into account. However, software-directed approaches have also several disadvantages. First, prefetch instructions introduce an overhead, at the very least the execution of the prefetch instruction and possibly other computations such as effective addresses and prefetch predicates. Although an intelligent compiler may be able to reduce much of the unnecessary overhead, it could still be relatively significant, especially as a result of code expansion and increasing register pressure, or when the memory latency is small. Second, dynamic information such as conflict or capacity cache misses (thus preventing the prefetching of replaced data) and estimates of execution time for loops calling subroutines (thus not being able to prefetch at the right time) may not be uncovered. Third, the optimizations are language and compiler dependent while the hardware schemes do not require any change in the executable code.

5.3.2 Identifying Cache Misses

The success of software prefetching depends primarily on whether the prefetch instruction overhead can be significantly reduced. To minimize the number of prefetch instructions, a compiler should be able to identify those accesses that are most likely going to be cache misses. Mowry *et al.*'s algorithm exploits three kinds of reuse: *temporal*, *spatial*, and *group*. The *temporal* reuse occurs when a reference within a loop accesses the same data location in different iterations. A reference preserves *spatial* reuse when the same cache line is used in consecutive iterations. Different references have *group* reuse if

(a) Original code

```
for j = 0 to 100
    for i = 0 to 100
        A[j][i] = B[i][0] + B[i+1][0]
end
```

(b) Instrumented code (inner loop only)

```
prefetch(&A[j][0])                               }
for i = 0 to 5 by 2                              }
    prefetch(&B[i+1][0])                         }
    prefetch(&B[i+2][0])                         } prologue
    prefetch(&A[j][i+1])                         }
end                                              }

for i = 0 to 93 by 2                             }
    prefetch(&B[i+7][0])                         }
    prefetch(&B[i+8][0])                         }
    prefetch(&A[j][i+7])                         } main
    A[j][i] = B[i][0] + B[i+1][0]                } loop
    A[j][i+1] = B[i+1][0] + B[i+2][0]            }
end                                              }

for i = 94 to 100 by 2                           }
    A[j][i] = B[i][0] + B[i+1][0]                } epilogue
    A[j][i+1] = B[i+1][0] + B[i+2][0]            }
end                                              }
```

Figure 5.1: Example of instrumented loop

they refer to the same location or to the same cache line. Since reuses do not guarantee locality [Wolf & Lam 91], These reuses are mapped to data locality by taking into account the loop iteration count and the cache size. Let us take a typical inner loop as an example (as shown in Figure 5.1). The accesses of A[j][i] have spatial reuse in the loop. Both B[i][0] and B[i+1][0] share group reuse and also have temporal reuse when their addresses are invariant with respect to the outer loop that contains this inner loop. While misses for

memory accesses (e.g., A[j][i]) with spatial reuse are easily determined, the identification of cache misses for accesses with temporal and group reuse is made more complicated by other factors such as set associativity and replacement policy. Moreover, conflict misses due to self-interference from the same array references or cross-interference from different arrays are not predictable at all. Overall, the algorithm can be successful in identifying most compulsory misses and some of the capacity misses for linear array references, but is unable to handle conflict misses.

In contrast to the software analysis, the hardware scheme has no information that allows it to avoid unnecessary prefetches. Since it is a supporting unit for the cache, unlike the prefetch instruction in the software approach, these extra prefetches do not contribute any overhead as long as they are not on the critical path of the processor. Although prefetches are suppressed when the data block is already found in the cache, there remains the drawback that the additional lookup of the cache tag directory may still delay demand cache accesses or data refills from memory modules. Furthermore, since the prefetches have no knowledge of potential reuse, the hardware scheme is more likely to bring data which are not useful. On the other hand, the hardware mechanism can prefetch data which have been replaced due to conflict misses.

5.3.3 Prefetch Instruction and Predicate

If accesses have spatial or group locality in the same cache line, only the first access to the line will result in a cache miss. For example, if these accesses are with index i, a prefetch predicate $(i \ mod \ l) == 0$ should be tested before the prefetch is issued, where l is the number of array elements in a cache line. The execution of such a predicate is costly in the inner loop, especially when l is large. To avoid the overhead of such a prefetch predicate, the compiler algorithm usually performs loop splitting and loop unrolling.

Let us look at the inner-most loop of the previous example in Figure 5.1. Assume that the memory latency requires the prefetch to be scheduled six iterations ahead and that each data line contains two array elements. By loop splitting, the original loop is decomposed in three sections: prologue, main, and epilogue loops. The prologue loop prefetches the initial data set for the first six iterations. The main loop consists of the largest portion of the loop execution where the loop is in a steady state, that is, the demand of data can be satisfied by those prefetches occurring several iterations ahead. Finally, the epilogue loop finishes the last six iterations without any prefetching. After the loop is split, each

loop of the prologue and main loops is unrolled by a factor of two in order to eliminate the execution of the prefetch condition ($i \ mod \ 2$) == 0. Overall, in their study, Mowry et. al. [92] have reported that the instruction overhead per prefetch instance is low for those scientific programs they used. Unfortunately, one consequence of loop splitting and unrolling is that the code will expand significantly, in addition to the inherent increase caused by the prefetch insertions. It may result in an additional penalty because of the increase of cache misses in the instruction cache. One may argue that the only important part is that the steady-state main loop fits in the instruction cache, and that therefore the overhead of code size is not critical. However, as shown in the example, the loop has been increased roughly by two times (at most the code is within three times the original code [Lam 88]), the expansion of outer loops would be more significant and the problem in an instruction cache would be even more difficult to avoid. The other side effect as a result of code expansion is an increase in register pressure, which may introduce extra spilling store/load instructions.

By contrast, the hardware scheme executes the original loop without modification. However, at least two iterations are required before obtaining correct strides. Unlike the software approach where prefetches are dumped together in the prologue loop, the hardware scheme gradually prefetches the initial data set as the LA-PC runs continuously several iterations ahead of the PC. When the loop in the steady state, i.e., in the main loop, the prefetching is performed in a similar way in both schemes. One important drawback of the hardware approach is that the system still continues to prefetch data even in the last iterations (corresponding to the epilogue loop), since the hardware is unable to know when the loop will end.

5.3.4 Scheduling Prefetches

The purpose of prefetching is to bring data ahead of its use, so prefetches should be issued early enough to hide memory latency. However, they should not be too early so that they do not displace useful data in the working set or are replaced before use. The software algorithm usually schedules prefetches ahead by a number of iterations:

$$\left\lceil \frac{\delta}{s} \right\rceil$$

where δ is the prefetch latency and s is the length of the loop body. As a result, the software scheme prefetches a data item at least one iteration before it is used. The prefetch

is usually placed immediately before or after a corresponding reference to minimize the computation cost of the effective address.

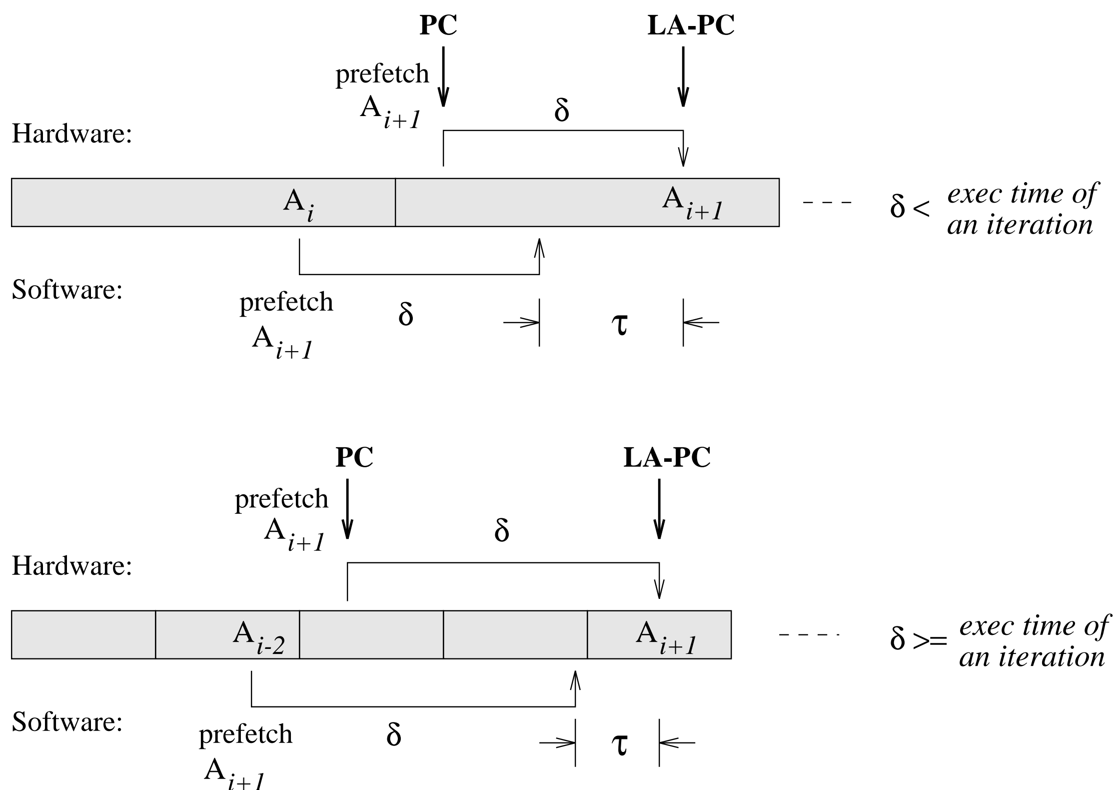


Figure 5.2: Scheduling prefetches

Similarly, the LA-PC in the hardware scheme is capable to identify a prefetch several iterations ahead depending on the prefetch latency. Furthermore, the data will arrive at the cache at a time closer to the actual use. We illustrate the occurrence of a prefetch for both schemes in Figure 5.2. As can be seen in the figure, the issue time of the hardware prefetching depends on the length of memory latency[1], and data is expected to be fetched one memory latency before its actual use. Hence, it happens independently of the corresponding load, and it may even occur in the same iteration as the actual load. In contrast, in software prefetching, a compiler always identifies accesses which are likely to be cache misses and inserts prefetches around the actual accesses. There is a timing window τ between the arrival time of prefetched data and its actual use. Prefetching within such a window can be vulnerable to the negative effects of prefetched data that is

[1] To take memory contention into account, the LA-limit is usually slightly greater than one memory latency

replaced or that displaces other useful data. The timing mechanism for issuing prefetches in the hardware scheme has two implications: (1) prefetched data has less side effects on the cache (such as replacement problems), (2) there will be less clustering of tag lookup by multiple prefetches and their actual loads. A difficulty with the software scheme is that δ may not always be predictable at compile time; however it does not have a drawback encountered in the hardware lookahead mechanism, namely relying on good branch prediction to predict useful lookahead stream.

5.3.5 Prefetching in Multiprocessors

Thus far we have been focusing on prefetching for uniprocessors. When we consider a multiprocessor environment, additional factors come into play:

1. prefetches increase memory traffic,
2. prefetching of shared data items may bring coherence traffic,
3. invalidation misses are not predictable at compile time, and
4. the cache affinity on which task scheduling and migration policy may depend is increased.

Since these factors are common to all prefetching approaches, we will not particularly focus on Mowry et. al.'s solution and our scheme in the following discussion.

The first factor, additional memory traffic, depends on how many unnecessary data are prefetched and how much impact they will have on the working set. Although the same problem may occur in uniprocessors, it becomes more sensitive in multiprocessors, especially when there is a possibility of saturating the interconnection network as in a shared-bus architecture. Ideal prefetching would be such that only data which are most likely to be used are prefetched and the prefetched data arrive at the cache just in time of actual use. The software scheme can be more successful with the first goal, while the hardware scheme may be better at achieving the second goal.

The fact that prefetching may increase coherence traffic is usually difficult to avoid in all prefetching approaches. The problem arises from two situations: the first is that a prefetched data item may need to be invalidated before it is used, and the second is due to the fact that an exclusive-prefetch causes invalidation misses on data that might yet have to

be used in other processors. If a relaxed consistency model is assumed, write propagations are usually delayed until synchronizations. In this case, the first situation is equivalent to the attempt at controlling data that arrive at the cache just in time for its use. The second situation occurs when there is high contention for some shared writable data. Approaches, such as binding prefetch [Gornish *et al.* 90], can reduce the problem by conservatively suppressing prefetches which may have data and control dependencies of accesses in other processors.

The fact that invalidation misses are not predictable at compile time is a weak point of the software approaches, since they lack the dynamic information necessary to initiate prefetches for missing data which have been invalidated. Hardware approaches should be able to fetch back the data which were invalidated, if the state information mandates the prefetching. In case that most invalidation misses are attributed to false sharing, those misses can be minimized by reorganizing the shared data. As a result, an algorithm [Jeremiassen & Eggers 92] that restructures shared data to reduce false sharing can be incorporated in the software prefetching schemes.

Task scheduling and task migration make prefetching in multiprocessors more complicated, because processor assignments may change before the prefetched data in the cache has been used. Parallel programs based on static task scheduling can still be handled by the software algorithm. However, a fine-grained task scheduling policy will be detrimental to prefetching, since the prefetching cost cannot be amortized by the insufficient cache miss reduction. The task scheduling problem is even more critical to the hardware scheme, which requires past access histories stored in a cache-like table. The table contents, like the context of data, should be accumulated in the cache affinity parameter used in the decision of task scheduling.

5.3.6 Other Aspects and Final Words

In this section, we discuss other issues which are not limited to one particular scheme. The first thing is the implementation cost. The hardware scheme requires the RPT and its associated logic (equivalent to a 4K-byte data cache as indicated in Section 4.3.3). The software solution requires little hardware complexity except the prefetch instruction in the processor, but a sophisticated compiler should be provided. Both schemes need a cache which can support multiple concurrent memory requests.

The second issue is whether or not more aggressive program-specific prefetches can be

supported. The software scheme can definitely provide better solutions than the hardware scheme in taking advantage of program information. Although it has not been shown in the literature, the software solution may be able to provide more flexible prefetching, such as pointer-chasing for linked lists, block prefetches (prefetching size being determined in terms of semantic object instead of cache line size), and data reorganization. Although Mowry and Gupta [91] have shown the success of several strategies by code-specific and programmer-directed techniques, it is still unknown if the techniques can be automated for general applications without programmers' intervention and be easily implemented in the compiler.

To summarize, we have compared software and hardware schemes in the context of uniprocessors and multiprocessors. In the domain of linear array references, both hardware and software schemes are able to generate prefetches to minimize cache misses. However, the software scheme may have a code expansion problem, while the hardware scheme has less clues on whether prefetching data will be used or not. The software approach may have more compile-time information to perform sophisticated prefetching such as program-specific prefetches for complex data patterns, whereas the hardware scheme has the advantage of manipulating dynamic information (such as conflict misses or input data dependence). Both of them face the problems of increasing memory traffic and coherence traffic in a multiprocessor environment. Tullsen and Eggers [93] have shown that the prefetching benefits are limited if memory bandwidth is a primary resource in the context of a bus-based shared memory multiprocessor. We examine the latter issues by performing a simulation evaluation for a multiprocessor with an interconnection network with more bandwidth.

5.4 Quantitative Evaluation Methodology

In this section, we first describe the architectural models, the simulation environment, and the benchmarks. We then present our model implementations of hardware and software prefetching.

5.4.1 Architectural Models

The architecture that we assume is a shared-memory multiprocessor (cf. Figure 5.3). It includes 16 MIPS R3000-like processors connected to memory modules through an interconnection network. Each processor has a local memory for private data and instructions, and primary caches for shared data. We assume that private or stack data are allocated in the local memory. Cache coherence is maintained using a full directory protocol [Censier & Feautrier 78]. The directory is distributed among the memory modules and dynamically maintains the states of the data blocks. Prefetched data are put into the caches so that the data still remain visible to the cache coherence protocol.

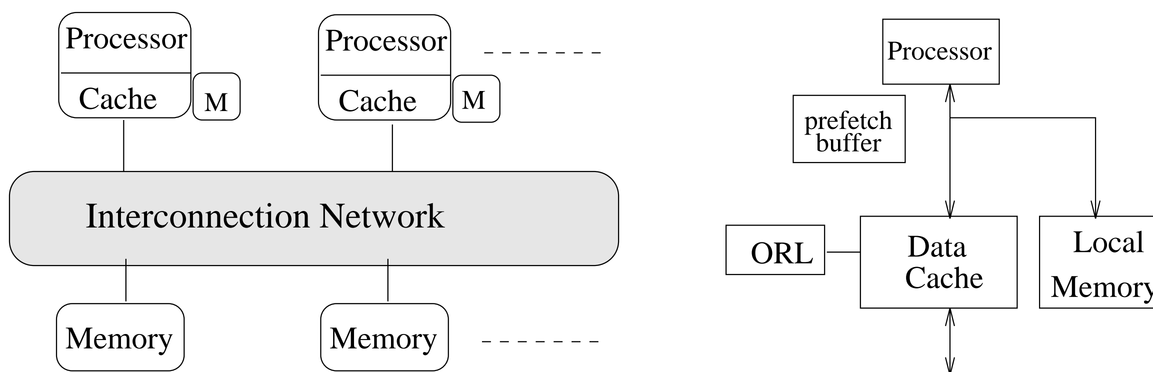


Figure 5.3: Model Architecture

Prefetch requests are generated by the processor. For software prefetching, a prefetch is initiated by a prefetch instruction. For hardware prefetching, a prefetch is triggered by an on-chip supporting unit. To handle prefetching, the system has a prefetch issue buffer, which can hold up to 16 prefetches. The prefetch request will check the tag directory in the cache and will be initiated to the memory system if there is no matched cache line. When the buffer is full, incoming prefetches are just discarded. Each processor has a 64K-byte data cache, which is direct-mapped and copy-back with a cache line size of 16 bytes. The caches are *lockup-free* [Kroft 81], thus allowing multiple outstanding data requests. A 16-entry outstanding request list (ORL) is used to keep track of pending requests, some of which might then become *hit-wait* accesses.

As mentioned above, the cache hierarchy is used only for storing shared data. Instructions and private data references are assumed to hit in the local memory with the processor incurring no time penalty. Since the configuration of the interconnection network is not of our primary interest, we simply assume that the memory bandwidth is sufficient

for any application and that a fixed latency time is used when a request travels through the network. The one-way latency time between caches and the global memory modules, that is, the one-way network latency, is 40 cycles. Hence, a reference that misses in caches incurs a total latency of at least 80 cycles (L_m). A read miss to a dirty block owned by another cache or a write request to a block that is already cached elsewhere will need at least two network round trips, i.e., 160 cycles. Although we do not model the contention in the network, we do take into account interference at the caches and at the memory directories since each cache and directory module can process only one request per cycle. Lock/unlock and barrier requests are handled using a queue-based protocol in the directory. A request waiting on a synchronization operation will not cause extra traffic for the caches and the network.

5.4.2 Simulation Environment and Benchmarks

We have developed a direct-execution simulator that simulates important events of interest in a shared-memory multiprocessor, while the computation instructions are directly executed by the host machine. A simulation module is assigned to each component of the architecture. Processor, cache, memory, and network simulation modules are built in the simulator and are replicated as needed for a given configuration (as shown in Figure 5.4). Clocks are associated with simulation modules. The modules behave like light-weight threads within a single UNIX process. A kernel of the simulator always drives the module with the earliest clock time. When a processor module is simulated, a user context of the simulated benchmark is restored and the corresponding user thread is directly executed until an operation that may have global effects, like a shared reference or a synchronization primitive, is encountered. As a result, the interactions among modules in the system reflect the dynamic execution of the user program with the correct actual delays and interleaving order of the global requests. The number of instructions executed is counted based on the annotations of basic blocks and global events which are instrumented by a preprocessor. Hence, the instruction stream is obtained and can be simulated as in the real execution.

The main goal of the simulator is to effectively simulate the execution of ‘‘parallel’’ programs in a uniprocessor environment. Parallel applications are developed in C using a set of parallel constructs, such as locks, barriers, forks. The constructs are expanded by a macro processor and then are translated to a set of simulator ‘‘system calls’’ by a code instrumenting program. Since the entire simulation and user execution reside in the same

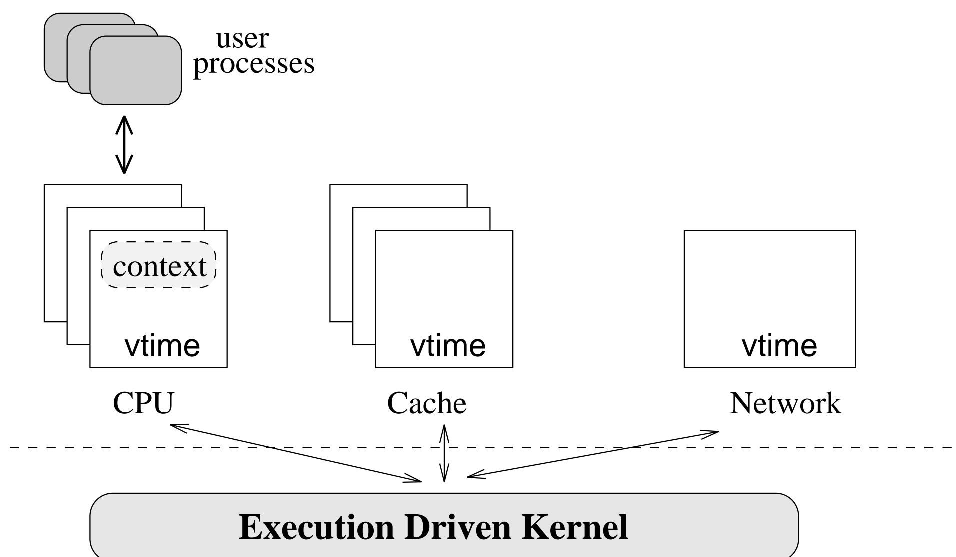


Figure 5.4: Direction Execution Simulator

process, when the execution reaches those system calls (parallel constructs), the simulator kernel will handle the internal context switching. Locks and barriers are implemented by the kernel synchronization mechanism and fork calls will cause the simulator to create new user threads running in the process. The advantages of this technique are that an efficient evaluation can be performed in an uniprocessor environment and that the simulated architecture modules can be easily developed and controlled. The idea of a simulation where the simulator and user programs execute concurrently appears in several simulators [Davis *et al.* 91, Brewer *et al.* 91, Grunwald *et al.* 91]. Specifically, our implementation is similar to Tango [Brewer *et al.* 91]. However, the differences are:

1. User synchronizations are handled in the simulator kernel, instead of UNIX processes and semaphores. This can significantly reduce the simulation overhead.

2. Each simulation module is running independently as an internal thread in the process. It allows the system to efficiently switch from one simulation module to another and thus reduces the simulation time. It also imposes a structured organization for the development of simulation components.

3. The original instruction stream is captured so that studies of branch prediction and instruction lookahead are possible in the multiprocessor environment.

Because of the execution-driven simulation paradigm, the total number of instructions executed in some dynamically scheduled programs (e.g., Cholesky -- see below) may vary across the architectural configurations. In addition, the amount of barrier waiting time can be quite variable. Thus, to avoid providing misleading statistics on total execution time per processor, the synchronization times that we will show will not include the time when a processor stalls for a barrier to complete.

Table 5.1: Benchmarks characteristics - average numbers for a single processor in the 16 processor simulation

Applications	Instructions executed (K)	shared reads (K)	shared writes (K)	Locks	Barriers	shared data size (K bytes)
Matmat	8,723	1,355	421	0	82	2109
Mp3d	7,231	1,334	426	10	60	3673
Water	21,173	1,033	72	8,737	25	156
Cholesky	38,233	6,809	524	5,671	81	6403

The benchmarks we used are Matmat and three SPLASH benchmarks [Singh *et al.* 92]. To study the architectures with a moderate cache size, we run the benchmark programs with larger data sets than what are provided in the benchmark. Table 5.1 summarizes the statistics collected on these benchmarks once their parallel sections are started up until the program is completed. Only shared references are recorded in the table and the column below ‘‘shared data size’’ indicates the total size of global shared area which is explicitly allocated in the program. Matmat is a blocked matrix multiplication program, run with two 300×300 matrices with proper cache buffer and block setting so that the effects of cache size and block size can be balanced. MP3D is a particle-based fluid flow simulation program. We ran MP3D with 100,000 particles in a $14 \times 24 \times 7$ space array for 10 time steps. Water, an N-body molecular application, was run with 288 molecules for 4 time steps. Cholesky performs parallel factorization of a sparse matrix, run with the test set `bcsttk15`.

5.4.3 Model Implementations

In this study, we experimented with three architectural choices: baseline caches, caches with lookahead prefetching, and caches with software prefetching. In prefetching caches, prefetching was performed for read misses only. Although write misses or writes on clean write-shared data can be helped by the use of an exclusive-prefetch, the prefetching overhead may be still substantial because it increases invalidation misses in other processors that are still using the data. Instead, our default consistency model is weak consistency [Dubois *et al.* 86], under which the write latency can be mostly hidden.

The baseline cache hierarchy without prefetching was described in Section 5.4.1. In both the baseline cache and caches with prefetching, we assume that reads are blocking, that is, a processor stalls on a miss or hit-wait until the data is ready in the cache. Our model of lookahead prefetching for each processor is the hardware lookahead prefetching scheme, which has been described in Section 3.3.

We now describe the methodology that we followed to simulate software prefetching. To achieve the maximum possible benefits of software prefetching, we try to identify those accesses which have the highest cache miss rates by profiling the benchmark programs. The instruction addresses of the cache misses (candidates for prefetching) are recorded by running each program based on the same configuration of the study with the same data set. After the accesses for prefetching are identified, we instrument the codes with prefetch instructions. Since there is no prefetching compiler available to us, we manually insert prefetch instructions related to these high miss frequency items at the source level based on the following strategies:

1. We estimate the execution time of an iteration for a loop. A data item accessed in the loop is prefetched one or more iterations ahead depending on the relative values of the loop execution time and the memory latency (e.g., 80 cycles).

2. Taking the block size of 16 bytes into account, we may unroll a loop or introduce a prefetch predicate to avoid unnecessary prefetches. Also, we perform the loop splitting such that in the prologue loop prefetches are started, and in the epilogue prefetches for the last final iterations will be suppressed.

3. By default, each prefetch will bring one cache line. We allow the possibility of block prefetching. If our profiling information detects that prefetching the whole

data object at once would be beneficial, we pipeline prefetch requests by a block prefetch. As a result, a prefetch request *initiated* by a single prefetch instruction may trigger the cache to *issue* several data access requests to memory modules.

4. If the address of the prefetch can only be determined dynamically, e.g., depending on result of a previous load (we call it a *load dependent* access), we attempt to schedule the instruction source of the dependence ahead in the instruction stream to provide as large a non-blocking span as possible.

Based on the above strategies, we try to keep the overhead associated with each prefetch in our implementation as low as possible. We analyze the following possible sources which may contribute to software prefetching overhead:

- The prefetch instruction itself requires the processor execution. As a result, the prefetch overhead is at least one cycle per prefetch instance.

- We assume that the data address of the prefetch can be specified in the instruction. The address computation of prefetch instructions is generally combined with the corresponding loads and the overhead for address computations is nearly nil, since we use the compiler to perform all optimizations. Note that when the prefetches are moved away from their loads due to instruction scheduling, the cost of prefetching may increase because address expressions cannot be completely eliminated.

- Each prefetch instruction implicitly fetches one cache line. An additional instruction is needed when the prefetch size is greater than one cache line size to specify the prefetch size for block prefetching,

- Prefetching may increase register pressure as a result of loop splitting and unrolling. The additional spilled code will contribute to the prefetching overhead.

Overall, the overhead in our implementation is relatively low (just over one instruction per prefetch instance) in order to emulate an effective compiler algorithm.

Next we describe the complications and limitations of software prefetching in our implementation. Adding prefetching to real benchmark programs gave us a different

experience from adding to kernel programs, such as Livermore Loops, or Nasa Kernels. For example, the number of iterations in a loop nest is usually variable. We need to add additional code when we perform loop unrolling and loop splitting. Also, the starting element of a data array in the loop does not necessarily align to a cache line boundary. This problem is getting complicated when the starting index and the ending index are depending on previous computations. Although a compiler has information to align data arrays, most of those difficulties are due to the fact that the execution depends on certain variables or input data, a problem which a compiler has difficulty to deal with as well. We therefore might end up with splitting loops into various sections and expanding the code to compromise any possible run-time results by adding more IF condition statements.

Another noteworthy point is that the codes were inserted with prefetches based on ‘‘as-is’’ benchmarks. We instrument the programs from the original SPLASH benchmarks, with some portion of codes being rewritten to better perform software prefetching. Admittedly, results could be different if programmers, with prefetching in mind, reorganize the entire codes so that prefetching insertion would be more effective and efficient by a compiler. A good solution in compiler design may be that programmers specify more program-specific hints or constructs, and then the compiler takes care of low-level prefetch insertions.

In summary, the goal of the instrumentation is to emulate a compiler algorithm that will carefully generate effective prefetches. Although these strategies may soon be within the realm of current optimizing compiler technology, it is our contention that our results will be optimistic.

5.5 Simulation Results

In this section, we present experimental results that contrast the hardware and software prefetching approaches. We first give general comparisons and then examine the benchmarks in more detail to better understand the effectiveness of prefetching. Section 5.5.3 discusses some negative effects introduced by prefetching. In sections 5.5.4 and 5.5.5, we study the effect of variations in memory latency and the impact of consistency models. Then in Section 5.6, we investigate the combination of hardware and software approaches.

5.5.1 General results

Figure 5.5 shows the simulation results of the average execution time from 16 processors with respect to various approaches. The left-most bar shows the breakdown of the execution time of the baseline cache (BASE). The next two bars are for hardware-based lookahead prefetching (HW-pf), and software prefetching (SW-pf) respectively. We present the data by normalizing the total execution time with respect to the baseline organization. Each bar contains several sections. The *exec* section denotes the time to execute instructions -- it also includes the extra instruction overhead for executing software prefetching instructions, necessary address/size computations, and execution of possible extra spilling loads due to the increase of register pressure (determined by the compiler optimization); *read* and *write* indicate the fraction of processor stall time for reads and writes ; *delay* shows the delay of demand accesses resulting from handling prefetch and tag updates in the cache; and *synch* gives the time waiting for lock and barrier accesses.

We look at the results by examining each stall time component. The instruction execution time corresponds to the processor utilization. As shown in Figure 5.5, the processor utilization is between 13% in Mp3d and 75% in Water. There is much room for improvement on the read access penalty by the lookahead (HW-pf) and software prefetching (SW-pf). A comparison between the BASE and the prefetching schemes shows that hardware prefetching can significantly reduce the read stall time by 10%-39% of the original total cycles, while software prefetching also achieves remarkable reduction (by 15%-43% of total cycles). We will dwell on these numbers in more detail in Section 5.5.4.

The next component contributing to CPU stall time is the write penalty. Since the system is based on the Weak Consistency model, it is not surprising that only a small portion of the write penalty is seen. Under a weak consistency model, the *write* portion is the sum of the time spent at a synchronization point to wait for previous pending writes to complete, the stall time because of the write buffer being full, and the time waiting for a read miss which needs the same line as another pending write. However, since those situations are rare, the sum of the stall times is nearly negligible with respect to the total CPU stall time.

As to stall time due to synchronizations, the overhead looks relatively small compared with other stall contributions. Synchronization delays are slightly visible only in Water and Cholesky, where synchronization activity is more apparent (c.f., Table 5.1). The synchronization overheads are not modified significantly by prefetching.

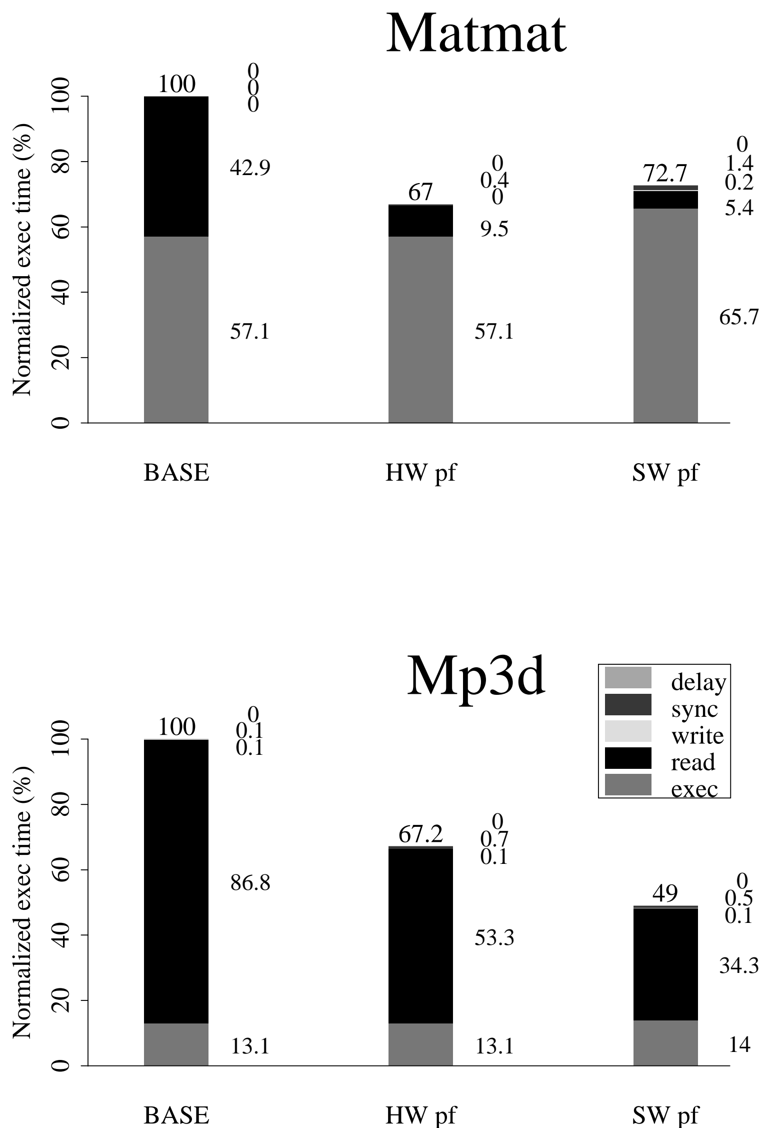


Figure 5.5a: Simulation results

The last component is the cache interference, an overhead introduced by the prefetching. It includes the number of processor stall cycles as a result of the cache handling the requests and tag updates of prefetched blocks. As seen from the *delay* section in Figure 5.5, the number of busy cycles are very small (only 0.05%-0.6%). Hence, this negative effect is almost negligible.

Extra instruction execution time is yet another overhead, which is present only in SW-pf. As shown in the *exec* section of SW-pf, the SW-pf instruction overhead can be

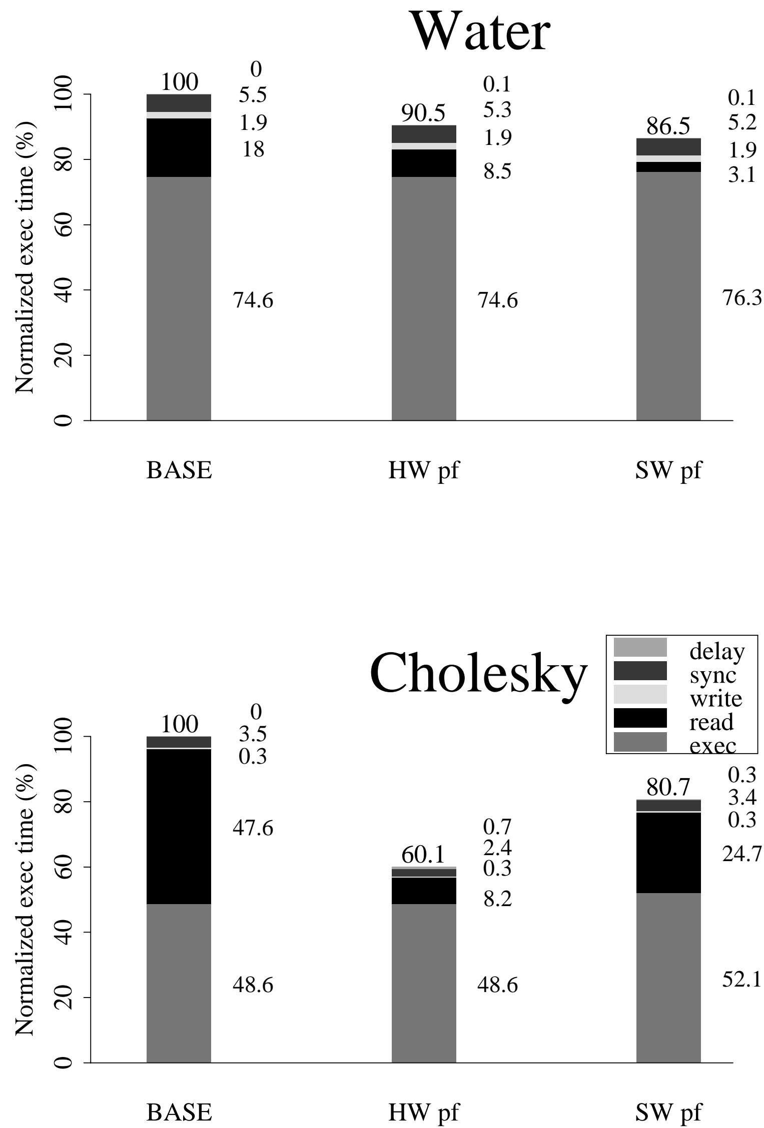


Figure 5.5b: Simulation results

substantial. The portion of normalized time due to the software overhead ranges from 0.9% in MP3D to 8.6% for Matmat and may offset part of what was gained in reducing the read penalty.

In summary, the results show that when the processor utilization is not high, the total execution time (read penalty) can be significantly reduced by prefetching. Under a weak consistency model and given sufficient memory bandwidth, writes and synchronizations do not contribute much to stall time. We also observe that the side effects of prefetching such

as processor interference and cache busy time are insignificant. However, the instruction overhead in SW-pf is substantial.

5.5.2 Detailed Analysis

We examine further the effectiveness of prefetching by looking in more detail at the individual behavior of the four benchmarks (cf. Table 5.1).

Matmat

Matmat is a blocked matrix multiplication program in which almost all references are regular and sequential. Both HW-pf and SW-pf perform as well on the Matmat benchmark where data access patterns are regular (read penalty reduced by 77% and 87% respectively). Even so they do not eliminate all of the read penalty. In HW-pf some of the read penalty is contributed by a portion of *hit-wait* cycles in the first iterations. Another portion of the remaining read penalty stems from the fact that the blocking technique tries to localize the referenced domain of inner loops and thus data blocks prefetched at the last iteration of an inner loop are generally unused. Similarly, SW-pf has a portion of *hit-wait* cycles. Since the register pressure is already very tight because of tiling of the inner-most loop, loop splitting due to prefetching would exaggerate the pressure. By looking in more detail at SW-pf shows that the execution time of one iteration of the inner-most loop (unrolled by factor of 2) takes 85 ideal cycles. It has been increased by 11%, compared with the execution time of the original code (76 cycles for two iterations). The increase comes from the prefetch instructions and extra spilling code. This explains the magnitude of the instruction overhead (8.6% of total time) for SW-pf. It indicates that the SW-pf should be more conservative when taking into account optimizations arising from locality considerations.

MP3D

The two data structures that account for most of the references are particles and space cells. The particles are statically allocated; the space cells are accessed in a relatively random manner depending on the location of the particle being moved. In such an application where data structures are more complex, SW-pf exhibits better performance in reducing the read penalty than HW-pf (38% for HW-pf in MP3D vs. 60% reduction for SW-pf). Although HW-pf has no difficulty in prefetching a particle record, it is not good at dealing with space cells because their locations vary with time. Thus only roughly half

of the cache misses are covered through HW-pf. In contrast, SW-pf performs much better than HW-pf. SW-pf can statically prefetch particle data and use load dependent prefetches to get the space cell when the address of an associated particle is determined. Moreover, particle objects and space cells can be prefetched by a single block prefetch instruction. Consequently, several memory access requests triggered by only one prefetch instruction can be pipelined to the memory system. The prefetching of space cells is scheduled so that it can be performed in parallel with other computations. Therefore the latency of the load dependent prefetch is further hidden. The use of block prefetches is also the reason that MP3D has a negligible instruction overhead in Figure 5.5.

Water

The main data structure is an array of molecules where each element holds all the data for one molecule. Each molecule requires about 38 cache lines. Data accesses preserve spatial locality in the *intramolecular* computations and data access patterns are predictable *intermolecular* computation phases. Since the ratio of the number of shared references to instructions is very small, the instruction time accounts for a large portion of the total execution time (cf. Table 5.1). In addition because the cache can almost hold the entire working set, most of the accesses result in cache hits. Therefore the read penalty contributes only 18% of the total execution time. While this benchmark has predictable access patterns but with small nested loops, the SW-pf moderately outperforms HW-pf (52% for HW-pf vs. 83% for SW-pf). The read penalty reduction is remarkable but does not improve performance that much since the read penalty is relative small. It is easy for both HW-pf and SW-pf to handle the shared references in the *intra* and *intermolecular* computation phases, with SW-pf being more effective. The main reason is that each computation of a molecule involves two or three nested small inner loops with only a small number of iterations in each level of loop. SW-pf simply prefetches data for all the iterations at one time, whereas the small loops hinder HW-pf from gaining sufficient prefetching distance. Loop unrolling and blocking techniques in the original code may remove the obstacles for HW-pf. However, unrolling multiple levels of loops may potentially expand the code by a significant amount.

Cholesky

Cholesky is dynamically scheduled with coarse task granularity (about 86,000 shared

references per task). Each task works on supernodes, which are sets of columns of a very large but sparse matrix. The input data file is a 3948-by-3948 matrix with only 56934 non-zeros. The primary operation is the column modification which involves the addition of one column into another in order to cancel a non-zero in the upper triangle. Since all non-zeros belonging to a certain column are stored contiguously in an array and the row numbers of the non-zero elements are stored in a compressed manner, the program iterates on the array of row numbers to find matching rows and then fetch the non-zero to perform computations. As a result, the starting and ending values of loops are generally unknown at compile time. In the benchmark, where the data structures are regular with input-dependent accesses, the hardware scheme performs better than SW-pf (82% vs. 48%). The HW-pf scheme can benefit from the assignment of large supernodes to the processors by sequentially prefetching the array and dynamically extracting data access patterns for the accesses of non-zeros. Similarly, SW-pf can prefetch the data for accesses to the array holding row numbers. However, it is conservative in prefetching the non-zeros by using load dependent prefetches only after the row pointer is known. This will usually cause prefetched blocks to arrive in the cache too late and thus to contribute a large portion of *hit-wait* cycles to the read penalty. In addition, because the starting and ending values are run-time variables, the code is significantly expanded as a result of loop unrolling and splitting as well as prefetch insertion. For example, an IF statement is required in the prologue loop to align the prefetch access on the cache line boundary. Hence, the instruction execution time is increased.

To summarize, our data show that SW-pf and HW-pf can achieve good performance improvements in programs with regular access patterns. HW-pf can handle applications with input data dependence if the loop granularity is not too small. SW-pf is flexible and even good at dealing with programs with complicated but well-organized data structures. However, the benefit of software prefetching may be offset by the extra overhead it incurs.

5.5.3 Negative Effect of Prefetching

As we have discussed in Section 5.3.5, one of the concerns for prefetching in multiprocessors is the negative effects that may offset the performance benefits. In this section, we examine the issue of negative effects brought by prefetching, including memory traffic and pressure on a cache for holding the working set.

Prefetching typically increases memory traffic. A multiprocessor is generally sensitive to such an increase in memory traffic, especially in the case of bus-based shared memory multiprocessor. The increase usually comes from: (1) prefetches of unused data lines, (2) extra cache misses due to conflicts with the current working set, (3) extra invalidates owing to additional write-sharing by prefetching, (4) the increase of invalidation misses due to exclusive prefetches. Since we do not perform prefetch for writes, the last problem does not exist in our study.

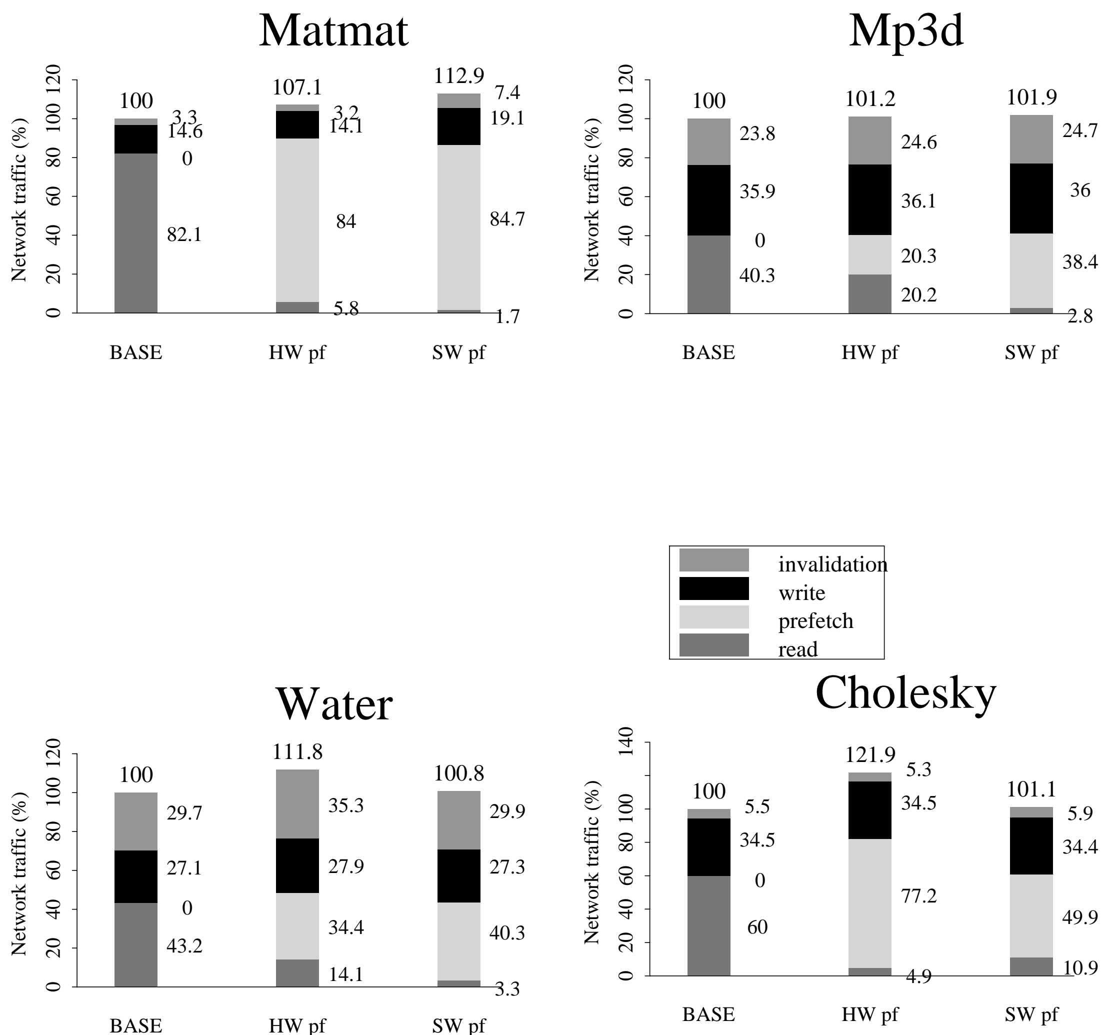


Figure 5.6: Network traffic

In Figure 5.6, we present the increase in network traffic. In terms of the number of

requests appearing in the network, we consider four kinds of network traffic: read misses, prefetch requests, write requests (write misses and write hits on clean), and invalidates. From these results, we observe that the total number of memory requests increases, as expected, for both types of prefetching for all benchmarks. However, the increase due to prefetching (especially SW-pf) is relatively insignificant with respect to the total traffic. Most of the memory traffic increase stems from the fact that the total requests of read misses and prefetches are greater than those of read misses for the baseline cache. Since prefetching may fetch write-shared data, a slight increase of write requests and invalidates can be also observed in the figure. In general, SW-pf is more conservative in introducing memory traffic than HW-pf. The reasons are that HW-pf has less information to avoid sending unnecessary prefetches to the system and that data blocks prefetched during the last iterations are usually unused. The traffic increase is more significant in benchmarks with small iterations, such as Water, where the penalty reduction by HW-pf is less than that by SW-pf, but where HW-pf brings more network traffic. One exception is Matmat, where SW-pf results in more network traffic than HW-pf. However, the increase is mainly because more writes and invalidates are issued since there is more prefetching of write-shared data.

Table 5.2: Proportions of Conflicts in the direct-mapped sets

Programs	Hardware			Software		
	ws ⇔ ws	ws ⇔ pf	pf ⇔ pf	ws ⇔ ws	ws ⇔ pf	pf ⇔ pf
Matmat	.924	.076	0	-[a]	-	-
Mp3d	.976	.024	0	.925	.062	.013
Water[b]	-	-	-	-	-	-
Cholesky	.710	.174	.117	.961	.034	.005

[a] Miss rate is too small (< 0.001).

[b] There are very few conflict misses left, since most of the data set fits in the cache and misses are mainly caused by invalidation misses.

To examine the impact of prefetching on the working set in the cache, we estimate the negative effect by measuring conflicts between the working set and prefetched data. We record the information of replaced data lines in a ‘‘shadow’’ direct-mapped cache with the same size as the data cache. If a cache miss finds a matched entry in the shadow

cache, we record the status of both replaced and current blocks. It indicates that the miss is due to a conflict with previous access[2]. As the prefetching schemes effectively reduce the number of cache misses, the miss ratios of the four benchmarks are generally low. Among the remaining misses, we are interested in conflict misses. Table 5.2 gives the proportions of those conflict misses among three categories: conflicts within the current working set itself, between the working set and prefetched blocks, and between prefetched blocks themselves. The results show that a large portion of conflicts occurs among data in the working set itself. When a prefetched data arrives in cache at a time close to the actual use, the probability of conflicts with the current working set is small. We also see that HW-pf in Cholesky brings more prefetched data than necessary and, thus causes more conflicts with data in the cache. This can be explained by the evident increase of data read (read misses and prefetches) traffic in the network, as shown in Figure 5.6.

To sum up, we observe that the negative effect of prefetching in network traffic and conflicts with the working set is not severe. The increase of network traffic is very small for SW-pf, whereas HW-pf may give a slight increase. Most conflict misses are caused by the working set itself.

5.5.4 Effect of Memory Latency

In this section we explore how variations in the secondary cache and main memory latencies influence the performance of the three prefetching schemes. We consider three sets of latencies: the one used previously (L_m = 80), one where we consider a processor twice as slow (L_m = 40), and one where the main memory latency is doubled (L_m = 160) with the rationale here that our 16 processor system might be a subset of a larger multiprocessor. In Figure 5.7, we show the read access times for these three organizations normalized with respect to the no-prefetch BASE default case (L_m = 80). The read access penalty is decomposed into two sections: *read miss*, the stall time due to cache misses, and *hit-wait*, the waiting time for a prefetch which is issued too late. In order to have a fair comparison for SW-pf, we modified and moved around some prefetch instructions in an attempt to provide a sufficient prefetching span for large latencies.

As can be seen in Figure 5.7, the reduction in the read penalty slightly degrades as the memory latency increases. This illustrates that both HW-pf and SW-pf still can be effective,

---

[2] Note that this metric just includes the number of conflicts which the current data have with most recently replaced data, instead of all of the conflict misses and capacity misses.

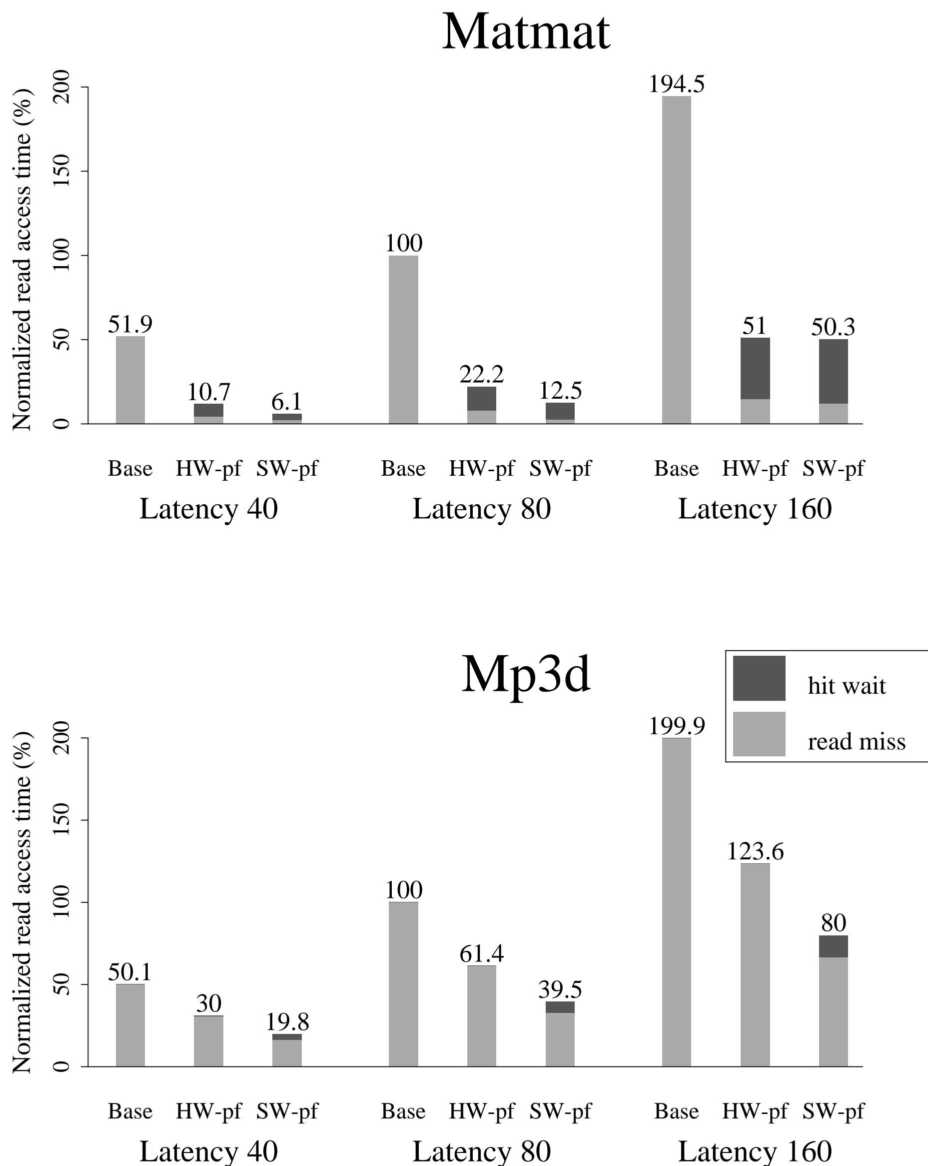


Figure 5.7a: Effect of memory latency

to a lesser extent, in tolerating large latencies by adjusting prefetching to occur several iterations ahead of the actual use. Note that since the number of instruction executed is generally fixed, the slight increase in the read penalty in SW-pf is more than compensated by the relative decrease in the overhead of the prefetch instructions. For example, when passing from L_m = 80 to L_m = 160, the overall execution time increases and the overhead from software prefetching (not shown in the figure), an almost constant number of instructions for each benchmark, decreases from 8.6% to 6% in Matmat, from 0.9% to

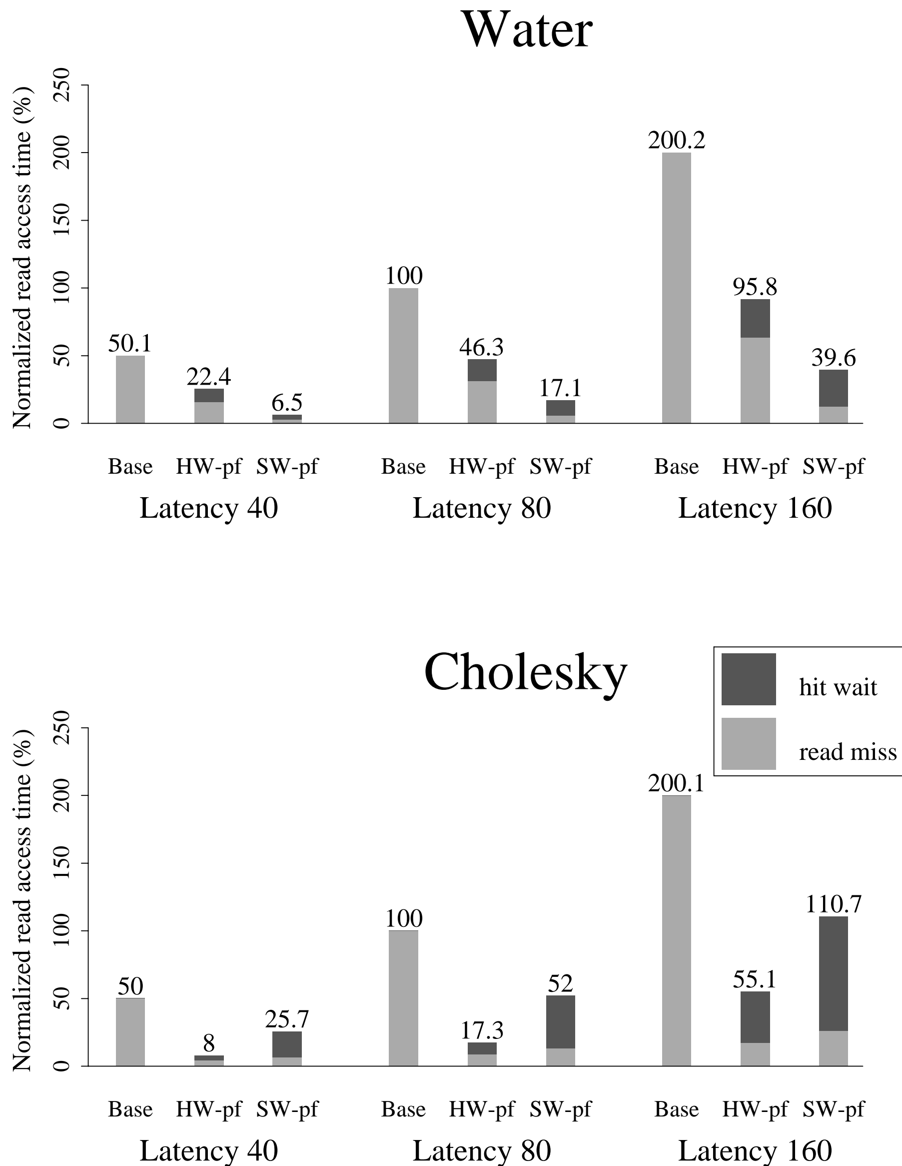


Figure 5.7b: Effect of memory latency

0.04% in MP3D, from 1.7% to 0.013% in Water, and from 3.5% to 0.3% in Cholesky. This leads us to conjecture that software prefetching should be more advantageous as the prefetch overhead becomes less significant with an increase in latency.

The cost of the *hit-wait* cycles is particularly important in prefetching. The read penalty in HW-pf contains a fair amount of *hit-wait* time. In this scheme, the lookahead mechanism needs to be reset to the value of the PC after each incorrect branch prediction. Therefore, the first few prefetches are not yet one ‘‘memory latency time’’ ahead of when their data

will be used. This phenomenon tends to be serious in those programs with nested inner loops with only a few iterations such as Matmat and Water. For SW-pf (cf. Cholesky), the *hit-wait* cycles are mostly contributed by the load dependent prefetches, which are constrained by the data dependencies. While SW-pf is generally able to identify most of the cache misses, there remains the challenge of scheduling useful computations to overlap with the prefetches, a task that becomes more difficult as latencies get larger.

To summarize, we show that the prefetching schemes can still be effective when the memory latency increases. The stall time due to *hit-wait* accesses is a significant portion of the read access penalty when the latency is large and will affect the relative gains of prefetching.

5.5.5 Impact of Consistency Models

The simulation architecture that we have been using so far assumes *weak consistency* (WC). Under this model, memory references between synchronization accesses can be completed out of order, subject of course to dependencies, and thus WC exploits overlap among memory accesses. This removes some constraints of *Sequential Consistency* SC, the strictest model requiring that both reads and writes be blocking; the processor must stall on a miss until the data is ready in the cache, although there may be several non-blocking prefetches in progress simultaneously. In order to show the impact of consistency models on the effectiveness of prefetching, we performed experiments under the sequential consistency model.

Figure 5.8 presents the experimental results comparing prefetching under SC and WC. We break down the execution time (normalized to the baseline under WC) into the same five sections as before (recall Figure 5.5). As can be seen from Figure 5.8, the overall execution time for the baseline architecture under SC (second bar) increases from 6% to 80% over that of WC (first bar). This increase is because of the significant portion of the write stall time. Similar results have been observed in the literature [Gharachorloo *et al.* 91a, Mowry & Gupta 91, Zucker & Baer 92]. When looking at the effects of HW-pf (third bar and fourth bar) and SW-pf (fifth bar and sixth bar), the reductions of the read access penalty under SC are generally similar to those realized under WC. The only subtle effect (not obviously seen in the figure) of consistency models on prefetching is that the read stall time slightly increases from SC to WC. This is because the latency of prefetch is likely to be hidden during the time that reads are blocking on pending

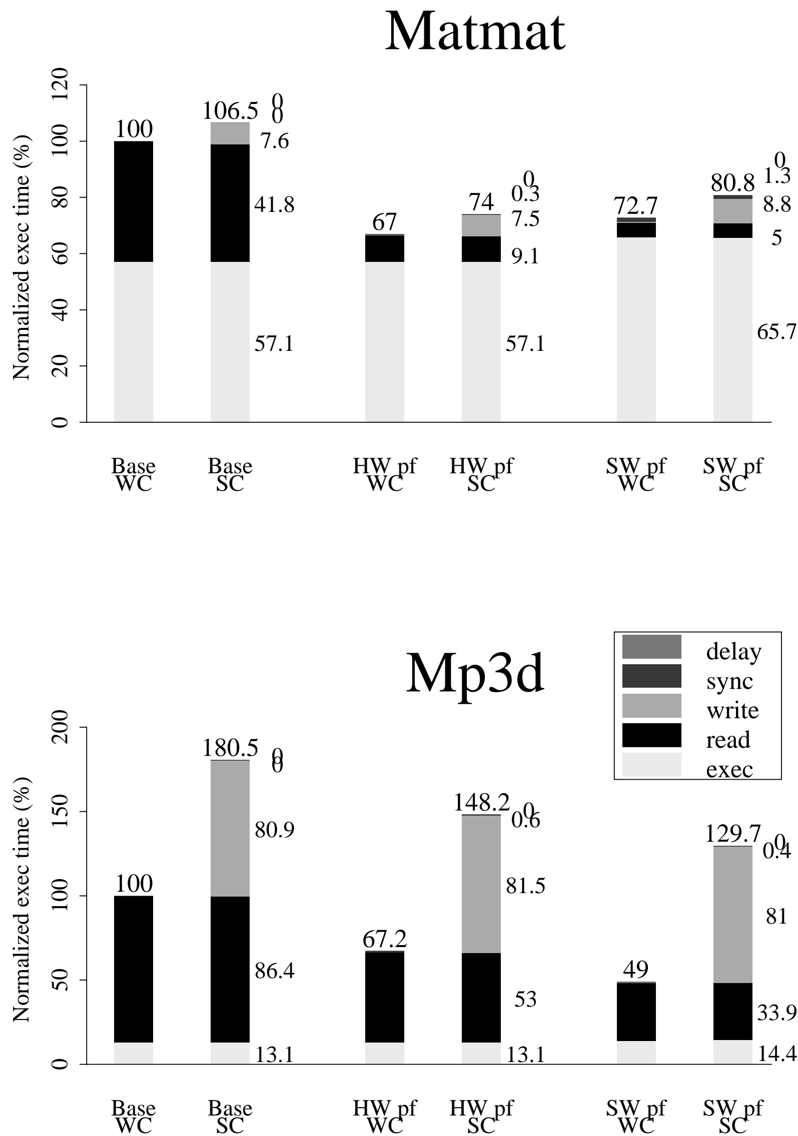


Figure 5.8a: Prefetching based on weak and sequential consistency

writes in SC. This is particularly true of HW-pf, where the LA-PC, under SC, has a good chance to get far enough in advance of the real PC because of write stalls.

5.6 Combining Hardware and Software Prefetching

In this section, we propose a combination of hardware and software solutions to prefetching. The main idea is that the compiler inserts prefetches for user's semantic data objects that

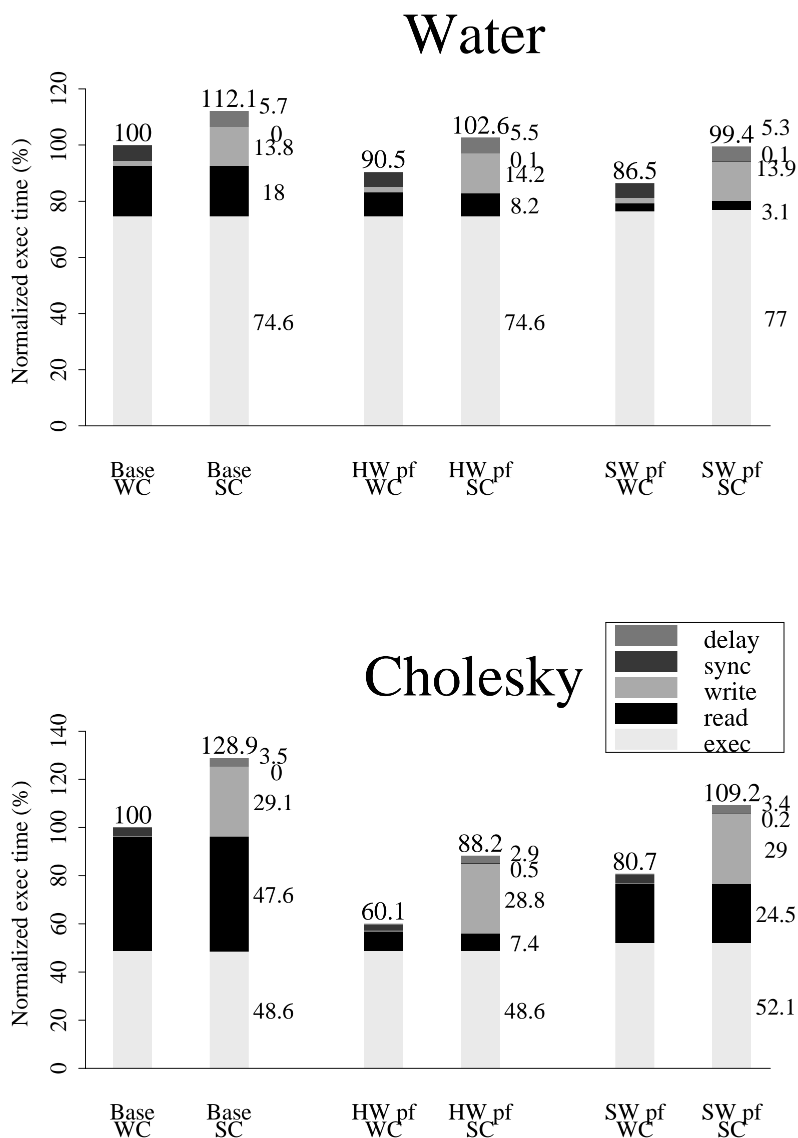


Figure 5.8b: Prefetching based on weak and sequential consistency

can be of any size, not necessarily of a cache line, in a manner more related to the program information available to the compiler, and that a hardware supporting unit will take care of accesses in the loop with a closer relation with the hardware organization. In addition to bringing appropriate data belonging to the working set, software prefetching on data objects reduces the instruction overhead incurring in the inner loop. Also, the problem of explosive code expansion in the software prefetching can be avoided by using the hardware component that handles loop or input-dependent accesses. To achieve maximum gains,

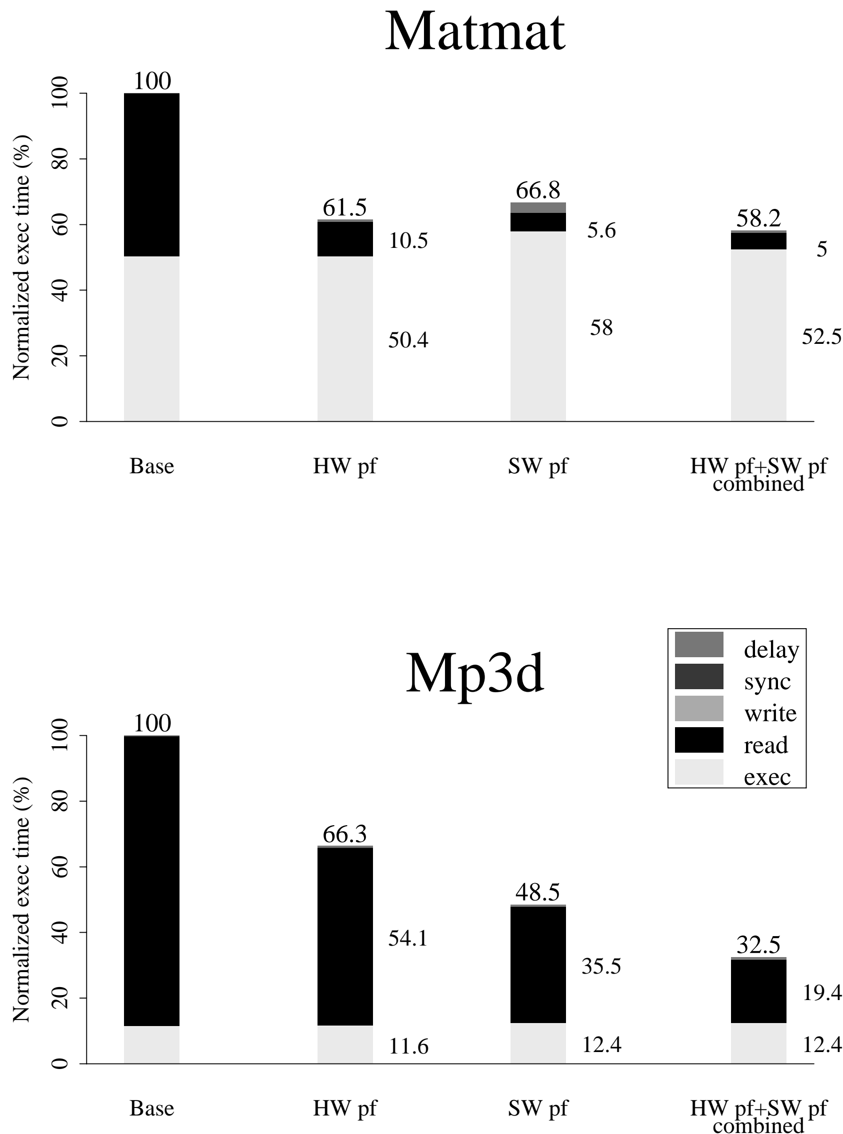


Figure 5.9a: Effectiveness of combining HW-pf and SW-pf

the hardware part is aimed at prefetching data from the secondary cache to a relatively small primary cache and the software part is aimed at a large block fetch from memory modules to the secondary cache. The reasons are that the hardware scheme can collect the dynamic information and perform well at small latencies and that the software scheme can be good at predicting the necessary working data that are brought into the larger secondary cache with only a few prefetch instructions. By adding a special control instruction to the instruction set, some unnecessary prefetches in the hardware prefetching scheme can

be further reduced by using the instruction as a control hint to enable (and disable) the hardware mechanism. Such control hints can be inserted around a loop body so that the hardware unit will operate only during the execution of loops.

Previously proposed approaches can be employed to realize the combined scheme. As we have shown in Chapter 4, our proposed hardware scheme can be effective for small caches. Gornish *et al.* [90] have proposed a compiler algorithm to identify the data for block prefetches. It would be promising if a modification of the algorithm is made for programs with more complex data structures in the context of non-binding prefetching, rather than of binding prefetching.

We performed experiments for studying the effectiveness of the HW-pf and SW-pf combined architecture. In the experiment, we consider a similar architecture to the earlier study, except that each processor has a 32K-byte primary cache (C1) backed up by a 256K-byte second-level cache (C2). Both caches are direct-mapped with a cache line size of 16 bytes and *lockup-free*. The one-way latency time between C1 and C2 is 5 cycles and thus the delay for a miss in C1 with a hit in C2 is 10 cycles. Misses in C2 trigger requests to the global memory modules. The one-way network latency is 35 cycles. Hence, a reference that misses in both caches incurs a total latency of at least 80 cycles. In the experiment, we modify the strategy for prefetch insertion in software prefetching: we do not prefetch data in inner-most loops, we do not perform loop unrolling and splitting, we insert prefetches for user data structures to be used (regardless of cache size, line size), and we move prefetches far ahead of actual use (may even move to a location before the loop).

Figure 5.9 gives the simulation results of the new architecture with the combined hardware and software schemes. As can be observed from the results, the read access penalty has been further decreased when compared to either the hardware approach or the software approach. This ends up with total reductions of the read penalty in the baseline by 90% for Matmat, 78% for Mp3d, 88% for Water, 80% for Cholesky. The instruction overhead of the new scheme is relatively small[3] when compared with the software approach. The portion of total normalized time due to the overhead ranges from 0.8% in Mp3d to 2.1% in Matmat. Overall, the total execution time can be significantly improved by the combination of software and hardware schemes.

---

[3] There is nearly no decrease in Mp3d. This is because SW-pf already performed block prefetching in Mp3d.

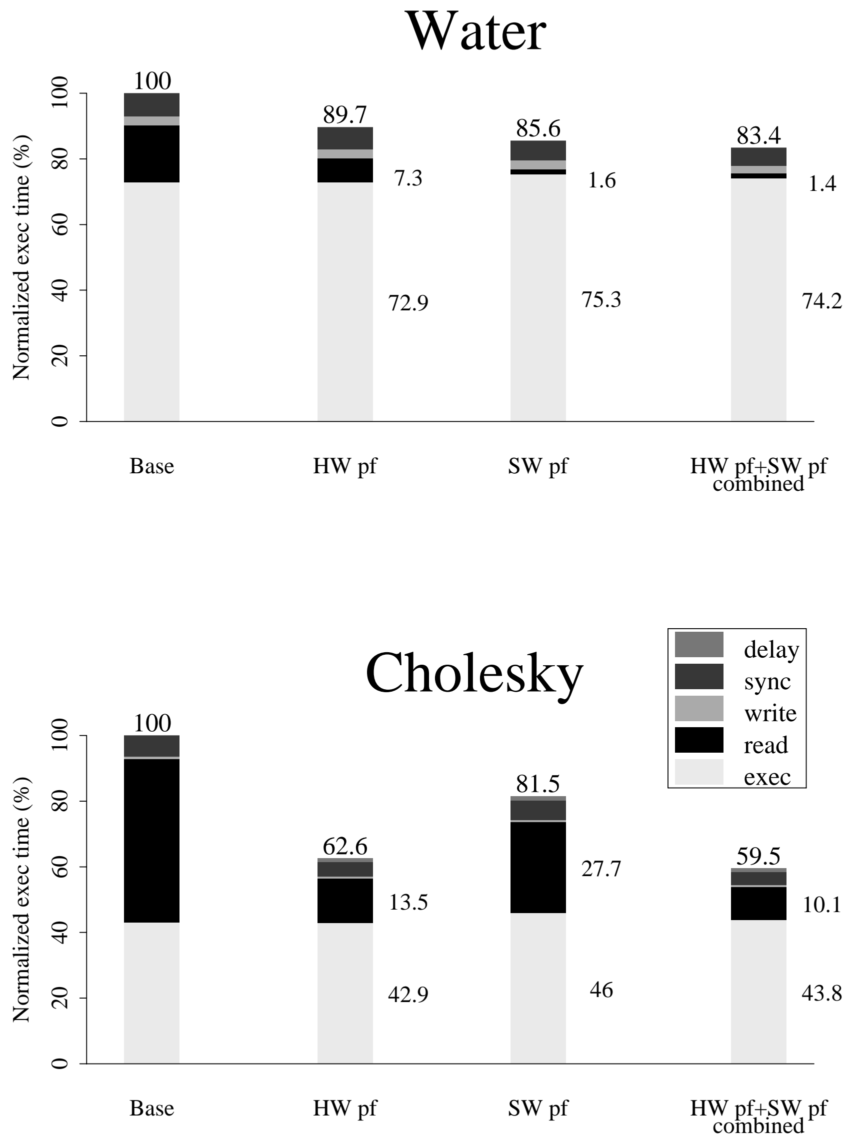


Figure 5.9b: Effectiveness of combining HW-pf and SW-pf

5.7 Summary

In this chapter we have studied the performance of hardware-based and software-directed prefetching schemes. Our qualitative comparisons indicate that in the domain of linear array references, both hardware and software schemes are able to generate prefetches for cache misses. However, the software scheme may have a code expansion problem, while the hardware scheme has less information on whether the prefetched data will be

used or not. As to other complex data access patterns, the software approach may have more compile-time information to perform sophisticated prefetching, whereas the hardware scheme has the advantage of manipulating dynamic information (such as conflict misses or input data dependence).

The quantitative evaluation was performed by running direct-execution simulations of a shared-memory multiprocessor using four benchmarks. Our experiments confirm the above observations. We observed that the cache interference incurred by prefetching is almost negligible. The software approach has less negative effect on network traffic and conflicts with the working set than the hardware approach. However, the overhead due to the extra prefetch instructions and associated computations is substantial in the software-directed approach. The performance gain of reducing the read penalty can be offset by the slight increase in instruction execution time.

We also evaluated the impact of varying memory latencies and of consistency models. Our results show that the effectiveness of prefetching is slightly degraded by the increase of memory latencies. Reductions in read penalty by prefetching were similar under sequential and weak consistency.

Finally, we proposed and examined an alternative of combining the software and hardware solutions. The main idea is that software will use program user's semantic to prefetch data objects into a secondary cache and that the hardware supporting unit will take care of accesses in the loop and fetch the data into the primary cache. The new approach can combine advantages of both hardware and software approaches and at the same time avoid most of their negative effects. Our experimental results show that the new solution is very attractive in reducing data access penalty without incurring much overhead.

Chapter 6

Non-blocking Caches

6.1 Overview

So far, we have discussed hardware and/or software prefetching techniques that can eliminate, or at least reduce, the cache miss penalty by generating prefetch requests to the memory system to bring the data into the cache before the data value is actually used. These techniques exploit the overlap of computation with memory accesses *prior to* an actual cache miss. In contrast, non-blocking is a technique to take advantage of a *post*-miss overlap. A non-blocking cache (also called lockup-free cache) allows execution to proceed concurrently with one (or more) cache misses until an instruction that actually needs a value to be returned is reached. The basic idea in both non-blocking and prefetching caches is to hide the latency of (read and write) data misses by the overlap of data accesses and computations to the extent allowed by the data dependencies or consistency requirements.

In this chapter, we evaluate the effectiveness of the non-blocking technique on reducing the memory latency. We consider ways to improve the approaches by compiler-based optimizations (e.g., code rescheduling, software register renaming). We also propose a hybrid design which is a combination of non-blocking and prefetching approaches. From the simulation of ten SPEC benchmarks, our results show that hardware prefetching caches, which require extra hardware complexity, generally outperform non-blocking caches and that the prefetching caches are less sensitive than non-blocking caches to the increase in memory latency. The compiler optimizations that we propose can significantly improve the effectiveness of non-blocking caches.

The rest of the chapter is organized as follows: Section 6.2 gives some background information on non-blocking caches. Section 6.3 describes the processor and memory architectures under study as well as the evaluation methodology. Simulation results are presented in Section 6.4. Section 6.5 describes the compiler optimization algorithms and discusses the results. In Section 6.6, we propose and evaluate a hybrid design. Finally, we summarize in Section 6.7.

6.2 Background and Performance Issues

We start this section with a brief description of non-blocking caches, including supporting non-blocking writes in write buffers and non-blocking reads by the processor. We then discuss performance issues and the extra hardware requirements.

6.2.1 Non-blocking Caches

Lockup-free caches were originally proposed by Kroft [81]. In his design, the following three features are included:

1. Load operations are non-blocking.

2. Write operations are non-blocking.

3. The cache is capable of servicing multiple cache miss requests.

In order to allow non-blocking operations and multiple misses, Kroft introduced Miss Information/Status Holding Registers (MSHRs) that are used to record the information pertaining to the outstanding requests. Each MSHR entry includes (1) the data block address, (2) the cache frame for the block, (3) the word in the block which caused the miss, and (4) the function unit or register to which the data is to be routed. As we have indicated in Section 2.2, the terms ‘‘lockup-free cache’’ and ‘‘non-blocking cache’’ are used interchangeably in the literature. We can somewhat clarify this confusion by considering two distinct features: (1) a cache supporting multiple outstanding memory requests, but with a processor stalling on read misses (blocking loads), and (2) a processor supporting non-blocking loads and writes. Our view is that non-blocking loads are features specified in the processor, non-blocking writes are supported by buffering writes, whereas whether the cache allows multiple pending accesses or not depends not only on the presence of MSHRs, but also on the available cache bandwidth as defined by the interface between caches and memory modules. In the discussion of this chapter, we specifically focus on non-blocking writes and non-blocking reads. We first discuss related designs which have been proposed to support these two features.

To support non-blocking writes, write buffers are used to eliminate stalls on write operations. They permit the processor to continue executing even though there may be outstanding writes. Write buffers in conjunction with write-through caches are especially

useful in reducing the processor stalls on writes. For write-back caches (with write-allocate), write buffers are used to temporarily store the written value until the missing data line is returned. Another example is the write-back buffer used for temporary storage of replaced dirty blocks in a write-back cache. As a further example, the lockup-free cache in RP3 [Brantley *et al.* 85] supports non-blocking prefetches and multiple outstanding writes. The cache allows partial writes, i.e., writes to only parts of a line and bypassing loads, i.e., the datum is directly forwarded to the CPU when the needed word is available in the cache, even though the rest of the words in the same line may not be ready yet. Extensions to write buffers have been proposed. For example, write caches [Bray & Flynn 91], organized like regular caches, hold partial written data lines and allow multiple writes on the same line to be combined, thus reducing the total number of writes to the next level of the memory hierarchy.

In addition to the MSHR's associated with a non-blocking cache, non-blocking loads require extra support in the execution unit of the processor. If static instruction scheduling in pipelines is used in the processor, some form of register interlock (like a full/empty bit for each register) is needed for preserving correct data dependencies. For instance, the register file in the MC88100 [Motorola 90] includes a scoreboard register, which contains one such bit for each of the general-purpose registers. In the case of dynamic instruction scheduling, introducing out-of-order execution, a more complicated scoreboarding mechanism is required. In addition, both static and dynamic instruction scheduling strategies need interrupt handling routines that can deal with interrupts generated by the non-blocking operations [Hennessy & Patterson 90].

A consistency problem can arise when the processor allows non-blocking writes since a later (in program order) read may be needed before a previous buffered write is performed. If these two operations are on the same data block, an associative check in the write buffer or the MSHRs must be done to provide the correct value to the following read. When the processor is part of a shared-memory multiprocessor, the problem becomes more complex and the solution depends on the model of memory consistency that is adopted.

6.2.2 Performances Issues

As mentioned previously, the non-blocking operations exploit the *post*-miss overlap of computation and memory access while prefetching exploits the *pre*-miss overlap. The following is a brief qualitative view of the expected benefits for both types of overlap.

Non-blocking loads delay processor's stalls until the necessary data dependence is encountered. Non-blocking loads will become necessary for processors, such as super-scalar processors, capable of issuing multiple instructions per cycle [Sohi & Franklin 91]. However, the *non-blocking distance*, which is the number of instructions that can be overlapped with the memory access (e.g., instructions between the reference and the first dependent instruction), is likely to be small in the case of static scheduling. It can be increased when compilers produce code optimized for this potential overlap (see Section 6.5). Dynamic instruction scheduling (out of order execution) obtained at a significant increased cost in hardware complexity, can provide a larger non-blocking distance. However, the effectiveness is still subject to data dependence effects, branch prediction, and the size of the lookahead window provided by the architecture [Gharachorloo *et al.* 92].

By comparison, non-blocking writes have more chances to fully hide the write miss latency because the non-blocking distance is usually equal to the memory access time[1]. Moreover, the write buffer, a FIFO queue buffering pending writes, does not need a supporting unit in the processor. However, the other side of the coin is that even without a write buffer the write miss penalty may not be a large fraction of the total data access penalty. In this study, we consider write buffers both with read *bypass* (i.e., read misses which have priority over writes can bypass buffering writes) and with *no-bypass*.

In contrast to the non-blocking distance, the *lookahead distance*, i.e., the number of cycles which a prefetch request is generated ahead of the reference instruction, can be tuned by the designer and be as large as a small multiple of the memory latency. In our hardware-based scheme, its magnitude is constrained by effects such as the capacity of the RPT, the amount of regular data access patterns, and the success of branch prediction techniques. The implementation costs of prefetching caches, additional on-chip support units and more hardware complexity, are substantially higher than those of non-blocking caches.

Another noteworthy point is that the scoreboarding mechanism for non-blocking loads would increase the critical path time, whereas the prefetching cache, a supporting unit to the processor, will have less impact on the critical cycle. A final point to mention is that in the case of non-blocking loads, the binding of a register with a certain value starts at the

---

[1] In other words, the processor does not stall on most write misses. A stall would occur only if a write miss is followed by a read miss on a different word in the same block. In that case, the stall is attributed to the read miss.

moment the non-blocking load is initiated. In contrast, the prefetch request is *non-binding*; it is only a *hint* to bring a data line close to the processor without involving any register binding.

6.3 Architectural Models and Evaluation Methodology

In this section, we first describe the architectural models on which we will base our evaluation of non-blocking and prefetching caches. We then present our simulation methodology and the benchmarks used in the evaluation.

6.3.1 Processor-cache Models

As in our previous experiments, the baseline system consists of a CPU with a load/store architecture similar to the MIPS R3000 and an ideal instruction cache (thus no I-cache misses). The CPU has an instruction decoding unit, a fixed point unit (FXU), a floating point unit (FPU), and a cache interface. The decoding unit issues an instruction per clock cycle and the FXU can execute an integer operation in one cycle (perfect pipelining).

Because we need more precise comparisons between the pre-miss and post-miss overlaps, we refine the model to include timings on floating-point operations. The FPU, which behaves like a co-processor, can accept one floating-point operation at every cycle until a data dependency on an unfinished instruction occurs. In this case, the dependent instruction needs to wait until the conflicting operation terminates. The FPU will handle FP operations in a multicycle pipeline with the execution times shown in the following table:

FP operations	# of cycles
Fadd	2
Fsub	2
Fmultiply	4
Fdiv	12
Fcvt	2

The cache interface can handle one data access at each cycle and, in case of a hit, the load latency is one cycle (i.e., delayed load with one delay slot). In the case of a write hit, an extra cycle is required to modify the data block in the cache. The refilling of a prefetched line will be delayed when it competes with real data accesses for the cache.

Also, real cache misses could conflict with prefetch or outstanding write requests in the cache interface. As before we will assume, conservatively, that a fetch in progress cannot be aborted. However, a real read miss will be given priority over buffered prefetch requests or writes.

All caches used in this study are direct-mapped with 32K bytes and a block size of 16 bytes, unless otherwise specified. These caches use a write-back, write-allocate policy. The prefetching cache we studied is based on the implementation of our lookahead scheme as described in Chapter 3. In both the baseline architecture and the prefetching cache, read and write operations are always blocking.

When studying non-blocking loads, we assume a static scheduling of the pipeline. A status bit is associated with each register. On a cache miss, the target register status bit is set and the outstanding information is recorded. When the miss is resolved, the register status bit is cleared. If an instruction requires a register that is to be read or written to, the status bit is checked for the availability of that register. An instruction needing the value from a register with its status bit set will cause the processor to stall until the value is returned from memory. If a cache miss occurs when one (or more) request is in progress, the cache controller will check to ascertain that the same block is not requested twice.

When studying non-blocking writes, we assume a full-associative write buffer with 8 entries. Each entry consists of a data block and associated ‘‘valid’’ bitmap. A write miss will allocate an entry in the write buffer, update the word in the entry, set the corresponding valid bit, and then initiate a data access to memory whenever the memory interface is available. Subsequent accesses check the write buffer. A read miss finding a matched valid word in the write buffer is treated as a cache hit. Since the buffer is fully-associative, a subsequent write on a matched entry in the write buffer can be merged by writing the data in the buffer and setting the corresponding valid bit. When the block is returned from memory, those words with valid bits set in the buffer entry replace the corresponding ones returned from memory before the entire block is written into the cache.

As we have shown in Chapter 4, we consider three memory models: *Non-overlapped*, *Overlapped*, and *Pipelined*, with each model showing an increasing possibility of concurrency of access to the next level of the memory hierarchy.

We experimented with various architectural choices summarized in Table 6.1. Since the above techniques for reducing the data access penalty are not mutually exclusive, each choice of architecture is based on the combination of components described earlier.

This allows us to study the effect and contributed performance gain of various techniques, including prefetching caches (PREFETCH), write buffer (WB), prefetching caches coupled with write buffer (PREFETCH/WB), non-blocking caches (NBC), and whether allowing bypassing of writes by reads or not.

Table 6.1: Architectural Choices

	Cache	Prefetch	Non-blocking		Ordering	
			write	read	no bypass	bypass
BASELINE	X					
PREFETCH	X	X				
WB	X		X		X	
PREFETCH/WB	X	X	X		X	
NBC	X		X	X	X	
WB	X		X			X
PREFETCH/WB	X	X	X			X
NBC	X		X	X		X
HYBRID	X	X	X	X		X

6.3.2 Simulation Method

The simulator we used is similar to what has been described in Chapter 4. We evaluated our proposed architectures using cycle-by-cycle trace generation combined with on-the-fly simulation. To simulate the interlock mechanism for non-blocking reads, the simulator needs to read the object code of the benchmark program and decodes instructions so that it is aware of which registers are involved in each instruction as well as boundary information on basic blocks. This knowledge is vital to simulate the detail behavior of non-blocking operations.

Then the simulator runs the *pixified* benchmark programs, which will generate address traces at the same time. The simulator reads the trace through a *pipe* facility and feeds the trace records to each simulation object. The experiment results are collected at the clock cycle level from the individual configurations. This on-the-fly trace simulation methodology provides the flexibility of simulating a rescheduled code.

Once again, we use the SPEC Benchmarks, which are compiled by the MIPS C compiler and the MIPS F77 compiler, both with default options. Table 6.2 shows some reference characteristics for the applications. The data is collected based on the simulation of our 32K-byte baseline cache. The number of writes is smaller than the number of reads and the proportion of write misses is considerably smaller than the proportion of read misses.

Table 6.2: Statistics of benchmarks (for first 100 million instructions on 32K baseline cache)

	ratio over total instructions			miss	proportion of cache miss (%)	
Name	data refs	reads	writes	ratio	read miss	write miss
Matrix	0.461	0.307	0.154	0.087	99.1	0.9
Espresso	0.182	0.167	0.015	0.184	99.5	0.5
Tomcatv	0.418	0.326	0.092	0.063	82.4	17.6
Spice	0.258	0.209	0.049	0.116	98.7	1.3
Doduc	0.301	0.223	0.078	0.017	58.7	41.3
Nasa	0.303	0.152	0.151	0.281	84.9	15.1
Xlisp	0.467	0.315	0.151	0.014	65.5	34.5
Eqntott	0.299	0.265	0.035	0.033	79.2	20.8
Fpppp	0.567	0.449	0.118	0.004	62.2	37.8
Gcc	0.338	0.223	0.115	0.018	65.3	34.7

In the following sections, we will use MCPI as a metric to present the results of our experiments. When an average reduction of MCPI is summarized, a geometric mean is used to average the percentages of the penalty reduction for the benchmarks [Jain 91a]. In the figures, we present a breakdown of the data access penalty as follows: the bottom section (light grey area) of each bar represents the stalls for reads, the black section shows the write miss penalty, and the section on top of that (grey area) represents the stalls due to the memory interface being busy or waiting due to the ORL or the write buffer being full.

6.4 Simulation Results

In this section, we present a comparative analysis of the performance achieved by the various architectural choices and show the impact of memory latency. In the discussion, we will show only a set of representative results. The rest of data can be found in Appendix A.2.

6.4.1 Effect of Architectural Variations

Figure 6.1 shows the results for the benchmarks when simulated on the various architectures. In this first set of experiments, we used the *Pipelined* memory model so that we could temporarily ignore bandwidth limitations. Thus, there is no busy time penalty. We examine the data of Figure 6.1 according to three groups of architectures: (i) processors always stalling on a miss (blocking caches: BASE and PREFETCH), (ii) architectures with non-blocking writes and no bypass, and (iii) architectures with non-blocking writes and bypass of writes by reads. In the last two categories, we consider a baseline cache with non-blocking writes (WB), a prefetching cache with non-blocking writes (PREFETCH/WB), and a non-blocking cache (NBC) (cf. Table 6.1).

A comparison between the baseline and the prefetching cache (the first two bars in the figures) was already done in Chapter 4. Since our model is slightly different, we repeated the experiment. The results are consistent with those of Chapter 4 and show a moderate to very significant reduction in the penalty for data access when the prefetching facility is added to the baseline cache. As can be seen in Figure 6.1 (only six of ten benchmarks are shown), the access penalty is reduced by 96% for Matrix, 96% for Espresso, 41% for Xlisp, 19% for Spice, 12% for Doduc, and 36% for Nasa (66% for Tomcatv, 4% for Fpppp and Gcc, and 19% for Eqntott, see Appendix A.2). The geometric mean of the MCPI reduction from these ten applications is 23%. As we have already demonstrated, the prefetching scheme can achieve reasonable gains at the cost of the RPT and the additional logic. The effectiveness of the prefetching technique relies mostly on the presence of regular data access patterns. Also since the prediction occurs for both reads and writes, the prefetching has a proportion of write miss similar to that of the baseline case.

The effect of non-blocking writes on the baseline architecture is shown by the difference between the first, third (for no-bypassing), and sixth (bypassing) bars in the figures. Recall that in the baseline architecture the processor stalls on a write miss until the write completes.

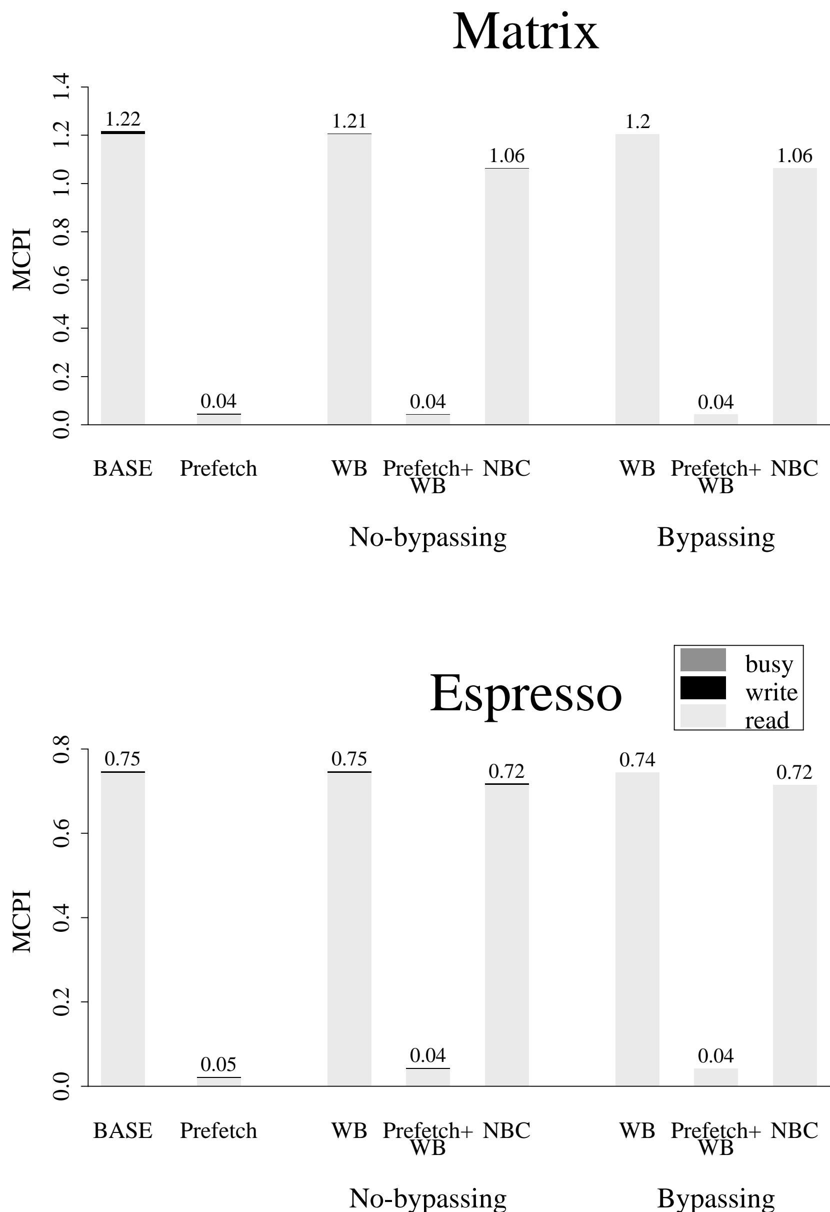


Figure 6.1a: Prefetching vs. Lockup-free for $\delta = 30$ (*Pipelined*)

In the non-blocking writes (WB) implementations, the write is put in the write buffer and the processor will stall at the next read operation in the case of no-bypass. In the case of bypass, the only case where stalls occur is on a read miss -- and the read will have priority over buffered writes -- or if the write buffer is full. As can be seen, the WB with no-bypass has almost no improvement on the write penalty (Nasa has a small gain). This is because the writes are most often followed by a read within a very small number of

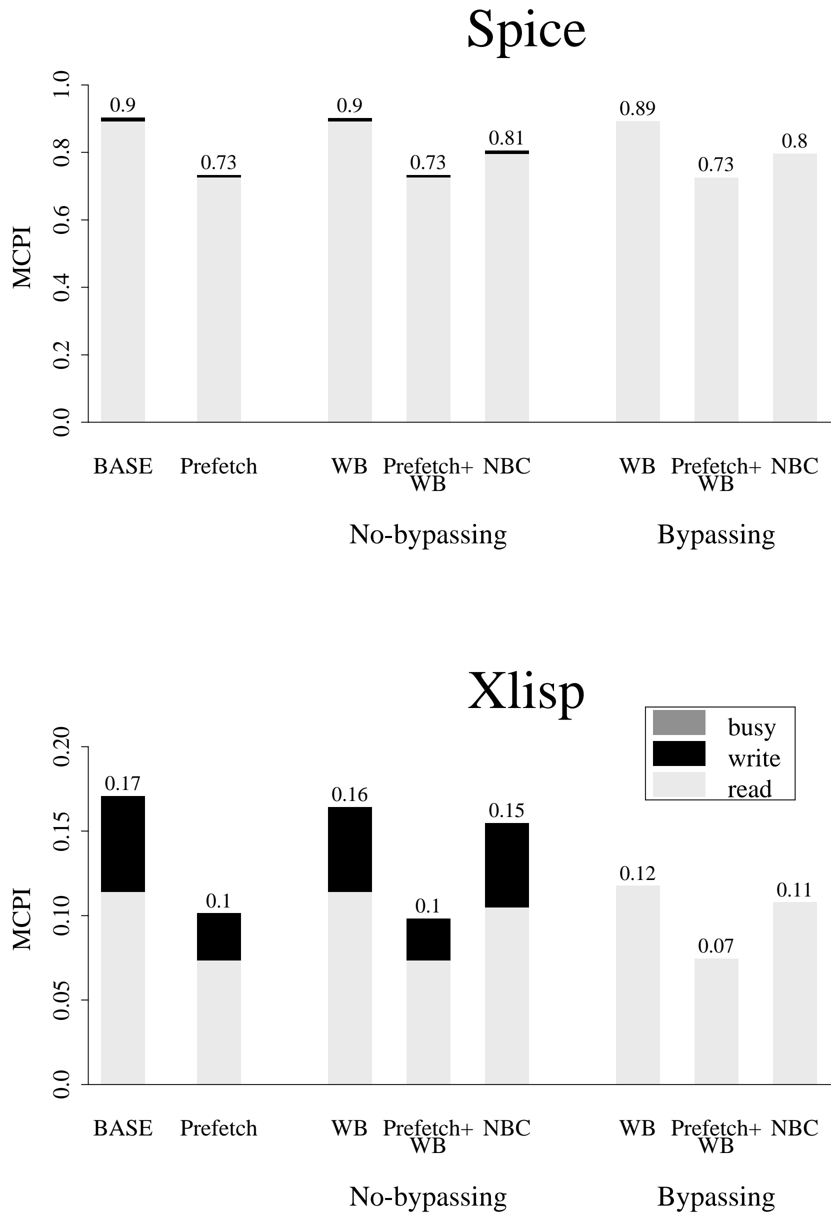


Figure 6.1b: Prefetching vs. Lockup-free for $\delta = 30$ (*Pipelined*)

instructions. When the restriction of stalling on a subsequent read is lifted, i.e., WB with bypass, the penalty due to write misses is in essence totally avoided (cf. Tomcatv, Doduc, and Nasa). However, such a reduction by the WB may not contribute to a significant overall performance improvement over the total penalty when the fraction of write miss is very small (cf. Table 6.2). A surprising but subtle result is that a write buffer may even reduce slightly the read miss penalty (e.g., 12% reduction for Nasa). This reduction is a

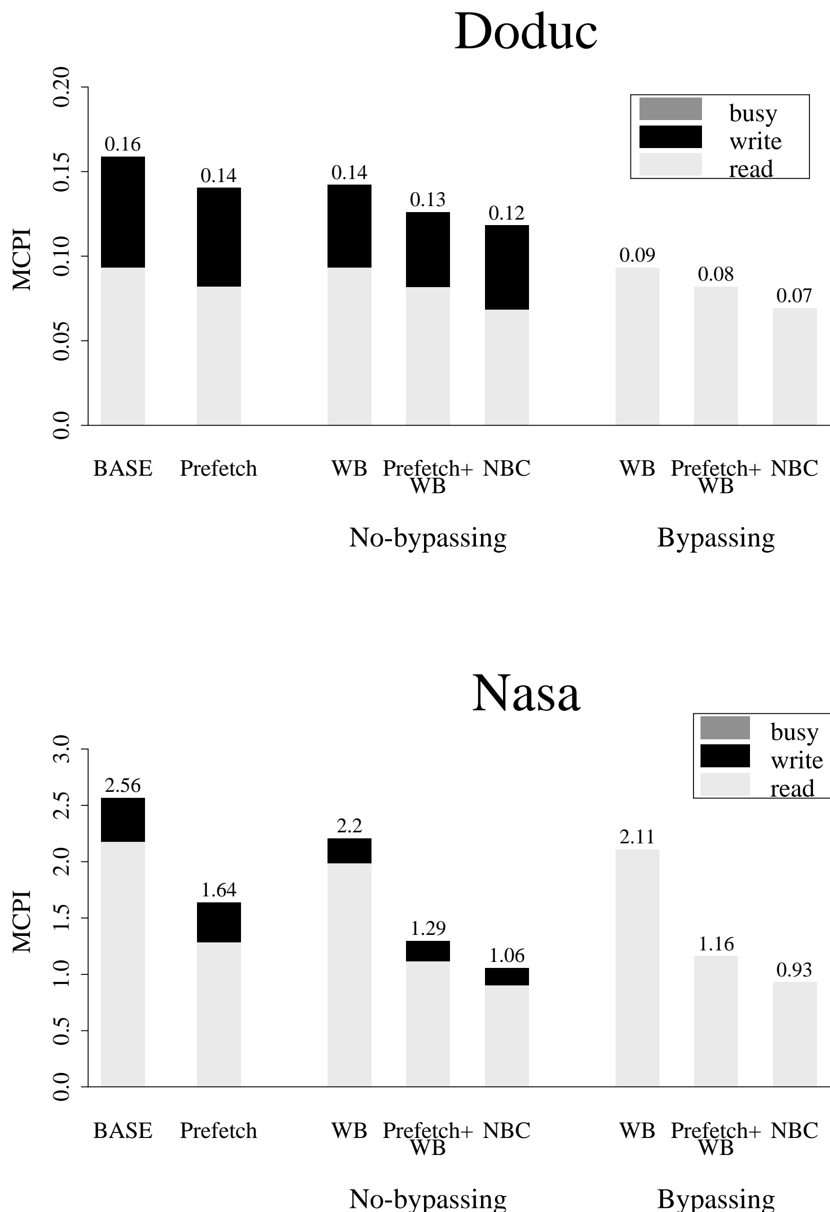


Figure 6.1c: Prefetching vs. Lockup-free for $\delta = 30$ (*Pipelined*)

consequence of forwarding data from a write to subsequent reads.[2]

We now look at the performance of WB in conjunction with prefetching (PREFETCH/WB, fourth and seventh bars) and non-blocking caches (NBC, fifth and eight bars). The purpose of showing PREFETCH/WB is to give a fair base of comparison to contrast the effect of

[2] There is also a small chance of increasing the read miss penalty since the penalty of a write miss followed by a read on the same line, but on a different word, is charged to the read miss penalty.

the read penalty reduction between the *pre*-miss overlap and the *post*-miss overlap. We focus only on WB with bypass. The results for no-bypass are qualitatively similar. The relative performances of NBC and PREFETCH/WB can be divided into three groups: i) PREFETCH/WB performs extremely well, ii) PREFETCH/WB has moderate improvement and also outperforms NBC, and iii) the performance of NBC is better than that of PREFETCH/WB.

The Matrix, Espresso and Tomcatv benchmarks belong to the first group. These programs have very good reference predictability. Although a non-blocking cache contributes to some penalty reduction (12% for Matrix, 4% for Espresso, and 31% for Tomcatv), prefetching still significantly outperforms NBC since a large portion of the data access penalty has disappeared.

Spice, Xlisp and Eqntott are the benchmarks in the second group. The effectiveness of NBC is even less than that in the first group (reductions of 10%, 8% and 12% respectively) but also is PREFETCH/WB's. As we shall see in Table 6.3 (Section 6.5), the average size of the basic blocks in these two programs is smaller than that of basic blocks in the other programs. The small size usually restricts the prediction of references for PREFETCH/WB (due to the impact of branch prediction) and also implies a limited non-blocking distance. Therefore, for Spice PREFETCH/WB has some moderate gains over the baseline WB, and NBC is only slightly better than WB. Also, WB does not help much since the fraction of write misses is fairly low.

Doduc and Nasa form the third group where NBC becomes more attractive than PREFETCH/WB. NBC is quite efficient for the two programs and the reductions are larger than those attained through PREFETCH/WB. The weak performance of PREFETCH/WB in Doduc can be related to the size (or associativity) of the reference prediction table. Doubling the size of the table, or making it of larger associativity, would remove the large number of conflicts (35% with a 256-entry direct-mapped RPT). In the case of Nasa, both schemes lead to a fair amount of performance gain, with NBC showing an advantage.

For Fpppp and Gcc, PREFETCH/WB and NBC have little performance improvement over WB. Since Fpppp has already a low miss ratio and a large loop size (as we discussed in Section 4.3.3), the improvement due to a prefetching scheme is very marginal. Gcc is a big program with many conditional branches. Since its predictability is very poor and the basic block size is small, prefetching will not be done often.

6.4.2 Effect of Large Latency

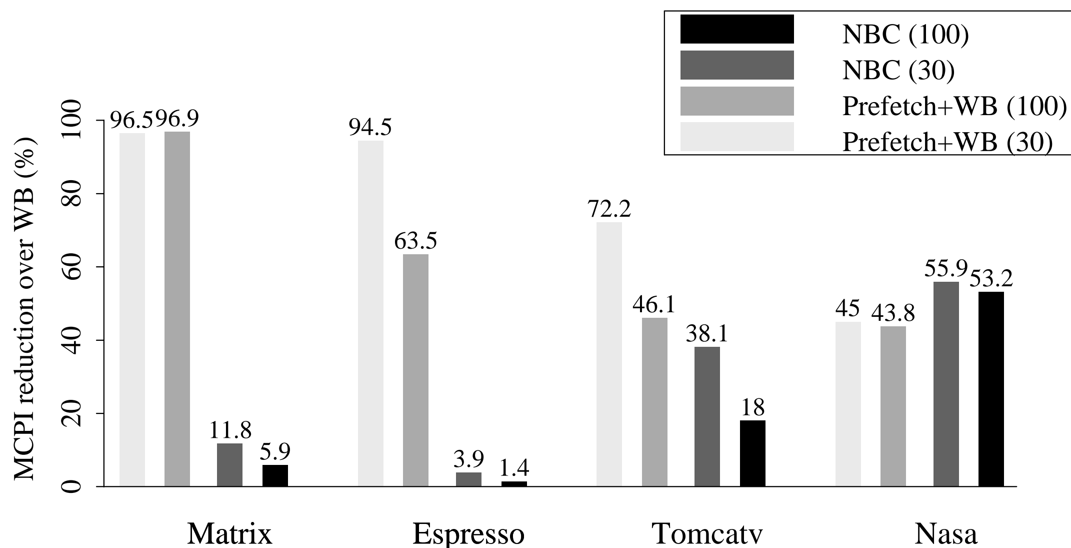


Figure 6.2a: Effect of a larger latency (for δ = 30 vs. δ = 100 *Pipelined*)

Figure 6.2 shows the experiment results of eight benchmarks[3] when the memory latency is larger. The figure plots percentages of reduction in data access penalty of PREFETCH/WB and NBC over WB with bypassing when the memory latency δ is either 30 (left bars) or 100 cycles (right bars). In general, the effectiveness of PREFETCH/WB is slightly sensitive to the (large) memory latency and is more stable than NBC's. This is because the lookahead distance of the prefetching can be *dynamically* as large as the memory latency so that data may be prefetched early enough to hide the latency. In contrast, the non-blocking distance, which is *statically* determined by the programs, becomes relatively small when the latency increases. Thus NBC's relative effectiveness is reduced significantly (almost a factor of 6 in Doduc). Note, however, that the predictability will decrease as the latency increases mostly because branch prediction becomes less reliable. The program Espresso, where the average size of basic blocks is 5.6 instructions, has significant performance degradation when there is an increase in the memory latency. As can be seen, the PREFETCH/WB's effectiveness is cut by one-third when δ increases from 30 to 100.

[3] Fpppp and Gcc are not shown because comparisons of the marginal improvement could be misleading.

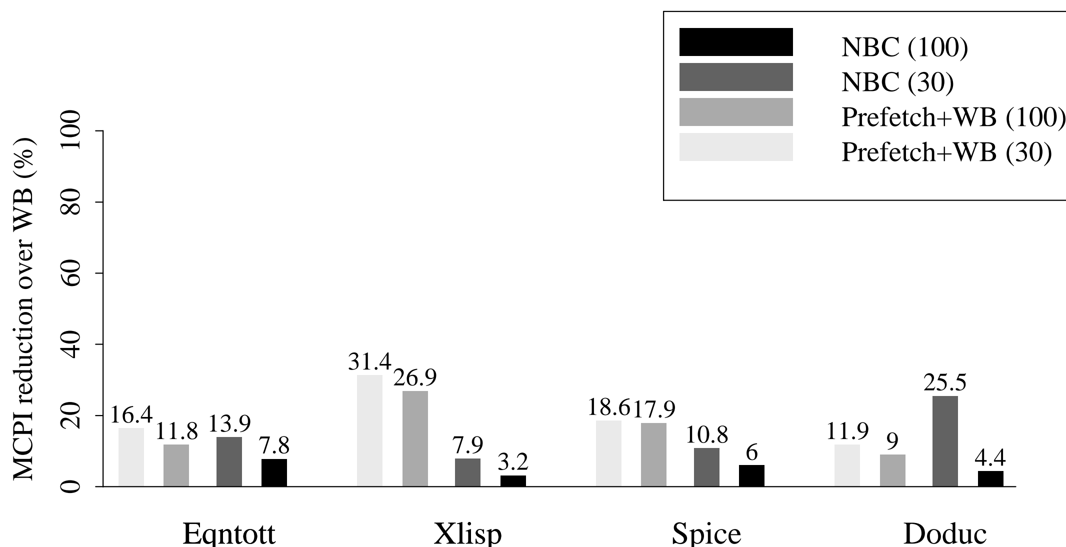


Figure 6.2b: Effect of a larger latency (for $\delta = 30$ vs. $\delta = 100$ *Pipelined*)

6.5 Compiler Assistance for Non-blocking Loads

In this section, we consider optimizations in the code generation phase for non-blocking loads. We examine two kinds of optimization: instruction scheduling for exploiting a possibly large non-blocking distance within a basic block and register renaming for removing false dependencies before the instruction scheduling is applied.

6.5.1 Instruction Scheduling and Register Renaming

With the advent of RISC architectures, compiler optimization techniques have become more important so that CPU performance can be increased. Instruction scheduling is a compiler optimization phase which schedules as many operations as possible in parallel on separate functional units. Several instruction scheduling techniques based on the architecture of a specific target machine have been proposed in the literature [Krishnamurthy 90]. Those traditional schedulers focus on instruction scheduling subject to machine resource constraints. More recently, Kerns and Eggers [92] have proposed ‘‘balanced scheduling,’’ where instructions are scheduled based on an estimate of the amount of instruction-level parallelism in the code. Their goal is to tolerate a wide range of variance in operation latency. The algorithm would be useful in scheduling code when the latency is unknown at compile time.

The purpose of register renaming, similar to that of dynamic instruction scheduling, is to remove write-after-read (WAR) and write-after-write (WAW) dependencies, thus allowing greater freedom in moving instructions around. Register renaming at compile time has been used in conjunction with software pipelining [Jain 91b]. The advantage of register renaming at compile time over dynamically renaming at run time is that the compiler can take more advantage of increased instruction parallelism (as a result of renaming) by distributing the parallelism more effectively in the code. Obviously, software renaming requires no hardware complexity.

6.5.2 Algorithm for Non-blocking Loads

The instruction scheduling that we study here, based on the scheme given by Gibbons and Muchnick [86], is performed after register allocation. The goal of the scheduling algorithm is to create as much distance as possible between a load and the first instruction dependent on that load. At the same time, we want to intersperse the loads so that the lack of memory bandwidth does not become too much of a constraint. As discussed previously, we define the *non-blocking distance* as the number of instructions that can be overlapped with the memory access, i.e., instructions between a reference and the first dependent instruction.

The algorithm schedules instructions only within basic blocks. Instructions within the block are the nodes of a weighted directed acyclic graph (DAG). As shown in Figure 6.3, edges represent dependencies and are labeled with latencies. The latency of an edge between two dependent nodes is one except when the first instruction is a load. In the latter, in order to achieve as large as possible a non-blocking distance and to avoid the clustering of loads at the beginning of a schedule, we estimate the latency of a load edge as the minimum of either the size (in number of instructions) of the basic block size divided by the number of loads in the basic block, or the actual memory latency. Once the latencies of the edges have been determined, we assign weights to the nodes of the DAG, with the weight of a node being the number of child nodes plus the maximum (over its children) of the sum of the weight of a child and of the weight of the edge leading to the child. The purpose of calculating the weight is to implement the heuristic of first picking the instruction with the greatest number of successors along the longest path from the node to a leaf node.

After the weighted DAG is built, we apply a list scheduling algorithm (shown in Figure 6.4) to derive the final schedule. The scheduling algorithm is a variation on list scheduling.

1. Build DAG $G(V,E)$ for a basic block:

   Each instruction is a vertex $v_i \in V$; an edge $e(v_i, v_j) \in E$ if v_j depends on v_i. $l(v_i, v_j)$ is the estimated latency between nodes v_i and v_j:

$$
l(v_i, v_j) = \begin{cases}
 & v_i & v_j & e(v_i, v_j) \\
\left\lfloor \frac{\text{basic block size}}{\text{\# of loads}} \right\rfloor & \textit{load} & \textit{any other}^a & \text{true dependency} \\
1 & \textit{any other} & \textit{any} & \text{true dependency} \\
1 & \textit{any} & \textit{any} & \text{false dependency} \\
1 & \textit{leaf} & \textit{branch} & \text{control dependency}
\end{cases}
$$

   [a] Any instruction node other than load

2. Define $weight(v_i)$:

$$
weight(v_i) = \begin{cases}
0 & \text{if } v_i \text{ is a leaf node} \\
n - 1 + \underset{1 \le k \le n}{MAX} \{ l(v_i, v_{j_k}) + weight(v_{j_k}) \} & \\
\quad \text{where } v_i \text{ has } n \text{ child nodes } v_{j_1}, \ldots, v_{j_n} &
\end{cases}
$$

Figure 6.3: Building the DAG for a basic block

Several sets of nodes are maintained: S_{ready} (a set of vertices that have all their predecessors already scheduled) and $S_{slot[i]}$ where i varies from 1 to the largest estimated latency. Initially S_{ready} contains the nodes which are ready to schedule, including the first instruction and other independent instructions. The scheduler always picks up in the set S_{ready} the node which has the largest weight and assigns it as the next instruction. After a node v_i from S_{ready} is scheduled, if it was the only unscheduled predecessor of its child node v_j, then v_j is included in the set $S_{slot[l(v_i,v_j)]}$. This allows at least $l(v_i, v_j)$ instructions to be interspersed between them. When an instruction is scheduled, all $S_{slot[i]}$ sets are shifted ''left'' by one slot with $S_{slot[1]}$ joining S_{ready}, because the latency of every node in $S_{slot[i]}$ (to be scheduled) has been decreased by one. When there is no instruction available in S_{ready}, we do not insert a NOP, but simply keep moving $S_{slot[i]}$ until S_{ready} is not empty. In this case, the processor

procedure reorder (G)

initialize $S_{slot[i]}$ with empty pointer

$S_{ready} = \{v_i | v_i$ has no parent node in DAG$\}$

$new_order = 1$

while $new_order \leq length\ of\ BB$

while S_{ready} is empty

$S_{ready} \Leftarrow S_{slot[1]} \Leftarrow \ldots \Leftarrow S_{slot[n]}$

choose a node v_i in S_{ready}, where $weight(v_i)$ is largest.

$order(v_i) = new_order^{++}$

for each child v_j of v_i (with an edge latency $l = l(vi, vj)$)

if v_j has no other unscheduled parent then

$S_{slot[l]} = S_{slot[l]} \bigcup \{v_j\}$

$S_{ready} = S_{ready} \bigcup S_{slot[1]}$

$S_{slot[1]} \Leftarrow \ldots \Leftarrow S_{slot[n]}$

end

end

Figure 6.4: Instruction Scheduler on the DAG

is expected to stall on the interlock, since there is no independent instruction available.

Figure 6.5 gives an example of a basic block code of 11 instructions including 3 loads and illustrates the corresponding code DAG. Nodes labeled **Lx** represent load instructions and nodes labeled **Ii** represent other instructions. After dependencies between nodes are determined, we first build a DAG for the code. We estimate the average latency of a load instruction as $\left\lfloor \frac{11}{3} \right\rfloor = 3$ and then compute the weight of each node on a bottom-up basis as shown in the right part of Figure 6.5. Referring to the new code schedule shown in Figure 6.6 and Figure 6.7, we demonstrate how the instructions are scheduled by the algorithm. At the very beginning, S_{ready} contains three independent instructions **La**, **Lb**, and **I1**. The instruction scheduler picks up the instruction **Lb**, since it has the largest weight. At the

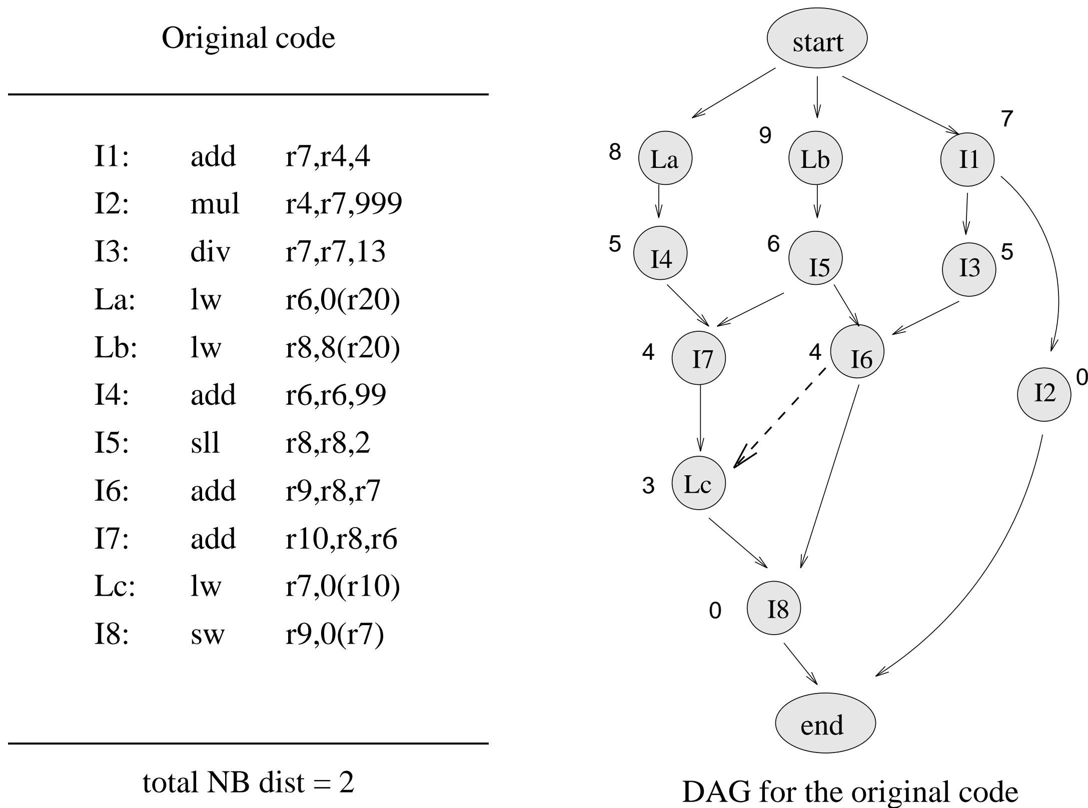


Figure 6.5: DAG of an example

Step	Scheduled	S_{ready}	$S_{slot[1]}$	$S_{slot[2]}$	$S_{slot[3]}$
0		**La**,**Lb**,**I1**			
1	**Lb**	**La**,**I1**			**I5**
2	**La**	**I1**		**I5**	**I4**
3	**I1**	**I2**,**I3**	**I5**	**I4**	
4	**I3**	**I2**,**I5**	**I4**		
5	**I5**	**I2**,**I4**,**I6**			
6	**I4**	**I2**,**I6**,**I7**			
....					

Figure 6.6: Scheduling the example

same time, it adds the instruction **I5** to the set $S_{slot[3]}$ because **I5** has no other predecessor node. The instructions in $S_{slot[i]}$ will be gradually shifted ‘‘left’’ once an instruction is

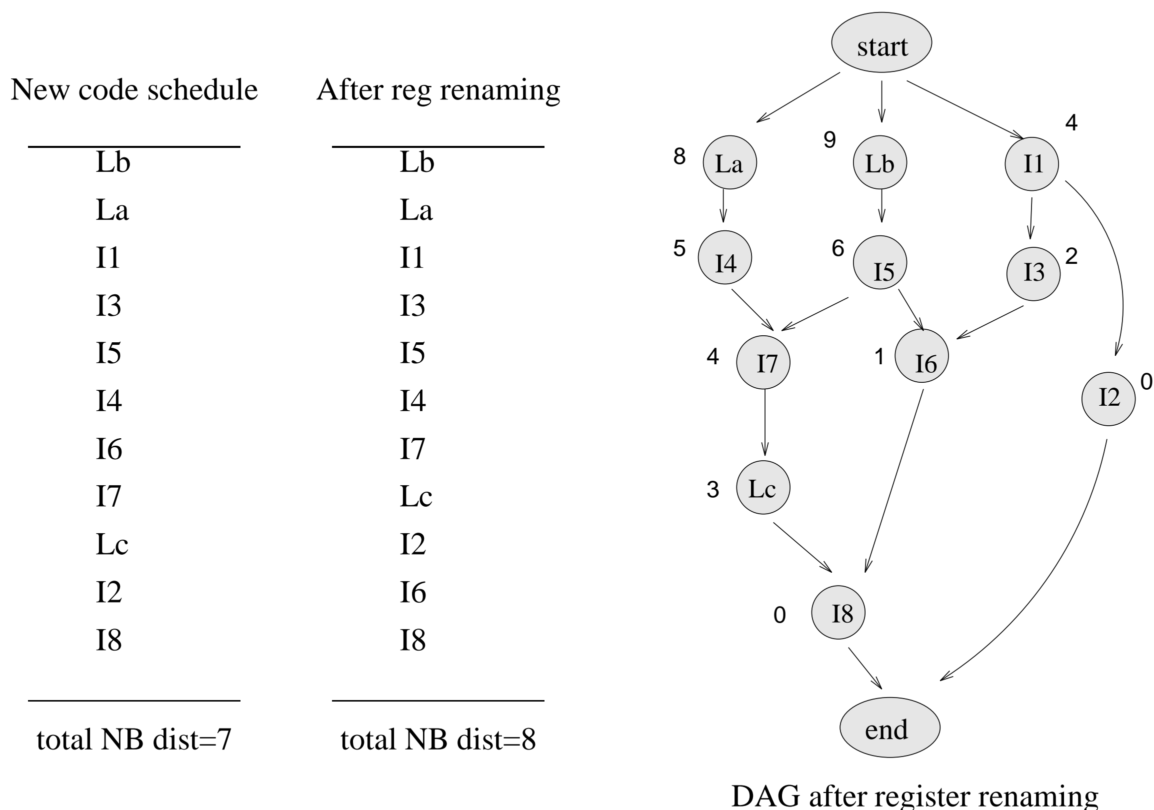


Figure 6.7: Instruction scheduling and register renaming

scheduled and eventually be included in S_{ready}. Consequently, the algorithm continues to schedule the remaining instructions **La** and **I1** in S_{ready}. Scheduling **La** will add **I4** to $S_{slot[3]}$. Scheduling **I1** will add its dependent instructions **I2** and **I3** to $S_{slot[1]}$ and then **I2** and **I3** will be moved to S_{ready}. Hence, **I3**, the one with the largest weight, is chosen after the first three instructions are scheduled. After that, the instruction **I5** is available in S_{ready}, since there are already three instructions filling the latency slots following the previous load **Lb**. At this point, S_{ready} contains **I2** and **I5**. However, **I5**, instead of **I2**, is chosen because **I5** has a larger weight. Although **I2** has been in S_{ready} for a while, it will not picked up until the last load **Lc** is scheduled, since its wweight is 0. This illustrates our strategy that we try to distribute the independent instructions among loads by estimating an equal load latency. In the same way, the algorithm schedules the rest of the instructions and obtains the new schedule as given in the left part of Figure 6.7. Overall, the new code has a total non-blocking distance of 7, compared to a non-blocking distance of 2 in the original code.

The way to estimate equal latencies of loads within a basic block could be improved

through program analysis by a compiler. The software prefetching techniques that we have discussed previously are able to identify accesses that are likely to have misses. For instance, a load of an array element with a large stride is likely to be a cache miss while accesses to the stack area will most often result in cache hits. An intelligent compiler could take this into account when assigning edge latencies. Balanced scheduling [Kerns & Eggers 92] can also be incorporated in the compiler for distributing instruction parallelism only to the accesses which have been determined as misses.

Before instruction scheduling, we may perform software register renaming on the code so that the instruction scheduler would have more freedom to move instructions. The algorithm we use first identifies the live ranges (from a new definition to the last use before the next definition) for each register to be renamed (local registers). Then, for the live ranges entirely falling within the basic block, except the last live range, the destination register used in a load operation is replaced by a new register. This renaming is carried on for those instructions using the same register within the live range. Since the scheduling is performed after register allocation, we assume that there is a set of ‘‘spare’’ registers available. This is in order to keep the algorithm simple. Otherwise, we would have to identify temporary registers and unused registers in the basic block and our algorithm would become global rather than being restricted to the basic block level. After the register renaming process, we apply the instruction scheduling described above on the new DAG from which some false dependence edges have been removed.

Referring again to Figure 6.5 and Figure 6.7, we note that the relationship between **I6** and **Lc** is due to a write-after-read (WAR) dependency. We may simply remove this false dependency by replacing the usage of ‘‘r7’’ in the instructions **Lc** and **I8** with another register available to the scheduler. Removing such a false dependence makes it possible to move **I6** to fill the latency slots of **Lc**. Such a renaming will result in different weights in a new DAG from those in the original DAG. Hence we schedule the code based on the new DAG as illustrated in Figure 6.7 and the average non-blocking distance has been increased by one.

A potential criticism of our study is that we adversely increase the register pressure in a basic block. A compensating factor is that WB may help, by buffering writes, the extra spilling store/load instructions that could be generated. Our point is that we give priority for register use to a load operation with a large latency, even at the cost of adding spill code. Although the results of our register renaming procedure are optimistic since we do

not limit the number of registers, the approach is still feasible if the compiler identifies the unused registers or performs a priority-based register allocation [Chow & Hennessy 90] by taking into account the cost of the data access penalty.

6.5.3 Effect of Instruction Scheduling

Table 6.3 shows the effect of the reordering of data accesses. All the data is in ‘‘weighted average’’ form where the weight is the execution frequency of the individual basic blocks. The data in the columns ‘‘non-blocking distance’’ are the average numbers of instructions between a read access and the subsequent instruction dependent on the value being read, regardless of whether the read is a cache miss or not. In general, the non-blocking distance based on the original code is fairly small.[4] A comparison between the second and third columns of the table shows that the instruction scheduling algorithm is very effective for increasing the non-blocking distance for Matrix and Tomcatv. This indicates that in these two benchmarks there are several data loading phases followed by computations on that data. The scheduling algorithm reorganizes the instructions to allow more overlap between the independent loading phases. For the other benchmarks[5] that do not have this characteristic, the distance is moderately increased.

When register renaming is added to instruction scheduling, the compiler has more flexibility to optimize the code reordering. As shown in Table 6.3, a significant increase in non-blocking distance is achieved in Doduc and Nasa with the use of a *small* number of extra registers (less than one per block on the average). On the other hand Matrix and Tomcatv need more registers with not much improvement for the latter. Note that the number of registers required for renaming is overestimated, since two live ranges, which are originally far apart, are less likely to be live at the same time after renaming because of other dependence chaining between them. This was not taken into account in our algorithm but could be checked out by the compiler.

Figure 6.8 and Figure 6.9 show the relative performance of the optimizations for the NBC architecture. The data access penalty for the two code optimization algorithms is normalized to the penalty of the original code. The results of the *Non-overlapped* model

---

[4] Note that the original MIPS compiler should strive to yield a non-blocking distance greater than one since the R3000 has a delayed load of one cycle.

[5] In the following discussion, we omit the results of the other benchmarks because the effects of Xlisp (4.97) and Eqntott (3.84) with small average block sizes are similar to those of Espresso and Spice.

Table 6.3: Average of basic block size, non-blocking distance, and extra registers needed (weighted by frequency of execution)

Name	Basic block size	Original code	scheduling only	Renaming + scheduling	
		non-blocking distance	non-blocking distance	non-blocking distance	extra regs needed
Matrix	39.15	2.23	8.33	15.14	8.65
Tomcatv	54.47	3.38	10.03	11.91	12.13
Spice	7.79	2.46	3.27	3.83	0.05
Espresso	5.63	1.47	2.25	2.93	0.11
Doduc	10.01	3.44	4.88	8.14	0.77
Nasa	27.49	2.09	4.89	8.17	0.28

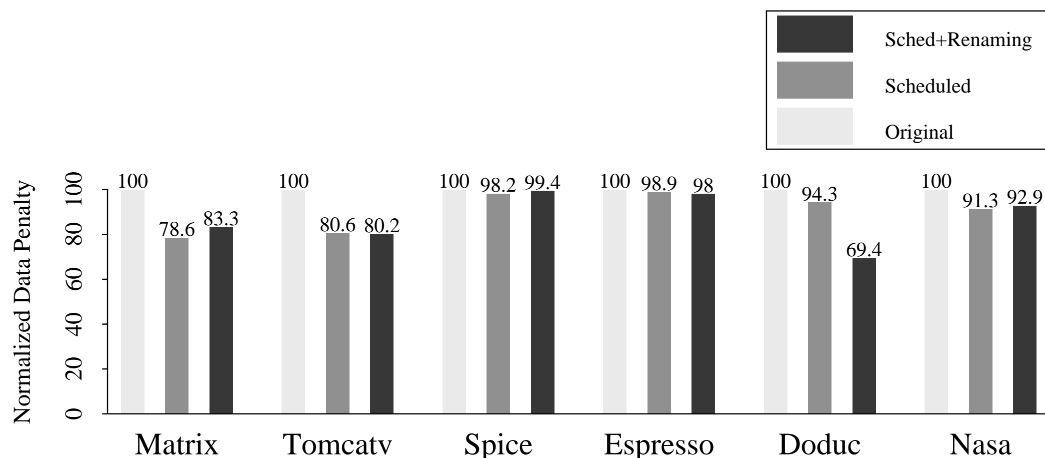


Figure 6.8: Effect of instruction scheduling on NBC for δ = 30 (*Non-overlapped*)

in Figure 6.8 show that only those programs with low miss ratios (Matrix, Tomcatv, and Doduc) can benefit from instruction scheduling. This is not surprising because the *Non-overlapped* model does not provide sufficient bandwidth to fully exploit the advantage of the overlap. Also, register renaming does not contribute much performance gain to the NBC in the *Non-overlapped* model and might even degrade the performance slightly (cf. Matrix). Instruction scheduling tends to increase the clustering of read accesses.

Scheduling instructions which have no false dependencies by applying register renaming causes the read accesses to be more clustered.

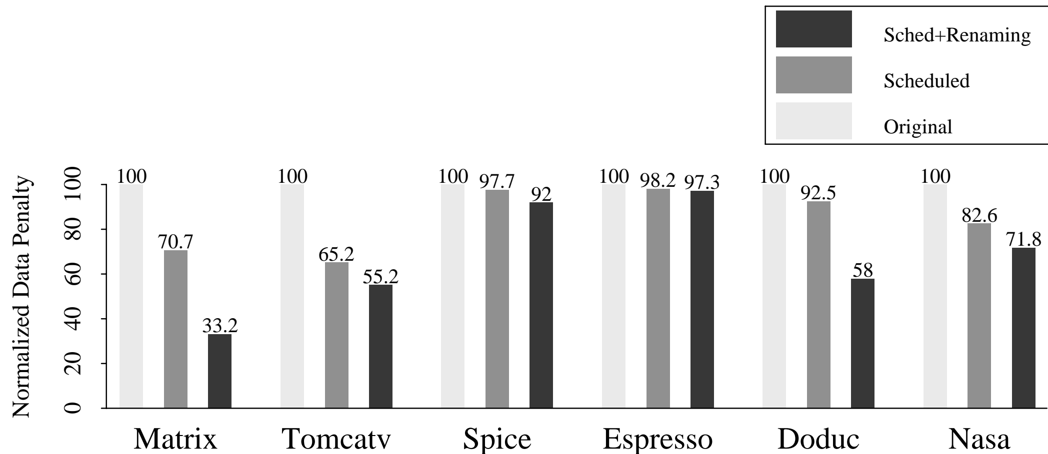


Figure 6.9: Effect of instruction scheduling on NBC for δ = 30 (*Pipelined*)

When the *Pipelined* model is assumed (shown in Figure 6.9), the clustering of reads becomes an advantage that can be exploited by the NBC. In all cases except Espresso and Spice, the experiments show significant gains from instruction scheduling (improvement varies from 2% for Espresso to 35% for Tomcatv). Even better results are achieved when register renaming is applied before instruction scheduling (improvement varies from 3% for Espresso to 67% for Matrix but recall that the results are optimistic). The geometric mean of penalty reduction by instruction scheduling for those benchmarks is 9.5% over the original code and when register renaming is added, this geometric mean is up to 24% over the original code. This illustrates that instruction scheduling and register renaming provide an inexpensive solution to help hide the large memory latency for non-blocking loads in processors whose design is based on static instruction scheduling. However, techniques of instruction scheduling across basic block boundaries should be further investigated.

6.6 A Hybrid Design

Since prefetching and non-blocking caches are not mutually exclusive, a further enhancement would be to combine the two schemes: a prefetch hint is provided prior to the load instruction and the binding of a loaded value with a register is delayed until the value is actually used. This hybrid design is attractive since the combined scheme can tolerate the

drawbacks of poor predictability and of short non-blocking distance. Moreover, the cost is not dramatically increased from either the prefetching cache or the non-blocking cache. For the prefetching cache, in addition to an RPT and associated logic, the register interlock mechanism is added in the processor. At the same time, an ORL (or MSHRs), which is searched associatively, must be extended to record the information of function units which is waiting for a miss data item.

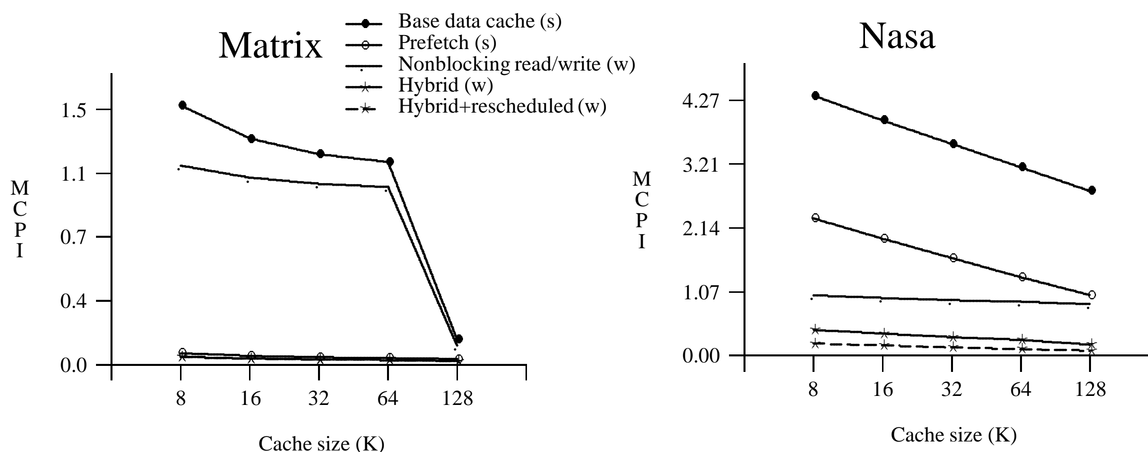


Figure 6.10: Hybrid design on varying cache size $\delta = 30$ (*Pipelined*)

In figure 6.10, we present the results of the simulation of such a hybrid scheme with and without instruction rescheduling when compared to the baseline cache, a prefetching only scheme, and a non-blocking cache with bypass. We vary the cache size from 8K bytes to 128K bytes and show only two benchmarks: Matrix where prefetching is performing much better than NBC, and Nasa where the converse is true. In Matrix, the prefetch scheme already had reduced the data access penalty to only a few hundredths of a cycle. The hybrid design has now nearly ideal performance. The performance of the hybrid scheme has more dramatic effects in Nasa. The data access penalty that is far from being negligible if either prefetching or NBC is applied alone becomes small even at the smallest cache size. Code optimization helps the hybrid combination further so that only 4% of the initial penalty incurred with a baseline cache remains. These results indicate that the length of the overlap from *pre*-miss to *post*-miss can be large enough to cover the memory latency to a great extent. The additional cost paid for the hybrid design is justified by the significant performance improvement.

6.7 Summary

In this chapter, we have evaluated the effectiveness of write buffers, and non-blocking caches in exploiting the overlap of data accesses with computation. These evaluations are made using the SPEC benchmarks and simulations are performed on a cycle by cycle basis. We confirm previous studies showing that buffering writes while allowing bypass of reads can eliminate entirely the write miss penalty. The results show that when the non-blocking write with bypass is used as a basis, the average percentage of read penalty reduction by prefetching caches was 37%, whereas the average percentage of read penalty reduction by non-blocking caches was 16%. Also, the effectiveness of prefetching caches is less sensitive to a large memory latency than that of non-blocking caches.

Code optimization via instruction scheduling can reduce prominently the data access penalty in the case of non-blocking caches. We have presented a local (at the basic block level) algorithm that, on the average, reduced the penalty by 9.5%. With the addition of an (optimistic) renaming scheme, this reduction went up to 24%. These results illustrate that a non-blocking cache assisted by a good code optimizer and associated with a statically scheduled processor can achieve remarkable gains at a cost of less complicated hardware complexity than what is needed for a dynamically scheduled processor.

Finally, we have proposed a hybrid design incorporating features from both prefetching and non-blocking caches. We have showed that the combination of pre-miss overlap and post-miss overlap present in such a scheme can be very effective in hiding large memory latencies.

Chapter 7

Conclusion

7.1 Summary of Results

In this dissertation, we have addressed issues concerning the design and analysis of caching techniques for tolerating memory latency in high-performance processors. Under the usual caching mechanism, the processor will stall on a cache miss. In order to make further progress towards the reduction in memory latency, memory accesses must proceed in parallel with processor execution. We have examined two techniques: data prefetching and non-blocking caches. The goal of the prefetching is to reduce the processor stall time by bringing data into the cache just before its use. A non-blocking cache allows execution to proceed concurrently with one (or more) cache misses until an instruction that actually needs the missed value is reached. Prefetching exploits the overlap of computations *prior to* an actual cache miss, whereas non-blocking takes the advantage of the *post*-miss overlap

We have proposed a hardware-based data prefetching scheme. The basic idea of the scheme is to keep track of data access patterns in a reference prediction table (RPT) organized as an instruction cache. Each entry in the RPT is associated with a finite state mechanism to prevent unnecessary prefetches. We have investigated three variations of the design of the RPT and associated logic. They differ mostly on the timing of the prefetching. In the simplest way, the generic scheme generates prefetches one iteration ahead of actual use. The lookahead scheme takes advantage of a look-ahead program counter that ideally stays one memory latency time, i.e., potentially several loop iterations, ahead of the real program counter. The pseudo program counter is also used as the control mechanism to generate the prefetches. Finally the correlated scheme uses a more sophisticated design to detect patterns across loop levels.

We have evaluated the three prefetching schemes by comparing them with a pure cache design at various cache sizes. These designs are evaluated by simulating the ten SPEC benchmarks cycle-by-cycle in a uniprocessor environment. The results show that the prefetching schemes are generally effective in reducing the data access penalty. The

cost of the hardware unit is not prohibitive; a moderately sized RPT (roughly equivalent to a 4K cache) is generally sufficient to capture the access patterns for the most frequently executed instructions. We observed that the lookahead scheme has a moderate win over the generic scheme, while the performance difference between the lookahead and correlated schemes is fairly small. The benefits of prefetching are greater when the hardware assist augments small on-chip caches.

We further examined the substantive performance gains that can be achieved with hardware-controlled and software-directed prefetching. Our qualitative comparisons indicate that on the domain of linear array references, both hardware and software schemes are able to generate prefetches for cache misses. However, the software scheme may have a code expansion problem, while the hardware scheme has less information on whether prefetching data will be used or not. When dealing with complex data access patterns, the software approach may have more compile-time information to perform sophisticated prefetching, whereas the hardware scheme has the advantage of manipulating dynamic information (such as conflict misses or input data dependence).

Then we quantitatively evaluated these two schemes through direct-execution simulation in a shared-memory multiprocessor environment. Our experiments confirm the previous observations. We also observed that the cache interference incurred by prefetching is almost negligible given sufficient memory bandwidth. The software approach has less of a negative effect on network traffic and conflicts with the working set than the hardware approach. However, the overhead due to the extra prefetch instructions and associated computations is substantial in the software-directed approach. In that scheme, the performance gain of reducing the read penalty will be offset by the increase in instruction execution time. Consequently we proposed and examined an alternative for combining the software and hardware solutions. The main idea is that through software prefetches we bring into the secondary cache data objects of the size determined by the user's semantics and let the hardware supporting unit take care of cache line size accesses in the loops and fetch the corresponding data into the primary cache. The new approach can combine the advantages of both hardware and software approaches and at the same time avoid most of their negative effects. Our experimental results show that the new solution is very attractive in reducing the data access penalty without incurring much overhead.

Finally, we discussed and evaluated the effectiveness of non-blocking caches and compared it with that of the lookahead prefetching hardware scheme. The results show

that when non-blocking writes with bypass are used as a basis, the average percentage of the read penalty reduction by non-blocking caches is 16%, compared to a average reduction of 37% by prefetching caches. The effectiveness of prefetching caches is less sensitive to a large memory latency than that of non-blocking caches. We considered compiler-based optimizations to enhance the effectiveness of non-blocking caches. We have shown that instruction scheduling and register renaming can reduce significantly the data access penalty in the non-blocking paradigm. Then we proposed a hybrid design based on the combination of prefetching and non-blocking schemes. We showed that the combination of pre-miss overlap and post-miss overlap present in such a scheme can be very effective in hiding large memory latencies.

7.2 Future Research

In this dissertation, we mainly focused on techniques for tolerating memory latency. As we believe that caches are crucial components in high-performance systems, we would like to further pursue the following issues regarding cache memory:

- Cache memory support for multithreading

  Multithreaded architectures have been shown to be effective for tolerating long memory latencies. A fast context switch mechanism is an essential requirement in those architectures. Caches can be used in these architectures (note that not all threaded architectures use caches, e.g., the Tera machine [Alverson *et al.* 90]) but it is important to realize their vulnerability. The problem stems from the dual functions of the cache: hiding memory latency and preserving locality for all the threads that can be activated. As too many threads share the cache, capacity and conflict misses will increase. Interesting research topics include a design of register-cache organization for fast task switching and conserving locality, compiler efforts in scheduling and protecting vulnerable pending threads from switching, and the study of granularity of thread parallelism with scheduling affinity in the register-cache complex.

- Programmable cache controller

  As multiprocessor architectures are getting more complicated, the role that caches play in these architectures becomes increasingly important and sophisticated. Coherence protocol, choice of block size, and associativity are a non-exhaustive list

of design issues. Several researchers have proposed various solutions to address those issues and performance results showed that no particular approach absolutely dominates the others. A good way to take advantage of various approaches is to allow adaptive mechanisms in the controller. Our research efforts will focus on the efficient interaction between processor and controller and a well-defined framework for dynamic adjustment in the controller.

- Efficient simulation environments

  We have implemented a trace-driven simulator for uniprocessors and an execution-driven simulator for multiprocessor architectures. We are investigating an adaptive time sampling technique to improve simulation run-time. This technique is expected to reduce trace size and simulation time for long traces that are generated from the complete execution of applications. Future research work may include:

  -- extending the sampling technique to allow on-the-fly decision-making so that trace reduction can be achieved in one pass,

  -- studying sampling techniques coupled to the execution-driven environment,

  -- building a formal validation of proposed architectures, and

  -- building up a complete simulation environment and testbed for evaluating parallel computer systems.

- Cache memory systems for data-processing programs

  So far, most of the quantitative research on cache systems in the literature is based on evaluations of scientific codes or general purpose programs. Another extremely important application domain for high-performance systems, namely data-processing programs and database systems, may illustrate different program characteristics from what had been derived from scientific programs. For example, the grain size of data sharing depends on the type of transactions, and computations in one processor may require more interactions with other processors. Some preliminary measurements should be performed on contemporary database systems. Current memory design may not be a suitable environment to perform such a study. We should characterize and abstract the features of memory accesses in those programs and then investigate the applicability of current memory caching techniques on these features. Overall,

observations which are made on a different application domain may lead to different views of the memory systems and may result in new approaches in exploring efficient caching techniques for high-performance systems.

Bibliography

[Adve & Hill 90] Adve, S. and Hill, M. (1990). Weak ordering - a new definition. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 2--14.

[Agarwal *et al.* 90] Agarwal, A., Lim, B.-H., Kranz, D., and Kubiatowicz, J. (1990). APRIL: A processor architecture for multithreading. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 104--114.

[Alverson *et al.* 90] Alverson, R., Callahan, D., Cummings, D., Koblenz, D., Porterfield, B., and Smith, B. (1990). The Tera computer system. In *Proc. 1990 Intl. Conf. on Supercomputing*, pages 1--6.

[Archibald & Baer 86] Archibald, J. and Baer, J. L. (1986). Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273--298.

[Baer & Chen 91] Baer, J.-L. and Chen, T.-F. (1991). An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing '91*, pages 176--186. Also TR 91-03-07, Department of Computer Science and Engineering, University of Washington.

[Baer & Wang 89] Baer, J.-L. and Wang, W.-H. (1989). Multi-level cache hierarchies: Organizations, protocols and performance. *Journal of Parallel and Distributed Computing*, 6(3):451--476.

[Ball & Larus 93] Ball, T. and Larus, J. R. (1993). Branch prediction for free. Technical Report #1137, Computer Science Department, University of Wisconsin - Madison.

[Boothe & Ranade 92] Boothe, B. and Ranade, A. (1992). Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*, pages 214--223.

[Brantley *et al.* 85] Brantley, W. C., McAuliffe, K. P., and Weiss, J. (1985). RP3 processor-memory element. In *Proc. of the Int. Conf. on Parallel Processing*, pages 782--789.

[Bray & Flynn 91] Bray, B. K. and Flynn, M. J. (1991). Writes caches as an alternative to write buffers. Technical Report CSL-TR-91-470, Stanford University.

[Brewer *et al.* 91] Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. (1991). PROTEUS: A parallel-architecture simulator. Technical Report LCS/TR-516, MIT.

[Censier & Feautrier 78] Censier, L. and Feautrier, P. (1978). A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112--1118.

[Chen *et al.* 91] Chen, W. Y., Mahlke, S. A., Chang, P. P., and Hwu, W.-M. (1991). Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th International Symposium on Mircoarchitecture*.

[Chen *et al.* 92] Chen, W. Y., Mahlke, S. A., and Hwu, W.-M. (1992). Tolerating data access latency with register preloading. In *Proc. 1992 Intl. Conf. on Supercomputing*.

[Chow & Hennessy 90] Chow, F. C. and Hennessy, J. (1990). The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501--536.

[Davis *et al.* 91] Davis, H., Goldschmidt, S., and Hennessy, J. (1991). Multiprocessor simulation and tracing using Tango. In *Proc. of the Int. Conf. on Parallel Processing*, pages II 99 -- 107.

[DEC 92] DEC (1992). *Alpha Architecture Handbook*. Digital Press.

[Dubois *et al.* 86] Dubois, M., Scheurich, C., and Briggs, F. (1986). Memory access buffering in multiprocessors. In *Proc. of the 13th Annual Intl. Symp. on Computer Architecture*, pages 434--442.

[Dubois *et al.* 88] Dubois, M., Scheurich, C., and Briggs, F. A. (1988). Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2).

[Dubois *et al.* 91] Dubois, M., Wang, J.-C., Barroso, L., Lee, K., and Chen, Y.-S. (1991). Delayed consistency and its effects on miss rate of parallel programs. In *Proc. of Supercomputing '91*, pages 197--206.

[Fu & Patel 91] Fu, J. W. C. and Patel, J. H. (1991). Data prefetching in multiprocessor vector cache memories. In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pages 54--63.

[Fu & Patel 92] Fu, J. W. C. and Patel, J. H. (1992). Stride directed prefetching in scalar processors. In *Proc. of the 25th Int'l Symp. on Microarchitecture*, pages 102--110.

[Gharachorloo *et al.* 91a] Gharachorloo, K., Gupta, A., and Hennessy, J. (1991a). Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 245--259.

[Gharachorloo *et al.* 91b] Gharachorloo, K., Gupta, A., and Hennessy, J. (1991b). Two techniques to enchance the performance of memory consistency models. In *Proc. of the Int. Conf. on Parallel Processing*, pages I:355--I:364.

[Gharachorloo *et al.* 92] Gharachorloo, K., Gupta, A., and Hennessy, J. (1992). Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*.

[Gibbons & Muchnick 86] Gibbons, P. B. and Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. In *Proc. of SIGPLAN Symp. on Compiler Construction*.

[Gornish *et al.* 90] Gornish, E., Granston, E., and Veidenbaum, A. (1990). Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proc. 1990 Intl. Conf. on Supercomputing*, pages 354--368.

[Grunwald *et al.* 91] Grunwald, D., Nutt, G. J., Wagner, D., and Zorn, B. (1991). A parallel execution evaluation testbed. Technical report, University of Colorado.

[Hennessy & Patterson 90] Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.

[Hum & Gao 91] Hum, H. J. and Gao, G. R. (1991). Efficient support of concurrent threads in a hybrid dataflow/von neumann architecture. Technical Report 35, McGill University.

[Jain 91a] Jain, R. (1991a). *The Art of Computer System Performance Anaylsis*. John Wiley & Sons, Inc.

[Jain 91b] Jain, S. (1991b). Circular scheduling: a new technique to perform software pipelining. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 219--228.

[Jeremiassen & Eggers 92] Jeremiassen, T. E. and Eggers, S. J. (1992). Computing per-processor summary side-effect information. In *Proc. of workshop on Language and Compilers for Parallel Computing*.

[Jouppi 90] Jouppi, N. P. (1990). Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 364--373.

[Kerns & Eggers 92] Kerns, D. R. and Eggers, S. (1992). Balanced scheduling: instruction scheduling when memory latency is uncertain. Technical Report 92-11-03, Department of Computer Science, University of Washington, Seattle WA.

[Klaiber & Levy 91] Klaiber, A. C. and Levy, H. M. (1991). An architecture for software-controlled data prefetching. In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pages 43--53.

[Kowalik 85] Kowalik, J. S. (1985). *Parallel MIMD Computation: the HEP Supercomputer and its application*. MIT Press.

[Krishnamurthy 90] Krishnamurthy, S. M. (1990). A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97--106.

[Kroft 81] Kroft, D. (1981). Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th Annual Intl. Symp. on Computer Architecture*, pages 81--87.

[Kurihara *et al.* 91] Kurihara, K., Chaiken, D., and Agarwal, A. (1991). Latency tolerance through multithreading in large-scale multiprocessing. In *Proc. of Int. Symp. on Shared Memory Multiprocessing*, pages 91--101.

[Lam 88] Lam, M. S. (1988). Sotfware pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 318--328.

[Laudon *et al.* 92] Laudon, J., Gupta, A., and Horowitz, M. (1992). Architectural and implementations tradeoffs in the design of multiple-context processors. Technical Report CSL-TR-92-523, Stanford University.

[Lee & Smith 84] Lee, J. K. F. and Smith, A. J. (1984). Branch prediction strategies and branch target buffer design. *Computer*, pages 6--22.

[Lee *et al.* 87a] Lee, R. L., Yew, P.-C., and Lawrie, D. H. (1987a). Data prefetching in shared memory multiprocessors. In *Proc. of the Int. Conf. on Parallel Processing*, pages 28--31.

[Lee *et al.* 87b] Lee, R. L., Yew, P.-C., and Lawrie, D. H. (1987b). Multprocessor cache design considerations. In *Proc. of the 14th Annual Intl. Symp. on Computer Architecture*, pages 253--262.

[Motorola 90] Motorola (1990). *MC88100 RISC Microprocessor User's Manual*. Prentice Hall.

[Mowry & Gupta 91] Mowry, T. and Gupta, A. (1991). Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87--106.

[Mowry *et al.* 92] Mowry, T., Lam, M. S., and Gupta, A. (1992). Design and evaluation of a compiler algoritm for prefetching. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62--73.

[Murakami *et al.* 89] Murakami, K., Irie, N., and Tomita, S. (1989). SIMP (single instruction stream / multiple instruction pipelining): A novel high-speed single-processor architecture. In *Proc. of the 16th Annual Intl. Symp. on Computer Architecture*, pages 78--85.

[Nikhi *et al.* 91] Nikhi, R. S., Papadopoulos, G. M., and Arvind (1991). *T: A multithreaded massively parallel architecture. Technical report, MIT Computer Science.

[Oehler & Groves 90] Oehler, R. R. and Groves, R. D. (1990). IBM RISC System/6000 processor architecture. *IBM J. Res. Development*, 34(1):23--36.

[Pan *et al.* 92] Pan, S.-T., So, K., and Rahmeh, J. T. (1992). Improving the accuracy of dynamic branch prediction using branch correlation. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 76--84.

[Perleberg & Smith 89] Perleberg, C. H. and Smith, A. J. (1989). Branch target buffer design and optimization. Technical Report UCB/CSD 89/552, University of California, Berkeley.

[Porterfield 89] Porterfield, A. K. (1989). Software methods for improvement of cache performance on supercomputer applications. Technical Report COMP TR 89-93, Rice University.

[Przybylski 90] Przybylski, S. (1990). The performance impact of block sizes and fetch strategies. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 160--169.

[Rogers & Li 92] Rogers, A. and Li, K. (1992). Software support for speculative loads. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 38--50.

[Scheurich & Dubois 91] Scheurich, C. and Dubois, M. (1991). Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25--36.

[Singh *et al.* 92] Singh, J. P., Weber, W.-D., and Gupta, A. (1992). SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5--44.

[Sklenar 92] Sklenar, I. (1992). Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20(4):31--37.

[Smith 82a] Smith, A. J. (1982a). Cache memories. *ACM Computing Surveys*, 14(3):473--530.

[Smith 82b] Smith, J. E. (1982b). Decoupled access/execute computer architectures. In *Proc. of the 9th Annual Intl. Symp. on Computer Architecture*, pages 112--119.

[Smith *et al.* 90] Smith, M. D., Lam, M., and Horowitz, M. A. (1990). Boosting beyond static scheduling in a superscalar processor. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 344--254.

[Smith *et al.* 91] Smith, R., Archibald, J., and Nelson, B. (1991). A timing based simulation study of prefetching in a second level cache. Technical Report TR-A105-91.3, Brigham Young University.

[Sohi & Franklin 91] Sohi, G. S. and Franklin, M. (1991). High-bandwidth data memory systems for superscalar processor. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 53--62.

[Stenstrom *et al.* 91] Stenstrom, P., Dahlgren, F., and Lundberg, L. (1991). A lockup-free multiprocesso cache design. In *Proc. of the Int. Conf. on Parallel Processing*, pages I--246 -- I--250.

[Tullsen & Eggers 93] Tullsen, D. M. and Eggers, S. J. (1993). Limitation of cache prefetching on a bus-based multiprocessor. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*.

[Weber & Gupta 89] Weber, W.-D. and Gupta, A. (1989). Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. of the 16th Annual Intl. Symp. on Computer Architecture*, pages 273--280.

[Wolf & Lam 91] Wolf, M. E. and Lam, M. (1991). A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30--44.

[Yeh & Patt 92] Yeh, T. and Patt, Y. N. (1992). Alternative implementation of two-level adaptive branch prediction. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*, pages 124--134.

[Zucker 92] Zucker, R. N. (1992). *Relaxed Consistency and Synchronization in Parallel Processors*. PhD thesis, Department of Computer Science and Engineering, University of Washington.

[Zucker & Baer 92] Zucker, R. N. and Baer, J.-L. (1992). A performance study of memory consistency models. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*, pages 2--12.

Appendix A

Supplemental Data

A.1 Evaluation of Data Prefetching

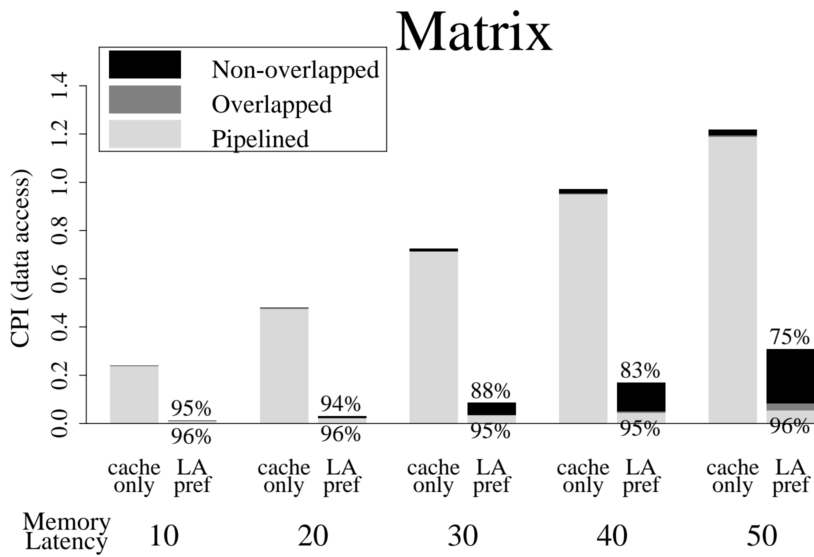


Figure A.1a: Effect of memory models and latencies (continued from Figure 4.4)

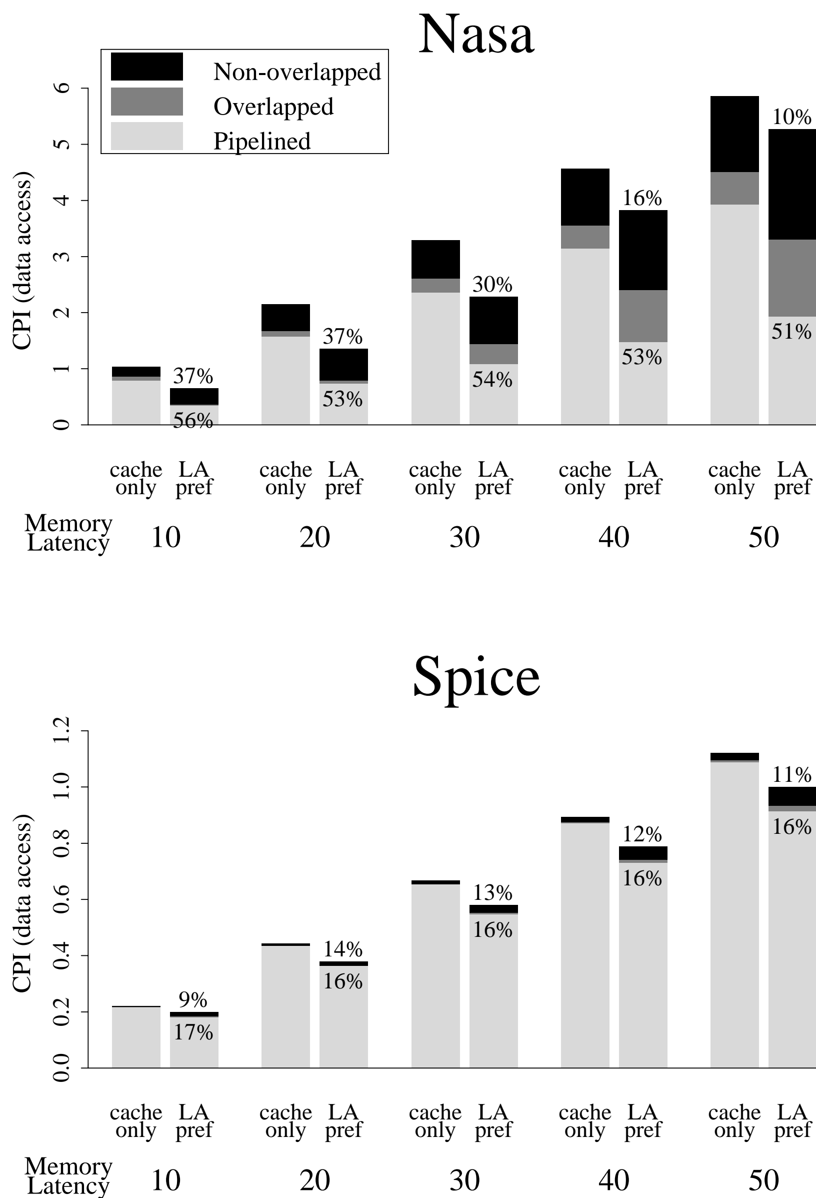


Figure A.1b: Effect of memory models and latencies (continued from Figure 4.4)

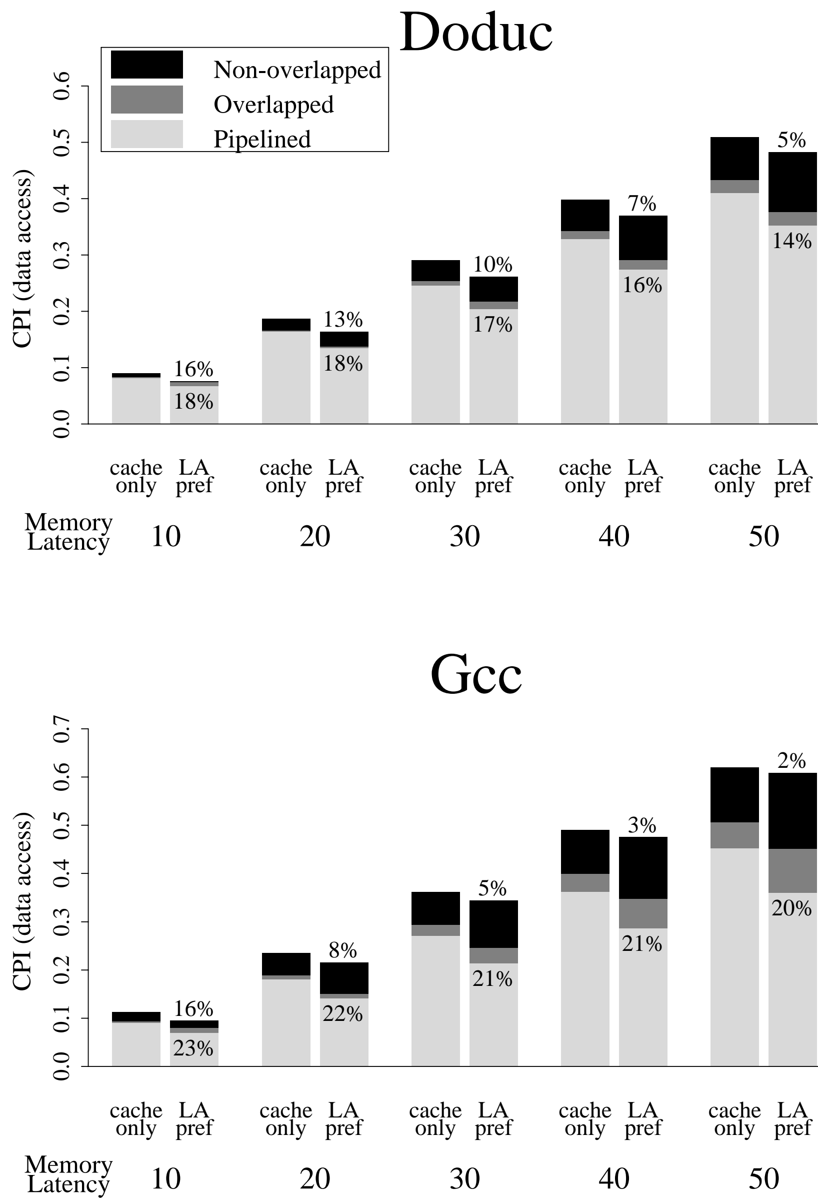


Figure A.1c: Effect of memory models and latencies (continued from Figure 4.4)

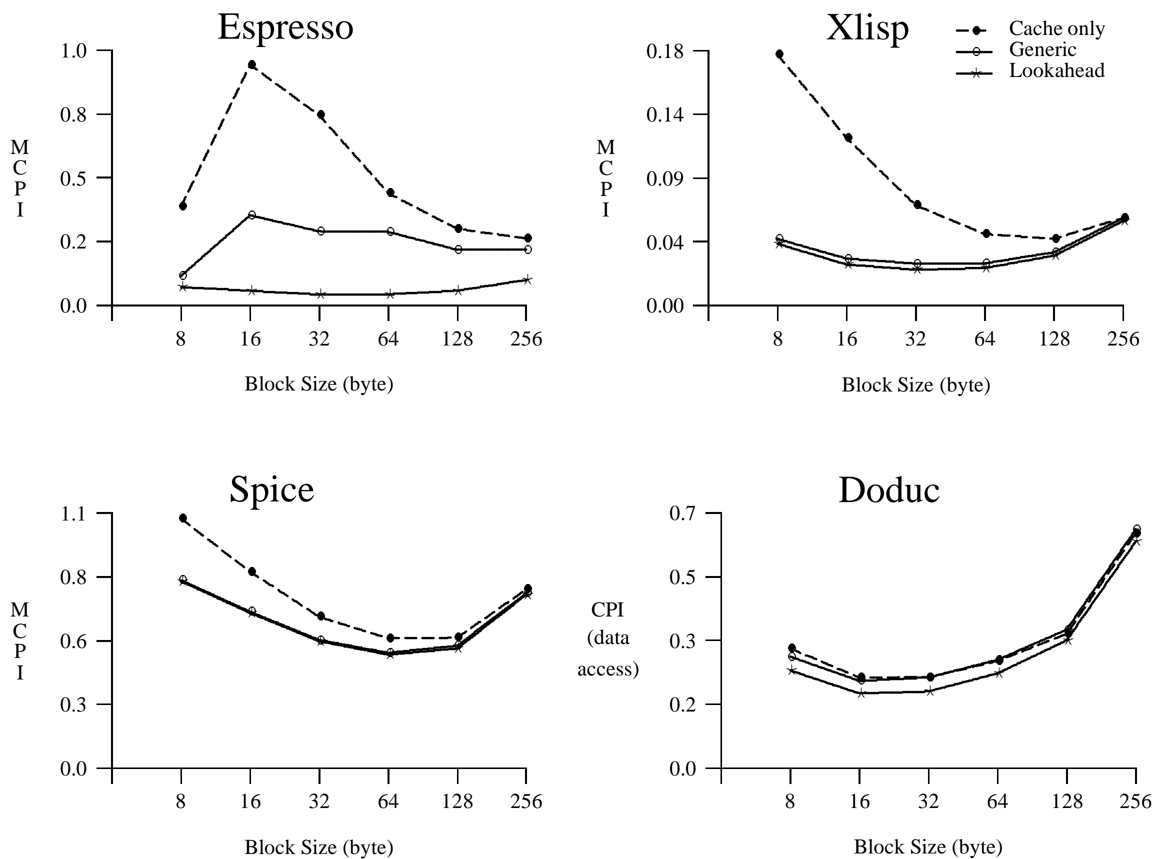


Figure A.2: MCPI vs. block size for 32K cache (continued from Figure 4.5)

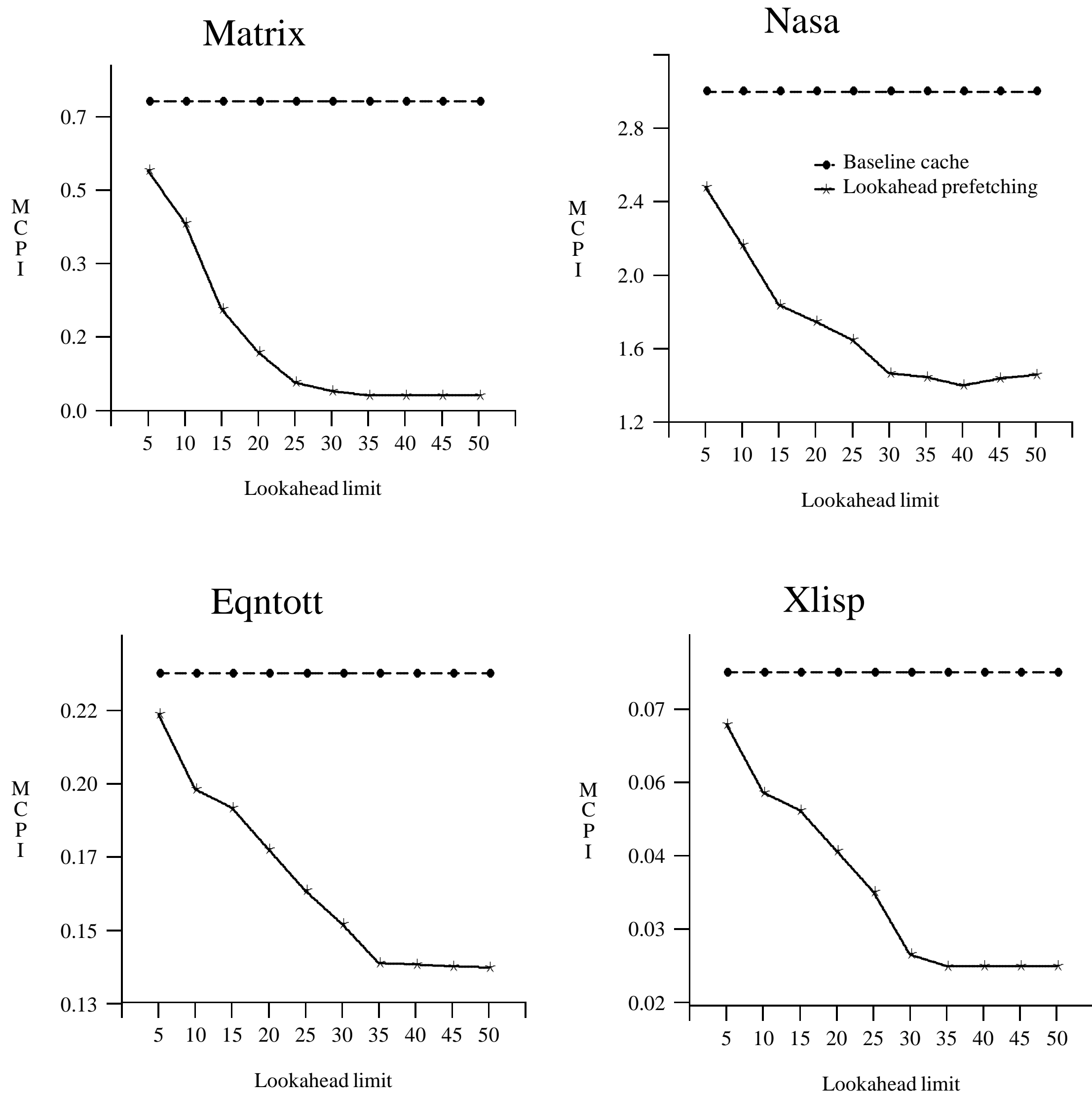
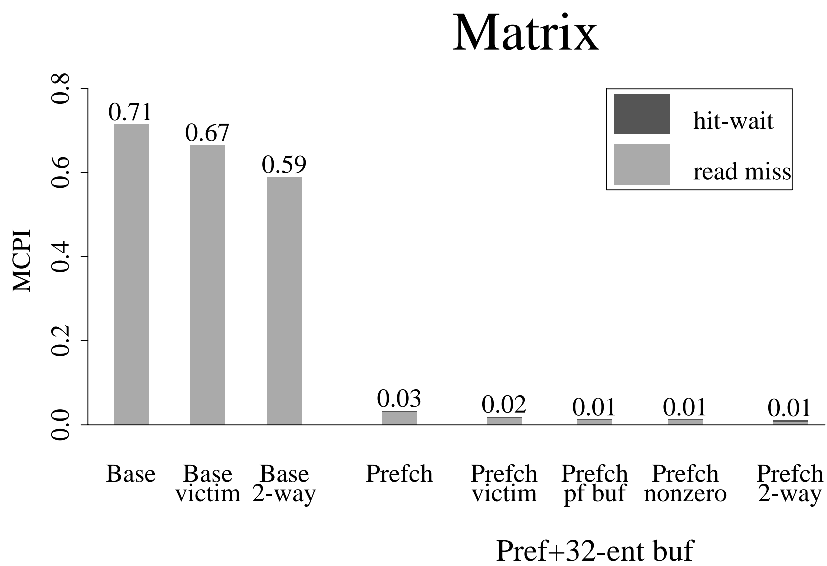


Figure A.3: MCPI vs. LA-limit (d) for $\delta = 30$ (continued from Figure 4.7)



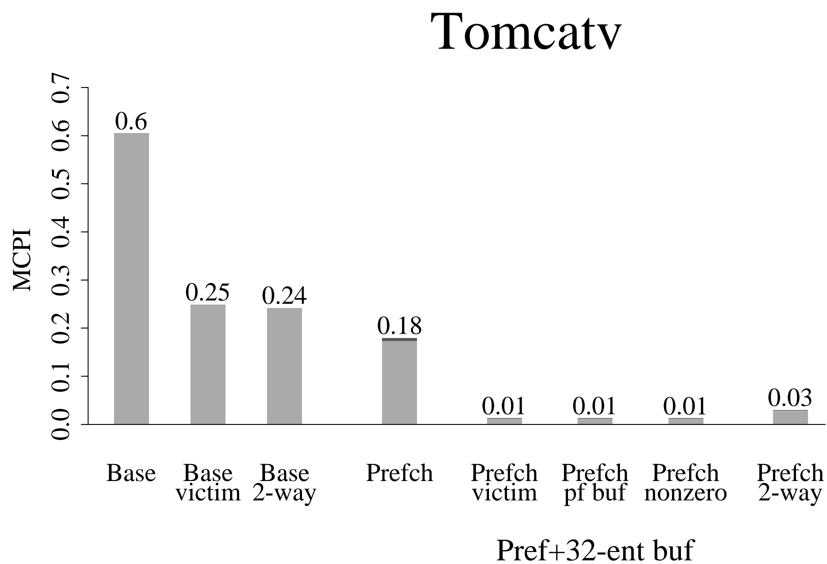


Figure A.4a: Variations in prefetching placement (continued from Figure 4.8)

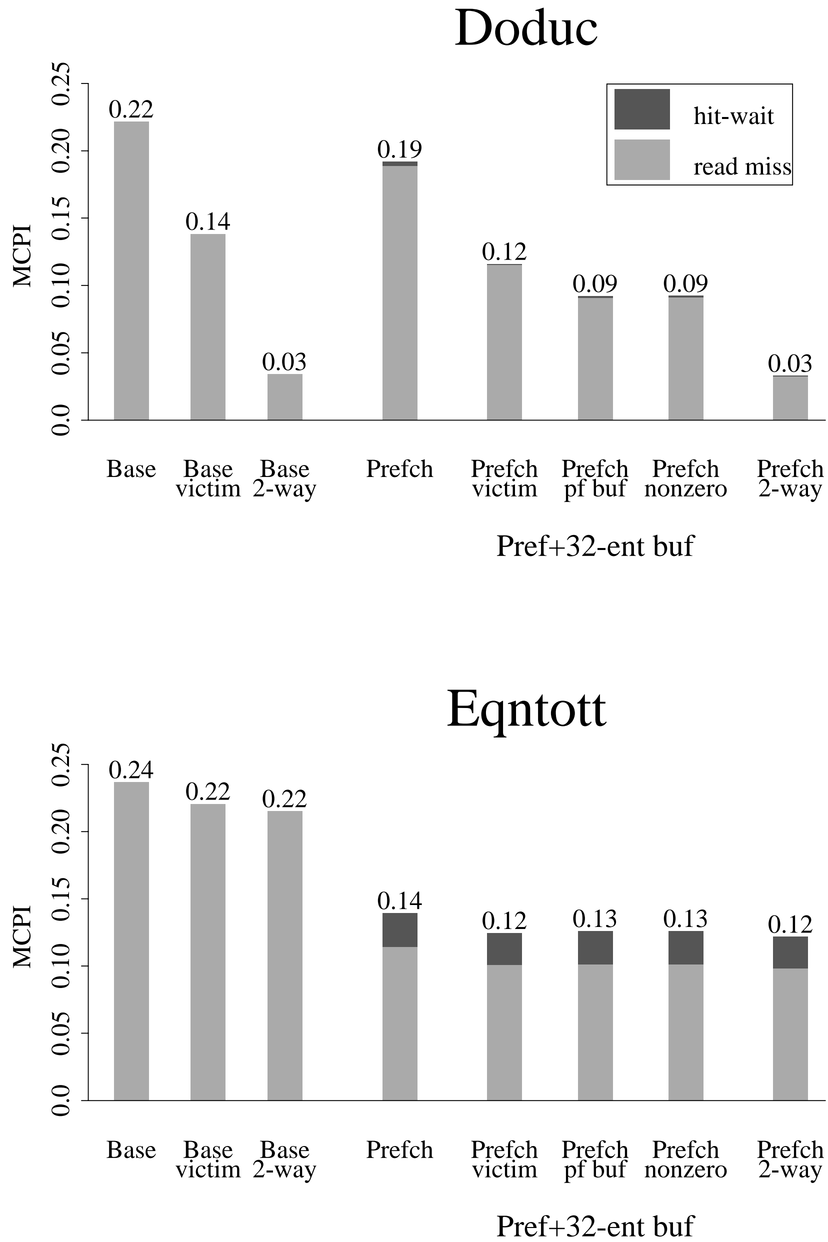


Figure A.4b: Variations in prefetching placement (continued from Figure 4.8)

A.2 Evaluation of Non-blocking Caches

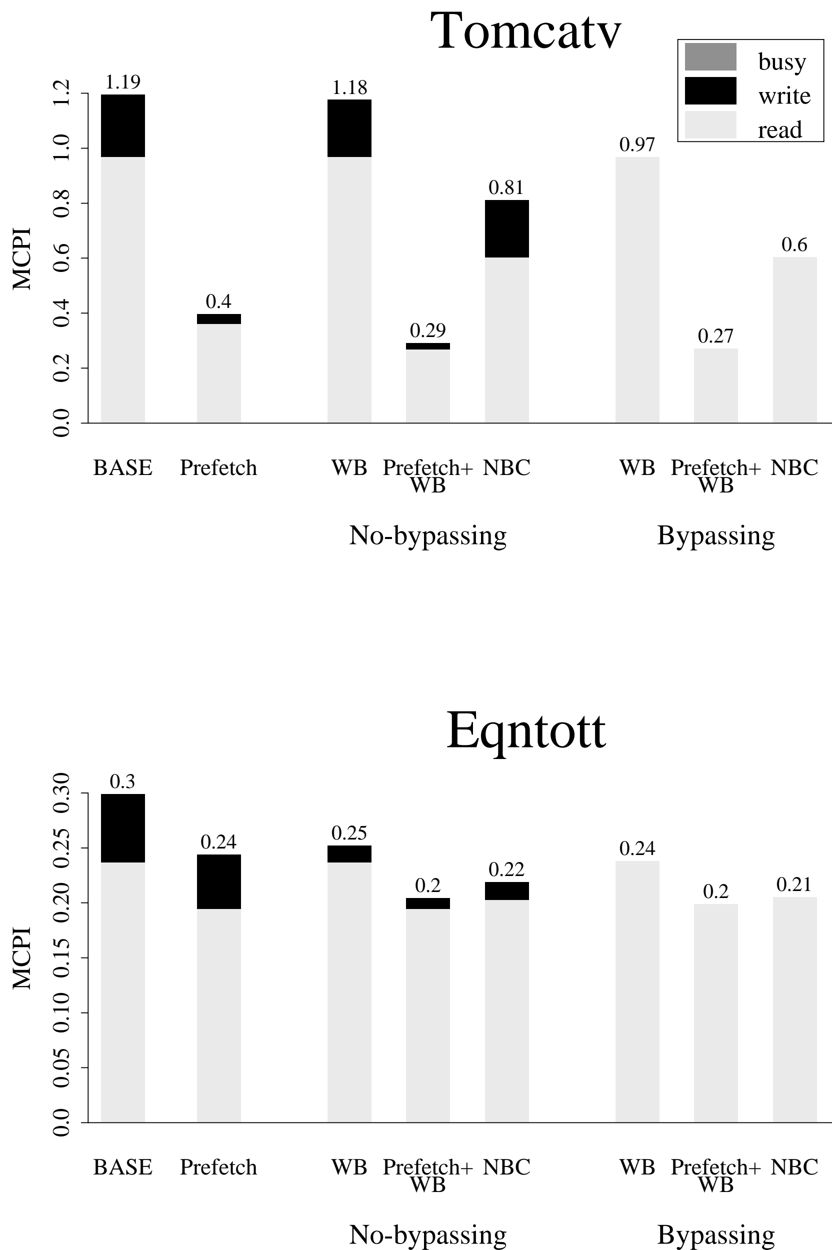
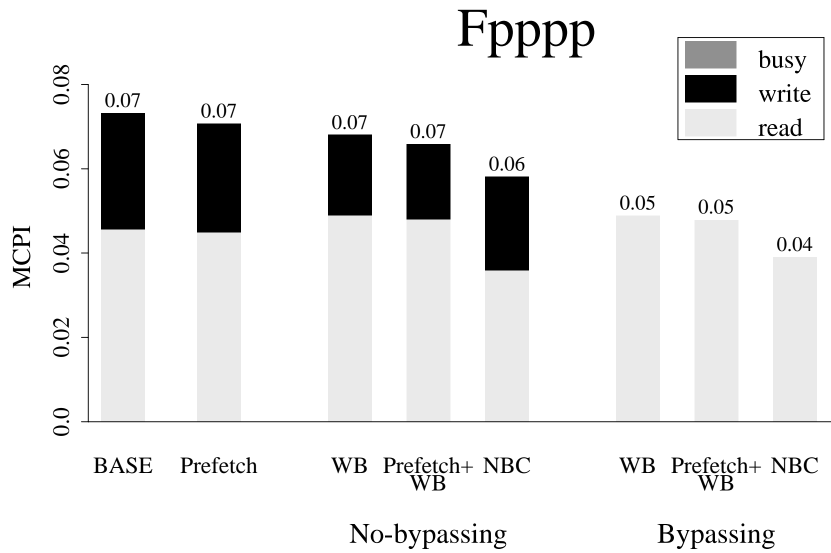


Figure A.5a: Prefetching vs. Lockup-free for $\delta = 30$ (continued from Figure 6.1)



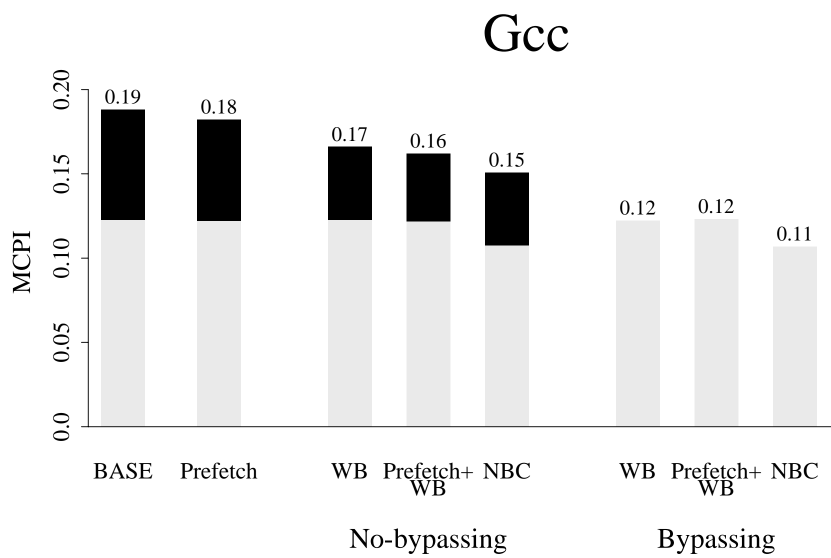


Figure A.5b: Prefetching vs. Lockup-free for δ = 30 (continued from Figure 6.1)

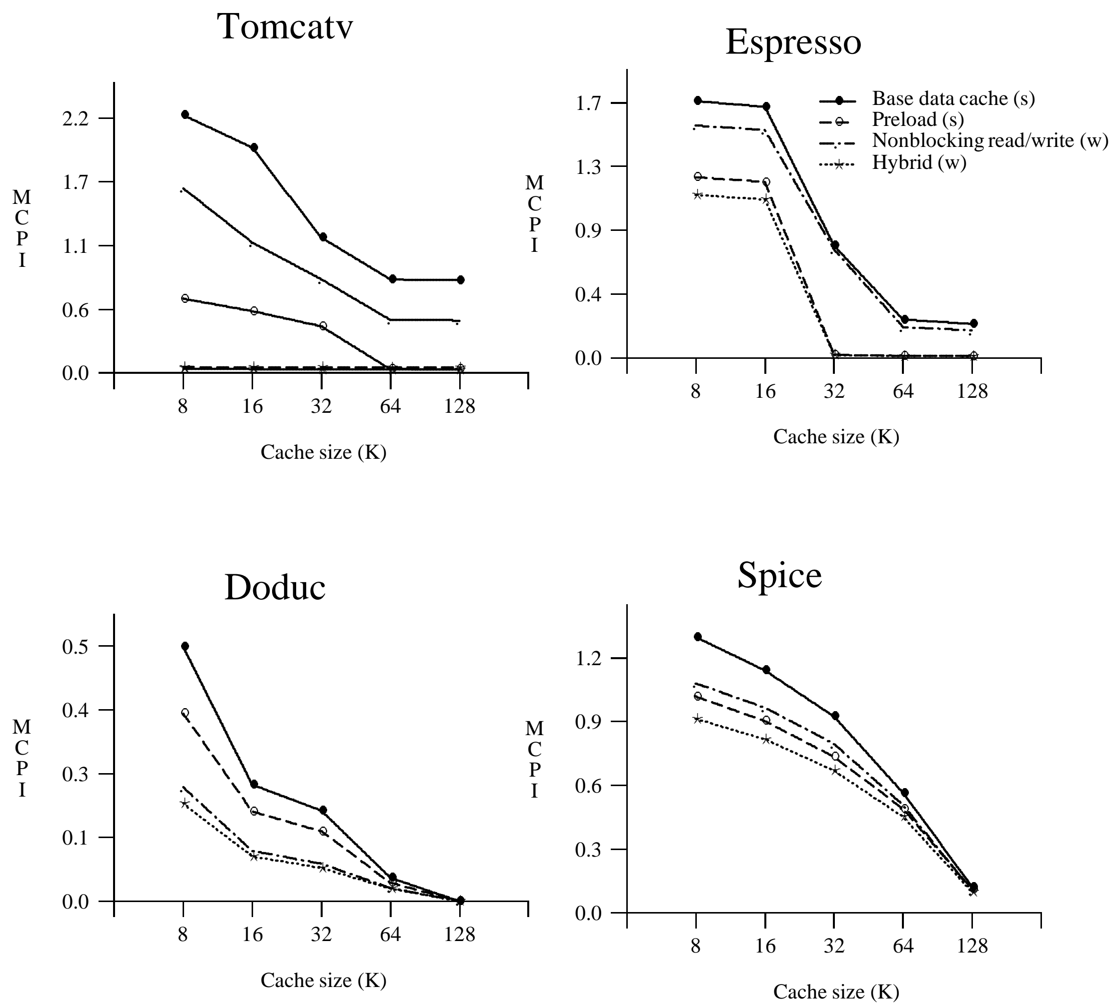


Figure A.6: Hybrid design on varying cache size $\delta = 30$ (continued from Figure 6.7)

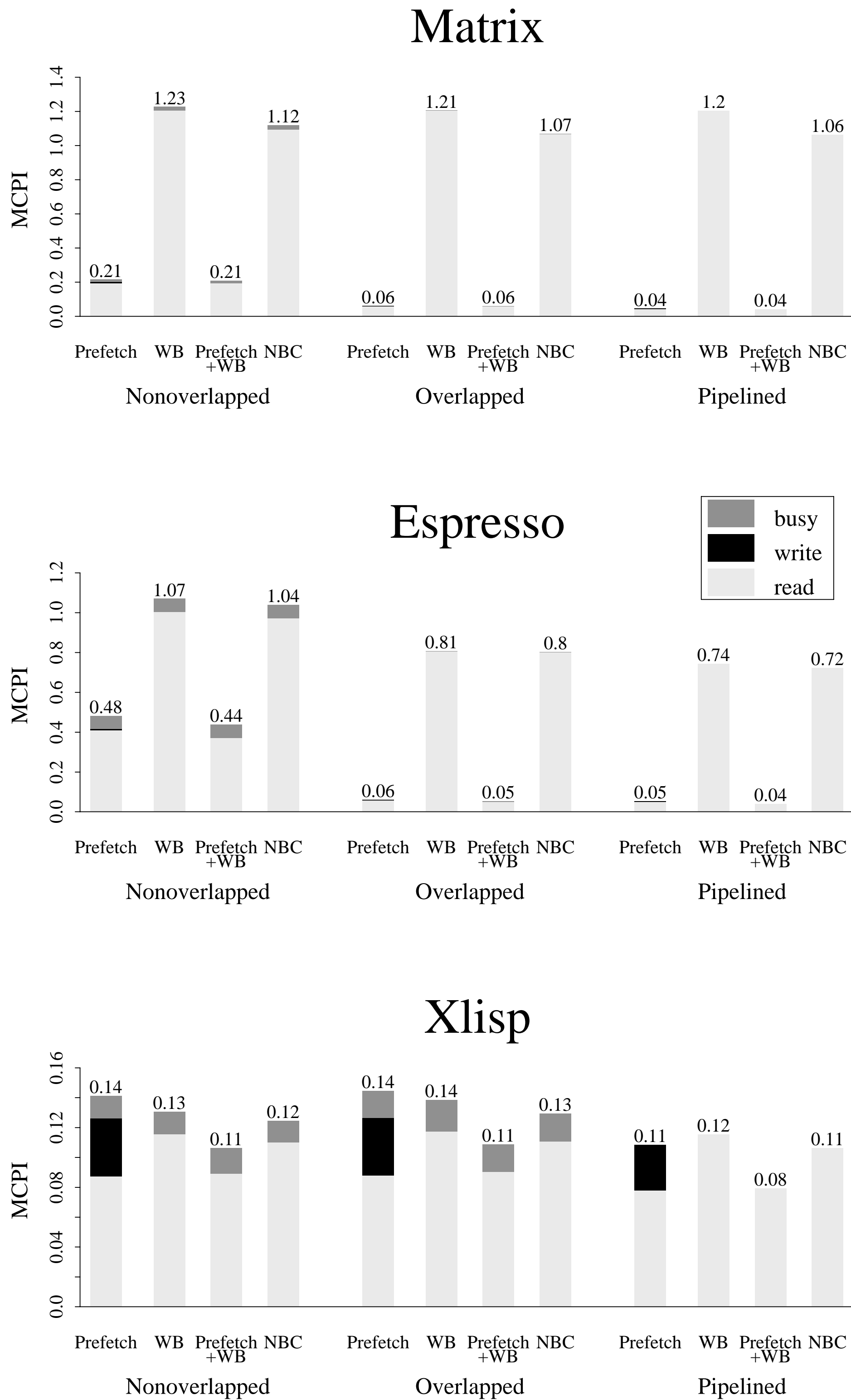
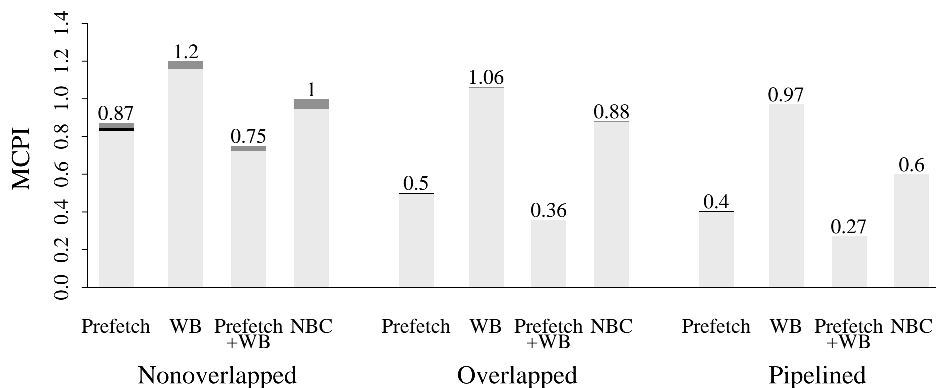
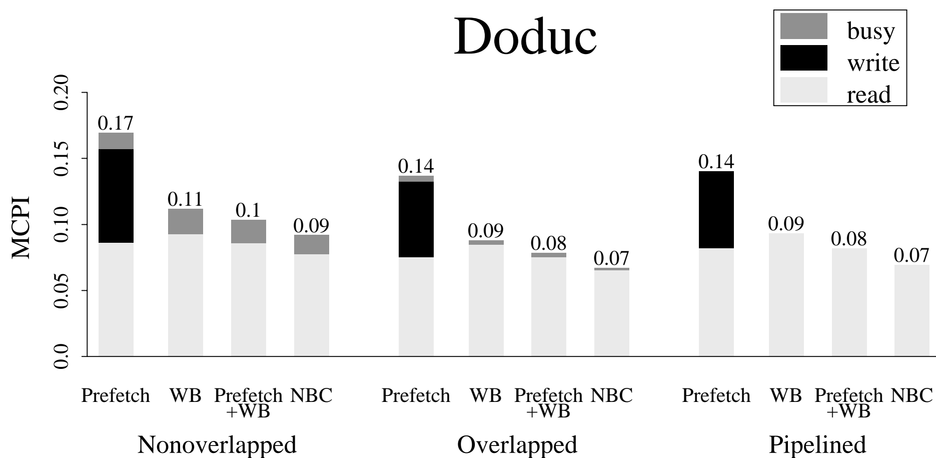


Figure A.7a: Effect of memory models: Prefetching vs. Lockup-free





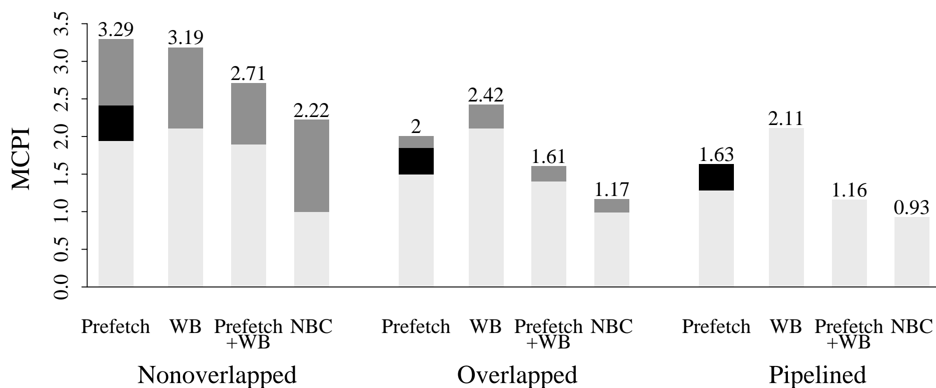


Figure A.7b: Effect of memory models: Prefetching vs. Lockup-free

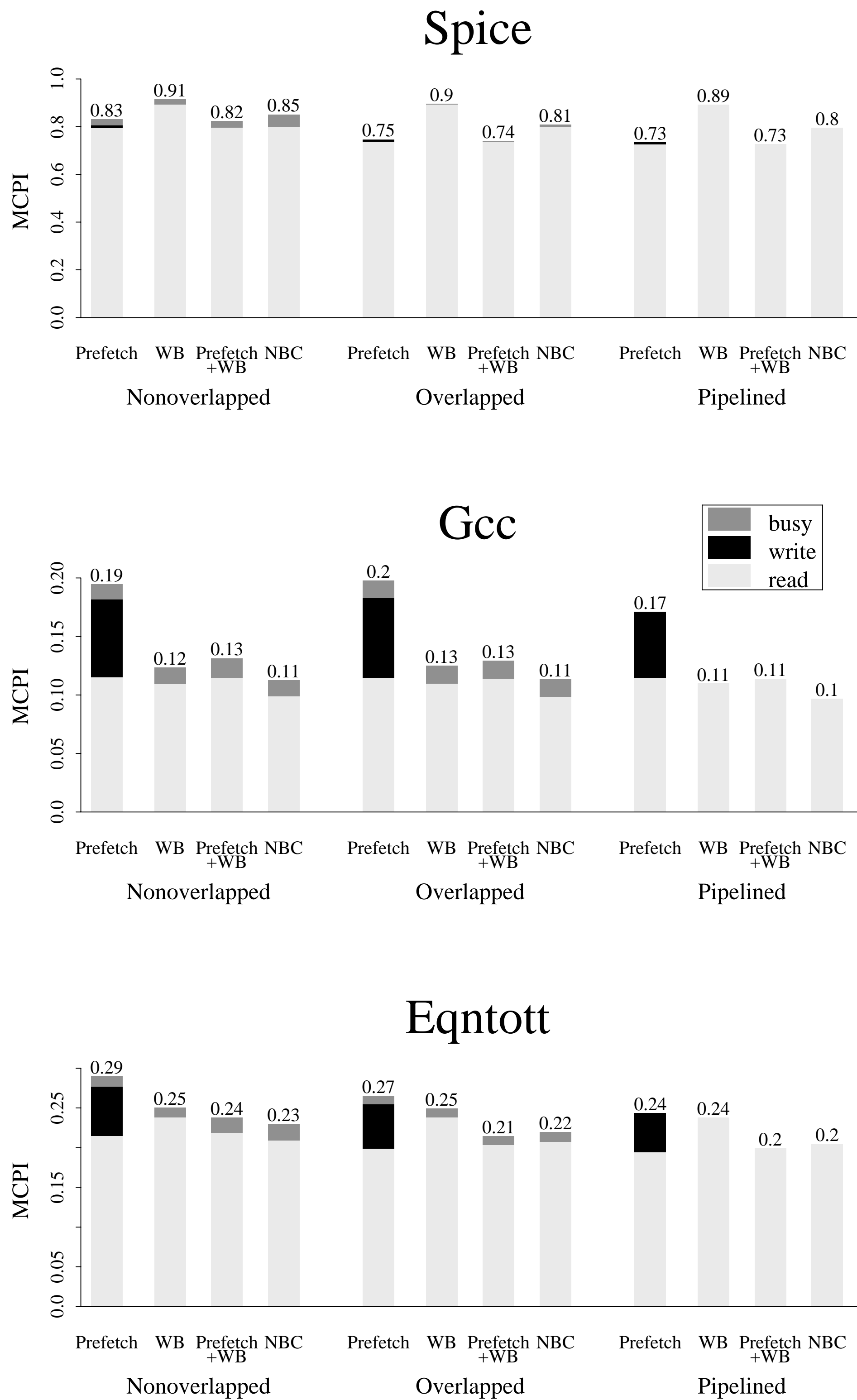


Figure A.7c: Effect of memory models: Prefetching vs. Lockup-free

Vita

Tien-Fu Chen was born in Kaohsiung, Taiwan, Republic of China on August 14, 1961. He graduated from Kaohsiung High School in Kaohsiung, Taiwan in 1979 and received a B.S. degree in Computer Science from National Taiwan University in 1983. After completed his military services, he joined Wang Computer Ltd., Taiwan as a software engineer for three years. From 1988 to 1993 he attended the University of Washington, receiving his M.S. degree in Computer Science in 1991 and his Ph.D. degree in Computer Science and Engineering. He will join the Department of Computer Science and Information Engineering at the National Chung Cheng University, Chiayi, Taiwan after completing his study at the University of Washington.