

2002

## Software and hardware methods for memory access latency reduction on ILP processors

Zhao Zhang

*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Zhang, Zhao, "Software and hardware methods for memory access latency reduction on ILP processors" (2002). *Dissertations, Theses, and Masters Projects*. Paper 1539623407.

<https://dx.doi.org/doi:10.21220/s2-97q8-0748>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).



**Software and Hardware Methods for Memory Access Latency  
Reduction on ILP Processors**

---

**A Dissertation**

**Presented to**

**The Faculty of the Department of Computer Science**

**The College of William & Mary in Virginia**

**In Partial Fulfillment**

**Of the Requirements for the Degree of**

**Doctor of Philosophy**

---

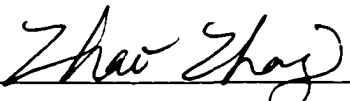
**by**

**Zhao Zhang**

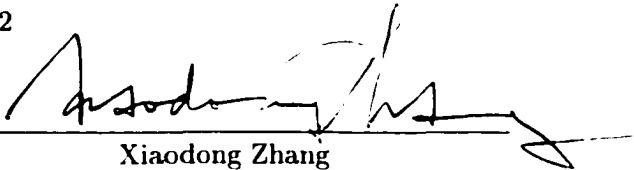
**2002**

# APPROVAL SHEET


This dissertation is submitted in partial fulfillment of  
the requirements for the degree of  
Doctor of Philosophy

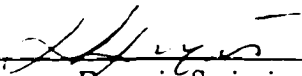
  
Zhao Zhang


Approved. June 2002

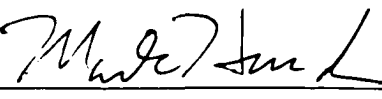
  
Xiaodong Zhang  
Thesis Advisor

  
William Bynum

  
Phil Kearns

  
Evgenia Smirni

  
Andreas Stathopoulos

  
Mark Hinders  
Department of Applied Science



# Table of Contents

<b>Acknowledgments</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Overview of Existing Studies . . . . .	3
1.2 Our Contributions . . . . .	7
1.3 Organization of the Dissertation . . . . .	11
<b>2 Background of Memory System Performance</b>	<b>12</b>
2.1 Cache Memories . . . . .	14
2.2 DRAM Technology . . . . .	16
2.2.1 DRAM Architecture and Operations . . . . .	16
2.2.2 Comparison of DRAM and SRAM . . . . .	19
2.2.3 DRAM Access Request and DRAM Operations . . . . .	19
2.2.4 DRAM Variants . . . . .	21
2.2.5 Memory Access Scheduling . . . . .	23
2.3 Dynamically Scheduled Superscalar Processors . . . . .	26

<b>3</b>	<b>Evaluation Methodology: An Experimental Approach</b>	<b>29</b>
3.1	Architectural Simulations . . . . .	30
3.2	SimpleScalar Tool Set . . . . .	32
3.3	Our SimpleScalar Extensions . . . . .	34
3.4	SPEC and TPC-C Benchmarks . . . . .	37
3.4.1	SPEC CPU95 Benchmark . . . . .	37
3.4.2	SPEC CPU2000 Benchmark . . . . .	37
3.4.3	TPC-C Benchmark . . . . .	40
<b>4</b>	<b>Fast Bit-Reversals on Uniprocessors and Shared-Memory Multiproces-</b>	
	<b>sors</b>	<b>41</b>
4.1	Blocking for Bit-reversals . . . . .	46
4.2	Blocking with Buffers . . . . .	47
4.2.1	Blocking With a Software Buffer and Its Limits . . . . .	47
4.2.2	Cache Structure Dependent Blocking . . . . .	48
4.2.3	Victim-cache-aided Blocking . . . . .	51
4.3	Blocking with Padding . . . . .	52
4.4	Blocking and Padding for a TLB . . . . .	55
4.4.1	Blocking for a Fully Associative TLB . . . . .	55
4.4.2	Padding for a Set-associative TLB . . . . .	56
4.5	Experimental Results and Performance Evaluation . . . . .	58
4.5.1	Experimental Environment and Evaluation Methodology . . . . .	58
4.5.2	Effects of TLB and Virtual Memory . . . . .	60

4.5.3	Performance of the Hybrid Method for Bit-reversals . . . . .	63
4.5.4	Performance Comparisons on the SGI O2 . . . . .	63
4.5.5	Performance Comparisons on the Sun Ultra-5 . . . . .	65
4.5.6	Performance Comparisons on the Sun E-450 . . . . .	65
4.5.7	Performance Comparisons on the Pentium-II 400 . . . . .	67
4.5.8	Performance Comparisons on the Compaq XP-1000 . . . . .	68
4.6	Performance Evaluation on SMP Multiprocessors . . . . .	69
4.6.1	Performance Comparisons on the Sun E-450 . . . . .	70
4.6.2	Performance Comparisons on the HP 9000 V2200 . . . . .	73
4.7	Summary . . . . .	74
<b>5</b>	<b>Reduce DRAM Row-buffer Conflicts by Breaking Address Mapping Sym-</b>	
	<b>metry</b>	<b>76</b>
5.1	Existing Address Mapping Schemes . . . . .	78
5.2	Mapping Symmetry and Row Buffer Conflicts . . . . .	79
5.2.1	Large-cache Condition in Computer Systems . . . . .	81
5.2.2	Effect of Cache Writebacks . . . . .	81
5.2.3	The Effect of Cache Conflict Misses . . . . .	83
5.3	A Permutation-based Page Interleaving . . . . .	84
5.3.1	The Scheme and Its Properties . . . . .	84
5.3.2	Comparisons with the Swapping Scheme . . . . .	88
5.4	Experimental Environment . . . . .	89
5.5	Performance Evaluation of Permutation-based Page Interleaving Scheme . .	90

5.5.1	Reductions of Row-buffer Miss Rates . . . . .	91
5.5.2	Effects of Memory Organization Variations . . . . .	93
5.5.3	Effects of Write Buffers . . . . .	94
5.5.4	Overall Performance Improvement . . . . .	96
5.6	Breaking Mapping Symmetry at Cache Level . . . . .	98
5.6.1	Bitwise-XOR and Polynomial Mapping . . . . .	98
5.6.2	Reduction of Miss Rates . . . . .	99
5.6.3	Comparisons of Overall Performance . . . . .	102
5.6.4	Tradeoffs between Cache Mapping Schemes and DRAM Interleaving Schemes . . . . .	103
5.7	Considerations of Large Cumulative Row Buffer Sizes . . . . .	104
5.8	Other Related Work . . . . .	106
5.9	Conclusion . . . . .	109
<b>6</b>	<b>Cached DRAM: A Simple and Effective Technique for Memory Access Latency Reduction on ILP Processors</b>	<b>111</b>
6.1	Structure and Operations of Cached DRAM . . . . .	114
6.2	Experimental Environment . . . . .	118
6.3	Comparisons of Overall Performance . . . . .	120
6.3.1	On-memory Cache Organizations . . . . .	121
6.3.2	Performance Improvement of Cached DRAM over SDRAM . . . . .	122
6.3.3	Performance Comparisons of Cached DRAM and Other DRAM Ar- chitectures . . . . .	124

6.4	Cached DRAM with Increasing ILP Degree . . . . .	126
6.5	Comparisons with Approaches Exploiting Row Buffer Locality . . . . .	128
6.6	Summary . . . . .	130
<b>7</b>	<b>Fine-grain Priority Scheduling on Multi-channel Memory Systems</b>	<b>131</b>
7.1	Memory System Considerations . . . . .	135
7.1.1	Multi-channel Memory Systems . . . . .	135
7.1.2	DRAM Mapping Scheme . . . . .	136
7.2	Fine-grain Priority Scheduling . . . . .	138
7.2.1	Granularity of Scheduling . . . . .	138
7.2.2	Scheduling Policies . . . . .	139
7.3	Complexity Analysis . . . . .	141
7.3.1	Complexity inside Processor . . . . .	141
7.3.2	Complexity in Memory Controller . . . . .	142
7.4	Experimental Environment . . . . .	144
7.5	Experimental Results . . . . .	145
7.5.1	Burstiness in Miss Streams . . . . .	145
7.5.2	Potentials of Fine-grain Priority Scheduling . . . . .	146
7.5.3	Performance Improvement of Fine-grain Priority Scheduling . . . . .	151
7.6	Other Related Work . . . . .	154
7.7	Summary . . . . .	156
<b>8</b>	<b>Constructing Large Size and Low Overhead Off-Chip Caches by Cached DRAM</b>	<b>157</b>

8.1	The CDC Design . . . . .	161
8.1.1	On-chip CDC Controller, Tag Cache, and Predictor . . . . .	161
8.1.2	CDC-DRAM Mapping . . . . .	163
8.1.3	CDC-cache Mapping . . . . .	164
8.1.4	The Predictor . . . . .	165
8.1.5	Write Policy . . . . .	166
8.1.6	Cache Coherence in Multiprocessor Environment . . . . .	166
8.2	Experimental Setup . . . . .	167
8.3	Performance Results . . . . .	169
8.3.1	Performance of SRAM-L3 . . . . .	171
8.3.2	Comparisons between CDC-predict and SRAM-L3 . . . . .	173
8.3.3	Comparisons of the CDC Variants . . . . .	174
8.4	Summary . . . . .	175
<b>9</b>	<b>Conclusions and Future Work</b>	<b>177</b>
9.1	Conclusions . . . . .	177
9.1.1	Memory Latency Bottleneck . . . . .	177
9.1.2	Our Approaches and Their Effectiveness . . . . .	179
9.2	Future Work . . . . .	181
9.2.1	Effective Memory Access Scheduling for Future Processors . . . . .	181
9.2.2	Improving the Performance of CDC . . . . .	183
<b>A</b>	<b>The Bit-reversal Program Using the Padding Method</b>	<b>185</b>



## ACKNOWLEDGMENTS

This dissertation is a result of not only many years of efforts, but also the training, mentoring, educating, and support of many people. First of all, I would like to sincerely acknowledge Dr. Xiaodong Zhang, my advisor, for his mentoring and support for these five years. I am grateful for the numerous hours he have spent generously with me, which he spared from his heavy duties of research, service, and teaching. I appreciate his hard working and persistence, vast knowledge in many areas, and vision and determination to build up an excellent research team.

The Department of Computer Science at the College of William and Mary is a wonderful place for me to pursue my Doctoral degree. I would like to thank all faculty and staff in this department for their help in these years. I would like to acknowledge Dr. William Bynum, Dr. Phil Kearns, Dr. Evgenia Smirni, and Dr. Andreas Stathopoulos for serving the dissertation committee. I would like to thank Dr. Mark Hinders from the Department of Applied Science to take time from his busy schedule to serve as the external member of the committee. Especially, I thank Dr. William Bynum for his reviewing many of my manuscripts, and Dr. Evgenia Smirni and Dr. Bruce Lowekamp for their help in my career development. Dr. Bob Collins talked with me frequently in my first year in this department, obviously with the intention to help me in speaking English. I would also like to acknowledge Vanessa Godwin for the great help she has offered to me in my graduate study.

I would like to thank the members of the High Performance Computing and Software Lab, who have helped to create an environment full of stimulation. Dr. Yong Yan and Dr. Xing Du were great help to me, not only when we worked together but also after they have started their own careers. I would like to acknowledge Dr. Guangzhi Li, Zhichun Zhu, Li Xiao, Stefan Kubricht, Songqing Chen, Xin Chen, and Song Jiang for those wonderful discussions and arguments. With many of them I not only share the excitement of research,



but also enjoy a lot of fun as friends.

I would like to thank other graduate students in the department, including Lei Guo, Aaron Hawkins, Zvezdan Petkovic, Alma Riska, Wei Sun, and Qi Zhang, but not exclusively, for those interdisciplinary discussions and for their friendship.

I would like to acknowledge Dr. Yiming Hu of the Department of Electrical & Computer Engineering and Computer Science at University of Cincinnati, and Dr. Hong Wang in Intel Microprocessor Research Labs, for both their insightful comments on technical issues and their kind help in my career development.

I would also like to acknowledge the funding agencies that provided funds and equipment that supported my research: Sun Microsystems, for their donations of workstations and servers, and the National Science Foundation and the Air Force Office of Scientific Research, whose grants funded the majority of my graduate research.

I am deeply grateful to my parents, Guoxiang Zhang and Xiyin Wan, and my sisters, Qing Zhang and Xi Zhang, and my brother-in-law, Bin Feng, all of whom have given me the upbringing and education that allowed me to reach this point. A very special thanks goes out to my mother, who started my education early by teaching me Chinese language, arithmetic, and a little bit of English when I was only a few years old.

Most of all, I must acknowledge my wife, Zhichun Zhu, for her constant support, assistance, and encouragement through these years. She helped me survive tough times, meanwhile she suffered through my all kinds of pressure, loss of direction, and rotten moods. I could have never made it this far without her.

## ABSTRACT

While microprocessors have doubled their speed every 18 months, performance improvement of memory systems has continued to lag behind. To address the speed gap between CPU and memory, a standard multi-level caching organization has been built for fast data accesses before the data have to be accessed in DRAM core. The existence of these caches in a computer system, such as L1, L2, L3, and DRAM row buffers, does not mean that data locality will be automatically exploited. The effective use of the memory hierarchy mainly depends on how data are allocated and how memory accesses are scheduled. In this dissertation, we propose several novel software and hardware techniques to effectively exploit the data locality and to significantly reduce memory access latency.

We first presented a case study at the application level that reconstructs memory-intensive programs by utilizing program-specific knowledge. The problem of bit-reversals, a set of data reordering operations extensively used in scientific computing program such as FFT, and an application with a special data access pattern that can cause severe cache conflicts, is identified in this study. We have proposed several software methods, including padding and blocking, to restructure the program to reduce those conflicts. Our methods outperform existing ones on both uniprocessor and multiprocessor systems.

The access latency to DRAM core has become increasingly long relative to CPU speed, causing memory accesses to be an execution bottleneck. In order to reduce the frequency of DRAM core accesses to effectively shorten the overall memory access latency, we have conducted three studies at this level of memory hierarchy. First, motivated by our evaluation of DRAM row buffer's performance roles and our findings of the reasons of its access conflicts, we propose a simple and effective memory interleaving scheme to reduce or even eliminate row buffer conflicts. Second, we propose a fine-grain priority scheduling scheme to reorder the sequence of data accesses on multi-channel memory systems, effectively exploiting the available bus bandwidth and access concurrency. In the final part of the dissertation, we first evaluate the design of cached DRAM and its organization alternatives associated with ILP processors. We then propose a new memory hierarchy integration that uses cached DRAM to construct a very large off-chip cache. We show that this structure outperforms a standard memory system with an off-level L3 cache for memory-intensive applications.

Memory access latency has become a major performance bottleneck for memory-intensive applications. As long as DRAM technology remains its most cost-effective position for making main memory, the memory performance problem will continue to exist. The studies conducted in this dissertation attempt to address this important issue. Our proposed software and hardware schemes are effective and applicable, which can be directly used in real-world memory system designs and implementations. Our studies also provide guidance for application programmers to understand memory performance implications, and for system architects to optimize memory hierarchies.

**Software and Hardware Methods for Memory Access Latency  
Reduction on ILP Processors**

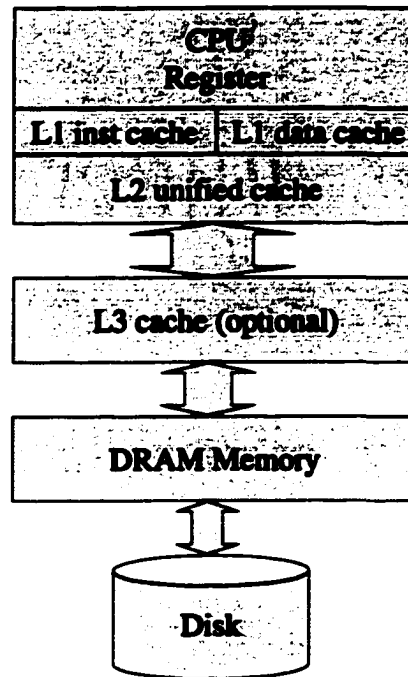
# Chapter 1

## Introduction

The performance of processors has been doubling every 18 months as predicted by Moore's Law for decades. This fast improvement comes from the advances in circuit design, fabrication technology, and architectural innovations. However, the improvement of memory systems has lagged far behind that rate. While the speed of high-end processors increases by 60% per year, the speed of DRAM increases by only 7% per year [44]. In other words, the speed gap between the CPU and the DRAM increases by more than 50% per year. Currently, the time cost of a DRAM access is equivalent to that of executing hundreds of instructions inside a processor. Without addressing the concern of the CPU-memory gap, further improvement of processor performance will eventually vanish because of the dominance of memory stall time.

Traditionally, this issue has been addressed by constructing a memory hierarchy, as shown in Figure 1.1, where small and fast SRAM caches are at the top levels, and slow and large DRAM memory is at the bottom level. When working effectively, the top-level caches can serve most memory references, minimizing the number of accesses to the lower-level DRAM memory. However, as the speed gap between the CPU and DRAM memory continues to widen, even a small fraction of memory references falling into the DRAM

memory will make the memory stall times dominate in the total execution times of memory-intensive applications.



**Figure 1.1:** A typical memory hierarchy in high-end workstations.

## 1.1 Overview of Existing Studies

Computer architecture research on memory hierarchy is focused on optimizing the hardware organization of caches [101, 76], especially on L1 instruction caches and L1 data caches, which satisfy more than 90 percent of memory references for most workloads. From computer architecture point of view, the cache performance is determined by the cache miss rates, cache access time, cache bandwidth, and miss penalty.

Increasing the cache size and using set-associative cache are effective methods to reduce cache miss rates. However, they usually cause an increase of access time, which increases either processor cycle time or pipeline length and delays the execution of all instructions. Many studies have been done to achieve the best trade-off between reducing cache miss rates and reducing cache access time. The study in [47] investigates the use of simple, direct mapped cache and the best balance between cache size and cache access time. New cache organizations (e.g. [76]) have been proposed to achieve the fast access times of direct mapped caches and the low cache miss rates of set associative caches, such as decoupled caches [103, 69, 98, 57, 77], multiple-access caches [2, 1, 123, 16, 78, 10], augmented caches [56, 87, 55], and multilevel caches [106, 5, 6, 116, 80, 110].

As processors aggressively exploit instruction-level parallelism (ILP), cache bandwidth becomes a serious limitation of processor performance. Issuing multiple instructions per cycle, processors need to fetch multiple instructions and data items per cycle. Wide instruction cache ports (outputting multiple consecutive instructions per access) and instruction buffers [41] exploit the spatial locality in the instruction fetch stream to increase cache bandwidth. Trace cache [92, 52, 36, 12] uses branch prediction information to concatenate non-contiguous instruction blocks separated by branch instructions into larger, contiguous blocks. In this way, more than one basic block (contiguous instructions without a branch) can be fetched in one cycle. Accesses to a data cache are less regular than those to an instruction cache. Thus, different techniques are needed to increase the bandwidth of data caches. Data caches with two interleaved banks [9, 120] allow two simultaneous accesses as long as they fall into different banks. A duplicating data cache [30] has two caches with identical contents, doubling the bandwidth with the cost of double-sized chip area. An-

other approach is phase-pipeline access [66], with which the cache outputs data on both clock edges. Although this technique saves chip area, the cache has to be as twice fast as the processor. As complexity-effective alternatives, authors in [88] propose locality-based interleaved cache that enhances the cache by adding a multi-ported single line buffer to each cache bank, and authors in [20] propose to use separate caches for program data, heap, and stack regions.

Software approaches, such as compiler ones, operating systems ones, and application-level ones also play an important role in reducing cache miss rate. Compiler approaches reduce cache miss rate by either improving program locality, optimizing data layout, or exploiting the opportunity of software prefetching. There are two types of locality optimization techniques: program transformation and data layout transformation. Program transformation [64, 72] tries to determine the best sequence of memory references that maximizes the reuse of cached data. Data layout transformation [53, 111] performs packing, padding, and aligning on data arrays and data structures to reduce data size and to avoid cache conflicts. Compilers can also insert prefetch instructions into programs [17, 74] to hide cache miss latency, effectively reducing cache miss rates. However, this approach is normally restricted to loop-based program code because the compiler must precisely predict memory access patterns.

Operating systems approaches reduce cache conflict misses by carefully mapping virtual memory pages onto physical memory pages. A static page mapping policy [60], such as page coloring, uses heuristics to select the mapping that is most likely to minimize cache conflict misses. A dynamic page mapping policy [11] employs special hardware to detect severe conflict miss conditions and dynamically changes the mapping of related pages to

avoid the conflicts.

Application-level approaches improve cache locality or reduce cache conflict misses by exploiting the knowledge of application memory access patterns as well as the knowledge of the architecture on which the application is running. As in compiler approaches, the program access pattern or the data layout or both are reconstructed. Unlike compiler approaches, it is the application programmers who can make the changes because only the programmers, not the compilers, have the global knowledge about the interaction between the program and the underlying architecture. For example, a run-time method has been proposed in [119] to optimize the cache locality of a set of programs with dynamic memory access patterns running on shared-memory multiprocessors. Reconstructing the execution of programs into small computation threads and using a bin-space-based task partitioning method to assign those threads onto computing nodes, this method maximizes the reuse of cached data and achieves load balance among computing nodes. In this dissertation (Chapter 4), we also present a study of cache-optimal methods for bit-reversals, an application with severe cache conflicts on typical cache organizations. One method we propose is to reconstruct the layout of data arrays by inserting small padding spaces such that concurrently accessed data will not conflict in the cache.

Although numerous studies have been done to improve the effectiveness of memory hierarchy and many of the proposed cache techniques have been implemented, memory stall time will continue to increase due to the ever increasing CPU-memory speed gap. For applications with large working sets and irregular access patterns, the cache misses due to limited cache capacity can not be eliminated. For those applications, even if the cache miss rate is reduced to the minimum level, the memory stall time due to DRAM accesses can



easily dominate in the total execution time because of the very long DRAM access latency (relative to processor speed). For example, if a program has 0.5 misses per 100 instructions executed, and the cost of a DRAM access is equivalent to executing 200 instructions, (a 2GHz 2-way issue processor can execute up to two instructions in  $0.5ns$ , while a typical DRAM access will take  $50ns$ .) the memory stall time can be up to one half of the total execution time of the program.

## 1.2 Our Contributions

We have started our memory system study at the application level to reconstruct memory-intensive programs by utilizing program-specific knowledge. The problem of bit-reversals, a set of data reordering operations extensively used in scientific computing programs, such as FFT, is identified for such a study. This application has a special data access pattern [58] that can cause severe cache conflicts. In this study, we examine different methods using techniques of blocking, buffering, and padding for efficient implementations of bit-reversals. We evaluate the merits and limits of each technique and their applications and architecture-dependent conditions for developing cache-optimal methods. Besides testing the methods on different uniprocessors, we conducted both simulation and measurements on two commercial SMP multiprocessors to provide architectural insights into the methods and their implementations. We made two contributions in this study: (1) Our integrated blocking methods, which match cache associativity and TLB cache size and which fully use the available registers, are cache-optimal and fast. (2) We show that our padding methods outperform other software oriented methods, and believe they are the fastest in terms of

minimizing both CPU and memory access cycles. Since the padding methods are almost independent of hardware, they could be widely used on many uniprocessor workstations and multiprocessors.

The main focus of this dissertation is on DRAM memory. The access latency to DRAM core has become increasingly long relatively to the CPU speed, causing DRAM accesses to be the performance bottleneck of many workloads. In order to exploit the potential of DRAM organizations to effectively shorten the overall memory access latency, we have conducted three studies at this level of the memory hierarchy. First, we investigate the potential of exploiting DRAM row buffer locality [124]. Each access to DRAM core must go through the row buffer, which has a shorter access latency than the DRAM core. DRAM row-buffer conflicts occur when a sequence of requests on different rows fall onto the same memory bank, causing a much higher memory access latency than requests to the same row or to different banks. We analyze the sources of row-buffer conflicts in the context of dynamically scheduled superscalar processors, and propose a permutation-based page interleaving scheme to reduce row-buffer conflicts and to exploit data access locality in the row buffer. Compared with several existing schemes, we show that the permutation-based scheme dramatically increases hit rates in DRAM row buffers and reduces memory stall time of SPEC2000 benchmark programs.

Second, we proposed a fine-grain priority scheduling scheme for recent multi-channel DRAM memory systems [126]. Configurations of contemporary DRAM memory systems become increasingly complex. A recent study shows that application performance is highly sensitive to DRAM memory configuration, and suggests that tuning burst sizes and channel configurations be an effective way to optimize the DRAM performance for a given memory-

intensive workload [24]. However, this approach is workload dependent. In this study, we show that by utilizing fine-grain priority access scheduling, we are able to find a workload independent configuration that achieves optimal performance on a multi-channel memory system. Our approach can well utilize the available high concurrency and high bandwidth on such memory systems, and can effectively reduce memory stall time.

In the final part of the dissertation, we first evaluate the performance of cached DRAM and its design alternatives in the context of ILP processors [124]. Then we propose a new memory hierarchy organization that uses cached DRAM to construct a very large off-chip cache. We show that this off-chip cache outperforms an off-chip SRAM cache in a standard memory system for memory-intensive applications.

Cached DRAM is an existing technology that integrates a small cache into DRAM chip. By exploiting the spatial locality of memory access streams missing from the L2 cache, a cached DRAM can reduce the average DRAM access time. Previous studies have shown that cached DRAM is effective in a relatively simple processor model with small or even without data caches on the processor chip. Some recent studies have shown that this technique can be effective on modern ILP processors as well. Aiming at further investigating the ILP effects and comparing cached DRAM with other advanced DRAM organizations and interleaving techniques, we present a study of the design and optimization of cached DRAM in the context of processors with full ILP capabilities and large data caches. Conducting experiments on execution-driven simulation, we have evaluated its performance effectiveness using eight selected data-intensive SPECfp95 programs and TPC-C workload. Our study provides three new findings: (1) cached DRAM is able to consistently show its performance advantage as the ILP degree increases; (2) contemporary DRAM schemes,

such as SDRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM, do not exploit memory access locality of memory-intensive workloads as effectively as cached DRAM does; and (3) compared with a highly effective permutation-based DRAM interleaving technique, cached DRAM can still gain substantial performance improvement, because its set-associative structure minimizes conflict misses in the on-memory caches.

Following the above study, we proposed a design that exploits the high density of cached DRAM to construct large off-chip L3 caches. Large off-chip L3 caches are beneficial to memory-intensive applications. However, there are two potential problems of using L3 caches: (1) The size of an SRAM cache is limited due to its low density and high cost, thus the working sets of many memory-intensive applications cannot fit into it. (2) Since the L3 tag checking overhead is significant for a cache miss, performance of some memory-intensive applications can be harmed by the existence of L3 cache due to increased memory access latency. To address these two problems, we present a new memory hierarchy organization that uses cached DRAM to construct a large size and low overhead off-chip cache. The high density DRAM in the cached DRAM can hold large data sets, while the small SRAM cache exploits the spatial locality that appears in L2 miss streams to reduce the access latency. A small on-chip tag cache is used to minimize the access overhead. Utilizing a prediction technique, the hit/miss status of an access to the cached DRAM can be accurately predicted, further reducing the access latency.

Memory access latency of application programs has become a major performance bottleneck. As long as DRAM technology remains its most cost-effective position for making main memory, the memory performance problem will continue to exist. The studies conducted in this dissertation attempt to address this important issue. The significance, potential

impacts, and contributions of this dissertation are:

- Our performance studies on memory systems have provided insights into the effects of application program interactions at different levels of the memory hierarchy, and provide motivation and guidance for application programmers to understand memory performance implications, and for system architects to optimize the memory hierarchy and its system organization.
- Our proposed software and hardware schemes are effective and applicable, which can be directly used in the real-world memory system designs and implementations.

### 1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 presents the technical background of this dissertation. Chapter 3 presents the experimental performance evaluation methodology. Chapter 4 describes and evaluates the memory-effective algorithms and implementations of bit-reversals. Chapter 5 presents the permutation-based page interleaving scheme and its evaluation. Chapter 6 presents the evaluation of cached DRAM organization. Chapter 7 describes and evaluates the fine-grain priority access scheduling for multi-channel memory system. Chapter 8 discusses the cached-DRAM cache techniques. Finally, Chapter 9 concludes the dissertation and discusses the future work.

## Chapter 2

# Background of Memory System Performance

An ideal memory is infinitely large and meanwhile allows instantaneous access to every word stored in it. In practice, the access latency of a memory system increases with its capacity. Although the speed of small memories is able to catch up the speed of processor, the speed of large memories is many times slower due to technical limitations. To provide the illusion of an ideal memory system to programmers, architects have built memory hierarchies that contain fast, small memories at the top levels and large, slow memories at the bottom levels. A typical memory hierarchy in today's high performance computers has two-level on-chip caches that are integrated with the CPU, an optional off-chip cache, a DRAM memory system, and hard disks, as shown in Figure 1.1. (In this dissertation, we focus on applications that are memory-intensive but not I/O intensive, thus we do not consider the performance related to disk I/O.)

The effectiveness of the memory hierarchy is based on the principle of program locality. There are two types of locality: temporal locality and spatial locality. Assume a memory word is accessed at a certain time during the execution of a program. The temporal locality

states that the memory word is likely to be accessed again in the near future, and the spatial locality states that the neighbor words are likely to be accessed soon. Thus, the word in use and its neighbor words should be stored at the top levels of the memory hierarchy to reduce the frequency of accessing slow memories. Both types of locality exist in the references to the program code and those to the program data. Most programs have good temporal locality and/or spatial locality so that the use of memory hierarchy can effectively hide the long latency of the slow memories.

The performance of memory hierarchy is highly dependent on program behavior. For programs that have small working sets, most memory references can be satisfied by the on-chip caches. The performance of those programs is then determined by the performance of the on-chip caches, whose access time is close to the processor cycle time. However, most real-world programs have large working sets that cannot be held by the on-chip caches. For those programs, on-chip caches can still filter out most memory references. However, as the performance of the processor increases, the performance loss in the DRAM memories increases dramatically.

The processor model has significant impacts on the design and optimization of caches and DRAM memories. Contemporary superscalar processors have the features of multiple issue, speculative execution, and non-blocking load to exploit ILP aggressively. The use of multiple issue increases the number of instructions that the processor can execute every cycle, thus increases the frequency of cache misses. Speculative execution and non-blocking load allows multiple outstanding cache misses, providing the opportunity of exploiting memory access concurrency.

The performance of a memory hierarchy can be measured by the increase of program

execution time due to memory access latency<sup>1</sup>. The shorter the increase, the better the performance of the memory hierarchy for the program. For simple processors, the performance of a memory hierarchy can be estimated by cache miss rates, cache hit latency, and cache miss latency (See [44] and Chapter 5). For dynamically scheduling processors, however, the insights of memory system performance must be discovered by detailed simulations. The experimental methodology will be discussed in the next chapter.

The rest of the chapter is organized as follows. Section 2.1 introduces the design issues and optimization of caches. Section 2.2 presents the details of DRAM technology. Finally, Section 2.3 discusses the processor model used in this dissertation.

## 2.1 Cache Memories

A hardware cache buffers data contents that are likely to be accessed soon. A typical cache is divided into blocks, each of which stores multiple words that are continuous in the memory space. There are two common methods to map memory space onto the cache: direct mapped and set associative. A direct mapped cache maps a memory block onto a single location in the cache, while a set associative cache maps the block onto multiple locations. The replacement policy decides which block in the cache will be replaced when a cache miss happens. There is only one choice of replacement for direct mapped cache. For set associative cache, the least-recently used (LRU) policy is commonly used.

Cache misses can be divided into three categories: *compulsory*, *capacity*, and *conflict* [44]. Compulsory misses are those misses happening when data items are loaded into the cache

---

<sup>1</sup>When comparing the performance of different configurations for the same program, *CPI* (cycle per instruction) is usually used as a substitute for execution time.



for the first time. Capacity misses are related to the limited capacity of cache. In other words, the capacity misses would not happen if the cache were infinitely large. Conflict misses happen when too many memory blocks are mapped onto some cache sets. Even if the cache is large enough to hold the data items, only a small part of the cache can be effectively used, causing conflict misses.

For simple processors with a single level of cache, the performance of the cache is determined by cache miss rate, cache hit time, and cache miss penalty. The cache miss rate is the ratio of the number of cache misses to the number of memory references. Cache hit time is measured as the number of processor cycles that are needed to fetch data from the cache. Cache miss penalty is the time to fetch and access the data item from the lower level memory. Reducing the miss rate, the hit time, or the miss penalty will improve cache performance. High-performance superscalar processors require high-bandwidth instruction cache, high-bandwidth data cache, and non-blocking cache because of their multiple issue and dynamically scheduling features.

Increasing the cache capacity is an effective way to reduce the number of capacity misses. However, the cache hit time increases logarithmically as the cache capacity increases, which causes an increase of either the pipeline length or the processor cycle time. Thus, the best balance between the speed and the capacity must be determined by careful performance evaluations with realistic workloads [47]. As the transistor budget for building a cache increases, two-level or three-level caches are built to exploit the additional transistors without increasing the hit time of the first-level cache [5, 6, 116, 110]. Furthermore, more caches are integrated into the processor chip, which reduces the cache hit time effectively.

Increasing cache associativity is an effective method to reduce conflict misses. For

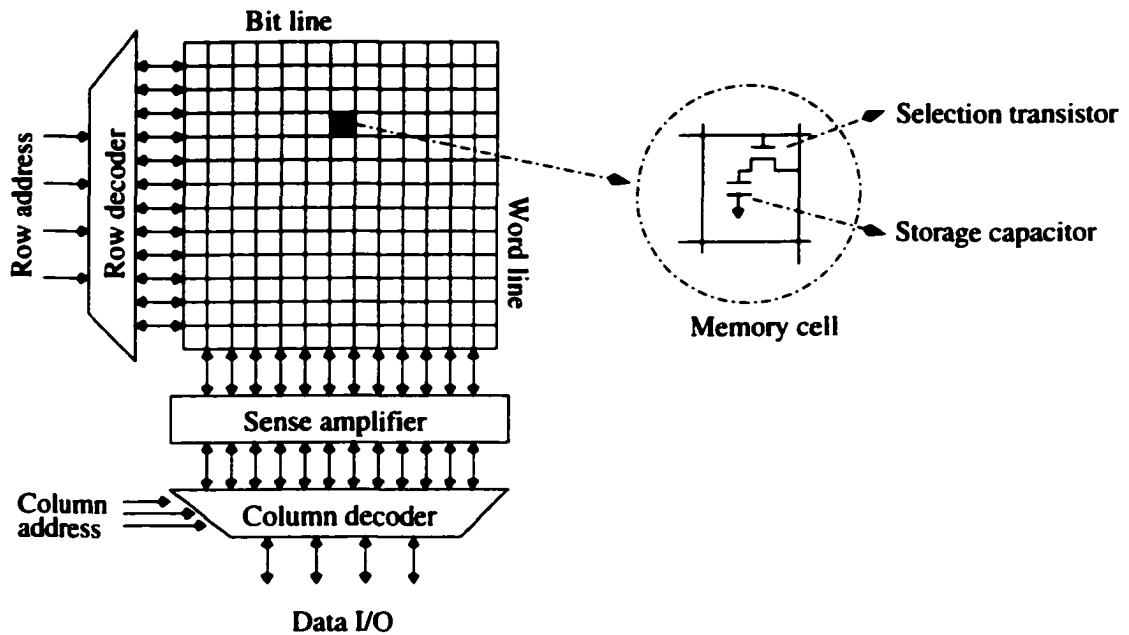
example, authors in [76] report that the miss rate of a direct mapped 16KB cache is about 50% higher than that of a four-way associative counterpart for a commercial workload. However, a set associative cache needs more tag storage, and has a longer hit time because its tag comparison is more complex. To address the long hit time, researchers have proposed a number of *multiple-access cache* techniques [2, 1, 16, 123, 78, 10], which allow fast accesses for most hits while avoiding the high miss rates of a direct mapped cache.

High-performance ILP processors run at a high frequency and issue multiple instructions per cycle, presenting new challenges for cache designs. Architects employ new techniques such as non-blocking cache, high-bandwidth design, and trace cache. With a non-blocking cache [63, 35, 34], the processor can issue other instructions even when a cache miss causes a load instruction to be suspended. MSHR [63] (Missing Status/information Holding Registers) is used to resume the execution of the load instruction when the missed data returns. A high-bandwidth data cache [30, 120, 66, 88, 20] can return more than one data item each cycle so that the processor can execute two or more load instructions simultaneously. Trace cache [92, 52, 36, 12] combines instruction cache with branch prediction unit to meet the demand for high instruction fetch bandwidth.

## 2.2 DRAM Technology

### 2.2.1 DRAM Architecture and Operations

The basic structure of DRAM (Dynamic Random Access Memories) is an array of memory cells, as shown in Figure 2.1. Each cell consists of one transistor and one capacitor. The capacitor, called storage capacitor, stores one binary bit whose value is determined by the



**Figure 2.1:** The structure of a DRAM memory bank.

amount of charge carried by the capacitor. The transistor, called selection transistor, is used to control the input and output of the binary bit. Millions of memory cells form an array, attached to thousands of bit lines and word lines. Such a block, including its peripheral circuits, forms a memory bank. To read a word from a memory bank, the row decoder first activates a word line (a horizontal line in Figure 2.1) according to the row address. The charges carried by the memory cells on the word line propagate on the bit lines (a vertical line in Figure 2.1). The sense amplifier reads the signals on the bit lines and buffers the binary values in its own storage structure. The sense amplifier is also called *row buffer*. Then the column decoder selects a word in the row buffer according to the column address, and outputs it to the external world. This step is called *column access* operation. Reading a row of data to the row buffer is called *row access* operation.

Reading contents from DRAM memory cells is destructive. When the charges of a row of memory cells propagate on the bit lines, the capacitors in those cells lose the charges and the information they represent. Thus, the data stored in the row buffer must be written back into the memory cells after the row access. The process of writing data into memory cells also consists of row access and column access. The row access is the same as that of the read process. During the column access, a word stored in the row buffer is replaced by the input data, then all data stored in the row buffer is written back to the memory cells.

After a column access, a *precharge* operation must be performed to the memory bank before the next row access to the DRAM memory array can happen. The effect of precharge is to raise the voltage of bit lines, which is necessary for the sense amplifier to read signals from the memory cells.

The capacitors of the memory cells lose their charges gradually over time. Thus, a memory bank must be refreshed periodically. The process of *refresh* is just to read the data stored in a row of memory cells to the row buffer, and write the data back to the same memory cells. This process is done row by row.

The DRAM memory cell structure, which has only one selector transistor and one storage capacitor, is possibly the most compact structure allowed by current semiconductor technology. Because of the compactness of its memory cells and its array organization, DRAM memory has very high density and is relatively inexpensive. Unfortunately, those properties also limit the improvement of DRAM access latency. For contemporary DRAM memories, the precharge time is 20~30ns, and the row access time and the column access time are around 20ns. For more details and discussions of DRAM memory technology, interested readers may refer to [81, 59, 27].

### 2.2.2 Comparison of DRAM and SRAM

Compared with SRAMs (Static Random Access Memories), DRAMs have four major technical limitations. First, the simple cell structure with one capacitor and one transistor makes the row access latency longer than that of SRAMs, which use multiple transistors to facilitate a cell. Second, performing “read” operations on the DRAM cells is destructive to the original signals. The signals have to be written back to the selected memory cells. In contrast, the signals in SRAM cells are restored by themselves after read operations. Third, each DRAM cell must be *refreshed* periodically to charge the capacitor. In contrast, SRAMs hold their data bits using flip-flop gate circuits, where the memory contents are retained as long as the power is on. Finally, the DRAM memory array must be *precharged* for the next memory access.

SRAMs are fast, but expensive due to their low density. DRAMs are relatively slow, but have high density and low cost because of their one-transistor cell structure and other circuit characteristics. DRAMs have been widely used to construct the main memory for most computer systems. The only exception has been for some vector computer systems where expensive SRAMs are used for the main memory. This is because high speed is the first consideration for those vector computers. All contemporary workstations, multiprocessor servers, and PCs use DRAMs to build the main memory modules, and only use SRAMs to construct caches.

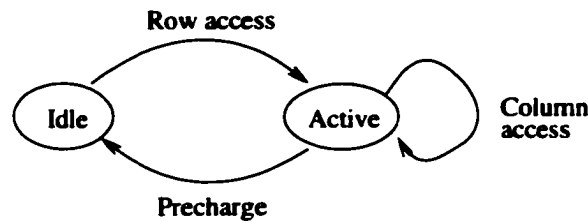
### 2.2.3 DRAM Access Request and DRAM Operations

As discussed earlier, an access to a DRAM bank may involve three types of operations: precharge, row access, and column access. Those operations are not performed in a fixed

order, and some of them may not be necessary. The sequence of the operations to satisfy an access request depends on the *bank state* and the address to be accessed. From the viewpoint of scheduling, a memory bank can be in two states: active or idle [90]. When a bank is active, its row buffer contains valid data. When a bank is idle, it is precharged but the data in its row buffer is lost. Figure 2.2 shows how the bank state transits. We summarize the relationship between the operations and the bank states as following:

- Precharge can be performed when the bank state is active. After a precharge, the data in the row buffer is lost, but the next row access can be performed. The bank state will be idle after the operation.
- Row access can be performed when the bank state is idle. After a row access, a row of data is read from the memory cell array to the row buffer. The bank state will be active after the operation.
- Column access can be performed when the bank state is active. For a read request, a block of data is selected and read from the row buffer. For a write request, a block of data is selected and written to the row buffer. Only the data existing in the row buffer can be read or written. The bank keeps in the active state after the operation.

The sequence of operations that are necessary for an access request is dependent on the bank state and the data address. If the bank is active and the data to be accessed is already in the row buffer, only a column access is necessary. This type of access has the shortest latency. If the bank is active but the data to be accessed is not in the row buffer, then the sequence of operations is precharge, row access, and column access. This type of access has the longest latency. If the bank is idle, the sequence is always row access and column



**Figure 2.2:** Memory bank states.

access. Notice that precharge does not need a data address so it can be performed before a request arrives, and column access is always the last step for an access.

#### 2.2.4 DRAM Variants

There are a number of DRAM variants developed to improve DRAM latency and data transfer rate. Recent commercial examples include Synchronous DRAM (SDRAM), Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM.

**Synchronous DRAM (SDRAM):** The data access operations of an SDRAM are synchronized with the processor by an external clock. The SDRAM technology improves upon the efficiency and the data transfer rate of DRAMs in three ways. First, SDRAM supports *burst mode data access* which reads or writes continuously allocated data blocks in the same row sequentially without idle intervals. The burst data access is defined by the starting address and the data length. Some asynchronous DRAMs also support burst mode data accesses, but require the memory controller to provide the column address for each data block. The burst mode in SDRAM significantly improves the data transfer rate. Second, the synchronous clock coordinates the processor and the main memory better, reducing processor idle time. Finally, SDRAMs

normally have two or four independent data banks, providing opportunity to overlap concurrent data accesses.

**Enhanced SDRAM (ESDRAM):** A small SRAM cache is integrated into the row buffer of each SDRAM memory bank. If an access is a hit in the buffer, the access time is equivalent to that of accessing the fast SRAM cache. The other advantage of ESDRAM is its ability of overlapping memory precharging and refreshing operations with cache accesses.

**Rambus DRAM (RDRAM):** A special high speed but narrow (one-byte wide) bus is designed to bridge between the processor and multiple memory banks. This bus is multiplexed for transferring address/command and data. Both edges of the bus clock signal are used for data transfer to double the data transfer rate. The memory banks in Rambus DRAM can be independently accessed, precharged, or refreshed to make accesses to different banks in a pipelining mode. Currently, Rambus DRAMs support eight or sixteen banks.

**Direct Rambus DRAM (DRDRAM):** This is an advanced version of RDRAM which provides a one-byte wide address bus and a two-byte wide data bus to connect a large number of memory banks (16 or 32). The number of row buffers is roughly an half of the number of memory banks, which makes each pair of adjacent banks share a row buffer. The buffer sharing reduces the hardware cost.

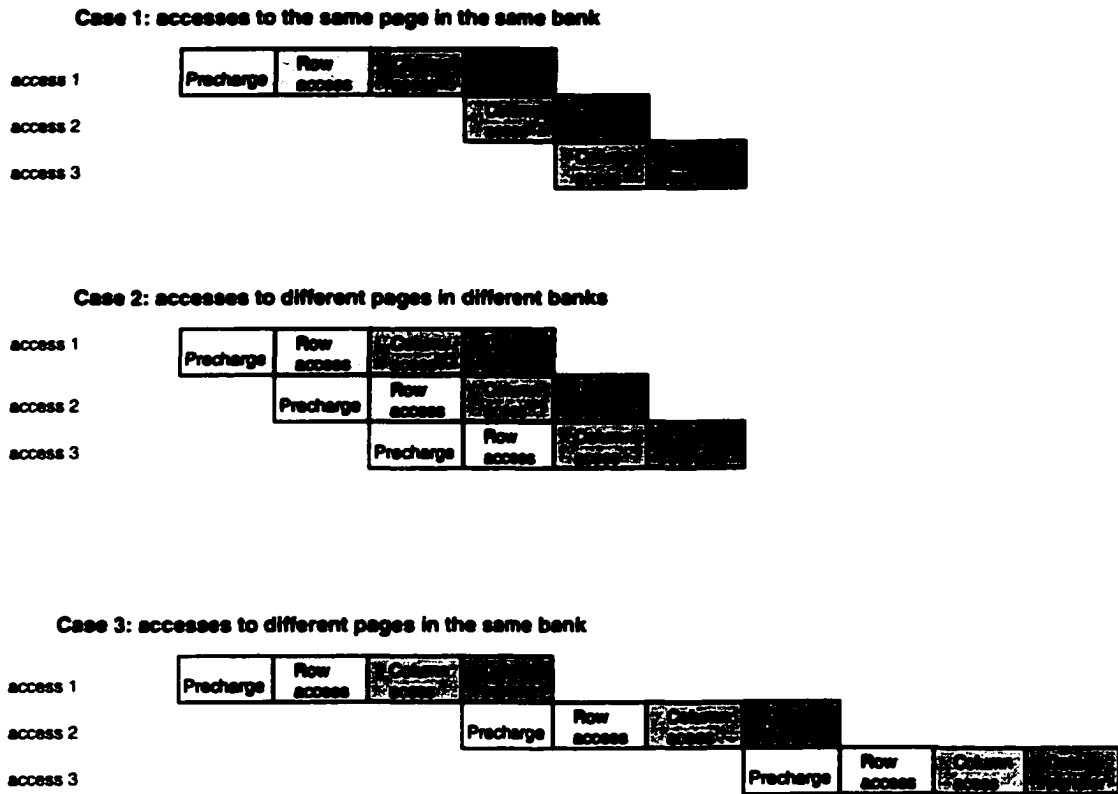


### 2.2.5 Memory Access Scheduling

Today most DRAMs have multiple independent banks, and each bank can be operated independently. For example, one memory bank can be read when another memory bank is in precharge. Thus, contemporary DRAM memory systems can serve multiple accesses concurrently. Scheduling those concurrent requests can make a big difference in performance. Memory access scheduling can reduce access latency and improve bandwidth utilization by re-arranging the order and issue time of DRAM operations for a group of concurrent requests [75, 71, 48, 89, 90, 70].

Concurrent memory accesses can be classified into one of the following three categories:

1. *Accesses to the same page in the same bank.* These accesses not only fully exploit the spatial data locality, but the corresponding operations can also be well pipelined. Case 1 in Figure 2.3 shows the pipelined execution of three reads to the same page in the same bank. Writes to the same page in the same bank can be well overlapped in the same way. In both cases, the data bus is fully utilized during pipelined execution.
2. *Accesses to different pages in different banks.* Since the accesses can be done in parallel, the operations can also be well pipelined. Case 2 in Figure 2.3 shows the pipelined execution of three reads to different pages of different banks. Writes to different pages in different banks can be well pipelined in the same way. In both cases, the data bus is fully utilized during pipelined execution. Reads and writes can also be overlapped, except there is a small bus turn-around gap between a write and a following read.
3. *Accesses to different pages in the same bank.* These accesses not only have little data



**Figure 2.3:** Pipelined executions of three types of memory accesses. (This figure only shows the pipelining operations at a conceptual level.)

locality to exploit, the corresponding operations are difficult to pipeline as well. Case 3 in Figure 2.3 shows the pipelined execution of three reads to three different pages in the same bank. Precharge and row access are needed to initiate each access. The precharge of an access can be overlapped with the data transfer of another access. However, the DRAM row access of a request cannot start until the access ahead of it finishes. For each access in the pipelined execution, the idle time of the data bus can be the sum of a row access and a column. Similarly, write accesses to different pages in the same bank can not be effectively pipelined.

The more independent banks in a memory system, the larger the potential that page conflicts can be avoided (but a carefully designed memory address mapping is a necessity). Normally a memory system consists of a number of DRAM chips. Each DRAM chip has four or eight physically independent memory banks with current technique. The total number of independent banks depends on the organization of memory system, the number of chips, and the number of physical banks in a chip. Rambus DRAM systems tend to have more independent banks than a conventional SDRAM systems. For example, a Rambus DRAM system with 8 chips of 4 physical banks each have 32 independent banks. In comparison, a SDRAM system with 128-bit bus, 8 chips with 16-bit data output and 4 physical banks have only 4 banks; this is because the 8 chips must be operate together to supply 128-bit data to the bus.

Another scheduling issue is to decide the time to precharge a bank when it has no pending requests. There are two strategies: *close page* and *open page*. The close page strategy begins the precharge immediately after the current column access finishes. The next access to the bank will require a row activation and a column access. In contrast, the open page strategy delays the precharge, hoping that the next access is a row buffer hit, thus only the column access is needed. However, if the next access is a row buffer miss, all three operations will be required. Which strategy wins depends strongly on the page hit rate. If the hit rate is high, then the open page strategy is likely to reduce the memory stall time<sup>2</sup>. The row buffers are like a cache with a small number of blocks with very large cache size. It is well known that the performance of such a cache structure is comparable

---

<sup>2</sup>The memory stall time is not decided only by the hit rate, especially for dynamically scheduled processors. In the general trend, however, the memory stall time decreases as the hit rate increases.

to large caches with small block size. There is a great potential to exploit this cache.

Recently, multi-channel memory systems have been used with high performance processors that require high bandwidth DRAM memories. Each channel can be scheduled independently. Direct Rambus DRAM is such a representative memory system. A Direct Rambus DRAM system generally consists of multiple channels, where each channel provides 1.6 GB/s bandwidth. Each channel has its own row control bus, column control bus, and two-byte wide data bus. The separation of row and column control buses eliminates the contention in the address bus between row operations (precharges and row activations) and column accesses. The bus clock rate is 400 MHz and the data is transferred on both edges of the clock. The row and column addresses/commands and the data are transferred in packets, each taking four bus cycles. The minimal data packet length is 16 bytes. Each channel can connect multiple devices (chips). Each device can have 32 banks and 33 half-page row buffers (this may vary according to the configuration). Those banks may be operated independently, which provides high concurrency at the bank level. The Intel Pentium 4 processor supports two channels, and the Compaq Alpha 21364 processor supports up to eight channels.

## **2.3 Dynamically Scheduled Superscalar Processors**

The processor model has significant impact on the design and optimization of memory systems. The work in this dissertation is done in the context of dynamically scheduled superscalar processors with the following features:

- **Dynamically scheduled** [3]: The processor can *issue* (start the execution of) instructions out of program order. An instruction can be issued as long as previous instructions that provide its inputs have finished, and the hardware resources for the execution are available.
- **Multiple issue**: Multiple instructions can be issued every processor cycle. If the processor is capable of issuing  $n$  instructions per cycle, it is called an  $n$ -way processor. Most contemporary processors are from 2-way to 8-way. The actual number of instruction issued per cycle may be less than  $n$ .
- **Speculative execution**: The processor may issue instructions speculatively to exploit more instruction-level parallelism before their dependencies are fully examined. In case the speculation is wrong, the execution of the related instructions is nullified.
- **Non-blocking load**: The processor may not stall when a cache miss happens upon the execution of a load instruction. The execution of the load instruction must be suspended, but the processor can issue other instructions that are not dependent on the load instruction.

One would think that dynamic scheduling combined with a non-blocking load can eliminate the processor stall due to memory accesses. In fact, the processor can tolerate only the latency of L2 cache accesses. For a 2GHz processor, the latency of DRAM accesses is around one hundred processor cycles, while the processor can tolerate a delay of about ten cycles.

On the other hand, superscalar processors can reduce memory stall time from that of simple processors as they increase the concurrency of DRAM accesses. A processor can issue

dozens of instructions before it stalls for a cache miss, and is likely to find more cache misses during this period. This phenomena, called miss clustering, is widespread in the execution of realistic benchmark programs. Then, with proper hardware support, the processor can make multiple DRAM access requests to the DRAM memory system. An advanced DRAM memory system can serve those requests in parallel, reducing the average latency of DRAM accesses. Such an example has been shown in Figure 2.3.

## Chapter 3

# Evaluation Methodology: An Experimental Approach

We have extensively used *architectural simulation* and standard benchmarks to evaluate our software methods and hardware designs. We conduct almost all experiments using SimpleScalar [15], a set of simulation tools designed for program performance analysis and detailed microarchitectural modeling. To evaluate our hardware designs, we have incorporated new software modules in the simulator to simulate those hardware components. The only exception is the study in Chapter 4, in which we use direct measurements and SimOS [91] simulator to do performance evaluation. We have run SPEC95 [108] and SPEC2000 [45] benchmark programs and TPC-C [112] workload in the simulation environment to evaluate the performance of our designs.

This rest of this chapter is organized as follows. Section 3.1 discusses the background of architectural simulations. Section 3.2 introduces the SimpleScalar tool set. Section 3.3 describes our extension to SimpleScalar. Finally, Section 3.4 introduces SPEC95 and SPEC2000 benchmark programs and TPC-C workload.

### 3.1 Architectural Simulations

Simulation is currently the predominant method for performance evaluation in computer architecture research, complemented by direct measurement and analytical modeling. For example, among 27 papers published in proceedings of the 29th International Symposium on Computer Architecture (ISCA-29, 2002), 24 papers use simulators as the major tool of performance evaluation. The authors come from both academia and computer industries such as Intel, IBM, HP, and Sun. (In fact, those companies have their own teams developing high-fidelity simulators to validate the design of their products.)

The use of simulation becomes popular as architects and researchers study advanced techniques to exploit instruction-level parallelism. Using analytical modeling to evaluate those techniques is extremely difficult, if not infeasible. The other alternative, direct measurement, is not available until a prototype can be made. Thus, simulation has become the only realistic choice in most cases. After years of development, today's architectural simulation has the following merits:

- **Simulation of native ISA (Instruction Set Architecture):** The simulators can emulate the execution of native binary codes (compiled on a real machines) without any alternation, and produce exactly the same results as on the real machine.
- **Cycle-by-cycle accuracy:** Detailed simulators model the status of processor components, including registers, instruction fetch with branch prediction, scheduling logic, functional units, L1/L2 caches, and memory. The status changes of those components are accurate cycle by cycle.



- **Modular design:** A good simulation program consists of well designed modules. Architects and researchers, as users of the simulation program, can easily change existing modules or incorporate new modules to test and validate their ideas.

The three most popular collections of architectural simulation software are:

- **SimpleScalar [15]:** SimpleScalar is a tool set of simulation that contains several simulators providing different simulation detail levels. It was originally developed in the Computer Sciences Department at the University of Wisconsin-Madison. Now SimpleScalar can emulate four ISAs: Alpha, PISA, ARM, and x86. SimpleScalar is widely used in academia and industry and for both research and instruction. Section 3.2 describes SimpleScalar in more detail.
- **SimOS [91]:** SimOS is developed in the Department of Computer Science at Stanford University. It is a complete machine simulation environment for both uniprocessors, shared-memory multiprocessors, and a network of uniprocessors or multiprocessors. It models the MIPS R4000 and R10000 and allows the IRIX 5.3 OS to run on the top of the simulator. An extension of SimOS models the Digital Alpha processor families and supports Digital Unix OS.
- **RSIM [50]:** RSIM (Rice Simulator for ILP Multiprocessor) is an architectural simulator for shared-memory multiprocessors based on dynamically scheduled processors. It was developed at the Department of Electrical and Computer Engineering at Rice University.

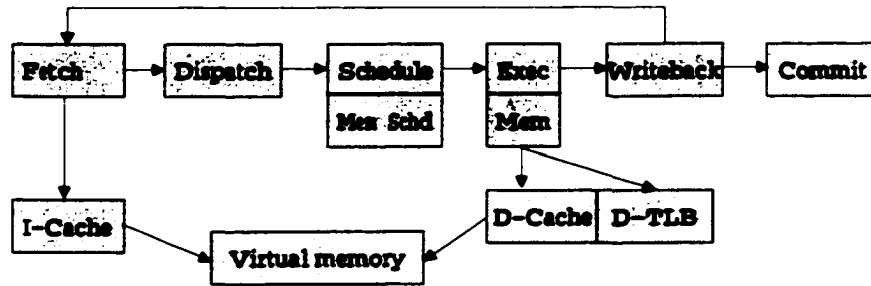
Developers of those software tools have made the source code open and granted licenses to academic users free of charge. Nowadays, those tools have been widely used as simulation

infrastructures on which researchers are able to make their own extensions. Proprietary simulation software has also been developed in industry, but those programs are normally not available for academic users.

Although architectural simulation is powerful, it has several limitations. First, the speed of high-fidelity simulation is much lower than that of real machines. The simulation slowdown, i.e., the time to finish the execution of a program on a simulator vs. that on a real machine, ranges from 3000 to 10000 for uniprocessor simulation. This fact limits the variety of workloads and the number of configurations that can be investigated, and incurs a high computation cost for running simulations. Second, the development and maintenance of high-performance simulation infrastructures have become very expensive, requiring extensive human and equipment support. Finally, validating simulation is a daunting task. Because of its complexity, identifying errors in simulation software is extremely difficult. Simulation developers did make efforts to validate simulation infrastructures against real machines [39, 28]. However, the results were not available until years after those simulation tools had been extensively used.

## 3.2 SimpleScalar Tool Set

The SimpleScalar tool set [15] is a suite of processor simulation tools containing both detailed and high-performance simulations. Ported to multiple instruction sets including Alpha, PISA (Portable Instruction Set Architecture), ARM, and x86. SimpleScalar has provided a powerful simulation infrastructure for both academic and industry users. Many extensions have been made to SimpleScalar, including value prediction, trace cache, multipro-



**Figure 3.1:** Processor pipeline in SimpleScalar. (from [15], reproduced with permission.)

cessor, and multithreaded processor. According to news on <http://www.simplescalar.com/>, the SimpleScalar web site, SimpleScalar is used by more than two out of five papers published in the 34th International Symposium on Microarchitecture (2001), more than one third of the papers in the 8th International Symposium on High-Performance Computer Architecture (2002), and more than one half of the papers in the 29th Annual International Symposium on Computer Architecture (2002).

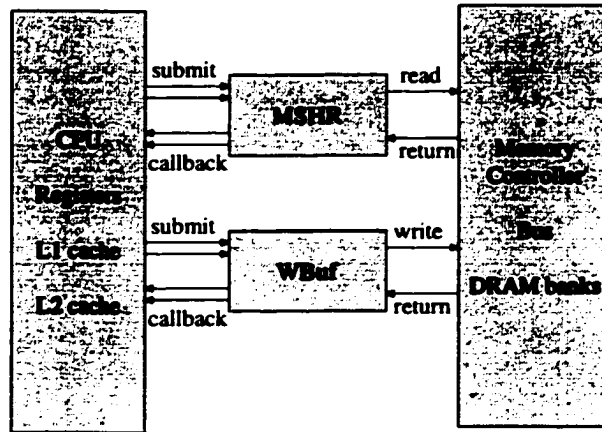
We have used the *sim-outorder* program in the SimpleScalar tool set to obtain most performance results reported in this dissertation. Sim-outorder simulates the full detail of a dynamically scheduling processor. It is a cycle-driven simulation: the simulator tracks the change of the processor status every cycle, including of pipeline, register, load/store queue, cache, TLB, and so on. The pipeline of the modeled processor has five stages: fetch, dispatch, exec/mem, writeback, and commit, as shown in Figure 3.1.

### 3.3 Our SimpleScalar Extensions

We have made a set of extensions to *sim-outorder* to model the main memory system in detail. The original *sim-outorder* implementation models a “perfect” main memory systems, of which the memory access latency is a constant for a cache fill request. Our extensions model the full detail of DRAM memory systems, including bus contention, bus synchronization, bank status, row access, column access, precharge, and refresh. We also model a writeback buffer and an MSHR unit.

Figure 3.2 shows the overall structure of the extensions we have made. First, we modified the interface of the cache module in *sim-outorder*. The original cache module returns a fixed integer value of the access latency for both cache hit and miss. When a cache miss happens, the new cache module returns a negative value when it detects a cache miss. This change allows the miss latency be determined after the detection of a cache miss. The new module submits cache-related information to the MSHR module, and the processor module submits processor-related information to the MSHR module. The MSHR module records both sets of information and sends a message to the DRAM module. The DRAM module simulates the DRAM operations, determine the finish time, and send a message to the processor to inform it that the DRAM access has finished. Then the MSHR module find out the related miss information and make a requests to perform cache fill and to wake up the waiting instruction.

The interactions between the modules are based a message passing functionality that we added into SimpleScalar. With this functionality, a simulation module can send/receive messages to/from other modules. When sending a message, a module provides the des-



**Figure 3.2:** Diagram of our extension of DRAM memory simulation.

mination module, the message type, the time when the message should be delivered, and a flexible set of parameters. Upon receiving a message, the *message-driven* code of the destination module parses the message and processes accordingly.

The original processor module in SimpleScalar is written in *cycle-driven* style. There is a variable `sim_cycle` that records the current clock cycle. Each cycle the activities of the processor pipeline stages are simulated, then the clock cycle (`sim_cycle`) advances. We add message-driven code into the processor module so that it can process messages from the DRAM module. Messages are delivered at the beginning of each cycle.

The DRAM module encapsulates the performance simulation of the memory controller and the DRAM memory. It maintains request queues, the bus/channel status, and the bank status. When a DRAM request arrives as a message sent by the processor module, the DRAM module determines which bank should be used according to the DRAM mapping

scheme, then buffers the request into the request queues. The DRAM module checks the memory bus/channel status cycle by cycle. When the memory bus/channel is idle, a request is picked up from the queues if (1) the memory bank to access is idle, and (2) the request has the highest rank according to the scheduling policy. Then the DRAM module determines which operation should be issued and when the operation will finish. If the operation is a column access, the last step of a DRAM access, a message is sent back to the processor module to inform it that the DRAM access finishes.

The DRAM module contains both message-driven code and cycle-driven code. The message-driven code accepts DRAM requests as messages from the processor module, then inserts the DRAM requests into its request queue. At the beginning of each simulation cycle, the cycle-driven code checks the availability of memory bus/channel and the memory bank status, determining if a DRAM operation can be started. When the code determines that a column access operation will finish, it sends a message back to the processor module. The cycle-driven code also sends messages to the DRAM module itself to inform it the finish times of DRAM operations. When those messages are delivered, the message-driven code updates the status of the bus and memory banks accordingly. In summary, the message-driven code processes external DRAM requests and internal notifications of status changes, while the cycle-driven code schedules DRAM operations and determines the time of internal status changes and when DRAM accesses finish.

### 3.4 SPEC and TPC-C Benchmarks

We have used SPEC CPU95 [108] benchmark, SPEC CPU2000 [45] benchmark, and TPC-C [112] workload in our performance evaluations.

#### 3.4.1 SPEC CPU95 Benchmark

SPEC CPU95 (briefly, SPEC95) benchmark had been extensively used to study the performance of processors, memory systems, and compilers until it had retired with the release of SPEC CPU2000. As a representative mixture of compute-intensive applications, SPEC95 consists of two sets of benchmarks: CINT95 for compute-intensive integer performance and CFP95 for compute-intensive floating point performance. CINT95 contains eight applications written in C, and CFP95 contains 10 applications written in FORTRAN. Table 3.1 gives short descriptions of SPEC95 programs (from the SEPC's CPU95 Press Q&A [107]).

#### 3.4.2 SPEC CPU2000 Benchmark

The SPEC [107] organization states that SPEC CPU2000 (briefly, SPEC2000) “is the next-generation industry-standardized CPU-intensive benchmark suite”. It is provided as “a comparative measure of compute intensive performance across the widest practical range of hardware”. Like SPEC95, SPEC2000 consists of two sets of applications, CINT2000 for compute-intensive integer performance and CFP2000 for compute-intensive floating point performance. CINT2000 contains eleven applications written in C and one in C++ (252.eon), and CFP2000 contains fourteen applications (six in Fortran-77, four in Fortran-90, and four in C). Table 3.2 gives short descriptions of the SPEC2000 programs [107].

**CINT95**

099.go	Artificial intelligence; plays the game of "Go"
124.m88ksim	Moto 88K Chip simulator; runs test program
126.gcc	New version of GCC; builds SPARC code
129.compress	Compresses and decompresses file in memory
130.li	LISP interpreter
132.jpeg	Graphic compression and decompression
134.perl	Manipulates strings (anagrams) and prime numbers in Perl
147.vortex	A database program

**CFP95**

101.tomcatv	A mesh-generation program
102.swim	Shallow water model with 513 x 513 grid
103.su2cor	Quantum physics; Monte Carlo simulation
104.hydro2d	Astrophysics; Hydrodynamical Navier Stokes equations
107.mgrid	Multi-grid solver in 3D potential field
110.applu	Parabolic/elliptic partial differential equations
125.turb3d	Simulates isotropic, homogeneous turbulence in a cube
141.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants
145.fpppp	Quantum chemistry
146.wave5	Plasma physics; Electromagnetic particle simulation

**Table 3.1:** SPEC95 programs.

Compared with SPEC95, a significant change in SPEC2000 is that the applications in SPEC2000 have larger working sets, resulting in higher cache miss rates and more significant memory stall times. For instance, in one of our experiments with a 1MB L2 cache configuration, seven of the eleven CINT2000 applications have large portions of memory stall times in their total execution times, while all the eight CINT95 programs have negligible memory stall time. This change reflects the fact that real-world memory-intensive applications are putting high pressure on memory systems.



**CINT2000**

<b>Name</b>	<b>Remarks</b>
164.gzip	Data compression utility
175.vpr	FPGA circuit placement and routing
176.gcc	C compiler
181.mcf	Minimum cost network flow solver
186.crafty	Chess program
197.parser	Natural language processing
252.eon	Ray tracing
253.perlbnk	Perl
254.gap	Computational group theory
255.vortex	Object Oriented Database
256.bzip2	Data compression utility
300.twolf	Place and route simulator

**CFP2000**

<b>Name</b>	<b>Remarks</b>
168.wupwise	Quantum chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver in 3D potential field
173.applu	Parabolic/elliptic partial differential equations
177.mesa	3D Graphics library
178.galgel	Fluid dynamics: analysis of oscillatory instability
179.art	Neural network simulation; adaptive resonance theory
183.equake	Finite element simulation; earthquake modeling
187.facerec	Computer vision: recognizes faces
188.amp	Computational chemistry
189.lucas	Number theory: primality testing
191.fma3d	Finite element crash simulation
200.sixtrack	Particle accelerator model
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants

**Table 3.2:** SPEC2000 Programs.

### 3.4.3 TPC-C Benchmark

TPC benchmarks represent commercial workloads, which are widely used by computer manufacturers and database providers to test, evaluate, and demonstrate the performance of their products. TPC-C, a part of the TPC benchmarks, is an on-line transaction processing (OLTP) benchmark. It is a mixture of read-only and update-intensive transactions that simulate a complex computing environment in which a population of terminal operators execute transactions against a database. The database system we have used to support the TPC-C workload is the PostgreSQL (version 6.5) [79]. This is the most advanced open source database system available for basic research.

There are five transaction types in TPC-C: new order, payment, order status, delivery, and stock-level transactions. The transactions are generated by emulated users. Each emulated user selects a transaction type, enters the required number of input fields, waits for the output, and idles for a defined thinking time before submitting another request to the system. The scale of the database is defined by the number of warehouses in the whole system.

In our experiments, we set up ten warehouses, the maximum size of database supported on the machine on which we run the simulation. We configure 320 MBytes of database memory buffer which is also the maximum size supported on the machine. The memory buffer size in practice is 1 GBytes to 2 GBytes.

## Chapter 4

# Fast Bit-Reversals on Uniprocessors and Shared-Memory Multiprocessors

Memory performance can be significantly improved by reconstructing programs based on program-specific knowledge. This chapter focuses on cache-optimal methods of implementing *bit-reversals*. Bit-reversals are often repeatedly used as fundamental subroutines in scientific programs, such as FFT. If the bit-reversal operations are not implemented properly, those FFT operations can slow down significantly. On the other hand, it is easy to improperly implement bit-reversals on uniprocessors and multiprocessors, because the performance of bit-reversals is highly sensitive to the ways the memory hierarchies are exploited. In other words, a fast bit-reversal implementation must be cache effective. Several papers have well addressed the significance and effects of considering memory hierarchies to bit-reversals (e.g., [7, 58, 109]). Besides the important usage for FFT, different versions of bit-reversal implementations can also be used as benchmark programs to evaluate the

memory hierarchy of various computer systems.

Performance degradation of bit-reversals is mainly caused by cache conflict misses. Although compiler optimizations are effective in reducing cache conflicts for programs with regular access patterns, they are not effective for bit-reversals because of their special access pattern. Thus, in order to gain the best performance, cache-optimal methods and their implementations should be carefully and precisely done at the programming level for those special programs.

A standard bit-reversal program is described as follows:

```
for i = 1, N
    Y[i'] = X[i]
```

The values of array  $X$  in their sequential positions  $i$  are copied to array  $Y$  in their bit-reversal positions,  $i'$ , for  $i = 1, \dots, N$ , where  $N = 2^n$ . The above program says that  $X$  is a bit-reversal reordering of  $Y$ . The indices of  $i$  and  $i'$  of  $X$  and  $Y$  are represented by a sequence of  $n$  binary digits. Positions  $i$  and its bit-reversal  $i'$  are defined in [58] as:

$$i = \sum_{j=0}^{n-1} a_j 2^j \quad \text{and} \quad i' = \sum_{j=0}^{n-1} a_j 2^{n-1-j},$$

where  $a_j$  is either 0 or 1. For example, a 5-bit reversal of  $i = 10010$  is  $i' = 01001$ .

The bit-reversal operations have following unique characteristics: First, in many implementations, each element in an array is only used (read or written) once for its copy operation. Thus, the reorderings have only spatial locality but no temporal locality for elements. Second, the loops follow certain sequences with high spatial locality. Bit-reversals are highly sensitive to problem sizes, cache sizes, and cache line sizes. Since the data array

sizes are a power of two, multiple elements stored in different memory locations could map to the same cache line, causing severe cache conflict misses and cache thrashing. The reason is simple. Most commercial computers use direct-mapped or  $n$ -way associative caches where the mapping functions of cache sizes are also related to powers of two.

We use an identical unit, called an “element”, to represent the sizes of data arrays, caches and others such as buffers and blocking. One element may represent a 4-byte integer, a 4-byte floating point number, or an 8-byte double floating point number. Because the sizes of caches and cache lines are always a multiple of an element in practice, this identical unit for all the sizes is practically meaningful for both architects and application programmers, and makes the discussions straightforward. Here are the algorithmic and architectural parameters we will use to describe cache-optimal methods of bit-reversals:

- $C$ : data cache size, which could be further defined as  $C_{L1}$  and  $C_{L2}$  for data cache sizes of L1 and L2 respectively.
- $L$ : the size of a cache line, which could be further defined as  $L_{L1}$  and  $L_{L2}$  for cache lines of L1 and L2 respectively.
- $K$ : cache associativity, which could be further defined as  $K_{L1}$  and  $K_{L2}$  for cache associativity of L1 and L2 respectively.
- $K_{TLB}$ : TLB cache associativity. (A TLB cache is a small buffer that holds the most recent memory page mappings. The concept will be discussed in detail later in the chapter.)
- $T_s$ : number of entries in the TLB cache.

- $N$ : the data size for the bit-reversal vector of size  $N = 2^n$ , where  $n$  is the number bits used in the vector index.
- $B_{cache}$ : blocking size of a  $B \times B$  submatrix for cache.
- $B_{TLB}$ : blocking size for TLB.
- $P_s$ : a memory page size.

In this chapter, we examine different methods for efficient implementations of bit-reversals using techniques of blocking, buffering, and padding. We evaluate the merits and limits of each technique and their application and architecture-dependent conditions for developing cache-optimal methods. Although our methods are developed for out-of-place bit-reversals, they are also applicable to in-place bit-reversals, where  $X$  and  $Y$  are the same array.

Symmetric Multiprocessor (SMP) systems have become practical and cost-effective servers for scientific computing and other applications. Although parallel efficiency and communication latency reduction are major performance concerns, computations on an SMP share many common considerations with uniprocessors. The most important one is the effective usage of memory hierarchies. When the cache locality of each processor is effectively exploited, the memory accesses to the shared-memory will be reduced, and so will be the memory access contention. People have studied parallel data reordering algorithms on distributed-memory systems with special networks, such as hypercubes (see e.g. [29] and [54]). In this study, we target parallel bit-reversals on SMPs and show the significant impact of the cache and TLB considerations for efficient method development

and implementations. We also evaluate the performance impact of SMP interconnection networks.

Our algorithm designs and implementations are optimized by considering several non-traditional, but practical and performance effective factors, namely, programming complexity, memory space requirement, instruction count, cross interference among the data arrays, and program portability. We will summarize the limits and merits of different bit-reversal methods based on these considerations after we have discussed the designs and presented the performance results, aiming at providing a guideline for performance programming and memory performance optimization for other scientific computing applications.

We present two contributions in this study: (1) Our integrated blocking methods, which match cache associativity and TLB cache size and fully use the available registers, are cache-optimal and fast. (2) We show that our padding methods outperform other software oriented methods, and believe they are the fastest in terms of minimizing both CPU and memory access cycles. Since the padding methods are almost independent of hardware, they could be widely used on many uniprocessor workstations and SMP multiprocessors. Using direct measurement on five different platforms including PC, workstations, and servers, we will show that our methods consistently outperform existing ones.

The rest of this chapter is organized as follows. Section 4.1 discusses the inherently blocking nature of bit-reverse operations and the effectiveness and limits of blocking techniques for solving the problems. Section 4.2 evaluates a software buffering technique and our methods using existing hardware components for implementing the data reordering. Section 4.3 presents our new method integrating blocking and padding. Section 4.4 discusses blocking and padding techniques for the TLB. Sections 4.5 and 4.6 report the experimental

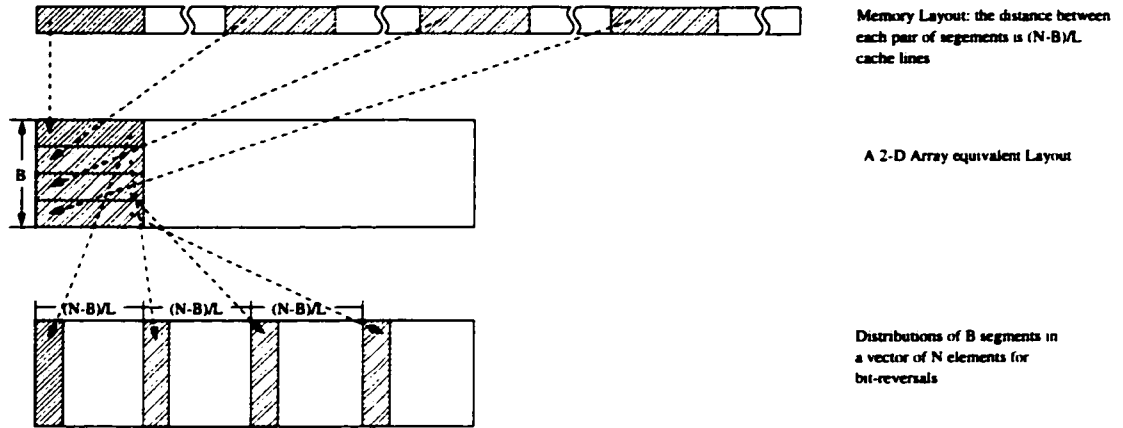


Figure 4.1: Memory layout of a blocked bit-reversals, where  $B = B_{cache}$ .

measurements and analyses for evaluating different methods on uniprocessor workstations and SMP multiprocessors, respectively. Finally, Section 4.7 summarizes this study.

## 4.1 Blocking for Bit-reversals

The blocked memory access patterns of bit-reversals can be easily viewed when we convert the one dimensional vector to a 2-D equivalent array in Figure 4.1. All the reordering elements and elements in other groups will be allocated along the column in the 2-D equivalent array forming a block.

In this blocking method, the bit-reversal reordering is performed block by block, where the operations for each block are implemented similarly to the Evans' method [33] (the Evans' method is used to construct a hybrid method in [58]). The program in the Appendix presents such an implementation along with padding technique. (The padding technique will be discussed in Section 4.) The blocking algorithm we have used can be classified as a hybrid method.

In general, for a bit-reversal vector of  $N = 2^n$  elements, the block size  $B_{cache}$  is a power



of 2, denoted by  $B_{cache} = 2^b$ . Each of the  $B_{cache}$  elements in  $X$  has the address format of  $fg$ , where  $g$  is  $B_{cache}$  bits, and  $f$  has  $n - b$  bits. Each of the corresponding  $B_{cache}$  elements in  $Y$  has the address format of  $g'f'$ . Therefore, the distance between two nearest elements in the same group in  $Y$  is  $2^{n-b} = N/B_{cache}$ .

Choosing the cache line size as the minimum blocking size ( $B_{cache} = L$ ), we can easily calculate the maximum  $N$ s for the bit-reversal vector based on different data cache sizes. For example, for a large cache of 2 MBytes, the blocking technique is effective up to an 18-bit-reversal reordering which represents 268,144 data elements, where each element is an 8-byte double type, and the cache line is 32 bytes. In practice, the data size of bit-reversals could easily be larger than  $n = 20$  [58].

## 4.2 Blocking with Buffers

As we have shown, the effectiveness of blocking is limited by the size of the data arrays. In theory, the smallest blocking size could be  $2 \times 2$ . A cache line in a modern processor usually holds more than 2 elements, i.e., is larger than 16 bytes. If we choose a  $2 \times 2$  block, the data in a cache line will not be fully used before its replacement, causing more cache misses in the reorderings. The bit-reversal reordering demands large cache space to make blocking effective. In order to effectively use limited cache space, Gatlin and Carter [38] present an effective method using an additional buffer to first hold the conflict-missed elements of a block in one array temporarily, and then copy the block to their reordered positions in the other array. In this section, we discuss implementations of blocking methods supported by both software and hardware buffers.

### 4.2.1 Blocking With a Software Buffer and Its Limits

Because this buffer is defined in a reordering program, we call it a “software buffer”. This buffer shares the allocation space with the data arrays  $X$  and  $Y$  in the cache.

There are two major limits in this approach. First, the buffer itself may interfere with arrays of  $X$  and  $Y$ , causing additional access conflicts. This interference is certain when the sizes of  $X$  and  $Y$  are larger than the size of the cache,  $C$ . Each cache block or set is mapped from arrays  $X$  and  $Y$  more than once. No matter where the buffer is located in the cache, it will interfere with the arrays. The larger the buffer size, the more interference will occur.

The second limit is the additional copy overhead time involved in moving data from the array  $X$  to the buffer and then in moving them to the target array in their reordered positions. This overhead exactly doubles the instruction cycles for data copying. The data copy through a buffer is a worthy investment if the number of cycles lost from cache misses is much higher than the additional CPU cycles for the data copy.

To overcome the two limits, we propose several alternatives to eliminate cache interference caused by the software and to reduce or eliminate the data copy time.

### 4.2.2 Cache Structure Dependent Blocking

#### Blocking based on set associativity

The cache associativity,  $K$ , is an important factor to consider for blocking. If  $K \geq L$ , an  $L \times L$  or a  $K \times K$  blocking method for bit-reversals would effectively avoid conflict misses. Because the hit time is a less sensitive performance factor than the cache misses in the L2 cache, a higher associativity of the L2 cache is more effective than that of L1. For

example, if a cache line holds four double floating point elements ( $L = 4$  elements of 32 bytes in Pentium processors), a  $4 \times 4$  blocking method without any data buffer is able to fully use the cache associativity. The blocking method would gain more benefit from caches of associativity higher than 4, such as a design in [123].

What would we do if the associativity is not sufficiently high for the blocking, or  $K < L$ ? One solution is to make a  $K \times L$  rectangular blocking. Unfortunately bit-reversals require an  $L \times L$  blocking.

#### **Supplement with registers**

We may also consider using the available registers to supplement a low associativity cache. The number of registers available to a user program is limited. Normally, a uniprocessor provides up to 16 registers to users. For example, for a 2-way associative cache, we need 8 registers to buffer two additional cache lines so that we could effectively make a  $4 \times 4$  blocking as if we ran the program on a 4-way associative cache.

We develop a more efficient blocking method for bit-reversals, which requires only  $(L - K) \times (L - K)$  registers. The operation sequence of this method is in three steps: (1) The  $L - K$  cache lines of  $X$  are stored in  $K$  cache lines of  $Y$  and accessed by copying its  $(L - K) \times K$  elements to  $Y$  in the reordered positions, and copying the rest of  $(L - K) \times (L - K)$  elements to a buffer consisting  $(L - K) \times (L - K)$  registers. (2) The rest of  $K$  lines of  $X$  are brought to the cache set, and its  $K \times K$  elements are copied to  $Y$  in the reordered positions. (3) Finally, the  $(L - K) \times (L - K)$  elements in the register buffer and the rest of the  $(L - K) \times K$  elements are copied to  $Y$  in their reordered positions. A cache set will be used more than twice if  $K < L/2$ .

Besides the advantage of no access conflicts between the register buffer and the arrays

of  $X$  and  $Y$ , there is another advantage of using registers to buffer the data in a load/store processor. A data copy through the registers from  $X$  to  $Y$  is equivalent to the two-step process of load and store, and thus there will be no additional overhead. We will show our experimental performance in section 5.

### Using Registers As the Buffer

If the cache is direct-mapped, we have to fully rely on a buffer for blocking. Here we discuss some ways to use registers to serve the buffer in order to eliminate the potential cache conflicts and eliminate extra data copying by taking advantage of the load/store operations. The number of registers for a buffer of  $L \times L$  elements is determined by the number of elements a cache line can hold. The length of a cache line of the L1 cache in some processors, such as Sun SPARC Micro I and II, is  $L = 2$  of 16 bytes, which holds only two floating point elements. The blocking size could be as small as  $2 \times 2$  using a buffer of 4 registers.

The cache line length of the L1 cache in many advanced workstations is 32 bytes, such as the Sun Ultra and Intel Pentium processors, each of which holds 4 double floating point elements. In this case, we need a buffer of  $4 \times 4 = 16$  registers for a blocking. This would be difficult due to the limited number of available registers. We have two solutions for this. First, we only use the number of registers available to form a smaller buffer than it should be, which will not make each cache line fully used and will cause additional cache misses. Our experiments show that this blocking method of using a buffer of insufficient number of registers still achieves a reasonable performance improvement and outperforms of the implementation using a software buffer.

The second method is to further reduce the size of the buffer, which reduces the required

number of registers by using our  $(L - K) \times (L - K)$  blocking method.

### **L1 cache versus L2 cache**

The main objective of building two-level caches is to make the L1 cache small enough to catch up to the cycle time of the fast CPU, and to make the L2 cache large enough to capture as many accesses as possible [44]. In practice, the data size of a bit-reversal is larger than the size of the L2 cache. L1 and L2 caches offer different sizes of the cache line,  $L$ , and the associativity,  $K$ . Both of the following alternatives are effective for blocking. (1) Taking advantage of a short cache line and fast hit time of the L1 cache, we could effectively use limited registers as the buffer and make a small  $L \times L$  blocking effective. (2) Taking advantage of high associativity of the L2 cache, we could effectively use both associativity and supplemental registers as the buffer and make a large  $L \times L$  blocking effective.

### **4.2.3 Victim-cache-aided Blocking**

Victim cache [56] is a small fully associative cache serving as the buffer containing only cache blocks due to conflict misses from L1 cache. This is an on-chip cache connected between L1 and the next level cache or memory. On a miss in L1, the victim cache is first checked before going to the next level. If the missed block is found there, the victim cache block and the L1 cache block are swapped and then the block is delivered to CPU from the L1 cache. Victim cache has been available in some commercial workstations, such as HP7200.

The minimum number of victim cache lines required for  $L \times L$  blockings of transpose and bit-reversal reorderings is  $L - K$ . In execution,  $L \times L$  elements of each blocking are allocated in a set of  $K$  lines in L1 cache, and the rest of the elements are allocated in the  $L - K$  lines of the victim cache. The victim cache is able to hold all the conflict misses in

the reorderings by a  $L \times L$  blocking. In addition, a conflict miss in the L1 cache that hits in the victim cache has only one additional cycle miss penalty. Thus, a simple  $L \times L$  blocking method would be effective if such a victim cache is available.

However, the victim cache does not have a direct connection with the CPU. When a data hit happens in the victim cache, it has to be first swapped to the L1 cache and then delivered to CPU. This swapping operation is unnecessary for our reordering algorithms. Without counting the cold misses of bringing the elements in the first column for a  $L \times L$  blocking, and considering the LRU replacement policy, the entire blocking will have  $L \times (L - 1)$  conflict misses in the L1 cache, which are then found in the victim cache. This also means that each of such a blocking needs  $L \times (L - 1)$  additional swapping cycles between the L1 cache and the victim cache, which is independent of the associativity,  $K$ . In contrast with the blocking method based on the associativity supplemented by registers, the swapping cycles in the victim cache are additional overhead. Despite this, a victim-cache-aided blocking is more efficient than a blocking method with a software buffer because there are no cross interference conflicts between the victim buffer and arrays of  $X$  and  $Y$ .

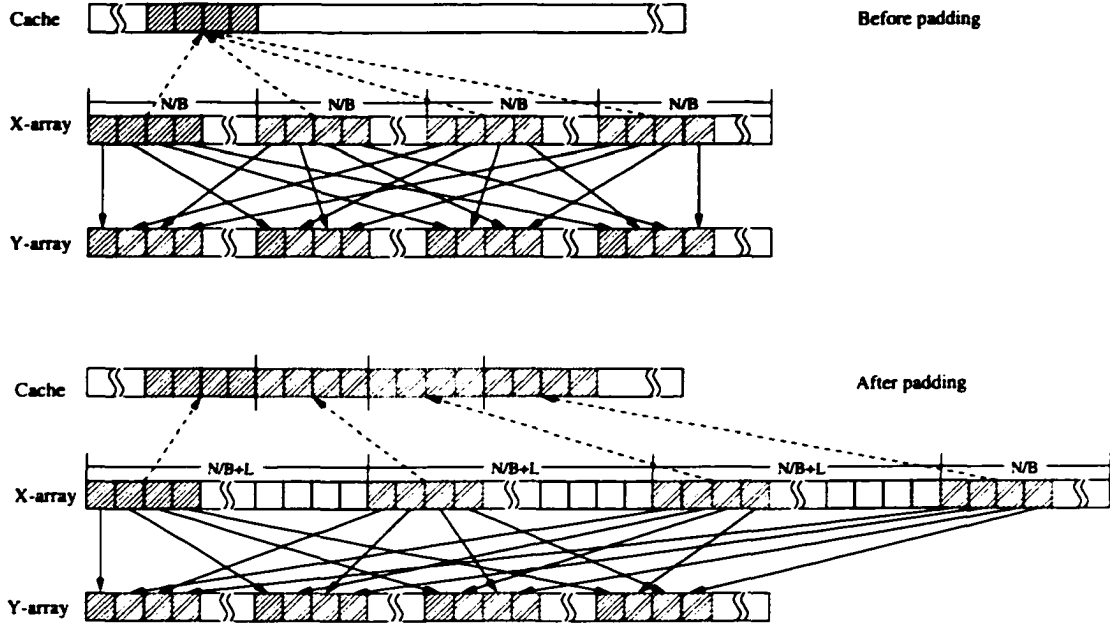
### 4.3 Blocking with Padding

Padding is a technique that modifies the data layout of a program so that the conflict misses are reduced or eliminated. The data layout modification can be done at run-time by system software [11, 119], or at compile-time by compiler optimization [86]. Sharing the same objective of compiler optimization to change the base addresses of potentially conflicting cache blocks in the reorderings, we insert padding variables inside the data array. For

example, the padding can be done as part of the last butterfly for the decimation in an FFT computation without additional cost, and the output is not padded.

However, we notice that this free padding opportunity may not be easily found, and the bit-reversal result may be padded in some cases. For example, the padding of a recursive implementation of the Cooley-Tukey FFT algorithm [23] is more complex than the padding in our implementations. The padding method produces padded results in a vector if the bit-reversals are done in place. The accesses to the padded results need to go through a simple address converting process with additional CPU cycles. In addition, our methods target bit-reversals based on the data size of powers of 2. However, FFT algorithms are not limited to this data size. If the data size is not a power of 2, the padding method will be more complex to implement. Poor memory performance of bit-reversals has been reported even for non-power of 2 data sizes (see e.g. [7]).

Since the data arrays of bit-reversals form a vector whose size is power of 2, the padding is highly regular, inserting  $L$  elements or a cache line space starting at the vector positions of  $N/L$ ,  $2 \times N/L$ , ..., and  $(L - 1) \times N/L$ . Using  $L$  elements or a section data of a cache line to separate the vector at these  $L$  points can completely eliminate the cache conflicts caused by the address mapping based on powers of 2. Again during execution, the reordering data copies are directly conducted between the arrays  $X$  and  $Y$  without going through a data buffer. Another advantage is that the number of padding elements needed is only  $L \times L$  or  $L$  cache lines, and is independent of the data array size,  $N$ . Compared with the data size of bit-reversals, the number of padding elements is insignificant. Figure 4.2 shows how the data layout of a bit-reversal vector is modified by padding so that conflict misses are eliminated.



**Figure 4.2:** Data layout of a bit-reversal is modified by padding, where  $B = B_{cache} = L$ .

Compiler optimization targets a large range of application programs, and automatically inserts padding variables in the programs for users. An optimal padding is application program dependent. For example, padding positions are different from different applications in order to effectively change base addresses of conflicting cache blocks. Based on the unique nature of the data reordering, the optimal padding unit used by our methods for bit-reversals is a cache line with  $L$  elements. In contrast, a compiler optimization normally uses an element as the basic padding unit. How many padding units to use and where to pad in the data arrays are determined by some approximation models which may not precisely fit the unique memory access patterns of each case. In addition, applying the padding technique to bit-reversals embedded in applications would not increase complexity in the entire computation. For example, when a padded bit-reversal is performed in a FFT computation, it has little effect on the neighboring butterfly operations.



## 4.4 Blocking and Padding for a TLB

The TLB (Translation-Lookaside Buffer) is a special cache that stores the most recently used virtual-physical page translations for memory accesses. The TLB is a small and usually fully associative cache. Each entry points to a memory page of 4 KBytes to 64 KBytes. The page size is normally fixed at the operating system level and cannot be changed by user programs. A TLB cache miss will make the system retrieve the missing translation from the page table in memory, and then select a TLB entry to replace. When the data to be accessed in our blocking method is larger than the amount of data of all the memory pages that the TLB can hold, we will have TLB thrashing. In this section, we will discuss and present blocking and padding methods for TLB cache optimizations.

### 4.4.1 Blocking for a Fully Associative TLB

Before giving a general model to show how blocking size is affected by TLB size, let's go through an example to show that a moderate  $N$  for bit-reversals would easily lead to TLB cache thrashing. The 64 pages in the TLB of the Sun UltraSparc-II processor hold  $64 \times 1024 = 65536$  elements, which represents a 16-bit-reversal of  $N = 2^{16}$ . Since we have two vectors  $X$  and  $Y$ , the TLB can hold a 15-bit-reversal of  $N = 2^{15}$  elements. This is also consistent with our experiments on this machine, where execution time per element was a constant until  $n = 15$ , but sharply increased at  $n = 16$  bit-reversals caused by the TLB misses.

In our cache-optimal methods, we include an outer loop to form a blocking for the TLB, whose size is denoted as  $B_{TLB}$ . The blocking size of  $B_{TLB}$  for bit-reversals when

$N \geq T_s \times P_s$  is

$$B_{TLB} \leq T_s,$$

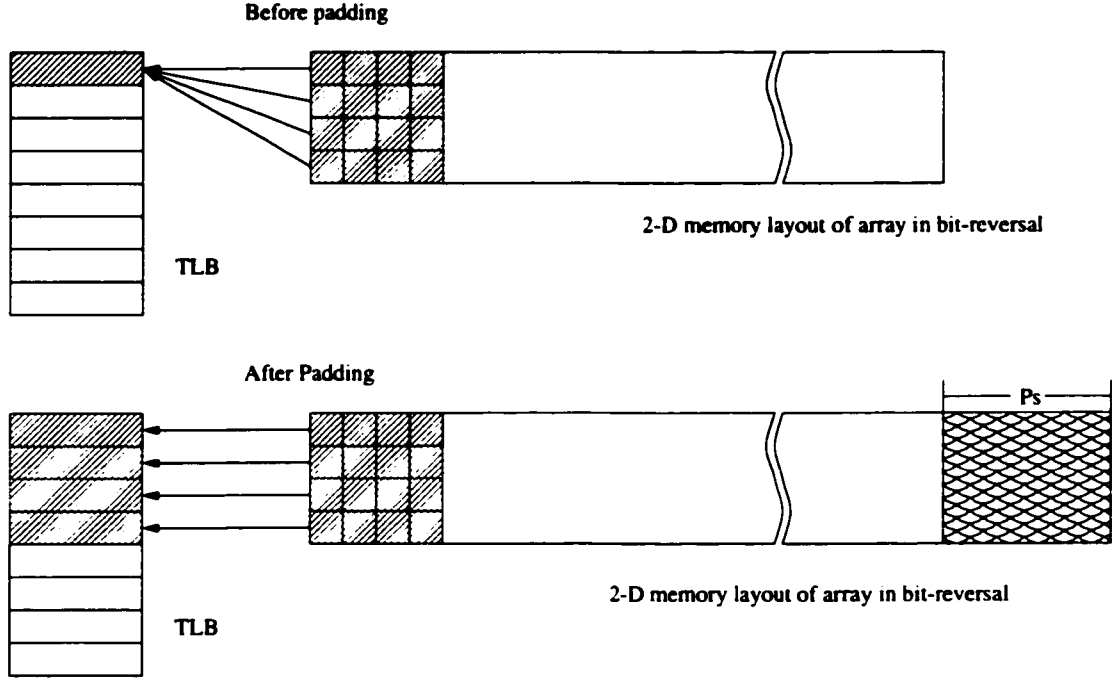
where  $P_s$  is the page size in elements, and  $T_s$  is the number of entries of the TLB. On the other hand, the  $B_{TLB}$  should be chosen as large as possible to make effective use of the page space. When  $N < T_s \times P_s$ , the data size of a bit-reversal will be less than the data size covered by the TLB. Thus there is no need for TLB optimizations.

#### 4.4.2 Padding for a Set-associative TLB

Some processors' TLBs are not fully associative, but set-associative. For example, the TLB in the Pentium-II 400 processor is 4-way associative ( $K_{TLB} = 4$ ). A simple blocking based on the number of TLB entries is not cache-optimal, because multiple pages within a TLB-size-based blocking may map to the same TLB cache set and cause TLB cache conflict misses.

If the size  $N$  of a bit-reversal vector is a multiple of  $T_s \times P_s$ , where  $T_s$  is the number of TLB entries and  $P_s$  is the page size in elements, and if  $K_{TLB} < B_{TLB}$ , then TLB cache conflict misses will occur. This could easily happen in practice. For example, on the Pentium-II 400,  $N$  is equal to 128K elements (one element = 8 bytes) for a 17-bit-reversal, and this  $N$  is two times of the value  $T_s \times P_s$  of the machine, where  $T_s = 64$ , and  $P_s = 1024$  elements.

In a way similar to the technique of padding for the data cache, we insert a page of elements or a page of space starting at the vector positions of  $N/L$ ,  $2 \times N/L$ , ... and  $(L - 1) \times N/L$  to eliminate the conflict of TLB cache misses. Figure 4.3 gives an example of the padding for a TLB, where the TLB is a direct-mapped cache of 8 entries, blocking



**Figure 4.3:** Padding for a TLB: the data layout is modified by inserting a page space at multiple locations, where  $B_{TLB} = 4$ ,  $K_{TLB} = 1$ ,  $T_s = 8$ .

size is  $B_{TLB} = 4$ , and the number of elements of a row is a multiple of 8 page elements. Before padding, each of blocking row is mapped to the same cache line of the TLB. After padding, these rows are mapped to different cache lines of the TLB.

Combining padding for data cache and padding for a TLB cache, we are inserting  $L + P_s$  elements or a page plus a cache line space in  $L$  locations separated by a distance of  $N/L$  elements.

In practice, we selected more than  $N/L$  points to insert the padding variables to eliminate both data cache and TLB conflict misses. This approach could effectively merge two nested paddings (one for the data cache and the other one for the TLB) into a single padding. An optimal number of inserting points can be easily determined experimentally based on the size of the TLB cache. The padding optimizations are all based on the L2

cache in our experiments.

Partial index mapping addresses of bit-reversals are pre-calculated and stored in a small table as shown in the program in the Appendix. This approach further improves performance because the table will be accessed in the cache during the computation, and the pre-calculation overhead is trivial. The time for the pre-calculation is included in the total execution time.

## 4.5 Experimental Results and Performance Evaluation

We have implemented and tested all the bit-reversal methods discussed in the previous sections on an SGI O2 workstation, a Sun Ultra-5 workstation, a Sun SMP server E-450, a Pentium PC, and a Compaq XP1000 workstation. We will present and evaluate the performance of different methods on different machines.

### 4.5.1 Experimental Environment and Evaluation Methodology

We used “lmbench” [73] to measure the latencies of memory hierarchies at different levels on each machine. The architectural parameters of the 5 machines are listed in Table 4.5.1.

We focus the performance evaluation on methods and implementations of bit-reversals in this chapter. We compared all our methods with the method of blocking with a software buffer which was recently published in [38]. We denote this method as “bbuf-br” — blocking with buffer for bit-reversals. Two of our methods are experimentally compared: “breg-br” — blocking with associativity and registers for bit-reversals, and “bpad-br” — blocking with padding for bit-reversals. We have also applied blocking or padding techniques for the TLB in these two methods based on the TLB associativity.

Workstations	SGI O2	Sun Ultra 5	Sun E-450	Pentium	XP1000
Processor type	R10000	UltraSparc Ili	UltraSparc II	Pentium II 400	Alpha 21264
clock rate (MHz)	150	270	300	400	500
L1 cache (KBytes)	32	16	16	16	64
L1 block size (Bytes)	32	32	32	32	64
L1 associativity	2	1	1	4	2
L1 hit time (cycles)	2	2	2	2	3
L2 cache (KBytes)	64	256	2048	256	4096
L2 block size (Bytes)	64	64	64	32	64
L2 associativity	2	2	2	4	1
L2 hit time (cycles)	13	14	10	21	15
TLB size (entries)	64	64	64	64	128
TLB associativity	64	64	64	4	128
Memory latency (cycles)	208	76	73	68	92

**Table 4.1:** Architectural parameters of the 5 workstations we have used for the experiments. All specifications on L1 cache refer to the L1 data cache, and all L2s are uniform. Each L2 cache block on UltraSPARC-Ili consists of 2 16-Byte sub-blocks. The hit times of L1, L2 and the main memory are measured by lmbench [73], and their units are converted from nanosecond (*ns*) to their CPU cycles.

All the programs use a standard subroutine to calculate the bit-reversal value for a given address. The execution times were collected by “gettimeofday()”, a standard Unix timing function. The resolution of this function is 1  $\mu$ s on the machines being measured, which is significantly smaller than the execution times of any programs we have measured. A small bit-reversal table is pre-calculated, and we exclude this calculation time. The reported time unit is cycles per element (*CPE*):

$$CPE = \frac{\text{execution time} \times \text{clock rate}}{N},$$

where *execution time* is the measured time in seconds, *clock rate* is the CPU speed (cycles/second) of the machine where the program is run, and *N* is the number of elements of the bit-reversal program. Besides the different methods of bit reversals, we also measured

the execution time of a program copying elements between  $X$  and  $Y$ . This program has the same number of data copying operations with a continuous memory access patterns. We use the execution time of this program to provide a base line reference for bit-reversal programs and show how close a bit reversal execution is to its ideal time. We denote this reference program as “base”. Each method is further divided into “float” data type using 4 bytes to represent an element, and “double” type using 8 bytes to represent an element. The data type divisions will show the performance impact of the cache line length.

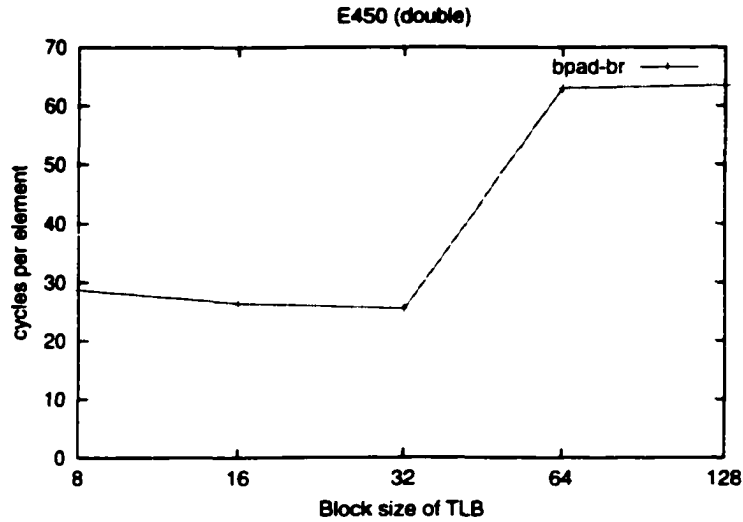
For all experiments on different machines, the bit-reversal programs first call a routine to flush the cache to make sure that all the data are allocated only in the memory. All experiments were repeated at least five times and the smallest execution times are reported.

#### 4.5.2 Effects of TLB and Virtual Memory

Before measuring and comparing the performance of different bit-reversal methods, we experimentally evaluated the effects of TLB and virtual memory to confirm our assumptions and analyses.

##### Selection of TLB blocking size

The TLB blocking size is a sensitive performance parameter to be selected, which is determined by the size of the TLB if it is fully associative. We executed program “bpad-br” (blocking with padding for bit-reversals) with  $n = 20$  on a single node of Sun E-450 by changing the blocking sizes for the TLB from 8 to 128. The TLB of the E-450 is a fully associative cache with 64 entries. Figure 4.4 shows the measured cycles per element of the program of different blocking sizes on the node. Our experimental results are consistent with our analyses in the previous section. When the blocking size for the TLB was 64, the

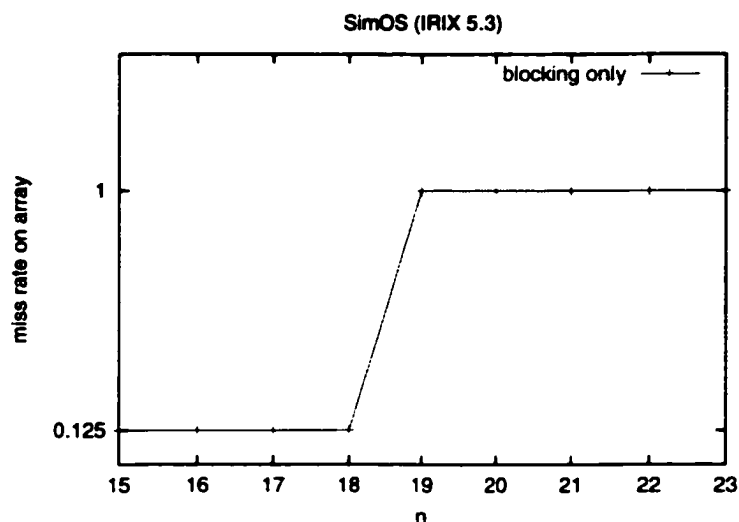


**Figure 4.4:** Changing the TLB blocking sizes on a single node of the Sun E450: when the blocking size for TLB was larger than 32, the execution time curve was sharply increased.

execution time curve increased sharply. This is because arrays  $X$  and  $Y$  together demanded more than 64 pages and caused TLB thrashing.

### Virtual memory versus physical memory addresses

All our analyses are based on cache mappings between memory pages in the virtual address space and cache blocks in the physical memory address space. This assumes that contiguous memory pages will be contiguously mapped to the cache. This assumption is guaranteed for the virtual-address caches [18]. However, all our experiments have been performed on machines with physical address L2 caches. Since the virtual-physical translations for L2 caches are handled by operating systems, our assumptions may not be accurate sometimes. In order to show that many operating systems attempt to map contiguous virtual pages to cache blocks contiguously so that our virtual-address-based study is practically meaningful and effective, we conducted a simulation by using the SimOS [91] and measurements on different workstations to observe how an operating system makes translations



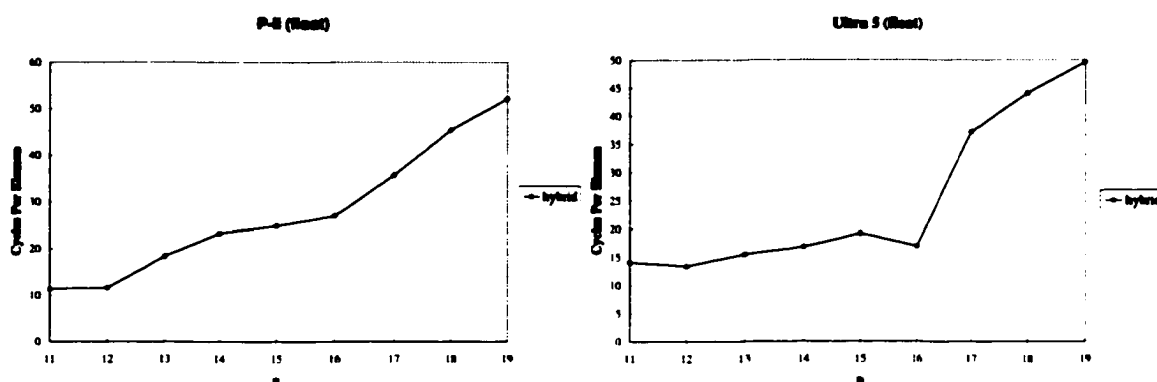
**Figure 4.5:** Using the SimOS to observe the miss rates by changing the the size of the bit-reversal arrays of a blocking-only program: when  $n > 18$ , the miss rate was sharply increased to 100%.

from virtual memory addresses to their physical addresses.

The SimOS simulates a complete hardware of SGI machines and runs the IRIX 5.3 operating system in the simulation. We executed a blocking-only program of bit-reversals using the cache line  $L$  as the blocking size. The bit-reversal vector size was changed from  $n = 15$  to  $n = 22$ . We measured the miss rates on array  $X$ . The cache size was set to 2 MBytes holding two double type arrays up to  $n = 18$  in the virtual memory space. Figure 4.5 gives consistent results from the SimOS simulation: when  $n > 18$ , the miss rate on array  $X$  was sharply increased to 100% from 12.5%. From this experiment, we have observed that virtual-physical translations from the IRIX 5.3 operating system are quite consistent to our assumption of “contiguous allocations”.

We have also run the similar experiments on different targeted workstations with different operating systems, such as Linux and Solaris, to measure the changes of execution times when the data size is changed. Our measurements are also consistent to the SimOS results,





**Figure 4.6:** Execution times of the hybrid method on the Pentium-II (left figure) and on the Ultra-5 machine (right figure).

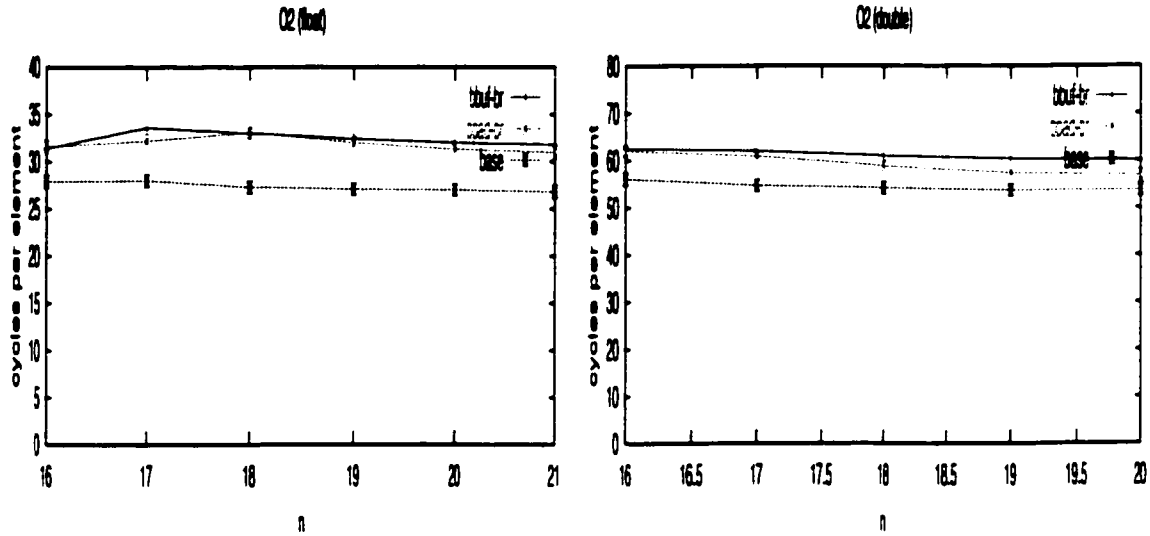
and indicate that the larger the data arrays to be used, the more likely an operating system will allocate the pages contiguously. Because our study targets large data sets, our analyses based on the virtual memory space is reasonably accurate. In addition, our methods assume that the operating system uses a uniform page size for page allocation, which is consistent with most commercial and commonly used operating systems.

### 4.5.3 Performance of the Hybrid Method for Bit-reversals

In order to show the effectiveness of our cache optimizations, we first plot the measured execution times of the hybrid method [58] in “float” data types on the Pentium-II and the Ultra-5 machines in Figure 4.6. Although the hybrid method did reasonably well for  $n \leq 16$  on Pentium-II and  $n \leq 12$  on Ultra-5, the execution times significantly increased due to limited cache performance after the data size was further increased.

### 4.5.4 Performance Comparisons on the SGI O2

The SGI O2 is a 1995 product using an R10000 processor of 150 MHz, 32 KB 2-way associative L1 cache, and 64 KB 2-way associative L2 cache. The cache line of L2 is 64



**Figure 4.7:** Execution comparisons on the SGI O2 workstation: “bbuf-br” represents the method of blocking with software buffer; “bpad-br” represents the method of blocking with padding; and “base” represents the ideal base line reference.

bytes. Since the associativity of L2 is low, and the cache line of L2 is relatively long, it is difficult to do blocking with associativity and available registers. We only implemented the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled bit-reversal methods from  $n = 16$  to  $n = 21$ . Figure 4.7 shows the comparisons of cycles per element among the three programs of both “float” type and “double” type on the SGI O2 machine. The measurements show that the padding method slightly reduced the execution time compared with the method of blocking with software buffer. The time reduction was up to 6%. The reason for the small performance improvement comes from the extremely long memory latency (208 cycles) of the O2 machine. The reduction and saving of instruction cycles for data copies from padding became less significant because memory latencies caused by the required cold misses in both methods were dominant in execution.

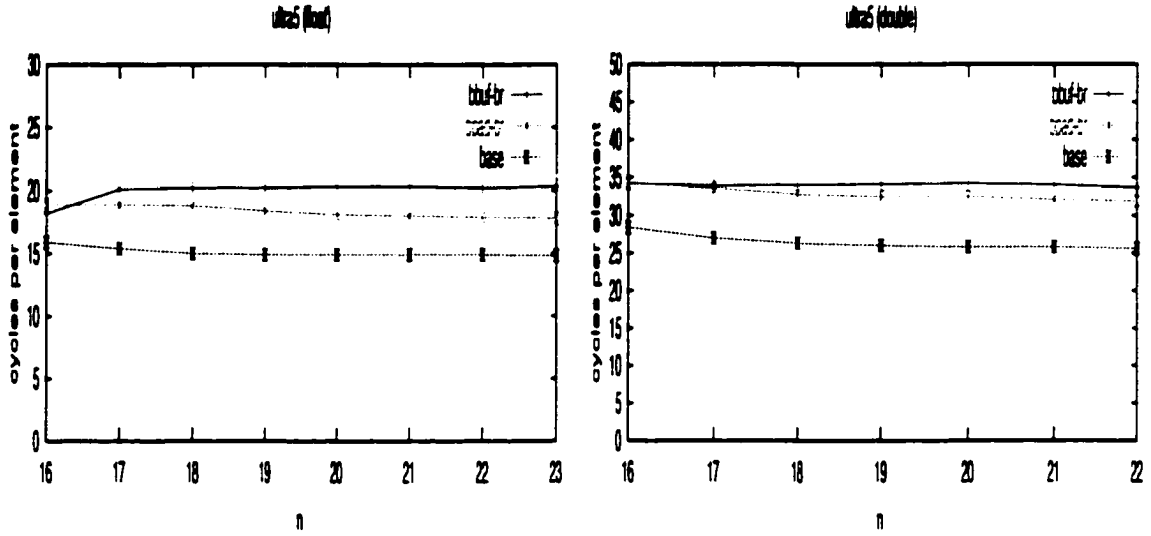
#### 4.5.5 Performance Comparisons on the Sun Ultra-5

The Sun Ultra-5 is a 1998 product using an UltraSparc-III processor of 275 MHz, 16 KB direct-mapped L1 cache, and 256 KB 2-way associative L2 cache. The cache line of L1 is 32 bytes consisting of two 16 byte subblocks, and L2 is 64 bytes long. Similar to the SGI O2, the associativity of L2 on the Ultra-5 is low, and the cache line of L2 is relatively long, so it is difficult to do blocking with associativity and available registers. We only implemented the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from  $n = 16$  to  $n = 23$ . Figure 4.8 shows the comparisons of cycles per element among the three programs of both “float” type and “double” type on the Ultra-5. The memory latency of the Ultra-5 (76 cycles) is significantly lower than that of the O2. We observed a more significant performance improvement from the method of blocking with padding over that of blocking with software buffer. For example, using “float” type, the padding program is 14% faster than that of blocking with buffer for  $n = 20$  or larger. A L2 cache line of the Ultra-5 holds 16 “float” type elements ( $L = 16$ ), and 8 “double” type elements ( $L = 8$ ). The larger the  $L$  (the number of elements in one cache line), the higher overhead the blocking with software buffer will have. This has been confirmed by our comparative experiments between the “float” and “double” types on the Ultra-5 shown in Figure 4.8.

#### 4.5.6 Performance Comparisons on the Sun E-450

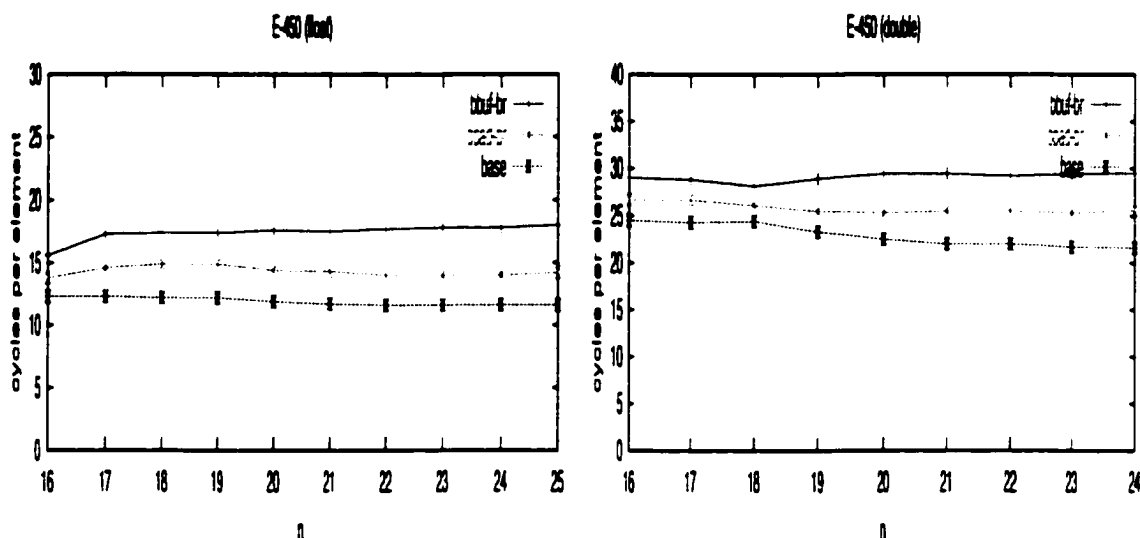
The Sun E-450 is a 1998 4-processor SMP product. Each of the 4 nodes is an UltraSparc-2 processor of 300 MHz, 16 KB direct-mapped L1 cache, and 2 MB 2-way associative L2



**Figure 4.8:** Execution comparisons on the Sun Ultra-5 workstation: “bbuf-br” represents the method of blocking with software buffer; “bpad-br” represents the method of blocking with padding; and “base” represents the ideal base line reference.

cache. The cache line of L1 is 32 bytes consisting of two 16 byte subblocks, and L2 is 64 bytes long. Due to the limited associativity and a relatively long L2 cache line, we only implemented the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from  $n = 16$  to  $n = 25$ . Figure 4.9 shows the comparisons of cycles per element among blocking with software buffer, blocking with padding, and the base program on a single node of E-450, each of which has both “float” type and “double” type. The memory latency of the Ultra-5 (73 cycles) is slightly lower than that of Ultra-5. On this machine, we observed higher performance improvement from the method of blocking with padding over that of blocking with software buffer. For example, using “float” type, the padding program is 22% faster than that of blocking with buffer for  $n = 20$  or larger. Our comparative experiments between the “float” and “double” types on E-450 in



**Figure 4.9:** Execution comparisons on the Sun E-450 SMP: “bbuf-br” represents the method of blocking with software buffer; “bpad-br” represents the method of blocking with padding; and “base” represents the ideal base line reference.

Figure 4.9 also confirms that the larger the  $L$ , the higher performance the padding method would achieve.

#### 4.5.7 Performance Comparisons on the Pentium-II 400

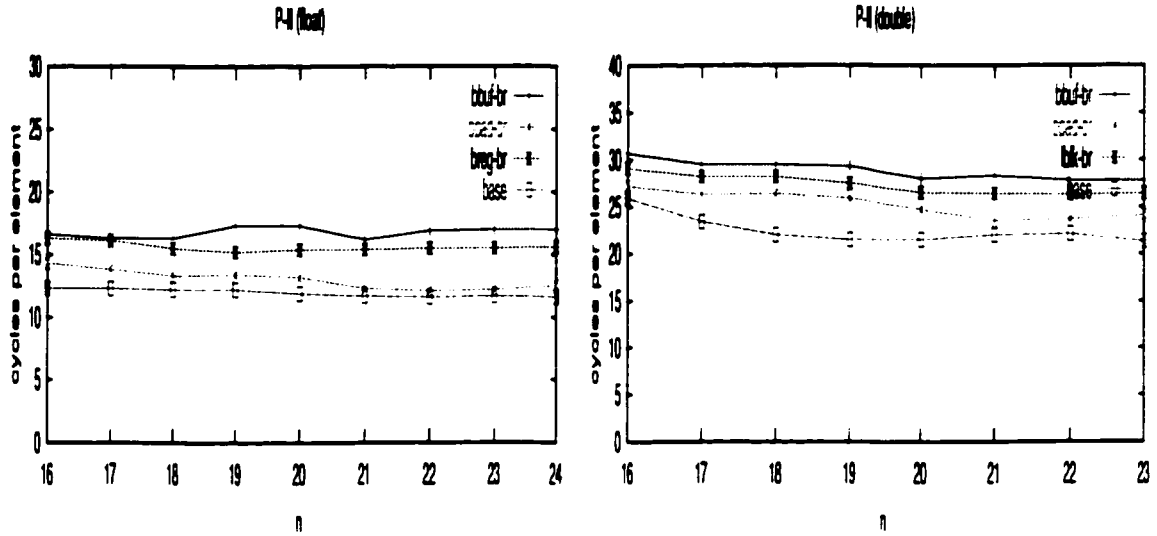
The Pentium PC we used is a 1998 product using a Pentium-II 400 processor of 400 MHz, 8 KB direct-mapped L1 cache, and 256 KB 4-way associative L2 cache. The cache lines of both L1 and L2 are 32 bytes. Since the L2 associativity is high, we are able to implement the method of blocking with associativity and available registers, L2 cache line  $L = 8$  elements for a “float” type, and we need  $(L - K)(L - K) = 16$  registers to supplement the 4-way associative cache. An L2 cache line holds 4 “double” type elements ( $L = 4$ ). Thus, we do not need any registers to supplement, but simply make a  $4 \times 4$  blocking. The TLB of the Pentium processor is a 4-way associative cache of 64 entries. We used our padding for the

TLB technique to avoid TLB misses. We implemented the blocking with padding method and the blocking with associativity and registers to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from  $n = 16$  to  $n = 24$ . Figure 4.10 shows the comparisons of cycles per element among the four programs. As we expected, the paddings for both cache and TLB were highly effective, and the padding program performed the best. For example, using “float” type, the padding program is about 40% faster than that of blocking with buffer for  $n = 22$  or larger. We also show that the method using available registers to supplement associativity is effective. Although it is not as good as the padding program due to the increase of the instruction counts for additional data copies, it still achieved up to 12% execution reduction over the blocking with software buffer program. As we expected, the execution time of the method using the 4-way associative L2 cache without the supplement of registers to form a  $4 \times 4$  blocking was delayed mainly by the longer L2 cache hit time. The performance of this method still outperformed the method of blocking with a software buffer.

#### 4.5.8 Performance Comparisons on the Compaq XP-1000

The Compaq XP-1000 is a 1999 product using an Alpha 21264 processor of 500 MHz, 64 KB 2-way associative L1 cache, and 4 MB 2-way associative L2 cache. The cache lines of both L1 and L2 are 64 bytes long. Similar to the SGI and Sun machines, the associativity of L2 on the XP 1000 is low, and the cache line of L2 is relatively long, so it is difficult to do blocking with associativity and available registers. We only implemented the blocking with padding method to compare with blocking with software buffer and the base reference.

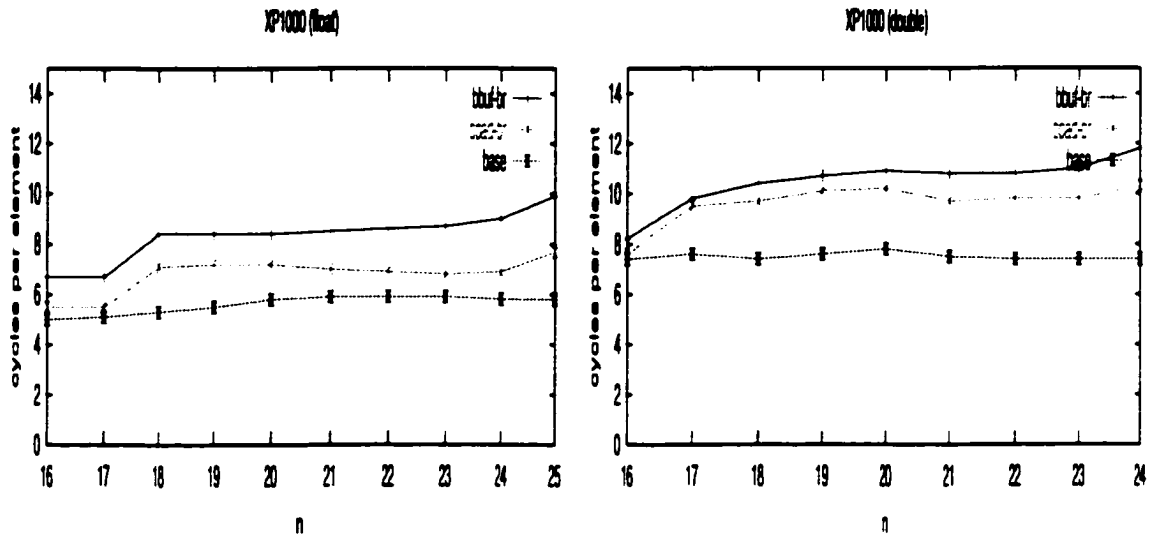


**Figure 4.10:** Execution comparisons on the Pentium-II 4000 PC: “bbuf-br” represents the method of blocking with software buffer; “bpad-br” represents the method of blocking with padding; “breg-br” represents the method of blocking with associativity and registers; and “base” represents the ideal base line reference.

We scaled the bit-reversal methods from  $n = 16$  to  $n = 25$ . Figure 4.11 shows the comparisons of cycles per element among the three programs of both “float” type and “double” type on the XP-1000 machine. As we expected, we achieved better or comparable performance to the ones on the Sun machines. For example, using “float” type, for  $n = 24$  or larger, the padding program is 30% faster than that of blocking with buffer; and 15% faster for “double” type.

## 4.6 Performance Evaluation on SMP Multiprocessors

We implemented the bit-reversal methods on two SMP multiprocessors: the The Sun E450 and the HP 9000 V2200. The parallel bit-reversal program on an SMP with  $M$  processors is described using POSIX thread primitives [51] as follows:



**Figure 4.11:** Execution comparisons on the Compaq XP-1000 workstation: “bbuf-br” represents the method of blocking with software buffer; “bpad-br” represents the method of blocking with padding; and “base” represents the ideal base line reference.

```

bit_reversal(id)

my_start = id*(N/M);

my_end = (id-1)*(N/M);

for i = 1, N

    Y[i'] = X[i];

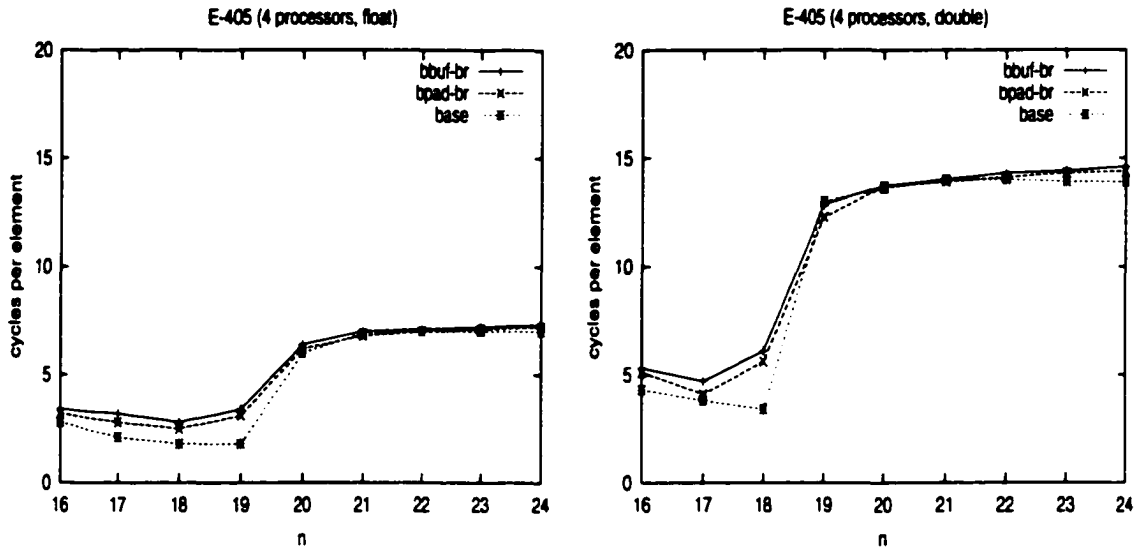
```

The bit-reversal operations are evenly distributed among  $M$  processors.

#### 4.6.1 Performance Comparisons on the Sun E-450

The Sun E450 is a 1998 4-processor SMP product. Each of the 4 nodes is an UltraSparc-2 processor of 300 MHz, 16 KB direct-mapped L1 cache, and 2 MB 2-way associative L2 cache. The cache line of L1 is 32 bytes consisting of two 16 byte subblocks, and L2 cache line is 64 bytes. Due to the limited associativity and a relatively long L2 cache line, we only



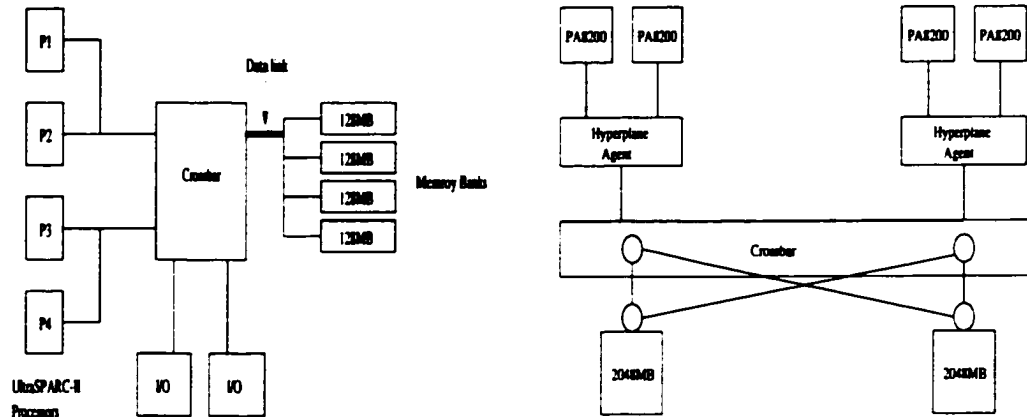


**Figure 4.12:** Execution comparisons on Sun E450 SMP of 4 processors: “bbuf-br” represents the algorithm of blocking with software buffer; “bpad-br” represents the algorithm of blocking with padding; and “base” represents the ideal base line reference.

implemented the blocking with padding algorithm to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal algorithms from  $n = 16$  to  $n = 24$ . Figure 4.12 shows the comparisons of cycles per element among blocking with software buffer, blocking with padding, and the base program on the E-450 of 4 nodes, each of which has both “float” type and “double” type. On this machine, we observed some performance improvement when  $n \leq 18$  from the algorithm of blocking with padding over that of blocking with software buffer.

However, when  $n > 18$  of double type or  $n > 19$  of float type, each processor has to process a data set larger than its cache capacity. Multiple processors simultaneously accessing the memory through a shared data link would cause the contention to degrade the performance. Since the data to be accessed from different processors are distributed



**Figure 4.13:** Architecture comparisons between Sun E450 SMP (left) and HP 9000 V2200 SMP (right): the memory data link of the E450 may become a bottleneck when simultaneous memory access requests from multiple processors; the HyperPlane crossbar connected between the memory modules and the processors on the HP 9000 V2200 can effectively reduce the contention.

in different locations, a crossbar interconnection network to link each processor to all the memory modules would significantly reduce the contention. The E450 does have a  $5 \times 5$  crossbar to connect 2 pairs of processors, 2 I/O ports and the memory. The communications between the 4 processors the memory modules are connected through the single memory data link. Figure 4.13 shows the crossbar interconnections of the E450 among the processors, the shared-memory modules and the 2 I/O ports. The contention occurs in the memory data link when the multiple processors request memory accesses simultaneously.

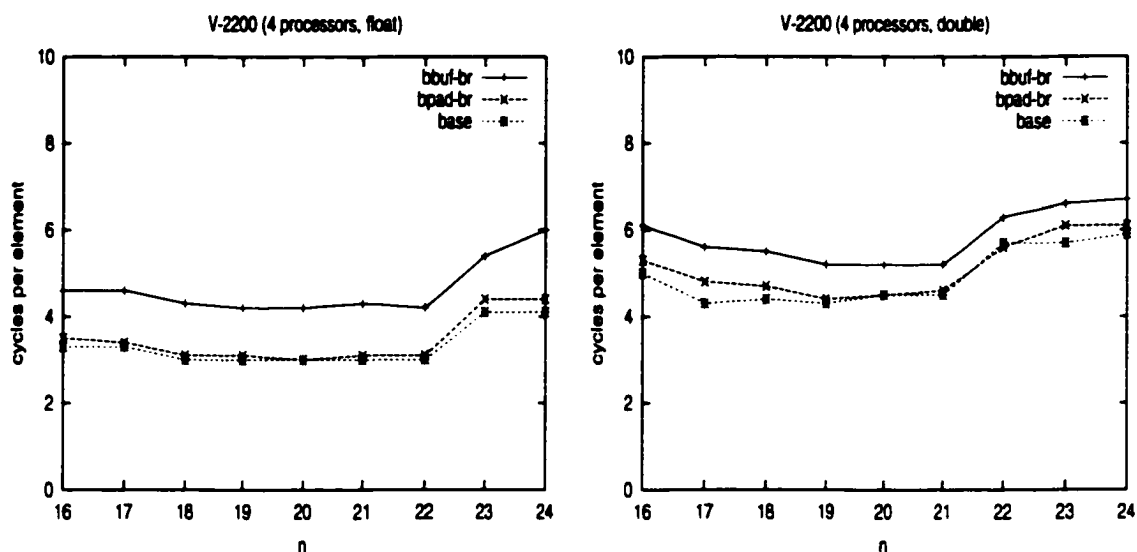
We have observed severe performance degradation caused by memory access contention. Figure 4.12 shows that this contention makes the execution time curves of the three programs jump sharply and merge together when  $n > 18$  of double type and  $n > 19$  of float type. In contrast, on a single processor of E-450, accesses to the the memory through the memory bus has no contention so that the algorithms scale well.

#### 4.6.2 Performance Comparisons on the HP 9000 V2200

HP 9000 V2200 is a 1997 SMP product with up to 16 processors. We used 4 processors for performance comparisons. Each node is a HP PA-8200 processor of 200 MHz with a 2 MB direct-mapped L1 data cache. The cache line is 32 bytes. Due to limited associativity, we only implemented the blocking with padding algorithm to compare with blocking with software buffer and the base reference.

The HP SMP has a crossbar interconnection network, the HyperPlane crossbar, to connect up to 8 pairs of processors to 8 memory modules. Multiple pairs of processors can access different memory modules simultaneously. Each pair of the processors is connected to the crossbar through an adapter called HyperPlane Runway Agent. Figure 4.13 gives the interconnection structure of the HP 9000 v2200 of 4 processors.

In our experiments, the 4 processors are divided into two pairs which are connected to two memory modules by a  $2 \times 2$  hyperplane crossbar. Each pair of processors may have contention competing for the adapter, but the crossbar is able to allow simultaneous data accesses among the memory modules. The negative performance effect due to the data link contention observed on Sun E450 was significantly reduced on the HP SMP, which shows the effectiveness of the crossbar. Figure 4.14 shows comparative execution time curves between the “float” and “double” types on E450 in Figure 4.14. The execution times of the three programs are quite stable and independent of the size of  $n$ . Both the padding programs of the float type and of the double type outperformed the blocking methods with buffer up to 40% and 18% respectively. Their execution curves almost merge together with the base reference curve.



**Figure 4.14:** Execution comparisons on HP 9000 V2200: “bbuf-br” represents the algorithm of blocking with software buffer; “bpad-br” represents the algorithm of blocking with padding; and “base” represents the ideal base line reference.

## 4.7 Summary

We have examined and developed cache-optimal methods for bit-reversal data reorderings. These methods have been tested on 5 representative uniprocessor workstations of 1995 to 1999 products to show their effectiveness. Different methods have their merits and limits. The blocking only method is limited by data sizes. Although the blocking with software buffer method is architecture independent, it increases cross interference, instruction count and needs additional memory space. The blocking with a register buffer method is fast but is limited by the number of available registers. Blocking with associativity and registers works well on high associativity caches. We have shown that the methods of blocking with padding, blocking for TLB and padding for TLB can effectively exploit cache locality, and are almost independent of hardware. Thus, they could be widely used on many uniprocessor workstations and SMP multiprocessors. We summarize different techniques and their merits

methods	cross interference	Instruction count	memory space	program complexity	comments
blocking only				0	limited by data sizes.
blocking with software buffer	+	+	+	1	system independent.
blocking with register buffer				1	limited by the number of available registers.
blocking with associativity and registers				2	works well on high associativity caches.
blocking with padding			+	1	works well on all systems.
blocking for TLB				0	a TLB size dependent outer loop, effective for fully associative TLBs.
padding for TLB			+	1	paddings by using $L$ pages, effective for set associative TLBs.

**Table 4.2:** Summary of the blocking methods and their impact on the three aspects of performance (cross interference, instruction count, and memory space) and on the program complexity. The performance of “blocking only” method is the base line for comparisons. Note: + means that the method quantitatively increases the factor and hurt the performance; and blank means it has no impact. The program complicity is subjective, and compared with the “block only” method, with 1 being a slightly more complex, and 2 a moderately more complex.

and limits in Table 4.2, which gives a guideline for application users to choose a technique based on the size of the problem and the machines available.

The methods have also been tested on two commercial SMP multiprocessors. By exploiting cache locality of each processor, we have effectively eliminated the conflict misses so that accesses to the shared memory and contention are minimized. However, another potential bottleneck on SMPs is the data access contention to the shared-memory. We show that crossbar interconnections between processors and memory modules play an important role in parallel bit-reversal data reorderings.

## **Chapter 5**

# **Reduce DRAM Row-buffer**

## **Conflicts by Breaking Address**

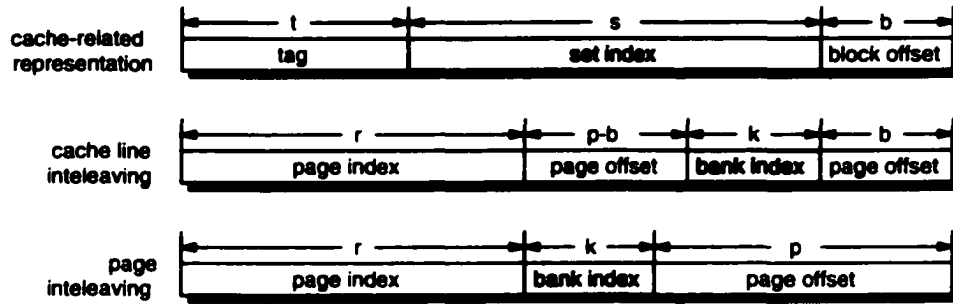
## **Mapping Symmetry**

Chapter 4 presents software methods at the application level to reduce memory stall time. In this chapter, we target directly reducing DRAM access latency using hardware methods. Our focus is on DRAM row buffer, a cache structure existing in DRAM chips. We first show that the address mapping symmetry between cache and DRAM is the inherent source of row-buffer conflicts in a typical memory hierarchy. Breaking the symmetry to reduce the conflicts and to retain the spatial locality, we propose and evaluate a permutation-based page interleaving scheme. We have also evaluated and compared two representative cache mapping schemes that break the symmetry at the cache level. We show that the proposed page interleaving scheme outperforms all other mapping schemes based on its overall performance and on its implementation simplicity.

We have made three contributions in this study:

- We show that the address mapping symmetry between the cache level and the DRAM level is the architectural source of row-buffer conflicts. Breaking the mapping symmetry can remove this source of conflict. We propose a permutation-based page interleaving scheme to break the mapping symmetry at the DRAM level and retain the spatial locality of the row buffer. Our performance results show that the new scheme effectively reduces the row buffer miss rate and the memory stall time for SPEC2000 programs.
- We examine existing cache mapping schemes and their effects on reducing the conflicts at the DRAM row buffers, and show that breaking the address mapping symmetry at the cache level is effective in reducing conflicts at both cache and DRAM row-buffer levels. However, the reduction of average cache miss rates is insignificant, and the increase of processor core complexity by this approach is nontrivial. The results of this study show that our permutation-based page interleaving scheme has the lowest row buffer miss rates and the best overall performance, while the increase of complexity by this approach is trivial and is outside the processor core.
- We evaluate the effects of large cumulative DRAM row-buffer sizes on the effectiveness of the permutation-based page interleaving scheme. We find that the scheme can still be effective for large row buffers, even when the cumulative row-buffer size is larger than the L2 cache size.

The rest of this chapter is organized as follows. Section 5.1 presents the existing address mapping scheme. Section 5.2 provides insights into the address mapping symmetry, and shows how conflicts are caused by the symmetry. Section 5.3 describes the permutation-



**Figure 5.1:** Bit representations of a memory address for both cache addressing and memory addressing with conventional cache-line and page interleaving schemes.

based page interleaving scheme. Section 5.4 describes the experimental environment. Section 5.5 presents the performance results of using the permutation-based page interleaving scheme. Section 5.6 evaluates the effectiveness of several cache mapping schemes on reducing row-buffer conflicts, and discusses their merits and limits. Section 5.7 investigates the case that the cumulative row buffer size is very large. Finally, Section 5.8 discusses the related work, and Section 5.9 summarizes this study.

## 5.1 Existing Address Mapping Schemes

Almost all computer systems today use conventional interleaving schemes for both caches and DRAM. Figure 5.1 shows the bit representations of a memory address for conventional cache-line and page interleaving, and gives the relationship between the cache-related representation and the memory-related representation for given memory hierarchical configuration.

The cache-line interleaving scheme uses the  $k$  bits above the low order  $b$  bits (L2 block offset) as the memory bank index. In the uniprocessor system, the processor usually requests



data from the memory in a unit of an L2 cache line. The cache-line interleaving scheme attempts to access multiple memory banks uniformly (e.g. [25]). However, since continuous cache lines are distributed in different memory banks, this scheme can not effectively exploit the data locality in the row buffer.

The conventional page interleaving scheme uses the  $k$  bits above the low order  $p$  bits (page offset) as the bank index. This balances between exploiting the data locality in row buffer and referencing memory banks uniformly. However, it may cause severe row buffer conflicts in some typical cases that we will discuss next.

The high order interleaving scheme uses the high order  $k$  bits as the bank index. This exploits more data locality than low order interleaving, but also makes accesses to multiple banks less uniform. In addition, continuous accesses in DRAMs crossing the page boundary will incur precharge and row access penalty. Thus, there is no benefit to exploit spatial locality beyond the page size.

## 5.2 Mapping Symmetry and Row Buffer Conflicts

We consider row buffer conflicts in the context of writeback caches with the conventional cache address mapping and DRAM memory with the page-interleaving scheme. We define that two addresses are *cache-conflicting* if they have the same cache index but different cache tags. In other words, they are in different blocks that are mapped to the same cache set. We define that two addresses are *row-buffer-conflicting* if they have the same bank index but different page indices, i.e., they are in different pages of the same bank. We have the following findings:

- Two cache-conflicting addresses are row-buffer-conflicting, provided the cache size divided by the cache associativity is larger than or equal to the cumulative row buffer size. We call this condition as the *large-cache condition*. When the large-cache condition holds, the bits for selecting the bank index is a subset of the bits for selecting the cache set index, and the bits for selecting the page index is a super set of the bits for selecting the cache tag, as shown in Figure 7.2. Two cache-conflicting addresses have the same cache set index, thus they have the same bank index. On the other hand, they must have different cache tags, so their page indices are different. Therefore, they are row-buffer-conflicting.
- Assume the large-cache condition holds. For writeback caches, the block address of a writeback is row-buffer-conflicting with the block address of the miss that causes the replacement. The two block addresses must be mapped onto the same cache set, and thus are cache-conflicting. Thus, they are row-buffer-conflicting.
- Assume the large-cache condition holds. Cache conflict misses may possibly result in row-buffer conflicts. We will use examples to explain this effect in Section 5.2.3.

Mapping symmetry refers to the fact that both cache and DRAM address mappings use the simple interleaving scheme, and use many common bits for selecting the module to map (cache set at cache level and bank at DRAM level). In particular, when the large-cache condition holds, all bits for selecting DRAM bank are used in the bits for selecting cache set.

### 5.2.1 Large-cache Condition in Computer Systems

The large-cache condition is common in today's computers. For a given cache and DRAM chip configuration, there is a threshold of memory size under which the large-cache condition will hold, and this threshold is generally large. For example, assume a computer has a 2MB 2-way associative L2 cache, and its memory system uses DRAM chips that have 8192 rows (pages) per bank<sup>1</sup>. For those chips, the ratio of row buffer size to DRAM capacity is 1:8192. In this example, the large-cache condition holds until the memory size increases beyond 8 GBytes. In practice, it is possible that the memory size is larger than the threshold. However, row buffer conflicts can still be severe. We will discuss this in Section 5.7.

### 5.2.2 Effect of Cache Writebacks

The writeback policy is commonly used for an L2 cache to reduce memory bandwidth demand, which has been a crucial issue as the processor speed increases [13]. When a writeback happens, as discussed above, the addresses of the related miss and writeback are row-buffer-conflicting. For a writeback and write-allocate cache, either a read miss or a write miss results in a memory read request. The writeback results in a memory write request. Normally, programs have spatial locality. When a sequence of replacements of a dirty cache blocks happens, the read requests and the write requests conflict on the row buffer. This causes frequent row-buffer conflict misses while the pages with the read addresses and the write addresses are replaced and retrieved back and forth.

We will use the following example to show this effect:

---

<sup>1</sup>This is common for 256Mbit SDRAM chips commercially available.

```
double X[N], Y[N], sum = 0;

int i;

...

for (i = 0; i < N; i ++){

    X[i] = i;

    ...

    for (i = 0; i < N; i ++){

        sum += Y[i];
```

We assume that the cache is direct-mapped, array  $X$  and array  $Y$  are mapped onto the same cache sets, and array  $Y$  is not loaded into the cache at the beginning of execution. At the time array  $Y$  is accessed, a sequence of misses happens and each miss causes a writeback. From the DRAM point of view, a sequence of read requests and a sequence of write requests come to different pages in the same bank during a short time frame when the bank is accessed. In this worst case, each read or write results in a row buffer miss.

Write buffers can be used to reduce processor stalls waiting for memory writes [31, 100]. The write buffer can be implemented with read bypass (read misses have higher priority than writes) or with no-bypass. The write buffer with no-bypass will not change the access patterns causing row-buffer conflicts. The write buffer with read bypass can alleviate row buffer conflicts by postponing the writebacks and grouping consecutive reads together. The effectiveness of the write buffer depends not only on its size, but also on when the buffered data are written to memory. One write policy for reducing row-buffer conflicts is to write the buffered data to memory only when the number of pending writes reaches a threshold.

However, since writebacks are not issued immediately when the memory system is free, the delayed writebacks may compete with subsequent reads and increase their latencies. Another write policy is to write the buffered data to main memory whenever there are no outstanding reads. However, the memory access patterns do not change so much in this case. In Section 5.5.3, we will show with experiments that using write buffers may reduce row-buffer miss rates but fails to reduce memory stall time.

### 5.2.3 The Effect of Cache Conflict Misses

Some typical patterns of cache conflict misses will result in row buffer conflicts. For example,

```
double X[N];  
  
...  
  
double Y[N], sum, i;  
  
...  
  
for (i = 0; i < N; i ++)  
  
    sum += X[i] * Y[i];
```

Without losing generality, assume the cache is direct-mapped, the arrays are contiguous in the physical memory space, and  $X[0]$  and  $Y[0]$  are mapped to the same cache block. Severe cache conflict will happen and each access to  $X[i]$  or  $Y[i]$  will result in a cache miss. From DRAM point of view, two sequences of read requests to the same bank are interleaved during a short time frame while the bank is accessed. Each read request will result in a row buffer miss.

Cache conflicts may be reduced by increasing cache associativity, by using a victim cache [56], or using other hardware/software approaches. However, this does not alleviate

the row-buffer conflicts due to writeback. In those cases, cache conflict misses can be a secondary source of row buffer conflicts.

### 5.3 A Permutation-based Page Interleaving

In order to address the problem of row-buffer conflicts caused by cache writebacks and cache conflict misses, we introduce a new memory interleaving scheme that generates different bank indices in a way that retains spatial locality and reduces row-buffer conflicts.

#### 5.3.1 The Scheme and Its Properties

Our memory interleaving scheme, called *permutation-based page interleaving*, is shown in Figure 5.2. The low order  $k$  bits of the L2 tag and the original bank index are used as the input to a  $k$ -bit bitwise XOR logic to generate the new bank index. The page index and the page offset are unchanged. The selection of  $k$  bits from the bank index under the conventional page interleaving scheme keeps the same degree of data locality, while the selection of  $k$  bits from the L2 tag attempts to make a wide distribution of pages among banks for exploiting concurrency. Other design choices could be used with the same mapping principle. We will discuss these later.

Let  $\langle a_{m-1}a_{m-2}\cdots a_0 \rangle$  be the binary representation of a memory address  $A$ . Then the bank index under the conventional page interleaving,  $I$ , is  $\langle a_{k+p-1}\cdots a_p \rangle$ . The new bank index after applying the permutation-based page interleaving scheme,  $I'$ , is

$$a'_i = a_i \oplus a_{m-t+i-p} \quad \text{for } i = p, \dots, k+p-1 \quad (5.1)$$

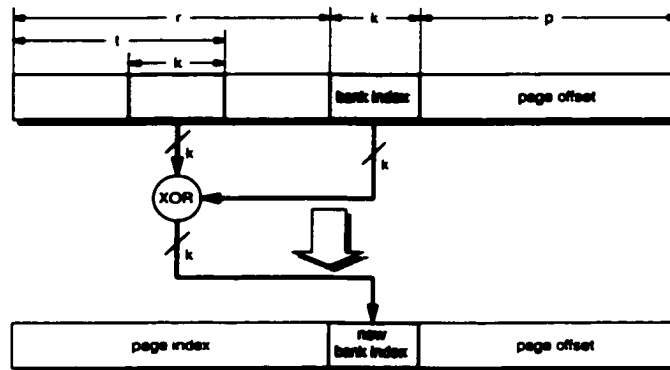


Figure 5.2: The permutation-based page interleaving scheme.

This interleaving scheme has the following properties, which are useful in achieving the objectives of exploiting both the concurrency and the data locality:

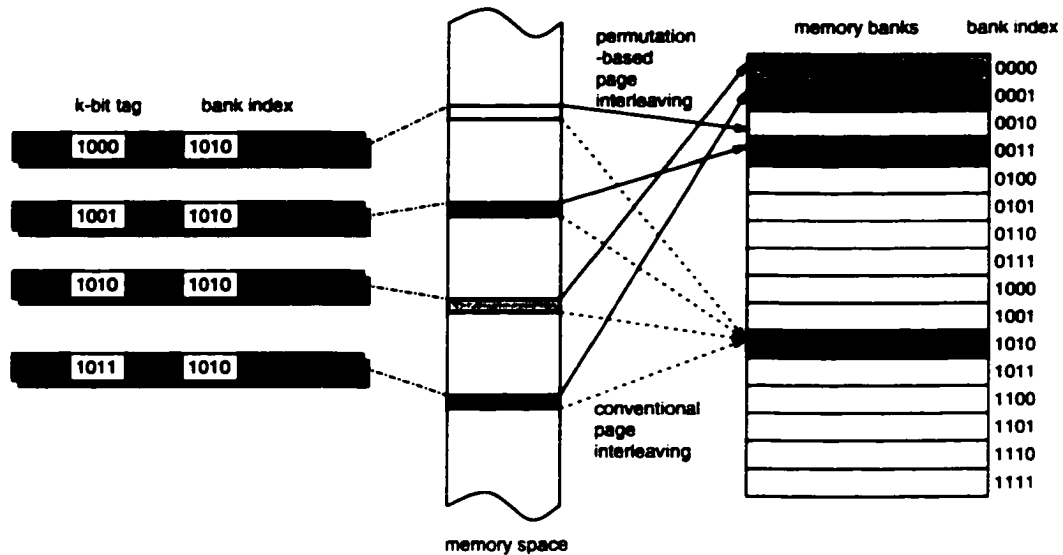
1. *Cache-conflicting addresses are distributed onto different banks.*

Given any two cache-conflicting addresses, their bank indices in conventional page interleaving are identical, but their  $t$ -bit L2 tags are different. As long as the low order  $k$  bits of the two tags are different, the  $k$ -bit XOR function will produce two different bank indices. Figure 5.3 shows an example of mapping four L2-conflict addresses onto 16 banks. All the four addresses are mapped onto the same bank in conventional page interleaving. After applying the permutation-based page interleaving scheme, they are distributed onto four different banks.

2. *The spatial locality of memory references is preserved.*

All addresses in the same page are still in the same page after applying our interleaving scheme.

3. *Pages are uniformly mapped onto multiple memory banks.*



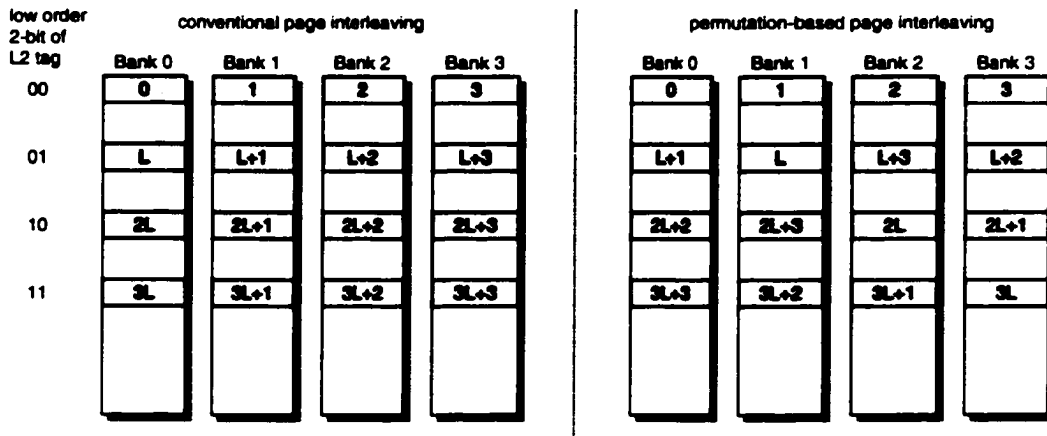
**Figure 5.3:** An example of mapping four memory addresses with the conventional page interleaving and the permutation-based page interleaving schemes. Only the *k*-bit bank index and the low order *k*-bit of L2 tag are shown for each address.

The permutation-based page interleaving scheme still uniformly maps continuous pages onto multiple memory banks, since the conventional bank index information is used in the mapping. Figure 5.4 gives an example to show that continuous pages are uniformly mapped onto four memory banks by both the conventional and the permutation-based page interleaving schemes.

One would think that spatial locality of memory references could be maintained and page conflicts could be reduced by using only the low order *k* bits of the L2 tag as the bank index, thus avoiding the XOR operation. The limit of this approach is that it maps a large fraction of the memory space (of the L2 cache size) onto the same bank. This would create hot spots on some memory banks and introduce a new source of page conflicts.

There are several alternatives to the selection of *k* bits among the *t*-bit L2 tag. Since pro-





**Figure 5.4:** An example of mapping continuous pages onto four memory banks under the conventional and the permutation-based page interleaving schemes, where  $L$  is the number of pages the L2 cache can hold.

grams have data locality, it is more likely that higher order bits of L2-conflict addresses are the same. Our experiments show that choosing the low order  $k$  bits achieves or approaches the lowest row-buffer miss rate for all the benchmark programs used.

We will later show that the risk for the XOR operation to cause more row-buffer conflicts is very small in practice. A major reason for this is as follows. The memory space can be divided into segments in the unit of the cache size. The XOR operation uses the same  $k$ -bit L2 tag for the addresses in each segment. Thus, it does not change the conflicting relationship between any pair of addresses in each segment, which is defined as whether the pair is mapped onto the same row buffer or not. Our analysis also shows that the XOR operation may increase the chance of conflicts only for addresses in some specific segment boundaries. Since the cache size is sufficiently large in current computer systems, these addresses form a very small subset in the entire memory address space.

The mapping function of a memory interleaving scheme must satisfy the one-to-one

property [84]. For a given memory address  $A$ , we can obtain its memory location  $A'$  using the permutation-based interleaving scheme by computing its bank index  $I'$  using equation (5.1). Conversely, for a given memory location  $A'$ , we can obtain its address  $A$  by computing  $\langle a_{k+p-1} \dots a_p \rangle$  as  $a'_i \oplus a'_{m-t+i-p}$  for  $i = p, \dots, k+p-1$ . When the large-cache condition holds,  $(s+b) > (k+p)$ . Thus, for  $i = p, \dots, k+p-1$ ,

$$a'_i \oplus a'_{m-t+i-p} = (a_i \oplus a_{m-t+i-p}) \oplus a_{m-t+i-p} = a_i. \quad (5.2)$$

Therefore, the permutation-based mapping function has the one-to-one property.

### 5.3.2 Comparisons with the Swapping Scheme

The swapping scheme is another interleaving scheme that is proposed to reduce the row buffer conflicts. Zurawski, Murray, and Lemmon [127] present the scheme that swaps partial bits of the L2 tag and partial bits of the page offset, which is used in the AlphaStation 600 5-series workstations. We call it the swapping scheme in this study. Wong and Baer [118] study the performance of the swapping scheme for selected SPEC92 benchmark programs by finding the optimal number of bits to be swapped for these programs.

Figure 5.5 describes the swapping scheme. This scheme maps every  $2^n$  L2 conflict addresses (with the same  $\langle a_{p-1} \dots a_{p-n} \rangle$ ) to the same page. Thus, if two L2 conflict misses have the same high order  $n$  bits in their page offsets, they will cause page hits. However, if two L2 conflict misses have different high order  $n$  bits in their page offsets, they will still cause page conflicts. In addition, the swapping scheme may degrade the spatial locality of memory references because the block size of continuous addresses inside a page is decreased from  $2^p$  to  $2^{p-n}$ . The more bits that are swapped using this method, the more conflict misses

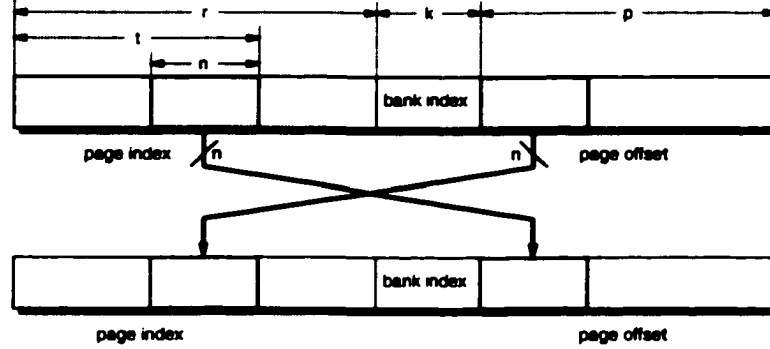


Figure 5.5: The swapping scheme.

can be removed, but the less spatial locality is retained. In contrast, the permutation-based scheme reduces page conflicts and preserves data locality at the same time.

The swapping scheme attempts to convert accesses to different pages in the same bank into accesses to the same page. The permutation-based scheme attempts to convert accesses to different pages in the same bank into accesses to different banks. The permutation-based scheme not only reduces the row-buffer conflicts of current accesses, but also potentially increases the row-buffer hit rates for subsequent accesses.

## 5.4 Experimental Environment

We use our enhanced *sim-outorder* to configure an 8-way processor, to set the load/store queue size to 32, and to set the register update unit size to 64 in the simulation. The processor allows up to 8 outstanding memory requests, and the memory controller has the ability to accept up to 8 concurrent memory requests. Reads are allowed to bypass writes. The outstanding writes are scheduled to memory modules as soon as there are no outstanding reads. Table 5.1 gives the major architectural parameters.

CPU Clock rate	1.6 GHz
L1 inst. cache	32 Kbytes, 2-way, 32-byte block
L1 data cache	32 Kbytes, 2-way, 32-byte block
L1 cache hit time	2 processor cycles
L2 cache	2 Mbytes, 2-way, 64-byte block
L2 cache hit time	10 processor cycles
memory bus width	32 bytes
memory bus frequency rate	133 MHz
number of memory banks	4~256
row buffer size	1~8 Kbytes, and 64 KBytes
DRAM precharge time	24 ns
DRAM row access time	24 ns
DRAM column access time	24 ns
L2 MSHR	8 entries
Write buffer	8 entries

**Table 5.1:** Architectural parameters of simulation

We use the SPEC2000 [45] as workloads, which are more memory-intensive than SPEC95. There are thirteen programs with significant memory stall times (measured by the differences using two simulations, one with an infinite L2 cache and one with a 2-way 2-MByte L2 cache). We include all those programs in experiments, as shown in Table 5.2. We use the precompiled SPEC2000 benchmarks provided by Weaver [117](ISA-Alpha). For all programs, we fast-forward 4000M instructions and collect program execution statistics on the next 200M instructions (here  $1\text{M} = 10^6$ ).

## 5.5 Performance Evaluation of Permutation-based Page Interleaving Scheme

In this section, we evaluate the permutation-based page interleaving scheme by comparing it with three other interleaving schemes: cache-line interleaving, page interleaving, and

Name	Remarks
181.mcf	Minimum cost network flow solver
197.parser	Natural language processing
168.wupwise	Quantum chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver in 3D potential field
173.applu	Parabolic/elliptic partial differential equations
178.galgel	Fluid dynamics: analysis of oscillatory instability
179.art	Neural network simulation; adaptive resonance theory
183.quake	Finite element simulation; earthquake modeling
187.facerec	Computer vision: recognizes faces
188.amp	Computational chemistry
189.lucas	Number theory: primality testing
301.apsi	Solves problems regarding temperature, wind,

Table 5.2: SPEC2000 programs used in performance evaluation.

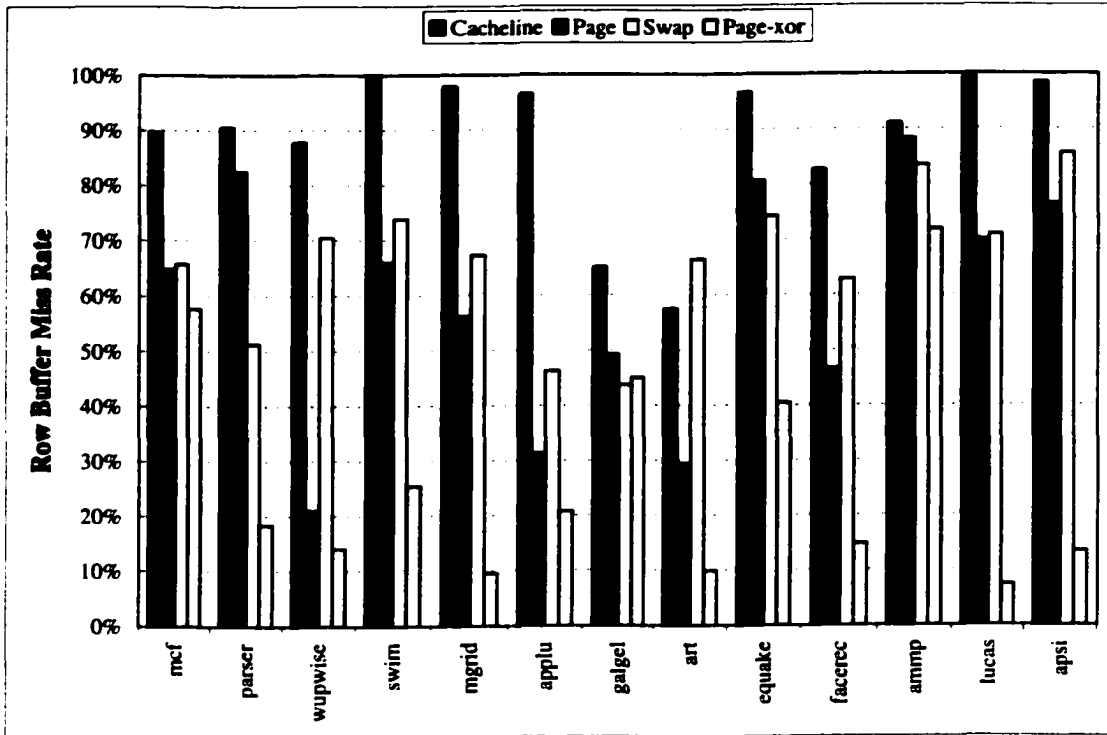
swapping.

### 5.5.1 Reductions of Row-buffer Miss Rates

Figure 5.6 shows the row buffer miss rates of SPEC2000 programs with the four interleaving schemes: cache-line interleaving (*cacheline*), page interleaving (*page*), swapping interleaving (*swap*), and our permutation-based page interleaving (*page-xor*) schemes. The memory system contains 32 memory banks. The row-buffer size of each bank is 2KB. We use *sim-outorder* in the SimpleScalar toolset to collect the row buffer miss rate.

We have the following observations:

- All programs using cache-line interleaving have the highest row buffer miss rates compared with the other three interleaving schemes. The average miss rate is 88.7%. Since the cache-line interleaving is normally associated with the close-page mode, its high row-buffer miss rates do not necessarily mean poor overall performance. The



**Figure 5.6:** Row buffer miss rates for different interleaving schemes when the number of banks is 32, and the row buffer size is 2KB. *Cacheline* represents cache-line interleaving, *page* represents conventional page interleaving, *swap* represents the swap scheme, and *page-xor* represents the permutation-based page interleaving.

other schemes are used with the open-page mode, where the high miss rates do mean poor performance.

- All programs using page interleaving have lower miss rates than those using cache-line interleaving. However, the miss rates are still very high. The average miss rate (arithmetic mean) is 58.6%. Only one program has a miss rate less than 30.0%.
- The swapping scheme may reduce the row-buffer miss rates for some programs but increase the miss rates for others. The average miss rate is 66.3%, higher than that of the page interleaving scheme. The swapping scheme could make programs exploit

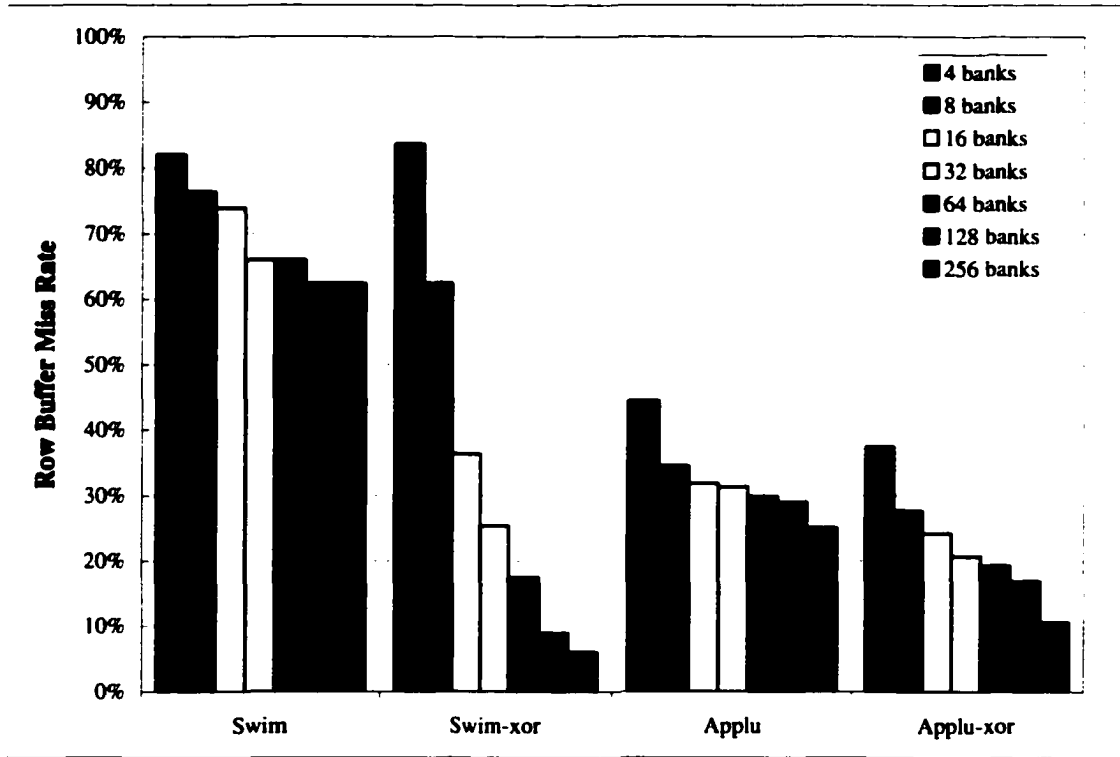
less locality than page interleaving, as we have discussed in Section 5.3.

- For almost all programs, our permutation-based interleaving scheme obtains the lowest row-buffer miss rates compared with the other three interleaving schemes. The only exception is *178.galgel*, whose miss rate is slightly higher than that using the swapping scheme. The average miss rate is 26.8%. Six programs have miss rates less than 15.0%.

### 5.5.2 Effects of Memory Organization Variations

Changing the number of memory banks and the row-buffer size of each memory bank, we have evaluated the effects of memory system organization variations on the interleaving schemes and on memory performance. We use the programs *171.swim* and *173.applu* as examples, which is memory intensive and well representative for the group of workloads. Figure 5.7 and Figure 5.8 show how the row-buffer miss rates of the two programs change under the four interleaving schemes, as the number of banks varies from 4 to 256 and the row-buffer size varies from 1 KBytes to 8 KBytes, respectively.

For each memory system variation, our experiments show that the permutation-based page interleaving scheme reduces the row-buffer miss rate effectively. Furthermore, the permutation-based scheme reduces row-buffer miss rate more closely proportional to the increase in the number of memory banks or the row buffer size than the conventional page interleaving schemes. The reason behind this fact is that the permutation-based bank index generation can widely distribute the conflicted pages among the memory banks. The larger the number of memory banks, the more effective of the permutation-based bank index generation.



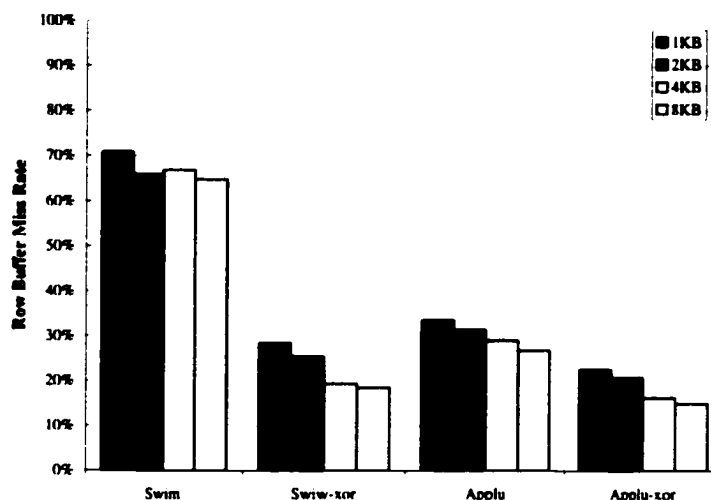
**Figure 5.7:** Row buffer miss rates of program *171.swim* and *173.applu* under conventional page interleaving scheme (without "-xor") and permutation-based page interleaving (with "-xor") as the number of bank changes from 4 to 256 with a fixed 2-KByte row buffer size.

### 5.5.3 Effects of Write Buffers

For the thirteen programs, the ratios of the number of memory writes to the number of memory reads range from 0.10 to 0.76. Using SPEC2000 programs *172.mgrid* as an example, we show the effects of write buffer<sup>2</sup> with different write policies on the row-buffer miss rates. The performance of the other workloads is similar. We have compared the following two write policies: *write after reaching threshold* (writes are issued together only when the number of writes reaches a threshold), and *write when memory is idle* (writes are scheduled

<sup>2</sup>This write buffer is located between the L2 cache and the main memory and is used to hold writebacks.





**Figure 5.8:** Row buffer miss rates of program *171.swim* and *173.applu* using the conventional page interleaving scheme (without "-xor") and the permutation-based page interleaving (with "-xor") as the row buffer size changes from 1-KBytes to 8-KBytes with fixed 32 banks.

to memory banks whenever there are no outstanding reads). We have used the latter policy through all other experiments.

Although workloads scheduled by the *write after reaching threshold* policy normally get lower row-buffer miss rates than those scheduled by the policy of *write when memory is idle*, the *write after reaching threshold* policy may cause higher total execution time due to longer memory stall time. For example, our experiments show that the program *172.mgrid* scheduled by the *write after reaching the threshold* policy has a 23% row-buffer miss rate with the page interleaving scheme, compared with a 56% row-buffer miss rate using the policy of *write when memory is idle*, however, the CPI is increased from 0.68 to 0.92. This is because buffered write requests will stall read requests when those requests are

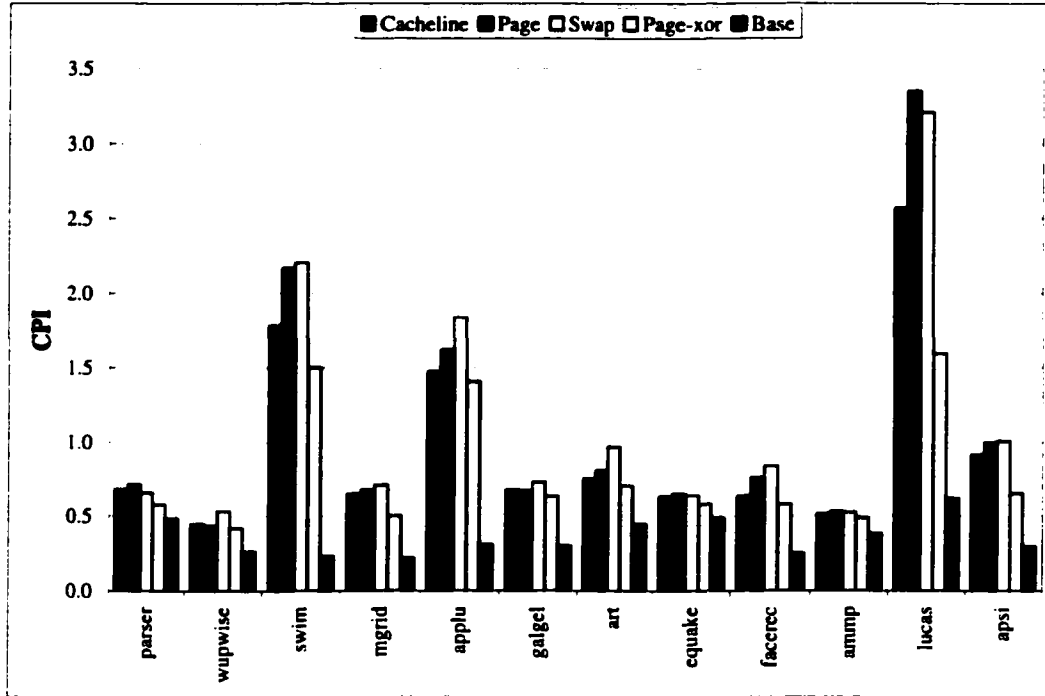
issued together, and in turn they stall the processor. For this reason, the policy of *write when memory is idle* is used for comparing the overall performance of different interleaving schemes in our study.

A major function of the write buffer is to allow memory reads to bypass memory writes so that write requests will not stall the processor. To improve bus utilization, write requests should be issued as long as the bus is idle and there is no pending read. To avoid row buffer conflicts, however, write requests should be held until no future reads will access the same pages. To design a scheduling policy to meet those two conflicting goals is difficult, and may significantly increase the size requirement for the write buffer. In contrast, using our interleaving scheme to avoid such conflicts is much simpler.

#### 5.5.4 Overall Performance Improvement

Figure 5.9 gives the CPI of twelve SPEC2000 programs (excluding *181.mcf*) using the four schemes. We exclude *181.mcf* because its CPI values are much higher than other programs which would distort the other bars. The close-page mode is used for cache line interleaving, while the open-page mode is used for the other three schemes. We also show the CPI of a base system, which is a system with an infinitely large L2 cache to eliminate all main memory accesses. The CPI of the base system provides a lower bound for any performance improvement on DRAM memory systems. We use CPI instead of IPC in order to show how much the permutation-based mapping reduces the memory stall time, which is represented by the difference between the CPI of the base scheme and those of other schemes. We will use the harmonic means of IPC to compare average performance.

Among the three mapping schemes, except our permutation-based page interleaving,



**Figure 5.9:** CPI of the twelve SPEC2000 programs using the four interleaving schemes. The number of memory banks is 32, and the row buffer size is 2KB.

the average performance of cache-line interleaving is better than the other two. This is because it uses close-page mode, and because the row buffer miss rates for the other two schemes are very high. The permutation-based page interleaving is better than the cache-line interleaving on all programs except *181.mcf*, which is not shown in Figure 5.9. The CPI values of *181.mcf* are 7.3 and 7.0 for permutation-based mapping and cache-line interleaving, respectively. This program is bandwidth-bounded. Except *181.mcf*, the permutation-based scheme outperforms all other schemes on all programs. The harmonic mean of IPC for cache-line interleaving, page interleaving, the swapping scheme, and permutation-based interleaving are 0.70, 0.61, 0.57, 0.77, respectively (including *181.mcf*). Using this metric, the average improvement of the permutation-based scheme over the cache-line scheme is

11%.

## 5.6 Breaking Mapping Symmetry at Cache Level

Researchers have studied cache mapping schemes to reduce cache conflict misses. Two representative schemes are bitwise-XOR [40] and polynomial mapping [83, 40, 95]. Those cache mapping schemes also break the address mapping symmetry but at the cache level. Thus, they may also reduce the row buffer conflicts. For this purpose, the effectiveness of those schemes is determined by how successfully they reduce the possibility that two cache-conflicting addresses are row-buffer-conflicting.

In this section, we examine cache mapping schemes aiming at reducing row buffer miss rates, and discuss the tradeoffs between using cache mapping schemes and using DRAM interleaving schemes.

### 5.6.1 Bitwise-XOR and Polynomial Mapping

In the bitwise-XOR scheme, the least significant  $s$  bits of the tag are XORed with  $s$  set index bits to form the new cache set index, where  $s$  is the number of bits in cache set index. The polynomial mapping scheme [83] uses equation  $R(x) = A(x) \bmod P(x)$  to map a given address onto a module (here a cache set), where  $R(x)$ ,  $A(x)$ ,  $P(x)$  are polynomials over the Galois Field  $\text{GF}(2)$ . In the equation,  $A(x)$  is the polynomial associated with the address to be mapped,  $R(x)$  is the polynomial associated with the cache set index, and  $P(x)$  is an irreducible polynomial of order  $s$ . The polynomial mapping is effective in avoiding conflicts for strided access patterns. It has been proven that any sub-sequence of length  $M$  within strides of form  $2^k$  will be evenly mapped onto  $M$  module, where  $k$  is a positive integer. The

bitwise-XOR scheme can be implemented using single-level XOR gates with two inputs. The polynomial mapping can be implemented using single-level XOR gates with multiple inputs.

### 5.6.2 Reduction of Miss Rates

We first compare the cache miss rates of the two cache mapping schemes with conventional cache mapping. We use those schemes only for L2 caches but not for L1 caches. For the polynomial mapping, we choose arbitrarily the polynomial  $P(x)$  associated with prime number 1572821, as there is no theory yet on “good” prime numbers used with the polynomial mapping. We have experimented with a few other randomly chosen prime numbers for a subset of the programs, and found the results are consistent with the reported ones.

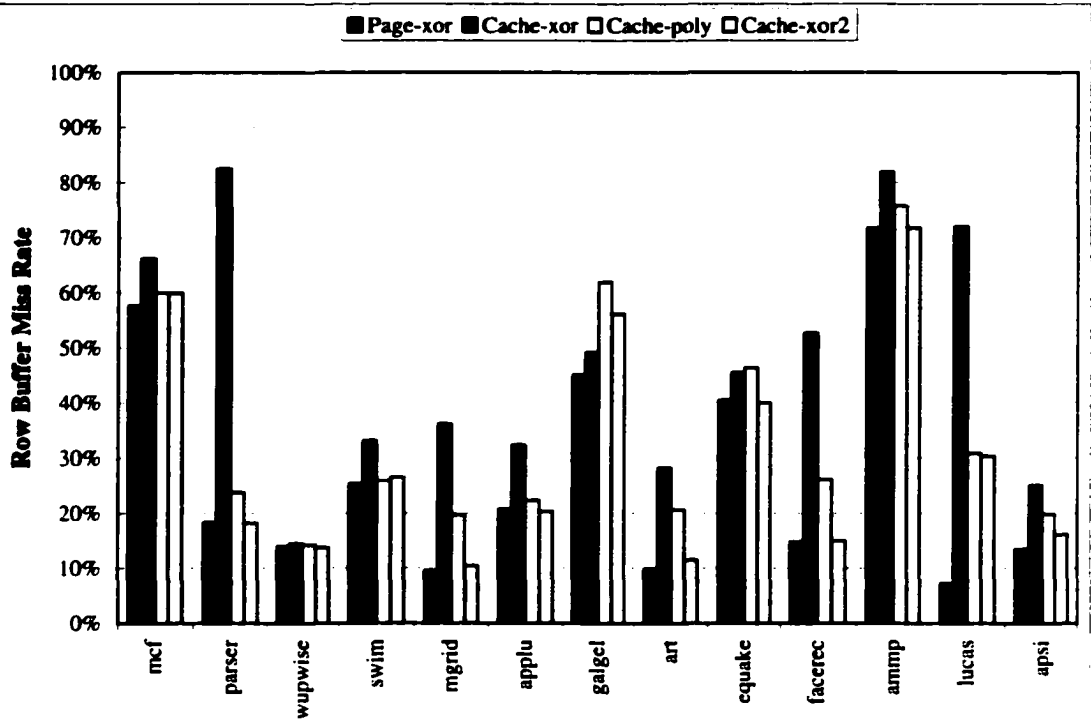
Table 5.3 shows the L2 cache miss rates for the thirteen SPEC2000 programs. The *cache-xor2* represents a revised bitwise-XOR that we will discuss soon in this section. The miss rates of the two cache mapping schemes are almost identical with the conventional mapping schemes except for program *178.galgel*, *179.art*, *188.ammmp*, and *189.lucas*. For *178.galgel*, the polynomial mapping scheme reduces the cache miss rate dramatically, but the bitwise-XOR does not. For *179.art*, both schemes increase the cache miss rates by almost two times. For *188.ammmp*, both schemes reduce the miss rates and the bitwise-XOR does better. For *189.lucas*, the bitwise-XOR increases the miss rate by almost 35%, but the polynomial mapping reduces the miss rate by more than 20%. On average, the polynomial mapping reduces the miss rate from 23.5% to 22.0%, and the bitwise-XOR increases the miss rate to 25.0%. The increase or decrease of the average miss rate is not significant, which confirms the previous studies.

Programs	Default	Cache-xor	Cache-poly	Cache-xor2
mcf	44.1%	43.2%	44.1%	44.0%
parser	8.4%	8.5%	8.6%	8.4%
wupwise	39.7%	39.7%	39.8%	39.7%
swim	27.9%	27.9%	28.0%	27.9%
mgrid	22.8%	22.8%	22.9%	22.8%
applu	37.2%	37.2%	37.1%	37.2%
galgel	18.2%	18.3%	1.7%	1.4%
art	3.2%	11.8%	11.5%	12.0%
quake	10.6%	10.6%	10.8%	10.6%
facerec	22.7%	22.3%	23.0%	22.8%
ammp	6.5%	2.9%	3.9%	7.0%
lucas	43.5%	58.6%	33.3%	33.3%
apsi	21.2%	21.2%	21.2%	21.2%
Average	23.5%	24.0%	22.0%	22.2%

**Table 5.3:** L2 cache miss rates for the conventional cache mapping (default), the bitwise-XOR (cache-xor), the polynomial mapping (cache-poly), and the revised bitwise-XOR (cache-xor2).

We show the row-buffer miss rates of the two cache mapping schemes in Figure 5.10. We also include the row-buffer miss rates of the permutation-based DRAM page interleaving for comparison, where the conventional cache mapping is used. The bitwise-XOR scheme has the highest row buffer miss rates for all programs. If we compare it with the conventional DRAM mapping (in Figure 5.6), we will find this scheme only moderately reduces the row buffer miss rate. The row buffer miss rates using the polynomial mapping are close to those of the permutation-based DRAM page interleaving. However, the latter one is still better for all applications. The average row buffer miss rates are 47.6%, 34.4%, and 26.8% for the bitwise-XOR cache mapping, the polynomial cache mapping, and the permutation-based DRAM mapping, respectively.

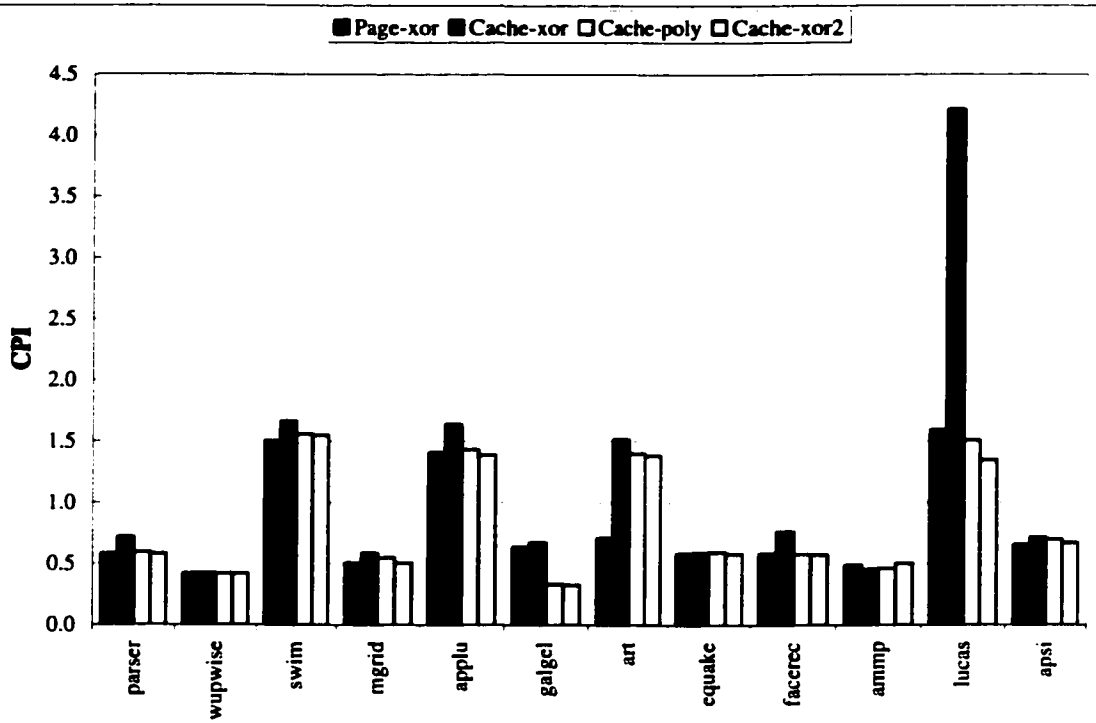
Here is why the bitwise-XOR scheme results in high row buffer miss rates. In this scheme, the  $k$  tag bits that are XORed with the  $k$  bank index bits are not the least significant  $k$  bits



**Figure 5.10:** Row-buffer miss rates for the permutation-based mapping (page-xor), the bitwise-XOR mapping (cache-xor), the polynomial mapping (cache-poly), and the revised bitwise-XOR (cache-xor2).

in the tag. In the program memory space, cache-conflicting addresses that differ only in the least significant  $k$  bits have a shorter distance than other cache-conflicting addresses. Thus, because of program locality, the least significant  $k$  bits change more frequently than other bits in the access stream generated by a program. Under this scheme, the least significant  $k$  bits are XORed with bits for selecting DRAM page offset, and the  $k$  bits XORed with the bank index changes less frequently. Consequently, from the DRAM point of view, two cache-conflicting addresses appearing within a short time frame are likely to have the same bank index, causing conflicts at the row buffer.

To confirm this, we switch the two portions of the tag bits such that the least significant



**Figure 5.11:** CPI of twelve SPEC2000 programs using the permutation-based DRAM mapping (*page-xor*), the bitwise-XOR cache mapping (*cache-xor*), the polynomial cache mapping (*cache-poly*), and the revised bitwise-XOR cache mapping (*cache-xor2*).

$k$  bits are XORED with the bits for selecting the bank index. This revised bitwise-XOR is labeled as *cache-xor2*. The new scheme significantly reduces the row buffer miss rates, and is slightly better than polynomial cache mapping. The average row buffer miss rate is 30.0%. The cache miss rates of this scheme are also shown in Table 5.3.

### 5.6.3 Comparisons of Overall Performance

Figure 5.11 shows the CPI of the programs (again *181.mcf* is excluded). For the cache mapping schemes, we do not consider in the simulation the possible delay of a critical path by using the mapping. The bitwise-XOR cache mapping has the worst performance for



most programs because of the severe row buffer conflicts and the slightly higher average cache miss rate compared with other schemes. Its performance for *189.lucas* is worse than the other schemes because the other schemes have both lower cache miss rate and lower row buffer miss rate. The permutation-based DRAM mapping has the best performance for most programs because it has the lowest row buffer miss rates. It is much better than the others for *179.art* because the other three schemes increase the cache miss rate. However, it is worse than the polynomial mapping and the revised bitwise-XOR for *178.galgel* because the two cache mapping schemes reduce the cache miss rate dramatically. When being successful in reducing cache miss rates, the polynomial cache mapping and the revised bitwise-XOR cache mapping perform better than the permutation-based DRAM mapping. Otherwise, the permutation-based DRAM mapping performs slightly better than the revised bitwise-XOR cache mapping, and the latter one performs slightly better than the polynomial cache mapping, because of the difference in row buffer miss rates.

The harmonic means of IPC are 0.77, 0.60, 0.74, and 0.75 for the permutation-based DRAM mapping, the bitwise-XOR cache mapping, the polynomial cache mapping, and the revised bitwise-XOR cache mapping, respectively. The result indicates that the advantage of the permutation-based DRAM mapping in reducing the row buffer miss rate is so effective that its disadvantage of not considering of cache miss reduction becomes insignificant.

#### 5.6.4 Tradeoffs between Cache Mapping Schemes and DRAM Interleaving Schemes

As discussed in the previous subsection, using different cache mapping schemes (for example, the polynomial one or the revised bitwise-XOR) may significantly reduce the row buffer

miss rate. However, our performance results have shown that the average performance improvement from reducing cache conflict misses is insignificant. Compared with conventional cache mapping and DRAM interleaving, almost the entire overall performance gain comes from the reduction of row buffer conflict misses. Nevertheless, using a polynomial cache mapping scheme may have the advantage of good predictability of cache behavior [95]. If the predictability of cache behavior is important, cache mapping schemes like the polynomial mapping are attractive because they can reduce conflicts at both cache and row buffer levels.

However, the implementations of these cache mapping schemes are nontrivial [95]. They should not increase the delay in the critical path. When multiple-level caches maintain the property of inclusion, i.e., the data cached at a higher level must be cached at the lower level, it is necessary to enforce explicit invalidation in the higher level cache when a cache block in the lower level cache is replaced. Although those issues are addressable, the solutions do increase the complexity of the processor core. In comparison, the permutation-based page interleaving scheme does not have such implementation concerns, and the logic is implemented outside the processor core. In short, the scheme is much more cost-effective by considering both the significant performance gain and its simplicity.

## 5.7 Considerations of Large Cumulative Row Buffer Sizes

All of our analyses of row buffer conflicts so far have been based on the large-cache condition, which is normally realistic. However, if the memory size is very large, the cumulative row buffer size may be larger than the cache size divided by the cache associativity. In this

section, we examine a memory size threshold for the large-cache condition to hold, and investigate how the increase of memory size beyond the threshold will affect the effectiveness of the permutation-based scheme.

Assume  $W$  is the value of the cache size divided by the cache associativity. In a DRAM memory system, the ratio of DRAM capacity to the row buffer size is usually a constant  $R$  for all DRAM chips. Thus, the ratio of the memory size,  $m$ , to the row buffer size is  $R$ . The product of  $W$  and  $R$ , denote as  $M$ , is a threshold for  $m$ . The large-cache condition holds when and only when  $m \leq M$ .  $R$  is large in practice, for example, 8192 for today's 256Mbit SDRAM chips. In other words, the threshold  $M$  is 8 GBytes with a 2-way set associative, 2-MByte L2 cache and a DRAM system with such chips.

When  $m > M$ , we are specially interested in the cases when  $m$  is a small multiple of  $M$ , for example,  $2M$ ,  $4M$ , or  $8M$ , for practical reason. As  $m$  increases, eventually the row buffer miss rate will be close to zero even under the conventional DRAM mapping. However, this requires a very large memory size. The advantage of the permutation-based scheme is still obvious when  $m/M$  is small. Under the conventional DRAM mapping, any two cache-conflicting addresses can now be distributed to  $m/M$  row buffers instead of one row buffer (assume  $m/M$  is less than the number of row buffers). Thus, the increase of  $m/M$  will reduce the row buffer miss rate. However, the permutation-based scheme can distribute those addresses onto all row buffers, whose number can be much larger than  $m/M$  in practice.

When  $m > M$ , the tag bits and the  $k$  bits of bank index are partially overlapped. We slightly change the permutation-based scheme as follows: instead of using the least significant  $k$  bits in the tag for XORing, we use the least significant  $k$  bits in the tag portion

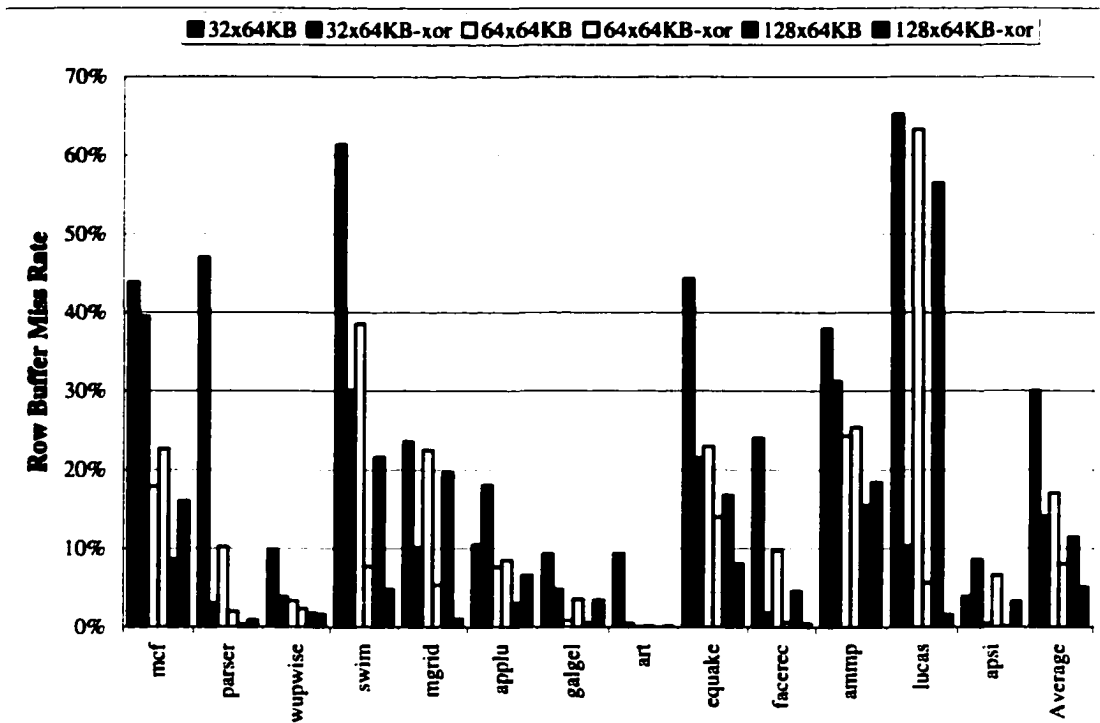
that are not overlapped with the bank index. This is necessary to guarantee the correctness of the scheme because XORing overlapping bits will make the scheme lose the one-to-one property.

Figure 5.12 shows the row buffer miss rates for all the thirteen programs for different numbers of banks and a fixed row buffer size, each with the conventional page interleaving and with the permutation-based page interleaving. The number of banks is 32, 64, or 128, and the row buffer size is 64KB. The  $W$  here is 1MB, and the cumulative row buffer size is 2, 4, and 8 times  $W$ , respectively. With  $r = 8192$ , the memory size threshold  $M$  is 16 GBytes, 32 GBytes, and 64 GBytes respectively. With the conventional page interleaving, the average row-buffer miss rates are 30.0%, 14.1%, and 17.1%, respectively. With the permutation-based page interleaving, the average row-buffer miss rates are 14.5%, 8.1%, and 5.1%, respectively. In summary, using the permutation-based page interleaving is still effective in improving the application performance even when the memory size is beyond the threshold.

Using memory access scheduling techniques to exploit row-buffer locality and concurrency is another attractive approach (e.g. [90]). We believe the combination of access scheduling and the permutation-based interleaving scheme can further improve memory performance.

## 5.8 Other Related Work

Hsu and Smith propose and evaluate several memory interleaving schemes that can both increase data locality and avoid generating hot banks in vector supercomputers with cached



**Figure 5.12:** The row buffer miss rates of conventional and permutation-based page interleaving schemes when the cumulative row buffer size is larger than cache size divided by cache associativity. In the legend, 32, 64, or 128 before the “x” represents the number of bank, 64KB represents the size of the row buffer, and the “xor” indicates using the permutation-based page interleaving.

DRAM [49], where processors do not have data caches. Our study targets superscalar processors with DRAM memory systems. The large caches in our targeted systems make the memory access patterns significantly different from those in the vector system without caches.

There are several other research papers dealing with the bank conflict problem of vector accesses in vector supercomputers. Authors in [37] and [96] attempt to use prime memory systems to address the conflict issues. Other papers focus on memory interleaving schemes on vector systems [19, 84, 93, 99, 104, 114]. Authors in [42], [19], and [93] study skew

schemes. Rau, Schlansker, and Yen propose a pseudo-random interleaving technique using the XOR function to randomize the mapping of references to memory modules in [84]. Their scheme can eliminate the occurrence of long clusters due to structured data access. Sohi studies permutation-based interleaving schemes that can improve memory bandwidth for a wide range of access patterns for vector computers [104]. Valero, Lang, and Ayguadé [114] divide the memory address into several portions according to the width of bank index, then XOR all the address portions to generate the bank index. Their method can avoid bank conflict due to power-of-two strides in vector machines. Seznec and Lenfant [99] propose the Interleaved Parallel Scheme, which uses the XOR operation and parameters related to the numbers of processors, logical memory banks, and physical memory banks to induce more equitable distribution over memory banks for a wider set of vectors than the normal mappings.

The above cited studies are based on vector supercomputers with SRAM memory systems. Besides different memory access patterns on those machines, the sources of access conflicts in our targeted systems are also different from those in the vector machines without DRAM memory systems. For example, elimination of DRAM row buffer conflicts without reducing the available locality is a major issue in our study. Therefore, our study has a different objective with a different focus.

Besides memory bank interleaving techniques, there are other approaches to address the memory latency problem, such as a blocking-free cache, prefetching, thread changing, and data prediction and speculation.

## 5.9 Conclusion

We have shown that the address mapping symmetry is the inherent source of row buffer conflicts under conventional cache and DRAM address mapping. Breaking the mapping symmetry, the proposed permutation-based page interleaving scheme can eliminate or significantly reduce severe row buffer conflicts and retain spatial locality. Conventional schemes, such as cache-line and page interleaving, can not effectively exploit both DRAM concurrency and spatial locality in the row buffer. Our execution-driven simulations show that the permutation-based scheme can significantly reduce row buffer miss rates and improve the overall performance.

We have also shown that mapping symmetry can be broken at the cache-level to remove this source of row buffer conflicts. We have evaluated two representative cache mapping schemes, bitwise-XOR and polynomial mapping, which are proposed originally for avoiding cache conflict misses. The polynomial mapping can reduce the row buffer miss rate close to that of the permutation-based page interleaving, but the bitwise-XOR must be modified to avoid conflicts. Our results indicate that, conflict-avoiding cache mapping schemes should also consider the conflicts at the row buffer. We show that almost all performance gains come from reductions of row buffer miss rates, and the permutation-based page interleaving scheme has the best overall performance. Considering the scheme does not increase the complexity of processor core, it is also the most cost-effective approach.

In Table 5.4, we give a summary of the three cache mapping and memory interleaving schemes, namely, the bitwise-XOR cache mapping, the polynomial cache mapping, and our permutation-based page interleaving scheme. We present their impacts on three aspects

Mapping Scheme	Cache conflict reduction	Row-buffer conflict reduction	Overall performance improvement	implementation complexity
Bitwise-XOR cache mapping	moderate	low	low	moderate
Polynomial cache mapping	moderate	high	high	high
Permutation-based DRAM page interleaving	N/A	highest	highest	low

**Table 5.4:** Summary of the three cache mapping and memory interleaving schemes, and their impact on three aspects of performance, and on implementation complexity.

of performance, namely, cache conflict reduction, row-buffer conflict reduction, and overall performance improvement, and their impacts on increasing implementation complexity. Our study shows that the permutation-based page interleaving scheme outperforms all other schemes based on its performance improvement and on its implementation simplicity.



## Chapter 6

# Cached DRAM: A Simple and Effective Technique for Memory Access Latency Reduction on ILP Processors

In the previous study on the permutation-based page interleaving scheme, we have used the existing DRAM row buffers to reduce DRAM access latency. In this chapter, we further study a hardware method to enhance the DRAM row buffer with a small on-memory SRAM cache, which is integrated into each DRAM chip and is attached to the DRAM banks. We call it an *on-memory cache*. Like the previous method, this approach exploits the spatial locality in cache miss streams. The SRAM cache outperforms the existing row buffers for two reasons. First, the access time of SRAM cache is shorter than that of the row buffer. With the current manufacturing technology, the hit time of the on-memory cache is roughly half of the row access time. Second, the on-chip cache can be set-associative with DRAM

banks. In contrast, the row buffer is always direct-mapped with DRAM banks. Thus, the on-memory cache is likely to have lower miss rates than the row buffers. Third, the on-memory cache can be accessed independently from DRAM core. Even if a DRAM bank is in precharge or in a row access, its cached data in the SRAM can be accessed simultaneously. Because of those advantages, the on-memory cache can reduce more accesses to the DRAM core and can return cached data faster than the row buffer.

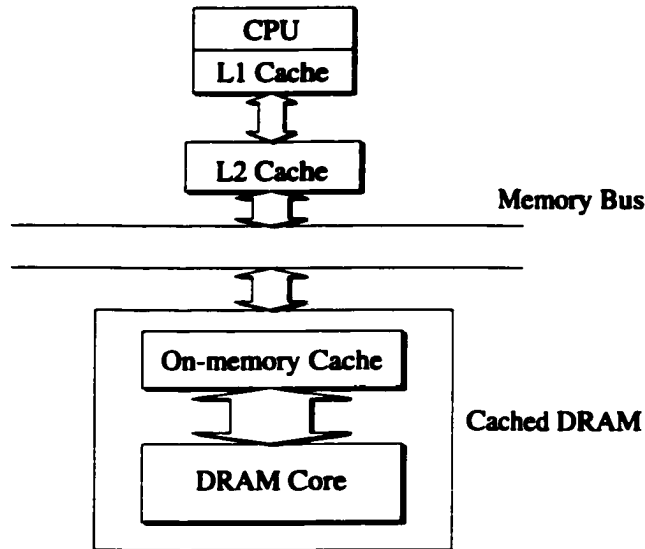
Hsu and Smith [49] classify cached DRAM organizations into two groups: (1) the on-memory cache contains only a single large line buffering an entire row of the memory array; and (2) the on-memory cache contains multiple regular data cache lines that are organized as direct mapped or set associative structures. A combination of (1) and (2) forms the third class: the on-memory cache contains multiple large cache lines buffering multiple rows of the memory array, which are organized as direct mapped or set associative structures. Our work and some other related studies (e.g., [61] and [118]) belong to the third class.

Previous studies [46, 43, 61, 118] have shown that cached DRAM can improve memory access efficiency for technical workloads on a relatively simple processor model with small data caches (and in some cases, even without data caches). In a modern computer system, the CPU is a complex ILP processor, and caches are hierarchical and large. Thus, the architectural context of a cached DRAM has dramatically changed and evolved. Koganti [62] investigated the performance potential of cached DRAM in systems with ILP processors, and found it is effective as well. Aiming at further investigating the ILP effects and comparing cached DRAM with other advanced DRAM organizations and interleaving techniques, we present a study of cached DRAM in the context of processors with full ILP capabilities and large data caches.

We have compared cached DRAM with several commercially available DRAM schemes: SDRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM. We find that a cached DRAM has limited benefits for SPECint95 programs, but can significantly improve the performance of most SPECfp95 programs and the TPC-C workload, which are more data-intensive. A small on-memory cache of  $16 \times 4$  KBytes can reduce the execution time by 10% to 39% for eight selected SPECfp95 programs and 14% for the TPC-C workload, respectively. The comparison is done with a conventional four-bank SDRAM. Our results also show that the cached DRAM outperforms other DRAM architectures for these applications.

Our study provides three new findings: (1) cached DRAM consistently shows its performance advantage as the ILP degree increases; (2) other contemporary DRAM schemes, such as SDRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM, do not exploit memory access locality of data-intensive workloads as effectively as a cached DRAM does; and (3) compared with a highly effective permutation-based DRAM interleaving technique, cached DRAM can still gain substantial performance improvement.

The rest of this chapter is organized as follows. Section 6.1 describes the structure and the operations of cached DRAM. Section 6.2 describes the experiment environment. Section 6.3 presents the performance of cached DRAM with a 4-way issue processor. Section 6.4 presents the performance of cached DRAM as the ILP degree of the processor increases. Section 6.5 compares cached DRAM with the previous approach that exploits the row buffer locality. Finally, Section 6.6 summarizes the study.



**Figure 6.1:** General concept of cached DRAM.

## 6.1 Structure and Operations of Cached DRAM

**Cached DRAM** is an existing technology that adds a small SRAM cache onto the DRAM core. It takes advantage of the huge internal bandwidth existing in the DRAM core, so that the cache block can be as large as a page. The processor usually has large-size caches but the cache block is much smaller because of the limited bandwidth between the processor and main memory. In general, a small cache with large block size can get a miss rate comparable to that of a large cache with a small block size [43]. In addition, the cached DRAM has a higher hit rate of its cache than other DRAM architectures because of its fully-associative cache organization. Figure 6.1 shows the general concept of cached DRAM.

Contemporary processors aggressively exploit instruction-level parallelism by using su-

perscalar, out-of-order execution, branch prediction, and non-blocking caches. As a result, the processor may issue multiple memory requests before the previous request is finished. Although the processor can keep running before the memory requests are finished, its ability to tolerate long memory latency is limited.

A cached DRAM is an integrated memory unit consisting of an on-memory cache and a DRAM core. Inside the cached DRAM, the on-memory cache is connected to the DRAM core by a wide internal bus for moving data between the cache and the DRAM core. Cached DRAM can give fast response for a single memory request, and pipeline multiple requests to achieve a high throughput.

The processor sends memory requests to the memory controller when an L2 cache miss happens. In order to take advantage of the low access latency of the on-memory cache in cached DRAM, the memory controller maintains the tag of the on-memory cache, and compares each tag with the tag portion of the address for every memory request. The memory controller also maintains a dirty flag for each on-memory cache block, which indicates whether the block has been modified after it is loaded from the DRAM core.

For the on-memory cache, a memory request is either (1) a read hit, (2) a write hit, (3) a read miss, or (4) a write miss. Each case is handled by memory controller as follows:

1. **Read Hit:** The memory controller sends to the on-memory cache a read command along with the block index and the column address via the address/command bus, which will arrive at the cached DRAM after one bus cycle. The row address is not needed because the DRAM core is not accessed. In one bus cycle, the on-memory cache outputs the data which is sent back to the memory controller/processor after

Bus cycle	0	1	2	3	4	5	6	7	8
Address/command bus	R1		R2		R3				
Cached DRAM data			D1	D1	D2	D2	D3	D3	
Processor data				D1	D1	D2	D2	D3	D3

**Table 6.1:** An example of pipelining three continual read hits. An “R1” on the “Address/command bus” indicates the block index and the column address of the first read is sent on the address/command bus. A “D1” on the “Cached DRAM data” indicates a block of data for the first read is available in the cached DRAM. A “D1” on the “Processor data” means a block of data for the first read is available for the processor. “R2” and “R3” in the table correspond to the second and the third read commands/addresses, respectively; and “D2” and “D3” correspond to the data items for the second read and the third read, respectively.

another bus cycle. If the memory controller receives consecutive memory requests that are read hits, it processes them in a pipelined mode.

2. **Read Miss:** There are two separate processing steps. In the first step, the row in the DRAM core that contains the data is read and transferred to the on-memory cache. In the second step, the data is read from the on-memory cache as if it were a read hit. For the first step, the memory controller sends the DRAM a read command along with the row address and the replaced block index on the command/address bus. This activates the row access of the DRAM core. The memory controller uses a modified LRU policy (which will be discussed in detail later) to find the block for a replacement. The second step of processing is the same as that of a read hit. A read miss can be overlapped with read hits.
3. **Write Hit:** The write-back policy is enhanced and the data will only be written into the on-memory cache. The memory controller sends the write command along with the block index and the column address on the address/command bus, and sends the data on the data bus. At the same time, the dirty flag is set for that block on the

Bus cycle	0	1	2	3	4	5	6	7	8	9
Address/command bus	R1	R2		R1		R3				
Cached DRAM data				D2	D2	D1	D1	D3	D3	
Processor data					D2	D2	D1	D1	D3	D3

**Table 6.2:** An example of pipelining a read miss (R1) and two read hits (R2 and R3). The first “R1” in the “Address/command bus” indicates that the DRAM read command along with the row address and the block index are sent on the address/command bus. The second “R1” indicates that a cache read command along with the column address and the block index are sent; “R2” and “R3” are signals of the two read hits on the bus. “D1”, “D2”, and “D3” on the “Cached DRAM data” indicate the data for the reads are available on the cached DRAM. “D1”, “D2”, and “D3” on the “Processor data” mean the data for these reads are available for the processor.

memory controller. The row address is not needed because the DRAM core is not accessed. The processing of a sequence of write hits can overlap with each other, and can overlap with read requests.

4. **Write Miss:** The memory controller uses the modified LRU policy to select a block for replacement. The write-allocate policy is enhanced with two steps as follows. In the first step, the memory controller sends the DRAM read command along with the block index and the row address onto the address/command bus. Then the row in the DRAM core that contains the writing address is first read from the DRAM core and transferred to the on-memory cache. The second step is to write the data into the on-memory cache, operated as a write hit. These two steps can overlap with other read or write requests.

Table 6.1 shows an example of pipelining three continual read hits. Table 6.2 shows an example of a read miss that is overlapped with two read hits. The pipelining operations of write hits and write misses are similar.

The replacement policy of on-memory cache is a modified LRU policy that avoids choos-

ing a dirty block for replacement. If a dirty block were chosen for replacement, the block would have to be written back into the DRAM core first, which would increase the latency of the memory request that causes the replacement. To increase the number of clean blocks available for replacement, the memory controller will schedule write back requests for the dirty cache blocks as soon as the DRAM core is not busy.

ILP processors do not stall for a single L2 miss so they may issue more memory requests when a previous request is in processing. An outstanding read request may prevent dependent instructions in the instruction window from being issued to execution units, which is likely to reduce the instruction-level parallelism or make the instruction window full. On the other hand, outstanding writes do not influence ILP processors as long as the write buffer is not full. Therefore, the memory controller should schedule read requests prior to write requests.

As we have discussed, the processing of read misses or write misses consists of two steps: DRAM operations that move data between the on-memory cache and the DRAM core, and on-memory cache accesses, which do not involve the DRAM core. Because the DRAM operations are slow, they should be issued as soon as possible.

## 6.2 Experimental Environment

We use SPEC95 and TPC-C as the workloads, and use SimpleScalar [14] as the base simulator. The database system used to support the TPC-C workload is the PostgreSQL (version 6.5). This is the most advanced open source database system for basic research. We run the complete set of SPECint95 and SPECfp95 in our experiment, using the precompiled



SPEC95 programs in the SimpleScalar tool set.

We have used *sim-outorder* to configure an 8-way processor, set the load/store queue size to 32, and set the register update unit size to 64 in the simulation. The processor allows up to 8 outstanding memory requests, and the memory controller can accept up to 8 concurrent memory requests. Table 6.3 gives other architectural parameters. The 500 MHz processor and the 256-bit (32 Bytes), 83 MHz data bus are used in Compaq Workstation XP1000. The on-memory cache access time is set to 12 ns which is the same as that in [43]. The on-memory cache hit time is the sum of the time for transferring the command/address to the cached DRAM (1 bus cycle), the on-memory cache access time, and the time for the first chunk of data to be sent back (1 bus cycle). The on-memory cache miss time is the sum of the time for transferring the command/address to the cached DRAM, the DRAM precharge time if the accessed memory bank needs precharge, the DRAM row access time, the time to transfer a row from the DRAM core to the on-memory cache (1 bus cycle), the on-memory cache access time, and the time for the first chunk of data to be sent back.

The Rambus DRAM is connected to the processor by a one-byte wide, high-speed bus. The Direct Rambus DRAM is connected to the processor by a one-byte wide address bus and a two-byte wide data bus, and the bus speed is 400MHz. Data is transferred on both edges of the block signal. For single channel Direct Rambus DRAM, the effective bandwidth is 1.6 GBytes/second, which is not so large as the 2.6 GBytes/second bandwidth of the bus used in our simulation. To make a fair comparison, we simulate the internal structure of the Rambus DRAM and the Direct Rambus, but set their bus simulation the same as other DRAMs. We will show that the advantage of the cached DRAM is on its on-memory cache structure, not on its bus connection. In fact, the cached DRAM could also be connected to

CPU Clock rate	500 MHz
L1 inst. cache	32 KBytes, 2-way, 32-byte block
L1 data cache	32 KBytes, 2-way, 32-byte block
L1 cache hit time	6 ns
L2 cache	2 MBytes, 2-way, 64-byte block
L2 cache hit time	24 ns
memory bus width	32 Bytes
memory bus clock rate	83 MHz
on-memory cache block number	1-128
on-memory cache block size	2-8 KBytes
on-memory cache associativity	1-full
on-memory cache access time	12 ns
on-memory cache hit time	36 ns
on-memory cache miss time	84 ns
DRAM precharge time	36 ns
DRAM row access time	36 ns
DRAM column access time	24 ns

**Table 6.3:** Architectural parameters of simulation

the processor by a high-speed narrow bus.

### 6.3 Comparisons of Overall Performance

We have measured the memory access portion of CPIs of the TPC-C workload and all the SPEC95 programs. In order to show the memory stall portion in each benchmark program, we used a method similar to the one presented in [13] and [25]. We simulated a system with an infinitely large L2 cache to eliminate all memory accesses. The application execution time on this system is called the base execution time. We also simulated a system with a perfect memory bus as wide as the L2 cache line, which is used to separate out the memory stall portion due to the limited bandwidth. The CPI is divided into three portions: the *base* portion which is the number of cycles spent for CPU operations and cache accesses,

the *latency* portion which is the number of cycles spent for accessing the main memory, and the *bandwidth* portion which is the number of cycles lost due to the limited bus bandwidth. The memory access portion of the CPI is the sum of the latency portion and the bandwidth portion.

We have compared the cached DRAM with four DRAM architectures: SDRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM. We use the TPC-C workload and eight SPECfp95 programs: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu*, *turb3d*, and *wave5*. We found that the memory access portions of the CPI of all SPECint95 programs and the two other SPECfp95 programs are very small. As a result, the performance of these programs is not sensitive to the improvement of the main memory system. Although the memory access time reduction by using cached DRAM is also significant on those programs, it is not well rewarded in the overall performance.

### 6.3.1 On-memory Cache Organizations

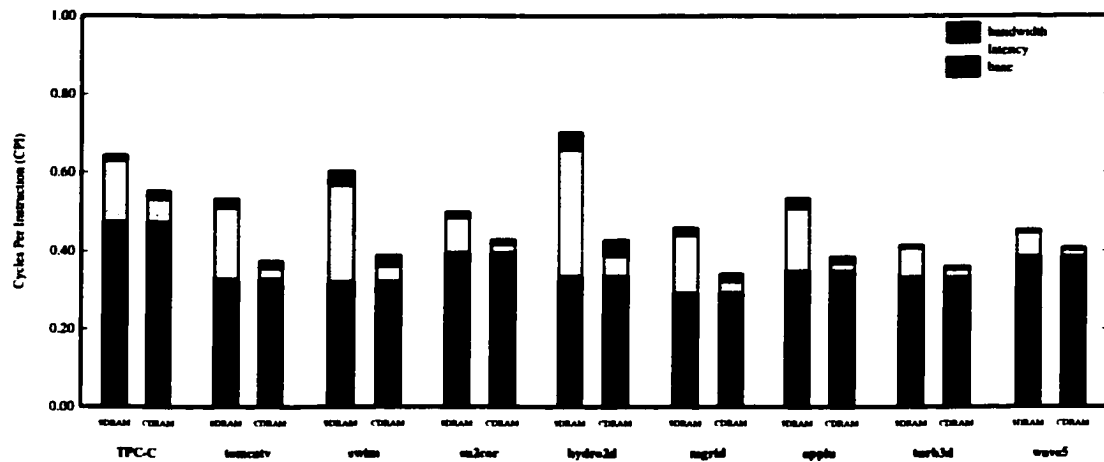
We have investigated the effects of changing the cache size and the cache associativity on the performance of the TPC-C workload and the eight selected SPECfp95 programs. Our experiments show that a small cache block size is not effective for the on-memory cache. The miss rates for TPC-C workload on a fully associative on-memory cache of 32 KBytes with cache block size of 128 Bytes, 256 Bytes, and 512 Bytes are 62%, 36%, and 22%, respectively. On the other hand, a small number of blocks are also not effective for the on-memory cache. The miss rate for *su2cor* on a fully associative on-memory cache having four blocks with size of 4 KBytes is as high as 80%. When the number of cache blocks increases to eight, the miss rate is still more than 40%. Only after the number of cache

blocks increases to sixteen, the miss rate is effectively reduced to 5%. The experiments also show that the advantage of using a full associativity is significant. Direct mapped on-memory caches, even with a large number of blocks, still have high miss rates. The finding is important because most commercial DRAMs use the direct mapped structure. Our study of the effects of cache size and associativity on the on-memory cache performance confirms the results reported in [62]. Since increasing the block size and the number of blocks will increase the space requirement of the on-memory cache on the memory chip, it is a tradeoff between performance and cost. We find that a fully associative on-memory cache of  $16 \times 4$  KBytes is very effective for all workloads. The on-memory cache miss rates of the TPC-C workload and six SPECfp95 programs are below 5%, and the miss rates of two other programs are below 20%. Thus, this on-memory cache configuration is used in the rest of the experiments.

### 6.3.2 Performance Improvement of Cached DRAM over SDRAM

Figure 6.2 presents the CPIs and their decompositions for the TPC-C workload and the eight SPECfp95 programs on both the cached DRAM and the SDRAM. The CPI reductions by using the cached DRAM range from 10% to 39%. The effectiveness of the cached DRAM for reducing the CPI is mainly determined by the percentage of memory access portion in the total CPI of each program, which is listed in Table 6.4. We show that the CPI reduction by the cached DRAM mainly comes from the reduction of the latency portion of the CPI, and the bandwidth portion of the CPI is almost unchanged in each program.

Table 6.5 further presents the reductions of the latency portions of the CPIs by the cached DRAM. For all the selected SPECfp95 programs, the latency portions of the CPIs



**Figure 6.2:** The CPIs of the TPC-C workload and the selected SPECfp95 programs

Program	TPC-C	tomcatv	swim	su2cor	hydro2d
Memory portion	27%	39%	47%	21%	52%
Program	mgrid	applu	turb3d	wave5	
Memory portion	37%	35%	20%	15%	

**Table 6.4:** The percentage of memory access portion in the CPI

are reduced by more than 71%. The reduction rate for the TPC-C workload is 62%. The latency reduction rates of the programs are mainly determined by the hit rates of the on-memory cache in the cached DRAM, and of the row buffer in the SDRAM. For example, the on-memory cache hit rate of the cached DRAM for the TPC-C workload is 93%, while the row-buffer hit rate of the SDRAM is 50%. For the case of *tomcatv*, the on-memory hit rate of the cached DRAM is 98%, while the row-buffer hit rate of the SDRAM is as low as 8%.

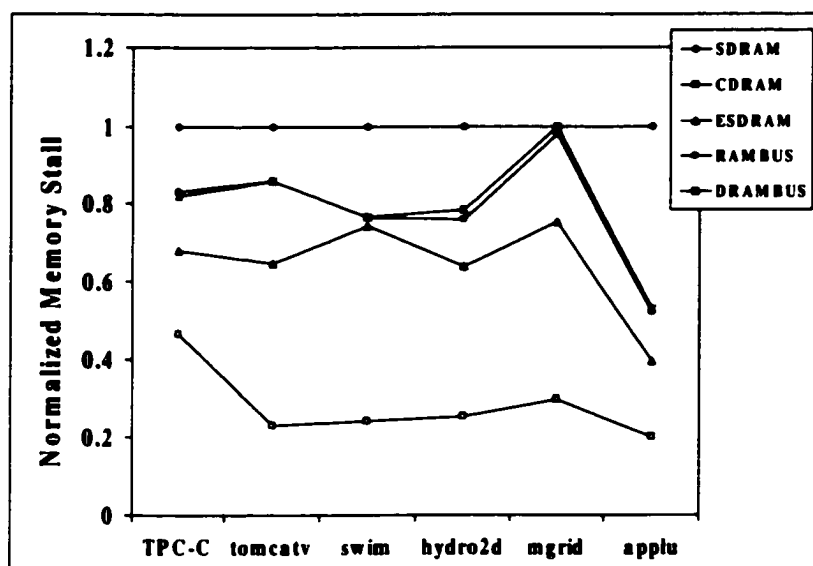
Program	TPC-C	tomcatv	swim	su2cor	hydro2d
Reduction	62%	84%	83%	75%	83%
Program	mgrid	applu	turb3d	wave5	
Reduction	79%	87%	71%	72%	

**Table 6.5:** The reduction of the latency portion of CPI by using the cached DRAM

### 6.3.3 Performance Comparisons of Cached DRAM and Other DRAM Architectures

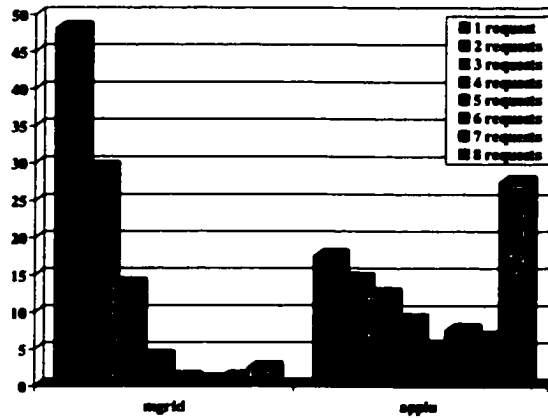
Figure 6.3 shows the memory access portions of the CPIs of the TPC-C workload and five selected SPECfp95 programs on the cached DRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM. The performance values are normalized to the memory access portion of the CPI of the SDRAM. The cached DRAM outperforms other DRAMs significantly on all programs. The Enhanced SDRAM performs better than the Rambus DRAM and the Direct Rambus DRAM because of the low latency accessing the on-memory cache. The cached DRAM outperforms the Enhanced SDRAM because the large number of blocks and the fully associate structure in the cached DRAM make the hit rate very high.

The Rambus DRAM and the Direct Rambus DRAM have the ability to support high bandwidth by overlapping accesses among different banks. However, this is little help in reducing the access latency. Although both the Rambus DRAM and the Direct Rambus DRAM have a large number of row buffers, the hit rates are still low because of the direct mapped structure. In contrast, when the number of accesses to the DRAM core is very low, the cached DRAM acts almost as an SRAM main memory, providing both low latency and high bandwidth. As a result, the performance differences between the cached DRAM and the other DRAM architectures are large.



**Figure 6.3:** The comparisons of memory stall time of cached DRAM, Enhanced SDRAM, Rambus DRAM, and Direct Rambus DRAM. The memory stall times are normalized by the stall time of SDRAM.

We show that the performance of some programs on the Rambus DRAM or the Direct Rambus DRAM can be significantly different (for the programs shown in Figure 6.3, the worst one is *mgrid*, and the best one is *applu*). Both DRAMs perform better for programs with a large number of concurrent memory requests because they can effectively overlap these memory accesses. Figure 6.4 compares the distributions of concurrent memory requests of *mgrid* and *applu* on the SDRAM when the memory system is busy. Our experiments show that the memory access concurrency degree of *applu* is much higher than that of *mgrid*. For example, 27% of the memory accesses in *applu* have the concurrency degree of 8, and the percentages of the memory accesses with concurrency degrees of 3 to 7



**Figure 6.4:** The distributions of the number of concurrent memory requests for programs *mgrid* and *applu* on the SDRAM system when the memory is busy.

range from 5% to 13%. In contrast, the majority of memory accesses in *mgrid* have low concurrency degrees of 1 (which is 48% in the total memory accesses), and 2 (which is 29% in the total memory accesses). This explains why the Rambus DRAM and the Direct Rambus DRAM are more effective to *applu* than *mgrid* although both programs have comparable memory access portions in their CPIs (see Table 6.4).

## 6.4 Cached DRAM with Increasing ILP Degree

When the ILP degree (issue width) of the processor increases, the processor will increase the demand on the main memory system. Thus, it is interesting to observe how cached DRAM will perform as ILP degrees change. In this section, we compare the performance of the cached DRAM and the SDRAM with 4-way issue, 8-way issue, and 16-way issue processors.



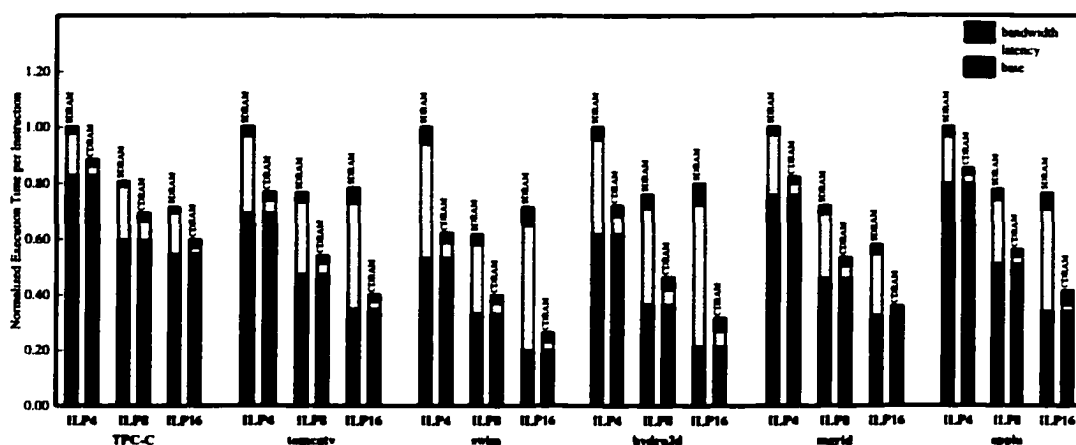


Figure 6.5: CPIs as the ILP degree changes

Figure 6.5 shows the CPIs and their decompositions for the TPC-C workload and four SPECfp95 programs: *tomcatv*, *hydro2d*, *mgrid*, and *applu*. The base portion of CPI decreases proportionally for all programs as the ILP degree increases, which means that, with an ideal main memory system, the performance always improves as ILP degree increases. For the TPC-C workload, increasing ILP degree from 8 to 16 only causes the base portion of CPI decreases slightly, which indicates that the increase of ILP is not so effective for the TPC-C workload as for SPECfp95 programs.

Our experiments show that the CPIs of all the programs on the 16-way issue processor with the SDRAM are higher than that of the 8-way issue processor with the SDRAM. The performance degradation mainly comes from the heavy demand on the main memory system, which results in reducing the execution throughput of instructions. The heavy demand also causes congestion at the main memory system, and enlarges the memory access latency due to queuing effects. Consequently, the number of instructions retiring from the

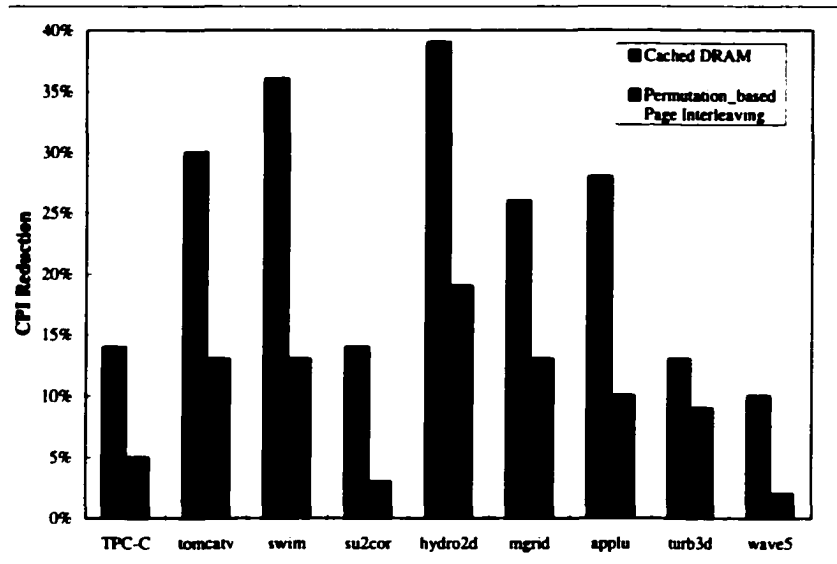
instruction window is reduced, which slows down the speed of new instructions entering the instruction window. We find that, for example, the instruction dispatch rate on the 16-way issue processor is 30% less than that of the 8-way issue processor for *hydro2d*. The average time that an instruction stays in the instruction window is 43 processor cycles for the 8-way issue processor, but increases to 88 processor cycles for the 16-way issue processor.

In contrast, the cached DRAM performs well as the ILP degree increases. When the on-memory cache hit rate is high, the cached DRAM can support high bandwidth accesses, thus the congestion at the memory system is not as severe as with SDRAM. The memory stall time does not increase as the ILP degree increases from 8 to 16.

The benefit of reducing computing time due to the increase of ILP degree will eventually be offset by not effectively tolerating long memory access latency on the SDRAM. In contrast, the effectiveness of the cached DRAM will increase as the ILP increases to a much larger degree.

## 6.5 Comparisons with Approaches Exploiting Row Buffer Locality

Figure 6.6 compares the performance improvements by cached DRAM and by the permutation-based interleaving scheme over SDRAM systems, using the identical simulation configuration (see Table 6.3) except the difference described as follows. The cached DRAM has  $16 \times 4$  KByte on-memory cache, while the permutation-based interleaving scheme are used for 32 banks and  $32 \times 2$  KByte row buffers. We show that cached DRAM reduces the CPI by 23% on average, while the average CPI reduction by the permutation-based scheme



**Figure 6.6:** CPI reduction by cached DRAM and by XOR-based interleaving scheme.

is 10%.

The performance results show clearly that using on-chip memory outperforms using the permutation-based interleaving scheme. However, the permutation-based interleaving scheme requires little additional cost, and does not require any change in the DRAM chip. In contrast, cached DRAM requires an additional chip area for the SRAM cache and additional circuits for cache management.

## 6.6 Summary

Using technical and commercial workloads, we have evaluated the cached DRAM technique in the context of processors with full ILP capabilities and large data caches. A small and fully associative on-memory cache in the cached DRAM effectively exploits the locality of memory access streams missing the L2 cache, and takes advantage of the high bandwidth in its connection to the DRAM core. This study indicates that commercially available DRAM schemes, such as SDRAM, ESDRAM, RDRAM, and DRDRAM, do not exploit memory access locality of data-intensive workloads as effectively as a cached DRAM does. The main reason for this is related to the direct mapped structures on the memory banks/row-buffers of these commercial DRAM organizations. This study further indicates that cached DRAM can consistently show its advantages as the ILP degree increases, and that cached DRAM can outperform the permutation-based interleaving technique with a justifiable cost increase.

## Chapter 7

# Fine-grain Priority Scheduling on Multi-channel Memory Systems

In Chapters 5 and 6, we have discussed techniques to reduce memory access latency by exploiting row buffers and on-memory caches. Another approach to reduce memory access latency is memory access scheduling. In this chapter, we focus on memory access scheduling on multi-channel memory systems.

Configurations of contemporary DRAM memory systems has become increasingly complex. Those memory systems, such as Direct Rambus DRAM systems, can support multiple memory channels, while each channel can connect multiple devices (chips). Each chip consists of multiple banks, where concurrent accesses to different banks can be pipelined. For memory-intensive applications running on contemporary computer systems, the occurrence of multiple outstanding memory requests is frequent. Memory access scheduling can reorder the sequence of concurrent accesses to reduce access latency and improve memory bandwidth utilization [75, 71, 48, 89, 90, 70]. In addition, a memory request for a cache miss can be further split into several sub-requests that can be processed separately. Normally, a cache miss causes a cache line fill request, which fetches more data than that is immediately

required to resume processor execution. This provides an opportunity to improve performance by splitting the request into multiple sub-requests with smaller sizes and serving the critical ones (containing immediately required data) first. On a multi-channel memory system, such a scheduling method requires a number of considerations, such as how to split a single reference, how to assign sub-requests to channels, and how to schedule concurrent accesses.

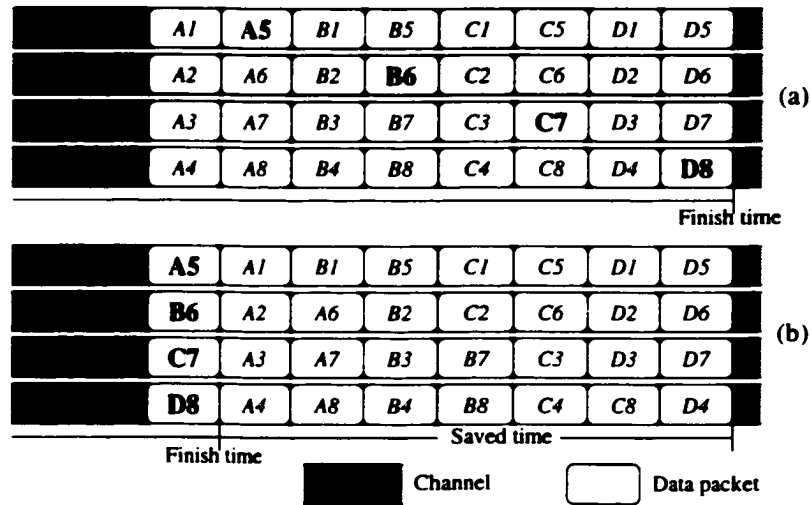
A recent study [24] finds that program performance is highly sensitive to the DRAM system configuration, and suggests that tuning burst (sub-block) sizes and channel configurations is an effective way to optimize the DRAM system performance for a given memory-intensive workload. Specifically, they evaluate the performance effect of sub-block size on burst ordering, where each cache block is split into multiple sub-blocks and critical sub-blocks are served before non-critical ones. In their study, all sub-blocks from a cache line are mapped to the same channel and the same page. Thus, in order to exploit concurrency within a single channel, the choice of sub-block size becomes a trade-off between reducing latency of critical data access and lowering system overhead. They find that different applications have optimal performance on different sub-block sizes and the optimal sub-block sizes scale with the channel width.

In this study we show that, by utilizing fine-grain priority access scheduling, we are able to find a workload independent configuration that achieves optimal performance on a multi-channel memory system. In order to fully utilize the available bandwidth and concurrency, our approach splits a memory reference into sub-blocks with minimal granularity, and maps sub-blocks from a reference into different channels. All channels can be used to process a single cache line fill request. In order to increase the parallelism between processor execution

and memory accesses, fine-grain priority scheduling is exploited. Sub-blocks that contain the desired data are marked as critical ones with higher priorities and are returned earlier than non-critical sub-blocks. This approach is similar to the method of “critical word first”, but it also allows critical sub-blocks of one cache block to bypass non-critical sub-blocks from other cache blocks. By combining with existing DRAM scheduling policies, choosing the minimum sub-block size allows faster access to critical data without increasing the memory system overhead.

Figure 7.1 gives an example that shows the performance potential of fine-grain priority scheduling. In this example, a 4-channel memory system is processing four cache misses concurrently. Each cache block is split into eight sub-blocks, and the four critical sub-blocks are mapped to different channels. With fine-grain priority scheduling, all the critical sub-blocks finish earlier than non-critical sub-blocks, saving seven time units in fetching all critical data. In this example, the clustering of the four cache misses provides the scheduling opportunity. Our study will show that the cache miss clustering is frequent, i.e., the burstiness of cache misses is high. As a result, queuing delay can be a major component of access time. Fine-grain priority scheduling can reduce the memory stall time by reducing the queuing delay of critical data.

In this study, we quantitatively investigate the miss burstiness for memory-intensive applications from the SPEC2000 benchmark suite on ILP processors with multi-channel Direct Rambus DRAM systems. We also analyze the combination of fine-grain priority scheduling with other DRAM access scheduling techniques, and compare the performance with that of gang scheduling [67] and burst scheduling [24]. Our study provides the following performance results and findings.



**Figure 7.1:** The order of transferring sub-blocks on a system with four memory channels: (a) without priority scheduling and (b) using fine-grain priority scheduling. The letters A–D represent cache blocks, each of which is split into eight sub-blocks. The boxes with bold letters represent the critical sub-blocks that contain the desired data.

- Fine-grain priority scheduling is effective in reducing memory stall time and increasing IPC (Instructions Per Cycle). Compared with gang scheduling that serves a single cache miss request with multiple channels grouped together, the IPC improvement is 13% on average (up to 34%) for fifteen selected SPEC2000 programs on a 2-channel Direct Rambus DRAM memory system, and 8% on average (up to 22%) on a 4-channel memory system. Compared with burst scheduling that serves multiple sub-requests of a single cache miss with one channel but critical sub-requests first, the average IPC improvement is 16% and 14% on the 2-channel and 4-channel memory systems, respectively. The processor is 2 GHz and 4-way issue.



- Combined with other scheduling policies, fine-grain priority scheduling is able to effectively utilize the memory system resource. For six of the programs, the 2-channel system with fine-grain priority scheduling can achieve performance comparable to that on the 4-channel system with gang scheduling or with burst scheduling.
- We suggest that a DRAM system configuration and its optimization be emphasized on access scheduling and DRAM mapping schemes. Taking this approach, we are able to find an optimal memory configuration that is workload independent.

The rest of this chapter is organized as follows. The next section briefly introduces the background of multi-channel memory systems. Section 7.2 discusses the issues in fine-grain priority scheduling and its combination with other DRAM scheduling policies. Section 7.3 discusses the design and implementation complexity of fine-grain priority scheduling. Section 7.4 describes the experimental method. Section 7.5 presents the experimental results. Finally, Section 7.6 discusses the related work, and Section 7.7 summarizes the study.

## 7.1 Memory System Considerations

### 7.1.1 Multi-channel Memory Systems

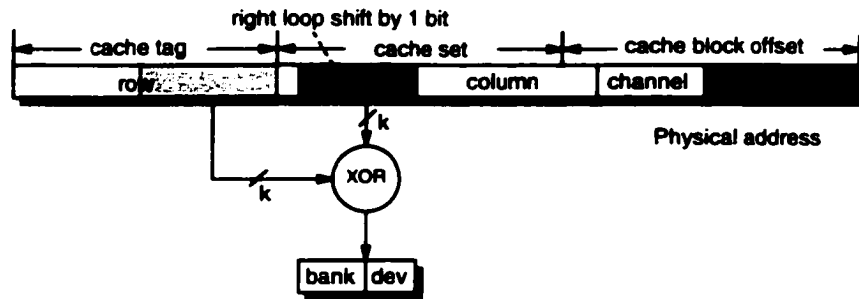
Multi-channel memory systems have been used with high performance processors that require high bandwidth DRAM memories. Each channel can be scheduled independently. Direct Rambus DRAM is such a representative memory system. A Direct Rambus DRAM system generally consists of multiple channels, where each channel supports 1.6 GB/s bandwidth. Each channel has its own row control bus, column control bus, and two-byte wide data bus. The separation of row and column control buses eliminates the contention in the

address bus between row operations (precharges and row activations) and column accesses. The bus clock rate is 400 MHz and the data is transferred on both edges of the clock. The row and column addresses/commands and the data are transferred in packets, each taking four bus cycles. The minimal data packet length is 16-byte. Each channel can connect multiple devices (chips). Each device can have 32 banks and 33 half-page row buffers (this may be different according to the configuration). Those banks may be operated independently, which provides high concurrency at the bank level. The Intel Pentium 4 processor supports two channels, and the Compaq Alpha 21364 processor supports up to eight channels.

### 7.1.2 DRAM Mapping Scheme

The DRAM mapping scheme determines how to map a physical address to a location in the DRAM system. The choice of DRAM mapping scheme directly affects the row buffer hit rate and the memory system performance [124].

A word in a Direct Rambus DRAM system is addressed by the channel index, the device index, the bank index, the row address, and the column address. The first mapping consideration is how to map the sub-blocks in a cache line onto multiple channels. We use a method interleaving the sub-blocks onto all channels, which is the same as that used in [67]. This interleaving scheme allows the aggregate bandwidth of all channels to be used to transfer a single cache line (assume the number of sub-blocks in a cache line is no less than the number of channels). The mapping scheme in [24] maps all cache lines in a DRAM page-sized block on a single channel. The requests on different channels are scheduled independently. However, this scheme cannot fully utilize the available bandwidth of all channels for a single cache line fill request. In addition, program access locality within



**Figure 7.2:** The mapping from the physical address to the address in Direct Rambus address space.

the DRAM page-sized block may cause unbalanced usage of memory channels. In contrast, our method groups channels together to serve each cache line fill request, but schedules operations on each channel independently to return critical sub-blocks earlier.

Another mapping consideration is how to map continuous addresses to multiple banks. Our approach interleaves page-sized blocks onto banks using the XOR-based page-interleaving scheme [124, 67]. It maps a continuous DRAM page-sized block onto a DRAM bank to exploit the locality in the row buffer, and XOR two portions of address bits (conventional bank index and cache tag) to permute the mapping of pages to banks. Consequently, accesses causing row buffer conflicts in the conventional page-interleaving scheme are distributed to different banks without changing the locality in the row buffer. The studies in [124, 67] show that the scheme can significantly improve the row buffer hit rate. Figure 7.2 describes the mapping scheme used in our experiments.

## 7.2 Fine-grain Priority Scheduling

### 7.2.1 Granularity of Scheduling

Current ILP processors have the ability to generate multiple cache misses before stalling. This provides an opportunity for performance improvement by scheduling concurrent memory requests for those cache misses. In general, only a portion of a cache line contains the currently required data (although other portions may be needed in the near future). The fine-grain priority scheduling tries to exploit this opportunity. It issues multiple DRAM requests for a single cache miss, where each request fetches a *sub-block* of the cache line. Sub-blocks that contain the desired data are *critical sub-blocks*. The requests for critical sub-blocks are given higher priority over those requests for non-critical ones.

Each DRAM system has a limit on the minimal request length. Thus, the sub-block size should be no less than that minimal length. Using smaller sub-block size allows the current request to finish faster and makes newly arrived requests to be issued earlier. However, it is a concern that a small sub-block size may reduce the burst length of DRAM accesses and thus increase the system overhead [24]. Nevertheless, we will show that if fine-grain priority scheduling is combined with other scheduling techniques and suitable mapping schemes, the overhead will not exceed that of coarse-grain scheduling on the Direct Rambus DRAM platform. For this reason, we choose the smallest granularity available for Direct Rambus DRAM system as the sub-block size, which is 16 bytes, in this study.

### 7.2.2 Scheduling Policies

In this chapter, we discuss three scheduling policies: fine-grain priority scheduling, gang scheduling, and burst scheduling. Each term actually represents a combination of several basic access scheduling policies, a channel configuration, a DRAM mapping scheme, and a choice of scheduling granularity. We assume a scheduler architecture similar to that presented in [90] (see Section 7.3) is used to enforce the three policies.

- Pending requests on a DRAM bank are queued in a bank scheduler. Each bank scheduler has an arbiter to determine the next operation on the associated bank.
- Each independent channel has a channel scheduler, which includes a row arbiter and a column arbiter. The row arbiter selects a precharge or a row access (if any) to use the row control bus, based on the selections of bank schedulers. The column arbiter selects which column access (if any) to use the column control bus and the data bus.

All the three scheduling policies are combined with four basic scheduling policies that are enforced in the following order: *read-bypass-write*, *hit-first*, *explicit priority*, and *in-order*. For example, a non-critical read request that requires a column access is issued first even when there is another critical read request that requires a precharge to the same bank. The hit-first policy is enforced before the explicit priority scheduling so that fine-grain priority scheduling would not cause severe row buffer thrashing when multiple requests are mapped onto the same bank. In contrast, enforcing only explicit priority scheduling may cause more precharges when bank conflicts occur.

There are three levels of explicit priorities for read requests, namely *critical priority*, *load priority*, and *store priority*, from highest to lowest. The critical priority is assigned to

critical sub-blocks of read misses, and load priority is assigned to non-critical sub-blocks of read misses. The store priority is assigned to read requests for write misses, as a write-back and write-allocate L2 cache is used in this study.

In fine-grain priority scheduling, each L2 cache miss results in multiple DRAM requests that are mapped to multiple channels evenly; each DRAM request is associated with an explicit priority; and concurrent requests are scheduled based on the policies discussed above. It uses a fixed 16-byte as the sub-block size, which is the smallest granularity with current Direct Rambus technology. Each physical channel is configured as an independent unit and has its own channel scheduler. Instructions stalled for a critical sub-block are resumed when the data of the sub-block is returned.

Gang scheduling uses the cache block size as the burst size. All channels are grouped together as one logical channel, and there is only one channel scheduler. Instructions stalled for a missed block are resumed when the whole block is returned.

In burst scheduling, each L2 cache miss results in multiple DRAM requests that are mapped to the same independent channel; each DRAM request is associated with an explicit priority. In this study, the sub-block size is set to 32-byte. For a 2-channel system, each physical channel is an independent channel. For a 4-channel system, two physical channels are grouped together. There are two channel schedulers in both cases.

When a miss on a cache block happens, the sub-block containing the desired data is marked critical. Due to program locality, it is very likely that when the requests of this miss are being processed, more misses happen on other sub-blocks of the same cache block. In this case, the sub-blocks containing the newly arrived requests become critical ones and gain higher priority. Both fine-grain priority scheduling and burst scheduling will consider

this change and update the priority information dynamically.

The read-bypass-write and hit-first policies not only improve the performance by themselves but also help fine-grain priority scheduling avoid the potential increase of system overhead. There is one case that the system overhead may still increase. When the number of banks is very small, the bank conflicts can be severe, and thus fine-grain priority scheduling may cause more precharges than burst scheduling. Fine-grain priority scheduling always balances the utilization of multiple channels, however, which scheduling performs better will depend on application access patterns. In practice, Direct Rambus memory systems have a sufficient number of banks to avoid severe bank conflicts. SDRAM memory systems usually have large size row buffers which lead to less precharges when the locality in row buffer is good. In addition, large size SDRAM memory systems may also have enough banks to avoid severe bank conflicts. The DRAM mapping scheme used in our study produces high row buffer hit rates. Thus, open page mode is used in our experiments.

## 7.3 Complexity Analysis

### 7.3.1 Complexity inside Processor

**Cache and Cache Controller** One concern on fine-grain priority scheduling is that the scheduling might change the internal of L2 cache and/or its controller, because data returns from the memory in a unit of sub-block instead of cache block. Such a change is definitely undesirable. Fortunately, there are existing mechanisms on high-performance processors that can address this issue. For example, the MIPS R10000 has a four-entry incoming buffer that can accept returning data at any rate and in any order [120]. Up to

four outstanding read requests to memory are supported, thus each outstanding request is guaranteed to have one allocated incoming buffer entry. The Power-PC 604 has a similar line-fill buffer [105]. The incoming buffer can be used to merge out-of-order returning sub-blocks with only trivial changes. Future high-performance processors that support multiple outstanding memory requests will likely have such kind of mechanisms.

**Address Path to Memory Controller** There will be additional lines for transferring priority information. Priority information can be transferred as a bitmap or the position index of a sub-block. Using a bitmap requires additional lines, but allows prioritizing multiple sub-blocks simultaneously, which helps the case when multiple cache misses happen to the same cache line at the same cycle.

**Priority Updates** The MSHR needs to send priority update to the memory controller when a read miss happens on a non-prioritized sub-block of a cache block that is already missed. A bitmap can be used with each MSHR entry to memorize which sub-blocks have been prioritized<sup>1</sup>. The contention due to priority updates on the address path between the processor and the memory controller is negligible because the speed of the path is much faster than the service rate of the memory system.

### 7.3.2 Complexity in Memory Controller

The basic function of memory controller is to issue DRAM operations (precharge, row activation, or column access) to DRAM banks under the DRAM timing constraints for DRAM access requests. With high-performance processors and high-bandwidth memory systems, the memory controller must have the access scheduling ability to order DRAM operations

---

<sup>1</sup>We assume that the MSHR implementation in [34] is used.



for multiple outstanding requests. Without this ability, the opportunity to exploit the memory access concurrency allowed by the processor and the memory system will be wasted, and the performance penalty is unacceptable for memory-intensive applications.

A *memory access scheduler architecture* is proposed in [90], which can enforce a number of scheduling policies. A fine-grain priority scheduling policy can be implemented on that architecture. The scheduler architecture organizes incoming requests by DRAM banks. Each bank has its own arbiter to determine its next operation. A global arbiter determines which bank gets the shared resources, such as the address bus and the data bus. This scheduler architecture can be adapted to work with Direct Rambus memory systems. Each independent channel needs a channel scheduler, and each channel scheduler needs two arbiters, one for the row control bus and the other for the column control bus. Although Direct Rambus memory systems can have a large number of banks, the bank schedulers can be assigned to busy banks dynamically, thus only a limited number of bank schedulers are needed.

With fine-grain priority scheduling on an  $n$ -channel system,  $n$  channel schedulers are needed. In comparison, gang scheduling requires one channel scheduler because there will be one logical channel. On the other hand, fine-grain priority scheduling does not complicate the structure of each individual bank scheduler or channel scheduler. Another change is that the memory controller may split one memory reference request into multiple requests onto those channels, and need to accept priority updates. In this aspect, burst scheduling has almost the same complexity as fine-grain priority scheduling.

Speed	2GHz, 4-way issue
RUU size	64
LSQ size	32
MSHR size	16
write buffer size	8
L1 cache	4-way 64KB inst./data, 2-cycle hit latency, 64B cache line, write-back
L2 cache	unified 4-way 1MB, 8-cycle hit latency, 128B cache line, write-back

**Table 7.1:** Key processor parameters.

Parameters	Values
Precharge delay	8 bus cycles
Row access delay	8 bus cycles
Column access delay	8 bus cycles
Length of packets	16 bytes
Banks per device	32
Page size	2KB
Row buffer	33 half-page size

**Table 7.2:** Key parameters of the Direct Rambus DRAM used in the simulation. The bus cycle time is 2.5 ns (400 MHz).

## 7.4 Experimental Environment

We use SimpleScalar 3.0b [14] to simulate an out-of-order execution processor. An event-driven simulation of a multi-channel Direct Rambus DRAM system is incorporated into the original simulator. Table 7.1 gives the key parameters of the processor model.

We use the parameters of 256 Mbit Direct Rambus DRAM [82] as the parameters of DRAM memory system simulated in our experiments. Table 7.2 describes the key parameters of this DRAM. We configure the simulated system as 2-channel and 4-channel systems, where each channel has four devices.

We use the pre-compiled SPEC CPU2000 Alpha binaries in [117]. Fifteen programs

(five integer programs and ten floating point ones) are selected, which have relatively large memory access demands. For all the applications, we fast-forward 4000M instructions and collect program execution statistics on the next 200M instructions.

## 7.5 Experimental Results

### 7.5.1 Burstiness in Miss Streams

We first measure the fraction of program execution time with bursty memory accesses. Figure 7.3 shows the fraction of program execution time with two or more outstanding memory references on a 2-channel system with gang scheduling for the selected SPEC2000 programs. We can see that the fraction of bursty phase is highly application dependent, which ranges from about 6% to 90%.

Figure 7.4 further presents the distribution of the number of concurrent accesses in the bursty phase. Figure 7.4(a) contains programs with the fraction of bursty phase higher than 40% and Figure 7.4(b) contains programs with the lower bursty phase fraction. In general, programs with a higher fraction of bursty phase tend to have higher probabilities of a large number of concurrent accesses. For all the programs presented in the left figure, more than 40% of bursty references are grouped with at least three other references. Even for some programs with a small bursty phase fraction, the burstiness inside the bursty phase is still high. For example, program *256.bzip2* only spends 6% of its execution time in the bursty phase, however, more than 60% of concurrent accesses are clustered as groups with at least eight references.

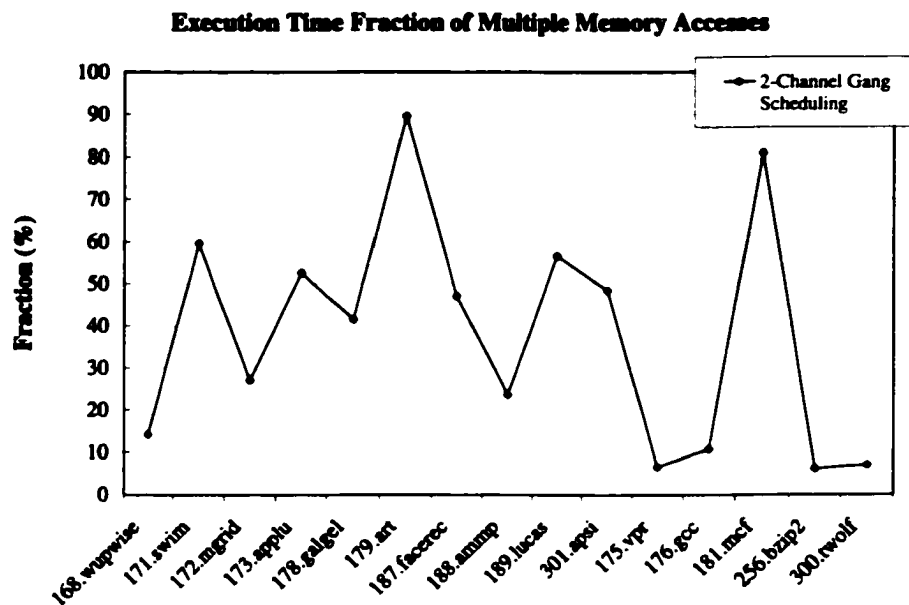
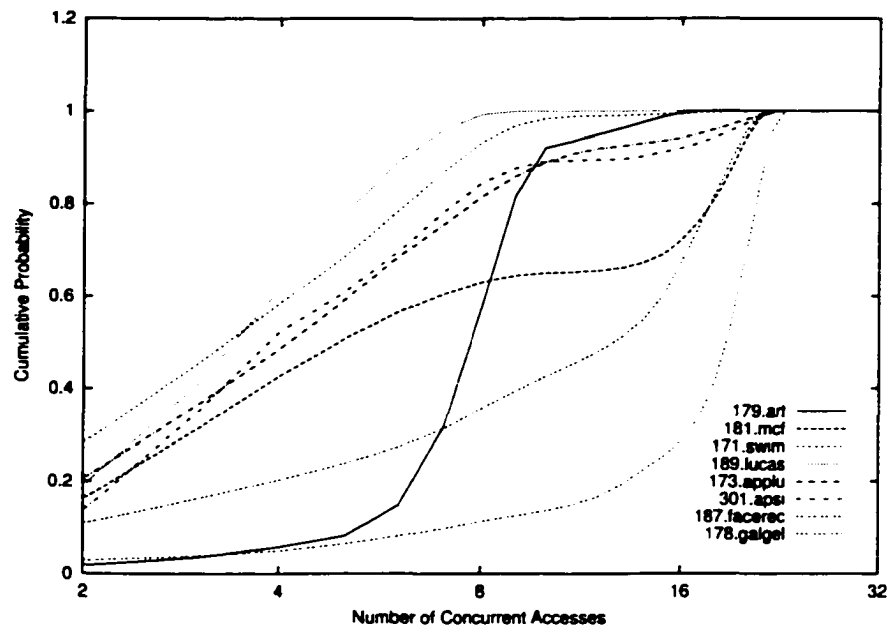


Figure 7.3: Fractions of bursty phase in execution for SPEC2000 programs.

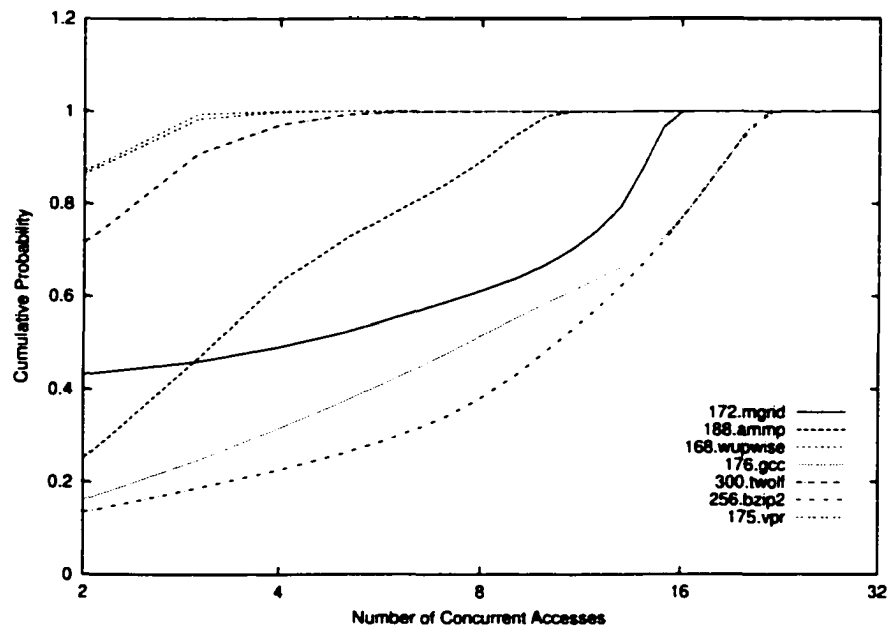
### 7.5.2 Potentials of Fine-grain Priority Scheduling

Fine-grain priority scheduling targets at reducing the latency for critical sub-blocks by serving the critical ones before the non-critical ones. However, if all sub-blocks are critical, fine-grain priority scheduling will not make any difference. To evaluate the potential of fine-grain priority scheduling, we measure the percentage of critical sub-blocks in a cache line when the whole cache line fill request completes. Our experiments indicate that on the 2-channel system, for the fifteen programs, this percentage ranges from 15.3% to 57.7%. On average, 33.8% of sub-blocks are critical ones. This indicates that there is a large space left for fine-grain scheduling to reorder requests based on their priorities.

Figure 7.5 shows the waiting time distribution of critical sub-blocks and non-critical



(a)

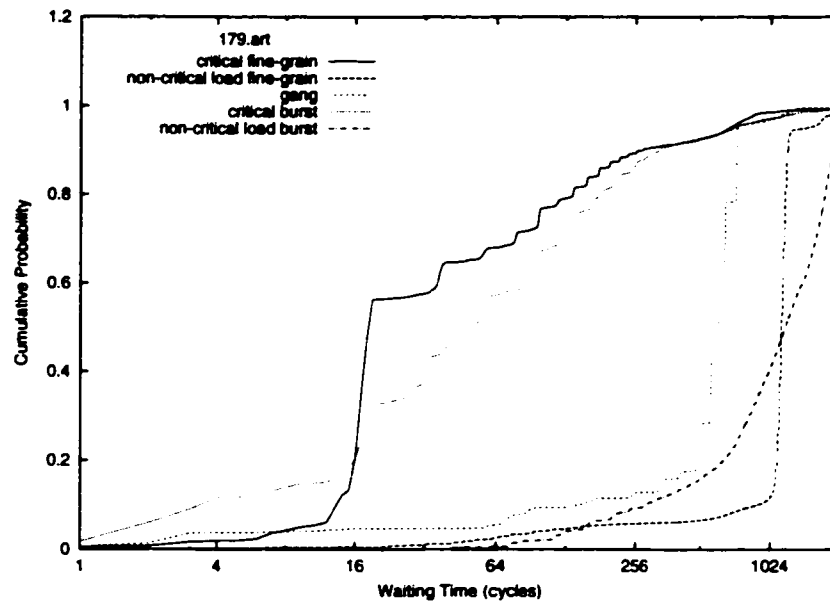


(b)

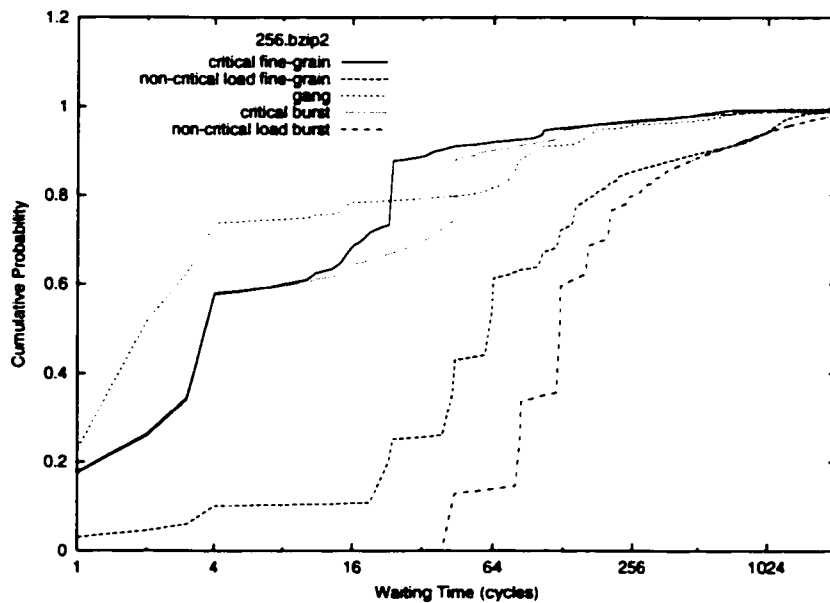
**Figure 7.4:** Distribution of the number of concurrent accesses.

sub-blocks of read misses. We can see that critical sub-blocks have much shorter queuing delay than non-critical ones. We use two programs *179.art* and *256.bzip2* as examples here. Program *179.art* has very high bursty phase fraction (about 90%) and high burstiness within the bursty phase. For this application, the waiting time is a significant portion of the total access time. Fine-grain priority scheduling effectively reduces the waiting time for critical sub-blocks. Compared with burst scheduling, the average waiting time for critical sub-blocks reduces from 133 cycles to 104 cycles, and the average waiting time for non-critical load sub-blocks reduces from 1157 cycles to 1070 cycles. With fine-grain priority scheduling, 60% of critical sub-blocks have waiting time less than 36 cycles. In comparison, with burst scheduling, 40% of critical sub-blocks have waiting time longer than 80 cycles. Compared with gang scheduling, the average waiting time for critical sub-blocks reduces from 557 cycles to 104 cycles, but the average waiting time for non-critical load sub-blocks increases from 557 cycles to 1070 cycles. *256.bzip2* has low bursty phase fraction (only 6%) but high burstiness in the bursty phase. Compared with gang scheduling and burst scheduling, the average waiting time for critical sub-blocks is reduced from 42 cycles and 32 cycles, respectively, to 27 cycles.

Figure 7.6 shows the probability that multiple critical sub-blocks are mapped to the same channel under fine-grain priority scheduling. We can see that for most programs, fine-grain priority scheduling can evenly distribute critical requests to different channels. However, for applications with high burstiness, it is still possible that multiple critical requests are mapped to the same channel. The existence of multiple critical requests in the same channel indicates that fine-grain priority scheduling can reduce the processor waiting time for currently required data.

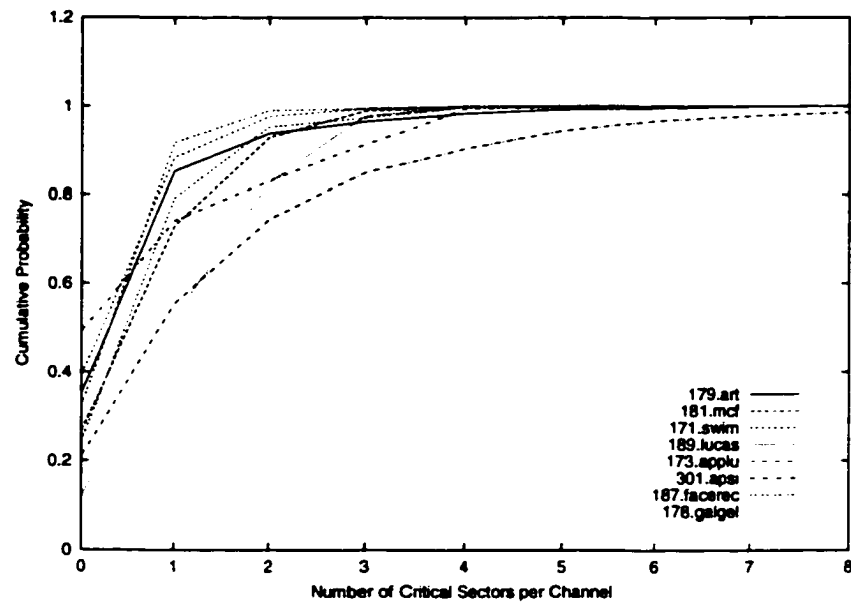


(a)

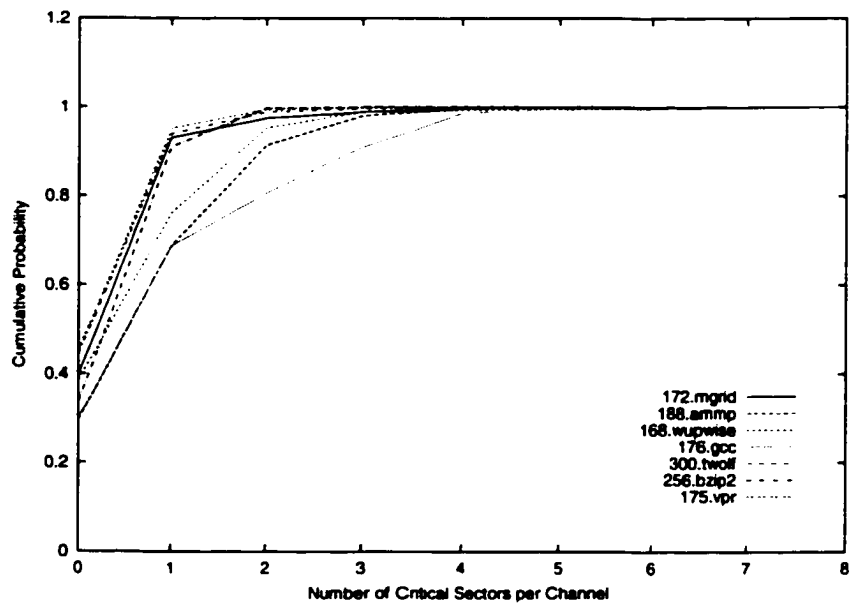


(b)

**Figure 7.5:** Waiting time distribution of critical and non-critical load sub-blocks.



(a)



(b)

**Figure 7.6:** Probabilities of multiple critical sub-blocks mapping to the same channel.



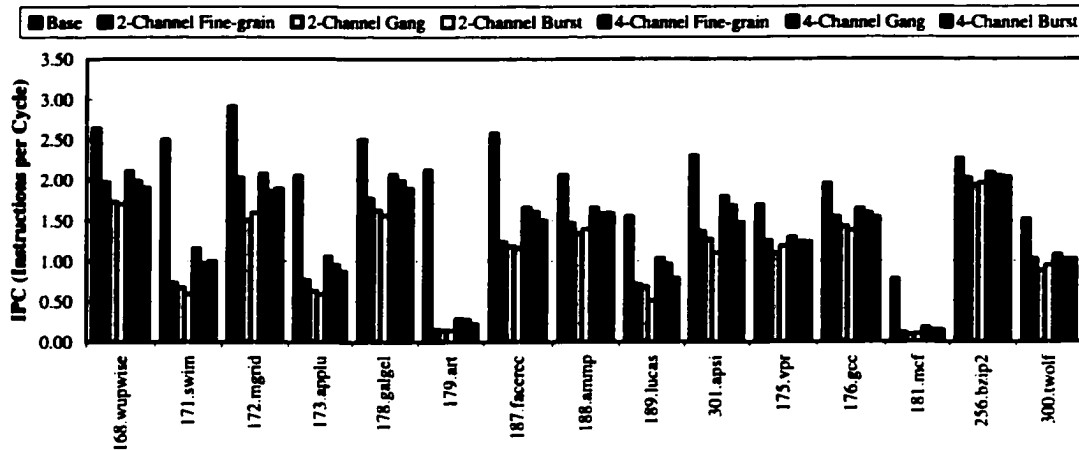


Figure 7.7: IPC on 2-channel and 4-channel Direct Rambus DRAM systems.

### 7.5.3 Performance Improvement of Fine-grain Priority Scheduling

Figure 7.7 presents the performance improvement of fine-grain priority scheduling in terms of IPC for 2-channel and 4-channel Direct Rambus DRAM systems. In this figure, the base IPC of each application is the IPC on a system with the perfect DRAM configuration that has the latency of L2 cache hit and an infinite bandwidth. The base IPC reflects the ideal performance after eliminating the memory stall time.

Compared with gang scheduling, fine-grain priority scheduling can increase the IPC by up to 34% for the 2-channel DRAM system. For the fifteen selected programs, the average IPC increase is 13%. Four programs (*172.mgrid*, *173.applu*, *181.mcf*, and *300.twolf*) have performance improvement higher than 15%. This implies that fine-grain priority scheduling effectively increases the parallelism between processor execution and DRAM accesses by reducing the latency for critical accesses.

Compared with burst scheduling, fine-grain priority scheduling can increase the IPC

by 16.3% on average (up to 38.7%) for the 2-channel DRAM system. This indicates that fine-grain priority scheduling can better utilize the available concurrency of DRAM systems by spreading requests evenly onto multiple channels.

Fine-grain priority scheduling is especially effective for applications with a relative large memory stall portion, modest memory bandwidth demand<sup>2</sup>, and high burstiness in miss streams. For applications with small memory bandwidth demand and relatively fewer cache misses, the performance improvement from fine-grain priority scheduling is modest. For example, the memory bandwidth demand of *256.bzip2* is only 0.8 GB/s, the fraction of bursty phase is only 6%, and the number of L2 cache misses per 100 instructions is only 0.11. For this application, the memory stall time is not a significant portion of the total execution time. The fine-grain priority scheduling scheme improves the performance modestly by 5% and 3% compared with gang scheduling and burst scheduling, respectively.

For applications with extremely high memory bandwidth demands, such as *179.art*, fine-grain priority scheduling improves performance modestly (6.0%) compared with gang scheduling. This is not surprising. The bandwidth demand of the program is so high (64.0 GB/s) compared with the available bandwidth (3.2 GB/s). Returning critical data earlier does not provide a large improvement in this case. Compared with burst scheduling, the performance improvement is promising (13%). This indicates that fine-grain priority scheduling can better utilize the available bandwidth and concurrency by evenly distributing sub-requests to multiple channels.

As the number of channels increases to four, the congestion at the main memory sys-

---

<sup>2</sup>We use the memory bandwidth achieved by the application on the perfect DRAM system as the bandwidth demand of the application.

tem is reduced because of the increasing bandwidth and concurrency. As expected, the speedup by using fine-grain priority scheduling drops for most applications. For the fifteen programs, fine-grain priority scheduling increases the IPC by 8% on average (up to 22%) compared with gang scheduling. However, the speedup for some programs with high memory bandwidth demands increases. For example, the performance improvement for *171.swim* increases from 9% to 19% as the number of channels doubles. It indicates that, when the bandwidth pressure is alleviated for bandwidth-bounded applications, the performance potential of fine-grain priority scheduling increases. Compared with burst scheduling, the average performance improvement of the fifteen programs is 14%. As the number of channels increases, the imbalance of request distribution on multiple channels may also increase for burst scheduling. Burst scheduling does not utilize the available bandwidth and concurrency as well as fine-grain priority scheduling does.

It is interesting to observe that for six of the fifteen programs, the performance on the 2-channel DRAM system after applying fine-grain priority scheduling is comparable or even better than that on the 4-channel DRAM system with gang scheduling. For *168.wupwise*, *176.gcc*, and *256.bzip2*, the IPC on the 2-channel DRAM system with fine-grain priority scheduling is within 3% lower than that on the 4-channel DRAM system with gang scheduling. For *172.mgrid*, *175.vpr*, and *300.twolf*, the IPC on the 2-channel DRAM system with fine-grain priority scheduling is higher than that on the 4-channel DRAM system with gang scheduling by up to 10%. Compared with burst scheduling on the 4-channel DRAM system, fine-grain priority scheduling gains comparable or better performance on the 2-channel DRAM system for these six programs.

Compared with the 2-channel system, the 4-channel system not only doubles the avail-

able bandwidth, but also doubles the number of memory chips. Fine-grain priority scheduling can better utilize the existing resources and achieve performance comparable to that on a system with much higher cost. Of course, for those applications whose performance is limited by the available bandwidth, paying more to increase the bandwidth is the most effective way to improve performance.

For all applications on both the 2-channel and the 4-channel systems, fine-grain priority scheduling always achieves the best performance. In comparison, for gang scheduling and burst scheduling, which one performs better is application and configuration dependent.

## 7.6 Other Related Work

There are two reasons why DRAM memories cause performance degradation: long latency and limited bandwidth. From the performance viewpoint, all approaches targeting at DRAM performance improvements work at one or more of the following four aspects: increasing the peak bandwidth of the memory bus/channel and/or DRAM chips; reducing the access latency of DRAM chip; improving the bandwidth utilization; and increasing the parallelism between processor execution and DRAM operation.

Peak bandwidth can be improved by increasing the data transfer rate, the total bus/channel width, or both. SDRAM has improved the data transfer rate to 100MHz, to 133MHz, and now to 166 MHz. The DDR technique allows SDRAM to transfer data at both clock edges, and thus doubles the data transfer rate. Buses as wide as 256 bits or more have been used in high-end workstations [22]. Direct Rambus DRAM [82] uses narrow buses and increases the data transfer rate to 800 MHz. Multiple memory channels are supported

not only in high-end servers but also in workstations. Currently, the Pentium-4 processor can support two Direct Rambus channels, and the Compaq Alpha 21364 can support up to eight channels.

Most approaches to reduce the access latency of DRAM chips take advantage of the locality available in the miss stream from the processor. Cached DRAM [46] integrates SRAM cache into DRAM chips. Enhanced DRAM incorporates a single line of SRAM cache with each row buffer of DRAM banks. Virtual Channel DRAM contains multiple SRAM lines that is one fourth of the row buffer size. A hit to the SRAM caches can save the DRAM row access time. Because of the huge bandwidth available inside DRAM chips, the block size of the SRAM caches can be much larger than those of L1 and L2 caches. Thus, the hit rate on the SRAM caches can be as high as 90%. Some recent studies proposed XOR-based mapping schemes to improve the hit rate of DRAM row buffers, thus improve the performance without the cost of adding SRAM caches [124, 67].

Peak bandwidth may not be fully utilized due to limited available concurrency or bus overhead. Today, high-performance processors can support multiple outstanding loads and stores, and DRAMs can serve multiple requests in a pipelined manner. The limitation of concurrency still comes from bank conflicts, i.e., when multiple requests ask for data in different pages located in the same bank. Access scheduling [75, 71, 48, 89, 90, 70] that groups accesses to the same pages together has been proven to be effective for streaming applications. The above approaches that utilize data locality at the DRAM side also reduce the bank conflicts. The bus overhead includes the turnaround time that appears between read and writes. This overhead can be reduced by using write buffer [100] to delay writes and group them together. Another overhead is the asymmetry of the timings of read and

write operations. This can be removed by changing the write timing so that they are symmetric with read timing [24].

Out-of-order execution processors have the ability to overlap the processor execution and DRAM memory accesses to some extent. Hardware or software prefetching [115] is an effective approach to increase the parallelism between processor execution and DRAM operations. Most prefetching studies do not consider DRAM properties, however, [67] shows that combining prefetching and DRAM access scheduling can effectively improve the performance.

## 7.7 Summary

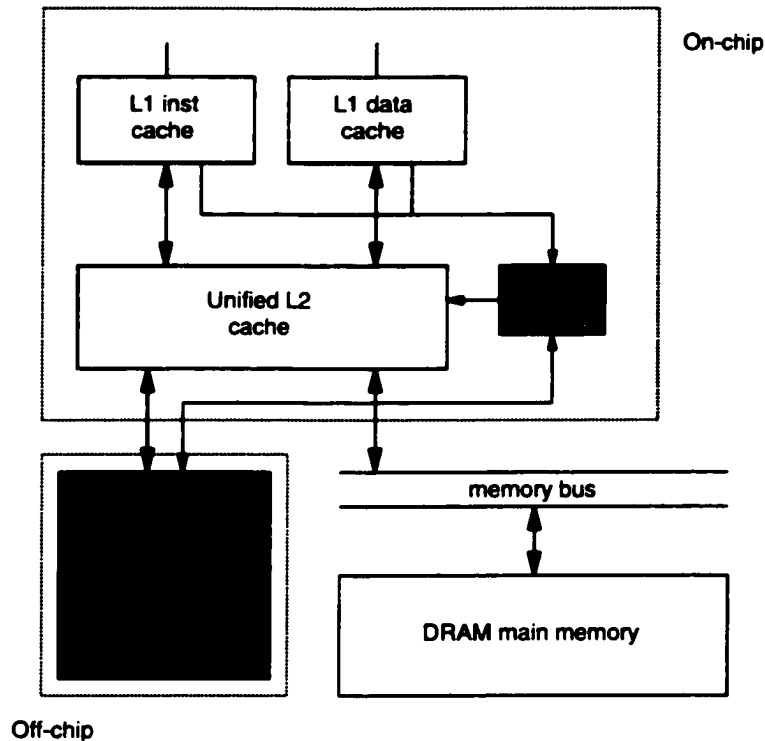
Although careful tuning of DRAM parameters can effectively improve memory performance, its workload dependent feature may limit its usage in practice. In order to address this limit, we present a workload independent approach by focusing on optimizing fine-grain priority scheduling, and show its effectiveness using SPEC2000 benchmark programs. In addition to supporting workload independent configurations, fine-grain priority scheduling can increase the parallelism between processor execution and DRAM memory accesses, and improve the resource utilization of the memory system.

## **Chapter 8**

# **Constructing Large Size and Low Overhead Off-Chip Caches by Cached DRAM**

Large off-chip caches are beneficial to memory-intensive applications whose working sets cannot fit into caches. In this chapter, we present a new design of off-chip cache that uses cached DRAM presented in Chapter 6 to construct a large and low-overhead off-chip cache.

Off-chip caches are normally made by SRAM, the same technology used for on-chip caches. SRAM is fast, but has two major limitations being off-chip cache. First, the size of an SRAM cache is usually limited to less than ten megabytes due to the low density and high cost of SRAM. This size is not large enough for holding the working sets of many memory-intensive applications. Second, because the volume of L3 tags is large and the tags are usually stored off-chip, the L3 tag checking overhead is significant and increases the penalty of cache misses. The performance of some memory-intensive applications can be even harmed by the existence of SRAM off-chip cache due to increased memory access



**Figure 8.1:** A Diagram of Memory Hierarchy using CDC.

latency. For instance, a previous study [26] reported that adding a 2MB off-chip L3 SRAM cache to AlphaServer 4100 can degrade the performance of some applications by up to 10%.

To address these two problems, we present CDC (Cached-DRAM Cache) as a substitute of SRAM off-chip cache. In cached DRAM, a small SRAM cache is integrated with the DRAM to exploit the spatial locality that appears in miss streams from the upper level cache [46, 49, 61, 125]. In a CDC, the DRAM storage is called *CDC-DRAM*, and the SRAM cache is called *CDC-cache*. Figure 8.1 presents a memory hierarchy with two-level on-chip caches and an off-chip CDC. We highlight the CDC design here and will discuss the details in Section 8.1:



- CDC-DRAM has the structure of sector cache, where each CDC-DRAM page can hold dozens of on-chip cache blocks.
- Recently accessed CDC-DRAM pages are cached in the CDC-cache, and their tags are stored in the on-chip *CDC tag cache*.
- The CDC is connected to the processor by high-bandwidth data paths. We assume that they are put in the same module by using the multi-chip module (MCM) technology.

Besides providing a much larger storage than an SRAM off-chip cache, the proposed CDC constructing the L3 off-chip cache has following additional advantages:

- *Minimizing on-chip space overhead.* The CDC tags are stored with data pages in the CDC-DRAM. Only the tags of pages cached in the CDC-cache are stored on-chip. The storage requirement for the on-chip tag cache is less than 1KB in the default configuration. The on-chip predictor for predicting CDC-DRAM hit/miss is very simple and occupies a very little space.
- *Minimizing cache miss overhead.* Since the on-chip CDC tag cache is very small and the CDC-DRAM hit/miss prediction is very simple, the CDC tag checking and the prediction can be performed in parallel with the L2 cache access for every L1 cache miss. When an L2 cache miss is detected, the results of the CDC tag checking and the prediction are already available. There is no additional miss overhead for correctly predicted CDC-DRAM misses. The performance results show that the pre-

diction accuracy is very high and the miss overhead related to incorrect predictions is insignificant.

In contrast, the tag directory of the SRAM L3 cache is much larger than that of the L2 cache, and has longer latency and cycle time than those of the L2 cache. Thus, there will be a significant overhead for cache misses to the main memory.

- *Exploiting the locality in the CDC-Cache.* Many recent studies have shown that the locality in the cache miss streams is very high [61, 118, 25, 124, 67]. Because of the large block size, the CDC-cache is able to effectively utilize this locality to speedup accesses to the CDC.

We use SimpleScalar to simulate a CDC of 128KB SRAM cache and 64MB DRAM, attached to an 8-way issue 2GHz processor that has a 32KB instruction cache, a 32KB data cache and a 1-MByte unified L2 cache on chip. We compare the performance of the CDC with an 8-MByte SRAM L3 cache with the same processor configuration, using nineteen memory-intensive SPEC CPU2000 benchmark programs. The underlying assumption that the density of the CDC-DRAM is eight times that of the SRAM is conservative. The CDC outperforms the L3 SRAM cache for most programs by up to 64%. Unlike the off-chip SRAM cache, the CDC does not degrade the performance of any programs. The average IPC (harmonic mean) of the system with the cached-DRAM off-chip cache is 11% higher than that of the system with the L3 SRAM cache.

The rest of this chapter is organized as follows. Section 8.1 presents the details of our CDC design. Section 8.2 describes the experiment setup. Section 8.3 presents the performance results. Finally, Section 8.4 summarizes the study.

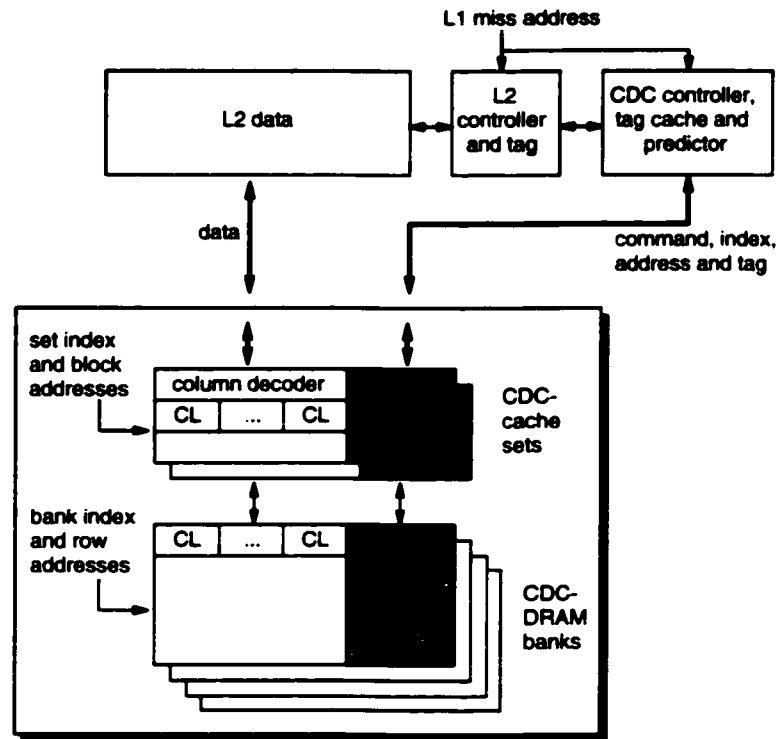
## 8.1 The CDC Design

Figure 8.2 shows a detailed structure of CDC and the data paths connecting the CDC-cache, the CDC-DRAM, the L2 cache, and the tag cache. Each CDC-DRAM page contains a page of data, address tags, and an array of the selection and valid bits. For simplicity, the collection of the address tags and the selection and valid bits of one CDC-DRAM page is called the *page tag* of that page. Inside the CDC, the page tag is stored and transferred together with the associated data blocks. Outside the CDC, the data blocks are selected and transferred to and from the L2 cache, and the tags are transferred to and from the tag cache.

For convenience of discussion, the following configurations and parameters are assumed if they are not mentioned specifically. The CDC-DRAM is 64-MByte and the page size is 4KB. Each page is a sector holding 32 128-Byte L2 cache blocks. The CDC-cache is 4-way associate with 128KB. For the calculation of storage requirement, we assume the physical address in the system is 48-bits long.

### 8.1.1 On-chip CDC Controller, Tag Cache, and Predictor

The tag cache stores the tags of pages that are cached in the CDC-cache. For each L1 cache miss, the address is compared with the associated address tags in the tag cache in parallel with the L2 cache access. Since the tag cache is very small, the comparison can be done before the L2 cache access finishes. If a match is found and the valid bit is set, the demanded data block exists in CDC-cache, and its set index and block index in the CDC-cache are known. The CDC controller sends a request to fetch the block from the



**Figure 8.2:** The CDC structure and the data paths between the CDC-cache, the CDC-DRAM, the L2 cache, and the on-chip tag cache. *CL* represents a block of the cache line size. *T* represents the address tags of a page, *S* represents the selection bits of a page, and *V* represents the valid bits of a page.

CDC-cache to the L2 cache (the CDC-DRAM is not involved). Although the CDC-cache is set associative, it is accessed as a direct mapped cache because the set index and the block index are known at the time of the access.

If a match cannot be found, the data block may exist (1) in the CDC-DRAM, or (2) only in the main memory. The predictor then makes a prediction, directing an access to the CDC-DRAM or to the main memory (we will discuss the predictor design in Section 8.1.4). In the case of a CDC-DRAM access, the CDC-DRAM page is first transferred to the CDC-cache. The page tag is then transferred to the on-chip tag cache to compare with the miss address,

and the demanded data block is selected and transferred simultaneously. The data block is written into the L2 cache if data is actually found in the CDC-DRAM; otherwise, the block is discarded and the data block is read from the main memory.

In the case of a main memory access, both the CDC-DRAM and the main memory are accessed simultaneously. If the block is not found in the CDC, the data fetched from the main memory is sent to the L2 cache, and is also written into the CDC-cache. If the block is found in the CDC-DRAM, the data fetched from the main memory is discarded.

The predictor enables exploiting the locality in the high-bandwidth CDC-DRAM even if the data is not found in the CDC-cache. From another point of view, the predictor filters out unnecessary traffic to the main memory, which could cause congestion on the memory bus and stall the follow-up accesses. Working together, the controller, the tag cache, and the predictor ensure fast tag checking, fast CDC-cache accesses, and low memory bus traffic.

### 8.1.2 CDC-DRAM Mapping

The CDC-DRAM has a sector cache structure [76], where the data part of a page is a sector holding 32 L2 cache blocks. We consider two mapping methods. The first one is the direct mapping scheme of the sector cache, in which each cache block is mapped onto a single location (page and block) in the CDC-DRAM. This method is simple and requires only one tag for each page. However, a potential drawback for this scheme is that the CDC-DRAM storage may not be efficiently used. The selection bit array  $S$  is not needed for this method.

The second method is to use two different mapping schemes for the data and for the tag, as used in the decoupled sector cache [97]. We still use the direct mapping for the data part, because a set associative mapping would require accessing more than one CDC-

DRAM page simultaneously. However, the mapping for the tag part is set associative. Each CDC-DRAM page is associated with  $K$  tags, and blocks from up to  $K$  pages in the physical memory can be cached in the same page. For each block,  $\log K$  selection bits determine the tag that the block is associated with. In the default configuration,  $K = 4$ . In this way, we reduce the chance of page thrashing when two pages conflict in the CDC. For the details of the decoupled sector cache, interested readers may refer to [97]. This decoupled mapping is used in the default configuration.

Although the decoupled cache structure is complex, the major complexity lies in the on-chip tag cache. The only changes in the off-chip CDC are the additional tags and the selection bit array, which do not change the manufacturing process of the underlying cached DRAM. Because the tag cache is very small compared to the L2 cache, the tag cache access time is faster than L2 cache access time, and does not affect the CDC access times. In the default configuration, the tag cache consists of only 32 entries of 208 bits each. The total storage is 832 bytes. For each CDC-DRAM page, the page tag occupies less than 1% of storage.

### 8.1.3 CDC-cache Mapping

The CDC-cache mapping decides how the CDC-DRAM pages are mapped to the cache blocks in the CDC-cache. It is actually decided by the organization of the underlying cached DRAM. Recent studies have shown that set associative cached DRAMs have significant lower cache miss rates than those of the direct mapped ones when used as main memories [118, 61, 125]. Thus, we only consider set associative CDC-cache in this study. It is 4-way associative in the default configuration.

### 8.1.4 The Predictor

Cache hit/miss prediction has been used for a variety of purposes. Authors in [122] use cache hit/miss prediction for improving load instruction scheduling. In order to reduce the memory bandwidth requirement, authors in [113] use miss prediction to dynamically mark which load instruction is cacheable/non-allocatable. We want to apply the prediction technique to predict CDC-DRAM hit/miss for CDC-cache misses, so that accesses to the main memory will be not delayed when the CDC-DRAM miss is correctly predicted.

We adopt a two-level adaptive predictor using a global history register and a global pattern history table (GAg) [121]. The original predictor design is used for dynamic branch predictions. It has a two level structure. The first level is a branch history register which records the current branch pattern. The second level is a pattern history table that uses saturating up-down counters to record the branch history for that pattern. When a branch is encountered, the saturating counter indexed by the current pattern is used to predict whether the branch should be taken or not. The real branch behavior will be fed back to the history register and the counter.

We adapted the prediction mechanism so that the miss pattern instead of the branch pattern is used to train the predictor. The predictor is simple with a low implementation cost. For example, a predictor with an 8-bit history register, a 256-entry pattern history table, and 2-bit saturating counters requires only 520-bit storage. The additional cost to implement the logic is also small. The predictor is fast enough so that the prediction will finish sooner than the L2 cache access.

There may be other predictor design alternatives. However, we have found this simple

scheme is very effective to predict the CDC-DRAM hit/miss, as will be shown in Section 8.3.

For this reason, we have not explored other alternatives.

### 8.1.5 Write Policy

The CDC can be organized as a write-through cache or a writeback cache. The write-through policy is simple to implement for the CDC, but it increases the traffic to the main memory. The writeback policy is more complicated for the CDC. When a CDC-cache miss is encountered, the CDC-DRAM must be accessed for correctness no matter whether the access is predicated a hit or miss (the predictor still works to filter out unnecessary accesses to the main memory). Furthermore, replacing a CDC-DRAM page with multiple dirty blocks will cause a burst of writebacks to the main memory. Additional buffers may be needed to hold those dirty blocks.

We believe the write policy has little influence on the CDC performance for the following reasons. Writeback L2 cache has been extensively used to reduce the memory traffic. Furthermore, a recent study shows that eager writeback [65] can minimize the effect of write traffic on the overall system performance (this technique may also be applied to writeback CDC to avoid the bursts of writebacks). Thus, we only consider write-through CDC in our study.

### 8.1.6 Cache Coherence in Multiprocessor Environment

The CDC design can be used in multiprocessor systems with large main memories as well, in which cache coherence must be maintained. For simplicity, we only discuss the cache coherence issue for bus-based shared-memory symmetric multiprocessors (SMPs) with a



write invalidate, snooping protocol. Maintaining cache coherence for CDC is simplified by using the write-through policy. Because the main memory has a consistent copy for data cached in CDC, one processor does not need to check its CDC for read requests from another processor. Invalidations may take a longer time for data blocks in a CDC than those in an on-chip cache. A queue can be used to buffer those invalidation requests when the CDC is updated. Every access to CDC should be checked with the queue to avoid accessing stale data. After the CDC is updated, the related request can be released from the queue. Currently, the simulator only simulates a uniprocessor system without the maintenance of cache coherence.

## 8.2 Experimental Setup

The simulation parameters are listed in Table 8.1. We use SimpleScalar 3.0 [14] to simulate an 8-way issue, 2GHz processor, and use the SPEC CPU2000 benchmark [107] as the workload. We add the simulations of the CDC, the MSHR (miss information/status holding register), the writeback buffer, and the DRAM main memory system. The L1 and L2 cache latencies are estimated using the Cacti model [85], assuming the  $0.13\mu m$  technology is used.

The latency of the L3 SRAM cache is estimated as follows. We assume that the L3 SRAM cache is on a separate die but in the same module with the CPU, using multichip module technology. The cache is connected to the CPU through a 1GHz, 32-byte wide internal bus (this is not aggressive as the IA-64 Itanium processor cartridge has a 16-byte internal bus that operates at the full CPU speed [94]). The access latency is 9.12ns, estimated by using the Cacti model, which is nineteen CPU cycles. Adding two bus cycles

Processor speed	2GHz, 8-way issue
RUU (register update unit)	128-entry
LSQ (load/store queue)	32-entry
L1 inst. cache	32KB, 4-way associative 64-byte block, 2 cycle latency
L1 data cache	32KB, 4-way associative 64-byte block, 2 cycle latency
Unified L2 cache	1MB, 8-way associative 128-byte block, 8 cycle latency
MSHR	32 entries
Writeback buffer	16 entries
Memory bus bandwidth	6.4 GB/s
DRAM latency (excluding data transfer time)	50ns (100 cycles)
Off-chip cache bandwidth	25.6GB/s
L3 SRAM cache	8MB, direct mapped, 128-byte block, 23 cycle latency
CDC-cache	16 block, 4-way associative, 20 cycle latency
CDC-DRAM	64MB, 4KB page size, 16 banks, 20ns (40 cycles) row access latency
CDC hit/miss predictor	GAg with 8-bit history register and 2-bit saturating counter

**Table 8.1:** Simulation parameters.

for bus delay, the total latency for transferring the first trunk is 23 CPU cycles. Adding three more bus cycles for transferring the following trunks, the total latency to fetch a 128-byte L2 block is 29 CPU cycles.

The latency of the CDC-cache is estimated as follows. Since the CDC-cache tag checking is decoupled from a CDC-cache access, the block index is known before accessing the CDC-cache. Thus, the CDC-cache is accessed like a direct mapped cache. The access time is estimated as 4.7ns by the Cacti model. However, the layout of CDC-cache may not be optimized as the Cacti model expects. Nevertheless, the Enhanced DRAM product has achieved an on-chip SRAM access time of 10.6ns [32]. Thus, we assume that the CDC-cache access time is 8ns (16 CPU cycles), which should be a conservative value. The total

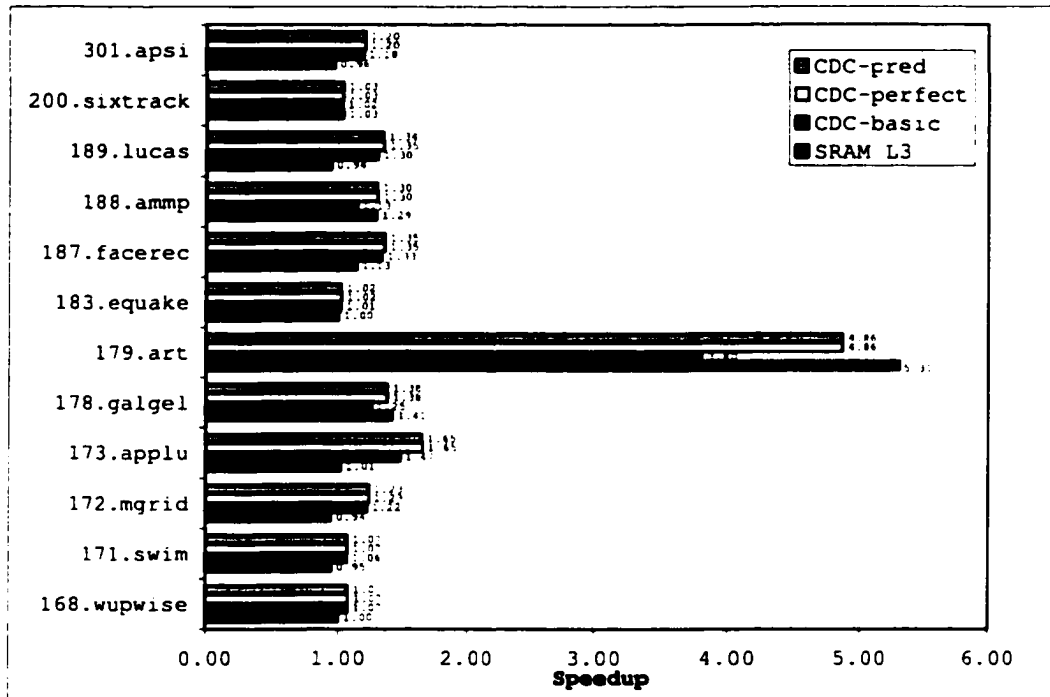
latency to fetch a 128-byte L2 block from the CDC-cache is 26 CPU cycles.

We simulate a DRAM main memory system that supports split bus transactions and can schedule reads prior to earlier writes. At the DRAM level, we assume the close-page mode and auto-precharge are used, and the bank conflict is negligible because of the large number of banks in today's DRAM. The memory parameters are based on an 800MHz 4-channel Direct Rambus DRAM system. Each channel has 1.6GB/s bandwidth and the total bandwidth is 6.4GB/s. The initial DRAM latency is the delay from the time of sending the request to DRAM to the time of receiving the first bit of data. The delay includes 20ns row access time, 20ns column access, and 10ns bus delay. The total delay to fetch a 128-byte block from the main memory is 70ns (140 processor cycles).

We use the precompiled SPEC CPU2000 binaries [117]. All programs are fast forwarded by four billion instructions and are executed for 200 million instructions.

### 8.3 Performance Results

We present the performance improvements of the four L3 cache variations, the SRAM-L3, the CDC-basic, the CDC-predict, and the CDC-perfect, by their speedups over a system that has only the DRAM main memory below the on-chip caches in the memory hierarchy. The SRAM-L3 represents the 8MB SRAM off-chip L3 cache. The CDC-basic, the CDC-predict, and the CDC-perfect behave differently when CDC-cache misses happen. In the CDC-basic, the CDC controller always fetches the data from the main memory. It is used to show the performance by exploiting only the locality in the CDC-cache. In CDC-perfect, the CDC controller magically knows whether the valid block is in the CDC-DRAM or not.



**Figure 8.3:** Speedup for SPECfp2000 programs.

Both the CDC-basic and the CDC-perfect do not incur an overhead for main memory accesses when CDC-cache misses happen. The CDC-predict uses the predictor (discussed in Section 8.1) to predict where to fetch the data. There is an overhead for a wrongly predicated CDC-DRAM hit.

Figure 8.3 shows the speedups of the L3 variants for the twelve SPECfp2000 programs, and Figure 8.4 for the seven SPECint2000 programs. Table 8.2 shows the hit rates of those L3 variants for all the nineteen programs. To show how effectively the CDC-DRAM holds the working sets of programs, we also present the hit rates of a 64-MByte SRAM L3 cache. The CDC-DRAM hit rates are almost the same as those of the 64-MByte L3 cache.

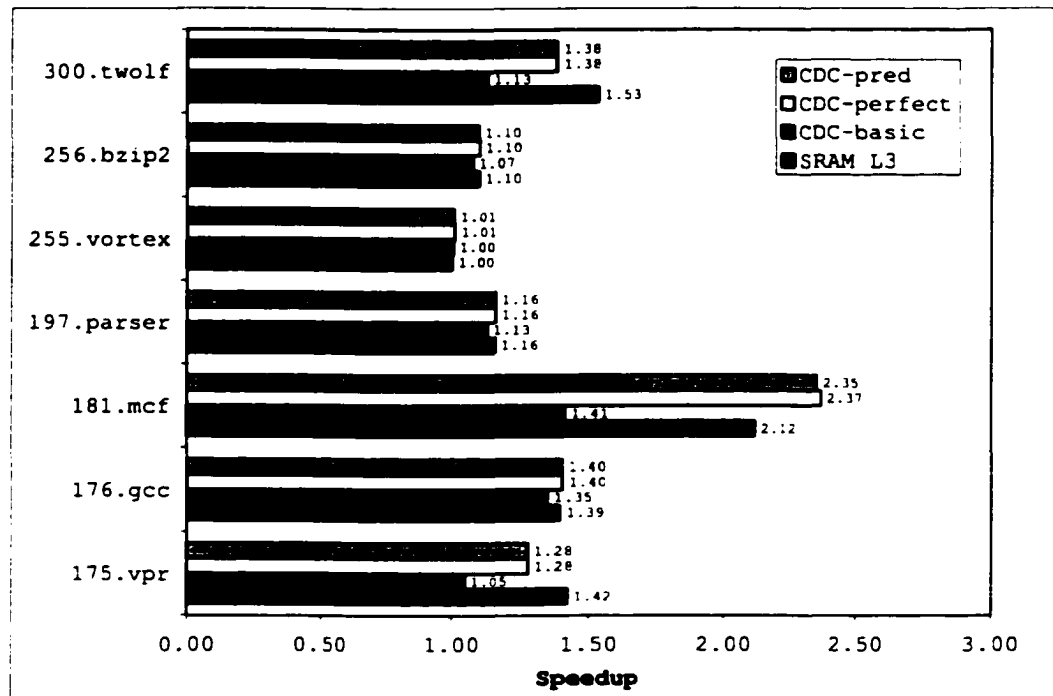


Figure 8.4: Speedups for SPECint2000 programs.

### 8.3.1 Performance of SRAM-L3

The SRAM-L3 provides at least 10% speedups (see Figure 8.3 and Figure 8.4) for ten of the nineteen programs. The average speedup (harmonic mean [44]) is 45% for the ten programs, and 19% for all the nineteen programs. This confirms the effectiveness of large off-chip cache for memory-intensive applications. The SRAM-L3 hit rates range from 38% to 99.8%. For four programs, *175.vpr*, *300.twolf*, *178.galgel*, and *179.art*, the SRAM-L3 hit rates are as high as 97% or more.

However, the speedups of SRAM-L3 for four programs, *171.swim*, *172.mgrid*, *189.lucas*, and *301.apsi*, are negative ( $-3\% \sim -5\%$ ). The hit rates for those programs (see Table 8.2) range from 0.2% to 1.0%, which means the SRAM-L3 is almost useless for those programs.

Program	CDC-DRAM	CDC-cache	8M SRAM L3	64M SRAM L3
176.gcc	95.0%	80.5%	93.3%	95.0%
197.parser	67.3%	51.5%	66.6%	67.3%
256.bzip2	68.9%	43.5%	68.8%	68.9%
168.wupwise	25.5%	23.9%	17.8%	25.5%
172.mgrid	40.7%	38.5%	1.0%	40.7%
178.galgel	97.6%	71.7%	97.5%	97.6%
183.equake	17.7%	10.6%	11.1%	17.7%
188.amp	94.3%	37.6%	79.7%	94.3%
200.sixtrack	72.9%	57.5%	72.9%	72.9%
Average	63.8%	42.1%	49.6%	63.8%

Table 8.2: Hit rates of the selected SPEC CPU2000 programs.

Furthermore, the miss overhead increases the main memory access latency, causing the performance losses. The five other programs have miss rates from 10% to 18%, and the speedups are from 1% to 4%. The SRAM-L3 is not big enough for those programs, and the miss overhead hinders the performance improvement. Only five programs in all the nineteen programs have hit rates more than 80%, indicating a large room for further improvement.

The SRAM-L3 is more effective for the SPECint2000 programs than the SPECfp2000 programs. The average speedup of SRAM-L3 for the seven SPECint2000 programs is 31%, and 13% for the twelve SPECfp2000 programs.

### 8.3.2 Comparisons between CDC-predict and SRAM-L3

The CDC-predict provides at least 10% speedups (see Figure 8.3 and Figure 8.4) for fourteen programs. The average of the ten most significant speedups is 56%, and the average speedup for all the nineteen programs is 29%. Thus, the CDC-predict is beneficial to more programs than the SRAM-L3, and provides a higher average speedup. The average harmonic mean of IPC for all the nineteen programs using the CDC-predict is 11% higher than that of using the SRAM-L3. As expected, it causes no performance loss for all the programs.

The CDC-predict does not outperform the SRAM-L3 on every program. For the four programs, *175.vpr*, *300.twolf*, *178.galgel*, and *179.art*, the SRAM-L3 hit rates range from 97.5% to 99.8%, while the performance of CDC-predict is 2.3% ~ 8.8% lower than that of SRAM-L3. However, the CDC-predict outperforms the SRAM-L3 by 6.1% ~ 59.6% for eight programs, and the average speedup over the SRAM-L3 for the eight programs is 23%. For the three programs *171.swim*, *172.mgrid*, and *189.lucas* that the SRAM-L3 hit rates are only 0.2% ~ 1.0%, the CDC performs much better than the SRAM-L3.

The SPECint2000 programs tend to have smaller working sets and less spatial locality than the SPECfp2000 programs. The average speedups of the CDC-predict for both the SPECfp2000 and the SPECint2000 programs are 30%, while the same speedups of the SRAM-L3 are 13% and 31%, respectively. The CDC-predict performs significantly better than the SRAM-L3 for the twelve SPECfp2000 programs, and very close for the seven SPECint2000 programs <sup>1</sup>.

In general, the SRAM L3 cache outperforms the CDC-predict only when the applica-

---

<sup>1</sup>We include all programs in the SPEC CPU2000 suite that the memory stall time is a large portion of the total execution time.

tion's working sets can fit in the L3 cache. However, only four in all the nineteen programs belong to this category, and the CDC-predict can perform comparably with the SRAM-L3 even for those programs. The SRAM L3 cache performs poorly for programs with large working sets, and the CDC-predict outperforms the SRAM-L3 significantly. For the other programs, the CDC-predict performs the same well or slightly better than the SRAM-L3.

### 8.3.3 Comparisons of the CDC Variants

We compare the performance of CDC-basic, CDC-perfect, CDC-predict, and SRAM-L3 to distinguish (1) the effectiveness of exploiting CDC-DRAM locality, and (2) the effectiveness of using the prediction. As discussed in Section 8.1, if the data can be found in the CDC-DRAM but not in the CDC-cache, it is still performance-beneficial to access the CDC-DRAM instead of the main memory. This is especially true for bandwidth-bound applications. The difference between CDC-prefect and CDC-basic shows this effect. We use the GAg two-level adaptive prediction to predict CDC-DRAM hits. The difference between CDC-predict and CDC-perfect shows the effectiveness of this prediction.

CDC-perfect outperforms CDC-basic significantly for programs that have high CDC-DRAM hit rates but relatively low CDC-cache hit rates (see Table 8.2). All programs that the SRAM L3 cache outperforms CDC-basic are in this category, because CDC-DRAM can hold a working set if the SRAM L3 cache can hold it. For example, *179.art* is a program that SRAM-L3 achieves a 99.8% hit rate and CDC-basic achieves a 95.7% CDC-cache hit rate. Because it is very memory-intensive and bandwidth-bound, the program runs 28% slower on CDC-basic than on SRAM-L3. With CDC-perfect, those misses are directed to CDC-DRAM, making the program run only 8% slower on the CDC-predict than on



Program	175.vpr	176.gcc	181.mcf	197.parser	255.vortex
Accuracy	99.2%	95.4%	98.4%	97.8%	80.7%
Program	256.vortex	300.twolf	168.wupwise	171.swim	172.mgrid
Accuracy	80.7%	81.2%	81.2%	98.4%	98.6%
Program	173.applu	178.galgel	179.art	183.equake	187.facerec
Accuracy	98.2%	99.4%	99.9%	84.6%	97.8%
Program	188.amp	189.lucas	200.sixtrack	301.apsi	
Accuracy	99.2%	93.0%	98.3%	99.9%	

**Table 8.3:** The accuracies of the CDC-DRAM hit/miss prediction.

SRAM-L3. Program *181.mcf* performs worse on the CDC-basic than on SRAM-L3, but performs better on CDC-perfect than on SRAM-L3. CDC-basic achieves a 52.5% cache-hit rate for this program, and SRAM-L3 achieves a relatively better hit rate, 76.9%. However, CDC-DRAM hit rate is 99.0%, thus CDC-perfect can avoid almost all accesses to the main memory and outperform SRAM-L3.

CDC-predict achieves almost the same performance as CDC-perfect for all the programs, as shown in Figure 8.3 and Figure 8.4. Table 8.3 shows the prediction accuracies of the predictor, which is more than 90% for sixteen programs, and more than 95% for thirteen applications. The lowest accuracy is 80.7%. For three applications with prediction accuracy lower than 90% (*255.vortex*, *168.wupwise*, and *183.equake*), the performance of CDC-basic is already very close to CDC-perfect, which indicates the accuracy of the prediction is insignificant for the overall performance of the three programs.

## 8.4 Summary

We have presented the design and performance evaluation by constructing CDC (cached DRAM cache) as the L3 off-chip cache. We show that the CDC effectively addresses two

major concerns of the SRAM off-chip cache: the relatively small size and the miss overhead.

More applications can benefit from the CDC than from the SRAM off-chip cache and no applications will lose performance due to the existence of CDC.

## Chapter 9

# Conclusions and Future Work

### 9.1 Conclusions

#### 9.1.1 Memory Latency Bottleneck

The performance of computer systems has been rapidly advanced by the improvement of critical systems technologies of processor, interconnect, storage, and memory. In fact, the processor speed has been improved in a pace exceeding the prediction of Moore's Law. It is predicted that a standard processor will be as fast as 3-5 GHz in the year of 2004, and 10-15 GHz in the year of 2010. A CPU cycle time will soon be reduced to less than 0.1 *ns*. Meanwhile, the unit price for one MIPS (million instructions per second) will be no more than 10 cents.

With the advancement of networking technology, the interconnection speed measured by link bandwidths (GByte per second) has improved more than 100% per year. These interconnection switches are used for different interconnections of different components of computer systems, such as CPU-memory interconnects and local area networks. The advancement of networking technology continues to improve the data communication speeds of inter-computers and intra-computers.

The technology of disk storage has also improved rapidly. The storage density doubles every year. The unit price per giga bytes of disk space has improved in an even faster pace. In other words, disk space for computer systems become increasingly large and cheap.

Unfortunately, memory technology is the slowest improving technology compared with the other three technologies. On the positive side, memory bandwidth is improving quickly due to the advancement of networking technology. For example, memory bandwidth now is up 8.5 GB/s, and it is predicted that bandwidth will reach to 40 GB/s in the year of 2010. In addition, the price of DRAM is also dropping rapidly. For example, the unit price for one MByte DRAM is more than \$1,000 in 1976, while the price for the same size memory space is less than \$1.00 now. We will continue to see increasingly large memories and improved data transfer speed between the memory and caches. However, memory latency has improved in a much slower pace (5-7% per year) compared with the CPU speed. The relative memory latency will at least double every 3 to 4 years. Although the unit price of DRAMs alone decreases consistently, the improvement still lags behind the cost improvement of other system components. Thus, the cost of memory in a typical system grows relatively at about 30% per year! The memory system has been characterized as the “single most costly single component of system excluding storage subsystem”, and the “single most costly single component of storage subsystem (including disks themselves)” [102].

In this dissertation, we have addressed this important issue of computer systems: memory latency reduction. We have proposed several software and hardware methods to achieve the goal of reducing memory stall time.

### 9.1.2 Our Approaches and Their Effectiveness

In a typical computer system, a multi-level caching organization has become standard, which provides fast data accesses before the data have to be accessed in DRAM core. The existence of these caches in a computer system, such as L1, L2, L3, TLB, and DRAM row buffers, does not mean that data locality will be automatically exploited. The effective usage of the memory hierarchy mainly depends on how data is allocated and how memory accesses are scheduled, which is the major theme of this dissertation.

In the first part of the dissertation, we have presented a case study of optimizing cache performance of bit-reversals. We have shown that a program can be effectively reconstructed based on its specific data access patterns and cache structures of the systems to reduce cache miss rates. This software-library oriented approach is cost- and execution-effective for those commonly and routinely used programs.

Although the next three DRAM memory projects presented in the dissertation attempt to reduce the memory latency by using different technical approaches, they are related and can complement each other.

In order to reuse the data in the DRAM row buffer, the data must stay there for a reasonable period of time. Our study shows that a conventional memory interleaving scheme causes severe row buffer conflicts so that the data is frequently replaced, and the locality in the access stream cannot be exploited. We propose a simple and effective memory interleaving scheme to reduce or even eliminate those row buffer conflicts. The hardware cost of this scheme is trivial. This technique can maximize data reuse in the row buffer and significantly reduce the access frequency to the DRAM core, effectively shortening the

average memory access latency.

Another approach to reduce memory latency is to make the processor get the required data as quickly as possible. As we have discussed earlier, the processor has increasingly long memory stall time waiting for relative slow memory accesses. One method to reduce memory latency in this direction is to effectively reorder the memory accesses by giving more urgently required data a higher priority. We propose a fine-grain priority scheduling scheme for data accesses on multi-channel memory systems, effectively exploiting available bandwidth and concurrency and significantly reducing overall memory latency. The memory interleaving technique can be integrated with fine-grain priority scheduling, e.g., the scheduling prioritizes accesses to data which is already in the row buffer.

In the final part of the dissertation, we first evaluate the performance of cached DRAM and its design alternatives with ILP processors. Compared with DRAM row buffers, the on-memory cache in the cached DRAM can further reduce miss rates and allow faster accesses to cached data. We have shown that the performance improvement is significant for memory-intensive workloads running on ILP processors. However, the better performance comes with an increase in the manufacturing cost. Therefore, memory system architects should consider the trade-off between cost-effectiveness and high performance gains in their designs.

We then propose a new memory hierarchy organization that uses the cached DRAM to construct a very large off-chip cache. We show that this structure outperforms a standard memory system with an off-chip L3 cache for memory-intensive applications. This organization is particularly beneficial to the applications whose working sets are larger than the L3 cache. The interleaving technique is complementary to this organization because exploiting

row buffer locality will further improve the memory performance.

Table 9.1 presents the relationships among the four techniques. The sign of “+” means the two techniques are complementary to each other, while the sign of “-” means the two techniques may conflict. All the techniques are complementary to each other, except for the permutation-based interleaving technique and the cached DRAM. When the cumulative row buffer size increases to a certain threshold, the row buffer conflicts in the DRAM will be minimized. Under such a condition, the permutation-based interleaving technique will no longer be effective.

	Interleaving	CDRAM	Scheduling	CDC
Interleaving		-	+	+
CDRAM	-		+	+
Scheduling	+	+		+
CDC	+	+	+	

**Table 9.1:** Relationship of the four techniques: permutation-based page-interleaving (Interleaving), Cached DRAM (CDRAM), fine-grain memory access scheduling (Scheduling), and CDC (cached-DRAM cache).

## 9.2 Future Work

### 9.2.1 Effective Memory Access Scheduling for Future Processors

The memory access scheduling has been an increasingly complex issue for processors that exploit instruction-level parallelism (ILP) aggressively. As more aggressively speculative techniques are used in ILP processors, we believe that memory access scheduling should consider more information from the processor. Processors have extensively exploited speculative execution techniques to boost instruction-level parallelism. One example is branch

prediction [121], which allows the processor to guess the output of a conditional branch instruction and to speculatively execute the program along the predicted path. Another example is value prediction [68], which allows the processor to reuse the previous computation of a block of instructions. Although speculative execution is necessary and effective, it introduces speculative memory references that may cause false cache misses and increase memory traffic. Memory access scheduling can reduce or eliminate this negative effect. One approach is to prioritize accesses from non-speculative memory references over those from speculative references. Another approach is to hold memory accesses from speculative memory references for a certain amount of time until the related speculation is resolved.

The recent introduction of precomputation-based prefetching techniques [4, 8, 21] exposes much more memory access concurrency to the scheduling. Those techniques can accurately predict future memory accesses by executing a large amount of instructions in a speculative mode. They can also provide the information about critical memory blocks so that the fine-grain memory access scheduling which we have proposed can be applied to memory accesses in the speculative execution. On the other hand, memory access scheduling can improve the accuracy of the speculative execution because it shortens the latency of critical memory blocks and reduces the time length of speculative execution. We believe that the performance can be further improved by combining the two approaches.

The complexity of memory access scheduling lies in the interconnection and interactions between the processor and the memory controller. The processor will need to pass more information associated with each access request. This information may be updated dynamically, for example, when an instance of speculation is resolved. Our study on the fine-grain memory access scheduling has included preliminary discussions on the complexity issue.



However, further and more detailed study must be done to determine the best trade-off between complexity and performance improvement.

### 9.2.2 Improving the Performance of CDC

Our CDC design uses cached-DRAM to construct a large, low-overhead off-chip cache. We believe this design meets the future needs of memory-intensive applications, which cannot be met by SRAM off-chip caches. In our preliminary study, we have shown that the CDC outperforms SRAM cache for most workloads. There are two approaches to further improve the performance of CDC. The first approach is to identify frequently used data of spatial locality and to use CDC as the preferable place to cache the data. The benefit is two-fold. First, it improves the spatial locality of the data cached in CDC. Consequently, the data is more likely to be found in CDC-cache. Second, more space in the relatively small on-chip SRAM caches is spared for data of poor spatial locality but of good temporal locality. This type of data is common in programs using dynamically-allocated data structures. Thus, those programs will benefit from the approach when on-chip caches are not large enough to hold both types of data.

Another approach is to use aggressive prefetch techniques with CDC. There can be three forms of prefetch. First, prefetch can be done to prompt data from the CDC-DRAM to the CDC-cache, utilizing the huge internal bandwidth inside the CDC. Unlike prefetching from DRAM main memory to on-chip caches, the waste of bandwidth due to inaccurate prefetch has a negligible impact on performance. Prefetch can also be done from the CDC to on-chip caches, further reducing the latency of accessing data stored in CDC. The bandwidth between the on-chip caches and the CDC is several times higher than the bandwidth between

the on-chip caches and the DRAM main memory. Thus, the possible waste of bandwidth has a much smaller impact on the overall performance. The third choice is to prefetch data from the DRAM main memory to the CDC. Since the CDC is much larger than on-chip caches, this type of prefetching causes much less cache pollution than prefetching data directly into on-chip caches.

## Appendix A

# The Bit-reversal Program Using the Padding Method

```
void bit_reversal()
{
    int blk, blk_rev, i, i_rev, j, jump = PAD_LENGTH, k;

    int D = N >> 2*b, d = n - 2*b;

    DATA_TYPE *Xp[B];

    DATA_TYPE *Yp, f0, f1, f2, f3;

    for (i = 0; i < B; i++)

        Xp[i] = &X[bitrev_tbl[i]*jump];

    for (blk = 0; blk < D; blk++) {

        bitrev(blk, blk_rev, d);

        for (i = 0; i < B; i++) {
```

```
    i_rev = bitrev_tbl[i];

    k = (blk << b) + i;

    Yp = &Y[(blk_rev<<b) + (i_rev<<(n-b))];

    for (j = 0; j < B; j += 4) {

        f0 = Xp[j][k];

        f1 = Xp[j+1][k];

        f2 = Xp[j+2][k];

        f3 = Xp[j+3][k];

        Yp[j] = f0;

        Yp[j+1] = f1;

        Yp[j+2] = f2;

        Yp[j+3] = f3;

    }

}

}
```

# Bibliography

- [1] A. AGARWAL AND S. PUDAR. Column-associative caches: a technique for reducing the miss rate for direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, San Diego, CA, 1993.
- [2] A. AGARWAL, M. HOROWITZ, AND J. HENNESSY. Cache performance of operating systems and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.
- [3] G. M. AMDAHL, G. A. BLAAUW, AND F. J. BROOKS, JR. Architecture of the IBM system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.
- [4] M. M. ANNAVARAM, J. M. PATEL, AND E. S. DAVIDSON. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, Göteborg, Sweden, 2001.
- [5] J.-L. BAER AND W.-H. WANG. Architectural choices for multi-level cache hierarchies. In *Proceedings of the International Conference on Parallel Processing*, pages 258–261, 1987.
- [6] J.-L. BAER AND W.-H. WANG. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, Honolulu, Hawaii, 1988.
- [7] D. H. BAILEY. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, March 1990.
- [8] R. BALASUBRAMONIAN, S. DWARKADAS, AND D. H. ALBONESI. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 26–37, Göteborg, Sweden, 2001.
- [9] F. BASKETT AND A. SMITH. Interference in multiprocessor computer systems with interleaving memory. *Communications of the ACM*, 19(6):327–334, June 1976.
- [10] B. BATSON AND T. N. VIJAYKUMAR. Reactive-associative caches. In *Proceedings of the International Symposium on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, 2001.

- [11] B. BERSHAD, D. LEE, T. ROMER, AND B. CHEN. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, San Jose, CA, 1994.
- [12] B. BLACK, B. RYCHLIK, AND J. P. SHEN. The block-based trace cache. *ACM SIGARCH Computer Architecture News*, 27(2):196–207, May 1999.
- [13] D. BURGER, J. R. GOODMAN, AND A. KÄGI. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, Philadelphia, PA, 1996.
- [14] D. BURGER. System-level implications of processor-memory integration. Technical Report CS-TR-1997-1349, University of Wisconsin, Madison, June 1997.
- [15] D. C. BURGER AND T. M. AUSTIN. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
- [16] B. CALDER, D. GRUNWALD, AND J. EMER. Predictive sequential associative cache. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 244–253, San Jose, CA, 1996.
- [17] D. CALLAHAN, K. KENNEDY, AND A. PORTERFIELD. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, 1991.
- [18] M. CEKLEOV AND M. DUBOIS. Virtual-address caches, part 1: Problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, September/October 1997.
- [19] C.-L. CHEN AND C.-K. LIAO. Analysis of vector access performance on skewed interleaved memory. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 387–394, Jerusalem, Israel, 1989.
- [20] S. CHO, P.-C. YEW, AND G. LEE. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proceedings of the 32st IEEE/ACM International Symposium on Microarchitecture*, pages 100–110, Haifa, Israel, 1999.
- [21] J. D. COLLINS, H. WANG, D. M. TULLSEN, C. HUGHES, Y.-F. LEE, D. LAVERY, AND J. P. SHEN. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, Göteborg, Sweden, 2001.
- [22] Compaq Computer Corp. *Technology for performance: Compaq professional workstation XP1000*, January 1999. White paper (document number ECG050/0199).
- [23] J. W. COOLEY AND J. W. TUKEY. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965.

- [24] V. CUPPU AND B. JACOB. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 62–71, Göteborg, Sweden, 2001.
- [25] V. CUPPU, B. JACOB, B. DAVIS, AND T. MUDGE. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233, Atlanta, GA, 1999.
- [26] Z. CVETANOVIC AND D. D. DONALDSON. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.
- [27] B. T. DAVIS. *Modern DRAM Architectures*. PhD thesis, University of Michigan, Department of Computer Science and Engineering, 2001.
- [28] R. DESIKAN, D. BURGER, AND S. W. KECKLER. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, Göteborg, Sweden, 2001.
- [29] A. EDELMAN. Optimal matrix transposition and bit reversal on hypercubes: all-to-all personalized communication. *Journal of Parallel and Distributed Computing*, 11(4):328–331, April 1991.
- [30] J. EDMONDSON, P. RUBINFELD, R. PRESTON, AND V. RAJAGOPALAN. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.
- [31] J. S. EMER AND D. W. CLARK. A characterization of processor performance in the VAX-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 301–310, Ann Arbor, MI, 1984.
- [32] Enhanced Memory Systems Inc. *64 Mbit ESDRAM Components, Product Brief r1.8*. 2000. product manual.
- [33] D. M. W. EVANS. An improved digit-reversal permutation algorithm for the fast fourier and hartley transforms. *IEEE Transactions on Acoustics, Speech, Signal Processing*, ASSP-35:1120–1125, 1987.
- [34] K. I. FARKAS, P. CHOW, N. P. JOUPPI, AND Z. VRANESIC. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, Denver, Colorado, 1997.
- [35] K. I. FARKAS AND N. P. JOUPPI. Complexity/Performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Chicago, IL, 1994.
- [36] D. H. FRIENDLY, S. J. PATEL, AND Y. N. PATT. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th IEEE/ACM*

- International Symposium on Microarchitecture*, pages 24–33, Research Triangle Park, NC, 1997.
- [37] Q. S. GAO. The chinese remainder theorem and the prime memory system. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 337–340, San Diego, CA, 1993.
  - [38] K. S. GATLIN AND L. CARTER. Memory hierarchy considerations for fast transpose and bit-reversals. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 33–43, Orlando, FL, 1999.
  - [39] J. GIBSON, R. KUNZ, D. OFELT, M. HOROWITZ, J. HENNESSY, AND M. HEINRICH. FLASH vs. (simulated) FLASH: closing the simulation loop. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, San Jose, CA, 1998.
  - [40] A. GONZALEZ, M. VALERO, N. TOPHAM, AND J. M. PARCERISA. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th International Conference on Supercomputing*, pages 76–83, Vienna, Austria, 1997.
  - [41] G. GROHOSKI AND C. MOORE. Instruction buffer to support multiple fetches and dispatches. *IBM Technical Disclosure*, 21(12):25–40, September 1989.
  - [42] D. T. HARPER III AND J. R. JUMP. Performance evaluation of vector accesses in parallel memories using a skewed storage scheme. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 324–328, Tokyo, Japan, 1986.
  - [43] C. A. HART. CDRAM in a unified memory architecture. In *Proceedings of CompCon '94*, pages 261–266, Los Alamitos, CA, 1994.
  - [44] J. L. HENNESSY AND D. A. PATTERSON. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
  - [45] J. L. HENNING. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.
  - [46] H. HIDAKA, Y. MATSUDA, M. ASAKURA, AND K. FUJISHIMA. The cache DRAM architecture: a DRAM with an on-chip cache memory. *IEEE Micro*, 10(2):14–25, April 1990.
  - [47] M. D. HILL. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.
  - [48] S. I. HONG, S. A. MCKEE, M. H. SALINAS, R. H. KLENKE, J. H. AYLOR, AND W. A. WULF. Access order and effective bandwidth for streams on a direct Rambus memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 80–89, Orlando, FL, 1999.



- [49] W.-C. HSU AND J. E. SMITH. Performance of cached DRAM organizations in vector supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 327–336, Los Alamitos, CA, 1993.
- [50] C. J. HUGHES, V. S. PAI, P. RANGANATHAN, AND S. V. ADVE. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, 2002.
- [51] IEEE, Piscataway, NJ. *POSIX P1003.4a: Threads Extension for Portable Operating Systems*, 1994.
- [52] Q. JACOBSON, E. ROTENBERG, AND J. E. SMITH. Path-based next trace prediction. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, Research Triangle Park, NC, 1997.
- [53] T. E. JEREMIASSEN AND S. J. EGGERS. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, California, 1995.
- [54] S. L. JOHNSON AND C.-T. HO. Algorithms for matrix transposition on boolean n-cube configured ensemble architectures. *SIAM Journal on Matrix Analysis and Applications*, 9(3), July 1988.
- [55] T. L. JOHNSON AND W.-M. HWU. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, Denver, Colorado, 1997.
- [56] N. P. JOUPPI. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, 1990.
- [57] T. JUAN, T. LANG, AND J. J. NAVARRO. The difference-bit cache. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 114–120, Philadelphia, PA, 1996.
- [58] A. H. KARP. Bit reversal on uniprocessors. *SIAM Review*, 38(1):1–26, March 1996.
- [59] Y. KATAYAMA. Trends in semiconductor memories. *IEEE Micro*, 17(6):10–17, November/December 1997.
- [60] R. KESSLER AND M. D. HILL. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.
- [61] R. P. KOGANTI AND G. KEDEM. WCDRAM: a fully associative integrated Cached-DRAM with wide cache lines. In *Proceedings of the 4th IEEE Workshop on the Architecture and Implementation of High Performance Communication Systems*, 1997.

- [62] R. P. KOGANTI. WCDRAM: a fully associative integrated Cached-DRAM with wide cache lines. Master's thesis, Department of Electrical and Computer Engineering, Duke University, 1997.
- [63] D. KROFT. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87. Minneapolis, MN, 1981.
- [64] M. S. LAM, E. E. ROTHBERG, AND M. E. WOLF. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–105, Santa Clara, CA, 1991.
- [65] H.-H. LEE, G. TYSON, AND M. FARRENS. Eager writeback – a technique for improving bandwidth utilization. In *Proceedings of the 33rd IEEE/ACM International Symposium on Microarchitecture*, Monterey, CA, 2000.
- [66] D. LEIBHOLZ AND R. RAZDAN. The Alpha 21264: A 500 mhz out-of-order execution microprocessor. In *Proceedings of CompCon '97*, pages 28–36, 1997.
- [67] W. LIN, S. REINHARDT, AND D. BURGER. Reducing dram latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001.
- [68] M. H. LIPASTI. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, May 1997.
- [69] L. LIU. Cache designs with partial address matching. In *Proceedings of the 27th IEEE/ACM International Symposium on Microarchitecture*, pages 128–136, San Jose, CA, 1994.
- [70] B. K. MATHEW, S. A. MCKEE, J. B. CARTER, AND A. DAVIS. Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 39–48, Toulouse, France, 2000.
- [71] S. A. MCKEE AND W. A. WULF. Access ordering and memory-conscious cache utilization. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 253–262, Raleigh, NC, 1995.
- [72] K. S. MCKINLEY AND O. TEMAM. A quantitative analysis of loop nest locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, MA, 1996.
- [73] L. MCVOY AND C. STAELIN. Imbench: Portable tools for performance analysis. In *Proceedings of USENIX 1996 annual technical conference*, pages 279–294, San Diego, California, 1996.

- [74] T. C. MOWRY, M. S. LAM, AND A. GUPTA. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, 1992.
- [75] S. A. MOYER. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, Department of Computer Science, April 1993. Also as TR CS-93-18.
- [76] J.-K. PEIR, W. W. HSU, AND A. J. SMITH. Functional implementation techniques for CPU cache memories. *IEEE Transactions on Computers*, 48(2):100–110, February 1999.
- [77] J.-K. PEIR, W. W. HSU, H. YOUNG, AND S. ONG. Improving cache performance with balanced tag and data paths. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 268–278, Cambridge, MA, 1996.
- [78] J.-K. PEIR, Y. LEE, AND W. W. HSU. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, San Jose, CA, 1998.
- [79] PostgreSQL Inc. *PostgreSQL 6.5*. <http://www.postgresql.org>.
- [80] S. PRZYBYLSKI. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 160–169, Seattle, WA, 1990.
- [81] J. M. RABAEY. *Digital Integrated Circuits, A Design Perspective*. Prentice Hall, Inc., 1996.
- [82] Rambus Inc. *256/288-Mbit Direct RDRAM (32 Split Bank Architecture)*, 2000. <http://www.rambus.com/developer/downloads/r dram.256d.0105-1.1.book.pdf>.
- [83] B. R. RAU. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, Toronto, Canada, 1991.
- [84] B. R. RAU, M. S. SCHLANSKER, AND D. W. L. YEN. The CYDRA 5 stride-insensitive memory system. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 242–246, 1989.
- [85] G. REINMAN AND N. JOUPPI. An integrated cache timing and power model. Technical report, COMPAQ Western Research Lab, 1999.
- [86] G. RIVERA AND C.-W. TSENG. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal Canada, 1998.

- [87] J. RIVERS AND E. DAVIDSON. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the International Conference on Parallel Processing*, pages 151–162, Bloomingdale, IL, 1996.
- [88] J. A. RIVERS, G. S. TYSON, E. S. DAVIDSON, AND T. M. AUSTIN. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*, pages 46–56, Research Triangle Park, NC, 1997.
- [89] S. RIXNER, W. J. DALLY, U. J. KAPASI, B. KHAILANY, A. LÓPEZ-LAGUNAS, P. R. MATTSON, AND J. D. OWENS. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st IEEE/ACM International Symposium on Microarchitecture*, pages 3–13, Los Alamitos, 1998.
- [90] S. RIXNER, W. J. DALLY, U. J. KAPASI, P. MATTSON, AND J. D. OWENS. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, Vancouver, Canada, 2000.
- [91] M. ROSENBLUM, E. BUGNION, S. DEVINE, AND S. A. HERROD. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.
- [92] E. ROTENBERG, S. BENNETT, AND J. E. SMITH. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*, pages 24–35, Paris, France, 1996.
- [93] T. SAKAKIBARA, K. KITAI, T. ISOBE, S. YAZAWA, T. TANAKA, Y. INAGAMI, AND Y. TAMAKI. Scalable parallel memory architecture with a skew scheme. In *Proceedings of the 7th International Conference on Supercomputing*, pages 157–166, Tokyo, Japan, 1993.
- [94] W. A. SAMARAS, N. CHERUKURI, AND S. VENKATARAMAN. The IA-64 Itanium processor cartridge. *IEEE Micro*, 21(1):82–89, January/February 2001.
- [95] F. J. SÁNCHEZ AND A. GONZÁLEZ. Cache sensitive modulo scheduling. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*, pages 338–348, Research Triangle Park, NC, 1997.
- [96] A. SEZNEC. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, 1993.
- [97] A. SEZNEC. Decoupled Secteded Caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 384–393, Chicago, IL, 1994.
- [98] A. SEZNEC. DASC cache. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 134–143, Raleigh, NC, 1995.

- [99] A. SEZNEC AND J. LENFANT. Interleaved parallel schemes: Improving memory throughput on supercomputers. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 246–255, Toronto, Canada, 1992.
- [100] K. SKADRON AND D. W. CLARK. Design issues and tradeoffs for write buffers. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 144–155, Monterey, Mexico, 1997.
- [101] A. J. SMITH. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
- [102] T. B. SMITH. What will have the greatest impact in 2010: the processor, the memory, or the interconnect? Panel in the 8th International Symposium on High-Performance Computer Architecture, 2002.
- [103] K. SO AND R. RECHTSCHAFFEN. Cache operations by MRU change. In *Proceedings of the 1986 International Conference on Computer Design*, pages 584–586, 1986.
- [104] G. S. SOHI. High-bandwidth interleaved memories for vector processors - a simulation study. Technical Report CS-TR-1988-790, University of Wisconsin - Madison, September 1988.
- [105] S. P. SONG, M. DENMAN, AND J. CHANG. The PowerPC-604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.
- [106] F. SPARACIO. Data processing system with second level cache. *IBM Technical Disclosure*, 21(6):2468–2469, November 1978.
- [107] Standard Performance Evaluation Corporation. <http://www.spec.org>.
- [108] Standard Performance Evaluation Corporation. *SPEC CPU95 Version 1.10*, May 1997.
- [109] P. N. SWARZTRAUBER. FFT algorithms for vector computers. *Parallel Computer*, 1(1):45–63, August 1984.
- [110] J. M. TENDLER, J. S. DODSON, J. S. FIELDS JR., , H. LE, AND B. SINHARROY. POWER4 system microarchitecture. *IBM Systems Journal*, 46(1):5–26, 2002.
- [111] J. TORRELLAS, M. S. LAM, AND J. L. HENNESSY. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.
- [112] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification, Revision 3.3.3*, April 1998.
- [113] G. TYSON, M. FARRENS, J. MATTHEWS, AND A. R. PLESZKUN. A modified approach to data cache management. In *Proceedings of the 28th IEEE/ACM International Symposium on Microarchitecture*, pages 93–103, Ann Arbor, MI, 1995.

- [114] M. VALERO, T. LANG, AND E. AYGUADÉ. Conflict-free access of vectors with power-of-two strides. In *Proceedings of the 6th International Conference on Supercomputing*, pages 149–156, Washington, D.C., 1992.
- [115] S. P. VANDERWIEL AND D. J. LILJA. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.
- [116] W.-H. WANG, J.-L. BAER, AND H. M. LEVY. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, Jerusalem, Israel, 1989.
- [117] C. WEAVER. <http://www.simplescalar.org/spec2000.html>. SPEC2000 binaries.
- [118] W. WONG AND J.-L. BAER. DRAM on-chip caching. Technical Report UW CSE 97-03-04, University of Washington, February 1997.
- [119] Y. YAN, X. ZHANG, AND Z. ZHANG. Cacheminer: A runtime approach to exploit cache locality on SMP. *IEEE Transactions on Parallel and Distributed Systems*, 11(4):357–374, April 2000.
- [120] K. C. YEAGER. The MIPS R10000 superscalar microprocessor: Emphasizing concurrency and latency-hiding techniques to efficiently run large, real-world applications. *IEEE Micro*, 16(2):28–40, April 1996.
- [121] T.-Y. YEH AND Y. N. PATT. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, Toronto, Canada, 1992.
- [122] A. YOAZ, M. EREZ, R. RONEN, AND S. JOURDAN. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, Atlanta, GA, 1999.
- [123] C. ZHANG, X. ZHANG, AND Y. YAN. Two fast and high-associativity cache schemes. *IEEE Micro*, 17(5):40–49, September/October 1997.
- [124] Z. ZHANG, Z. ZHU, AND X. ZHANG. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd IEEE/ACM International Symposium on Microarchitecture*, pages 32–41, Monterey, CA, 2000.
- [125] Z. ZHANG, Z. ZHU, AND X. ZHANG. Cached DRAM: A simple and effective technique for memory access latency reduction on ILP processors. *IEEE Micro*, 21(4):22–32, July/August 2001.
- [126] Z. ZHU, Z. ZHANG, AND X. ZHANG. Fine-grain priority scheduling on multi-channel memory systems. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 107–116, Cambridge, MA, 2002.
- [127] J. H. ZURAWSKI, J. E. MURRAY, AND P. J. LEMMON. The design and verification of the AlphaStation 600 5-series workstation. *Digital Technical Journal*, 7(1):89–99, 1995.

## VITA

### Zhao Zhang

Zhao Zhang is born in Changsha, Hunan, China, in 1970. He received his B.S. and M.S. degrees in Computer Science from Huazhong University of Science and Technology (HUST), Wuhan, Hubei, China, in 1991 and 1994, respectively. He entered the Ph.D. program in Computer Science at the College of William and Mary in Fall 1997. He was a research intern in the Hewlett-Packard Labs, Palo Alto, California, in the summer of 2000, participating in a project of improving resource management in Web servers. His research interests are computer architecture and parallel and distributed computing. He is a member of the IEEE, the ACM, and the USENIX.