The Dissertation Committee for Chang Joo Lee
certifies that this is the approved version of the following dissertation:

# DRAM-Aware Prefetching and Cache Management

Committee:

Yale N. Patt, Supervisor

Nur A. Touba

Derek Chiou

Hossein Namazi

Onur Mutlu

# DRAM-Aware Prefetching and Cache Management

**by**

**Chang Joo Lee, B.S.E.; M.S.E.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2010

Dedicated to my family

# Acknowledgments

Many people contributed both directly and indirectly to this dissertation. First of all, I would like to thank the current and previous members of the HPS research group. The HPS group was everything to me during my long graduate student life.

I thank my advisor, Yale N. Patt for giving me the opportunity to work with many people who are great human beings as well as smart and talented, for helping me build a strong foundation in computer architecture, for motivating me to perform serious research, and, most of all, for being patient with me in making good progress on serious research topics.

Many thanks to Onur Mutlu for the mentorship he provided and the professionalism he demonstrated to me. He taught me how to write and publish ideas. He always encouraged me to continue working hard, and his technical feedback and criticism on my work made this dissertation stronger. I also thank him for always being a friend and bearing with my complaints and unreasonable rants. My graduate student life would have been miserable without both his technical and non-technical support.

I thank Veynu Narasiman for working with me on all the topics presented in this dissertation, for correcting both my spoken and written English, and for being a good friend and listening to me whenever I was discouraged or depressed. I cannot forget the valuable discussions with him on both research and life.

I had a wonderful time working with Eiman Ebrahimi on the cache topics proposed in this dissertation. He cared about me on both work and private issues, helped me write and speak English better, and put up with my overreactions, harsh jokes, and complaints. I could not have had any fun in the past three years without him.

I thank José A. Joao for always being a friend and listening to me on both technical and private issues. I enjoyed the time with him when we rebuilt and enhanced our simulation infrastructure environment together.

I also had a great time with Aater M. Suleman, Rustam Miftakhutdinov, and Khubaib during my graduate studies. They provided valuable technical feedback and criticism, and tried to help me all the time. Their feedback on my research also made this dissertation stronger and clearer. Especially, I thank Rustam for always being joyful and enjoying my jokes, and Khubaib for proofreading many chapters of this dissertation.

I thank Hyesoon Kim for providing mentorship and encouraging me to perform meaningful work. I thank Moinuddin K. Qureshi, Francis Tseng, Daniel N. Lynch, Santhosh Srinath, David Thompson, and other previous HPS members for their mentorship and friendship. Many thanks to Leticia Lira for her long-standing administrative support in the HPS group. I also thank recently joined members, Milad Hashemi and Faruk Guvenilir for proofreading some chapters in this dissertation.

Besides the HPS members, I would like to express my gratitude to other people and organizations.

I thank Derek Chiou, Nur Touba, and Hossein Namazi for serving on my dissertation committee and for giving me valuable comments on this dissertation. I also thank Thomas Puzak, Philip Emma, Vijayalakshmi Srinivasan, and James Holt for providing me with a chance to have a great experience as an intern at IBM and Freescale. I gratefully acknowledge the government of Korea and IBM for providing me with fellowships during my graduate studies.

I appreciate the support and friendship of my friends, Dam Sunwoo and Joonsoo Kim. We all had fun making our serious project successful in 382N. I will never forget the great feeling we had at that time.

Special thanks to the “Lunch Bank” lady who delivered a lunch box to me on campus everyday during several semesters, which helped me manage time more

efficiently.

Finally, I would like to thank my grandmother, Jung Hee Yong who passed away during my studies, my parents, Jong Rak Lee and Jong Kyo Kwon, my brother, Seung Hoon Lee, and my wife, Eunyoung Park for giving me their endless love and unconditional support. I dedicate this dissertation to them.

Chang Joo Lee
December 2010, Austin, TX

# DRAM-Aware Prefetching and Cache Management

Chang Joo Lee, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Yale N. Patt

Main memory system performance is crucial for high performance microprocessors. Even though the peak bandwidth of main memory systems has increased through improvements in the microarchitecture of Dynamic Random Access Memory (DRAM) chips, conventional on-chip memory systems of microprocessors do not fully take advantage of it. This results in underutilization of the DRAM system, in other words, many idle cycles on the DRAM data bus. The main reason for this is that conventional on-chip memory system designs do not fully take into account important DRAM characteristics. Therefore, the high bandwidth of DRAM-based main memory systems cannot be realized and exploited by the processor.

This dissertation identifies three major performance-related characteristics that can significantly affect DRAM performance and makes a case for DRAM characteristic-aware on-chip memory system design. We show that on-chip memory resource management policies (such as prefetching, buffer, and cache policies) that are aware of these DRAM characteristics can significantly enhance entire system performance. The key idea of the proposed mechanisms is to send out to the DRAM system useful memory requests that can be serviced with low latency or in parallel with other requests rather than requests that are serviced with high latency

or serially. Our evaluations demonstrate that each of the proposed DRAM-aware mechanisms significantly improves performance by increasing DRAM utilization for useful data. We also show that when employed together, the performance benefit of each mechanism is achieved additively: they work synergistically and significantly improve the overall system performance of both single-core and Chip MultiProcessor (CMP) systems.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Problem

Memory system performance is crucial for high performance computing. Dynamic Random Access Memory (DRAM) is the most commonly used technology for building the main memory system in modern computer systems. Therefore, computer architects need to understand the characteristics of DRAM in order to build high performance memory systems. There are three main performance-related characteristics associated with DRAM: *bank-level parallelism*, *row buffer locality*, and *write-caused interference*.

- Bank-Level Parallelism: A modern DRAM chip consists of multiple banks that can be accessed independently. Memory requests to different DRAM banks can proceed concurrently. Therefore, the requests' access latencies can be overlapped, thereby increasing DRAM throughput. The notion of servicing multiple requests in parallel in different DRAM banks is called DRAM *Bank-Level Parallelism (BLP)*.

- Row Buffer Locality: Each DRAM bank consists of rows and columns of DRAM cells. A row contains a fixed-size block of data (usually several Kbytes). Each bank has a *row buffer* (or *sense amplifier*), and a DRAM access can be made only by reading (writing) data from (to) the row buffer using a column address. To perform a complete access, 1) a row is loaded into the row buffer and 2) the data in the row buffer is read (written to). The row buffer keeps the most recently accessed row in the DRAM bank. A subsequent access to the last accessed row can be serviced significantly faster than

an access to a different row. This concept is referred to as *row buffer locality*. Prioritizing a request among multiple memory requests to the currently open row results in higher DRAM throughput.

- Write-Caused Interference: Write requests interfere with read requests in the DRAM system by causing idle cycles on the DRAM data bus. Once a write is serviced, subsequent reads and even some writes (e.g., writes to different rows in the same bank) cannot be started for a certain amount of time even after the write is fully serviced. This introduces idle cycles on the data bus and in turn degrades DRAM throughput. We call this *write-caused interference* in the DRAM system.

We define a processor's *on-chip memory system* as the collection of the following: 1) the memory controller, 2) the structures that generate main memory requests (e.g., last-level cache and prefetcher structures), 3) the buffer structures which memory requests go through until they are serviced by the DRAM system, and 4) the corresponding management policies associated with 1), 2), and 3). If the on-chip memory system takes into account bank-level parallelism, row buffer locality, and write-caused interference, DRAM performance and in turn system performance can be significantly improved. However, conventional on-chip memory systems do not fully consider these DRAM system characteristics and therefore often do not provide the best system performance. This problem becomes more significant for Chip MultiProcessor (CMP) systems where the DRAM system is shared by multiple cores on a chip. Figure 1.1 shows the average system performance and DRAM data bus utilization for single, 4, and 8-core systems. In this experiment, we used a DDR3 DRAM system [49] and aggressive 4.8 GHz x86 microprocessors [1] with and without an aggressive stream prefetcher [77, 73, 36]. We ran the 20 most memory-intensive SPEC CPU 2000/2006 benchmarks on the

[1] We deliberately chose an aggressive processor frequency to account for future technology advancements. The performance and DRAM bus utilization trends shown here do not change significantly with less aggressive frequencies (e.g., 3.2GHz).

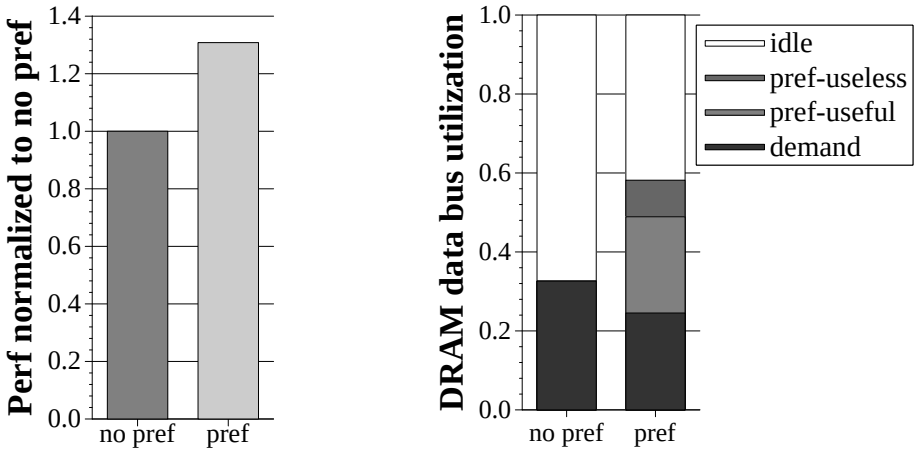
single-core system. We simulated 30 and 20 pseudo-randomly chosen multiprogrammed workloads [39] on the 4-core and 8-core CMP systems respectively. [2] We make four observations from Figure 1.1.

First, with no prefetching, as the number of cores increases DRAM bus utilization increases. This is because multiple applications run together on different cores on the chip and generate more memory requests to the DRAM system.

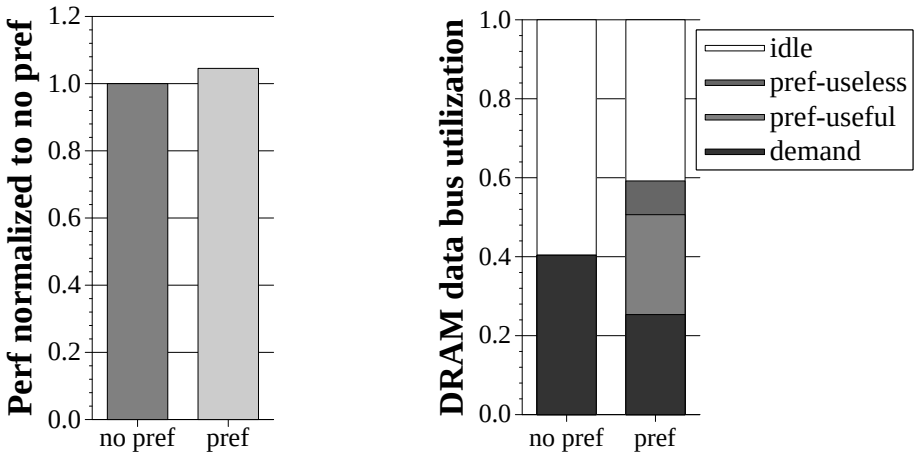
Second, the DRAM data bus is not fully utilized for any of the three systems even with prefetching. For the single and 4-core systems, when the stream prefetcher is employed, bus utilization increases and performance improves (by 30.8% and 4.5%) compared to no prefetching. However, there are still a significant number of idle data bus cycles. One of the main reasons is that conventional on-chip memory systems do not fully take advantage of the DRAM system. They sometimes limit the amount of row buffer locality and bank-level parallelism exploited by the DRAM controller or do not try to minimize write-caused interference. System performance can be improved by exploiting or reducing those idle cycles for useful requests.

Third, even though prefetching increases bus utilization in the 4 and 8-core systems, the performance improvement is not very significant. In fact, the 8-core system suffers performance degradation (by 1.3%) compared to no prefetching even though more DRAM bandwidth is consumed. This is because the increased memory request contention due to the increased number of cores is not managed efficiently by conventional memory systems since they do not take into account the DRAM system's characteristics and applications' behavior together. For example, contention between memory requests from applications A and B running together can cause application B to close a row buffer that was opened by application A. This results in longer DRAM latency for application A's later memory access to the closed row since the closed row must be reopened. Also, memory requests to different banks from application A that could potentially be serviced in parallel in

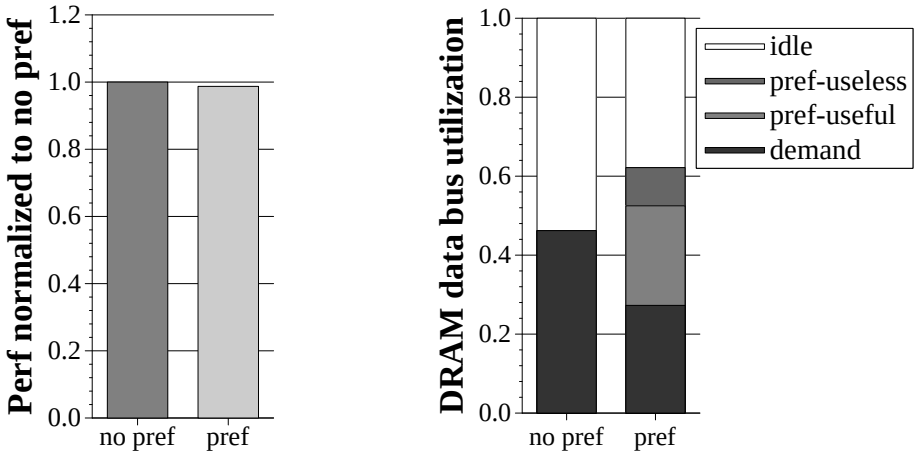
[2] Chapter 9 explains the system configurations and the workloads in detail.




(a) Single-core processor




(b) 4-core chip multiprocessor




(c) 8-core chip multiprocessor

Figure 1.1: Performance and DRAM bus utilization for a conventional memory system with no prefetching and a stream prefetcher (with the peak DRAM bandwidth of 12.8, 25.6, 25.6GB/s for single, 4, and 8-core systems respectively)

multiple DRAM banks may end up being serviced serially due to interference from application B's memory requests. Both examples can result in poor performance even though data bus utilization has increased compared to no prefetching.

Finally, a significant number of cycles are spent for useless prefetch requests (brought from DRAM but not used). Useless prefetches consume DRAM bandwidth without contributing to performance. Even worse, they sometimes even hurt performance as shown in Figure 1.1(c). Removing useless prefetches results in more efficient data bus utilization, allowing useful requests to be serviced faster.

We would like to develop a comprehensive on-chip memory system design that can efficiently exploit or reduce the idle DRAM data bus cycles for high performance. To this end, this dissertation proposes new low-cost on-chip memory system (i.e., prefetcher, buffer, and cache structures) designs that take into account DRAM characteristics. The proposed mechanisms significantly improve system performance by reducing DRAM access latency and increasing DRAM access parallelism for useful memory requests for both single-core and chip multiprocessor systems.

## 1.2 Thesis Statement

Performance of microprocessors can be improved significantly by taking into account the main memory system's characteristics in their on-chip memory system designs.

## 1.3 Contributions

This dissertation makes the following contributions.

- This dissertation introduces the notion of main memory (DRAM)-aware design of a microprocessor's on-chip memory system. It identifies three major DRAM characteristics in state-of-the-art DRAM systems which significantly affect performance: row buffer locality, bank-level parallelism, and

write-caused interference. It shows that conventional on-chip memory system designs that do not take into account these characteristics result in underutilization of the DRAM system, thereby limiting overall system performance. To overcome this problem, this dissertation proposes and evaluates DRAM characteristic-aware prefetch scheduling/issuing and cache management techniques.

- This dissertation identifies problems of the conventional DRAM controller design in the presence of prefetching. It presents a prefetch-aware DRAM controller design that aims to maximize row buffer locality only for demand and useful prefetch memory requests and to minimize the negative effect of useless prefetch requests. The proposed technique significantly improves performance by reducing the latency of useful requests and removing useless prefetches.

- This dissertation shows that conventional request issue policies to resource-limited on-chip buffers can limit the amount of Bank-Level Parallelism (BLP) realized by the DRAM controller. This reduces the effectiveness of prefetching and out-of-order execution. This dissertation presents and analyzes on-chip request issue policies that aim to maximize DRAM BLP. The evaluations show that the proposed BLP-aware policies significantly increase BLP and therefore improve system performance.

- This dissertation demonstrates that due to the DRAM characteristics, not all misses and evictions of the last-level cache incur the same cost. It proposes a DRAM-aware last-level cache replacement policy that favors the replacement of low-cost cache lines that will likely take advantage of row buffer locality and BLP and lines that can reduce write-caused interference. The evaluations show that the DRAM-aware replacement policy can improve performance by exploiting all DRAM characteristics.

- This dissertation identifies limitations of our DRAM-aware replacement policy that aims to reduce write-caused interference in the DRAM system. It

proposes a more aggressive writeback technique for the last-level cache to further reduce write-caused interference. The proposed writeback mechanism proactively sends writebacks from dirty lines that can be serviced fast due to row buffer locality. The results presented in this dissertation show that this mechanism allows the DRAM controller to service more writes quickly, thereby resulting in less write-caused interference than our DRAM-aware replacement policy.

- This dissertation evaluates the performance and DRAM efficiency of all the proposed DRAM-aware techniques when employed together. The results show that the techniques work synergistically and increase DRAM utilization significantly. The proposed mechanisms significantly improve performance on both single-core and chip multiprocessor systems.

## 1.4 Dissertation Organization

This dissertation is organized into ten chapters. Chapter 2 provides background information on the three DRAM performance-related characteristics based on the industry standards. Chapter 3 provides an overview of four proposed mechanisms that aim to improve DRAM performance. Chapter 4 discusses related work. In the following four chapters, we propose and evaluate four mechanisms. Chapter 5 presents and analyzes a prefetch-aware DRAM controller that tries to maximize row buffer locality for demand and useful prefetches and minimize the negative effect of useless prefetches. Chapter 6 proposes and discusses two Bank-Level Parallelism (BLP)-aware memory request issue policies in order to improve BLP. Chapter 7 presents and evaluates a DRAM-aware last-level cache replacement policy that aims to improve all three DRAM characteristics. Chapter 8 proposes and analyzes a DRAM-aware last-level cache writeback mechanism that can significantly reduce write-caused interference. Chapter 9 evaluates and discusses performance and DRAM efficiency when all four proposed DRAM-aware mechanisms are employed together on both single-core and multi-core systems. Chapter 10 con-

cludes this dissertation.

# Chapter 2

# Background: DRAM Performance-Related Characteristics

In this chapter, we provide background on three DRAM characteristics based on the Double Data Rate 3 (DDR3) SDRAM Joint Electron Device Engineering Council (JEDEC) standard. We follow the abbreviations of the standard. We refer readers to the DDR standard documentations and product datasheets [22, 49] for further detailed information. We accurately model all these performance-related timing constraints in our DRAM simulation model for our experimental evaluations of the proposed mechanisms.

## 2.1 Row Buffer Locality

Each DRAM bank is arranged in rows and columns of DRAM cells. The size of a row is several Kbytes (1 or 2 Kbytes in each bank per DRAM chip) in modern DRAM systems. To perform a complete access to a data element, three steps are required for the DRAM controller. First, a *precharge* command is sent to precharge the bank's bitlines. Second, an *activate* command is sent to open the source/destination row through the sense amplifier (which we call row buffer throughout this dissertation) in the bank. Finally, a *read* or *write* command is scheduled to access the appropriate columns from the row data in the row buffer. Every access can be performed only by reading from or writing to the row buffer. Therefore, if a subsequent access to the bank is mapped to a different row, these three steps (i.e., precharge, activate, and read/write) must be performed again. We call an access to a different row a *row conflict*. On the other hand, a subsequent access which is mapped to the same row as the previous row can be performed simply by

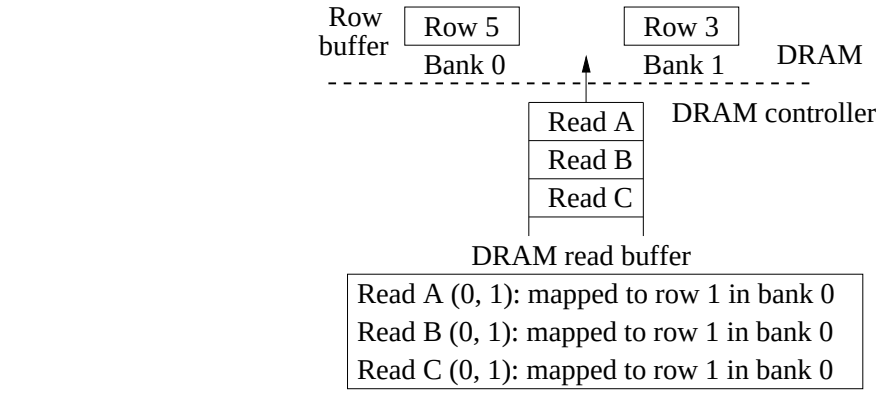
accessing the appropriate column from the currently open row. We call this access a *row hit*. Since a row hit requires only the third of the three steps, its DRAM service time is much less than that of a row conflict.

Figure 2.1 illustrates exactly how the DRAM system works for these accesses. Figure 2.1(a) shows that three reads (A, B, and C) are waiting in the DRAM read buffer for DRAM scheduling. Figure 2.1(b) shows the resulting DRAM timing when these reads are serviced. The DDR3 DRAM's *prefetch buffer* enables a burst mode of up to eight (burst length, $BL$ = 8) by bringing (eight) consecutive columns from the row buffer to the prefetch buffer. [1] Each command (e.g., read, write, or precharge) takes a DRAM bus cycle and every data transfer is done in burst mode at twice the rate of the clock (i.e., double data rate, 4 DRAM clock cycles for BL = 8). [2]

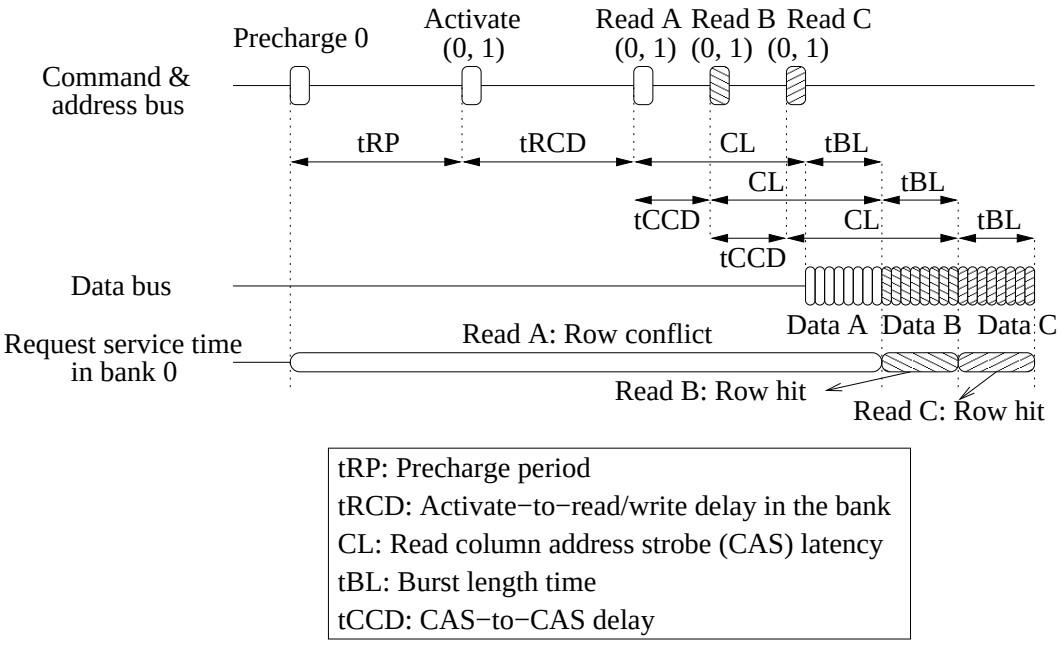
In this example, all reads are mapped to the same row (Row 1) in Bank 0. Currently Row 5 is open in the row buffer of bank 0. Read A has to go through all three steps since it is a row conflict. The total service time for Read A is the sum of the latencies for the three steps (precharge period + Activate-to-read/write delay + column address strobe latency, $t_{RP} + t_{RCD} + CL$) as shown in Figure 2.1(b). After this latency, the data required by Read A is put onto the data bus. Since the burst length is eight, eight bursts of data are sent to the data bus. The subsequent two reads can simply access the row opened by Read A. Even though accessing a given column within a row takes only column address strobe latency ($CL$), consecutive row-hit reads are serviced even faster. This is because the DDR3 system allows row-hit latencies ($CL$s) to overlap in order to support back-to-back data transfers among row-hit reads (even among row-hit reads in different banks). Therefore the effective latency of a row hit can be simply data burst latency from the processor's

[1] This is called the *8n-bit prefetch architecture* in the DDR3 technology, where n is the number of data pins in a DDR3 DRAM chip. The DRAM prefetch buffer is shared by all sense amplifiers (i.e., row buffers, each of which is in a bank).

[2] Throughout this dissertation, we assume that the DRAM system has a DRAM Dual Inline Memory Module (DIMM) with a 64-bit wide data bus per DRAM channel. Therefore, the data transfer for a 64-byte cache line can be completed with a burst length of eight.




**(a) DRAM state**




**(b) DRAM timing**

Figure 2.1: Row conflict and row hit in modern DRAM system

point of view. This makes a row-hit request much faster than a row-conflict request (up to 9 times faster in a DDR3-1600 chip [49]). Note that such back-to-back data transfers are supported among row-hit writes as well by overlapping write column address strobe latencies ($CWL$s).

Since row hits can be serviced (effectively $3 \sim 9$ times) faster than row conflicts, many DRAM controllers prioritize row hits over row conflicts in their scheduling decisions [92, 66, 48].

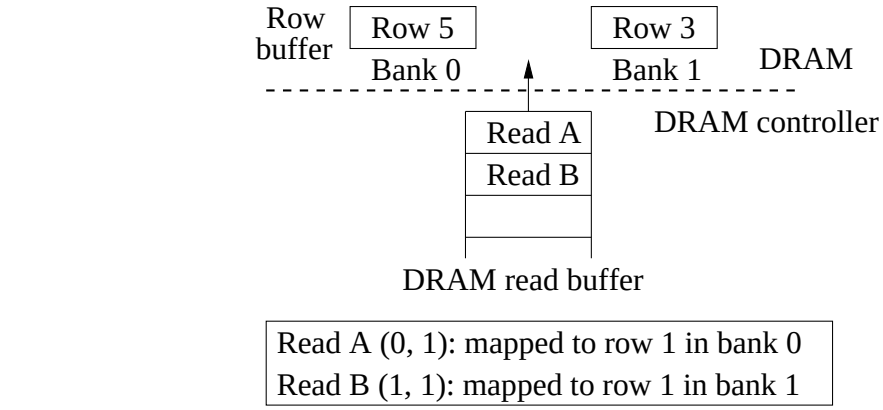
## 2.2 Bank-Level Parallelism

A DRAM chip consists of multiple ($4 \sim 8$) independent banks and accesses to different banks can be serviced concurrently. Figure 2.2 shows the DRAM behavior of two row conflict accesses to different banks. Read A is mapped to Row 1 in Bank 0 and Read B is mapped to Row 1 in Bank 1 as shown in Figure 2.2(a). Even though they are row conflicts (i.e., the current open rows are different from the rows they access), their DRAM service times can be significantly overlapped as shown in Figure 2.2(b). Therefore the effective stall time of the processor for these two requests is much less than the sum of the two access latencies. Note that if two row conflicts are mapped to different rows in the same bank, they are serviced completely serially and the processor experiences the sum of two row-conflict accesses.[3]

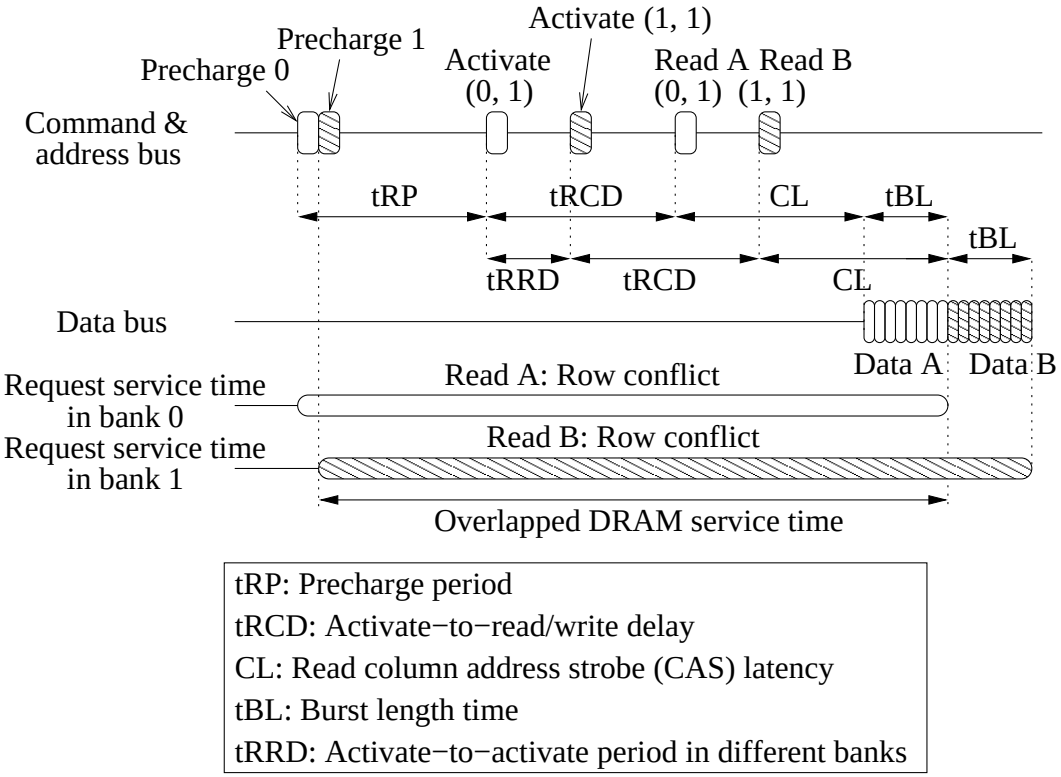
## 2.3 Write-Caused Interference

Write-caused interference in DRAM comes from read-to-write, write-to-read, and write-to-precharge latency penalties. Read-to-write and write-to-read latencies dictate the minimum latencies between a read command and a write command regardless of what DRAM banks they belong to. In contrast, write-to-precharge

[3]To be precise, the total service time of two consecutive row conflicts in the same bank is more than the sum of two row conflict latencies due to other DRAM timing constraints such as the activate-to-activate command period ($t_{RC}$) and the activate-to-precharge command period ($t_{RAS}$).




**(a) DRAM state**




**(b) DRAM timing**

Figure 2.2: DRAM bank-level parallelism

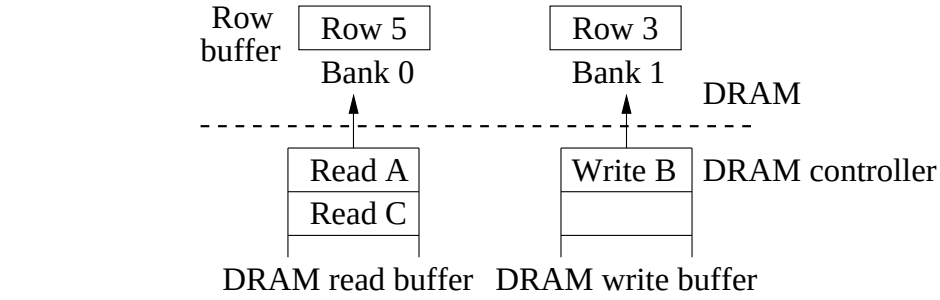
specifies the minimum latency between a write burst and a subsequent precharge command to the same bank. We first describe read-to-write and write-to-read latencies.

### 2.3.1 Read-to-write and write-to-read latencies

Read-to-write latency is the minimum latency from a read data burst to a write data burst. This latency is required to change the data bus I/O pins' state from read state to write state. Therefore, during this latency the bus has to be idle. This latency must be satisfied regardless of whether the read and the write access the same bank or different banks. In DDR3 DRAM systems, read-to-write latency is **two DRAM clock cycles**.

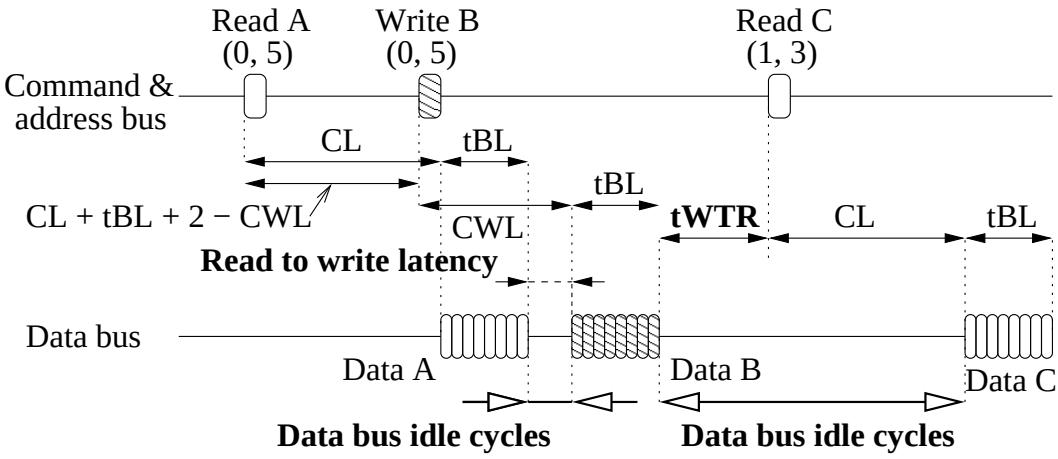
Write-to-read ($t_{WTR}$) latency is the minimum latency from a write burst to a subsequent read command. In addition to the time required for the I/O state change from write to read, this latency also includes the time required to guarantee that modified data (in the DRAM's prefetch buffer) can be safely written to the row buffer (i.e., sense amplifier). A common internal bidirectional bus connects the prefetch buffer and the row buffers of all DRAM banks. All read and write transfers use this bidirectional bus. Therefore, a subsequent read cannot use the common internal bus to bring data into the prefetch buffer until the current write's modified data is completely written back to the corresponding bank's row buffer. Therefore $t_{WTR}$ is much larger (e.g., **six DRAM clock cycles** for DDR3-1600) than read-to-write latency and introduces more idle DRAM data bus cycles. Also, write-to-read latency must be satisfied regardless of whether the write and the read are to the same bank or different banks.

We demonstrate these penalties using an example in Figure 2.3. Figure 2.3(a) shows the initial state of the DRAM read/write buffer and the row buffer state of two banks. Two reads (A and C) and one write (B) are in the read and write buffer respectively. Read A and Write B are mapped to the currently open row in Bank 0 whereas Read C is mapped to the currently open row in Bank 1. Hence they are




Read A (0, 5): mapped to row 5 in bank 0
Write B (0, 5): mapped to row 5 in bank 0
Read C (1, 3): mapped to row 3 in bank 1
**Scheduling order: Read A, Write B, Read C**

**(a) DRAM state**




CL: Read column address strobe (CAS) latency
CWL: Write column address strobe (CAS) latency
tBL: Burst length time
tWTR: Write−to−read latency

**(b) DRAM timing**

Figure 2.3: Read-to-write and write-to-read latencies

all row hits. Let us assume that the underlying DRAM controller schedules these requests in the order of Read A, Write B, and Read C. Figure 2.3(b) shows the resulting DRAM timing diagram.

The command for Write B after Read A must be scheduled such that the read-to-write latency between the corresponding data bursts is satisfied. In order for the write burst to be on the bus two DRAM cycles after the read burst, the command for Write B has to be scheduled by the DRAM controller at least $CL + t_{BL} + 2 - CWL$ DRAM clock cycles after the read command is scheduled [22]. [4] Also, Read C after Write B satisfies $t_{WTR}$ (i.e., write-to-read latency). Read C can only be scheduled $t_{WTR}$ cycles after the data burst for Write B is completed. In contrast to read-to-write latency, the data bus must be idle for $t_{WTR} + CL$ cycles since the subsequent read command cannot be scheduled for $t_{WTR}$ cycles.

Due to read-to-write and write-to-read penalties, switching service between reads and writes frequently in the DRAM system results in many idle cycles. This problem can be mitigated by a good write buffer policy as we will discuss in Chapter 8. However a write buffer policy cannot solve the problem completely due to write-to-precharge (or write recovery time, $t_{WR}$) penalties as we show below.

### 2.3.2 Write-to-precharge latency

Write-to-precharge latency (write recovery time, $tWR$) comes into play when a subsequent precharge command is scheduled to open a different row after a write to a bank. Write-to-precharge latency specifies the minimum latency from a write data burst to a precharge command in the same DRAM bank. This latency is very large (**12 DRAM clock cycles** for DDR3-1600) because the written data in the DRAM's prefetch buffer must be written back to the corresponding DRAM row through the row buffer before precharging the DRAM bank. This needs to be done

[4] We assume that the additive latency (AL) is zero in this dissertation. If a non-zero AL is considered, the subsequent write command can be scheduled $CL+AL+t_{CCD}+2-(CWL+AL)$ cycles after the read command, where $t_{CCD}$ is the minimum column strobe to column strobe latency). To maximize bandwidth we set up $t_{BL}$ to eight, therefore $t_{CCD}$ is equal to ($tBL$) [22].

to avoid the loss of modified data.

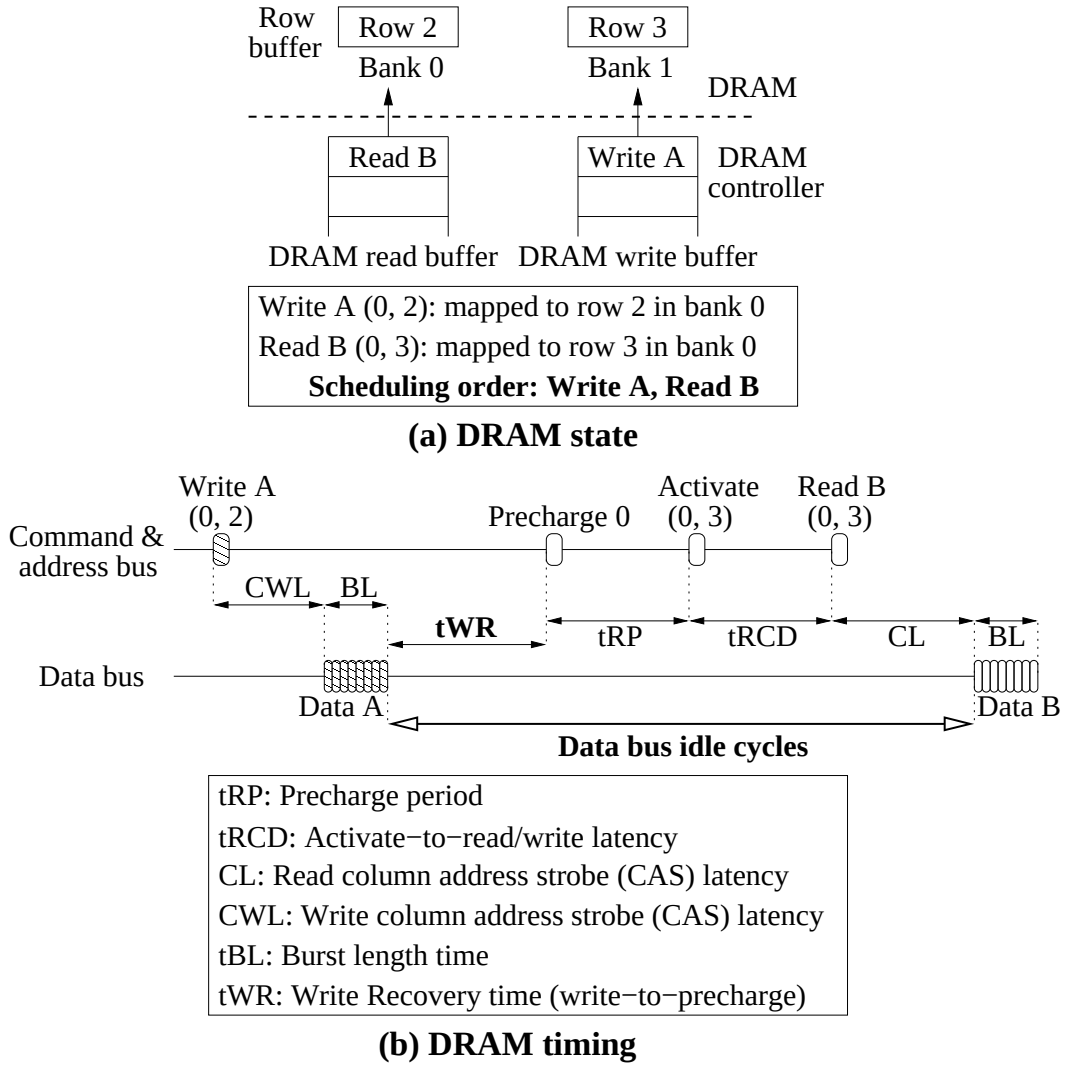



Figure 2.4: Write-to-precharge (write recovery time) latency

Figure 2.4 illustrates write-to-precharge penalty in a DRAM bank. Write A and Read B access different rows in the same bank (Bank 0). Therefore, after Write A is serviced, a precharge command is required to open the row for Read B (i.e., row conflict). Subsequent to the scheduling of Write A, the precharge command must wait until write-to-precharge latency is satisfied before it can be scheduled. Note that this penalty must be satisfied regardless of whether the subsequent precharge command is for a read or a write. The resulting data bus idle cycles is $t_{WR} + t_{RP} +$

$t_{RCD} + CL$ DRAM clock cycles unless there are other requests that are being read or written in different banks.

Since the write-to-precharge latency must be satisfied even for a precharge for a subsequent write, row conflicts among writes degrade DRAM throughput for writes. For example, a write to Row 1 after a write to Row 3 in the same bank must still satisfy this write-to-precharge penalty before the precharge command for the write to Row 3 can be scheduled. This problem cannot be solved by the DRAM write buffer and its policy. If writes in the write buffer access different rows (row-conflict writes) in the same bank, the total amount of write-to-precharge penalty becomes very large. This degrades DRAM throughput for writes and eventually results in delaying the service of reads, thereby degrading application performance.

# Chapter 3

# Overview of the Solutions

The dissertation makes a case for DRAM-aware on-chip memory system design. We propose DRAM characteristic-aware prefetching and cache management mechanisms that aim to maximize DRAM row buffer locality and bank-level parallelism and to minimize write-caused interference. We propose four different mechanisms, each of which works on a different on-chip memory resource structure to improve DRAM performance. Figure 3.1 illustrates where our mechanisms (shown in highlighted areas) would be employed in a conventional microprocessor. We briefly overview each of these mechanisms as follows.

The first mechanism is a prefetch-aware DRAM controller that tries to minimize DRAM access latencies for useful memory requests (demand and accurate prefetches) by exploiting row buffer locality when prefetching. We make the DRAM controller(s) prefetch-aware and take advantage of low latencies for row-hit prefetches when the prefetches are estimated as useful. To minimize the negative effect of useless prefetches, the DRAM controller delays and drops prefetches predicted to be useless. Chapter 5 analyzes this mechanism.

The second mechanism is DRAM bank-level parallelism-aware memory request issue policies in on-chip buffer structures that aim to maximize BLP in the presence of prefetching. They determine the order in which requests are sent from one on-chip buffer to another buffer so that requests to different banks are eventually exposed together to the DRAM controller. We discuss a BLP-aware prefetch issue policy from the prefetch request buffer to the Miss Status/Information Holding Registers (MSHRs) in order to maximize the BLP of requests (demands and useful prefetches) exposed to the DRAM controller. We also propose a BLP-preserving

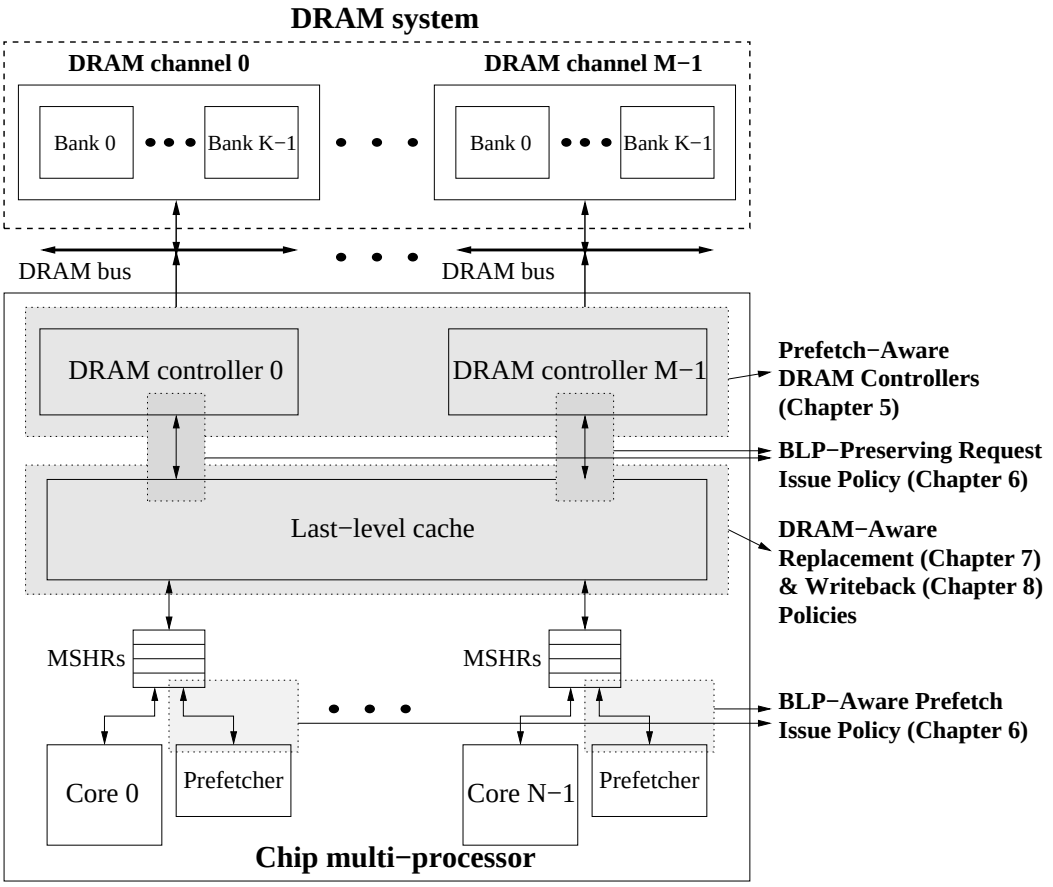



Figure 3.1: Overview of proposed DRAM-aware mechanisms

memory request issue policy from the last-level cache to the DRAM controller's buffers (DRAM request buffer). This policy tries to make sure that requests from each core can be serviced together by the DRAM controller without destroying the BLP of each core in CMP systems. Chapter 6 discusses and evaluates these BLP-aware issue policies.

The third mechanism is a DRAM-aware last-level cache replacement policy that tries to improve all three DRAM characteristics. It leverages the fact that a last-level cache replacement policy can fundamentally change the mixture/property of outstanding memory requests, which can affect DRAM performance due to the DRAM characteristics. The DRAM-aware replacement policy favors the eviction of cache lines that would be refetched in quickly due to row buffer locality or serviced

together with other misses in different DRAM banks, when they are refetched. It also evicts dirty lines that can be written back to DRAM quickly by exploiting row buffer locality, in order to reduce write-caused interference in the DRAM system. Since row-hit writes are serviced quickly (back-to-back), the DRAM controller can resume servicing reads sooner, which in turn improves DRAM performance. We discuss and analyze the DRAM-aware replacement policy in Chapter 7.

The fourth mechanism is a DRAM-aware last-level cache writeback policy that aims to further reduce write-caused interference in the DRAM system. In contrast to the DRAM-aware replacement, it proactively sends writebacks that are expected to hit in the DRAM row buffers even before a replacement happens. This significantly reduces write-caused interference because it allows more writes to be written back faster than the DRAM-aware replacement would. Chapter 8 studies and analyzes this DRAM-aware writeback policy.

Note that each of the four mechanisms manages a different on-chip memory system management policy to improve DRAM utilization. Therefore the four mechanisms are complementary. We evaluate and analyze the combination of all four mechanisms in Chapter 9.

# Chapter 4

# Related Work

How memory requests are managed in the on-chip memory system (DRAM controller, buffers, caches, and prefetching) of a processor significantly affects main memory (DRAM) performance. This chapter discusses studies that are relevant to on-chip memory system designs with respect to DRAM system, memory-level parallelism, prefetching, and last-level cache management.

## 4.1 Research in DRAM System Management

### 4.1.1 DRAM Access Scheduling

A number of DRAM scheduling policies have been proposed. McKee et al. proposed DRAM scheduling policies that exploit row buffer locality and bank-level parallelism for streaming applications in a page mode DRAM system [46, 45, 47]. Zuravleff and Robinson patented a DRAM scheduling policy similar to McKee et al.'s [92]. Carter et al. proposed an off-chip memory controller that aims to reduces wasteful memory bandwidth consumption by remapping physical addresses [3]. Their mechanism also prefetches data from DRAM into an SRAM buffer in the memory controller to hide DRAM access latency. Rixner et al. proposed and evaluated DRAM scheduling policies in a stream processor [66]. Zhang and McKee evaluated a stride (stream) prefetcher combined with a scheduling policy that reorders memory requests such that multiple requests can be serviced together in different banks in a Rambus DRAM system [86, 48]. Since then, many other scheduling policies have been proposed in single-threaded [18, 68] and multithreaded [65, 89, 57, 85] systems. In addition, several recent studies [58, 53, 54, 20, 30, 31] proposed techniques for fairness (quality of service) and/or high performance across different

applications sharing the DRAM system. These prior proposals have the following limitations.

First, some DRAM scheduling policies [46, 45, 47, 92, 66, 57, 18, 89, 58, 68, 53, 54, 20, 85, 30, 31] do not consider hardware prefetching. Hardware prefetching is an important memory latency-tolerance technique already employed in most commercial processors [77, 17, 80, 68, 34]. It is very important to intelligently manage demand and prefetch requests to the DRAM system, since the performance with a prefetcher can significantly differ depending on how the DRAM controller handles prefetch requests compared to demand requests. In contrast to these prior prefetch-unaware scheduling policies, the prefetch-aware DRAM controller proposed in Chapter 5 adaptively prioritizes demand and prefetch requests based on prefetch usefulness to maximize row buffer locality for useful requests and minimize the negative effect of useless prefetches. The concept of adaptive prefetch handling can be applied to the existing prefetch-unaware DRAM access scheduling policies.

Second, the DRAM controller proposals that do consider hardware prefetching take two different approaches to handling prefetch requests. Some proposals [42, 18, 19, 73] always prioritize demand requests over prefetch requests. Other proposals [86, 48, 65, 3] and some commercial processors [76, 28] treat prefetch requests the same as demand requests. Neither of these approaches works best for all types of applications. This is because they do not take into account both the DRAM characteristics and prefetch usefulness for their scheduling decisions. The prefetch-aware controller outperforms these two rigid prefetch handling policies in DRAM scheduling, as we show in Chapter 5.

Third, the performance of the DRAM scheduling policies is limited by the number and composition of requests in the DRAM controller's buffers, i.e., the DRAM request buffers. If requests in the DRAM request buffers are not mapped to different DRAM banks, bank-level parallelism will be low regardless of the DRAM scheduling policy. Similarly, if multiple requests that are mapped to the same row

are not present in the DRAM request buffer, high row buffer locality cannot be exploited by a DRAM scheduling policy. The BLP-aware request issue (in Chapter 6), DRAM-aware last-level cache replacement (in Chapter 7), and DRAM-aware last-level cache writeback (in Chapter 8) mechanisms send out requests that can expose more BLP and row buffer locality in the DRAM request buffers. This allows the underlying DRAM scheduling policy to exploit higher BLP and row buffer locality.

### 4.1.2 DRAM Write Buffer Management

Some previous proposals [40, 57, 68] discuss DRAM write buffer management polices to reduce write-caused interference in the main memory system. Writes in the write buffer are not considered for scheduling until the underlying write buffer policy decides to do so. Lee et al. [40] employed a write buffer management policy that allows the Rambus DRAM controller to schedule a write when the data bus is idle. Natarajan et al. [57] discussed different write buffer management policies that also opportunistically allow writes to be scheduled in a DDR2 DRAM system when there are no pending reads or when the write buffer is almost full (i.e., the number of writes is more than a threshold). Their policies also make sure that a certain number of writes are serviced, even when a new read comes while servicing the writes. Shao and Davis [68] proposed a DRAM scheduling policy in a DDR2 system which services writes when there are no reads in the DRAM read buffer, when the write buffer is full, or when a write hits the currently open row. If a new read comes into the DRAM request buffer, their mechanism allows the read to preempt writes that are being serviced.

Even though there are small differences among these write buffer policies, they are essentially based on the principle that scheduling writes when the bus is idle (no pending reads) can reduce the contention between reads and writes. However, we show in Chapter 8 that this principle is not the best with today's high-bandwidth DDR (DDR3) DRAM systems because of their large write-caused latency penalties. We show that the policy which services all writes present in the write buffer only when the write buffer becomes full (which we call the *drain_when_full* policy)

outperforms prior policies. This is because it 1) reduces the frequency of read-to-write/write-to-read switching, and 2) allows the DRAM controller to better exploit row buffer locality and bank-level parallelism exposed by more writes. We use this drain_when_full policy in our baseline memory system. Also, the aggressive DRAM-aware writeback policy in Chapter 8 further reduces write-caused interference by leveraging the benefits of this baseline drain_when_full policy.

## 4.2 Research in Improving Memory-Level Parallelism

Many memory latency-tolerant techniques exploit Memory-Level Parallelism (MLP) by increasing the number of outstanding memory requests in the on-chip memory system [15]. Out-of-order execution [78] and non-blocking caches [33] allow generating concurrent memory requests. Prefetching techniques [32, 50, 14, 27, 1, 26] also increase MLP by issuing concurrent memory requests that are predicted to be used by the program.

Pai and Adve proposed a compiler optimization that generates concurrent memory requests by reordering memory instructions [60]. Runahead execution [9, 55, 56] issues requests by executing future instructions that are independent of a long latency load instruction during the stall time of that load instruction. Zhou and Conte proposed a prefetching technique with the help of value prediction to generate data dependent misses earlier [88]. Chou et al. analyzed the impact of various microarchitecture parameters and structures on MLP [6]. Qureshi et al. proposed a cache replacement policy that favors eviction of cache lines that could be serviced together with other misses when they are refetched later [63]. Eyerman and Eeckhout proposed fetch policies for simultaneous multithreading that prefer to fetch threads that generate many concurrent misses [12].

All of these studies define MLP as the average number of outstanding memory requests when there is at least one outstanding request to memory. They implicitly assume that the DRAM latency of outstanding requests to memory will overlap. However, simply having a large number of outstanding requests does not necessar-

ily mean that their DRAM latencies will overlap. Multiple outstanding last-level cache misses that are all mapped to the same DRAM bank are serviced serially in the DRAM system. Therefore it is very important to send out multiple requests that are mapped to different DRAM banks to maximize the benefits of the MLP enhancement techniques. This is especially important since the total number of outstanding requests allowed in an on-chip memory system is limited. The Miss Status/Information Holding Registers (MSHRs) that keep track of all outstanding requests are costly to increase in size [79]. Simply filling up resource-limited MSHRs with many requests that are mapped to only a few DRAM banks can result in low BLP. In order to exploit true Memory-Level Parallelism (MLP), an on-chip memory resource management policy (e.g., buffer and cache policies) should be main memory (DRAM)-aware so that memory requests to different memory banks can be sent to the DRAM system at the same time. The BLP-aware request issue policy proposed in Chapter 6 and the DRAM-aware replacement policy presented in Chapter 7 aim to achieve this goal.

## 4.3 Research in Prefetching and Prefetch Handling

### 4.3.1 Prefetching Algorithms

Prefetching predicts memory access patterns and brings data into a cache or buffer before the data is needed by the processor. This technique also improves MLP by increasing the number of memory requests in the on-chip memory system. Software prefetching [2, 32, 50] tries to prefetch data by inserting prefetch instructions in the program. This technique is effective for regular memory access patterns. However it requires compiler support and modification of existing binaries.

Various hardware prefetching techniques have been proposed to capture runtime memory access pattern without requiring compiler support or modification of binaries: e.g., next-line prefetching [14], stream prefetching [27], stride prefetching [1], and correlation prefetching [5, 26]. A hardware prefetcher can generate many useless prefetches depending on the running application and exe-

cution phases. Useless prefetches can hurt performance, since they also consume memory system resources (DRAM bandwidth, buffers, and caches) and contend with demand requests. Also, depending on how the on-chip memory system (e.g., buffers and DRAM controller) handles prefetches with respect to demands, system performance with a prefetcher becomes dramatically different. Our mechanisms discussed in Chapters 5 and 6 aim to maximize the benefits of useful prefetches and minimize the negative effect of useless prefetches by taking DRAM characteristics into account in prefetch handling.

### 4.3.2 Useless Prefetch Filtering

To reduce useless prefetches, several prefetch filtering mechanisms were proposed [50, 4, 74, 90, 51].

Charney and Puzak proposed a useless prefetch filtering scheme for an L2 to L1 next sequential cache line prefetcher [4]. Using a confirmation bit per L2 cache line, their scheme does not service prefetch requests that have been proven to be useless in the past. Although this may work for an L2 to L1 prefetcher, this mechanism has high hardware cost for prefetching from memory to the last-level cache since every cache block in the entire physical memory needs to be tagged.

Mutlu et al. use the L1 cache as a prefetch filter for L2 cache pollution [51]. In their scheme, if a line that was prefetched into the L1 was never used, it would not be inserted into the L2 cache when it is evicted from the L1. Both of the above proposals unnecessarily consume memory bandwidth since useless prefetches are filtered out only after they are serviced by the DRAM system. In contrast, the prefetch-aware DRAM controller and BLP-aware prefetch issue policy in this dissertation remove useless prefetches before they consume valuable DRAM bandwidth.

Mowry et. al. proposed a prefetch dropping mechanism that cancels software prefetches when the prefetch issue queue is full to avoid processor stalls [50]. As opposed to dealing with software prefetches, which usually have high accuracy,

the prefetch-aware DRAM controller (Chapter 5) and BLP-aware prefetch issue policy (Chapter 6) deal with hardware prefetch requests based on runtime prefetcher accuracy. The former cancels useless hardware prefetches at the DRAM controller, and the latter limits the issue of useless prefetches to the on-chip memory system.

Zhuang and Lee proposed a mechanism that eliminates the prefetch request for an address if the prefetch request for the same address was useless in the past [90]. We show in Chapter 5 that this technique removes many useful prefetches as well as useless prefetches.

### 4.3.3 Adaptive Prefetching

In addition to the prefetch filtering mechanisms, adaptive prefetch management techniques [19, 73, 11] have been proposed to increase the benefits and also reduce the harm of prefetching. They adjust the aggressiveness of prefetching based on the contention in the memory system and/or prefetch usefulness information

Hur and Lin designed a probabilistic prefetching technique which adjusts prefetcher aggressiveness [19]. They schedule prefetch requests to DRAM adaptively based on the frequency of DRAM bank conflicts caused by prefetch requests. However, their scheme always prioritizes demand requests over prefetches.

Srinath et al. show how adjusting the aggressiveness of the prefetcher based on accuracy, lateness, and cache pollution information can reduce bus traffic without compromising the benefit of prefetching [73].

Ebrahimi et al. discuss how to manage multiple different prefetchers each of which can prefetch different access patterns [11]. Their mechanism adjusts the aggressiveness of each prefetcher depending on its accuracy and timeliness.

These techniques have limitations. First, none of them consider DRAM characteristics in order to achieve better performance benefits from prefetching. For example, as we show in this dissertation, 1) useful row-hit prefetches can be serviced significantly faster and 2) prioritizing the issue of prefetches that are mapped to different DRAM banks can improve DRAM BLP. Second, none discuss how to

manage demand and prefetch requests in a CMP's on-chip shared memory system for high system performance when multiple applications run on different cores.

More recently, Ebrahimi et al. proposed an adaptive prefetching technique to maximize system performance in CMP systems [10]. Their mechanism controls the aggressiveness of the prefetcher on each core based on the prefetch accuracy of each core and inter-core interference caused by each core's prefetcher in the memory system. Even though this mechanism's decision is based on the contention between requests from multiple cores in the DRAM system, it does not explicitly target improving row buffer locality or bank-level parallelism of memory requests. Therefore, the DRAM-aware mechanisms proposed in this dissertation are orthogonal to this proposal. In fact, Ebrahimi et al. show that the prefetch-aware DRAM controller in Chapter 5 is orthogonal to their mechanism [10].

## 4.4 Research in Cache Management

### 4.4.1 Cache Management for Locality

Caches [81] tolerate long memory latency by using small and fast on-chip storage. Kroft improved cache performance by allowing multiple outstanding cache misses using Miss Status/Information Holding Registers (MSHRs) in the memory system [33]. Also, many cache replacement/insertion policies have been proposed to improve temporal locality in on-chip caches (e.g., [16, 62, 21]).

These cache techniques have limitations. First, the working set size of some applications is too large to fit even in large on-chip caches. Second, some applications expose no temporal locality (e.g., streaming applications). Third, a last-level cache miss still experiences long memory latency.

Furthermore, due to DRAM characteristics, not all last-level cache misses incur the same memory latency from the processor's point of view. Some misses are serviced quickly by exploiting row buffer locality and other misses are serviced in parallel with misses in different DRAM banks. Therefore, it is very important

for cache management techniques to take into account DRAM characteristics for better performance as we show in this dissertation.

### 4.4.2 Cost-Aware Cache Management

Jeong and Dubois were the first to propose a replacement policy for a cache that has two miss costs (local memory access and remote memory access) [23, 24].

Qureshi et. al. showed that an MLP-aware replacement policy can improve performance by taking into account the level of concurrency of misses in the on-chip memory system [63].

Neither of these policies take DRAM characteristics into account in their replacement decisions. The MLP-aware policy assumes that misses to the same bank will be serviced in parallel with other misses. Also, neither considers the cost of writebacks. Instead, they consider only the future miss cost of a line when making eviction decisions. This can increase write-caused interference in the DRAM system by causing a large number of row-conflict writebacks. In Chapter 7, we show that the MLP-aware policy does not perform as well as our DRAM characteristic-aware replacement policy.

### 4.4.3 Writeback Management

Some prior studies propose aggressive early writeback policies which proactively send writebacks of dirty cache lines before they are evicted by a replacement policy. Some of these proactive policies [40, 75] aim to reduce write-caused interference in the DRAM system. Eager writeback [40] sends a writeback for a dirty LRU (Least Recently Used) line in a cache set whenever the cache set is accessed. However, this mechanism is not aware of DRAM characteristics. We show in Chapter 8 that simply sending writebacks for dirty LRU (Least Recently Used) cache lines does not reduce write-caused interference.

Virtual write queue [75] performs early writebacks for dirty LRU lines in a DRAM-aware way similar to our DRAM-aware writeback mechanism. This mech-

anism sends writes that can be written back with other writes together in different DRAM banks as well as writes that can be written back quickly due to row buffer locality. However, virtual write queue only considers writebacks for the two least recently used positions in a cache set, which can limit the number of writes that can be written quickly. Also, the mechanism is complex and requires communication between the last-level cache and DRAM controllers. In contrast, the DRAM-aware writeback mechanism we propose in Chapter 8 generates more writes that can be written quickly since writebacks for any LRU position can be sent out. Our mechanism can be implemented at a smaller cost, requiring no communication between the cache and DRAM controllers.

Other early writeback mechanisms [41, 29, 84] periodically send early writebacks to the next-level cache or DRAM to increase the reliability of on-chip caches at low cost. Even though our motivation is not to improve reliability but to reduce write-caused interference, our writeback mechanism can help reduce vulnerability in the last-level cache since it aggressively sends writebacks just like these early writeback policies do.

# Chapter 5

# Prefetch Management for Reducing DRAM Latency

In this chapter, we show how to manage demand and prefetch requests in DRAM controllers in order to reduce DRAM latency by exploiting row buffer locality.

## 5.1 Motivation

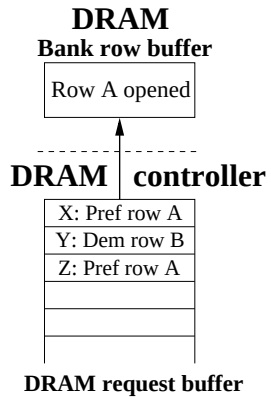
None of the existing DRAM scheduling policies take into account both the non-uniform nature of DRAM access latencies and the usefulness of prefetch requests. Existing DRAM scheduling policies take largely two different approaches as to how to treat prefetch requests with respect to demand requests. Some policies [86, 48, 65, 76, 28] give prefetch requests the same priority as demand requests. We call this policy *demand-prefetch-equal*. It is the same as the FR-FCFS (First Ready-First Come First Serve) policy [66] that prioritizes requests as follows: 1) row-hit requests over all others, 2) older requests over younger requests. This can significantly delay demand requests and cause performance degradation, especially when prefetch requests are not accurate. Other policies [26, 42, 18, 72, 73] always prioritize demand requests over prefetch requests so that data known-to-be-needed by the program can be serviced earlier. We call this policy *demand-first*. One might think that the demand-first policy provides the best performance by eliminating the interference of prefetch requests with demand requests. However, such a rigid policy does not consider the non-uniform access latency of the DRAM system (row-hits vs. row-conflicts). A row-hit prefetch request can be serviced much more quickly than a row-conflict demand request. Therefore, servicing the row-hit prefetch request first provides higher DRAM throughput and can sometimes

provide better system performance than servicing the row-conflict demand request first.

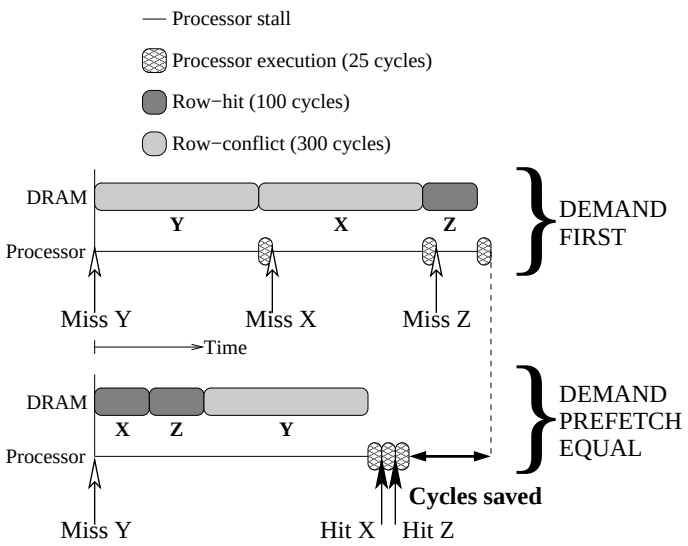
Figure 5.1 illustrates why a rigid, non-adaptive prefetch scheduling policy degrades performance. Consider the example in Figure 5.1(a), which shows three outstanding memory requests (to the same bank) in the DRAM request buffer. Row A is currently open in the row buffer of the bank. Two requests are prefetches (to addresses X and Z) that access row A while one request is a demand request (to address Y) that accesses row B.

For Figure 5.1(b), let us assume that the processor needs to load addresses in the order of Y, X, and Z (i.e., both of the prefetch requests are useful) and the computation between each load instruction takes a fixed, small number of cycles that is significantly smaller than the DRAM access latency. Figure 5.1(b) shows the service timeline of the requests in DRAM and the resulting execution timeline of the processor for two different memory scheduling policies, *demand-first* and *demand-prefetch-equal*. With demand-first (top), the row-conflict demand request is satisfied first, which causes the prefetch of address X to incur a row-conflict as well. The subsequent prefetch request to Z is a row-hit because the prefetch of X opens row A. As a result, the processor first stalls for approximately two row-conflict latencies (except for a small period of execution). The processor then stalls for an additional row-hit latency since it requires the data from address Z. The total execution time is the sum of two row-conflict latencies and one row-hit latency plus a small period of processor execution.

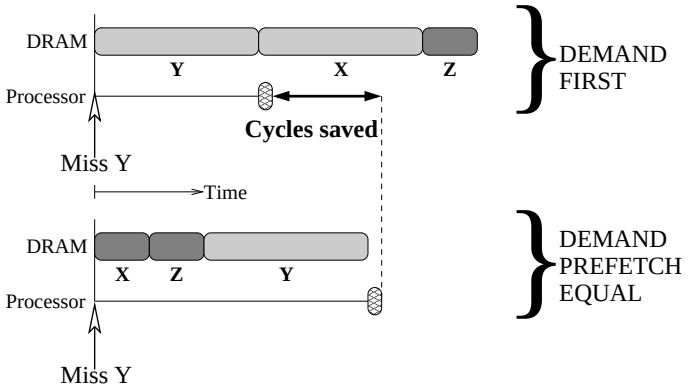
With the demand-prefetch-equal policy (bottom), the row-hit prefetch requests to X and Z are satisfied first followed by the row-conflict demand request to Y. The processor must stall until the demand request to Y is serviced. However, after that, the processor only needs to perform the computations between the load instructions because loads to X and Z hit in the cache. The total execution time is the sum of one row-conflict latency and two row-hit latencies (plus a small period of processor execution), which is less than with the demand-first policy. Hence, *treat-*




**(a) DRAM and controller state**




**(b) Timeline when prefetches (X and Z) are useful**




**(c) Timeline when prefetches (X and Z) are useless**

Figure 5.1: Example illustrating the performance impact of demand-first and demand-prefetch-equal policies

*ing prefetches and demands equally can significantly improve performance when prefetch requests are useful.*

However, prefetch requests might not always be useful. Let us assume that the processor needs to load only address Y but still generates useless prefetches to addresses X and Z in Figure 5.1(a). Figure 5.1(c) shows the resulting timeline. With demand-first, the processor stalls for only a single row-conflict latency which is required to service the demand request to Y. On the other hand, with demand-prefetch-equal, the processor stalls additional cycles since X and Z are serviced (even though they are not needed) before Y in the DRAM bank thereby delaying the useful request to Y. Hence, *treating prefetches and demands equally can significantly degrade performance when prefetch requests are useless*.

Figure 5.2 provides supporting data for our observation. This figure shows the performance impact of an aggressive stream prefetcher [77, 73] when used with the two different memory scheduling policies for 10 SPEC 2000/2006 benchmarks. The vertical axis is retired instructions per cycle (IPC) normalized to the IPC on a processor with no prefetching. The results show that *neither of the two policies provides the best performance for all applications*. For the leftmost five applications, prioritizing demands over prefetches results in better performance than treating prefetches and demands equally. In these applications, a large fraction (70% for demand-prefetch-equal, and 59% for demand-first) of the generated stream prefetch requests are useless. Therefore, it is important to prioritize demand requests over prefetches. In fact, for *art* and *milc*, servicing the demand requests with higher priority is critical to make prefetching effective. Prefetching improves the performance of these two applications by 2% and 10% respectively with the *demand-first* scheduling policy, whereas it reduces performance by 14% and 36% with the *demand-prefetch-equal* policy.

On the other hand, for the rightmost five applications, we observe the exact opposite behavior. Equally treating demand and prefetch requests provides significantly higher performance than prioritizing demands over prefetches. In particular,

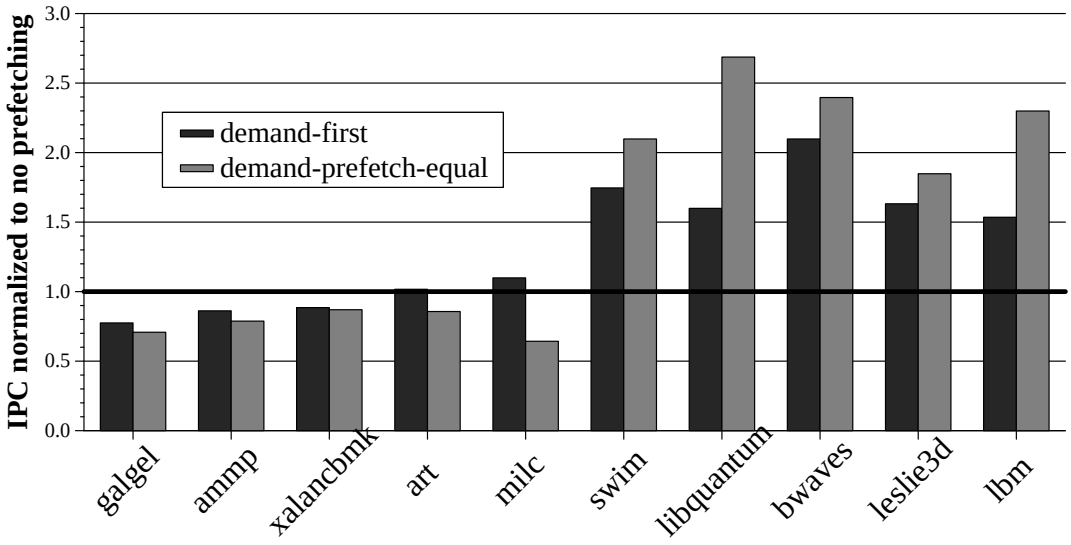



Figure 5.2: Performance of two rigid prefetch scheduling policies

for *libquantum*, the *demand-prefetch-equal* policy allows the prefetcher to provide 169% performance improvement, in contrast to the 60% performance improvement it provides with the *demand-first* scheduling policy. This is because prefetch requests in *libquantum* are very accurate (almost 100% of them are useful). Maximizing DRAM throughput by preferring row buffer hits in the DRAM system regardless of whether a memory request is a demand or a prefetch request allows for more efficient bandwidth utilization and improves the timeliness (and the coverage) of prefetches, thereby improving system performance. These results show that DRAM scheduling policies with rigid prioritization rules among prefetch and demand requests cannot provide the best performance and may even cause prefetching to degrade performance.

Note that even though the DRAM scheduling policy has a significant impact on the performance provided by prefetching, prefetching sometimes degrades performance regardless of the DRAM scheduling policy. For example, *galgel*, *ammp*, and *xalancbmk* suffer significant performance loss with prefetching because a large fraction (69%, 94%, and 91%) of the prefetches are not needed by the program. The negative performance impact of these useless prefetch requests cannot be mitigated solely by a *demand-first* scheduling policy because useless prefetches 1) occupy

memory request buffer entries in the memory controller until they are serviced, 2) occupy DRAM bandwidth while they are being serviced, and 3) cause cache pollution by evicting possibly useful data from the processor caches after they are serviced. As a result, useless prefetches could delay the servicing of demand requests and could result in additional demand requests. In essence, *useless prefetch requests can deny service to demand requests because the DRAM controller is not aware of the usefulness of prefetch requests in its DRAM request buffer*. To prevent this, the memory controller should intelligently manage the DRAM request buffer between prefetch and demand requests.

## 5.2 Mechanism: Prefetch-Aware DRAM Controller (PADC)

We propose Prefetch-Aware DRAM Controller (PADC) which adaptively controls the interference between prefetch and demand requests to improve system performance [37, 36]. PADC aims to maximize the benefits of useful prefetches and minimize the harm of useless prefetches by taking into account a DRAM characteristic: row buffer locality. PADC consists of two parts as shown in Figure 5.3: an Adaptive Prefetch Scheduling (APS) unit and an Adaptive Prefetch Dropping (APD) unit. APS adaptively schedules prefetch and demand requests to increase DRAM throughput for useful requests. APD cancels useless prefetch requests while preserving the benefits of useful prefetches. Both APS and APD are driven by the measurement of the prefetch accuracy of each processing core in a multi-core system. Therefore we first explain how prefetch accuracy is measured for each core.

### 5.2.1 Prefetch Accuracy Estimation

We measure the prefetch accuracy for an application running on a particular core over a certain time interval. The accuracy is reset once the interval has elapsed so that the mechanism can adapt to the phase behavior of prefetching. To measure the prefetch accuracy of each core, the following hardware support is required:

1. Prefetch (P) bit per last-level cache line and memory request buffer entry:

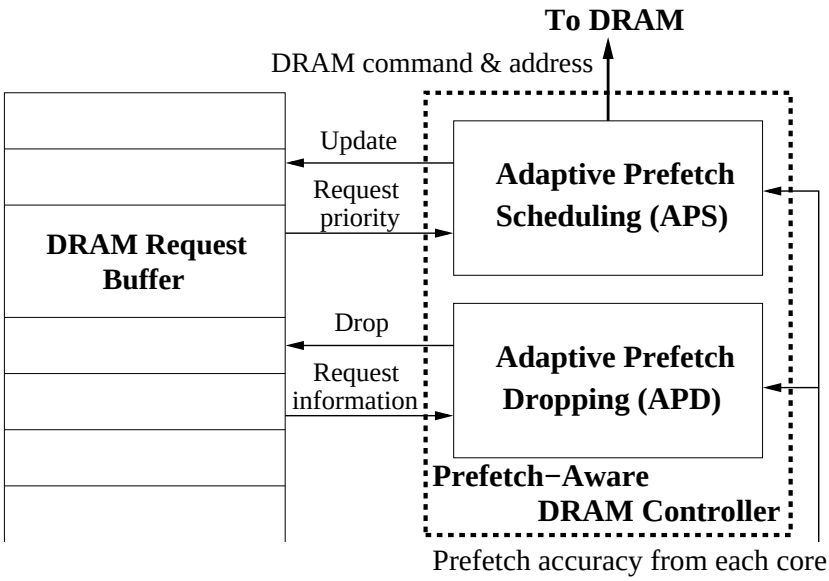



Figure 5.3: Prefetch-Aware DRAM Controller

For memory request buffer entries, this bit indicates whether or not the request was generated by the prefetcher. It is set when a new memory request is generated by the prefetcher, and reset when the processor issues a demand request to the same cache line while the prefetch request is still in the memory request buffer. For cache lines, this bit indicates whether or not a cache line was brought into the cache by a prefetch request. It is set when the line is filled (only if the prefetch bit of the request is set) and is reset when a cache hit to the same line occurs.

2. Prefetch Sent Counter (PSC) per core: This counter keeps track of the total number of prefetch requests sent by a core. It is incremented whenever a prefetch request is sent to the memory request buffer by the core.

3. Prefetch Used Counter (PUC) per core: This counter keeps track of the number of prefetches that are useful. It is incremented when a prefetched cache line is used (cache hit) by a demand request and also when a demand request matches a prefetch request already in the memory request buffer.

4. Prefetch Accuracy Register (PAR) per core: This register stores the prefetch accuracy measured every time interval. PAR is computed by dividing PUC by PSC.

At the end of every time interval, PAR is updated with the prefetch accuracy calculated during that interval and PSC and PUC are reset to 0 to calculate the accuracy for the next interval. The PAR values for each core are fed into the Prefetch-Aware DRAM Controller which then uses the values to guide its scheduling and memory request buffer management policies.

### 5.2.2 Adaptive Prefetch Scheduling

Adaptive Prefetch Scheduling (APS) determines the priority of demand/ prefetch requests from a processing core based on the prefetch accuracy estimated for that core. The basic idea is to 1) treat useful prefetch requests the same as demand requests so that useful prefetches can be serviced faster by maximizing DRAM throughput, and 2) give demand requests and useful prefetch requests a higher priority than useless prefetch requests so that useless prefetch requests do not interfere with useful requests.

If the prefetch accuracy of a core is greater than or equal to a certain threshold, $promotion\_threshold$, all of the prefetch requests from that core are treated the same as demand requests. We call such prefetch requests and all demand requests *critical* requests. Otherwise, if the prefetch accuracy of a core is less than $promotion\_threshold$, then demand requests of that core are prioritized over prefetch requests. We call such prefetch requests *non-critical* requests.

The essence of our proposal is to prioritize critical requests over non-critical ones in the memory controller, while preserving DRAM throughput. To accomplish this, our mechanism prioritizes memory requests in the order shown in Rule 1. Each prioritization decision in this set of rules is described in further detail below.

First, critical requests (useful prefetches and demand requests) are prioritized over others. This delays the scheduling of non-critical requests, most of which are likely to be useless prefetches. As a result, useless prefetches are prevented from interfering with demands and useful prefetches.

Second, row-hit requests are prioritized over others. This increases the

**Rule 1** Adaptive Prefetch Scheduling (APS)

1. **Critical request (C)**: Demand and useful prefetches are prioritized over all other requests.
2. **Row-hit request (RH)**: Row-hit requests are prioritized over row-conflict requests.
3. **Urgent request (U)**: Demand requests generated by cores with low prefetch accuracy are prioritized over other requests.
4. **Oldest request (FCFS)**: Older requests are prioritized over newer ones.

row-buffer locality for demand and useful prefetch requests and maximizes DRAM throughput as much as possible.

Third, demand requests from cores whose prefetch accuracy is less than $promotion\_threshold$ are prioritized. We call these requests *urgent* requests. Intuitively, this rule tries to boost the demand requests of a core with low prefetch accuracy over the critical requests of cores with high prefetch accuracy. We do this for two reasons. First, if a core has high prefetch accuracy, its prefetch requests will be treated the same as the demand requests of another core with low prefetch accuracy (due to the critical request prioritization rule). Doing so risks starving the demand requests of the core with low prefetch accuracy, resulting in a performance degradation since a large number of critical requests (demand *and* prefetch requests) from the core with high prefetch accuracy can contend with the critical requests (demand requests *only*) from the core with low prefetch accuracy. To avoid this, we boost the demand requests of the core with low prefetch accuracy. Second, the performance of a core with low prefetch accuracy is already affected negatively by useless prefetches. By prioritizing the demand requests of such cores, we aim to help the performance of cores that are already losing performance due to poor prefetcher behavior. We further discuss the effect of prioritizing urgent requests in Section 5.5.3.4.

Finally, if all else is equal, older requests have priority over younger requests.

### 5.2.3 Adaptive Prefetch Dropping

APS naturally delays (just like the demand-first policy) the DRAM service of prefetch requests from applications with low prefetch accuracy by making the prefetch requests non-critical as described in Section 5.2.2. Even though this reduces the interference of useless requests with useful requests, it cannot get rid of all of the negative effects of useless prefetch requests (bandwidth consumption, cache pollution) because such requests will eventually be serviced. As such, APS by itself cannot eliminate all of the negative aspects of useless prefetches. Our second scheme, Adaptive Prefetch Dropping (APD), aims to overcome this limitation by proactively removing old prefetch requests from the DRAM request buffer if they have been outstanding for a long period of time. The key insight is that if a prefetch request is old (i.e., has been outstanding for a long time), it is likely to be useless and dropping it from the memory request buffer eliminates the negative effects the useless request might cause in the future. We first describe why old prefetch requests are likely to be useless based on empirical measurements.

**Why are old prefetch requests likely to be useless?** Figure 5.4(a) shows the memory service time (from entry into the DRAM request buffer to entry into the last-level cache fill buffer) of both useful and useless prefetches for *milc* using the demand-first scheduling policy. Note that we show detailed data for only `milc` but found similar behavior in other applications. The graph is a histogram with nine latency intervals measured in processor cycles. Each bar indicates the number of useful/useless prefetch requests whose memory service time was within that interval. 56% of all prefetches have a service time greater than 1600 processor cycles, and 86% of these prefetches are useless. Useful prefetches tend to have a shorter service time than useless prefetches (1486 cycles compared to 2238 cycles on average for *milc*). This is because a prefetch request that is waiting in the request buffer becomes a demand request if the processor sends a demand request for that same address while the prefetch request is still in the buffer.[1] Such useful prefetches that

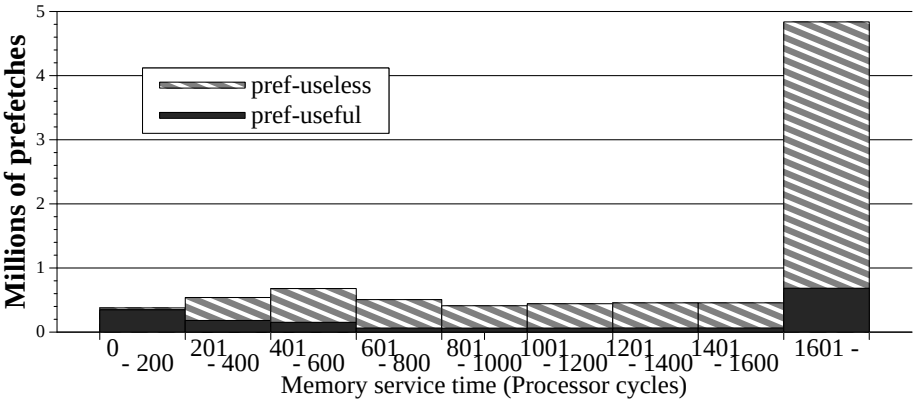
[1] A prefetch request that is hit by a demand request in the DRAM request buffer becomes a real demand request. However, we count it as a useful prefetch throughout this dissertation since it was

are hit by demand requests will be serviced earlier by the demand-first prioritization policy. Therefore, useful prefetches on average experience a shorter service time than useless prefetches. This is also true when we apply APS since it prioritizes critical requests over non-critical requests.

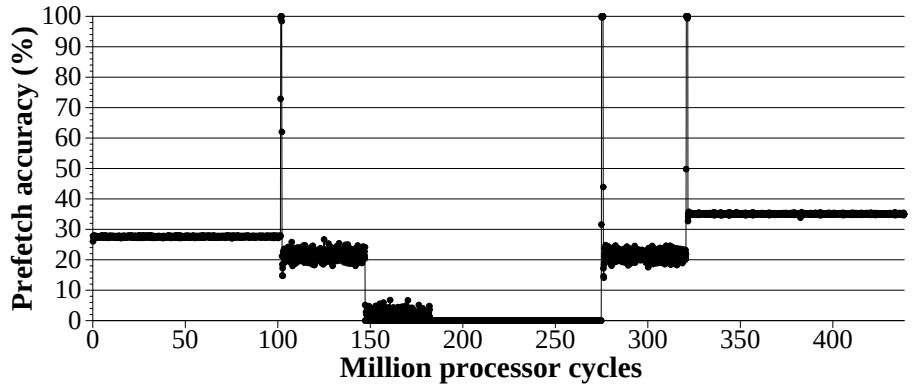
**Mechanism:** The observation that old prefetch requests are likely to be useless motivates us to remove a prefetch request from the request buffer if the prefetch is old enough. Our proposal, APD, monitors prefetch requests for each core and invalidates any prefetch request that has been outstanding in the DRAM request buffer for longer than $drop\_threshold$ cycles. We adjust $drop\_threshold$ based on the prefetch accuracy for each core measured in the previous time interval. If the prefetch accuracy in the interval is low, our mechanism uses a relatively low value for $drop\_threshold$ so that it can quickly remove useless prefetches from the request buffer. If the prefetch accuracy is high in the interval, our mechanism uses a relatively high value for $drop\_threshold$ so that it does not prematurely remove useful prefetches from the request buffer. By removing useless prefetches, APD saves resources such as request buffer entries, DRAM bandwidth, and cache space, which can instead be used for critical requests (i.e., demand and useful prefetch requests) rather than being wasted on useless prefetch requests. Note that APD interacts positively with APS since APS naturally delays the service of useless (non-critical) requests so that the APD unit can completely remove them from the memory system thereby freeing up request buffer entries and avoiding unnecessary bandwidth consumption.

**Determining $drop\_threshold$:** Figure 5.4(b) shows the runtime behavior of the stream prefetcher accuracy for *milc*, an application that suffers from many useless prefetches. Prefetch accuracy was measured as described in Section 5.2.1 using an interval of 100K cycles. The figure clearly shows that prefetch accuracy can have very strong phase behavior. From 150 million to 275 million cycles, the prefetch accuracy is very low (close to 0%) implying many useless prefetch requests

first requested by the prefetcher rather than the processing core.




(a) Prefetch service time with demand-first policy




(b) Prefetch accuracy every 100K cycles

Figure 5.4: Example of behavior of prefetches for *milc*

were generated during this time. Since almost all prefetches are useless during this period, we would like to be able to quickly drop them. Our mechanism accomplishes this using a low value for $drop\_threshold$. On the other hand, we would want $drop\_threshold$ to be much higher during periods of high prefetch accuracy. Our evaluation shows that a simple 4-level $drop\_threshold$ adjusted dynamically can effectively eliminate useless prefetch requests from the memory system while keeping useful prefetch requests in the DRAM request buffer.

## 5.3 Experimental Methodology

### 5.3.1 Metrics

We define the metrics used for experimental evaluation in this section. *Bus traffic* is the number of cache lines transferred over the bus during the execution of a workload. It comprises the cache lines brought in from demand, useful prefetch, and useless prefetch requests. We define *Prefetch accuracy (ACC)* and *coverage (COV)* as follows:

$$ACC = \frac{Number\ of\ useful\ prefetches}{Number\ of\ prefetches\ sent}$$

$$COV = \frac{Number\ of\ useful\ prefetches}{Number\ of\ demand\ requests + Number\ of\ useful\ prefetches}$$

To evaluate the effect of DRAM throughput improvement on the processing core, we define *instruction window Stall cycles Per Load instruction (SPL)* which indicates on average how much time the processor spends idly waiting for DRAM service.

$$SPL = \frac{Total\ number\ of\ window\ stall\ cycles}{Total\ number\ of\ load\ instructions}$$

To measure CMP system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [71], and *Harmonic mean of Speedups (HS)* [43]. As shown by Eyerman and Eeckhout [13], WS corresponds to system throughput and HS corresponds to the inverse of job turnaround time. In the equations that follow, $N$ is the number of cores in the CMP system. $IPC^{alone}$ is the IPC measured when an application runs alone on one core in the CMP system (other cores are idle) and $IPC^{together}$ is the IPC measured when an application runs on one core while other applications are running on the other cores of the CMP. Unless otherwise mentioned, we use the demand-first policy to measure $IPC^{alone}$ for all of our experiments to show the effectiveness of our mechanism on CMP systems.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad WS = \sum_i^N \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad HS = \frac{N}{\sum_i^N \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

### 5.3.2 System Model

We use an in-house cycle accurate x86 CMP simulator for our evaluation. Our processor faithfully models port contention, queuing effects, bank conflicts, and other DDR3 DRAM system constraints. The baseline configuration of each processing core is shown in Table 5.1. The shared resource configuration for single, 2, 4, and 8-core CMPs is shown in Table 5.2. Note that we evaluate our mechanism on CMP systems with private on-chip last-level caches (512KB for each core) rather than a shared cache to easily show and analyze the effect of PADC in the shared DRAM system by isolating the effect of contention in the DRAM system from the effect of interference in shared caches. We evaluate our mechanism for a shared last-level cache in Section 5.5.9 as well.

| | |
|---|---|
| Execution core | Out of order; 15 stages; decode/retire up to 4 instructions,<br>issue/execute up to 8 microinstructions<br>256-entry reorder buffer; 32-entry load-store queue |
| Front end | Fetch up to 2 branches; 4K-entry BTB; 64K-entry gshare[44],<br>64K-entry PAs [83], 64K-entry selector hybrid branch predictor [25] |
| On-chip caches | L1 I and D caches: 32KB, 4-way, 2-cycle, 1 read and 1 write ports;<br>Unified last-level cache: 512KB (1MB for 1-core), 8-way, 8-bank,<br>15-cycle, 1 read/write port; 64B line size for all caches |
| Prefetcher | Stream prefetcher with 32 streams, prefetch degree of 4,<br>cache line prefetch distance (lookahead) of 64 [77, 73] |

Table 5.1: Baseline configuration of each core for PADC

### 5.3.3 Workloads

We use the SPEC 2000/2006 benchmarks for experimental evaluation. Each single-threaded benchmark was compiled using ICC (Intel C Compiler) or IFORT

| | |
|---|---|
| DRAM controller | On-chip, demand-first FR-FCFS scheduling policy;<br>1 controller for 1, 2, 4, 8-core CMP (also 2 for 4, 8-core)<br>64, 64, 128, 256-entry last-level cache MSHR/DRAM request buffer for 1, 2, 4, 8-core |
| DRAM and bus | DDR3 1333MHz [49], 16B-wide data bus per controller<br>Latency: 15-15-15ns ($t_{RP}$,$t_{RCD}$, $CL$), BL = 4;<br>8 DRAM banks, 4KB row buffer per bank |

Table 5.2: Baseline configuration of shared CMP resources for PADC

(Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [61] as a representative portion of each benchmark.

We classify the benchmarks into three categories: prefetch-insensitive, prefetch-friendly, and prefetch-unfriendly (class 0, 1, and 2 respectively) based on the performance impact the stream prefetcher described in Table 5.1 has on the application. If MPKI (last-level cache Misses Per 1K Instructions) increases when the prefetcher is enabled, the benchmark is classified as 2. If MPKI without prefetching is greater than 10 (indicating memory intensive) and bus traffic increases by more than 75% when prefetching is enabled the benchmark is also classified as 2. Otherwise, if IPC increases by 5%, the benchmark is classified as 1. Otherwise, it is classified as 0. Note that memory intensive applications that experience increased IPC and reduced MPKI (such as *milc*) may still be classified as prefetch-unfriendly if bus traffic increases significantly. The reason for this is that although an increase in bus traffic may not have much of a performance impact on single core systems, in CMP systems with shared resources, the additional bus traffic can degrade performance substantially. The characteristics for a subset of benchmarks with and without a stream prefetcher are shown in Table 5.3. We evaluate the entire set of 55 SPEC CPU 2000/2006 benchmarks for single core experiments for our results. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from the 55 SPEC 2000/2006 benchmarks. We ran 54, 32, and 21 randomly chosen workload combinations (from the 55 SPEC benchmarks) for our 2,

4, and 8-core CMP configurations respectively.

| | No prefetcher | | Prefetcher with demand-first policy | | | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | IPC | MPKI | IPC | MPKI | RBH(%) | ACC(%) | COV(%) | Class |
| eon_00 | 2.08 | 0.01 | 2.08 | 0.00 | 84.93 | 37.37 | 52.64 | 0 |
| swim_00 | 0.35 | 27.57 | 0.62 | 8.66 | 42.83 | 99.95 | 68.58 | 1 |
| galgel_00 | 1.42 | 4.26 | 1.10 | 7.56 | 65.50 | 30.96 | 23.94 | 2 |
| art_00 | 0.18 | 89.39 | 0.18 | 65.52 | 91.46 | 35.88 | 34.00 | 2 |
| ammp_00 | 1.70 | 0.80 | 1.47 | 1.70 | 56.20 | 5.96 | 8.03 | 2 |
| gcc_06 | 0.55 | 6.28 | 0.81 | 2.23 | 81.57 | 32.62 | 65.37 | 1 |
| mcf_06 | 0.13 | 33.73 | 0.15 | 29.70 | 25.63 | 31.43 | 14.75 | 1 |
| sjeng_06 | 1.57 | 0.38 | 1.57 | 0.38 | 25.13 | 1.67 | 1.11 | 0 |
| omnetpp_06 | 0.41 | 10.16 | 0.44 | 9.57 | 61.86 | 10.50 | 18.33 | 2 |
| libquantum_06 | 0.41 | 13.51 | 0.65 | 2.75 | 81.39 | 99.98 | 79.63 | 1 |
| xalancbmk_06 | 0.80 | 1.70 | 0.71 | 2.12 | 49.35 | 8.96 | 13.26 | 2 |
| bwaves_06 | 0.59 | 18.71 | 1.23 | 0.37 | 83.99 | 99.97 | 98.00 | 1 |
| milc_06 | 0.41 | 29.33 | 0.46 | 20.88 | 81.13 | 19.45 | 28.81 | 2 |
| cactusADM_06 | 0.71 | 4.54 | 0.84 | 2.21 | 33.56 | 45.12 | 51.47 | 1 |
| leslie3d_06 | 0.53 | 20.89 | 0.86 | 2.41 | 77.32 | 89.72 | 88.66 | 1 |
| soplex_06 | 0.35 | 21.25 | 0.72 | 3.61 | 78.81 | 80.12 | 83.08 | 1 |
| GemsFDTD_06 | 0.44 | 15.61 | 0.80 | 2.02 | 55.82 | 90.71 | 87.12 | 1 |
| lbm_06 | 0.46 | 20.16 | 0.70 | 2.93 | 58.24 | 94.27 | 85.45 | 1 |

Table 5.3: Characteristics of 18 SPEC benchmarks for PADC: IPC, MPKI (last-level cache misses per 1K instructions), RBH (Row Buffer Hit rate), ACC (prefetch accuracy), COV (prefetch coverage), class

## 5.4 Implementation and Hardware Cost of PADC

An implementation of PADC requires storing additional information in each DRAM request buffer entry to support the priority and aging information needed by APS and APD. The required additional information (in terms of the fields added to each request buffer entry) is shown in Figure 5.5.

The C (as prefetch bit), RH, and FCFS fields are already used in the baseline demand-first FR-FCFS policy to indicate prefetch status (i.e., demand or prefetch), row-hit status, and arrival time of the request. Therefore the only additional fields are U, P, ID, and AGE, which indicate the urgency, prefetch status, core ID, and age of the request. Each DRAM cycle, the priority encoder logic of APS chooses the highest priority request using the priority fields (C, RH, U, and FCFS) in the order

shown in Figure 5.5.

The APD unit removes a prefetch request from the DRAM request buffer if the request is older than the $drop\_threshold$ of the core that generated the request. It does not remove a prefetch request (which is not scheduled for DRAM service) until it ensures that the prefetch cannot be matched by a demand request. This is accomplished by invalidating the MSHR entry of the prefetch request before actually dropping it. The APD unit knows if a request is a prefetch and also which core it belongs to from the P and ID fields. The AGE field of each request entry keeps track of the age of the request. APD compares the AGE of the request to the corresponding core's $drop\_threshold$ and removes the request accordingly. Note that the estimation of the age of a request does not need to be highly accurate. For example, the AGE field is incremented every 100 processor cycles for our evaluation.

The hardware storage cost required for our implementation of the PADC is shown in Table 5.4. Note that the storage cost for PADC linearly increases with the number of cores, request buffer entries, and cache lines. The storage cost for our 4-core CMP system described in Section 5.3.2 is only 34,720 bits (∼4.25KB) which is equivalent to only 0.2% of the last-level cache data storage in our baseline 4-core CMP. Note that the Prefetch bit (P) per cache line accounts for over 4KB of storage by itself (∼95% of the total required storage). Many previous proposals [14, 69, 90, 91, 73] already use a prefetch bit for each cache line. If a processor already employs prefetch bits in its cache, the total additional storage cost of our prefetch-aware DRAM controller is only 1,824 bits (∼228B). Note that the overhead of prefetch

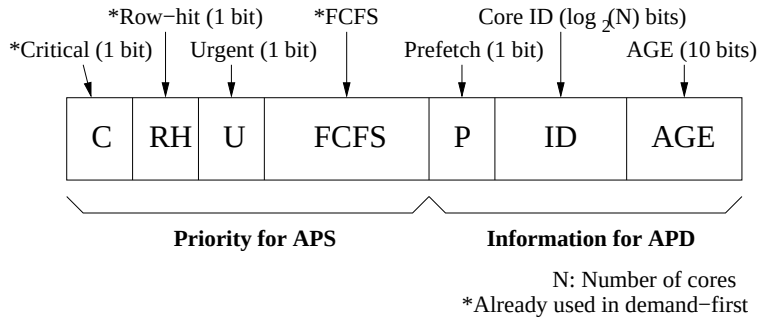



Figure 5.5: DRAM request field for PADC

bits can also be reduced by using set sampling [63], i.e. associating prefetch bits with only a selected number of sets.

| | Bit field | Cost equation (bits) | Cost (bits) |
|---|---|---|---|
| Prefetch accuracy | P (1 bit) | $N_{cache} \times N_{core} + N_{req}$ | 32,896 |
| | PSC (16 bits) | $N_{core} \times 16$ | 64 |
| | PUC (16 bits) | $N_{core} \times 16$ | 64 |
| | PAR (8 bits) | $N_{core} \times 8$ | 32 |
| APS | U (1 bit) | $N_{req}$ | 128 |
| APD | ID ($log_2 N_{core}$ bits) | $N_{req} \times log_2 N_{core}$ | 256 |
| | AGE (10 bits) | $N_{req} \times 10$ | 1,280 |
| Total storage cost for the 4-core system in Section 5.3.2 | | | 34,720 |
| Total storage cost as a fraction of the last-level cache capacity | | | 0.2% |

Table 5.4: Hardware storage cost of PADC: $N_{cache}$: number of cache lines per core $N_{core}$: number of cores, $N_{req}$: number of DRAM request buffer entries)

For the evaluation of our PADC, we use a prefetch accuracy value of 85% for $promotion\_threshold$ (for APS) and a dynamic threshold shown in Table 5.5 for $drop\_threshold$ (for APD). The accuracy is calculated every 100K cycles.

| Prefetch accuracy (%) | 0 - 10 | 10 - 30 | 30 - 70 | 70 - 100 |
|---|---|---|---|---|
| $drop\_threshold$ (processor cycles) | 100 | 1,500 | 50,000 | 100,000 |

Table 5.5: Dynamic $drop\_threshold$ values for Adaptive Prefetch Dropping based on prefetch accuracy

## 5.5 Experimental Evaluation and Analysis on PADC

We first evaluate PADC on single, 2, 4, and 8-core systems. Section 5.5.5 analyzes PADC's fairness and discusses additional techniques to improve CMP system fairness. Sections 5.5.6 through 5.5.14 analyze the effect of PADC on systems with different configurations and characteristics such as multiple DRAM controllers, different row buffer policies, different types of prefetchers, prefetch filtering, and runahead execution. This analysis shows that PADC is a general mechanism that is effective for a variety of systems and that it is orthogonal to previously

proposed prefetching and prefetch filtering techniques.

### 5.5.1 Single-Core Results

Figure 5.6 shows the performance of PADC on a single-core system. IPC is normalized to the baseline which employs the demand-first scheduling policy. We show the performance of only 15 individual benchmarks. The rightmost bars show the average performance of all 55 SPEC CPU 2000/2006 benchmarks (*gmean55*). As discussed earlier, neither of the rigid scheduling policies (demand-first, demand-prefetch-equal) provides the best performance across all applications. Demand-first performs better for most prefetch-unfriendly benchmarks (class 2) such as *galgel*, *art* and *ammp* while demand-prefetch-equal does better for most prefetch-friendly ones (class 1) such as *swim*, *libquantum* and *lbm*. Averaged over all 55 SPEC benchmarks, the demand-prefetch-equal policy outperforms demand-first by 0.5% since there are more benchmarks (29 out of 55) that belong to class 1.

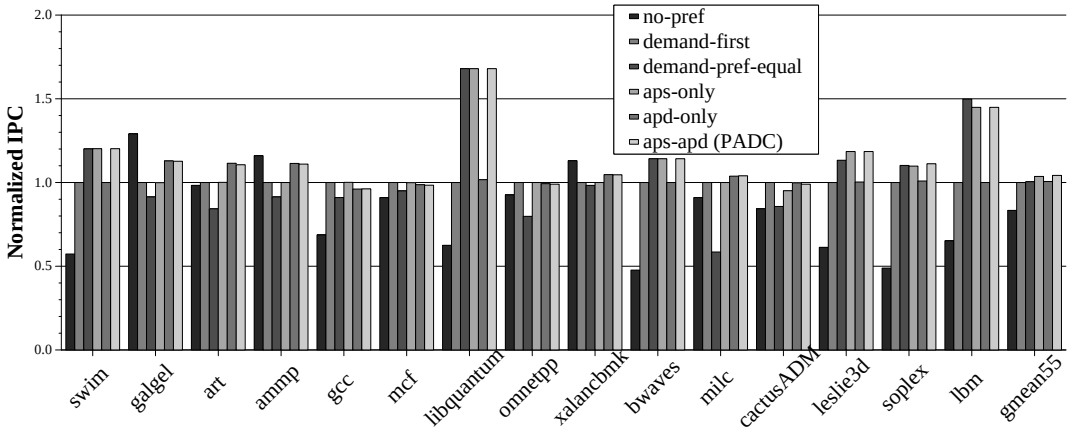



Figure 5.6: Performance of PADC on single-core system: Normalized IPC for 15 benchmarks and average for all 55 (gmean55)

Adaptive Prefetch Scheduling (APS), shown in the fourth bar from the left, effectively adapts to the behavior of the prefetcher. In most benchmarks, APS provides at least as good performance as the best rigid prefetch scheduling policy. As a result, APS improves performance by 3.6% over all 55 benchmarks compared to the baseline. APS (and demand-prefetch-equal) improves performance over

demand-first for many prefetch friendly applications such as *libquantum, bwaves,* and *leslie3d*. This is due to two reasons. First, APS increases DRAM throughput in these applications because it treats demands and prefetches equally most of the time. Doing so improves the timeliness of the prefetcher because prefetch requests do not get delayed behind demand requests. Second, improved DRAM throughput reduces the probability of the DRAM request buffer being full. As a result, more prefetches are able to enter the request buffer. This improves the coverage of the prefetcher as more useful prefetch requests get a chance to be issued. For example, APS improves the prefetch coverage from 80%, 98%, and 89% to 100%, 100%, and 92% for *libquantum, bwaves,* and *leslie3d* respectively (as shown in Figure 5.8).

On the other hand, even though APS is able to provide the performance of the best rigid prefetch scheduling policy for each application, it is unable to overcome the performance loss due to prefetching in some prefetch-unfriendly applications such as *galgel, ammp* and *xalancbmk*. The prefetcher generates many useless prefetches in these benchmarks that a simple DRAM scheduling policy cannot eliminate.

When adaptive prefetch dropping (APD) is employed with demand-first (APD-only), it improves performance for prefetch-unfriendly applications by eliminating many useless prefetches. This is also true when APD is employed with APS (i.e., PADC). Using APD recovers part of the performance loss due to prefetching in *galgel, ammp,* and *xalancbmk* because it eliminates 54%, 76%, and 54% of the useless prefetch requests respectively as shown in Figure 5.8. As a result, using both of our proposed mechanisms (APD in conjunction with APS) provides 4.3% performance improvement over the baseline for all 55 SPEC 2000/2006 benchmarks. Note that for 17 most memory intensive SPEC benchmarks, PADC improves performance by 11.8% (not shown in the figure).

Figure 5.7 provides insight into the performance improvement of the proposed mechanisms by showing the effect of each mechanism on the stall time experienced per load instruction (SPL). Our PADC reduces SPL by 5.0% compared

to the baseline. By providing better DRAM scheduling and eliminating useless prefetches, PADC reduces the amount of time the processor stalls for each load instruction and allows the processor to make faster progress. As a result, PADC significantly improves performance.

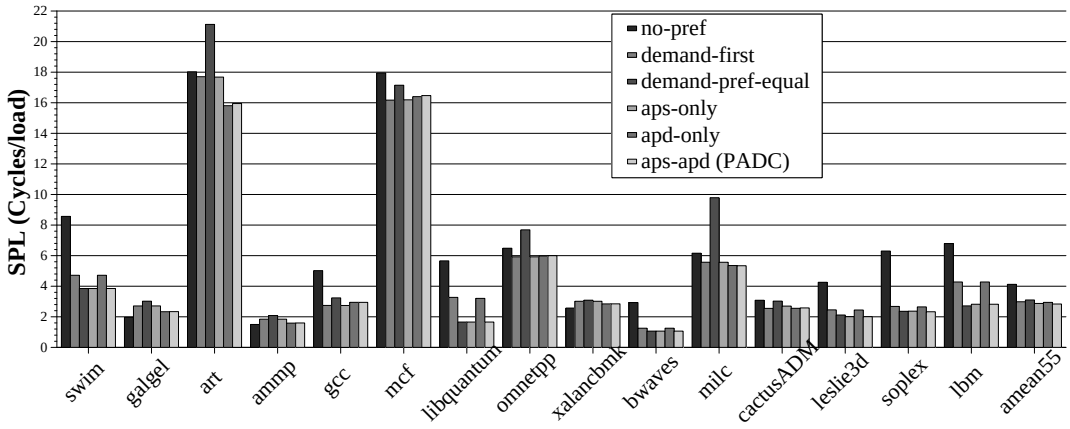



Figure 5.7: Stall time per load (SPL) of PADC on single-core system

Figure 5.8 breaks down the bus traffic into three categories: useful prefetches, useless prefetches, and demand requests. PADC reduces bus traffic by 10.4% across all benchmarks (amean55) as shown. Reduction in bus traffic is mainly due to APD which significantly reduces the number of useless prefetches. For many benchmarks, APS by itself provides the same bandwidth consumption provided by the best rigid policy for each benchmark. We conclude that our prefetch-aware DRAM controller is very effective at improving both performance and bandwidth-efficiency in the single-core system.

Note that simply turning off prefetching for prefetch-unfriendly applications may lose opportunity to improve performance. This is true for prefetch-unfriendly applications that have 1) significant phase changes, 2) some accurate prefetches interleaved with inaccurate prefetches. For such benchmarks, prefetching hurts performance in some phases but increases performance significantly in others. If the prefetcher is turned off, the performance benefits of useful prefetch phases and useful prefetch requests will be lost. In fact, due to this phase behavior, *art* and *milc* do not benefit much from prefetching unless adaptive prefetch management is used.

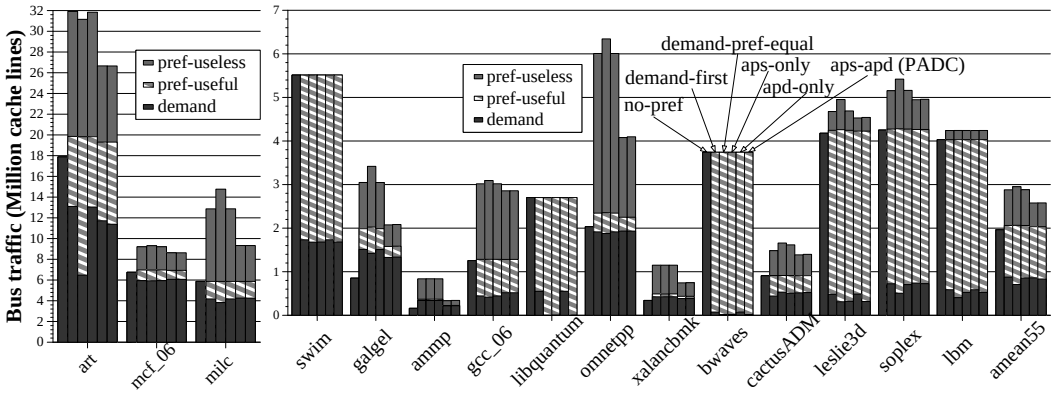



Figure 5.8: Bus traffic of PADC on single-core system

Figure 5.6 shows that PADC improves performance significantly for *art* and *milc* (compared to no prefetching) since it is able to adapt to different phases and eliminate useless prefetches while keeping useful prefetches.

#### 5.5.1.1 Adaptive Behavior of PADC

We analyze the adaptive runtime behavior of PADC in this section. APS prioritizes demands over prefetches (i.e., demand-first) when the estimated prefetch accuracy is less than $promotion\_threshold$. It treats demands and prefetches equally (i.e., demand-prefetch-equal) when prefetch accuracy is greater than or equal to $promotion\_threshold$. PADC continuously changes the DRAM scheduling mode (between demand-first and demand-prefetch-equal) for the application based on the prefetch accuracy estimated every interval.

Figure 5.9 shows the fraction of time APS and PADC spend in each of the two scheduling modes for the single-core system. APS and PADC spend a majority of their execution time in demand-prefetch-equal mode for prefetch-friendly applications but spend most of their execution time in demand-first mode for prefetch-unfriendly applications. Therefore, APS and PADC provide at least as good performance as the best rigid prefetch scheduling policy in most applications, as shown in Figure 5.6.

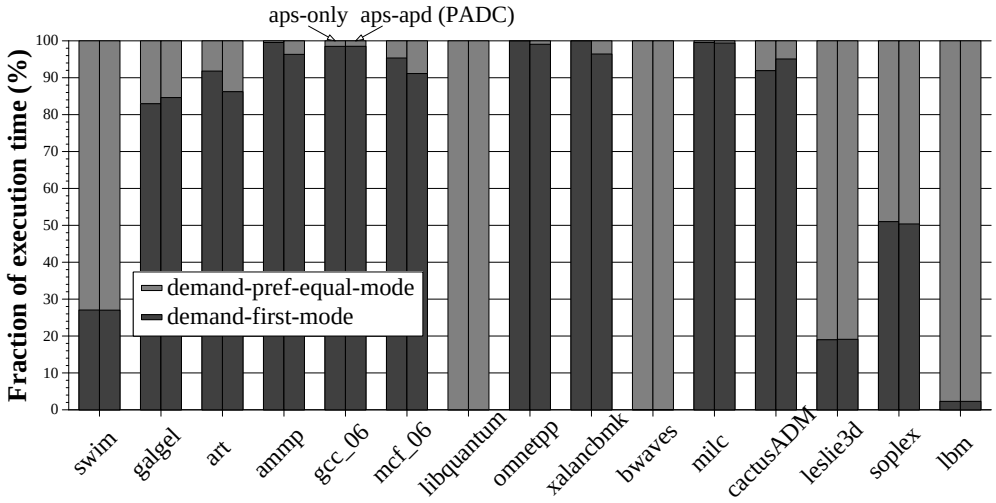



Figure 5.9: Fraction of execution time in different PADC scheduling modes on single-core system

#### 5.5.1.2 Effect of PADC on Row Buffer Hit Rate

Recall that the demand-prefetch-equal policy prioritizes row-hit requests regardless of whether a request is a prefetch or demand. If we consider all demand and prefetch requests (regardless of whether or not a prefetch is useful) for the entire run of an application, the demand-prefetch-equal policy will result in the highest row buffer hit rate (RBH) and therefore the lowest average DRAM access latency among all considered policies. However, this does not mean that this policy performs best since prefetches are NOT always useful as discussed in Section 5.5.1. When prefetching is enabled, we need a better metric to show how a mechanism reduces effective memory latency. Hereby, we define row buffer hit rate for useful (demand and useful prefetch) requests (RBHU) as follows:

$$RBHU = \frac{Number\ of\ row\text{-}hit\ demands + Number\ of\ useful\ row\text{-}hit\ prefetches}{Number\ of\ demands + Number\ of\ useful\ prefetches}$$

The demand-prefetch-equal policy will still show the highest RBHU since RBHU is also maximized by prioritizing row-hit requests. However, a good DRAM scheduling mechanism should keep its RBHU close to demand-prefetch-equal's RBHU because it should aim to maximize DRAM bandwidth for useful requests.

Table 5.6 shows RBHU values for 13 benchmarks on the single-core processor with no prefetching, demand-first, demand-prefetch-equal, APS, and PADC. The RBHU of APS is very close to that of demand-prefetch-equal and significantly better than the RBHU of demand-first since APS successfully exploits row buffer locality for useful requests.

Employing APD with APS (i.e., PADC) slightly reduces RBHU for some applications such as *galgel*, *ammp*, *mcf*, *omnetpp*, *xalancbmk*, and *soplex*. This is because adaptive prefetch dropping cancels some useful prefetches as shown in Figure 5.8, thereby reducing the fraction of useful row buffer hits. Nonetheless, APD improves overall performance for these applications since it reduces the contention between demands and prefetches by eliminating a significant number of useless prefetches as discussed in Section 5.5.1.

| Benchmark | swim | galgel | art | ammp | mcf_06 | libquantum | omnetpp |
|---|---|---|---|---|---|---|---|
| no-pref | 0.18 | 0.51 | 0.94 | 0.40 | 0.12 | 0.86 | 0.47 |
| demand-first | 0.44 | 0.58 | 0.94 | 0.48 | 0.19 | 0.86 | 0.56 |
| demand-pref-equal | 0.50 | 0.58 | 0.96 | 0.50 | 0.23 | 0.98 | 0.59 |
| aps | 0.50 | 0.58 | 0.94 | 0.48 | 0.19 | 0.98 | 0.56 |
| aps-apd (PADC) | 0.50 | 0.56 | 0.94 | 0.44 | 0.18 | 0.98 | 0.54 |

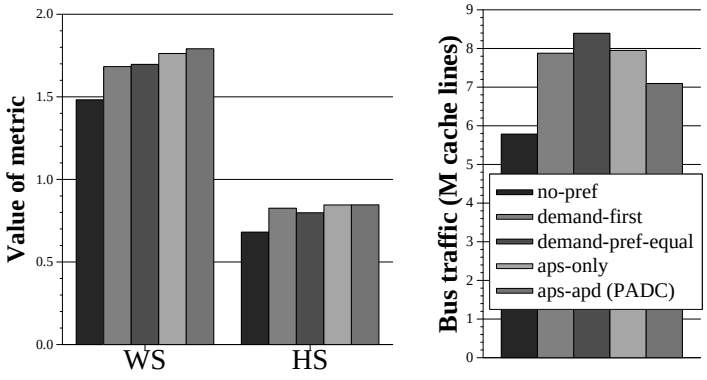
| Benchmark | xalancbmk | bwaves | milc | leslie3d | soplex | lbm | amean55 |
|---|---|---|---|---|---|---|---|
| no-pref | 0.23 | 0.76 | 0.85 | 0.71 | 0.81 | 0.53 | 0.55 |
| demand-first | 0.27 | 0.87 | 0.88 | 0.81 | 0.87 | 0.64 | 0.63 |
| demand-pref-equal | 0.28 | 0.89 | 0.90 | 0.91 | 0.93 | 0.92 | 0.68 |
| aps | 0.27 | 0.89 | 0.88 | 0.90 | 0.91 | 0.90 | 0.66 |
| aps-apd (PADC) | 0.25 | 0.89 | 0.88 | 0.90 | 0.90 | 0.90 | 0.65 |

Table 5.6: Row buffer hit rate of PADC for useful requests

### 5.5.2 2-Core Results

We briefly discuss only the average performance and bus traffic for the 54 workloads on the 2-core system. Figure 5.10 shows that PADC improves both performance metrics (weighted speedup and harmonic mean of speedups) by 8.4%, and 6.4% respectively compared to the demand-first policy and also reduces memory bus traffic by 10.0%. Thus, the proposed mechanism is effective for dual-core

systems. We do not discuss these results further since dual-core processors are no longer the state-of-the-art in multi-core systems. We extensively analyze PADC on 4-core systems in the next sections.




(a) System performance (b) Average traffic

Figure 5.10: Performance of PADC on 2-core system

### 5.5.3 4-Core Results

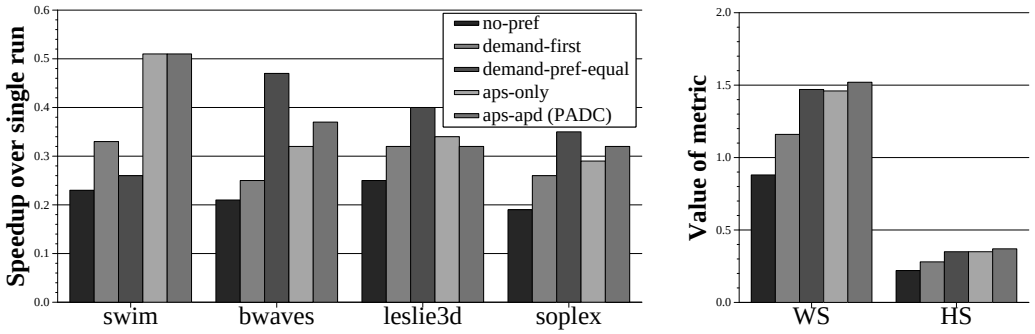
We ran 32 different workloads to evaluate the effectiveness of PADC on the 4-core system. In the following sections, we discuss three case studies in detail to provide insights into the behavior of the Prefetch-Aware DRAM Controller on a CMP system.

#### 5.5.3.1 Case Study I: All Prefetch-Friendly Applications

Our first case study examines the behavior of our proposed mechanisms when four prefetch-friendly applications (*swim, bwaves, leslie3d,* and *soplex*) run together on the 4-core system. Figure 5.11(a) shows the speedup of each application and Figure 5.11(b) shows system performance.

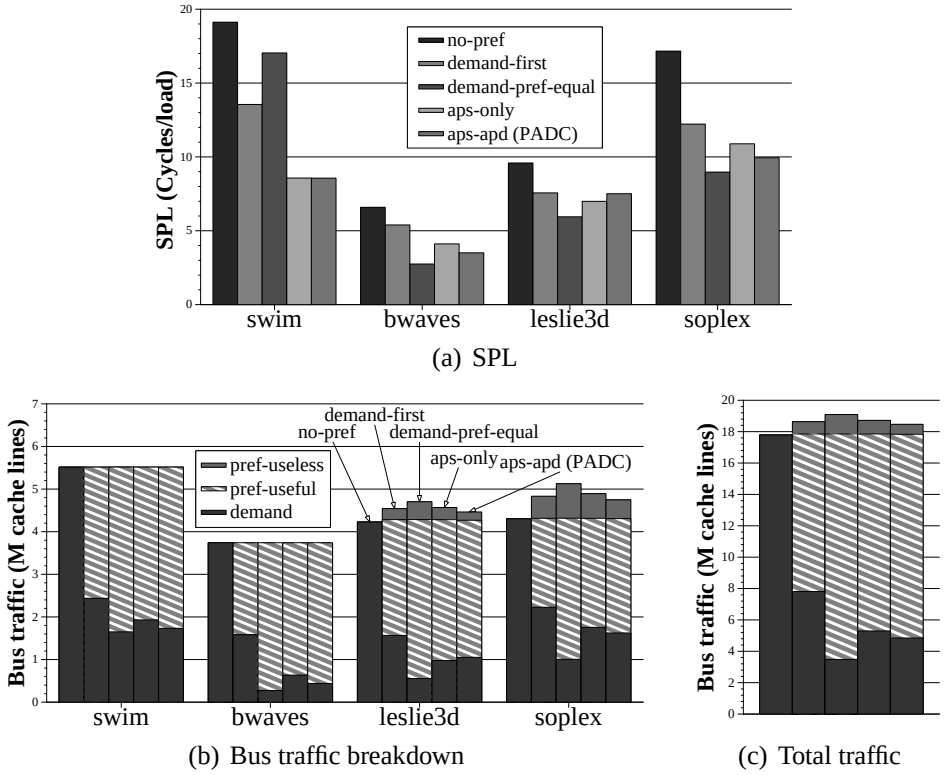
In addition, Figure 5.12 provides insight into the performance changes by showing how each mechanism affects stall-time per load as well as memory bus traffic. Several observations are in order:

First, since all four applications show very high prefetch accuracy/coverage




(a) Individual speedup (b) System performance

Figure 5.11: Performance of PADC for prefetch-friendly 4-core workload




(a) SPL

(b) Bus traffic breakdown (c) Total traffic

Figure 5.12: SPL and bus traffic of PADC for prefetch-friendly 4-core workload

(i.e., prefetch-friendly) as shown in Figure 5.12(b), prefetching provides significant performance improvement in all applications regardless of the DRAM scheduling policy. In addition, the demand-prefetch-equal policy significantly outperforms the demand-first policy (by 28% in terms of weighted speedup) because prefetches are very accurate in all four applications. The demand-prefetch-equal policy reduces stall-time per load as shown in Figure 5.12(a) because it improves DRAM throughput.

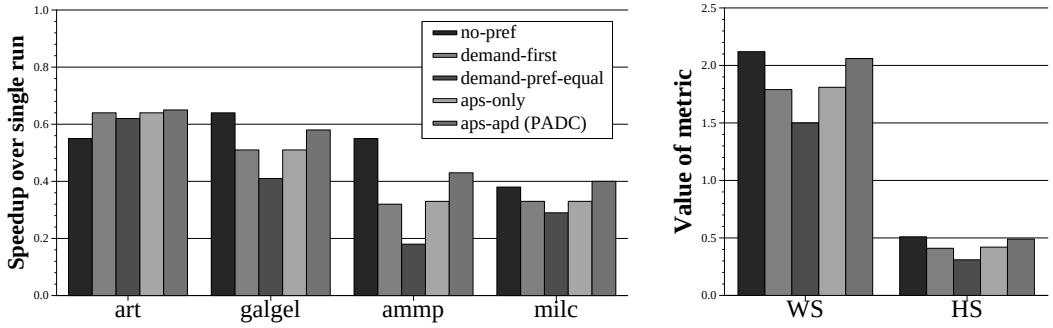
Second, PADC outperforms both of the rigid prefetch scheduling policies improving weighted speedup by 31.3% over the baseline demand-first policy. This is because it 1) successfully prioritizes critical (useful) requests over others thereby reducing SPL, and 2) drops useless prefetches in *leslie3d* and *soplex* thereby reducing their negative effects on all applications. Consequently, PADC also improves prefetch coverage from 56% to 73% as shown in Figure 5.12(c). This is because it improves DRAM throughput and reduces contention for memory system resources by dropping useless prefetches from *leslie3d* and *soplex* allowing more useful prefetches to enter the memory system.

Finally, the bandwidth savings provided by PADC is relatively small (0.9% compared to the baseline demand-first) because these applications do not generate a large number of useless prefetch requests. However, there is still a non-negligible reduction in bus traffic due to the effective dropping of useless prefetches in *leslie3d* and *soplex*. We conclude that the Prefetch-Aware DRAM Controller can provide system performance (WS and HS) and bandwidth-efficiency improvements even when all applications benefit significantly from prefetching.

#### 5.5.3.2 Case Study II: All Prefetch-Unfriendly Applications

The second case study examines the behavior of our proposed mechanisms when four prefetch-unfriendly applications (*art, galgel, ammp,* and *milc*) run together on the 4-core system. Since the prefetcher is very inaccurate for all applications, prefetching degrades performance regardless of the scheduling policy.

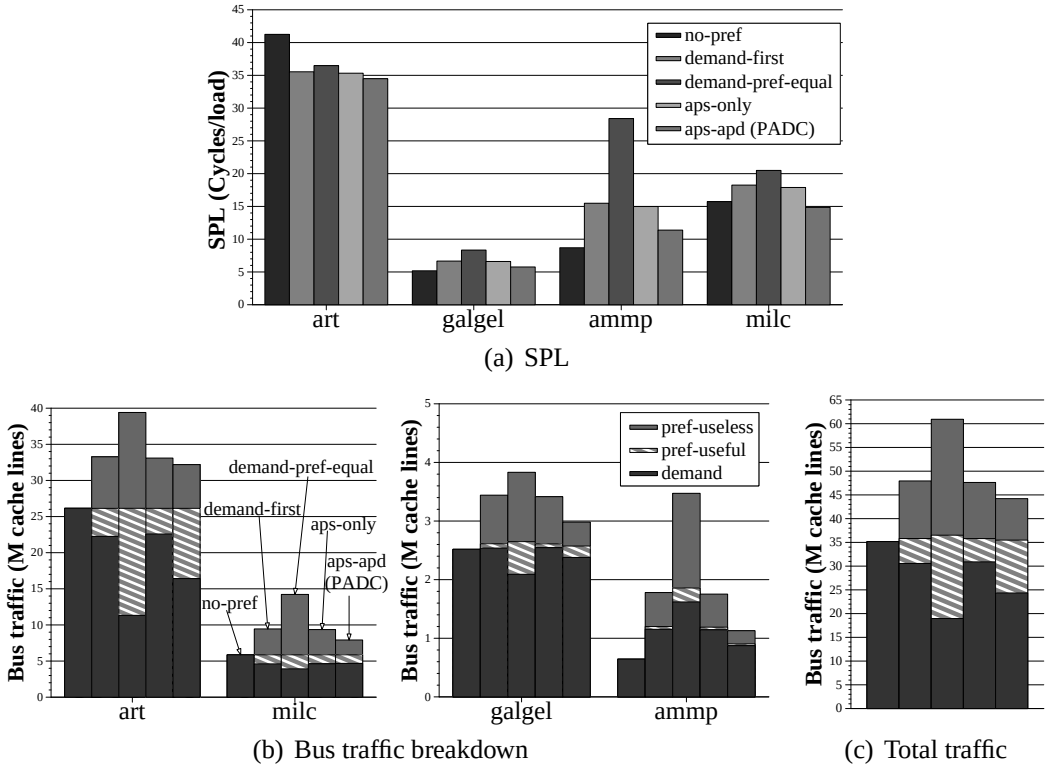
However, as shown in Figure 5.13, the demand-first policy and APS provide better performance than the demand-prefetch-equal policy by prioritizing demand requests over prefetch requests which are more than likely to be useless. Employing adaptive prefetch dropping drastically reduces the useless prefetches in all four applications as shown in Figure 5.14(b) and therefore frees up memory system resources to be used by demands and useful prefetch requests. The effect of this can be seen by the reduced SPL as shown in Figure 5.14(a) for all applications. As a result, our PADC performs better than either rigid prefetch scheduling policy for *all* considered applications.




(a) Individual speedup (b) System performance

Figure 5.13: Performance of PADC for prefetch-unfriendly 4-core workload

PADC improves system performance by 17.7% (weighted speedup) and 21.5% (harmonic mean of speedups), while reducing bandwidth consumption by 9.1% over the baseline demand-first scheduler as shown in Figure 5.14(c). By largely reducing the negative effects of useless prefetches both in scheduling and memory system buffers/resources, PADC almost eliminates the system performance loss observed in this prefetch-unfriendly mix of applications. Weighted speedup is within 2% and harmonic mean of speedups is within 1% of those obtained with no prefetching. We conclude that the Prefetch-Aware DRAM Controller can effectively eliminate the negative performance impact caused by inaccurate prefetching by intelligently managing the scheduling and buffer management of prefetch requests even in workload mixes where prefetching performs inefficiently for all applications.




(a) SPL

(b) Bus traffic breakdown

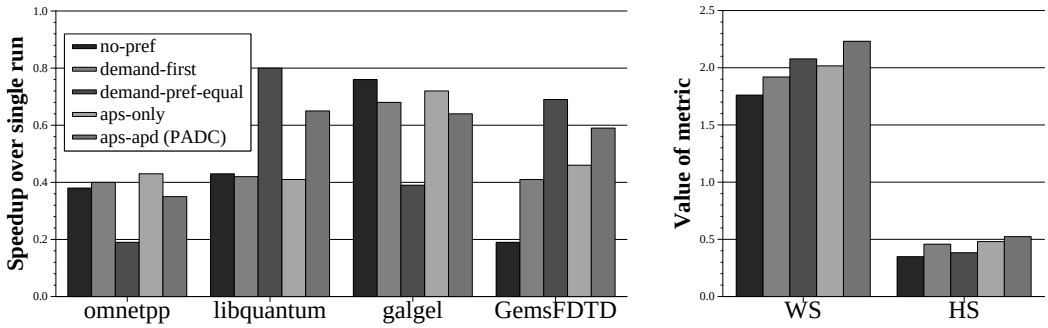
(c) Total traffic

Figure 5.14: SPL and bus traffic of PADC for prefetch-unfriendly 4-core workload

#### 5.5.3.3 Case Study III: Mix of Prefetch-Friendly and Prefetch-Unfriendly Applications

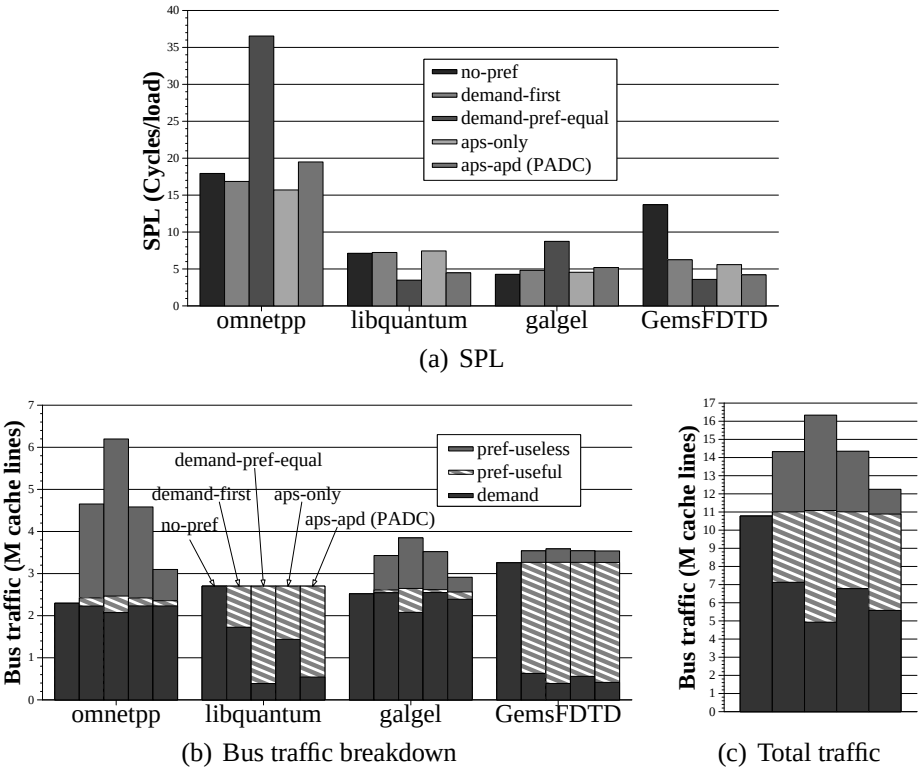
Our final case study examines the behavior of PADC when two prefetch-friendly (*libquantum* and *GemsFDTD*) and two prefetch-unfriendly (*omnetpp* and *galgel*) applications are run together on the 4-core system. Figures 5.15 and 5.16 show performance, SPL, and bus traffic.

The prefetches for *libquantum* and *GemsFDTD* are very beneficial. Therefore demand-prefetch-equal significantly improves weighted speedup. However, the prefetcher generates many useless prefetches for *omnetpp* and *galgel* as shown in Figure 5.16(b). These useless prefetches temporarily deny service to critical requests (demands and useful prefetches) from the two other cores. Because APD eliminates a large portion (67% and 57%) of all useless prefetches in *omnetpp* and *galgel*, it frees up both request buffer entries and bandwidth in the memory system. These freed up resources are utilized efficiently by the critical requests of




(a) Individual speedup (b) System performance

Figure 5.15: Performance of PADC for mixed 4-core workload




(a) SPL

(b) Bus traffic breakdown (c) Total traffic

Figure 5.16: SPL and bus traffic of PADC for mixed 4-core workload

*libquantum* and *GemsFDTD* thereby significantly improving their individual performance while slightly reducing *omnetpp* and *galgel*'s individual performance. Since it eliminates a large number of useless prefetches, PADC reduces total bandwidth consumption by 14.5% over the baseline demand-first policy. We conclude that PADC can effectively prevent the denial of service caused by the useless prefetches of prefetch-unfriendly applications on the useful requests of other applications.

#### 5.5.3.4 Effect of Prioritizing Urgent Requests

In this section, we discuss the effectiveness of prioritizing urgent requests using the application mix in case study III. We say that a multi-core system is *fair* if each application experiences the same individual speedup when multiple applications run together on the system. To indicate the degree of unfairness, we define *Unfairness (UF)* [13] as follows:

$$UF = \frac{MAX(IS_0,\ IS_1,\ \ldots,\ IS_{n-1})}{MIN(IS_0,\ IS_1,\ \ldots,\ IS_{n-1})}, \quad N:\ Number\ of\ Cores$$

Table 5.7 shows individual speedup, unfairness, weighted speedup, and harmonic mean of speedups for the workload from case study III for five policies: demand-first, versions of APS and PADC that do not use the concept of "urgent requests," and regular APS and PADC (with "urgent requests"). If the concept of "urgent requests" is not used, demand requests from the prefetch-unfriendly applications (*omnetpp* and *galgel*) unfairly starve because a large number of critical requests from the prefetch-friendly applications (*libquantum* and *GemsFDTD*) are given the same priority as those demand requests. This starvation, combined with the negative effects of useless prefetches, leads to unacceptably low individual speedups for these applications resulting in high unfairness. When urgency is used to prioritize requests, this unfairness is significantly mitigated as shown in Table 5.7. In addition, harmonic mean of speedups (i.e., average job turnaround time) significantly improves at the cost of very little weighted speedup (i.e., system throughput)

degradation. However, we found that for most workloads (30 out of the 32), prioritizing urgent requests improves weighted speedup as well. This trend holds true for most workload mixes that consist of prefetch-friendly and prefetch-unfriendly applications. On average (not shown in the table), prioritizing urgent requests improves UF, HS, and WS by 13.7%, 8.8%, and 3.8% respectively compared to PADC with no concept of urgency for the 32 4-core workloads. We conclude that incorporating the concept of urgency into PADC significantly improves system fairness while keeping system performance high.

| | Individual speedup | | | | UF | WS | HS |
|---|---|---|---|---|---|---|---|
| | omnetpp | libquantum | galgel | GemsFDTD | | | |
| demand-first | 0.40 | 0.42 | 0.68 | 0.41 | 1.69 | 1.92 | 0.46 |
| aps-no-urgent | 0.26 | 0.68 | 0.47 | 0.61 | 2.57 | 2.02 | 0.44 |
| aps | 0.43 | 0.41 | 0.72 | 0.46 | 1.73 | 2.02 | 0.48 |
| aps-apd-no-urgent | 0.21 | 0.94 | 0.42 | 0.70 | 4.55 | 2.26 | 0.41 |
| aps-apd (PADC) | 0.35 | 0.65 | 0.64 | 0.59 | 1.84 | 2.23 | 0.52 |

Table 5.7: Effect of prioritizing urgent requests in PADC

#### 5.5.3.5 Effect on Identical-Application Workloads

It is common that commercial servers frequently run multiple instances of identical applications. In this section, we evaluate the effectiveness of PADC when the 4-core system runs four identical applications together. Since APS prioritizes memory requests and APD drops useless prefetches (both based on the estimated prefetch accuracy), PADC should evenly improve individual speedup of each instance of the identical applications running together. In other words, all instances of the application are likely to show the same behavior and the same adaptive decision should be made for every interval.

Table 5.8 shows the system performance of PADC when four instances of *libquantum* run together on the 4-core system. Because *libquantum* is very prefetch-friendly and most prefetches are row-hits, the demand-prefetch-equal policy performs very well by achieving almost the same speedup for all four instances. APS and PADC perform similarly to demand-prefetch-equal (improving weighted

speedup by 18.2% compared to demand-first) since they successfully treat demands and prefetches equally for all four instances.

| | Individual speedup | | | | WS | HS | UF |
|---|---|---|---|---|---|---|---|
| | libquantum | libquantum | libquantum | libquantum | | | |
| no-pref | 0.60 | 0.60 | 0.60 | 0.59 | 2.40 | 0.60 | 1.01 |
| demand-first | 0.69 | 0.67 | 0.65 | 0.64 | 2.66 | 0.66 | 1.08 |
| demand-pref-equal | 0.80 | 0.79 | 0.78 | 0.77 | 3.14 | 0.78 | 1.05 |
| aps | 0.80 | 0.79 | 0.78 | 0.77 | 3.14 | 0.79 | 1.04 |
| aps-apd (PADC) | 0.80 | 0.79 | 0.78 | 0.77 | 3.14 | 0.79 | 1.04 |

Table 5.8: Effect of PADC on four identical prefetch-friendly applications

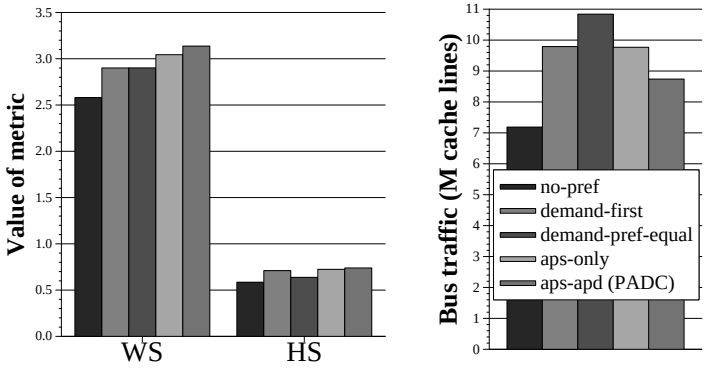
Table 5.9 shows the system performance of PADC when four instances of a prefetch-unfriendly application, *milc*, run together on the 4-core system. Because the prefetches generated for each instance are useless for most of the execution time of *milc*, demand-first and APS outperform demand-pref-equal for each instance. Incorporating APD into APS (i.e., PADC) further improves individual speedup of all instances equally by reducing useless prefetches from each instance. As a result, PADC significantly improves all system performance metrics. In fact, using PADC allows the system to gain significant performance improvement from prefetching whereas using a rigid prefetch scheduling policy results in a large performance loss due to prefetching. To conclude, PADC is also very effective when multiple identical applications run together on a CMP system.

| | Individual speedup | | | | WS | HS | UF |
|---|---|---|---|---|---|---|---|
| | milc | milc | milc | milc | | | |
| no-pref | 0.53 | 0.53 | 0.53 | 0.53 | 2.11 | 0.53 | 1.00 |
| demand-first | 0.52 | 0.51 | 0.50 | 0.46 | 1.99 | 0.50 | 1.13 |
| demand-pref-equal | 0.36 | 0.36 | 0.36 | 0.36 | 1.45 | 0.36 | 1.01 |
| aps | 0.52 | 0.51 | 0.50 | 0.46 | 1.99 | 0.50 | 1.14 |
| aps-apd (PADC) | 0.59 | 0.58 | 0.58 | 0.58 | 2.33 | 0.58 | 1.02 |

Table 5.9: Effect of PADC on four identical prefetch-unfriendly applications

#### 5.5.3.6 Overall Performance

Figure 5.17 shows the average system performance and bus traffic for the 32 workloads run on the 4-core system. PADC provides the best performance and lowest bandwidth consumption compared to all previous prefetch handling policies. It improves weighted speedup and harmonic mean of speedups by 8.2% and 4.1% respectively compared to the demand-first policy and reduces bus traffic by 10.1% over demand-first (the best-performing rigid policy).




(a) System performance (b) Average traffic

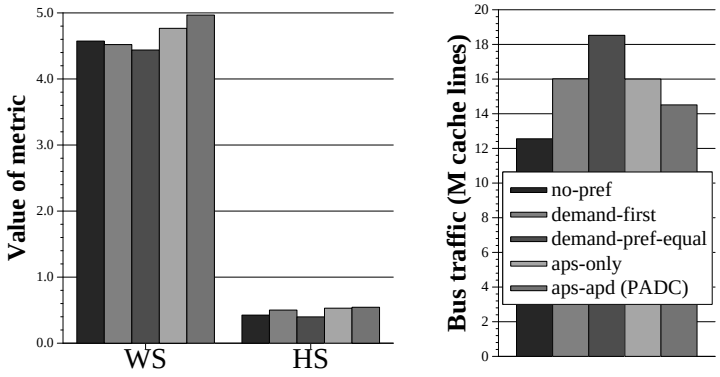
Figure 5.17: Performance of PADC on 4-core system

We found that PADC outperforms both the demand-first and demand-prefetch-equal policies for all but one workload we examined. The worst performing workload is the combination of *vpr, gamess, dealII*, and *calculix*. PADC's WS degradation is only 1.2% compared to the demand-first policy. These applications are either insensitive to prefetching (class 0) or not memory intensive (*vpr*).

### 5.5.4 8-Core Results

Figure 5.18 shows average performance and bus traffic over the 21 workloads we simulated on the 8-core system. Note that the rigid prefetch scheduling policies actually cause stream prefetching to degrade performance in the 8-core system. The demand-first policy reduces performance by 1.2% and the demand-prefetch-equal policy by 3.0% compared to no prefetching. DRAM bandwidth becomes a lot more valuable with the increased number of cores because the cores put

more pressure on the memory system. At any given time there is a much larger number of demand and useful/useless prefetch requests in the DRAM request buffer. As a result, it becomes more likely that 1) a useless prefetch delays a demand or useful prefetch (if demand-prefetch-equal policy is used), and 2) DRAM throughput degrades if a demand request causes significant reduction in the row-buffer locality of prefetch requests (if demand-first policy is used). Hence, performance degrades with a rigid scheduling policy.




(a) System performance (b) Average traffic

Figure 5.18: Performance of PADC on 8-core system

For the very same reasons, PADC becomes more effective when the number of cores increases. As resource contention becomes higher, the performance benefit of intelligent prioritization and dropping of useless prefetch requests increases. Our PADC improves overall system performance (WS) by 9.9% and also reduces memory bandwidth consumption by 9.4% compared to demand-first on the 8-core system. We conclude that the benefits of PADC will continue to increase as off-chip memory bandwidth becomes a larger performance bottleneck in future systems with many cores.

### 5.5.5 Optimizing PADC for Fairness Improvement in CMP Systems: Incorporating Request Ranking

PADC can be better tuned and optimized for the requirements of CMP systems. One major issue in designing memory controllers for CMP systems is the

need to ensure fair access to memory by different cores [58, 53, 54]. So far we have considered PADC only as a way to improve overall system performance. However, to be more effective in CMP systems, PADC can be augmented with a mechanism that provides fairness to different cores' requests. To achieve this purpose, this section describes a new scheduling algorithm that incorporates a request ranking scheme into our Adaptive Prefetch Scheduling (APS) mechanism.

Recall that APS prioritizes urgent requests (demand requests from cores whose prefetch accuracy is low) over others to mitigate performance degradation and unfairness for prefetch-unfriendly applications. However, APS follows the FCFS policy if all other priorities (i.e., criticality, row-hit, urgency) are the same. This FCFS rule can degrade fairness and system performance by prioritizing requests of memory intensive applications over those of memory non-intensive applications as was shown in previous work [58, 53, 54]. This happens because delaying the requests of memory non-intensive applications results in a lower individual speedup (or a higher slowdown) for those applications than it would for memory intensive applications which already suffer from long DRAM service time. Therefore, PADC (and APS) itself cannot completely solve the unfairness problem. This is especially true in cases where all of the applications behave the same in terms of prefetch friendliness (either all are prefetch-friendly or all are prefetch-unfriendly). In such cases, PADC will likely degenerate into the FCFS policy frequently (since the criticality, row-hit, and urgency priorities would be equal) resulting in high unfairness and performance degradation. For example, in case study II discussed in Section 5.5.3.2, all the applications are prefetch-unfriendly. Therefore, PADC prioritizes demands over prefetches most of the time. PADC mitigates performance degradation by prioritizing demand requests and dropping useless prefetches. However, *art* is very memory intensive and continuously generates many demand requests. These demand requests significantly interfere with other applications' demand requests resulting in high slowdowns for the other applications. However, *art* experiences the least slowdown thereby creating unfairness in the system as shown in Figure 5.13.

To take into account fairness in PADC, we incorporate the concept of ranking, as employed by Mutlu and Moscibroda [54]. Our ranking scheme is based on the *shortest job first* principle [70] which can better mitigate the unfairness problem and performance degradation caused by the FCFS rule. For each application, the DRAM controller keeps track of the total number of critical (demand and useful prefetch) requests in the DRAM request buffer. Applications with fewer outstanding critical requests are given a higher rank. The insight is that if an application that has fewer critical requests is delayed, the impact of that delay on that application's slowdown is much higher than the impact of delaying an application with a large number of critical requests. In other words, it is more unfair to delay an application that has a small number of useful requests (i.e., a "shorter" application/job) than delaying an application that has a large number of useful requests (i.e., a "longer" application/job). To achieve this while still being prefetch-aware, the DRAM controller schedules memory requests based on the modified rule shown in Rule 2. A highly-ranked request is scheduled by the DRAM controller when all requests in the DRAM request buffer have the same priority for criticality, row-hit, and urgency.

**Rule 2** Adaptive prefetch scheduling with ranking

1. **Critical request (C)**: Critical requests are prioritized over all other requests.
2. **Row-hit request (RH)**: Row-hit requests are prioritized over row-conflict requests.
3. **Urgent request (U)**: Demand requests generated by cores with low prefetch accuracy are prioritized over other requests.
4. **Highest rank request (RANK)**: Critical requests from a higher-ranked core are prioritized over critical requests from a lower-ranked core. Critical requests from cores that have fewer outstanding critical requests are ranked higher.
5. **Oldest request (FCFS)**: Older requests are prioritized over younger requests.

To implement ranking the priority field for each memory request is augmented as shown in Figure 5.19. A counter per core is required to keep track of the total number of critical requests in the DRAM request buffer. When the estimated prefetch accuracy of a core is greater than $promotion\_threshold$, the total number of outstanding demand and prefetch requests (critical requests) for that core is counted. When the accuracy is less than the threshold, the counter stores only the

number of outstanding demand requests. Cores are ranked according to the total number of critical requests they have in the DRAM request buffer: a core that has a larger number of critical requests is ranked lower. The RANK field of a request is the same as the rank value of the core determined in this manner. As such, the critical requests of a core with a lower value in its counter are prioritized. This process is done every DRAM bus cycle in our implementation. Alternatively, determination of the ranking can be done periodically since it does not need to be highly accurate and is not on the critical path.

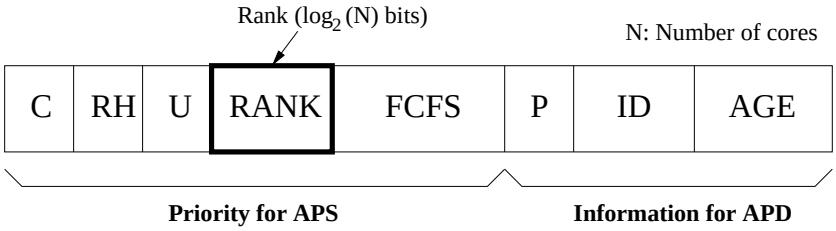
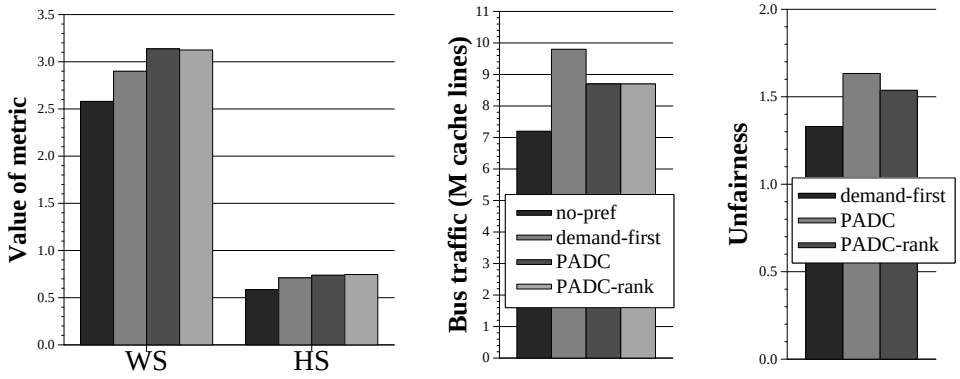



Figure 5.19: DRAM request fields for PADC with ranking

Note that in this study we do not rank non-critical requests (i.e., prefetch requests from cores whose prefetch accuracy is low). The RANK field of these requests is automatically set to 0 (the lowest rank value). We evaluated a mechanism that also ranks non-critical requests based on estimated prefetch accuracy and found that this mechanism does not perform better than the mechanism that ranks only critical requests.

Figure 5.20 shows the average system performance, bus traffic, and unfairness when we incorporate the ranking mechanism into PADC for the 32 4-core workloads. On average, the ranking mechanism slightly degrades weighted speedup (by 0.4%) and slightly improves harmonic mean of speedups (by 0.9%) and keeps bandwidth consumption about the same compared to the original PADC. Unfairness is improved from 1.63 to 1.53. The performance improvement is not significant because the contention in the memory system is not very high in the 4-core system. Nonetheless, the ranking scheme improves all the system performance and unfairness metrics for most workloads with memory intensive benchmarks. For the

workload in case study II, the ranking scheme improves WS, HS, and UF by 7.5%, 10.3%, and 15.1% compared to PADC without ranking.




(a) Performance (b) Average traffic (c) Unfairness

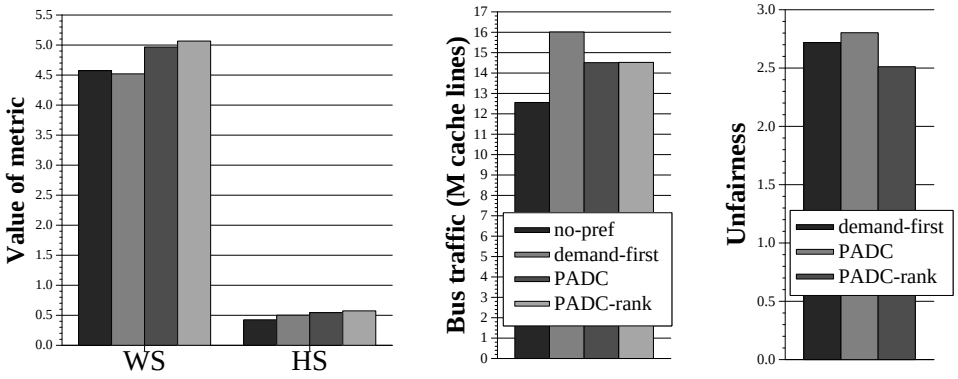
Figure 5.20: Optimized PADC with ranking on 4-core system

We also evaluate the optimized PADC scheme with ranking on the 8-core system which places significantly more pressure on the DRAM system. As shown in Figure 5.21, the ranking mechanism improves WS and HS by 2.0% and 5.4% respectively and reduces unfairness by 10.4% compared to PADC without ranking. The effectiveness of the ranking scheme is much higher in the 8-core system than the 4-core system since it is more critical to schedule memory requests fairly in many-core bandwidth-limited systems. Improving fairness reduces starvation of some cores resulting in improved utilization of the cores in the system, which in turn results in improved system performance. Since starvation is more likely when the memory system is shared between eight cores rather than four, the performance improvement obtained with the ranking scheme is higher in the 8-core system.

We conclude that augmenting PADC with an intelligent fairness mechanism improves both unfairness and system performance.

### 5.5.6 Effect on Multiple DRAM Controllers

We also evaluate the performance impact of PADC when two DRAM controllers are employed in the 4 and 8-core systems. Each memory controller works



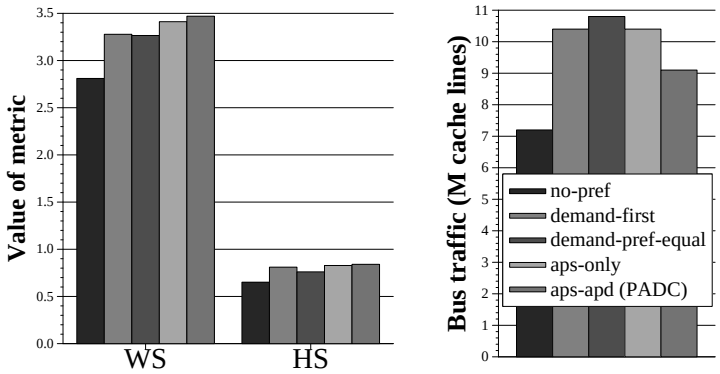

(a) Performance (b) Average traffic (c) Unfairness

Figure 5.21: Optimized PADC using ranking mechanism on 8-core system

independently through a dedicated channel (address, control, and data buses) doubling the peak memory bandwidth. Because there is more bandwidth available in the system, contention between prefetch and demand requests is significantly reduced. Therefore, the baseline system performance is significantly improved compared to the single controller. Adding one more DRAM controller improves weighted speedup by 16.9% and 30.9% compared to the single controller for 4 and 8-core systems respectively.

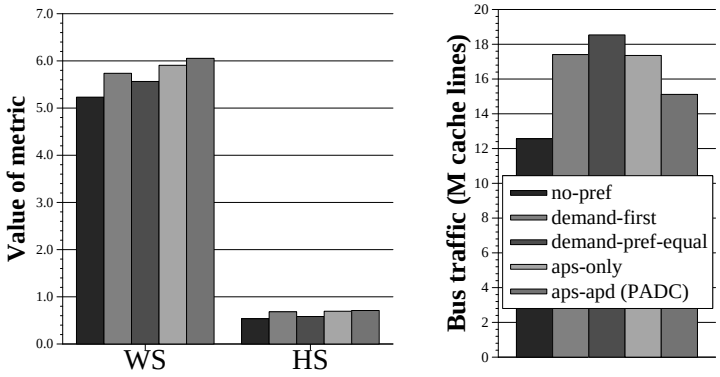
Figures 5.22 and 5.23 show the average performance and bus traffic for 4 and 8-core systems with two memory controllers. Note that for the 8-core system, unlike the single memory controller configuration shown in Figure 5.18(a) where adding a prefetcher actually degrades performance, performance increases when adding a prefetcher even for the rigid scheduling policies because of the increased memory bandwidth.

PADC is still very effective with two memory controllers and improves weighted speedup by 5.9% and 5.5% and also reduces bandwidth consumption by 12.9% and 13.2% compared to the demand-first policy for 4 and 8-core systems respectively. Therefore, we conclude that PADC still performs effectively on a multi-core processor with very high DRAM bandwidth.




(a) System performance (b) Average traffic

Figure 5.22: Performance of PADC on 4-core system with two DRAM controllers




(a) System performance (b) Average traffic

Figure 5.23: Performance of PADC on 8-core system with two DRAM controllers

### 5.5.7 Effect with Different DRAM Row Buffer Sizes

As motivated in Section 5.1, PADC takes advantage of and relies on the row buffer locality of demand and prefetch requests generated at runtime. To determine the sensitivity of PADC to row buffer size, we varied the size of the row buffer from 2KB to 128KB for the 32 workloads run on the 4-core system. Figure 5.24 shows the WS improvements of PADC and APS compared to no prefetching, demand-first, and demand-prefetch-equal.

PADC consistently outperforms no prefetching, demand-first, and demand-prefetch-equal with various row buffer sizes. Note that the demand-first policy starts degrading performance compared to no prefetching as the row buffer becomes very

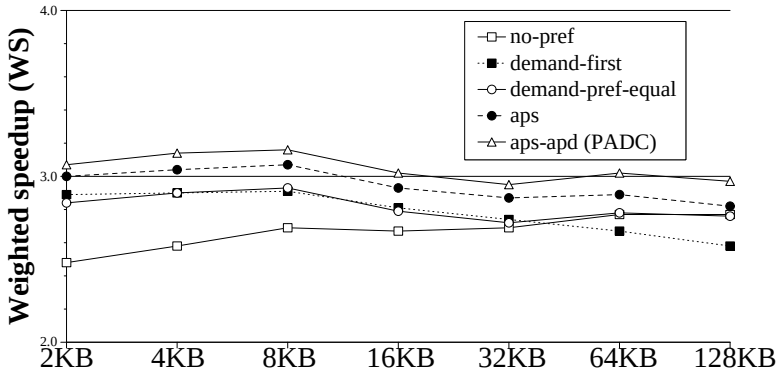



Figure 5.24: Effect of PADC with various DRAM row buffer sizes on 4-core system

large (64KB and 128KB). This is because preserving row buffer locality for useful requests is more critical when the row buffer size is large (especially when the stream prefetcher is enabled). No prefetching with larger row buffer sizes exploits row buffer locality more (higher row-hit rate) than smaller row buffer sizes. However, with demand-first, the negative performance impact of frequent re-activations of DRAM rows for demand and prefetch requests becomes significantly worse at larger row buffer sizes. Therefore, the demand-first policy experiences a higher memory service time on average than no prefetching with large row buffer sizes.

Similarly, the demand-prefetch-equal policy does not improve performance compared to no prefetching for 64KB and 128KB row buffer sizes since it does not take into account the usefulness of prefetches. With a large row buffer, useless prefetches have higher row buffer locality because many of them hit in the row buffer due to the streaming nature of the prefetcher. As a result, demand-prefetch-equal significantly delays the service of demand requests at large row buffer sizes by servicing more useless row-hit prefetches first.

In contrast to these two rigid scheduling policies, PADC tries to service only useful row-hit memory requests first, thereby significantly improving performance even for large row buffer sizes (8.8% and 7.3% compared to no prefetching for 64KB and 128KB row buffers). Therefore, PADC can make a prefetcher viable and effective even when a large row buffer size is used because it takes advantage of the increased row buffer locality opportunity provided by a larger row buffer *only* for

useful requests instead of wasting the increased amount of bandwidth enabled by a larger row buffer on useless prefetch requests.

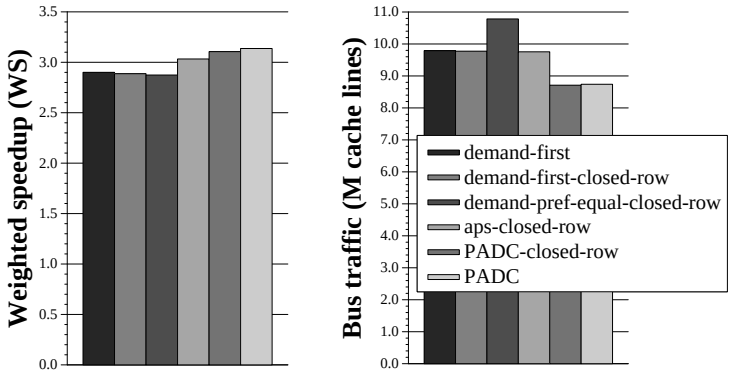
### 5.5.8 Effect with a Closed-Row DRAM Row Buffer Policy

So far we have assumed that the DRAM controller employs the open-row policy (i.e., it keeps the accessed row open in the row buffer after the access even if there are no more outstanding requests requiring the row). In this section, we evaluate the effectiveness of PADC with a closed-row policy. The closed-row policy closes (by issuing a precharge command) the currently-opened row when all row-hit requests in the DRAM request buffer have been serviced by the DRAM controller. This policy can hide effective precharge time by 1) overlapping the precharge latency with the row-access latency [22, 49] and 2) issuing the precharge command (closing a row buffer) earlier than the open-row policy. Therefore, if no more requests to the same row arrive at the DRAM request buffer after a row buffer is closed by a precharge command, the closed-row policy can outperform the open-row policy. This is because with the closed-row policy, the later requests do not need a precharge before activating the different row. However, if a request to the same row arrives at the DRAM request buffer soon after the row is closed, this policy has to pay a penalty (the sum of the non-overlapped precharge latency and the activation latency) which would not have been required for the open-row policy. Consequently, for applications that have high row buffer locality (i.e., applications that generate bursty row-hit requests) such as streaming/striding applications, the open-row policy outperforms the closed-row policy by reducing re-activations of the same rows that will be needed again in the near future.

Since the closed-row policy still services row-hit requests first until no more requests to the same row remain in the DRAM request buffer, it can increase DRAM throughput within the scope of the requests that are outstanding in the DRAM request buffer. Therefore, when a prefetcher is enabled with the closed-row policy, the same problem exists as for the open-row policy: none of the rigid prefetch scheduling policies can achieve the best performance for all applications since they are not

aware of prefetch usefulness. Therefore PADC can still work effectively with the closed-row policy.

Figure 5.25 shows the performance and bus traffic when PADC is used with the closed-row policy for the 32 4-core workloads. The closed-row policy with demand-first scheduling slightly degrades performance by 0.5% compared to the open-row policy with demand-first scheduling. This is because there is a large number of streaming/striding (and prefetch-friendly) applications in the SPEC 2000/2006 benchmarks whose performance can be significantly improved with the open-row policy. The performance improvement of the open-row policy is not very significant because there is also a large number of applications that work well with the closed-row policy as they do not have high row buffer locality.




(a) System performance (b) Average traffic

Figure 5.25: Effect of PADC on closed-row scheduling policy

The results show that PADC is still effective with the closed-row policy since it still effectively exploits row buffer locality (within the scope of the requests outstanding in the DRAM request buffer) and reduces the negative effects of useless prefetch requests. PADC improves weighted speedup by 7.6% and reduces bandwidth consumption by 10.9% compared to demand-first scheduling with the closed-row policy. Note that PADC with the open-row policy slightly outperforms PADC with the closed-row by 1.1% for weighted speedup. Overall, we conclude that PADC is suitable for different row buffer management policies but it is more

effective with the open-row policy due to the existence of a larger number of benchmarks with high row buffer locality.

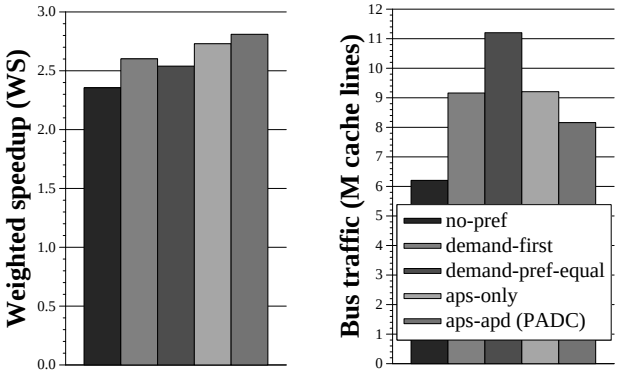
### 5.5.9 Effect with a Shared Last-Level Cache

Throughout this chapter, we evaluate our mechanism on CMP systems with private on-chip last-level caches rather than a shared cache where all cores share a large on-chip last-level cache. This allowed us to easily show and analyze the effect of PADC in the shared DRAM system by isolating the effect of contention in the DRAM system from the effect of interference in shared caches. However, many commercial processors already employ shared last-level caches in their CMP designs [77, 80]. In this section, we evaluate the performance of PADC in on-chip shared last-level caches on the 4 and 8-core systems to show the effectiveness of PADC in systems with a shared last-level cache.

For this experiment, we use a shared last-level cache whose size is equivalent to the sum of all the private last-level cache sizes in our baseline system. We scaled the associativity of the shared cache with the number of cores on the chip since as the number of cores increases, the contention for a cache set increases. Therefore the 4-core system employs a 2MB, 16 way set-associative cache and the 8-core system has a 4MB, 32 way set-associative cache. We selected 32 way set-associativity for the 8-core system in order to show how the mechanism works with a very aggressive last-level cache. If the associativity is less, our mechanism performs even better. We also assume that each core employs its own independent stream prefetcher that monitors the core's demand accesses and sends prefetched data into the shared last-level cache. Note that our mechanism can also work for a single prefetcher which monitors all cores' accesses and generates prefetches for all cores [77, 80] by simply associating core ID bits with each prefetch request, signifying which core generated the prefetch request. This way, PADC can update the appropriate per-core counters to estimate prefetch accuracy of each core.

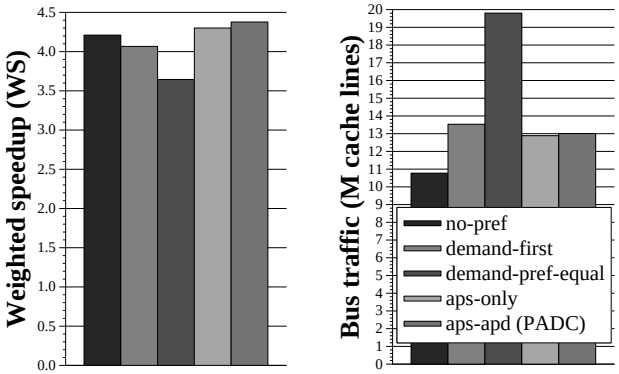
Figures 5.26 and 5.27 show weighted speedup and average bus traffic on the

4 and 8-core systems with shared last-level caches. PADC outperforms demand-first by 8.0% and 7.6% on the 4 and 8-core systems respectively. We conclude that PADC works efficiently for shared last-level caches as well.




(a) System performance (b) Average traffic

Figure 5.26: Effect of PADC on shared last-level cache on 4-core system




(a) System performance (b) Average traffic

Figure 5.27: Effect of PADC on shared last-level cache on 8-core system

Note that the demand-prefetch-equal policy does not work well on either of the shared cache systems (degrading WS by 2.4% and 10.4% compared to demand-first for 4 and 8-core systems). This is because the contention in the shared cache among the requests from different cores significantly increases compared to that of a private cache system. With private caches, useless prefetches from one core can only replace useful lines of that same core. However, with a shared cache, useless prefetches from one core can also replace the useful lines of all the other

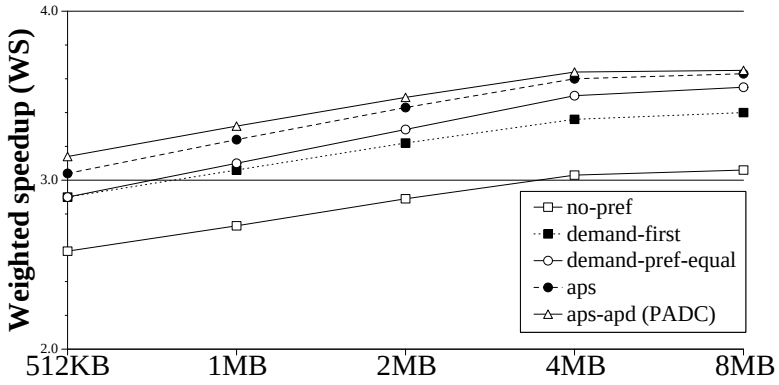
cores. These replaced lines must be brought back into the cache again from DRAM when they are needed. Therefore, the total bandwidth consumption significantly increases. This cache contention among cores becomes especially worse with demand-prefetch-equal for prefetch-unfriendly applications. This is because the demand-prefetch-equal policy results in high cache pollution since it blindly prefers to increase DRAM throughput without considering the usefulness of prefetches. The demand-prefetch-equal policy increases bus traffic by 22.3% and 46.3% compared to demand-first for the 4 and 8-core systems as shown in Figures 5.26(b) and 5.27(b). In contrast, PADC delays the service of useless prefetches and also drops them thereby mitigating contention in both the shared cache and the shared DRAM system.

### 5.5.10 Effect with Different Last-Level Cache Sizes

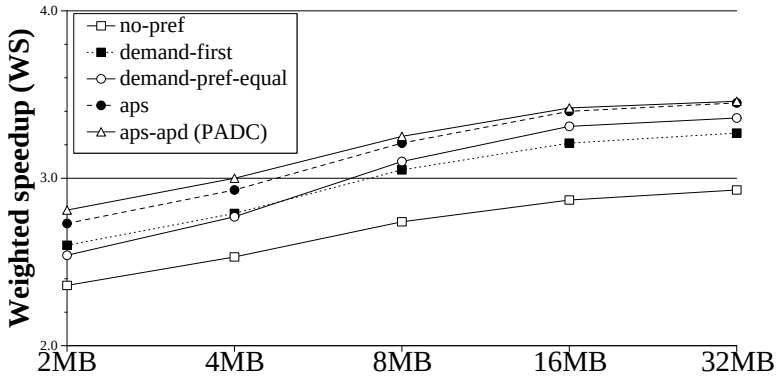
PADC aims to maximize DRAM throughput for useful memory (demand and useful prefetch) requests and to delay and drop useless memory requests. One might think that a prefetch/demand management technique such as PADC would not be needed for larger last-level caches since a larger cache can reduce cache misses (i.e., memory requests). However, a prefetcher can still generate a significant number of useful prefetch requests for some applications or program phases by correctly predicting demand access patterns which cannot be stored even in large caches due to the large working set size or streaming nature of the program. In addition, the prefetcher can issue a significant number of useless prefetches for other applications or program phases. For these reasons, the interference between demands and prefetches still exists in systems with large caches. Therefore, we hypothesize PADC is likely to be effective in systems with large last-level caches.

To test this hypothesis, we evaluate the effectiveness of PADC for various last-level cache sizes. We vary the private last-level cache size from 512KB to 8MB per core and the shared cache size from 2MB to 32MB (other cache parameters are as described in Section 5.5.9) on our 4-core CMP system. Figure 5.28 shows the system performance (weighted speedup) for the 32 4-core workloads.

As expected, baseline system performance improves with larger cache sizes. However, the stream prefetcher still effectively improves performance compared to no prefetching with either the demand-first or the demand-prefetch-equal policy. In addition, PADC consistently and significantly improves performance compared to both demand-first and demand-prefetch-equal policies for both private and shared caches, regardless of cache size. This is mainly because even with large caches there is still a significant number of both useful and useless prefetches generated. Therefore, the interference between prefetch and demand requests still needs to be intelligently controlled.




(a) Private last-level caches per core




(b) Shared last-level cache

Figure 5.28: Effect of PADC on various cache sizes on 4-core system

There are two other notable observations from Figure 5.28: 1) the demand-pref-equal policy starts outperforming the demand-first policy for private caches greater than 1MB (8MB for the shared cache), and 2) the performance of APS

(without APD) becomes closer to that of PADC (APS and APD together) as the cache size becomes larger. These trends hold for both private and shared last-level caches.

Both observations can be explained by two reasons. First, a larger cache reduces irregular (or hard-to-prefetch) conflict cache misses due to the increased cache capacity. This makes the prefetcher more accurate because it reduces the allocations of stream entries for hard-to-prefetch access patterns (note that only a demand cache miss allocates a stream prefetch entry). Second, a larger cache can tolerate some degree of cache pollution. Due to the increased cache capacity, the probability of replacing a demand or useful prefetch line with a useless prefetch in the cache is reduced.

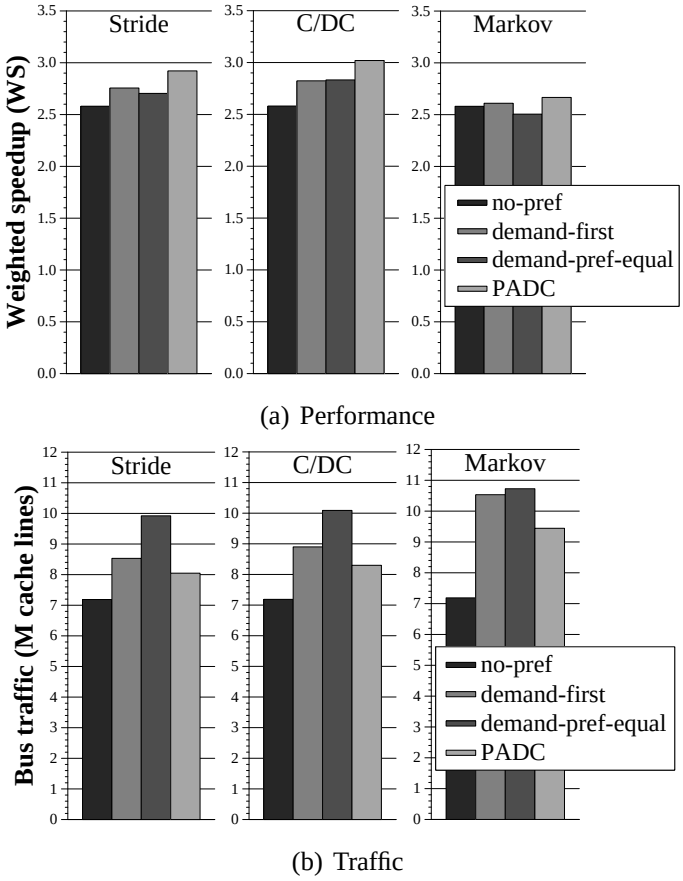
For these reasons, the effect of deprioritizing or dropping likely-useless prefetches becomes less significant with a larger cache. As a result, as cache size increases, techniques that prioritize demands (e.g., demand-first) and drop prefetches (APD) start becoming less effective. However, the interference between prefetch and demand requests is not completely eliminated since some applications still suffer from useless prefetches. PADC (and APS) is effective in reducing this interference in systems with large caches and therefore still performs significantly better than the rigid scheduling policies.

Note that PADC is cost-effective for both private and shared last-level caches. For instance, PADC with a 512KB private last-level cache per core performs almost the same as demand-first with a 2MB private last-level cache per core as shown in Figure 5.28(a). Thus, PADC (which requires only 4.25KB storage) achieves the equivalent performance improvement that an additional 6MB (1.5MB $\times$ 4 cores) of cache storage would provide in the 4-core system.

### 5.5.11 Effect on Other Prefetching Mechanisms

To show that the benefits of PADC are orthogonal to the prefetching algorithm employed, we briefly evaluate the effect of our PADC on different types of

prefetchers: PC-based stride [1], CZone Delta Correlation (C/DC) [59], and the Markov prefetcher [26]. Figure 5.29 shows the performance and bus traffic results averaged over all 32 workloads run on the 4-core system with the three different prefetchers. PADC consistently improves performance and reduces bandwidth consumption compared to the demand-first or demand-prefetch-equal policies with all three prefetchers.




(a) Performance

(b) Traffic

Figure 5.29: PADC on stride, C/DC, and Markov prefetchers

The PC-based stride and C/DC prefetchers successfully capture a significant number of memory access patterns as the stream prefetcher does thereby increasing the potential for exploiting row buffer locality. In addition, these prefetchers also generate many useless prefetches for certain applications. Therefore, PADC significantly improves performance and bandwidth efficiency with these prefetchers by increasing DRAM throughput for useful requests and reducing the negative impact

of useless prefetches.

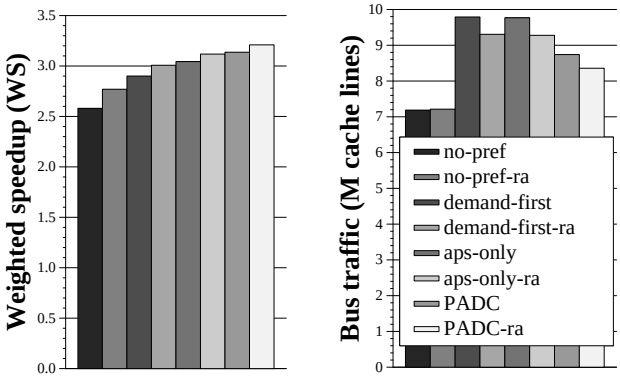
The performance improvement of PADC on the Markov prefetcher is the least. This is because the Markov prefetcher, which exploits temporal as opposed to spatial correlation, does not work as well as the other prefetchers for the SPEC benchmarks. It generates many useless prefetches which lead to significant waste and interference in DRAM bandwidth, cache space, and memory buffer resources. Furthermore, it does not generate many useful prefetches for the SPEC benchmarks and therefore its maximum potential for performance improvement is low. As such, the Markov prefetcher significantly increases bandwidth consumption and results in little performance improvement compared to no prefetching as shown in Figure 5.29. PADC improves the performance of the Markov prefetcher (mainly due to APD) by removing a large number of useless prefetches while keeping the small number of useful prefetches. PADC improves WS by 2.2% and reduces bandwidth consumption by 10.3% (mainly due to APD) compared to the demand-first policy. We conclude that PADC is effective with a wide variety of prefetching mechanisms.

### 5.5.12 Effect on a Runahead Execution Processor

Runahead execution [8, 55] is a promising technique that prefetches useful data by executing future instructions that are independent of a long latency (runahead-causing) load instruction during the stall time of the load instruction. Because it is based on the execution of actual instructions, runahead execution can prefetch irregular data access patterns as well as regular ones. Usually, runahead execution complements hardware prefetching and results in high performance. In this section, we analyze the effect of PADC on a runahead processor. We implemented runahead capability in our baseline system by augmenting invalid bits in the register files for each core. Since memory requests during runahead modes are very accurate most of the time [55], we treat runahead requests the same as demand requests in DRAM scheduling.

Figure 5.30 shows the effect of PADC on a runahead processor for the 32

workloads on the 4-core CMP system. Each runahead processor has exactly the same parameters as our baseline processor, but it also uses a 512-byte runahead cache to support store-load forwarding during runahead execution. Adding runahead execution on top of the baseline demand-first policy improves system performance by 3.7% and also reduces bandwidth consumption by 5.0%. This is because we use a prefetcher update policy that trains existing stream prefetch entries but does not allocate a new stream prefetch entry on a cache miss during runahead execution (*only-train*). Previous research [52] shows that this policy is best performing and most efficient. Runahead execution with the only-train policy can make prefetching more accurate and efficient by capturing irregular cache misses during runahead execution. These irregular misses train existing stream prefetch entries but new, more speculative, stream prefetch entries will not be created during runahead mode. This not only prevents the prefetcher from generating useless prefetches due to falsely created streams but also improves the accuracy and timeliness of the stream prefetcher since existing streams continue to be trained during runahead mode.




(a) Performance (b) Average traffic

Figure 5.30: Effect of PADC on runahead execution

Figure 5.30 shows that PADC still effectively improves performance by 6.7% and reduces bandwidth consumption by 10.2% compared to a runahead CMP processor with the stream prefetcher and the demand-first policy. We conclude that PADC is effective at improving performance and bandwidth-efficiency for an

aggressive runahead CMP by successfully reducing the interference between demand/runahead and prefetch requests in the DRAM controller.

### 5.5.13 Comparison with Dynamic Data Prefetch Filtering and Feedback Directed Prefetching

Dynamic Data Prefetch Filtering (DDPF) [91] tries to eliminate useless prefetches based on whether or not the prefetches were useful in the past. It records either the past usefulness of the prefetched address (or the PC of the instruction which triggered the prefetch) in a table similar to how a two-level branch predictor stores history information [82]. When a prefetch request is created, the history table is consulted and the previous usefulness information is used to determine whether or not to send out the prefetch request. Feedback Directed Prefetching (FDP) [73] adaptively adjusts the aggressiveness of the prefetcher in order to reduce its negative effects.

Recall that PADC has two components: APS (Adaptive Prefetch Scheduling) and APD (Adaptive Prefetch Dropping). Both DDPF and FDP are orthogonal to APS because they do not deal with the scheduling of prefetches with respect to demands. As such, they can be employed together with APS to maximize the benefits of prefetching. On the other hand, the benefits of DDPF, FDP, and APD overlap. DDPF filters out useless prefetches before they are sent to the memory system. FDP eliminates useless prefetches by reducing the aggressiveness of the prefetcher thereby reducing the likelihood that useless prefetch requests are generated. In contrast, APD eliminates useless prefetches by dropping them *after* they are generated. As a result, we find (based on our experimental analyses) that APD has the following advantages over DDPF and FDP:

1. Both DDPF and FDP eliminate not only useless prefetches but also a significant fraction of useful prefetches. DDPF removes many useful prefetches by falsely predicting many useful prefetches to be useless. This is due to the aliasing problem caused by sharing the limited size of the history table among many addresses. FDP can eliminate useful prefetches when it reduces the aggressiveness of

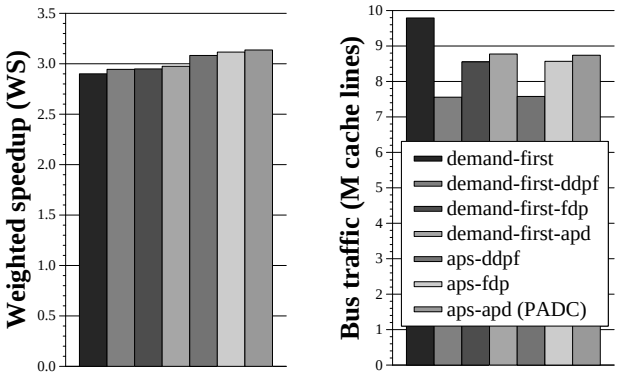
the prefetcher. In addition, we found that FDP can be very slow in increasing the aggressiveness of the prefetcher when a new phase starts execution. In such cases, FDP cannot issue useful prefetches whereas APD would have issued them because it always keeps the prefetcher aggressive.

2. The hardware cost of DDPF for an last-level cache is expensive since each cache line and MSHR must carry several bits for indexing the prefetch history table (PHT) to update the table appropriately. For example, for a PC-based gshare DDPF with a 4K-entry PHT, 24 bits (12-bit branch history and 12-bit load PC bits) per cache line are needed in addition to the prefetch bit per cache line. For the 4-core system we use, this index information alone accounts for 96KB of storage. In contrast, APD does not require significant hardware cost as we have shown in Section 5.4.

3. FDP requires the tuning of multiple threshold values to throttle the aggressiveness of the prefetcher which is a non-trivial optimization problem. APD allows the baseline prefetcher to *always* be very aggressive because it can eliminate useless prefetches after they are generated. As such, there is no need to tune multiple different threshold values in APD because the aggressiveness of the prefetcher never changes.

To evaluate the performance of these mechanisms, we implemented DDPF (PC-based gshare DDPF for last-level cache prefetch filtering [91]) and FDP in our CMP system. All the relevant parameters (FDP: prefetch accuracy of 90%, 40%, lateness of 1%, and pollution of 0.5% thresholds and pollution filter size of 4Kbits; DDPF: filtering threshold of 3, table size of 4K entry 2-bit counters) for DDPF and FDP were tuned for the best performance with the stream prefetcher in our CMP system. Figure 5.31 shows the performance and bus traffic of different combinations of DDPF, FDP, and PADC averaged across the 32 workloads run on the 4-core system. From left to right, the seven bars show: 1) baseline stream prefetching with the rigid demand-first policy, 2) DDPF with demand-first policy, 3) FDP with demand-first policy, 4) APD with demand-first policy, 5) DDPF combined with

APS, 6) FDP combined with APS, and 7) APD combined with APS (i.e., PADC). When used with the demand-first policy, DDPF and FDP improve performance by 1.5% and 1.7% respectively while reducing bus traffic by 22.8% and 12.6%. In contrast, APD improves performance by 2.6% while reducing bus traffic by 10.4%. DDPF and FDP eliminate more useless prefetches than APD resulting in less bus traffic. However, DDPF and FDP eliminate many useful prefetches as well. Therefore, their performance improvement is not as high as APD.




(a) System performance (b) Average traffic

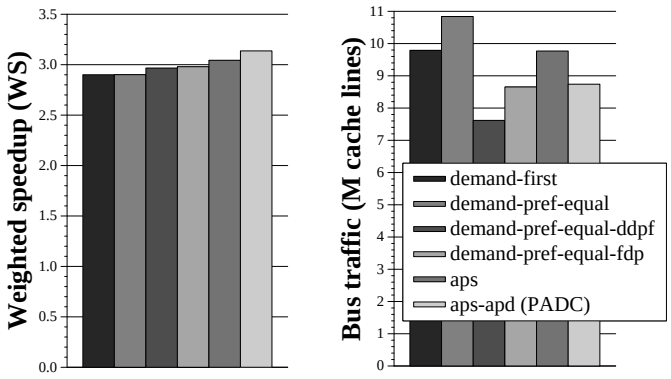
Figure 5.31: Comparison of PADC with DDPF and FDP with demand-first

Our adaptive scheduling policy and DDPF/FDP are complementary and improve performance significantly when combined together. When used together with Adaptive Prefetch Scheduling, DDPF and FDP improve performance by 6.3% and 7.4% respectively. Finally, the results show that PADC outperforms the combination of DDPF/FDP and APS which illustrates that Adaptive Prefetch Dropping is better suited to eliminate the negative performance effects of prefetching than DDPF and FDP. We conclude that 1) our adaptive scheduling technique complements DDPF and FDP whereas our APD technique outperforms DDPF and FDP, and 2) DDPF and FDP reduce bandwidth consumption more than APD but they do so at the expense of performance.

If a prefetch filtering mechanism is able to eliminate all useless prefetches while keeping all useful prefetches, the demand-prefetch-equal policy would be best performing. That is to say, we do not need an adaptive memory scheduling

policy since all prefetches sent to the memory system would be useful. However, it is not trivial to design such a perfect prefetch filtering mechanism. As discussed above, DDPF and FDP filter out not only useless prefetches but also a lot of useful prefetches. Therefore, combining those schemes with demand-prefetch-equal does not necessarily significantly improve performance since the benefits of useful prefetches are reduced.

Figure 5.32 shows performance and average traffic when DDPF and FDP are combined with demand-prefetch-equal. Since DDPF and FDP remove a significant number of useful prefetches, performance improvement is not very significant (only by 2.3% and 2.7% compared to demand-first). On the other hand, PADC significantly improves performance (by 8.2%) by keeping the benefits of useful prefetches as much as possible.




(a) System performance (b) Average traffic

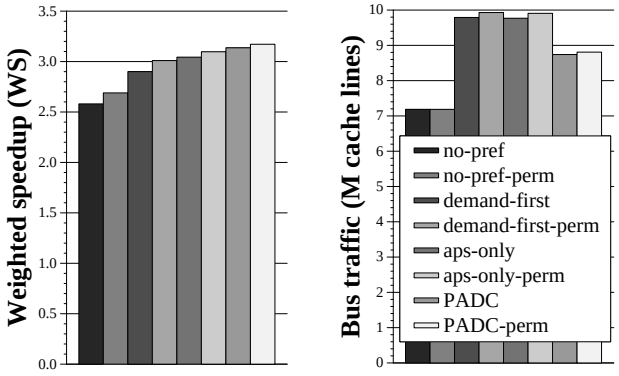
Figure 5.32: Comparison of PADC to DDPF and FDP with demand-prefetch-equal

### 5.5.14 Interaction with Permutation-Based Page Interleaving

Permutation-based page interleaving [87] aims to reduce row conflicts by randomly mapping the DRAM bank indexes of addresses so that they are more spread out across the multiple banks in the memory system. This technique significantly improves DRAM throughput by increasing utilization of multiple DRAM banks (exploiting bank-level parallelism). The increased utilization of the banks has the potential to reduce the interference between memory requests. However,

this technique cannot completely eliminate the interference between demand and prefetch requests in the presence of prefetching. Any rigid prefetch scheduling policy in conjunction with this technique will still have the same problem we describe in Section 5.1: none of the rigid prefetch scheduling policies can achieve the best performance for all applications since they are not aware of prefetch usefulness. Therefore, PADC is complementary to permutation-based page interleaving.

Figure 5.33 shows the performance impact of PADC for the 32 4-core workloads when a permutation-based interleaving scheme is applied. The permutation-based scheme improves system performance by 3.8% over our baseline with the demand-first policy. This is because the permutation scheme reduces row-conflicts by spreading out requests across multiple banks.




(a) Performance (b) Average traffic

Figure 5.33: Effect of PADC on permutation-based page interleaving

APS and PADC consistently work effectively combined with the permutation-based interleaving scheme by efficiently managing the interference between demands and prefetches based on usefulness of prefetches. APS and PADC improve system performance by 2.9% and 5.4% respectively compared to the demand-first policy with the permutation-based interleaving scheme. Also, PADC reduces bandwidth consumption by 11.3% due to adaptive prefetch dropping.

## 5.6 Summary

This chapter shows that existing DRAM controllers that employ rigid, non-adaptive prefetch scheduling and buffer management policies can limit performance since they do not take into account the usefulness of prefetch requests. To overcome this limitation, we propose a low hardware cost Prefetch-Aware DRAM Controller (PADC), which aims to 1) maximize the benefit of useful prefetches by adaptively prioritizing them, and 2) minimize the harm caused by useless prefetches by adaptively deprioritizing them and dropping them from the memory request buffers. To this end, PADC dynamically adapts its memory scheduling and buffer management policies based on prefetcher accuracy. We show that it is a general mechanism that is effective for a variety of systems and that it is orthogonal to previously proposed prefetching and prefetch filtering techniques.

# Chapter 6

# Prefetch Management for Increasing DRAM Bank-Level Parallelism (BLP)

This chapter studies how to manage prefetch and demand requests in on-chip request buffers to improve DRAM bank-level parallelism (BLP) in the presence of prefetching. We propose two techniques [39]. One is a prefetch issue policy that aims to maximize BLP for memory requests of the running application on each core. The other is a request issue policy which tries to minimize the destructive interference in the BLP of each application when multiple applications run together on a CMP system.

## 6.1 Prefetch Issue Policy to Increase BLP

### 6.1.1 Prefetching: Increasing Potential for DRAM BLP

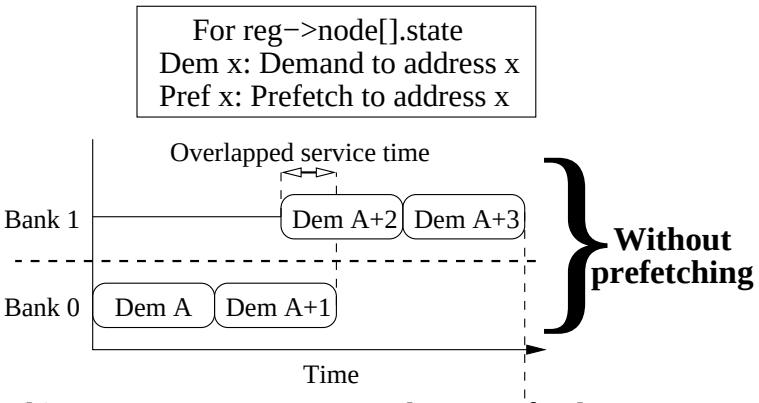
Hardware prefetchers can increase the potential for DRAM BLP because they generate multiple memory requests within a short period of time. With prefetching enabled, demand requests and potential future requests (useful prefetches) are both in the memory system at the same time. This increase in concurrent requests provides more potential to exploit DRAM BLP as shown in the following example.

Figure 6.1(a) shows a code example from *libquantum* where a significant number of useful prefetches are generated by the stream prefetcher we used in Chapter 5. With no prefetching, the demand accesses would be in the sequence: cache line addresses A, A+1, A+2, and A+3 (for reg->node[].state). When prefetching is employed, cache lines to A+1, A+2, and A+3 would be prefetched. We assume that the first two accesses (to cache line addresses A, and A+1) are mapped to the same DRAM bank and that the two subsequent accesses (to A+2, and A+3) are

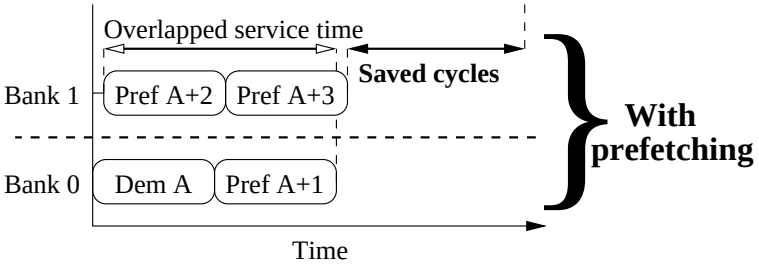
mapped to a different bank.

```
for(i=0; i<reg->size; i++)
{
  reg->node[i].state ^=
    ((MAX_UNSIGNED) 1 << target);
}
```

(a) Code example




(b) DRAM service time without prefetcher




(c) DRAM service time with prefetcher

Figure 6.1: How prefetching can increase DRAM BLP (*libquantum*)

Figure 6.1(b) shows the DRAM service time when the code is executed without prefetching. Due to the lookahead provided by the processor's instruction window, accesses to A+1 and A+2 are slightly overlapped. On the other hand, with the prefetcher enabled, if the prefetches reach the memory system (DRAM request buffers) quickly such that the DRAM controller can see all these requests, the DRAM service time of the prefetches significantly overlap as shown in Figure 6.1(c). Therefore, overall DRAM service time is significantly improved compared to no prefetching (shown as "Saved cycles" in the figure).

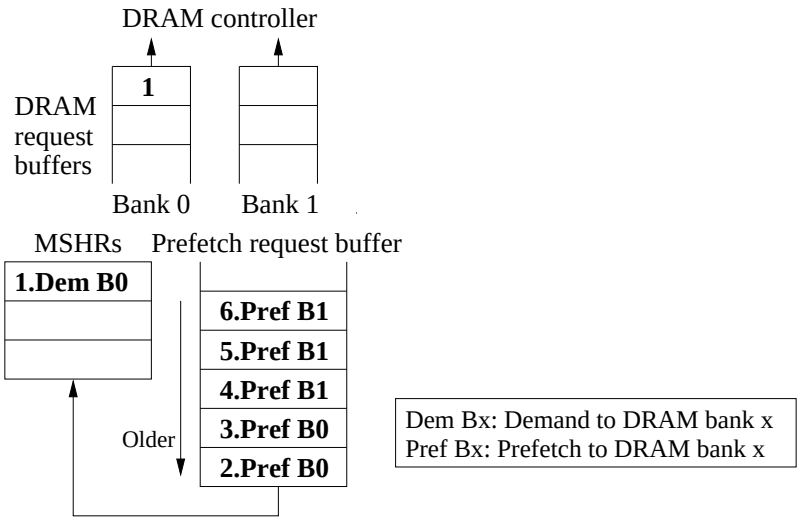
As shown in the example, a hardware prefetcher can increase the potential for improving DRAM bank-level parallelism. However, normally, this potential is NOT always fully exposed to the DRAM system.

### 6.1.2 What Can Limit Prefetching's Benefits?

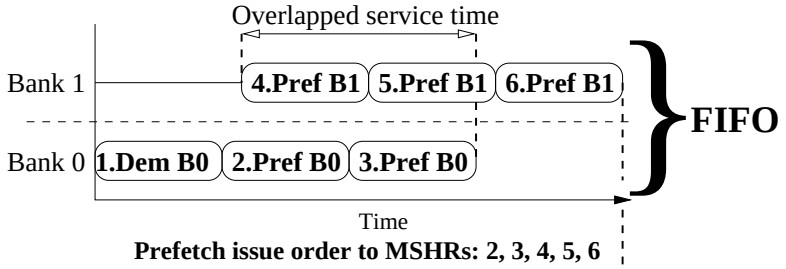
If an on-chip memory system design does not take DRAM BLP into account, it may limit the benefits of prefetching when the total number of outstanding requests allowed in the on-chip memory system is limited. This is true for Miss Status/Information Holding Registers (MSHRs) that keeps track of all outstanding cache misses in the system. All memory requests must first be allocated an MSHR entry before entering the DRAM request buffers where they are considered for DRAM scheduling. The request remains in the MSHR until serviced by DRAM. The MSHR structure is complex and therefore costly to increase in size [79] since it requires content-associative search. Therefore, the choice of which requests are placed into the resource-limited MSHRs and finally into DRAM request buffers significantly affects the amount of BLP exploited by the DRAM controller.

For example, the FIFO buffer (which we call the prefetch request buffer) in the Intel Core design [7] buffers prefetch requests until they can be sent to the memory system. This FIFO structure will always send the oldest prefetch request to the memory system provided that the memory system has room for an additional request. This design choice can limit the amount of DRAM BLP exploited when servicing the prefetch requests since the oldest prefetches in the buffer is always sent first regardless of whether or not it can be serviced in parallel with other requests. A more intelligent policy would consider DRAM BLP when sending prefetch requests to the memory system.

Figure 6.2 illustrates this problem. Figure 6.2(a) shows the initial state of the prefetch request buffer, MSHRs (three entries), and DRAM request buffers (three entries per DRAM bank). There is only one outstanding demand request (Request 1 in the figure). This request is mapped to Bank 0 and just about to be scheduled

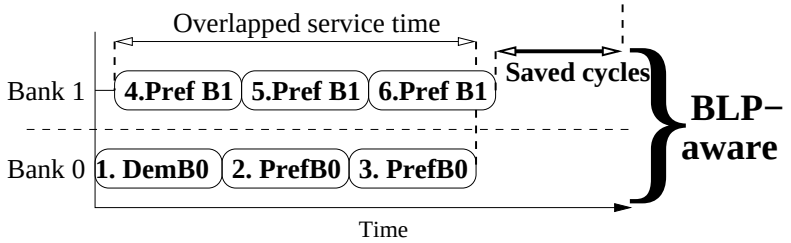



(a) Initial state of memory buffers




**Prefetch issue order to MSHRs: 2, 3, 4, 5, 6**

(b) DRAM service time for FIFO prefetch issue




**Prefetch issue order to MSHRs: 4, 2, 5, 3, 6**

(c) DRAM service time for DRAM BLP−aware prefetch issue

Figure 6.2: FIFO vs. DRAM BLP-aware prefetch issue policy

to access DRAM. There are five prefetches in the prefetch request buffer. The first two prefetches will access DRAM Bank 0 and the three subsequent prefetches will access DRAM Bank 1. For this example we assume that all the prefetches are useful and therefore will be required by the program soon.

Figure 6.2(b) shows the DRAM service timeline when prefetches are issued into MSHRs in a FIFO fashion. In this case, the demand request and the two prefetch requests to Bank 0 fill up the MSHRs and therefore the first prefetch to Bank 1 will not be issued until the demand request gets serviced by DRAM and its MSHR entry is freed. As a result, BLP is low.

A DRAM BLP-aware issue policy would send a prefetch to Bank 1 first, followed by a prefetch to Bank 0. In other words, we can alternately issue prefetches to Bank 1 and Bank 0. Using this issue policy, the service of prefetches to Bank 1 can start earlier and overlap with accesses to Bank 0 as shown in Figure 6.2(c). Therefore, BLP increases and overall DRAM service time improves (shown as "Saved cycles" in the figure).

This example provides two insights. First, *simply increasing the number of outstanding requests in the memory system does not necessarily mean that their latencies will overlap.* A BLP-unaware prefetch issue policy (to MSHRs) can severely limit the BLP exploited by the DRAM controller. Second, a simple prefetch issue policy that is aware of which bank a memory request will access can improve DRAM service time by prioritizing prefetches to different banks over prefetches to the same bank.

So far we assumed that all prefetches are useful. However, if prefetches are useless, the BLP-aware prefetch issue policy will not be helpful. It may increase DRAM throughput but only for useless requests. Useless prefetches should be not issued to the memory system regardless of whether it increases BLP or not.

### 6.1.3 Mechanism: BLP-Aware Prefetch Issue

We propose BLP-Aware Prefetch Issue (BAPI) to maximize BLP of useful memory requests exposed to the DRAM controller. BAPI tries to send prefetches from the prefetch request buffer to the MSHRs such that the number of different DRAM banks the requests access is maximized rather than sending the prefetches based on FIFO order. To achieve this, the following hardware support is required.

#### 6.1.3.1 Hardware Support

The FIFO prefetch request buffer is modified into the structures shown in Figure 6.3. Instead of having one unified FIFO buffer for buffering new prefetch requests before they enter MSHRs, BAPI contains multiple FIFOs (one per DRAM bank) that buffer new prefetch requests. However, to keep the number of supported new prefetch requests the same as the baseline and also to minimize the total storage cost dedicated to prefetch requests, we use multiple *index buffers* (one per DRAM bank) and a single, unified *prefetch request storage* structure. An index buffer stores indexes (i.e., pointers) into the prefetch request storage structure. The prefetch request storage structure is a regular memory array that stores prefetch addresses generated by the prefetcher. Last, there is a *free list* that keeps track of free indexes in the prefetch request storage structure. The index buffers and free list are all FIFO buffers and all of the buffers have the same number of entries as the baseline unified FIFO.

When the prefetcher generates a request, the free list is consulted. If a free index exists, the request address is inserted into the prefetch request storage structure at the index allocated to it. At the same time, that index is also inserted into the appropriate index buffer corresponding to the bank the prefetch is mapped to. BAPI selects one index among the oldest indexes from each index buffer every processor cycle. Then, the corresponding prefetch request (i.e., prefetch address) is obtained from the prefetch request storage and sent to the MSHR allocator. If the MSHR allocator successfully allocates an entry for the prefetch request, the selected index

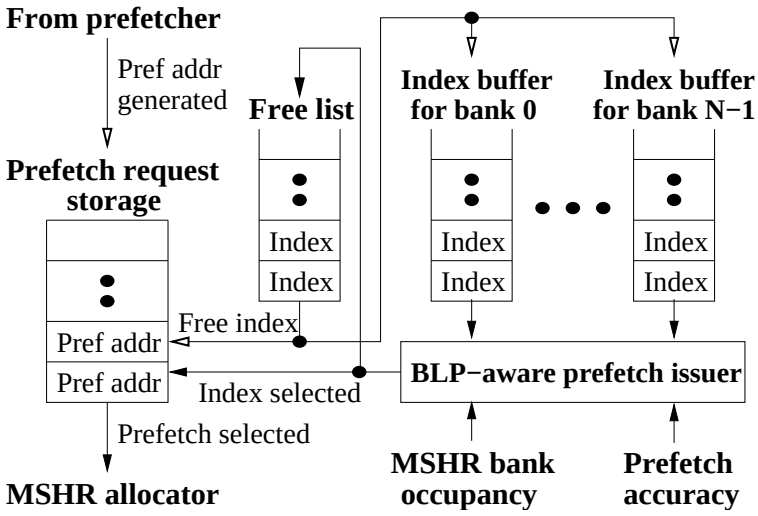



Figure 6.3: Hardware structures for BLP-Aware Prefetch Issue (BAPI)

is inserted into the free list and also removed from the index buffer.

#### 6.1.3.2 BLP-Aware Prefetch Issue Policy

BAPI, shown in Figure 6.3, decides which prefetch to send to the MSHR allocator among the prefetch indexes from each index buffer. It makes its decision based on the DRAM BLP currently exposed in the memory system. To monitor the DRAM BLP of requests, the processor keeps track of the number of outstanding requests (both demands and prefetches) in the MSHRs separately for each DRAM bank. To accomplish this, we use a counter for each DRAM bank, called *MSHR bank occupancy counter*, which keeps track of how many requests to that bank are currently present in the MSHRs. When a demand/prefetch request is allocated an MSHR entry, its corresponding bank occupancy counter is incremented. When a request is serviced and its MSHR is freed, the corresponding bank occupancy counter is decremented.

The key idea of BAPI is to select the next prefetch to place into the MSHRs by examining MSHR bank occupancy counters such that the selected request improves the potential DRAM BLP. To do so, one would choose a prefetch request to the bank whose MSHR bank occupancy counter is the smallest. However, we found

that this policy alone is not enough to expose more BLP to the DRAM controller for all applications. There are a large number of applications for which a prefetcher generates many prefetches to just a single bank but almost no prefetches to the other banks during a phase of execution (especially for streaming applications). For such applications, the issue policy based on MSHR occupancy alone still ends up filling the MSHRs with requests to only one bank. This results in two problems. First, it results in no BLP improvement because the prefetches/demands to other banks that are soon generated cannot be sent to the memory system because the MSHRs are already full. Second, the MSHRs can be filled up with prefetches and thus demands that need MSHR entries can be delayed.

To prevent this problem, BAPI uses a threshold, $prefetch\_send\_threshold$ to limit the maximum number of requests to a single bank that can be outstanding in the MSHRs. This policy reserves room in the MSHRs for requests to other banks when most requests being generated are biased to just a few banks. Because many applications exploit row buffer locality in DRAM banks (since the access latency to the same row accessed last time is relatively low), having too low a threshold can hurt performance by preventing many of the useful prefetches to the same row from being row hits (because the row may be closed before the remaining prefetch requests arrive). On the other hand, having too high a threshold will result in no BLP improvement as the MSHRs may get filled with accesses to only few banks. Therefore, balancing the threshold is important for high performance. We empirically found that a value of 27 (when the total number of MSHR entries is 32) for $prefetch\_send\_threshold$ provides a good trade-off for SPEC benchmarks by exploiting BLP without constraining the row-buffer locality of requests.

Rule 3 summarizes our prefetch issue policy to MSHRs.

#### 6.1.3.3 Adaptive Thresholding Based on Prefetch Accuracy

Prefetching does not work well for all applications or all phases of a single application. In such cases, performance improvement is low (or may even

**Rule 3** BLP-Aware Prefetch Issue policy (BAPI)

**for** each issue cycle **do**

1. Make the oldest prefetch to each bank *valid* only if the corresponding $MSHR\_bank\_occupancy\_counter$ value is less than $prefetch\_send\_threshold$.
2. Among those valid prefetches, select the request to the bank whose $MSHR\_bank\_occupancy\_counter$ value is least.

**end for**

degrade) since useless prefetches will eventually be serviced, resulting in artificially high BLP and wasted DRAM bandwidth. To mitigate this problem, our BLP-aware adaptive prefetch issue policy limits the number of prefetches allowed in the MSHRs by dynamically adjusting $prefetch\_send\_threshold$ based on the run-time prefetch accuracy estimation described in Section 5.2.1. This naturally limits the number of prefetches sent to memory when prefetch accuracy is low. This improves performance for two main reasons: 1) it reserves more room in the MSHRs for demands, thereby reducing contention between demand requests and useless prefetches and 2) it effectively stalls the prefetcher from generating more useless prefetches since the prefetch request buffer will quickly become full.

BAPI dynamically adjusts *prefetch_send_threshold* for each core based on the estimated prefetch accuracy in the previous interval. If the estimated accuracy is very low, a low *prefetch_send_threshold* value is used, which severely limits the number of useless prefetches sent to each bank. We empirically found that three levels of *prefetch_send_threshold* work well for SPEC workloads.

## 6.2 Preserving DRAM Bank-Level Parallelism in CMP systems

BLP-Aware Prefetch Issue (BAPI) increases the potential of DRAM BLP for individual applications on each core. In order for the DRAM controller to exploit this potential, the increased BLP should be exposed to the DRAM request buffers. However, in CMP systems, multiple cores share parts of the on-chip memory system. In our CMP system described in 6.3.2, the DRAM controller (s) is

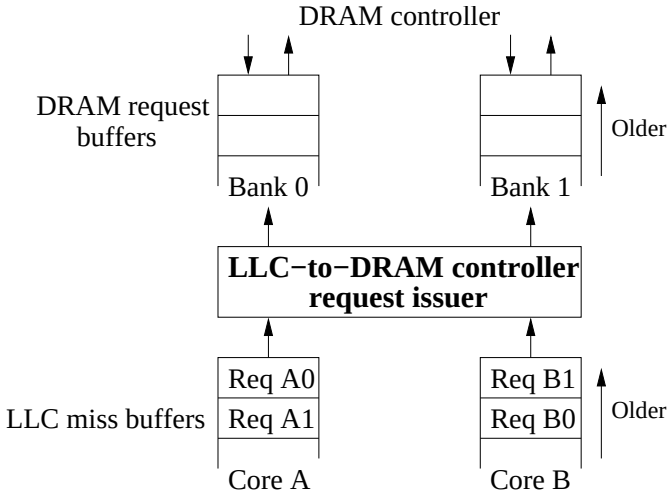
(are) shared by all cores. Therefore, requests from different cores contend for the shared DRAM request buffers in the DRAM controller. Due to this contention, a BLP-unaware Last-Level Cache-to-DRAM Controller (LLC-to-DC) request issue policy can destroy the BLP of an individual application.

### 6.2.1 What Can Destroy BLP of Applications Running Together?

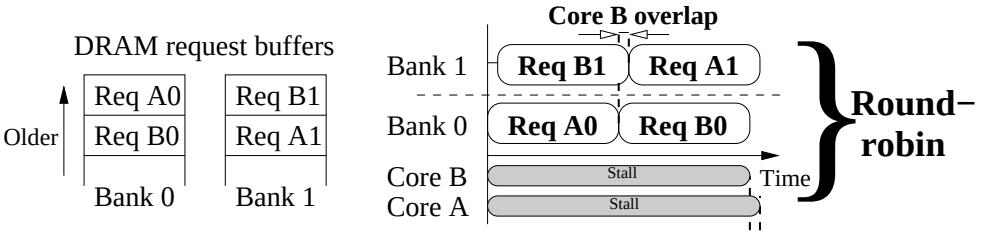
Figure 6.4 describes this problem. Figure 6.4(a) shows the initial state of the last-level cache (LLC) miss buffers of two cores (A and B) and the DRAM request buffers for two DRAM banks. Each core has potential to benefit from BLP in that one request of each core goes to Bank 0 and the other goes to Bank 1. The LLC-to-DC request issuer chooses a single request from the LLC miss buffers to be placed in the corresponding DRAM request buffer every cycle.

When a round-robin policy is employed in the LLC-to-DC request issuer, for each cycle, a request from a different core is issued into DRAM request buffers and the cores are prioritized in a round-robin order. If such a policy is used as shown in Figure 6.4(b), Core A's request to Bank 0 is sent to the DRAM request buffers the first cycle and Core B's request to Bank 1 is sent the next cycle. The DRAM controller based FR-FCFS [66, 76] would service these requests (A0 and B1) from different cores concurrently because they are the oldest in each DRAM bank request buffer. This results in the destruction of the BLP potential of each core because requests from the same core are serviced serially instead of in parallel. Hence, the full latency of each request is exposed to each core and therefore each core stalls for approximately two DRAM bank access latencies.

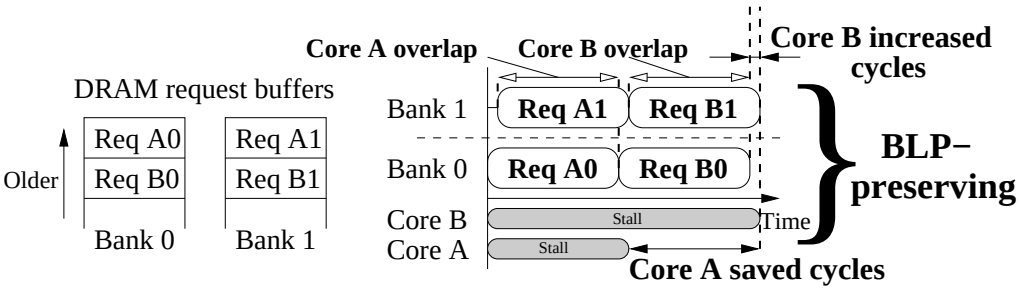
On the other hand, a BLP-preserving LLC-to-DC request issue policy would send all the requests from one core first as shown in Figure 6.4(c). Therefore, the DRAM controller will service Core A's requests (A0 and A1) concurrently since they are the oldest in each bank. The requests from Core B will also be serviced in parallel, after Core A's requests are complete. In this case, the BLP potential of each core is realized by the DRAM controller. The service of Core A's requests finishes




(a) Initial state




(b) Final state of DRAM request buffers & resulting scheduling for round−robin issue




(c) Final state of DRAM request buffers & resulting scheduling for BLP−preserving issue

Figure 6.4: Round-robin vs. BLP-preserving request issue policy

much earlier compared to the round-robin policy because Core A's requests are overlapped. Core A stalls for approximately a single DRAM bank access latency instead of two and core B's stall time does not change much. Therefore, overall system performance improves because Core A can make faster progress instead of stalling.

This example shows that a round-robin-based LLC-to-DC request issue policy can destroy the BLP within an application by consecutively placing requests from different cores into the DRAM request buffers. As such, the DRAM controller may not be able to exploit the BLP potential of each application, which ultimately results in performance degradation. To ensure that each application makes fast progress with its DRAM requests serviced in parallel instead of serially, the LLC-to-DC request issuer should preserve the BLP of requests from each core.

### 6.2.2 Mechanism: BLP-Preserving Multi-core Issue

BLP Preserving Multi-core Request Issue (BPMRI) tries to minimize the destructive interference in the BLP of each application on a CMP system. The basic idea is to consecutively send many memory requests from one core to the DRAM request buffers so that the BLP of that core (or application) can be preserved in the DRAM request buffers for DRAM scheduling. If requests from a single core arrive consecutively (back-to-back) into the DRAM request buffers, they will be serviced concurrently as long as the requests span multiple DRAM banks, thereby preserving the BLP within the individual application. Note that our first technique, BAPI, already increases the likelihood that outstanding memory requests of a core are to different banks; hence, BAPI and BPMRI are synergistic.

BPMRI continues issuing memory requests from a single core into DRAM request buffers until the number of consecutive requests sent reaches a threshold, $request\_send\_threshold$, or there are no more requests in that core's LLC miss buffer. When this termination condition is met, BPMRI chooses another core and repeats the process. BPMRI selects the next core based on how memory intensive

each application is. It prioritizes the core (application) that is the least memory intensive. To do this, BPMRI monitors the number of requests that come into the LLC miss buffer during predetermined intervals using a counter, *LLC miss counter*, for each core. At the start of an interval, BPMRI ranks each core based on the accumulated LLC miss counters (computed during the previous interval) and records the rank in a register, *rank register*, for each core. The core with the lowest value in its LLC miss counter is ranked the highest. The rank determined for each core is used to select the next core (upon meeting a termination condition) during that interval. The LLC miss counters are reset each interval to adapt to the phase behavior of applications. Rule 4 summarizes the BPMRI policy.

**Rule 4** BLP-Preserving Multi-core Request Issue policy (BPMRI)

A *valid request* is a request in a core's LLC miss buffer that has a free entry in the corresponding bank's DRAM request buffer.

**for** each issue cycle **do**
  **next core** ← previous core
  **cond1** ← no valid requests in **next core's** LLC miss buffer
  **cond2** ← consecutive requests from **next core** $>= threshold$
  **if cond1** OR **cond2 then**
    **next core** ← highest ranked core with valid request
  **end if**
  issue oldest valid request from **next core**
**end for**

We choose to limit the maximum number of consecutive requests sent and also choose to prioritize memory non-intensive applications since an uncontrolled "one core-first policy" can lead to the starvation of memory non-intensive applications. If a memory intensive application continuously generates many requests, once those requests start to be issued into the DRAM request buffers, requests from other applications may not get a chance to enter the DRAM request buffers. Limiting the maximum number of requests consecutively sent from a single core alleviates this problem. In addition, the performance impact of delaying requests from a memory non-intensive application is more significant than delaying requests from a memory intensive application. Therefore, prioritizing requests from mem-

ory non-intensive applications (ranking) leads to better overall system performance. Note that this approach is similar to the shortest-job-first policy in that it prioritizes shorter jobs (memory non-intensive cores that spend less time in the memory system) from the point of view of the memory system. The shortest-job-first policy was shown to lead to optimal system throughput [70].

## 6.3 Experimental Methodology

### 6.3.1 Metrics

To measure CMP system performance, we use Individual Speedup (IS), Weighted Speedup (WS), and Harmonic mean of Speedups (HS), which are defined in Section 5.3.1. We also use prefetch accuracy (ACC), prefetch coverage (COV), bus traffic, and instruction window Stall cycles Per Load instruction (SPL) as defined in Section 5.3.1 to analyze the performance of the mechanisms .

To measure the degree of BLP exploited by the DRAM controller quantitatively, we define a BLP metric. We define DRAM BLP as the average number of DRAM banks which are busy (servicing a request) when at least one bank is busy. More formally, $BLP_i$ is defined as the number of DRAM banks that are servicing a request in Cycle $i$. [1] $BUSY_i$ is set to one when at least one bank is servicing a request in Cycle $i$ and reset when no bank is servicing any requests. We define *Aggregate BLP* of an application's total execution as follows:

$$Aggregate\ BLP = \frac{\sum_i BLP_i}{\sum_i BUSY_i}$$

[1] More precisely, a DRAM bank can service multiple row hits at the same time to support back-to-back data transfers as discussed in Section 2.1. However, we assume that only the last request is being serviced in this case to simplify the metric.

### 6.3.2 System Model

We use a slightly different configuration of the x86 system model from the one in Section 5.3.2 for the experimental evaluation of BLP-aware request issue policies. The baseline configuration of each core is shown in Table 6.1 and the shared resource configuration for single, 4, and 8-core systems is shown in Table 6.2. Our simulator also models a DDR3-1600 DRAM system in detail and Table 6.3 shows the DDR3 DRAM timing specifications used for our evaluations.

| | |
|---|---|
| Execution core | Out of order; decode/retire up to 4 instructions,<br>issue/execute up to 8 microinstructions; 15 stages<br>256-entry reorder buffer; 32-entry MSHRs |
| Front end | Fetch up to 2 branches; 4K-entry BTB;<br>64K-entry gshare/PAs hybrid branch predictor |
| On-chip caches | L1 I and D: 32KB, 4-way, 2-cycle, 1 read/write ports;<br>Unified last-level: 512KB (1MB for 1-core), 8-way, 8-bank,<br>15-cycle, 1 read/write port; 64B line size for all caches |
| Prefetcher | Stream prefetcher: 32 stream entries,<br>prefetch degree of 4, prefetch distance of 64 [77, 73],<br>128-entry prefetch request buffer |

Table 6.1: Baseline configuration of each core for BLP-aware issue policies

| | |
|---|---|
| DRAM and bus | 800MHz DRAM bus cycle, DDR3 1600MHz [49],<br>8 to 1 core to DRAM bus frequency ratio;<br>8B-wide data bus per channel, BL = 8; 1 rank,<br>8 banks per channel, 8KB row buffer per bank; |
| DRAM controllers | On-chip, open-row, demand-first [36] FR-FCFS [66]<br>1, 2, 4 channels for 1, 4, 8-core CMPs; |
| DRAM request buffers | 64-entry ($8 \times 8$ banks) for single-core processor<br>256 and 512-entry ($16 \times 8$ banks per channel)<br>for 4 and 8-core CMPs |

Table 6.2: Baseline shared resource configuration for BLP-aware issue policies

### 6.3.3 Workloads

We use the same methodology for compiling and running the SPEC workloads as in Section 5.3.3. The characteristics of the 14 most memory intensive SPEC

| Latency | Symbol | DRAM cycles |
|---|---|---|
| Precharge | $t_{RP}$ | 11 |
| Activate to read/write | $t_{RCD}$ | 11 |
| Read column address strobe (CAS) | $CL$ | 11 |
| Write column address strobe (CAS) | $CWL$ | 8 |
| Additive | $AL$ | 0 |
| Activate to activate | $t_{RC}$ | 39 |
| Activate to precharge | $t_{RAS}$ | 28 |
| Read to precharge | $t_{RTP}$ | 4 |
| Burst length | $t_{BL}$ | 4 |
| CAS to CAS | $t_{CCD}$ | 4 |
| Activate to activate (different bank) | $t_{RRD}$ | 4 |
| Four activate windows | $t_{FAW}$ | 24 |
| Write to read | $t_{WTR}$ | 4 |
| Write recovery | $t_{WR}$ | 12 |

Table 6.3: DRAM timing specifications for BLP-aware issue policies

benchmarks with and without the stream prefetcher on the baseline single-core system model (in Section 6.3.2) are shown in Table 6.4. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran 30 and 15 pseudo-randomly chosen workload combinations for our 4 and 8-core CMP configurations respectively. We imposed the requirement that each of the multiprogrammed workloads have at least one memory intensive application since these applications are most relevant to our study. We consider an application to be memory intensive if its last-level cache Misses Per 1K Instructions (MPKI) is greater than 5.

## 6.4 Implementation and Hardware Cost of BLP-Aware Issue Policies

For evaluations of BAPI, we use $prefetch\_send\_threshold$ values based on the run-time prefetcher accuracy as shown in Table 6.5. We use a value of 10 for $request\_send\_threshold$ for BPMRI. The estimation of prefetch accuracy and rank recording is performed every 100K processor cycles. These values were empirically

| | | No prefetcher | | | Prefetcher | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Type | IPC | MPKI | BLP | IPC | MPKI | BLP | ACC(%) | COV(%) |
| 171.swim | FP00 | 0.29 | 27.58 | 2.60 | 0.61 | 10.81 | 3.58 | 99.95 | 60.79 |
| 178.galgel | FP00 | 1.05 | 12.62 | 3.78 | 0.93 | 11.53 | 3.35 | 23.98 | 12.50 |
| 179.art | FP00 | 0.14 | 130.80 | 1.25 | 0.13 | 106.74 | 1.60 | 46.76 | 18.40 |
| 183.equake | FP00 | 0.48 | 19.89 | 1.29 | 1.08 | 0.78 | 1.89 | 94.76 | 96.06 |
| 189.lucas | FP00 | 0.48 | 10.61 | 1.60 | 0.62 | 3.01 | 1.60 | 72.81 | 71.62 |
| 429.mcf | INT06 | 0.12 | 39.08 | 1.86 | 0.13 | 36.03 | 1.98 | 23.00 | 11.13 |
| 410.bwaves | FP06 | 0.58 | 18.71 | 1.56 | 1.25 | 0.08 | 1.69 | 99.96 | 99.57 |
| 433.milc | FP06 | 0.40 | 29.33 | 1.40 | 0.35 | 21.13 | 1.94 | 20.24 | 27.96 |
| 437.leslie3d | FP06 | 0.46 | 21.14 | 1.64 | 0.76 | 2.06 | 2.20 | 88.25 | 90.39 |
| 450.soplex | FP06 | 0.36 | 21.52 | 1.37 | 0.64 | 3.58 | 1.84 | 81.83 | 83.40 |
| 459.GemsFDTD | FP06 | 0.42 | 16.29 | 2.27 | 0.81 | 1.95 | 2.80 | 90.36 | 88.04 |
| 462.libquantum | INT06 | 0.45 | 13.51 | 1.01 | 1.03 | 0.00 | 1.19 | 99.98 | 99.99 |
| 470.lbm | FP06 | 0.36 | 20.16 | 2.12 | 0.40 | 7.46 | 1.91 | 92.37 | 63.01 |
| 471.omnetpp | INT06 | 0.39 | 11.47 | 1.46 | 0.39 | 9.89 | 1.77 | 11.40 | 19.84 |

Table 6.4: Characteristics of 14 memory-intensive SPEC benchmarks for BLP-aware issue: IPC, MPKI (last-level cache misses per 1K instructions), BLP, ACC (prefetch accuracy), COV (prefetch coverage)

determined by simulations.

| Prefetch accuracy (%) | 0 - 40 | 40 - 85 | 85 - 100 |
|---|---|---|---|
| $prefetch\_send\_threshold$ | 1 | 7 | 27 |

Table 6.5: Dynamic $prefetch\_send\_threshold$ values for BAPI

Table 6.6 shows the storage cost for our implementation of BAPI and BPMRI. The total storage cost for the 4-core system described in Tables 6.1 and 6.2 is 94,440 bits ($\sim$11.5KB), which is equivalent to only 0.6% of the last-level cache data storage. Note that the additional FIFOs (for index buffers and free lists) and prefetch bits account for 99% of the total storage. FIFOs are made of regular memory arrays and index registers (pointers to the head/tail) and therefore the actual design cost/effort is not expensive.

None of the issuing logic for BAPI or BPMRI is on the critical path of execution. Therefore, we believe that our mechanism is easy to implement with low design cost/effort.

| | Structure | Cost equation (bits) | Cost for 4-core |
|---|---|---|---|
| BAPI | Index buffer | $N_{core} \times N_{channel} \times N_{bank} \times N_{buffer} \times log_2 N_{buffer}$ | 57,344 |
| | Free list | $N_{core} \times N_{buffer} \times log_2 N_{buffer}$ | 3,584 |
| | MSHR bank occupancy counter | $N_{core} \times N_{channel} \times N_{bank} \times (log_2 N_{MSHR} + 1)$ | 384 |
| | Prefetch bit | $N_{core} \times (N_{line} + N_{MSHR})$ | 32,896 |
| | Prefetch sent counter | $N_{core} \times 16$ | 64 |
| | Prefetch used counter | $N_{core} \times 16$ | 64 |
| | Prefetch accuracy register | $N_{core} \times 8$ | 32 |
| BPMRI | LLC miss counter | $N_{core} \times 16$ | 64 |
| | Rank register | $N_{core} \times log_2 N_{core}$ | 8 |
| Total storage cost for the 4-core system in Table 6.1 and 6.2 | | | 94,440 |
| **Total storage cost as a fraction of the last-level cache capacity** | | | **0.6%** |

Table 6.6: Hardware storage cost of BAPI and BPMRI ($N_{line}$, $N_{core}$, $N_{MSHR}$, $N_{buffer}$, $N_{channel}$, $N_{bank}$: number of last-level cache lines, cores, MSHR entries, prefetch request buffer entries, DRAM channels, DRAM banks per channel)

## 6.5 Experimental Evaluation and Analysis on BLP-Aware Issue Policies

We evaluate the performance of BLP-Aware Prefetch Issue (BAPI) and BLP-Preserving Multi-core Request Issue (BPMRI) in this section. We first analyze only BAPI on the single-core system in Section 6.5.1 since BPMRI works only for in multi-core systems. We study both BAPI and BPMRI on multi-core systems in the following sections.

### 6.5.1 Single-Core Results

We evaluate BLP-Aware Prefetch Issue (BAPI) in this section. Recall that BAPI aims to increase the BLP potential of a single application whether the application is running alone on a single core machine or running together with other applications on a CMP system. To eliminate the effects of inter-application interference, we first evaluate BAPI on our single core system.

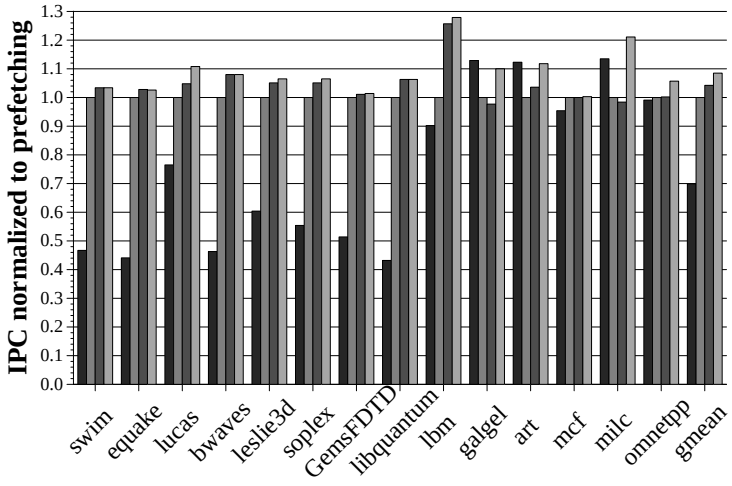
Figures 6.5 and 6.6 show IPC, DRAM BLP, stall cycles per load instruction (SPL), and bus traffic for the 14 most memory intensive benchmarks when we use 1) no prefetching, 2) the baseline with stream prefetching (using the FIFO prefetch issue policy), 3) BAPI with a static threshold (BAPI-static), and 4) BAPI (with adaptive thresholding; BAPI-dynamic or simply BAPI). BAPI-static uses a single constant value for *prefetch_send_threshold* which is set to 27 empirically, whereas BAPI-dynamic varies this threshold based on the accuracy of the prefetcher (as shown in Table 6.5). IPC is normalized to prefetching with the baseline issue policies.

On average, BAPI-dynamic improves performance over the baseline by 8.5%. This improvement is due to two major factors: 1) increased DRAM BLP of prefetches in phases where the prefetcher works well, and 2) limiting the issue of prefetches for applications or phases where the prefetcher is inaccurate. These two factors are analyzed in detail below.

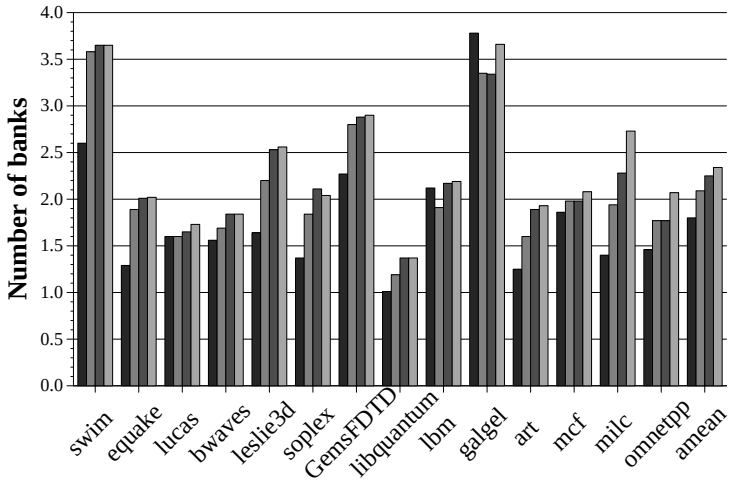
#### 6.5.1.1 Analysis

Both BAPI-static and dynamic improve performance for the nine leftmost benchmarks shown in Figure 6.5(a). These benchmarks are all prefetch friendly as can be seen in Figure 6.6: most of the prefetches are useful (high prefetch accuracy) and these useful prefetches cover a majority of the total bus traffic (high prefetch coverage).

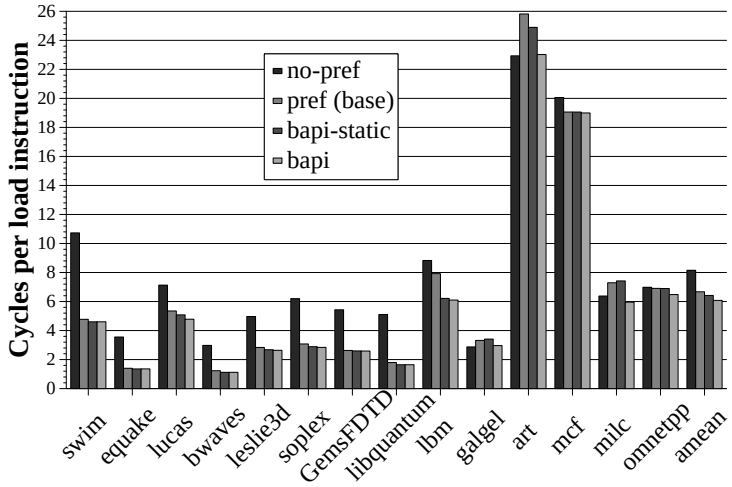
BAPI increases performance over baseline prefetching by exposing more DRAM BLP of prefetches to the DRAM controller. As shown in Figure 6.5(b), BAPI increases BLP for these nine applications and therefore improves DRAM throughput. This leads to significant reductions in stall cycles per load (SPL) as shown in Figure 6.5(c). DRAM throughput improvement also leads to high prefetch coverage. Since MSHR entries are freed sooner due to better DRAM throughput, more prefetches are able to enter the memory system which improves prefetcher coverage. This is best illustrated by the increase in useful prefetches with BAPI for




(a) Performance




(b) Aggregate DRAM BLP




(c) Stall cycles per load instruction

Figure 6.5: Performance, BLP, and SPL of BAPI on single-core system

*swim* and *lbm* as shown in Figure 6.6.

Note that for *lbm*, baseline prefetching with FIFO issue degrades DRAM BLP while improving performance by 10.9% compared to no prefetching. *Lbm* consists of multiple sequential memory access streams in a loop and therefore it exploits DRAM BLP even without prefetching. The stream prefetcher is beneficial by bringing in many cache lines earlier than needed; hence, it improves performance. However, this is done in a BLP inefficient way due to the FIFO prefetch issue policy as described in Section 6.1.2. In other words, the FIFO prefetch issue policy significantly limits the DRAM BLP potential for *lbm* by filling up the MSHRs with prefetch requests that span just a few banks even though there are many younger prefetches to other free DRAM banks waiting in the prefetch request buffer. As a result, the prefetcher's performance improvement is relatively small compared to the other prefetch friendly benchmarks. BAPI mitigates this problem by prioritizing prefetches to different banks, thereby improving DRAM BLP by 15.1% and overall performance by 27.9% compared to the FIFO issue policy.

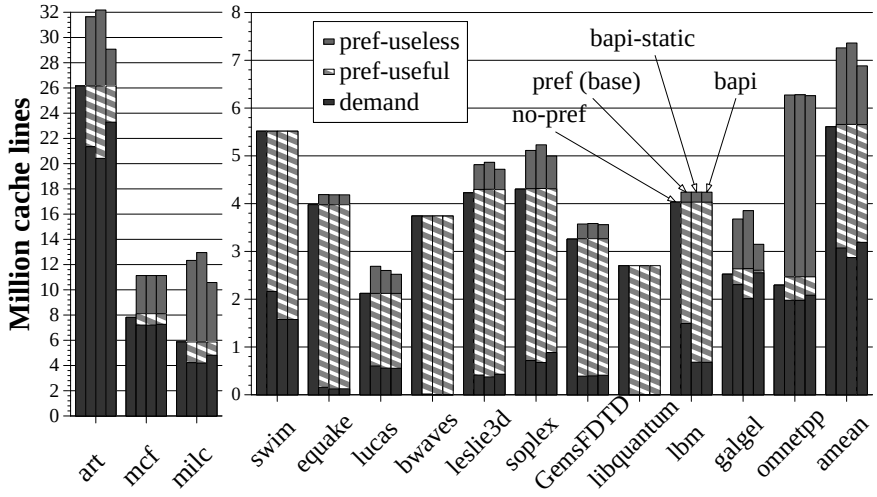



Figure 6.6: Bus traffic of BAPI on single-core system

#### 6.5.1.2 Adaptivity to Usefulness of Prefetches

On the other hand, for the five rightmost benchmarks, BAPI-static does not improve performance over the baseline. As shown in Figure 6.6, the stream

prefetcher does not work well for these benchmarks: it generates a large number of useless prefetches which unnecessarily consume on-chip buffer/cache resources and DRAM bandwidth. As shown in Figure 6.5(a), prefetching degrades performance for *galgel*, *art*, and *milc* compared to no prefetching. BAPI-static does not help these benchmarks either since the useless prefetches are still serviced. In fact, for *galgel*, *art*, and *milc*, BAPI-static increases the number of useless prefetches due to increased DRAM throughput as shown in Figure 6.6. Thus, BLP-aware prefetch issue alone does not help performance when prefetch accuracy is low.

BAPI-dynamic alleviates the problem of useless prefetches by limiting the number of prefetches issued into the MSHRs when the prefetcher generates a large number of useless prefetches. As a result, MSHR entries do not quickly fill up with useless prefetches and thus can be used by demand requests. This mechanism causes the prefetch request buffer to fill up, thereby stalling the prefetcher. As shown in Figure 6.6, BAPI-dynamic eliminates a large number of useless prefetches and reduces total bus traffic by 5.2% on average. BAPI-dynamic almost recovers the performance loss due to useless prefetches for *galgel* and *art*, and improves performance for both *milc* and *omnetpp* by 6.6%.

#### 6.5.1.3 Adaptivity to Phase Behavior

BAPI (or BAPI-dynamic) adapts to the phase behavior of *lucas, leslie3d, soplex, GemsFDTD*, and *lbm*. While most of the time the prefetcher generates useful requests, in certain phases of these applications it generates many useless prefetches. BAPI-dynamic improves performance for these benchmarks by adaptively adjusting $prefetch\_send\_threshold$ which removes many useless prefetches while keeping the useful ones as shown in Figure 6.6.

We conclude that BAPI significantly improves performance (by 8.5%) by increasing DRAM BLP (by 11.7%) while also reducing memory bus traffic (by 5.2%) in the single-core system.

#### 6.5.1.4 Sensitivity to MSHR Size

Thus far we have assumed that each core has a limited number of MSHR entries (32) because MSHRs are costly to scale since they require complex associative search [79]. In this section, we study the effect of our techniques with various MSHR sizes. We varied the total number of MSHR entries from 8 to 256 and measured the average IPC (gmean) for the 14 most memory-intensive benchmarks as shown in Table 6.7. To isolate the effect of limited MSHRs, we assume that there is an unlimited number of DRAM request buffer entries for this experiment (this is why the IPC improvement of BAPI with a 32-entry MSHR is different from that shown in Section 6.5.1). The values of $prefetch\_send\_threshold$ are empirically determined for both BAPI-static and BAPI separately for each MSHR size to provide the best performance.

| MSHR entries | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| Storage cost | 0.6KB | 1.3KB | 2.5KB | 5.1KB | 10.1KB | 20.3KB |
| no-pref IPC | 0.36 | 0.38 | 0.38 | 0.38 | 0.38 | 0.38 |
| pref (base) IPC | 0.43 | 0.50 | 0.53 | 0.56 | **0.59** | 0.58 |
| bapi-static IPC | 0.47 | 0.54 | 0.57 | 0.59 | 0.59 | 0.58 |
| bapi IPC | 0.48 | 0.55 | **0.59** | 0.60 | 0.61 | 0.61 |
| bapi-static's IPC $\Delta$ | 8.5% | 9.1% | 7.8% | 4.0% | 0.0% | -0.1% |
| bapi's IPC $\Delta$ | 10.5% | 10.3% | 10.0% | 6.4% | 3.0% | 4.3% |

Table 6.7: Average IPC performance of BAPI with various MSHR sizes

We make three major observations. First, as the number of MSHR entries increases, the performance of baseline prefetching increases since more BLP is exposed in DRAM request buffers. The performance improvement saturates at 128 entries because the DRAM system itself becomes the performance bottleneck when a high level of BLP is exposed. In fact, increasing the MSHR size from 128 to 256 entries slightly degrades performance because more useless prefetches of some applications (especially, *art* and *milc*) enter the memory system (due to the large number of MSHR entries) causing interference with demand requests both in DRAM and in caches.

Second, both BAPI-static and BAPI (with dynamic thresholding) continue to improve performance up to 64-entry MSHRs since they expose more BLP of prefetches to DRAM request buffers. Even though BAPI-static's performance saturates at 64 MSHR entries, BAPI improves performance with 128 and 256-entry MSHRs because it continues to expose higher levels of *useful* BLP without filling the memory system with useless prefetches. Its ability to adaptively expose useful BLP to the memory system and thereby more efficiently utilize the MSHR entries makes BAPI best-performing regardless of MSHR size.

Finally, BAPI with a smaller MSHR achieves the benefits of a significantly larger MSHR without the associated cost of building one: BAPI with 32-entry MSHRs performs as well as the baseline with 128-entry MSHRs. Similarly, BAPI with 16-entry MSHRs performs within 1% of the baseline with 64-entry MSHRs. Note that BAPI requires very simple extra logic and FIFO structures ($\sim$2KB storage cost for the single-core system) whereas increasing the number of MSHR entries is more costly in terms of both latency and area due to two reasons [79]: 1) MSHRs require associative search, 2) MSHRs require the storage of cache line data. We conclude that BAPI is a cost-effective mechanism that efficiently uses MSHRs and therefore provides higher levels of BLP without the cost of large MSHRs.

### 6.5.2 4-Core Results

In this section, we evaluate BLP-Aware Prefetch Issue (BAPI) and BLP-Preserving Multi-core Request Issue (BPMRI) when employed together in the 4-core CMP system. To provide insight into how our mechanisms work, we begin with a case study.

#### 6.5.2.1 Case Study

We evaluate a workload consisting of four prefetch-friendly (high prefetch accuracy and coverage) applications to show how our mechanisms further improve the benefits of prefetching and thus system performance by improving and preserv-

ing DRAM BLP. Figure 6.7 shows performance metrics when *libquantum, lucas, soplex,* and *GemsFDTD* run together on the 4-core system.

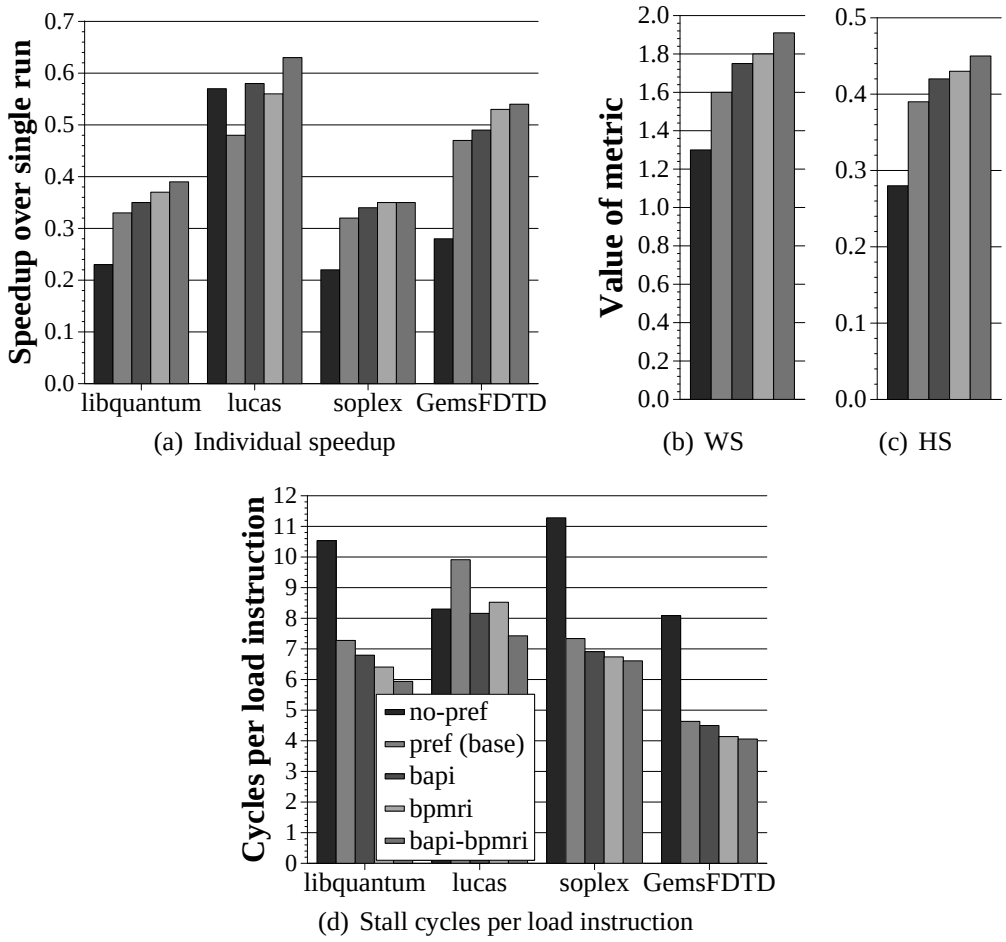



Figure 6.7: Performance of BLP-aware issue policies for prefetch-friendly workload

As shown in Figure 6.7(b), prefetching with the baseline issue policies (FIFO prefetch issue and round-robin LLC-to-DC request issue) improves WS by 23.5% compared to no prefetching. This increase is due to the performance improvement of *libquantum, soplex,* and *GemsFDTD*. The performance of *lucas* actually degrades even though baseline prefetching improves performance for *lucas* on the single-core system (as shown in Section 6.5.1). There are two reasons for this. First, the baseline round-robin LLC-to-DC issue policy destroys the BLP of requests for *lucas* the most among the four applications. Since *lucas* is the least

memory intensive (as shown in Table 6.4) of the four applications, the issue of *lucas*'s requests to DRAM request buffers is relatively infrequent compared to the others. As a result, 1) *lucas*'s requests starve behind more intensive applications' requests in the LLC miss buffer and 2) *lucas*'s BLP is more easily destroyed because requests from other applications intervene between *lucas*'s requests when a round-robin issue policy is used. Second, although amenable to prefetching in general, the prefetch accuracy of *lucas* is not as good compared to the other applications, and therefore *lucas* suffers the most from useless prefetches (as shown in Section 6.5.1).

BPMRI alleviates the first problem as shown in Figures 6.7(a) and (d). BPMRI ranks *lucas*'s requests highest because *lucas* is the least memory intensive application among the four. Whenever BPMRI needs to choose the next core to issue requests from, *lucas* gets prioritized and its requests are issued consecutively into the DRAM request buffers. Therefore, *lucas*'s starvation is mitigated and its BLP is preserved. BPMRI regains the performance lost due to baseline prefetching as shown in Figure 6.7(a). BPMRI also significantly improves the performance of the other three benchmarks by preserving the BLP of each application, thereby improving WS and HS by 12.0% and 11.3% respectively compared to the baseline.

BAPI mitigates the second problem of *lucas*. As discussed in Section 6.5.1, BAPI adapts to the phase behavior of *lucas*: when the prefetcher generates many useless prefetches, BAPI limits the issue of prefetches thereby reducing many of the negative effects of prefetching. On the other hand, BAPI exposes more BLP of prefetches to the memory system when the prefetcher is accurate. Therefore, BAPI increases performance for *lucas* as well as the other three applications, improving WS and HS by 9.4% and 7.9% compared to baseline prefetching.

When BPMRI and BAPI are combined, the performance of each application further improves as each application's SPL is reduced as shown in Figure 6.7(d). BAPI increases each application's BLP potential and BPMRI preserves this BLP thereby allowing the DRAM controller to exploit it. As a result, WS and HS improve by 19.4% and 17.4% respectively compared to the baseline prefetching with BLP-unaware request issue policies.

#### 6.5.2.2 Overall Performance

Figure 6.8 shows the average system performance and bus traffic for all 30 4-core workloads. When employed alone, BAPI improves average performance (WS) by 9.1%, BPMRI by 4.6% compared to the baseline. Combined together, BAPI and BPMRI improve WS and HS by 11.7% and 13.8% respectively, showing that the two techniques are complementary. Bus traffic is also reduced by 5.3%. The performance gain of the two mechanisms are due to 1) increased DRAM BLP provided by intelligent memory issue policies, 2) reduced waste in DRAM bandwidth and on-chip cache space due to limiting the number of useless prefetches.

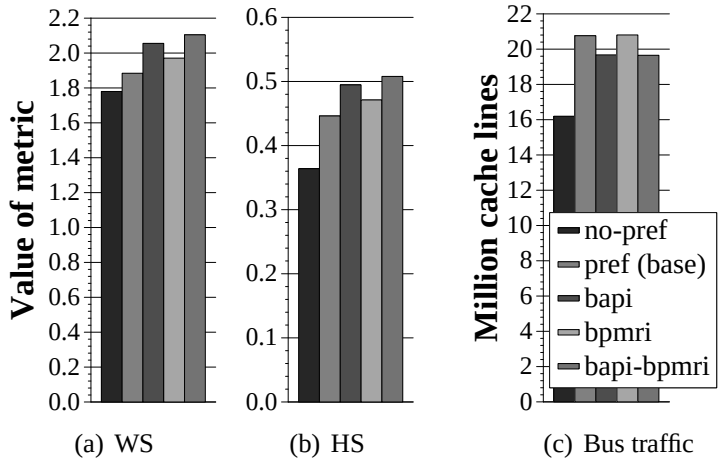
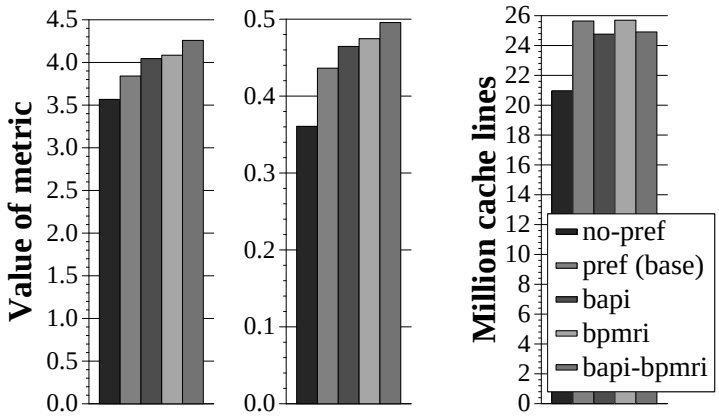



Figure 6.8: Performance of BLP-aware issue policies on 4-core system

### 6.5.3 8-Core Results

Figure 6.9 shows the average system performance and bus traffic for the 15 workloads we examined on the 8-core system. BAPI and BPMRI are still very effective and significantly improve system performance. Combined together, they improve WS and HS by 10.9% and 13.6%, while reducing bus traffic by 2.9%. In contrast to the 4-core system where BAPI alone provided higher performance than BPMRI alone, BPMRI alone improves performance more than BAPI alone. This is because as the number of cores increases, destructive interference in each application's BLP also increases, and reducing this interference becomes a lot more

important.




(a) WS (b) HS (c) Bus traffic

Figure 6.9: Performance of BLP-aware issue policies on 8-core system

We conclude that the proposed techniques are effective in terms of both performance and bandwidth-efficiency for a wide variety of multiprogrammed workloads on both 4-core and 8-core systems.

### 6.5.4 Effect on Other Prefetching Mechanisms

We evaluate our mechanisms on two different types of prefetchers: GHB (Global History Buffer)-based CZone Delta Correlation (C/DC) [59] and PC-based stride [1]. Both the C/DC and stride prefetchers accurately capture a substantial number of memory accesses that are mapped to different DRAM banks, just as the stream prefetcher does. Therefore, BAPI and BPMRI improve system performance compared to the baseline (WS: 10.9% and 5.4%, for C/DC and stride respectively) as shown in Figure 6.10. Our techniques also reduce bus traffic by 4.7% and 2.9% for C/DC and stride respectively. To conclude, our proposal is effective for a variety of state-of-the-art prefetching algorithms.

### 6.5.5 Comparison with Parallelism-Aware Batch DRAM Scheduling

Parallelism-Aware Batch Scheduling (PAR-BS) [54] aims to improve performance and fairness in DRAM request scheduling. It tries to service memory

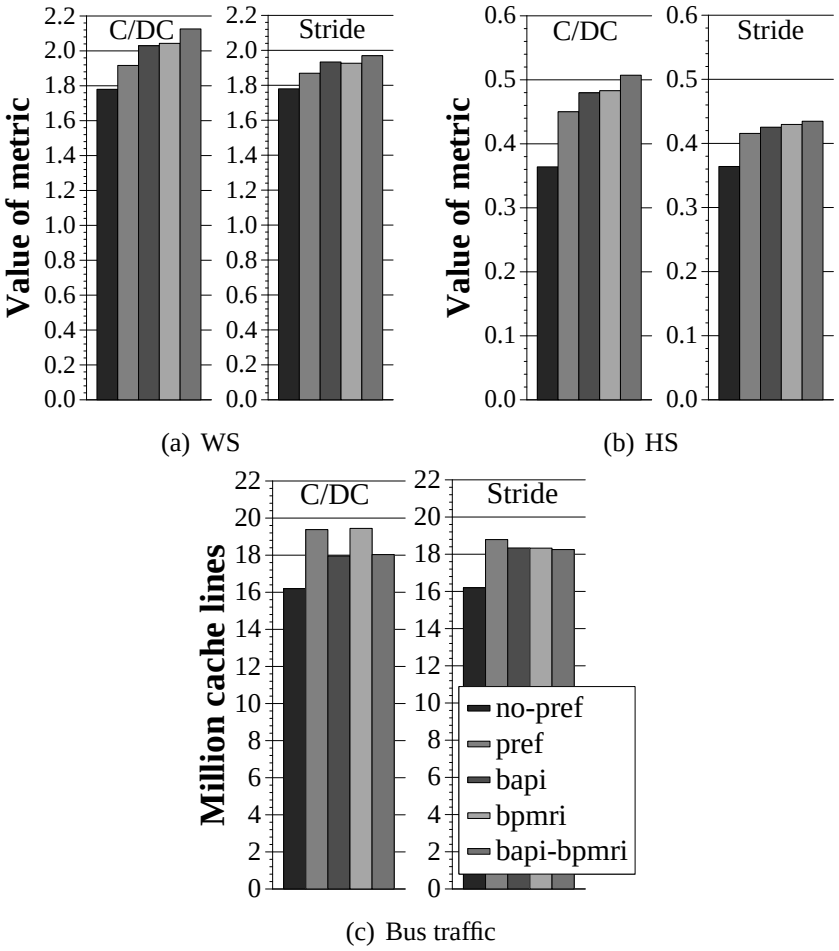



Figure 6.10: BLP-aware issue policies with stride and C/DC prefetchers

requests *in the DRAM request buffers* from the same core concurrently so that the DRAM BLP of each application is preserved in DRAM scheduling. Therefore, just like other scheduling mechanisms, the amount of BLP exploited by PAR-BS is also limited by the number of requests to different banks in DRAM request buffers.

BAPI complements PAR-BS: it increases the number of prefetches to different banks and PAR-BS can exploit this increased level of BLP to improve performance further. BPMRI also complements PAR-BS even though their benefits partially overlap. If an application's requests to different banks are not all in the DRAM request buffers, PAR-BS cannot exploit the full BLP of each application. BPMRI, by consecutively issuing an application's requests from the LLC miss buffer to the DRAM request buffers, increases the probability that each application's requests to different banks are all in the DRAM request buffers. Hence, BPMRI increases the potential of each application's BLP that can be exploited by PAR-BS.

In addition, by consecutively issuing requests from a core back-to-back into the DRAM request buffers, BPMRI enables *any* DRAM controller to service those requests in parallel. Hence, a first-come-first-serve based DRAM controller combined with BPMRI can preserve each application's BLP without requiring the DRAM controller to be BLP-aware.

To verify this, we implemented PAR-BS tuned for best performance for our 4-core workloads. Figure 6.11 shows the performance of 1) baseline prefetching with our baseline FR-FCFS DRAM scheduling policy which exploits row-buffer locality [66], 2) PAR-BS, 3) BPMRI, 4) PAR-BS with BPMRI, 5) PAR-BS with BAPI, 6) PAR-BS with BAPI and BPMRI, and 7) BAPI and BPMRI.

BPMRI's performance gain is equivalent to that of PAR-BS (with the round-robin LLC-to-DC issue policy) since it successfully preserves the BLP of each application and makes the simple FR-FCFS DRAM scheduling policy behave similarly to PAR-BS. When combined with PAR-BS, BPMRI improves WS and HS by an additional 1.9% and 1.4% by better preserving the BLP of requests from each application. BAPI along with PAR-BS significantly improves the performance of

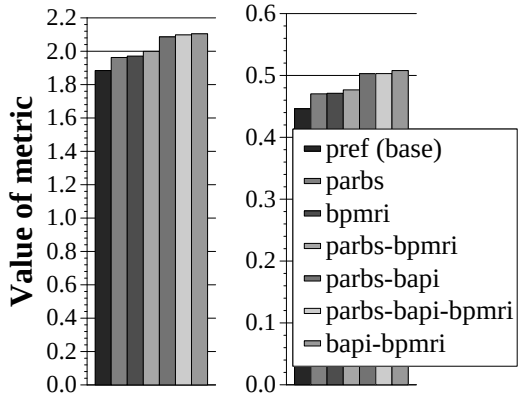



Figure 6.11: Comparison of BLP-aware issue policies with PAR-BS

PAR-BS (WS and HS improve by 7.1% and 7.3% respectively) because BAPI exposes more BLP potential of each application in the DRAM requests buffers for PAR-BS to exploit. To conclude, our mechanisms 1) complement PAR-BS, and 2) enable parallelism-unaware DRAM controllers to achieve similar performance as PAR-BS.

## 6.6 Combination of Prefetch-Aware DRAM Controller and BLP-Aware Issue Policies

Recall that we proposed Prefetch-Aware DRAM Controllers (PADC) to maximize DRAM row buffer hits for useful requests (demands and useful prefetches) in Chapter 5. PADC aims to minimize DRAM latency of useful requests by prioritizing useful row-hit requests over others to the same bank. In other words, the main goal of PADC is to exploit row buffer locality in each bank in a useful manner. The goal of BLP-aware issue policies is orthogonal: BAPI and BPMRI aim to maximize DRAM bank-level parallelism so that more requests from an application can be serviced in different DRAM banks in parallel.

Figure 6.12 shows the performance of PADC alone and PADC combined with our mechanisms for the 4-core workloads. PADC significantly improves WS and HS by 14.1% and 16.3% respectively compared to the baseline. When com-

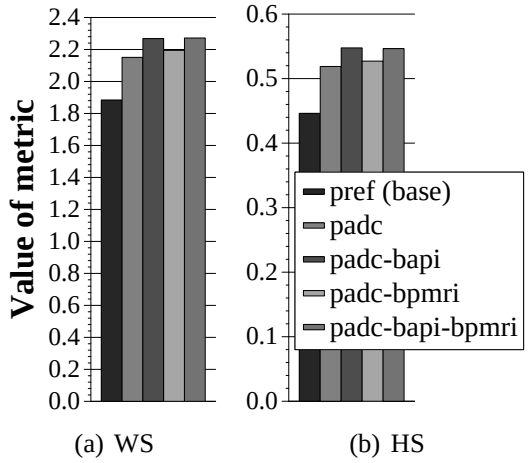



Figure 6.12: Combination of PADC and BLP-Aware Issue Policies

bined with PADC, BAPI and BPMRI improve WS and HS by 20.6% and 22.5%. We conclude that our DRAM-aware prefetch management mechanisms complement each other and significantly improve system performance.

## 6.7 Summary

In this chapter, we show that conventional uncontrolled memory request issue policies to resource-limited on-chip buffers limit the level of DRAM bank-level parallelism (BLP) that can be exploited by the DRAM controller, thereby limiting system performance. To overcome this limitation, we propose new cost-effective on-chip memory request issue mechanisms to improve and preserve BLP of the running applications. Our evaluations show that the mechanisms 1) work synergistically and significantly improve both system performance and bandwidth-efficiency, 2) work well with various types of prefetchers, and 3) complement various DRAM scheduling policies.

# Chapter 7

# Last-Level Cache Management for Improving DRAM Characteristics

In this chapter, we make a case for DRAM-aware last-level cache design: we show that designing the last-level cache replacement policies to be aware of major DRAM characteristics/state can significantly enhance entire system performance. Due to DRAM characteristics, not all misses and evictions of the last-level cache incur the same cost. Bank-level parallelism and row buffer locality allow different outstanding cache misses to be serviced at different latency costs to the processor: fast or slow, parallel or serial. On the other hand, write-caused interference can cause writebacks of dirty cache lines that delay the service of reads and even other writes. This makes cache line evictions incur different cost.

To leverage this, we propose two DRAM-aware last-level cache replacement policies that work together synergistically. The first is a replacement policy that favors the eviction of cache lines that can be refetched quickly due to row buffer locality or serviced together with other misses in different DRAM banks when they are refetched later. The second is a policy that evicts dirty lines that can be written back to DRAM quickly by exploiting row buffer locality, in order to reduce write-caused interference in the DRAM system.

## 7.1 Cache Replacement for Reducing Latency and Increasing BLP

Due to row buffer locality and bank-level parallelism, not all misses incur the same cost from the processor's point of view. Row-hit misses are serviced very quickly, so the processor does not stall very long even though many such misses

occur in the last-level cache. Row conflicts that are serviced in parallel in different banks can also reduce the processor's stall time even though each individual row conflict incurs a long latency. Taking into account these DRAM characteristics in the last-level cache replacement policy has advantages over previous work.

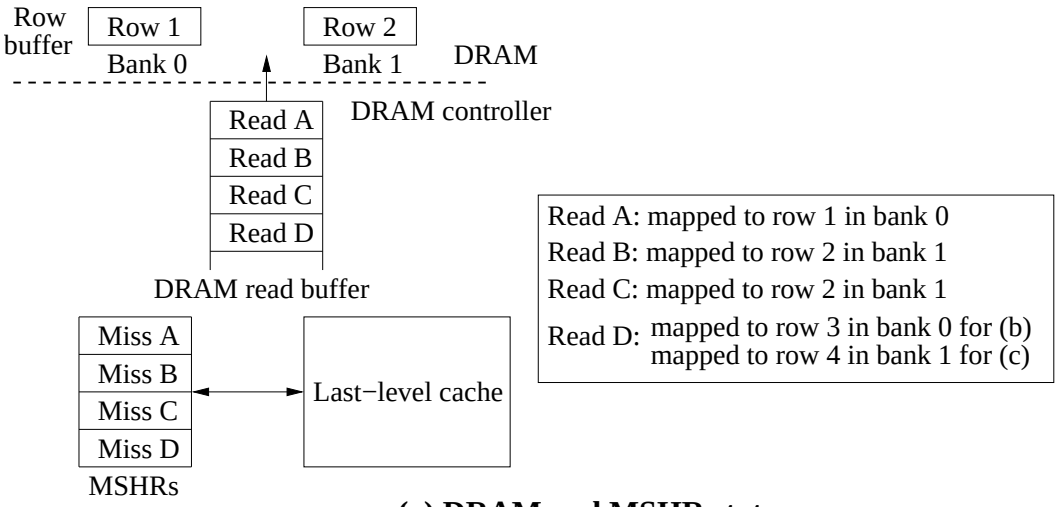
### 7.1.1 Why Should We Consider DRAM Characteristics in Cache Management?

Previously proposed Memory-Level Parallelism (MLP)-aware cache replacement [63] assumes that clustered cache misses incur lower cost than isolated misses. MLP-aware cache replacement makes the implicit assumption that the service times of all clustered cache misses are overlapped with each other. Therefore, such policies prefer to evict cache lines that are serviced concurrently with other misses. However, in many cases, concurrent outstanding misses are not necessarily serviced in parallel in the DRAM system. When multiple row-conflict misses are outstanding in the memory system, they are serviced in parallel *only if* they are mapped to different DRAM banks.

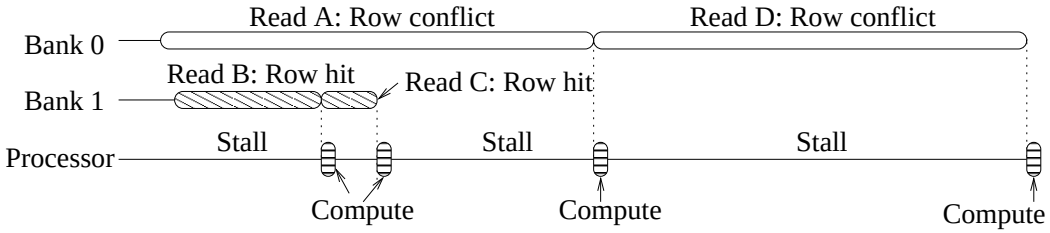
Figure 7.1 describes how the mix of outstanding last-level cache misses can affect DRAM performance and processor stall time. There are four outstanding misses present in the Miss Status/Information Holding Registers (MSHRs) as shown in Figure 7.1(a). Row 1 and Row 2 are open in the row buffer of Bank 0 and Bank 1 respectively. The four misses are waiting in the DRAM read buffer to be serviced by DRAM.

Figure 7.1(b) shows the DRAM service time and processor status when two reads (Reads A and D from Misses A and D) are row conflicts in Bank 0 and two other reads (Reads B and C) are row hits in Bank 1. Since the accesses to Bank 1 are row hits (and therefore low latency), their latencies are overlapped with Read A in Bank 0 (a row conflict). However, Read D is completely serviced alone. The processor must experience the sum of the two row-conflict latencies serially.

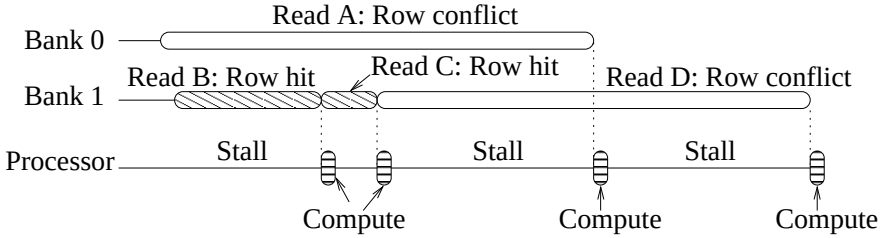
On the other hand, Figure 7.1(c) shows the DRAM service time and pro-




**(a) DRAM and MSHR state**




**(b) DRAM service time when read D is mapped to row3 in bank 0**




**(c) DRAM service time when read D is mapped to row 4 in bank 1**

Figure 7.1: DRAM and processor performance for two different mixtures of outstanding misses

cessor status when Read D is mapped to Bank 1 instead of Bank 0 and is still a row conflict (other requests are the same as Figure 7.1(b)). Read D still takes a long time since it is a row conflict. However, a significant portion of its latency is overlapped with the row-conflict latency of Read A. Therefore this composition of requests results in a significant reduction of processor stall time compared to the previous case.

In contrast to what the MLP-aware mechanism assumes, simply having many misses outstanding in the MSHRs does not necessarily mean that those misses are serviced in parallel. Even though Read D is outstanding with three other misses in both Figures 7.1(b) and (c), its latency is not at all overlapped in the former case yet mostly overlapped in the latter case. As such, depending on the mix of clustered misses, their memory service time (or cost) varies significantly.

Not only isolated misses but also clustered misses to different rows in the same bank incur very high cost. Also row hit misses can always be considered low cost due to their low latencies regardless of BLP (recall that multiple row hits' data is transferred back-to-back in the DRAM system as discussed in Section 2.1). Rather than simply clustering memory requests, an intelligent cache control mechanism should take advantage of low latency and high parallelism conditions in the DRAM system.

To minimize miss cost, a DRAM-aware cache replacement policy can control the mixture of requests such that 1) row-hit misses rather than row-conflict misses occur more frequently and 2) row conflict misses that can be serviced in parallel rather than serially in the DRAM system happen more frequently. Our replacement policy does exactly this by measuring these characteristics.

### 7.1.2 Mechanism: Latency and Parallelism-Aware (LPA) Replacement

We propose Latency and Parallelism-Aware (LPA) last-level cache replacement. The basic idea is to favor the eviction of cache lines that could be refetched quickly due to row buffer locality or serviced together with other misses in different

DRAM banks, when they are refetched later.

The LPA replacement policy leverages the observation that if memory requests of an application show high BLP or row buffer locality in a certain execution phase, similar BLP or row buffer behavior will likely occur in the future. For example, current high BLP requests show high BLP when they are refetched later. Previous research [63] also shows that the memory behavior of applications repeats. Therefore, LPA assumes that cache lines are *low-cost* if they show high BLP or row buffer locality when they are serviced in the DRAM system. Figure 7.2 illustrates the logic that performs this function.

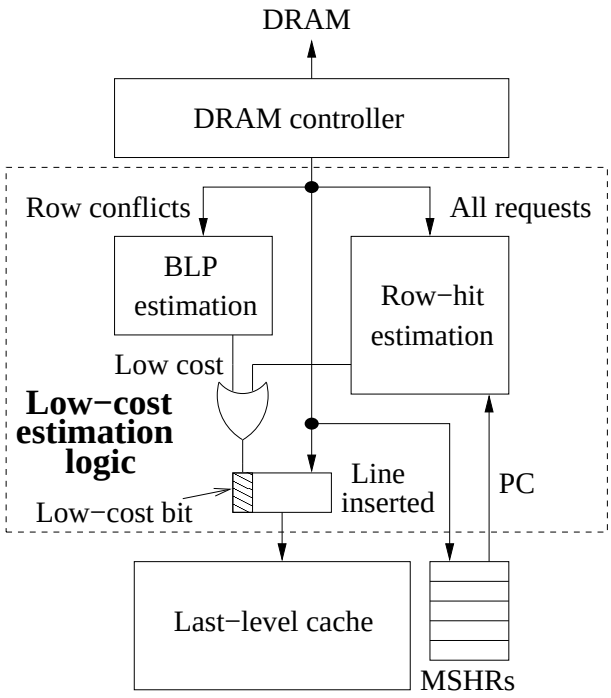



Figure 7.2: Low-cost estimation for LPA

LPA evicts cache lines that are predicted as low-cost. Low-cost cache lines are identified by a one-bit *low-cost field* in each line. LPA always prioritizes low-cost lines over less recently used lines in the set for eviction. If multiple low-cost lines exist, the least recently used (LRU) line among those is selected as the victim. If there is no low-cost line, the LRU line is evicted.

To take into account temporal locality in reused cache lines, the low-cost bit of a cache line that is reused in the cache is deasserted. Doing so enables LPA to outperform LRU replacement for SPEC benchmarks that perform well with LRU replacement. Lines whose low-cost bit are deasserted are retained in the cache by LPA. Additionally, the effective memory latency of misses to low-cost lines that did not exhibit reuse is significantly reduced by taking advantage of row buffer locality and BLP using LPA.

#### 7.1.2.1 Low-Cost Estimation Using BLP Information

To estimate the BLP of a request (or cache line), we need BLP information at runtime. This information is measured by the DRAM controller and sent to the estimation logic.

To measure the degree of BLP quantitatively, we define BLP metrics. $BLP_i$ is defined as the number of DRAM banks that are servicing a request in Cycle $i$. $BUSY_i$ is set to one when at least one bank is servicing a request or set to zero when no bank is servicing any requests in Cycle $i$. We define *Aggregate BLP* of an application's total execution (the same as in Section 6.3.1) and *individual BLP* of a request that is serviced from Cycle $N$ to Cycle $M$ as follows:

$$Aggregate\ BLP = \frac{\sum_i BLP_i}{\sum_i BUSY_i}$$

$$Individual\ BLP = \frac{\sum_{i=N}^{M} BLP_i}{M - N + 1}$$

Aggregate BLP indicates how many banks were busy servicing requests on average when at least one bank was busy, while an application was running. Its value is bound by one and the total number of DRAM banks. Individual BLP of a request indicates how many banks were busy servicing requests in parallel (including its bank) while the request was being serviced. Note that these metrics

can be measured in the DRAM controller at runtime since the DRAM controller already keeps track of which requests are being serviced in which bank.

For a multi-core system, these metrics can be easily gathered on a per-core basis. $BLP_i$ of a core is obtained by considering only the banks that are serving that core's requests. $BUSY_i$ of a core is one when at least one request of that core is being serviced in a bank. Aggregate BLP of a core and individual BLP of a core's requests are calculated using these modifications.

To estimate the BLP of a request (or cache line), we need two pieces of BLP information at runtime: the aggregate BLP during a predetermined execution interval of the application and the request's individual BLP. The DRAM controller measures this information and sends it to the estimation logic. Rule 5 shows how the low-cost estimation works. The estimation logic works only when the aggregate BLP is greater than $aggregate\_BLP\_threshold$. During a high BLP period, the estimation logic marks as low-cost those requests that had much higher individual BLP ($aggregate\_BLP\_offset$ greater) than the aggregate BLP during that interval.

**Rule 5** Low-cost estimation using BLP information

**for** each **row-conflict** request whose service is completed **do**
  **if** aggregate BLP $> aggregate\_BLP\_threshold$ **then**
    **if** individual BLP of the request $>$ (aggregate BLP + $aggregate\_BLP\_offset$) **then**
      mark the request as low-cost
    **end if**
  **end if**
**end for**

Starting estimation only when aggregate BLP is high prevents requests from being marked as low-cost during low BLP phases where there is no large performance benefit from BLP. Marking only those requests that show very high individual BLP compared to the aggregate BLP allows the logic to select only those lines for eviction that are likely to exploit high BLP (i.e., it allows the logic to distinguish very low-cost lines from others). We empirically determined the values

for $aggregate\_BLP\_threshold$ and $aggregate\_BLP\_offset$ (2.5 and 0.3 respectively in our evaluation).

#### 7.1.2.2 Low-Cost Estimation Using Row hit/conflict information

For the low-cost estimation due to row hits, we measure aggregate row hit rate for all requests of an application periodically (as we measure aggregate BLP). Row hit/conflict information of each request is also conveyed (using one bit) from the DRAM controller to the last-level cache.

To estimate whether a cache line is likely to be a row hit, we collect the average row hit rate of the load instruction that caused the miss. The insight behind this is that the majority of row-hit misses occur from a few static load instructions. An example is a load instruction that accesses array data elements in a loop.

The low-cost estimation for frequent row hits is described in Rule 6. We measure the average row hit rate of a load using a small table (a cache structure, 16-entry 4-way associative) each entry of which is associated with a load PC. Each entry keeps track of the total number of requests serviced and the total number of row hits for the load. Whenever a request is serviced, the table is looked up with the load's PC. If a match is found, its counters are updated as follows: 1) the counter for the total number of requests is incremented, and 2) if it was a row hit, the counter for the number of row hits is incremented. If no match is found, the LRU entry is replaced with a new entry and its counters are initialized.

Predicting whether a miss is low-cost or not is made using the information looked up from the load PC table before updating the table. If no match is found, the new cache line is estimated as high-cost (i.e., the low-cost bit is not set). If a match is found, the average row hit rate for the load is calculated by dividing the number of row hits by the number of serviced requests. Prediction is made based on this calculated average row hit rate and the aggregate row hit rate for all requests serviced during an interval.

A fetched line is only considered for low-cost estimation when the row hit

**Rule 6** Low-cost estimation using row hit/conflict information

```
for each request whose service is completed do
  match found ← look up load PC table (request's PC)
  if match found then
    (total number of row hits, total number of requests) ← load PC table (request's PC)
    load PC table (request's PC) ← (total number of row hits + (request row hit ? 1 : 0), total number of requests + 1)
    adjusted aggregate row hit rate ← MAX(aggregate row hit rate, aggregate_row_hit_rate_min)
    if total number of requests > request_threshold and row hit rate > adjusted aggregate row hit rate then
      Mark the request as low-cost
    end if
  else
    get entry from load PC table (request's PC)
    load PC table (request's PC) ← ((request row hit ? 1 : 0), 1)
  end if
end for
```

rate information is collected for long enough (more than $request\_threshold$) to indicate the load will likely generate many row hits. Not marking lines whose load had only few requests serviced prevents making a wrong decision about whether the load would generate many row hits or not.

The logic marks the line as low-cost only if the row hit rate of the load that caused the line's fetch is greater than the aggregate row hit rate (using adjusted aggregate row hit rate) for all fetched lines. We also impose a minimum value of aggregate row hit rate ($aggregate\_row\_hit\_rate\_min$) to avoid falsely marking lines as low-cost simply because their row hit rate, although quite low, is larger than a very low aggregate row hit rate. We empirically found a set of the parameter values ($request\_threshold$ of 30 and $aggregate\_row\_hit\_rate\_min$ of 0.6) for our evaluation.

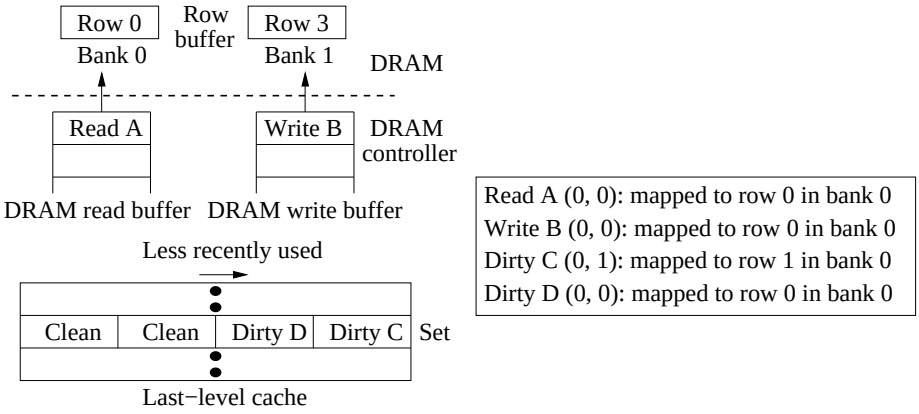
## 7.2 Cache Replacement for Reducing Write-Caused Interference

Not all dirty line evictions for the last-level cache incur the same cost. This is because row-conflict writes are much more expensive than row-hit writes as shown in Section 2.3. Long delays caused by row-conflict write accesses can delay the service of writes in the write buffer and eventually result in delaying the service of reads. In contrast, row-hit writes can be serviced back-to-back just like row-hit reads. Therefore, increasing row-hit writes that are concurrently outstanding is desirable. Note that the source of DRAM writes is the last-level cache's writebacks, i.e., dirty line evictions. A write-caused interference-aware replacement policy would find and evict dirty cache lines that cause row-hit write accesses to DRAM. The resulting row-hit writes can significantly improve the service time of the writes. The following example shows the implication on DRAM system performance for last-level cache replacement policies.

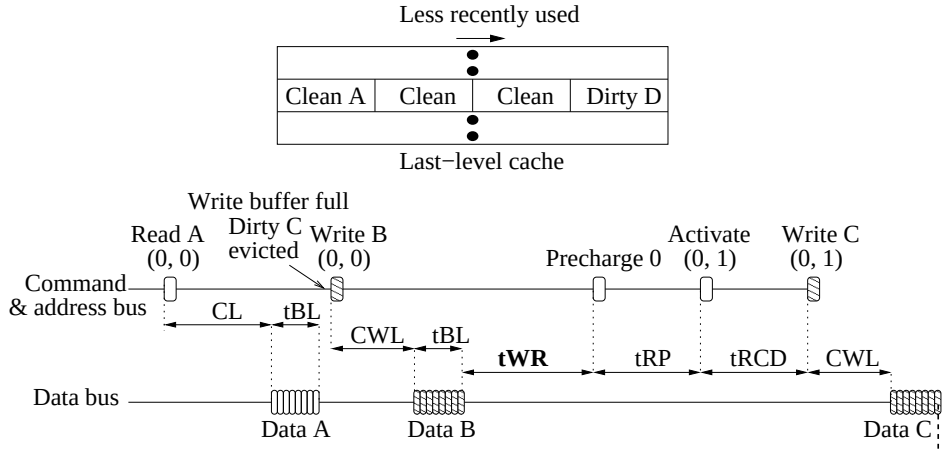
### 7.2.1 Why Should We Consider Write-Caused Interference in Cache Management?

Figure 7.3 shows how a write-caused interference-aware replacement policy can improve DRAM performance. Figure 7.3(a) shows the initial state of the DRAM read/write buffers and a set of the last-level cache. A row-hit read (Read A) and a row-hit write (Write B) are waiting to be scheduled to DRAM. Two dirty lines (Dirties C and D) are at the least recently used (LRU) positions of the shown last-level cache set. Dirty C is mapped to a different row from the currently open row in Bank 0 whereas Dirty D is mapped to the same row as Write B.

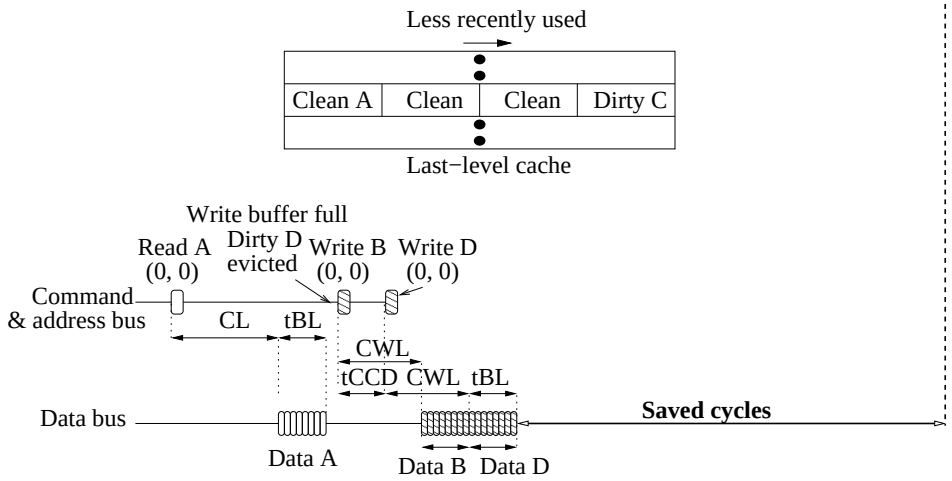
Figure 7.3(b) shows the resulting cache state and the DRAM timing when a conventional LRU policy is used in the cache. The LRU line (Dirty C) is evicted by the fetched line for Read A after Read A is serviced by DRAM. Therefore a write (Write C) is generated for Row 1 and is inserted into the write buffer. Writes are serviced in the order of Writes B and C. Because Write C accesses a different row from Write B (row conflict), precharging is required to open Row 1. Since a




**(a) Cache/DRAM buffer initial state**




**(b) Cache state and DRAM timing for conventional replacement (Dirty C is evicted)**




**(c) Cache and DRAM timing for interference−aware replacement (Dirty D is evicted)**

Figure 7.3: Conventional vs. write-caused interference-aware replacement policies

write was serviced before, write-to-precharge penalty must be satisfied before the precharge command for Write C is scheduled. This increases the idle cycles on the DRAM data bus since the write data for Write C must wait for $t_{WR} + t_{RP} + t_{RCD} + CWL$ cycles after the write burst for Write B.

On the other hand, as shown in Figure 7.3(c), if Dirty D is evicted instead of Dirty C, the two writes (Writes B and C) are serviced back-to-back, thereby resulting in significant reduction of DRAM service time. This example illustrates that a simple cache replacement policy which evicts row-hit writeback requests can improve service time for writes. Our Write-caused Interference-Aware (WIA) replacement policy is designed to achieve this.

### 7.2.2 Mechanism: Write-Caused Interference-Aware (WIA) Replacement

WIA evicts row-hit dirty lines when a replacement happens in the last-level cache. Ideally, row-hit dirty lines can be found by comparing the row address of each dirty line in the set (that is considered for replacement) with the address of every write in the DRAM write buffer. However, the hardware/design cost of this is not acceptable since it requires an associative search of the write buffer with the address of each dirty line in the cache set. To simplify implementation and hardware cost, we use a row address register for each DRAM bank to keep track of the address of the last evicted dirty line mapped to that bank. In our address mapping, the last-level set index field includes the DRAM bank index field [1]. Therefore all lines in a set belong to one DRAM bank. This requires one associative search: the stored row address in a register is compared to the address of each dirty line in the cache set. This can be performed by the tag comparison logic in the cache. The tag comparison structure should be modified to support comparing the stored row address with the

[1]This mapping can increase DRAM bank conflicts (among reads and writes with different row addresses) that causes many row conflicts. However, a write buffer policy that drains writes only when it is full can mitigate this problem significantly. We use this write buffer policy as presented in Section 7.6.2. Also, we found that keeping track of only the last evicted dirty line's row address globally regardless of the banks also works well. This option can be used for systems with different address mapping.

row addresses of all lines in the set. Figure 7.4 illustrates how WIA searches for row-hit dirty lines.

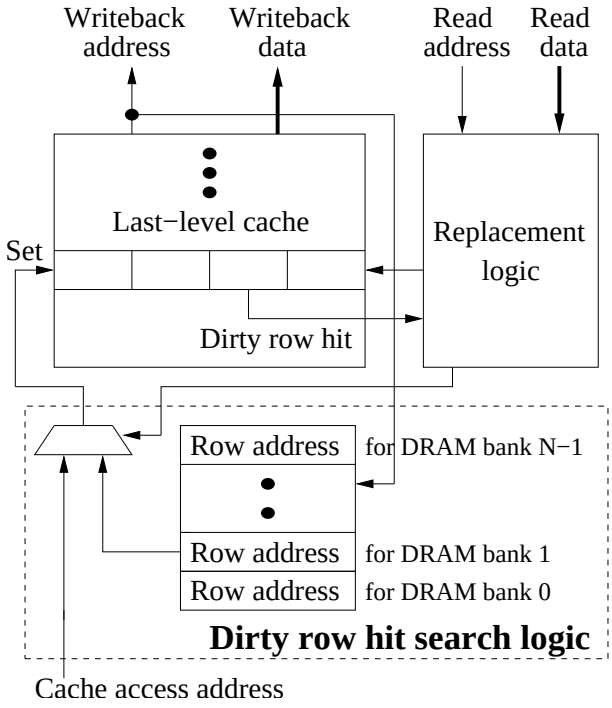



Figure 7.4: Dirty row-hit search for WIA

Whenever a dirty line is evicted (i.e., a writeback is generated), its DRAM bank's row address register is updated with the dirty line's row address. When a replacement happens in a cache set, WIA looks for a dirty line that is mapped to the same row as the last evicted dirty line for the corresponding DRAM bank using tag comparison logic in the cache. We found that keeping track of the last evicted row address is enough to gain most of the benefits of searching the row addresses of all writes in the entire write buffer.

WIA prioritizes row-hit dirty lines (if found) over the LRU line for eviction. If multiple row-hit dirty lines are found, the LRU among them is evicted. If none are found, the LRU line is evicted. We found that prioritizing row-hit dirty lines over LRU lines for eviction does not hurt performance due to loss of temporal locality. This is because 1) if the evicted dirty line is required, the write buffer forwards it to the cache unless it is already written back, 2) very few evicted dirty lines by WIA

are reused, and 3) performance benefits of evicting row-hit dirty lines outweighs the cost of re-fetching (a small number of) these lines from DRAM.

## 7.3 Combining Latency and Parallelism-Aware and Write-Caused Interference-Aware Policies

LPA and WIA can be combined to reduce both miss and dirty line eviction penalties. We found that prioritizing row-hit dirty lines (detected by WIA) over low-cost lines (predicted by LPA) for victim decision performs very well. The reasons are as follows.

First, LPA alone is unaware of the dirty line eviction cost. LPA can increase write-caused interference if it evicts costly dirty lines (i.e., row conflicts to the same bank) since it only predicts whether or not lines would be low-cost when they are fetched again later.

Second, WIA's detection of row-hit dirty lines is more accurate than LPA's prediction of low-cost read misses. This is because WIA looks for dirty lines that can be serviced very soon with other currently outstanding writes, whereas LPA predicts low-cost read misses that are required in the future.

Finally, WIA's penalty of wrong decisions, i.e., an evicted dirty line is reused, is mitigated by possible forwarding of such cache lines from the write buffer. In contrast, LPA's wrong decision, evicting a useful and costly cache line, can negatively affect performance more: the processor must stall for a long time as the cache line needs to be fetched from main memory.

## 7.4 Multi-Core System Considerations

In many chip-multiprocessors (CMP), multiple cores share the last-level cache and main memory resources. When multiple applications run on different cores, their requests compete with each other for the shared resources. Using *global* BLP and row hit rate (as opposed to per-application information) for the purposes

of our LPA replacement policy can cause unwanted cache replacement decisions. For example, cache lines of an application that generates many low-cost requests (high row-hit rate and high BLP) can be evicted too frequently. Similarly, cache lines of another application with many high-cost (low row-hit rate and low BLP) misses could be evicted very rarely. This can hurt system performance.

### 7.4.1 LPA Replacement in Multi-Core

We modify the LPA replacement policy to be core-aware to avoid this problem. To make LPA effective in CMPs, we estimate low-cost lines on a per-core basis. We measure aggregate BLP/row hit rate and individual BLP/row-hit for each core independently. As discussed in Section 7.1.2.1, the aggregate BLP for Core A and individual BLP for the requests of Core A are calculated by considering only Core A's requests that are serviced in different banks. Low-cost estimation for Core A's lines is performed using these aggregate BLP and individual BLP values. Row hit rate of Core A is measured by dividing the number of Core A's row-hit requests by the total number of Core A's requests serviced in the time interval. Finally, one load PC table is required for each core for low-cost estimation using row-hit information.

When a cache line is inserted into a cache set, LPA determines each core's victim by considering only its lines based on LPA policy discussed in Section 7.1.2. Among each of the cores victims, LPA chooses to evict the victim of the core to which the LRU line in the entire cache set belongs.

### 7.4.2 WIA Replacement in Multi-Core

On the other hand, WIA does not need to be core-aware. This is because writes are not critical to an application's progress. Writes become critical only when the DRAM controller cannot service reads due to write-caused interference. Therefore, servicing many writes (from any core) very quickly so that reads (from any core) can be serviced soon and without delay leads to high performance. As

such, the WIA policy in multi-core systems stays the same as we described for the single-core system.

We evaluate our mechanism using these techniques on a 4-core CMP system in Section 7.8.2.

## 7.5 Comparison to Memory-Level Parallelism-Aware Replacement

Qureshi et al. [63] proposed a MLP-aware cache replacement policy that prioritizes the eviction of a cache line that is likely to be serviced together with other misses when it is fetched next. Any misses that are outstanding concurrently in the miss buffers are assumed (and hoped) to be actually serviced in parallel in the main memory system. This policy does not take into account the state and characteristics of DRAM in its decision making. As such, it has multiple important limitations compared to our DRAM-aware policies.

First, the MLP-aware policy is not DRAM bank-aware. As we discussed in Section 7.1.1, clustered misses to different rows in the same bank incur very high cost. Since the MLP-aware policy estimates the "MLP cost" of a cache line using the absolute number of outstanding misses (in the MSHRs), it assumes that misses to the same bank will be serviced in parallel with other misses, which is not correct. As such, the MLP-aware policy is prone to mispredicting the cost of misses significantly.

Second, the MLP-aware policy does not consider the cost of writebacks. Instead, it considers only the future miss cost of a line when making eviction decisions. This can hurt performance because it can increase write-caused interference in the DRAM system by causing a large number of row-conflict writebacks. As we showed in Sections 7.2.1 and 7.8.1.2, row-conflict writebacks can degrade system performance significantly.

Third, the MLP-aware policy is unaware of the cause of low-latency misses. The MLP-aware policy implicitly identifies low-latency misses by estimating the

MLP cost for each miss. However, it does not know whether the low cost was due to high BLP or row buffer locality. This distinction is important since a row-hit request that is serviced slowly the first time (due to many outstanding requests) may be serviced quickly (and therefore low-cost) when refetched.

Finally, the hardware/design cost of the MLP-aware policy is more than our proposal. Since MLP cost is stored in each cache line, multiple bits are required in each line (e.g, 3 bits per cache line). In contrast, our LPA requires only one bit (indicating low-cost) per line.

We quantitatively compare the performance of the MLP-aware replacement policy to our mechanisms in Section 7.8.

## 7.6 Experimental Methodology

### 7.6.1 Metrics

To measure multi-core system performance, we use Individual Speedup (IS), Weighted Speedup (WS), and Harmonic mean of Speedups (HS), which are defined in Section 5.3.1.

### 7.6.2 System Model

The baseline configuration of processing cores and the memory system for single and 4-core CMP systems is shown in Table 7.1. Our simulator also models DDR3 DRAM performance-related timing constraints in detail as shown in Table 7.2. Note that our baseline employs a *drain_when_full* DRAM write buffer policy for the evaluation of the proposed replacement policies. This write buffer policy tolerates read-to-write switching penalties best with today's high-bandwidth DDR DRAM systems with their large write-caused interference. We discuss and compare this policy to other existing write buffer policies extensively in Chapter 8.

| | |
|---|---|
| Execution Core | Out of order, decode/retire up to 4 instructions,<br>issue/execute up to 8 microinstructions; 15 stages<br>256-entry reorder buffer; |
| Front End | Fetch up to 2 branches; 4K-entry BTB;<br>64-entry return address stack;<br>64K-entry gshare/PAs hybrid branch predictor |
| Caches and on-chip buffers | L1 I/D-cache: 32KB, 4-way, 2-cycle, 64B line size;<br>Shared last-level cache: 16-way, 8-bank, 15-cycle,<br>1 read/write port per bank, LRU replacement<br>writeback, 64B line size, 1, 2MB for 1, 4-core systems;<br>32, 128 MSHRs for 1, 4-core systems<br>32, 128-entry LLC access/miss/fill buffers for 1, 4-core systems |
| DRAM and bus | 1, 2 channels (DRAM controllers) for 1, 4-core systems;<br>800MHz DRAM bus cycle,<br>Double Data Rate (DDR3 1600MHz) [49];<br>8B-wide data bus per channel, BL = 8;<br>1 rank, 8 banks per channel, 8KB row buffer per bank; |
| DRAM controllers | On-chip, open-row, FR-FCFS scheduling policy [66];<br>64-entry ($8 \times 8$ banks) DRAM read/write buffers per channel<br>drain_when_full write buffer policy |

Table 7.1: Baseline configuration for DRAM-aware replacement policies

### 7.6.3 Workloads

We use the same methodology for compiling and running the SPEC workloads using ICC/IFORT and Pinpoints as discussed in Section 5.3.3.

Even though we evaluated all the 55 SPEC benchmarks, we report 16 memory intensive benchmarks on which the performance impact of our mechanisms is significant; the effect of our mechanisms on the remaining applications is negligible. Characteristics of the 16 SPEC benchmarks are shown in Table 7.3. We consider memory read (cache miss) and write (writeback) characteristics independently since LPA is designed for DRAM read efficiency and WIA targets DRAM write efficiency. Last-level cache Writebacks Per 1K Instructions (WPKI) indicates how intensively a benchmark generates write requests to the DRAM system.

To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran

| Latency | Symbol | DRAM cycles |
|---|---|---|
| Precharge | $t_{RP}$ | 11 |
| Activate to read/write | $t_{RCD}$ | 11 |
| Read column address strobe (CAS) | $CL$ | 11 |
| Write column address strobe (CAS) | $CWL$ | 8 |
| Additive | $AL$ | 0 |
| Activate to activate | $t_{RC}$ | 39 |
| Activate to precharge | $t_{RAS}$ | 28 |
| Read to precharge | $t_{RTP}$ | 6 |
| Burst length | $t_{BL}$ | 4 |
| CAS to CAS | $t_{CCD}$ | 4 |
| Activate to activate (different bank) | $t_{RRD}$ | 6 |
| Four activate windows | $t_{FAW}$ | 24 |
| Write to read | $t_{WTR}$ | 6 |
| Write recovery | $t_{WR}$ | 12 |

Table 7.2: DDR3-1600 DRAM timing specifications for DRAM-aware replacement policies

17 randomly chosen workload combinations for our 4-core CMP configuration.

## 7.7 Implementation and Hardware Cost of DRAM-Aware Replacement Policies

For evaluations, we periodically measure the aggregate row hit rate and BLP every 100K processor cycles for low-cost estimation. We empirically set $aggregate\_BLP\_threshold$ and $aggregate\_BLP\_offset$ to 2.5 and 0.3 respectively for high BLP estimation. We use a 16-entry 4-way set associative structure for the load PC table and set $request\_threshold$ and $aggregate\_row\_hit\_rate\_min$ to 30 and 0.6 for row-hit estimation. BLP and row-hit information required for LPA is collected only from reads (not writes).

Table 7.4 shows hardware storage cost for our mechanisms on the single and 4-core systems in Table 7.1. The BLP information (aggregate and individual BLP) is not sent from the DRAM controller to the last-level cache to avoid additional storage and long wires. The BLP estimation is performed in the DRAM controller,

| | | | Reads | | | Writes | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | Type | IPC | MPKI | RHR(%) | BLP | WPKI | RHR(%) | BLP |
| 179.art | FP00 | 0.26 | 90.92 | 95.43 | 1.78 | 9.79 | 86.75 | 1.49 |
| 482.sphinx3 | FP06 | 0.39 | 12.94 | 83.01 | 1.17 | 0.63 | 58.18 | 1.79 |
| 181.mcf | INT00 | 0.06 | 107.74 | 70.08 | 1.32 | 11.50 | 15.03 | 2.89 |
| 171.swim | FP00 | 0.35 | 23.10 | 36.95 | 2.31 | 8.24 | 78.33 | 2.55 |
| 173.applu | FP00 | 0.93 | 11.40 | 90.34 | 1.56 | 1.78 | 81.34 | 1.74 |
| 462.libquantum | INT06 | 0.67 | 13.51 | 94.96 | 1.01 | 5.87 | 89.13 | 1.06 |
| 437.leslie3d | FP06 | 0.54 | 20.88 | 70.50 | 1.95 | 2.72 | 73.80 | 2.05 |
| 481.wrf | FP06 | 0.72 | 8.11 | 72.95 | 1.47 | 2.52 | 76.17 | 1.70 |
| 459.GemsFDTD | FP06 | 0.49 | 15.63 | 45.81 | 2.21 | 6.91 | 50.60 | 2.70 |
| 189.lucas | FP00 | 0.61 | 10.61 | 61.00 | 1.36 | 2.38 | 34.19 | 1.08 |
| 450.soplex | FP06 | 0.40 | 21.24 | 81.64 | 1.30 | 3.75 | 42.48 | 1.60 |
| 436.cactusADM | FP06 | 0.63 | 4.51 | 7.42 | 1.36 | 1.22 | 33.31 | 1.54 |
| 471.omnetpp | INT06 | 0.49 | 10.11 | 63.45 | 1.27 | 4.17 | 6.88 | 2.46 |
| 176.gcc | INT00 | 0.93 | 3.24 | 90.62 | 1.07 | 0.54 | 39.53 | 1.56 |
| 178.galgel | FP00 | 1.42 | 4.84 | 54.45 | 2.99 | 1.16 | 11.51 | 3.03 |
| 464.h264ref | INT06 | 1.48 | 1.28 | 89.56 | 1.07 | 0.28 | 63.55 | 1.90 |

Table 7.3: Characteristics of 16 SPEC benchmarks for DRAM-aware replacement: IPC, MPKI (last-level cache misses per 1K instructions), WPKI (last-level cache Writebacks Per 1K Instructions), row hit rate (RHR), BLP

and a one-bit field (high/low BLP bit in Table 7.4) is carried by each read request. Similarly, one bit row hit/conflict field is also carried by each request for row-hit estimation before being inserted into the cache.

LPA and WIA require only 0.2% of the total last-level cache space on both systems. We assume that the core ID field is already available in each cache line on the 4-core system. If the core ID field (2 bits) is also considered, our mechanisms require 12.7KB (0.6% of last-level cache), which is still insignificant. Note that none of the logic or structures required for the mechanisms is on the critical path.

## 7.8 Experimental Evaluation and Analysis on DRAM-Aware Replacement Policies

We present experimental results for our mechanisms on the single-core and 4-core systems. We first analyze the DRAM-aware replacement policies intensively on the single-core system.

| | Structure | Cost equation (bits) | Cost for 4-core |
|---|---|---|---|
| LPA | Aggregate BLP & busy counters and BLP register | $16 \times 3 \times N_{core}$ | 192 |
| | Individual BLP & busy counters | $16 \times 2 \times N_{bank}$ | 512 |
| | High/low BLP bit | $1 \times N_{buffer}$ | 128 |
| | Aggregate row-hit & request counters and row hit rate register | $16 \times 3 \times N_{core}$ | 192 |
| | Load PC table's tag store (16-entry 4-way) | $27 \times 16 \times N_{core}$ | 1,728 |
| | Load PC table's data store (row-hit/request counters) | $2 \times 16 \times 16 \times N_{core}$ | 2,048 |
| | Row hit/conflict bit | $1 \times N_{buffer}$ | 128 |
| | Low-cost bit | $1 \times N_{line}$ | 32,768 |
| WIA | Row address registers | $32 \times N_{bank}$ | 512 |
| Total storage cost for the 4-core systems in Table 7.1 | | | 38,208 |
| Total storage cost as a fraction of the last-level cache capacity | | | 0.2% |

Table 7.4: Hardware storage cost for DRAM-aware replacement policies ($N_{core}$, $N_{line}$, $N_{bank}$, $N_{buffer}$: number of cores, last-level cache lines, DRAM banks, cache fill buffer entries)

### 7.8.1 Single-Core Results

Figure 7.5 shows IPC normalized to the baseline for the baseline LRU, MLP-aware, Latency and Parallelism-Aware (LPA), Write-caused Interference-Aware (WIA), and combined LPA-WIA replacement polices. The MLP-aware policy is implemented with a set-sampling mechanism that selects between (MLP-aware) linear and LRU policies as proposed by Qureshi et. al [63].

Overall, the best performing policy is the combination of LPA and WIA, which improves performance by 11.4% (6.9% excluding *art*) on average. In contrast, the MLP-aware policy improves performance by 4.6% (0.6% excluding *art*). LPA and WIA complement each other and act synergistically. We make the following major observations:

First, both LPA and MLP-aware policies improve performance for *art, sphinx3, mcf, gcc, galgel* and *h264ref*. However, LPA outperforms the MLP-aware policy for most benchmarks. Especially, for *swim*, LPA improves performance by 2.3%

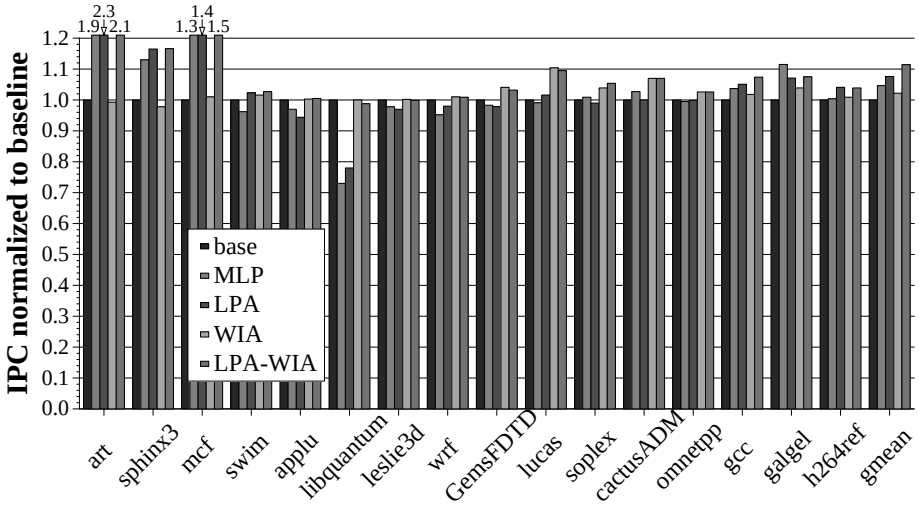



Figure 7.5: Performance of DRAM-aware replacement policies on single-core system

while the MLP-aware policy degrades performance by 3.8%. The reason why LPA outperforms the MLP-aware policy overall is that LPA is better at identifying and evicting low-cost lines that are serviced faster or in parallel in the DRAM system.

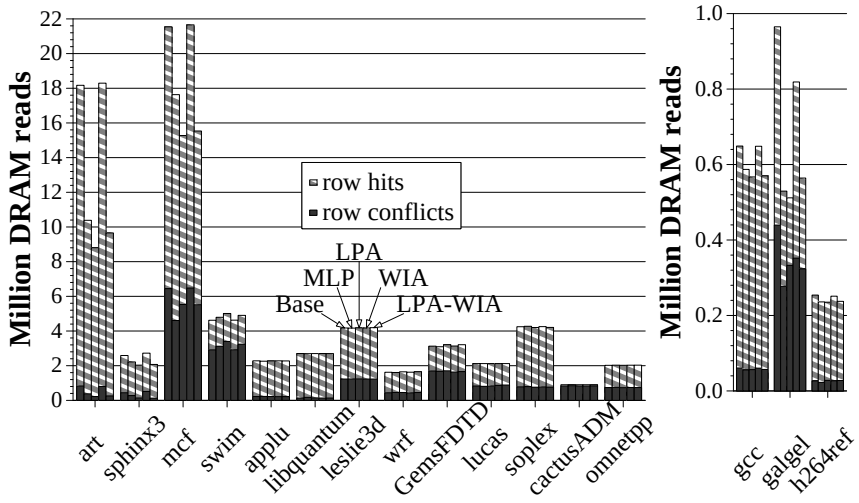
Second, both the LPA and MLP-aware policies degrade performance for *applu, libquantum, leslie3d, wrf,* and *GemsFDTD*. This is because neither of the two mechanisms are aware of write-caused interference when they evict dirty cache lines. This signifies the importance of write-caused interference when replacement decisions are made.

Third, the performance degradations due to LPA are recovered by employing WIA together with LPA. Additionally, WIA alone improves performance for *GemsFDTD, lucas, soplex, cactusADM*, and *omnetpp* mainly due to its ability to reduce write-caused interference in the DRAM system. As a result, using LPA and WIA (LPA-WIA) together provides the best performance among all policies.

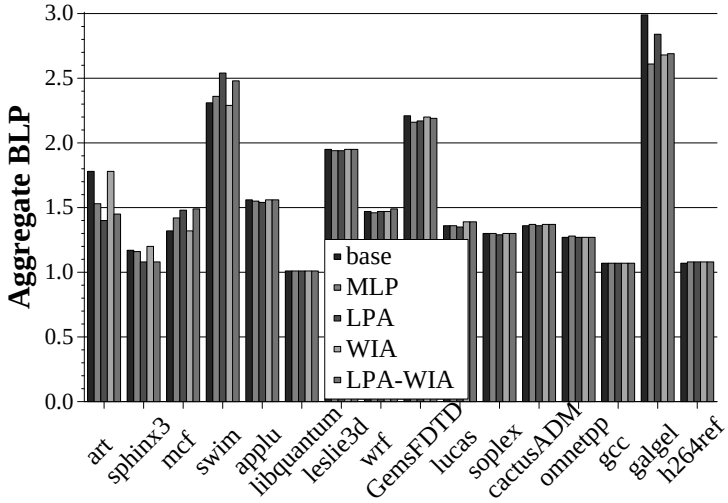
In the following subsections, we provide further insight using supporting data about DRAM characteristics.

#### 7.8.1.1 Why Does LPA Policy Perform Well?

Figure 7.6 shows the total read bus traffic (from DRAM to the processor) and aggregate DRAM BLP. Read traffic is essentially miss traffic and is divided into row hits and row conflicts. A good cache replacement policy would lead to less read traffic (i.e., fewer misses or higher cache locality), fewer row conflicts, and higher BLP.




(a) Traffic




(b) Aggregate BLP

Figure 7.6: DRAM read traffic and aggregate BLP of DRAM-aware replacement policies

LPA reduces row-conflict read traffic significantly for *art*, *sphinx3*, and *mcf*

(by 73.3%, 68.5%, and 14.2% compared to the baseline) in addition to reducing the overall read traffic as shown in Figure 7.6(a). This improves performance significantly for these applications. The MLP-aware policy also reduces read traffic, but much less so than LPA does.

LPA also increases BLP for *mcf* and *swim* by 12.3% and 10.0% compared to the baseline as shown in Figure 7.6(b). The increased BLP and reduced read traffic cause LPA to outperform the MLP-aware policy. The improved BLP due to LPA translates to performance improvement for *swim* even though LPA increases cache misses (total read traffic) by 8.5%. In contrast, the MLP-aware policy degrades performance of *swim* because many of the concurrent misses it estimates to be low-cost actually end up being high-cost row conflicts because they map to the same DRAM bank.

LPA significantly outperforms the MLP-aware policy in four applications: *art, sphinx3, mcf,* and *swim*. This is because the MLP-aware policy is not aware of DRAM banks and row buffer locality in the DRAM system. It relies on only the information about how many misses are outstanding at the same time, as discussed in Section 7.5. In contrast, our mechanism explicitly measures and estimates the BLP and row hit rate in the DRAM system to determine whether a line is likely to be low-cost when refetched later.

#### 7.8.1.2 Why Is Write-Caused Interference Awareness Desirable?

Both the MLP-aware and LPA policies degrade performance for *applu, libquantum, leslie3d, wrf,* and *GemsFDTD*, even though the read traffic (i.e., misses or row hits/row conflicts) or BLP does not change compared to the baseline, as shown in Figures 7.6(a) and (b). The reason for the degradation can be found by analyzing write traffic in Figure 7.7(a). Even though the total write traffic does not increase, LPA and MLP-aware increase the number of row-conflict writes compared to the baseline. This indicates that these policies increase write-caused interference, causing DRAM performance to degrade due to a large number of idle cycles on the

DRAM data bus. In fact, MLP-aware and LPA policies degrade *libquantum*'s performance by 27.0% and 22.0%.

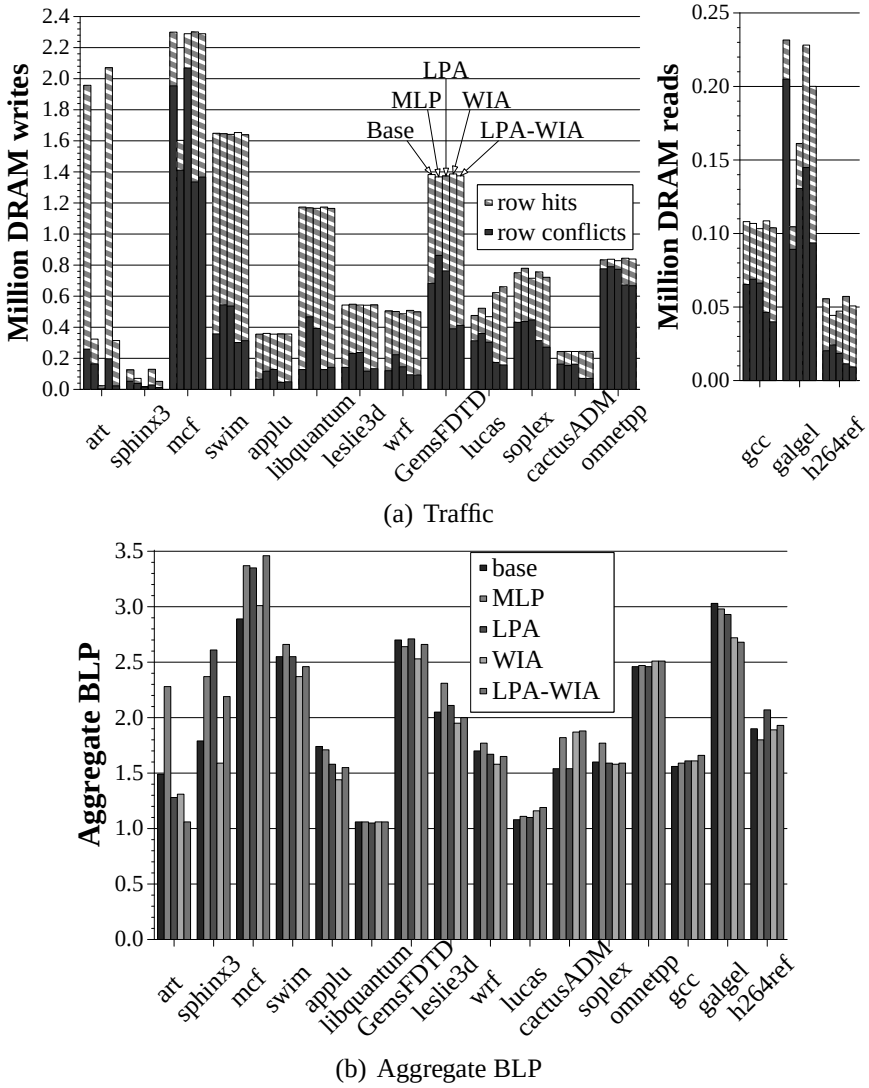



Figure 7.7: DRAM write traffic and aggregate BLP of DRAM-aware replacement policies

When employed with LPA, WIA reduces the number of row conflicts to as many as the baseline LRU for *applu*, *libquantum*, *leslie3d*, and *wrf* as shown in Figure 7.7(a). It also leads to fewer row conflicts than the baseline for *GemsFDTD*. Hence, by reducing write-caused interference when employed with LPA, WIA recovers the performance degradation due to LPA, and sometimes even improves per-

formance compared to the baseline (for *GemsFDTD* by 3.3%) as shown in Figure 7.5.

Additionally, WIA alone (without LPA) improves performance for *lucas*, *cactusADM*, *soplex*, and *omnetpp* by increasing row-hit writes (rather than row-conflict writes) compared to the baseline, thereby reducing write-caused interference in the DRAM system. Note that for *lucas*, *cactusADM* and *omnetpp*, WIA also increases aggregate BLP for writes, reducing their average latency cost.

On the other hand, the MLP-aware policy suffers performance degradation or cannot improve performance for these applications due to its unawareness of write-caused interference in the DRAM system.

#### 7.8.1.3 Combining LPA and WIA

We find that LPA and WIA are orthogonal to each other. When combined together in the way described in Section 7.3, the performance benefit of each mechanism is obtained additively. This can be justified by observing that improved DRAM characteristics for reads and writes of each individual mechanism in Figures 7.6 and 7.7 do not significantly change for LPA-WIA. We conclude that our DRAM-aware replacement policies largely reduce costly cache misses and evictions, thereby improving performance significantly on a single-core system.

#### 7.8.1.4 Effect on System with Prefetching

In this section, we discuss the DRAM-aware replacement policy in a system with prefetching. When the DRAM-aware policy is naively employed with prefetching, there are two issues that can affect its effectiveness. First, useful prefetches that are marked as low-cost by LPA can be evicted (just because they are marked as low-cost) from the last-level cache before it is used. This reduces the effectiveness of prefetching and therefore can hurt performance compared to the baseline LRU policy without LPA. Second, useless prefetches that are not marked (i.e., high cost prefetches) can stay in the cache for a long time consuming cache

space. This can reduce cache efficiency by evicting useful cache lines.

To overcome these problems, we take prefetch usefulness into account in LPA replacement decisions. The basic idea is 1) to ignore the low-cost bit of prefetches that are estimated as useful so that LPA does not evict low-cost useful prefetches that are not used yet and 2) to evict prefetches that are likely-useless earlier so that cache space can be used for demand and useful prefetches.

To implement this, we measure prefetch accuracy on an interval-basis as prefetch-aware DRAM controller and BLP-aware issue policies do as presented in Chapters 5 and 6. When the estimated prefetch accuracy from the previous interval is greater than a threshold, *useful_prefetch_threshold*, the low-cost bits of prefetched lines are disregarded by LPA in the current interval. Similarly, when the prefetch accuracy is less than another threshold, *useless_prefetch_threshold*, prefetched lines are prioritized over any other cache lines in the set considered for replacement. Note that prefetched lines are identified by examining the prefetch bit in each cache line, which is already used by the prefetch estimation (as explained in Section 5.2.1).

On the other hand, WIA is not required to be prefetch-aware. This is because writes are not immediately critical to an application's progress as we already discussed in Section 7.4.2. Writes become critical only when the DRAM controller cannot service demands and useful prefetches (i.e., reads) due to write-caused interference. Servicing many writes quickly so that reads can be serviced without interruption of writes for a long time leads to high performance.

Figure 7.8 shows the average performance of the baseline with no prefetching, the baseline prefetching, MLP-aware, and DRAM-aware replacement (LPA and WIA together) with these optimizations. We ran the 16 benchmarks on the single-core system with the stream prefetcher (used in Chapters 5 and 6). We empirically determined the two thresholds: *useful_prefetch_threshold* of 50% and *useless_prefetch_threshold* of 20%. The prefetch accuracy is measured every 100K processor cycles.

The DRAM-aware replacement policy improves performance by 8.2% com-

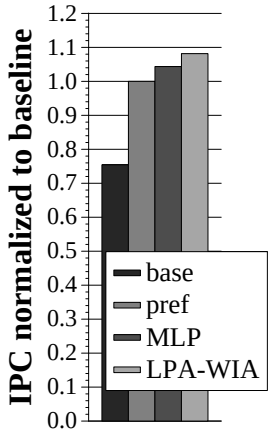


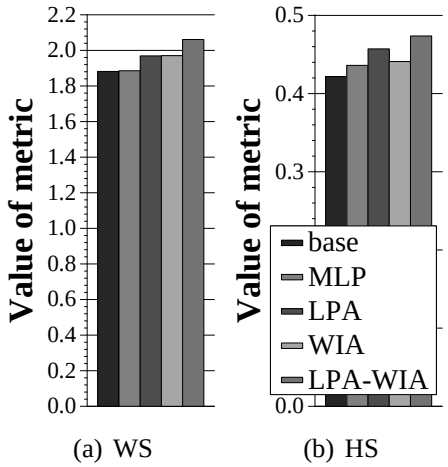

Figure 7.8: Performance of DRAM-aware replacement policies on single-core system with prefetching

pared to prefetching whereas the MLP-aware policy improves performance only by 4.4%. This is mainly because the MLP-aware policy is not aware of DRAM characteristics or prefetch usefulness. We conclude that DRAM-aware replacement is also effective in a system that employs prefetching.

### 7.8.2 4-Core Results

We evaluate our mechanisms on a 4-core system with a shared last-level cache in this section. Figure 7.9 shows average weighted speedup (WS) and harmonic mean of speedups (HS) for the baseline LRU, MLP-aware, LPA, WIA, and LPA-WIA replacement policies.

LPA alone improves both WS and HS by 4.6% and 8.4% compared to the baseline LRU by evicting low-cost lines while keeping high-cost lines for the application running on each core. WIA alone also significantly improves system performance (WS and HS by 4.7% and 4.6%) by servicing writes fast, and thereby reducing write-caused interference to more critical reads. When combined together, LPA and WIA improve WS and HS by 9.5% and 12.3%. On the other hand, the MLP-aware policy marginally improves only HS by 3.4%. Its performance benefit is insignificant mainly due to its unawareness of DRAM characteristics. We conclude that our DRAM-aware mechanisms are also very effective and improve




(a) WS (b) HS

Figure 7.9: Performance of DRAM-aware replacement policies on 4-core system

system performance significantly on multi-core systems.

## 7.9 Summary

This chapter makes a case for designing the last-level cache replacement policies in a manner that is aware of DRAM state and characteristics. Previous cache replacement policies overwhelmingly optimize for minimizing cache misses and ignore DRAM performance characteristics that affect the cost of each miss: row buffer locality, bank-level parallelism, and write-caused interference. We show that taking these DRAM performance characteristics into account in last-level cache replacement policies can significantly improve entire system performance. Our proposed policies estimate the performance cost of a cache miss/eviction in the DRAM system, and favor the eviction of the cache line that is estimated to have the least system performance impact. Our evaluations show that our proposed DRAM-aware cache replacement policies significantly improve performance on both single-core and multi-core systems.

# Chapter 8

# Last-Level Cache Management for Reducing Write-Caused Interference

In Chapter 7, we have proposed and discussed a cache replacement policy, Write-caused Interference-Aware (WIA) Replacement policy, which aims to reduce write-caused interference in the DRAM system. Recall that WIA evicts row-hit dirty lines that can be written back quickly to DRAM due to row buffer locality *only* when a replacement happens in the last-level cache. In this chapter, we propose a more aggressive writeback policy that proactively sends writebacks from the last-level cache even *before* a replacement happens, in order to further reduce write-caused interference in the DRAM system [38].

We first motivate the problem of write-caused interference in today and future DRAM systems in more detail, and then we discuss our baseline DRAM write buffer management policy that performs best among the existing write buffer policies by reducing read-to-write and write-to-read penalties. After that, we propose and evaluate our aggressive cache writeback mechanism which can further improve performance on top of the baseline write buffer policy.

## 8.1 Write-Caused Interference in the DRAM System

Read and write requests from the processor contend for DRAM data bus. In general, read requests (i.e., miss requests from the last-level cache) are critical for system performance since they are required for an application's progress whereas writes (i.e., writeback requests from the last-level cache) do not need to be performed immediately. In modern DDR (Double Data Rate)-based memory systems, write requests can interfere significantly with the servicing of read requests, degrad-

ing overall system performance by delaying the more critical read requests. There are two major sources of performance penalty when a write request is serviced instead of a read request. First, the critical read request is delayed for the duration of the service latency of the write request. Second, even after the write is serviced fully, the read cannot be started because the DDR DRAM protocol requires additional timing constraints to be satisfied which causes idle cycles on the DRAM data bus in which no data transfer can be done.

As discussed in Section 2.3, the two most important of these timing constraints are write-to-read ($t_{WTR}$) and write-to-precharge (write recovery, $t_{WR}$) latencies as specified in the current JEDEC DDR DRAM standard [22]. These timing constraints in addition to other DRAM latencies such as precharge, activate and column address strobe latencies ($t_{RP}, t_{RCD}$, and $CL/CWL$) dictate the number of cycles in which the DRAM data bus must remain idle after a write, before a read can be performed. Both latencies increase in terms of number of DRAM clock cycles as the bus clock frequency of the DRAM chip increases [67, 22] as do other DRAM latencies. The end result is that high penalties caused by write requests will become even larger in terms of number of cycles because the bus clock frequency of future DRAM chips will continue to increase to maintain high peak bandwidth.

An on-chip *write buffer* can mitigate this problem. A write buffer holds write requests on the chip until they are sent to DRAM according to the write buffer management policy. While write requests are held by the write buffer, read requests from the processor can be serviced by DRAM without interference from write requests. As a result, memory service time for reads that are required by the application can be reduced. As the write buffer size increases, write-caused interference in the memory system decreases. For example, an infinite write buffer can keep all write requests on-chip, thereby completely removing write-caused interference. However, a very large write buffer is not attractive since it requires high hardware cost and design complexity (especially to enable forwarding of data to matching read requests) and leads to inefficient utilization of on-chip hardware and power. In fact, a write buffer essentially acts as another level of cache (holding

only written-back cache lines) between the last-level cache and the main memory system.

### 8.1.1 Performance Impact of Write-Caused Interference in Today's DRAM System

To motivate the performance impact of write-caused interference, Figure 8.1 shows the simulated performance of a single-core system (with no prefetching) that employs a state-of-the-art DDR3-1600 DRAM system (12.8 GB/s peak bandwidth) [49] and a First Ready-First Come First Served (FR-FCFS) DRAM controller [66]. We evaluate four write request management policies: 1) a 64-entry write buffer with a management policy similar to previous proposals [40, 57, 68] which exposes writes (i.e., makes them visible) to the DRAM controller only when there is no pending read request or when the write buffer is full, and stops exposing writes when a read request arrives or when the write buffer is not full anymore (*service_at_no_read*), 2) a 64-entry write buffer with a policy that exposes all writes only when the write buffer is full and continues to expose all writes until the write buffer becomes empty (*drain_when_full*), 3) Write-caused Interference-Aware (WIA) replacement policy with *drain_when_full* (proposed in Chapter 7) and 4) ideally eliminating all writes assuming that there is no correctness issue (no_write). Ideally eliminating all writes removes all write-caused interference and therefore shows the upper bound on performance that can be obtained by handling write-caused interference intelligently. We chose 16 benchmarks among all SPEC2000/2006 CPU benchmarks that have at least 10% IPC (retired instruction per cycle) performance improvement compared to *drain_when_full* when all writes are ideally removed. The performance numbers are normalized to *drain_when_full*. The configuration of the system is identical to the baseline system presented in Section 7.6.

We make two main observations. First, the performance of *service_at_no_read* is worse than *drain_when_full*. This is because when a read arrives at the DRAM controller very soon after a write is serviced, a significant amount of write-caused penalty delays that read. This happens to all the benchmarks except for *lucas* where

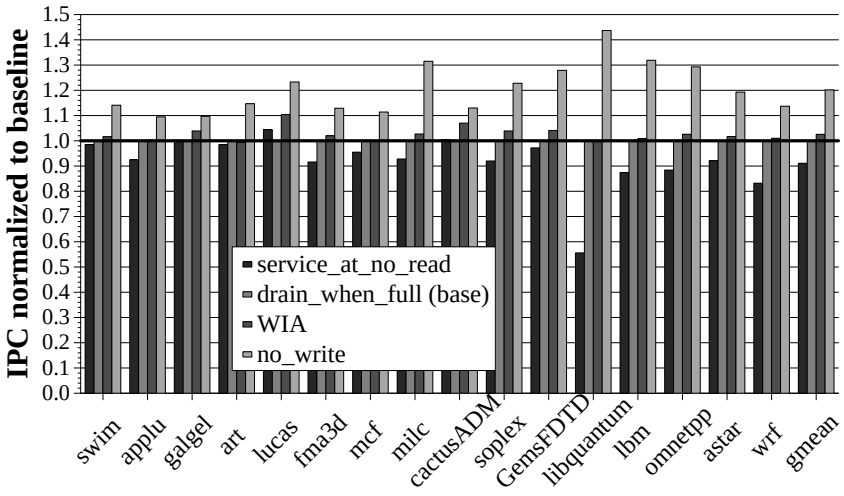



Figure 8.1: Potential (simulated) performance of intelligently handling write-caused interference in the DRAM system

there are long enough periods to satisfy the large write-caused penalties during which reads are not generated. Servicing writes opportunistically when there are no reads degrades performance due to two reasons: 1) it incurs the costly write-to-read and read-to-write switching penalties, thereby wasting DRAM bandwidth (i.e., incurring many idle cycles), 2) it does not exploit row buffer locality when servicing write requests since writes that go to the same row are serviced far apart from each other in time. In contrast, *drain_when_full* improves performance by 9.8% compared to *service_at_no_read* on average because it 1) delays service of writes as much as possible, 2) services all writes once it starts servicing one write, thereby amortizing write-to-read switching penalties across multiple writes by incurring them only once for an entire write-buffer worth of writes, and 3) increases the possibility of having more writes to the same DRAM row address or higher *row buffer locality* in the write buffer that is exploited by the DRAM controller for better DRAM throughput.

Second, even though *drain_when_full* improves performance compared to *service_at_no_read* and WIA outperforms *drain_when_full*, there is still large potential performance improvement (20.2% and 17.1% compared to *drain_when_full* and WIA respectively) that can be achieved by further reducing write-caused interfer-

ence, as shown by the rightmost set of bars.

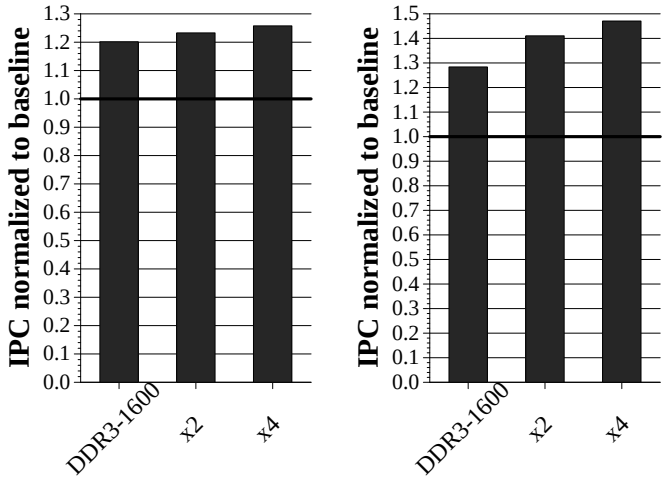
As shown above, the impact of write-caused interference on an application's performance is significant even with good write buffer/cache replacement policies with a decently-sized (i.e., 64-entry) write buffer. This is because a size-limited write buffer or a write buffer management policy cannot completely remove write-caused interference since 1) writes eventually have to be written back to DRAM whenever the write buffer is full and 2) servicing all writes in the write buffer still consumes a significant amount of time mainly due to the write-to-precharge penalties imposed to row-conflict writes to the same bank as discussed in 7.2.1. Note that the write-caused interference-aware replacement policy cannot remove all interference as well.

### 8.1.2 Performance Impact of Write-Caused Interference in the Future

We expect that write-caused interference will continually increase in terms of number of clock cycles as the bus clock frequency of the DRAM chip increases to maintain higher peak bandwidth. The write-to-read penalty which guarantees that modified data is written to the row buffer correctly (sense amplifier) will not be easily reduced in absolute time similar to other access latencies such as precharge period ($t_{RP}$) and column address strobe latency ($CL$/$CWL$). This is especially true for the write-to-precharge latency which guarantees modified data will be completely written back to the memory rows before a new precharge. This latency cannot easily be reduced because reducing access latency to the memory cell core is very difficult [67, 22]. We believe that this trend will continue to hold for any future memory technology (not limited to DRAM technology) that supports high peak bandwidth. This means that write-caused interference will continue to be a performance bottleneck in the future.

Figure 8.2 shows the performance improvement of the ideal writeback policy (i.e., all writes are removed) across future high bandwidth memory systems. We assume that the DRAM bus clock frequency continue to increase in the fu-

ture. Since the future memory specifications are unknown, we speculatively scaled the number of clock cycles for all DDR3-1600 performance-related latencies that cannot be easily reduced (e.g., $t_{WTR}$, $t_{WR}$, $t_{RP}$, $t_{RCD}$, $CL$, etc) in absolute time. For example, x2 of DDR3-1600 indicates a DDR system that maintains twice the DDR3-1600 peak bandwidth (25.6GB/s = 2 $\times$ 12.8GB/s). We also assume that the DRAM bus clock frequency increases as fast as the processor clock frequency. We show two cases: when no prefetching is employed and when the stream prefetcher (used in Chapters 5 and 6) is used in the processor.




(a) No prefetching (b) Prefetching

Figure 8.2: Performance potential by eliminating all writes as memory bus clock frequency increases

We make two observations from Figure 8.2. First, the higher the peak bandwidth, the larger the performance impact of write-caused interference. Second, removing write-caused interference is more critical for systems with prefetching. The performance impact of writes for the systems with prefetching is much higher due to larger contention between reads and writes (prefetch requests are all reads).

## 8.2 Motivation

### 8.2.1 Reducing Read-to-Write and Write-to-Read Penalties

As discussed in Section 2.3, read-to-write/write-to-read switching penalty is dictated by the read-to-write latency (latency from a read data burst to a write data burst, 2 DRAM clock cycles) and write-to-read latency ($t_{WTR}$, 6 DRAM clock cycles for DDR3-1600).

We demonstrate how these penalties can be mitigated by the *drain_when_full* policy with an example in Figure 8.3. Figure 8.3(a) shows the state of the DRAM read and write buffers. For brevity, we assume that each buffer has only two entries in this example. All the read requests in the DRAM read buffer are always exposed (or considered for scheduling) to the DRAM controller, whereas the writes are exposed based on the write buffer management policy. There is one read (Read A, a read request to Row A) and one write (Write B, a write to Row B) in the read and write buffers respectively. At time *t1*, another read (Read C) and a write (Write D) come from the last-level cache. We assume that each request goes to a different bank and that all requests hit the current open row in their corresponding DRAM banks (i.e., all requests are row hits).

Figure 8.3(b) shows the DRAM timing diagram for the policy which exposes writes to the DRAM controller only when there is no pending read request or when the write buffer is full and stops exposing writes when a read request comes in or when the write buffer is not full anymore (*service_at_no_read* in Section 8.1.1). Since no read is pending in the DRAM read buffer after Read A is scheduled, this policy schedules Write B from the write buffer. Subsequently Read C and Write D are scheduled.

Two observations are made. First, the command for Write B after Read A must satisfy read-to-write latency; it has to be scheduled by the DRAM controller at least $CL + t_{BL} + 2 - CWL$ DRAM clock cycles [22] after the read command is scheduled such that the write burst can be on the bus two DRAM cycles after the read burst (as discussed in Section 2.3). Second, Read C after Write B must satisfy

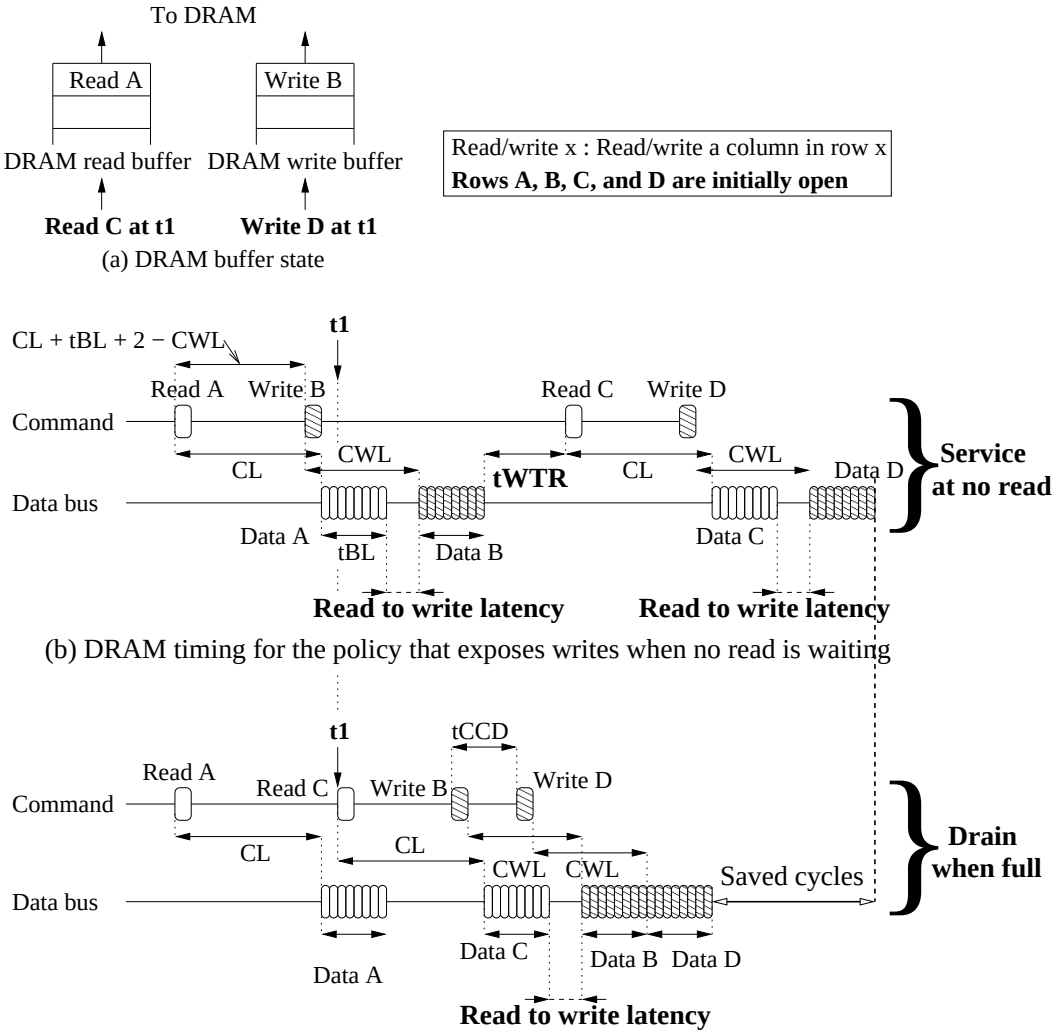



Figure 8.3: Service_at_no_read vs. drain_when_full write buffer policies

$t_{WTR}$. The command for Read C can only be scheduled $t_{WTR}$ cycles after the data burst for Write B is completed. In contrast to read-to-write latency, the data bus must be idle for $t_{WTR} + CL$ cycles since the subsequent read command cannot be scheduled for $t_{WTR}$ cycles. The last write is scheduled after read-to-write latency is satisfied as shown.

This policy results in many idle cycles (i.e., poor DRAM utilization) on the data bus. This is because it sends writes as soon as there are no pending reads which is problematic when a subsequent read arrives immediately after the write is scheduled to DRAM. The penalties introduced by the write cause a significant amount of interference and therefore increase both the read's and write's service time. This is the main reason why this policy does not perform well as shown in Figure 8.1.

On the other hand, if the write buffer policy that exposes all writes only when the write buffer is full and continues to expose all writes until the write buffer becomes empty (*drain_when_full*) is used, Reads A and C are serviced first (Write B is not serviced immediately after Read A since the write buffer is not full) and then Writes B and D are serviced. Figure 8.3(c) shows the DRAM timing diagram for this policy. Read C can be scheduled once the DRAM controller sees it since there is no unsatisfied timing constraint for Read C. Then Write B can be scheduled $CL + t_{BL} + 2 - CWL$ cycles after the command for Read A is scheduled. Note that the command for Write D can be scheduled very soon (more precisely, $t_{CCD}$ cycles after the command for Write B) since DDR DRAM chips support back-to-back data bursts for writes by overlapping column address strobe latencies ($CWL$) as we discussed in Chapter 2.

This policy results in better DRAM service time for the four requests compared to the policy shown in Figure 8.3(b). Since buffering writes in the DRAM write buffer and servicing all of them together when the buffer becomes full reduces the large read-to-write and write-to-read latency penalties, DRAM throughput increases. Also note that by delaying writes as much as possible, reads that are

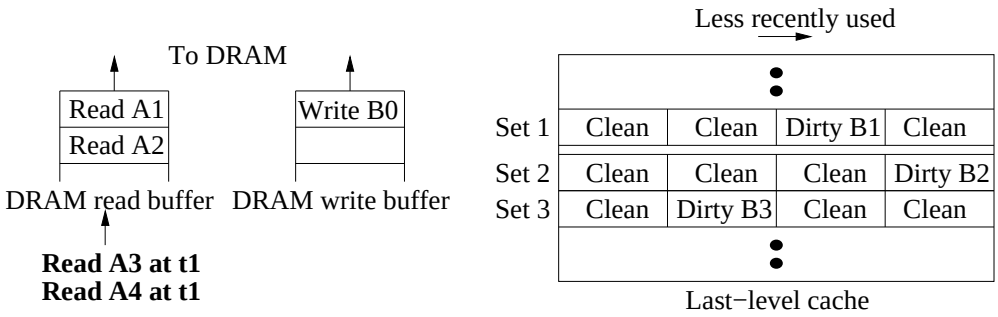
more critical to an application's progress can be serviced quickly thereby improving performance. This is the main reason this policy outperforms the *drain_when_full* policy as shown in Figure 8.1. We found that this policy is the best among the previously proposed write buffer policies we evaluated (as shown Section 8.7.1). We use this policy as our baseline write buffer policy.

### 8.2.2 Last-Level Cache Writeback: A Way to Further Reduce Write-Caused Interference

As discussed in Section 2.3.2, write-to-precharge penalty cannot be reduced by write buffer policies (such as drain when full). Servicing row-conflict writes in the same bank takes a significant number of cycles. This delays service of writes in the write buffer and eventually results in delaying service of reads. Service of writes can be done faster if the write buffer has many row-hit writes. The source of DRAM writes is the last-level cache's writebacks which are dirty line evictions in a writeback cache. To leverage this fact, we have already proposed the Write-caused Interference-Aware (WIA) replacement policy in Chapter 7. The WIA policy evicts row-hit dirty lines that can be written back fast to DRAM due to row buffer locality when a replacement happens in the last-level cache. Since this policy generates row-hit writes *only* when a replacement happens in a cache set, it looses opportunities that more row-hit dirty lines in other cache sets can be written back fast. Therefore overall reduction in write-caused interference can be small.

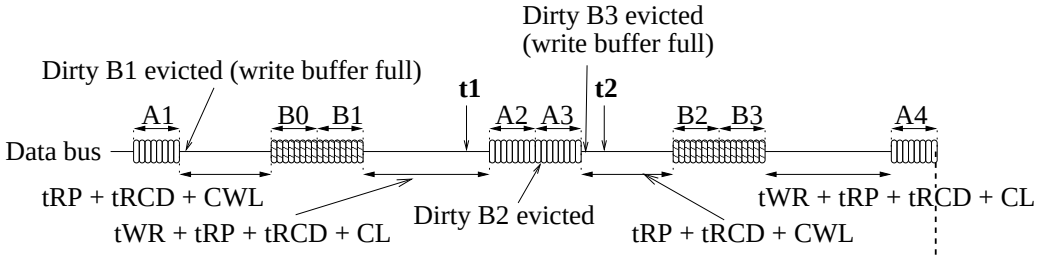
The last-level cache can more aggressively and proactively send out write-backs that can be written fast even *before* a line is evicted to improve service time of writes.

Figure 8.4 compares an aggressive writeback policy of the last-level cache to the WIA replacement policy. Figure 8.4(a) shows the initial state of the DRAM read/write buffers and three sets of the last-level cache. Two reads (Reads A1 and A2, both to Row 0) and a write (Write B0 to Row 1) are waiting to be scheduled in the DRAM read and write buffers (two entries for each) respectively. In each of the three cache sets shown, there is a dirty line that is mapped to the same row (Row 1)



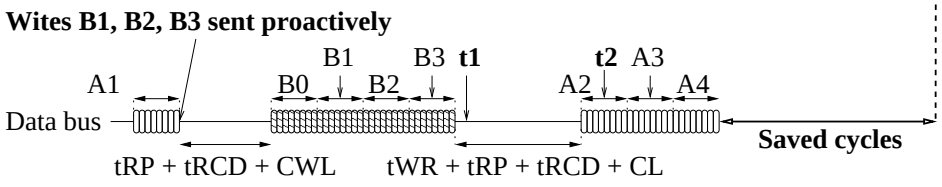

| |
|---|
| Reads A1, A2, A3, A4 are mapped to row 0 in bank 0 |
| Writes and dirties B0, B1, B2, B3 are mapped to row 1 in bank 0 |
| Rows 1 in bank 0 is initially open |
| Data for read A1 is stored in set 1 |
| Data for read A2 is stored in set 2 |
| Data for read A3 is stored in set 3 |

(a) Cache/DRAM buffer initial state




| **Scheduling order** |
|---|
| Read A1 (row hit), Write B0 (row conflict), Write B1 (row hit), Read A2 (row conflict), Read A3 (row hit), Write B2 (row conflict), Write B3 (row hit), Read A4 (row conflict) |

(b) DRAM timing for write−caused interference−aware replacement




| **Scheduling order** |
|---|
| Read A1 (row hit), Write B0 (row conflict), Write B1 (row hit), Write B2 (row hit), Write B3 (row hit), Read A2 (row conflict), Read A3 (row hit), Read A4 (row hit) |

(c) DRAM timing for DRAM−aware writeback: writebacks B1, B2, B3 are sent proactively

Figure 8.4: Write-cause interference-aware replacement vs. DRAM-aware write-back

as Write B. At time *t1*, a new read to Row 0 (Read A3) comes from the last-level cache. At time *t2*, another read to Row 0 (Read A4) comes as well. We assume that Reads A1, A2, and A3 are inserted to Sets 1, 2, and 3 in the cache respectively when serviced by DRAM. We also assume that all (read and write) requests are mapped to the same bank (Bank 0) for simplicity. Both policies employ the *drain_when_full* write buffer policy.

Figure 8.4(b) shows the resulting DRAM timing when the WIA replacement policy is used. Read A1 is serviced by DRAM first since the write buffer is not full. When Read A1 is serviced and inserted to Set 1, WIA evicts Dirty B1 since it is mapped to the same row as Write B0 in the write buffer (i.e., row hit). Therefore the DRAM write buffer becomes full, and Writes B0 and B1 are serviced back-to-back next after the row-conflict latency ($t_{RP} + t_{RCD} + CWL$). After this, Reads A2 and Read A3 are serviced. Read A2 must wait for a longer row-conflict latency ($t_{WR} + t_{RP} + t_{RCD} + CL$) since its precharge command must wait until write-to-precharge latency ($t_{WR}$) is satisfied after the write burst of Write B1. Read A3 is serviced right after Read A2 since it is a row hit read. The evicted dirty lines (Writes B2 and B3) due to Data A2 and A3's insertion are written back after the row-conflict latency ($t_{RP} + t_{RCD} + CWL$). Finally, Read A4 is serviced after another longer row-conflict latency. This policy results in idle cycles of two smaller row-conflict (row conflict after a read) latencies and two larger row-conflict (row conflict after a write) latencies.

On the other hand, as shown in Figure 8.4(c), if the writeback for Dirties B1, B2, and B3 in the cache can be sent out before Read A is completely serviced by DRAM, all writes are serviced back-to-back $t_{RP}+t_{RCD}+CWL$ DRAM cycles after Read A1's data burst. Reads A2, A3, and A4 are serviced back-to-back $t_{WR}+t_{RP}+t_{RCD}+CL$ after the write burst of Write B3. This policy results in idle cycles of one smaller row conflict and one larger row conflict. Since more writes are serviced back-to-back, the aggressive writeback policy can results in fewer idle DRAM bus cycles than the WIA policy. Servicing more writes quickly also results in higher

performance since subsequent reads can be serviced without being interfered by writes for a long time.

## 8.3 Mechanism: DRAM-Aware Writeback

Our mechanism, DRAM-aware writeback, aims to maximize the DRAM throughput for write requests in order to minimize write-caused interference. It monitors dirty cache lines (writebacks) that are evicted from the last-level cache and tries to find other dirty cache lines that are mapped to the same row as the evicted line. When found, the mechanism aggressively sends writebacks for those dirty cache lines to DRAM. The *drain_when_full* write buffer policy allows writes to be seen by the DRAM controller when the write buffer is full thereby allowing the DRAM controller to exploit row buffer locality of writes. The writeback mechanism only cleans (does not evict) cache lines by sending writebacks.

The mechanism consists of a global writeback monitor unit and a state machine in each last-level cache bank as shown in Figure 8.5. The writeback monitor unit watches evicted cache lines until it sees a dirty cache line being evicted. When it finds one, it records the row address of the cache line in each cache bank's state machine. Once a write's row address is recorded, the state machines start sending out writebacks for dirty lines whose row address is the same as the recorded row address (row-hit dirty lines). To find row-hit dirty cache lines, each state machine searches its cache bank. Each state machine shares the port of its cache bank with the demand cache accesses from the lower-level cache. Since the demand accesses are more critical to performance, they are prioritized over the state machine's accesses. Once a row-hit dirty line is found, the line's writeback is sent out through the conventional writeback ports regardless of the LRU position of the cache line. Because the cache lines which are written back in this manner may be reused later, the cache lines stay in the cache and only have their dirty bit reset (they become non-dirty or clean). The state machine in each core continues sending row-hit writebacks until all possible sets that may include cache lines whose row address is the same as

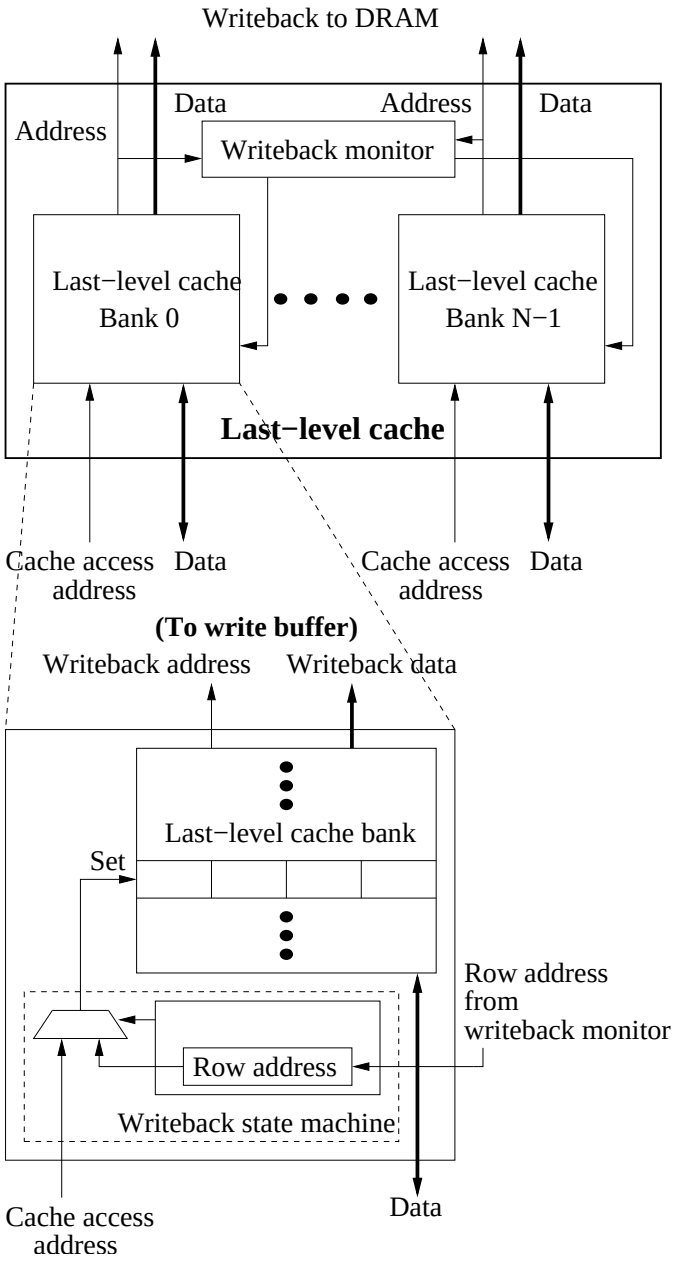



Figure 8.5: Writeback mechanism in last-level cache

the recorded row address have been checked. When all state machines in the banks finish searching, the writeback monitor unit starts observing the writebacks coming out of the cache to start another set of DRAM-aware writebacks.

The DRAM-aware writeback technique leverages the benefits of the write buffer and the baseline write buffer management policy (*drain_when_full*). It can send more row-hit writebacks than the number of write buffer entries within a very short time. In fact, a single dirty line eviction can trigger our mechanism to send up to row_size / cache_line_size writebacks. Once the write buffer becomes full, all state machines stall and delay the current searching. At the same time, the underlying *drain_when_full* write buffer management policy starts exposing the writes since the write buffer is full. As the DRAM controller services writes, free write buffer entries become available for new writebacks. The state machine resumes searching and sending row-hit writes to the write buffer. Because the *drain_when_full* policy keeps exposing writes until the write buffer becomes empty, all possible row-hit writebacks for a row can be serviced quickly by the DRAM controller since they are all row-hits. In this way, our mechanism can effectively enable more writes to be serviced quickly, which in turn reduces the number of write buffer drains over the entire run of an application. This results in fewer write-to-read switching penalties which improves DRAM throughput and performance.

Note that two conditions should be true for the DRAM-aware writeback to be effective. First, the last-level cache banks should have enough idle cycles for the state machine to look for row-hit writes. If this is true, the mechanism would not significantly contend with demand accesses from the lower-level caches for the cache bank and will be able to generate many row-hit writebacks. Second, rewrites to cache lines which our mechanism proactively writes back to DRAM should not occur too frequently. If rewrites happen too frequently, the mechanism significantly increases the number of writes to DRAM. Even though row-hit writes can be serviced quickly, the increased writes might increase time spent in servicing writes. We discuss these two issues in the following sections.

### 8.3.1 Does Last-Level Cache Have Sufficient Bandwidth for DRAM-Aware Writeback?

Table 8.1 shows the percent of last-level cache bank idle cycles (averaged over all banks) over the entire run for each of the 16 SPEC2000/20006 benchmarks in a single core system described in Section 8.5. For all benchmarks, except *art*, cache bank idle time is more than 95%.

| Benchmark | swim | applu | galgel | art | lucas | fma3d | mcf | milc | cactusADM |
|---|---|---|---|---|---|---|---|---|---|
| Idle cycles (%) | 0.96 | 0.97 | 0.92 | 0.91 | 0.98 | 0.97 | 0.97 | 0.97 | 0.99 |

| Benchmark | soplex | GemsFDTD | libquantum | lbm | omnetpp | astar | wrf |
|---|---|---|---|---|---|---|---|
| Idle cycles (%) | 0.98 | 0.97 | 0.97 | 0.95 | 0.98 | 0.98 | 0.98 |

Table 8.1: Last-level cache bank idle cycles (%) on single core system

Table 8.2 shows the average idle bank cycles of the last-level cache (shared cache for multi-core systems) of the single, 4, and 8-core systems described in Section 8.5. Even in multi-core systems, the shared last-level cache has many idle cycles. This is because last-level cache accesses are not too frequent compared to lower-level caches, since the lower-level caches and Miss Status Holding/Information Registers (MSHRs) filter out many accesses from the last-level cache. Therefore, we expect contention between demands and our DRAM-aware writeback accesses to be insignificant. We find that prioritizing demands over the accesses for DRAM-aware writeback is enough to reduce the impact of using the cache banks for our mechanism.

| | 1-core | 4-core | 8-core |
|---|---|---|---|
| Idle cycles (%) | 0.97 | 0.91 | 0.89 |

Table 8.2: Average last-level cache bank idle cycles (%) on single, 4, and 8-core systems

### 8.3.2 Dynamic Optimization for Frequent Rewrites

For applications that exploit temporal locality of the last-level caches, the cache lines which are written back by our aggressive writeback policy may be

rewritten by subsequent dirty line evictions of the lower-level cache. These *redirtied* cache lines may come to be written back to DRAM again by the last-level cache's replacement policy or the DRAM-aware writeback policy. This will increase the number of writebacks (i.e., writes to DRAM) which may hurt performance by delaying service of reads due to frequent services for writes.

We mitigate this problem using a simple optimization. We periodically estimate the rewrite rate of cache lines whose writebacks are sent out by the DRAM-aware writeback mechanism. Based on this estimation, our mechanism dynamically adjusts its aggressiveness. For instance, when the rewrite rate is high, the mechanism sends out only row-hit writebacks close to the LRU position. When the rewrite rate is low, the mechanism can send out even row-hit writebacks close to the MRU position. Since the estimation of rewrite rate is periodically done, the DRAM-aware writeback mechanism can adapt to the phase behavior of an application as well. When employing this optimization in the shared cache of a multi-core system, we adapt the mechanism to estimate the rewrite rate for each core (or application).

To implement this, each cache line keeps track of which core it belongs to using core ID bits and also tracks whether the cache line becomes clean (or non-dirty) due to the DRAM-aware writeback mechanism using an additional bit for each line. A counter for each core periodically tracks the total number of the core's writebacks sent out by the DRAM-aware writeback mechanism. Another counter counts the number of the core's rewrites to the clean cache lines whose writebacks were sent early by our mechanism. The rewrite rate for each core for an interval is calculated by dividing the number of rewrites by the total number of writebacks sent out in that interval. The estimated rewrite rate is stored in a register for each core and used to determine how aggressively the mechanism sends writebacks (from LRU or from other positions close to MRU) for the next interval.

We found that our mechanism without this optimization slightly degrades performance for only two applications (*vpr* and *twolf*, both of which are memory non-intensive) out of all 55 SPEC2000/2006 benchmarks by increasing the number of writebacks. Therefore the gain from this optimization is small compared to

design effort and hardware cost. We analyze this optimization with experimental results in detail in the results section (Section 8.7.2).

## 8.4 Comparison to Eager Writeback

Eager writeback [40] was proposed to make efficient use of idle bus cycle for writes in a Rambus DRAM system in order to minimize read and write contention. It sends writebacks for dirty LRU lines in a cache set to the write buffer when the set is accessed by a demand request. Writes in the write buffer are scheduled when the bus is idle. There are important key differences between eager writeback and our DRAM-aware writeback technique which we discuss below.

First, eager writeback is not aware of DRAM characteristics. We find that simply sending writebacks for dirty LRU cache lines does not work with today's high-frequency DDR DRAM systems because servicing those writes in DRAM is not necessarily completed quickly. For instance, servicing row-conflict writes causes large penalties (write-to-precharge latencies) as shown in Section 7.2.1. This eventually significantly delays the service of subsequent reads.

Second, the write-caused penalties of state-of-the-art DDR DRAM systems are too large to send a write only because the data bus is idle or there are no pending reads. To tolerate the large write-caused penalties, there must be no read request arriving at the DRAM system for a long time such that all write-caused timing constraints are satisfied before the subsequent read. However, for memory intensive applications whose working set does not fit in the last-level cache, it is very likely that read requests arrive at the DRAM system before all constraints are satisfied. Therefore subsequent reads suffer large write-to-read penalties.

In contrast, our mechanism does not aim to minimize immediate write-caused interference but targets minimizing the write-caused penalties for the entire run of an application. It allows to stop servicing current reads to service writes. However, once it does, it makes the DRAM controller service many writes fast by exploiting row buffer locality such that servicing writes next time can be performed

a long time later.

We extensively analyze and compare DRAM-aware writeback and eager writeback in Section 8.7.

## 8.5 Experimental Methodology

### 8.5.1 Metrics

To measure multi-core system performance, we use Individual Speedup (IS), Weighted Speedup (WS), and Harmonic mean of Speedups (HS), which are defined in Section 5.3.1.

### 8.5.2 System Model

The baseline configuration of processing cores and the memory system for single, 4, and 8-core CMP systems is shown in Table 8.3 (identical to the model in Chapter 7). The DDR3 DRAM performance-related timing constraints are the same as in Table 7.2 in Chapter 7.

To evaluate the effectiveness of our mechanism in systems with prefetching, we employ the aggressive stream prefetcher (32 streams, prefetch degree of 4, prefetch distance of 64 cache lines) in Chapters 5 and 6 for each core.

### 8.5.3 Workloads

We use the same methodology for compiling and running the SPEC workloads using ICC/IFORT and Pinpoints as discussed in Sections 5.3.3, 6.3.3 and 7.6.3.

We evaluate 18 SPEC benchmarks on the single-core system. The 16 benchmarks (which have at least 10% ideal performance improvement when all writes are removed) discussed in Section 8.1.1 and the two benchmarks, *vpr* and *twolf* mentioned in Section 8.3.2. The characteristics of the 18 SPEC benchmarks are shown in Table 8.4. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We

| | |
|---|---|
| Execution Core | Out of order, decode/retire up to 4 instructions,<br>issue/execute up to 8 microinstructions; 15 stages<br>256-entry reorder buffer; |
| Front End | Fetch up to 2 branches; 4K-entry BTB;<br>64-entry return address stack;<br>64K-entry gshare/PAs hybrid branch predictor |
| Caches and on-chip buffers | L1 I/D-cache: 32KB, 4-way, 2-cycle, 64B line size;<br>Shared last-level cache: 16-way, 8-bank, 15-cycle,<br>1 read/write port per bank, LRU replacement<br>writeback, 64B line size, 1, 2, 4MB for 1, 4 and 8-core systems;<br>32, 128, 256-entry MSHRs & LLC access/miss/fill buffers,<br>for 1, 4 and 8-core systems |
| DRAM and bus | 1, 2, 2 channels (DRAM controllers) for 1, 4, 8-core systems;<br>800MHz DRAM bus cycle,<br>Double Data Rate (DDR3 1600MHz) [49];<br>8B-wide data bus per channel, BL = 8;<br>1 rank, 8 banks per channel, 8KB row buffer per bank; |
| DRAM controllers | On-chip, open-row, FR-FCFS scheduling policy [66];<br>64-entry ($8 \times 8$ banks) DRAM read and write buffers per channel<br>drain_when_full write buffer policy |

Table 8.3: Baseline configuration for DRAM-aware writeback

ran 30 and 12 randomly chosen workload combinations for our 4 and 8-core CMP configurations respectively.

## 8.6 Implementation and Hardware Cost of DRAM-Aware Writeback

As shown in Figure 8.5, our DRAM-aware writeback mechanism requires a simple state machine in each last-level cache bank and a monitor unit. Most of the hardware cost is in logic modifications. For example, the comparator structure should be modified to support tag comparison with the row address in each state machine. The only noticeable storage cost is eight bytes per cache bank for storing the row address of the recent writeback. Note that none of the last-level cache structure we modify is on the critical path. As Tables 8.1 and 8.2 show, the accesses to the last-level cache are not very frequent.

| | | | Reads | | Writes | |
|---|---|---|---|---|---|---|
| Benchmark | Type | IPC | MPKI | RHR(%) | WPKI | RHR(%) |
| 171.swim | FP00 | 0.35 | 23.10 | 36.95 | 8.24 | 78.33 |
| 173.applu | FP00 | 0.93 | 11.40 | 90.34 | 1.78 | 81.34 |
| 175.vpr | IN00 | 1.02 | 0.89 | 16.11 | 0.27 | 25.67 |
| 178.galgel | FP00 | 1.42 | 4.84 | 54.45 | 1.16 | 11.51 |
| 179.art | FP00 | 0.26 | 90.92 | 95.43 | 9.79 | 86.75 |
| 189.lucas | FP00 | 0.61 | 10.61 | 61.00 | 2.38 | 34.19 |
| 191.fma3d | FP00 | 1.01 | 4.13 | 74.75 | 1.82 | 70.58 |
| 300.twolf | INT00 | 0.98 | 0.72 | 38.49 | 20.82 | 20.82 |
| 429.mcf | INT06 | 0.15 | 33.64 | 18.36 | 10.69 | 16.58 |
| 433.milc | FP06 | 0.48 | 29.33 | 90.78 | 5.19 | 48.26 |
| 436.cactusADM | FP06 | 0.63 | 4.51 | 7.42 | 1.22 | 33.31 |
| 450.soplex | FP06 | 0.40 | 21.24 | 81.64 | 3.75 | 42.48 |
| 459.GemsFDTD | FP06 | 0.49 | 15.63 | 45.81 | 6.91 | 50.60 |
| 462.libquantum | INT06 | 0.67 | 13.51 | 94.96 | 5.87 | 89.13 |
| 470.lbm | FP06 | 0.46 | 20.16 | 66.67 | 10.42 | 66.42 |
| 471.omnetpp | INT06 | 0.49 | 10.11 | 63.45 | 4.17 | 6.88 |
| 473.astar | INT06 | 0.47 | 10.19 | 55.16 | 3.80 | 8.96 |
| 481.wrf | FP06 | 0.72 | 8.11 | 72.95 | 2.52 | 76.17 |

Table 8.4: Characteristics for 18 SPEC benchmarks for DRAM-aware writeback: IPC, MPKI (last-level cache misses per 1K instructions), WPKI (last-level cache Writebacks Per 1K Instructions), DRAM row hit rate (RHR)

If we implement the optimization in Section 8.3.2, one additional bit and core ID bits (for multi-core systems) for each cache line are required. Three registers (2 bytes for each) are required to keep track of the number of writebacks sent, the number of rewrites, and the rewrite rate.

## 8.7 Experimental Evaluation

We first show that the baseline write buffer management policy that we use outperforms other policies and then we evaluate and analyze our proposed DRAM-aware writeback mechanism on the single and multi-core systems.

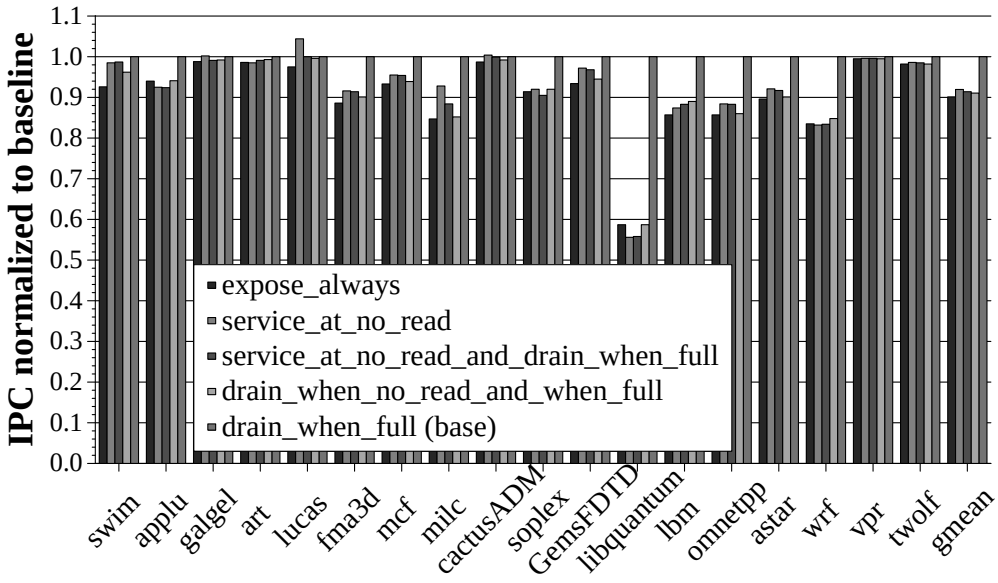
### 8.7.1 Performance of Write Buffer Management Policies

In addition to our baseline (*drain_when_full*), we evaluate four other write buffer management policies that are all based on the same principle as previous

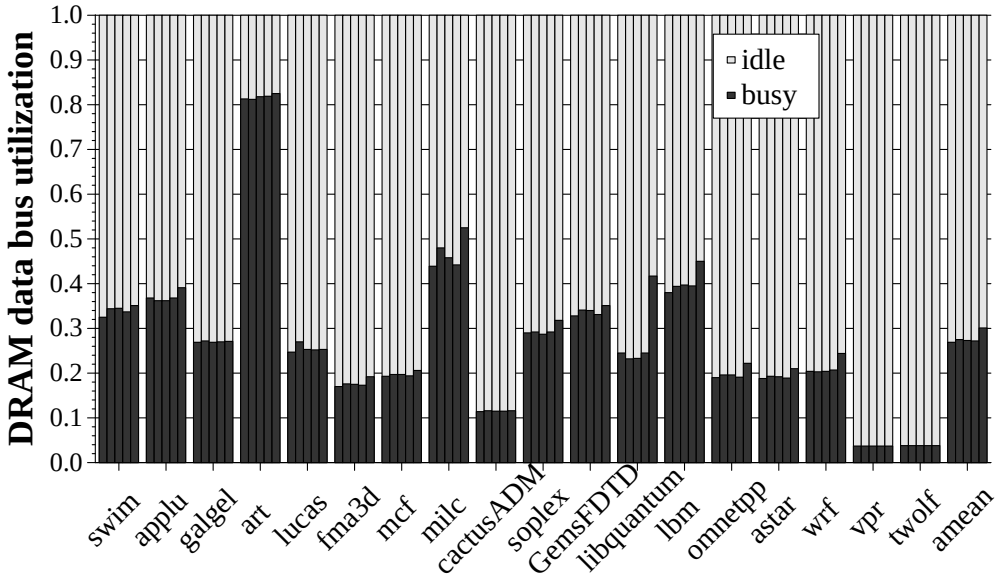
work [40, 57, 68]. The first one, *expose_always*, is a policy that always exposes DRAM writes and reads to the DRAM controller together. The DRAM controller makes scheduling decisions based on the baseline FR-FCFS scheduling policy while always prioritizing reads over writes. However, if all DRAM timing constraints are satisfied for a write, the write can be scheduled even though there are reads in the read request buffer. For example, while a precharge for a read is in progress in one bank, a row-hit write in a different bank can be scheduled and serviced if all timing constraints for the write are satisfied (assuming there is no pending read to the corresponding bank). The second policy is *service_at_no_read* which was discussed in Section 8.1.1. This policy exposes writes to the DRAM controller only when there is no pending read request or when the write buffer is full, and stops exposing writes when a read request arrives or when the write buffer is not full any more. The third policy is *service_at_no_read_and_drain_when_full* which is the same as service_at_no_read except that once the write buffer is full, all writes are exposed until the buffer becomes empty. The fourth policy, *drain_when_no_read_and_when_full* is the same as our baseline policy that exposes all writes and drains the buffer every time the write buffer is full, except that it also keeps exposing all writes until the buffer becomes empty even when writes are exposed due to no pending read in the read request buffer. The DRAM controller follows the FR-FCFS policy to schedule reads and exposed writes for all of the above policies.

Figure 8.6 shows IPC normalized to the baseline and DRAM data bus utilization on the single-core system for the above five write buffer policies. DRAM bus utilization is calculated by dividing the number of cycles the data bus transfers data (both reads and writes) by the number of total execution cycles. Note that since we only change the write buffer policy, the total number of reads and writes does not change significantly among the five policies. Therefore, we can meaningfully compare the DRAM data bus utilization of each policy as shown in Figure 8.6(b). A large number of busy cycles indicates high DRAM throughput. On the other hand, a larger number of idle cycles indicates more interference among memory requests.

Our baseline *drain_when_full* policy outperforms the other four polices sig-




(a) Performance




(b) Data bus utilization

Figure 8.6: Performance and DRAM bus utilization of various write buffer policies

nificantly for almost all benchmarks. The other policies cause many idle cycles due to frequent read-to-write and write-to-read switching as shown in Figure 8.6(b). The *expose_always* policy performs worst since writes are always exposed and can be scheduled more freely than other policies by the DRAM controller, hence the most read-to-write and write-to-read penalties. The *service_at_no_read_and_drain_when_full* and *drain_when_no_read_and_when_full* policies also cause some writes to be scheduled when there is no read in the read buffer. This results in many write-to-read switching penalties (i.e., many idle cycles) since reads usually arrive at the read buffer soon after writes are scheduled for most of the benchmarks shown.

In contrast, the *drain_when_full* policy increases data bus utilization by allowing the DRAM controller to service reads without interference from writes as much as possible. It also reduces write-to-read switching penalties overall because only one write-to-read switching penalty (also one read-to-write penalty) is needed to drain all the writes from the write buffer. Finally it also gives more chances to the DRAM controller to exploit better row buffer locality and bank-level parallelism (servicing writes to different DRAM banks concurrently, if possible) by exposing more writes together. To summarize, the *drain_when_full* policy improves performance by 8.8% on average and increases data bus utilization by 9.4% on average compared to the best of the other four policies (*service_at_no_read*).

Note that there is still a significant number of idle bus cycles in Figure 8.6(b) even with the best policy. Our DRAM-aware writeback mechanism aims to minimize write-caused interference so that idle cycles are better utilized.

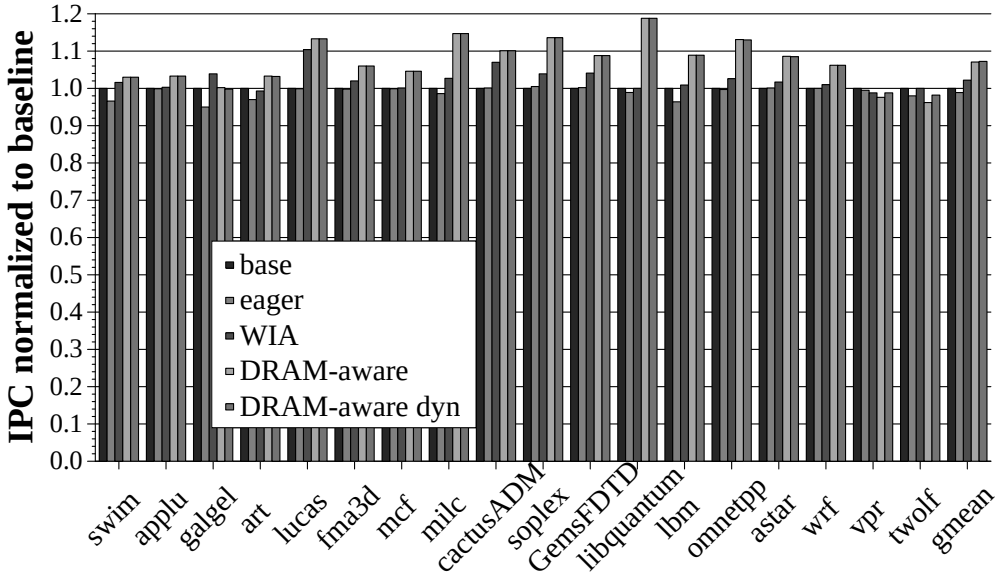
### 8.7.2 Single-Core Results

This section presents performance evaluation of the DRAM-aware writeback mechanism on the single-core system. Figure 8.7 shows IPC normalized to the baseline *drain_when_full* policy and DRAM data bus utilization for eager writeback technique, Write-caused Interference-aware (WIA) replacement (proposed in Chapter 7), DRAM-aware writeback, and DRAM-aware writeback with the opti-

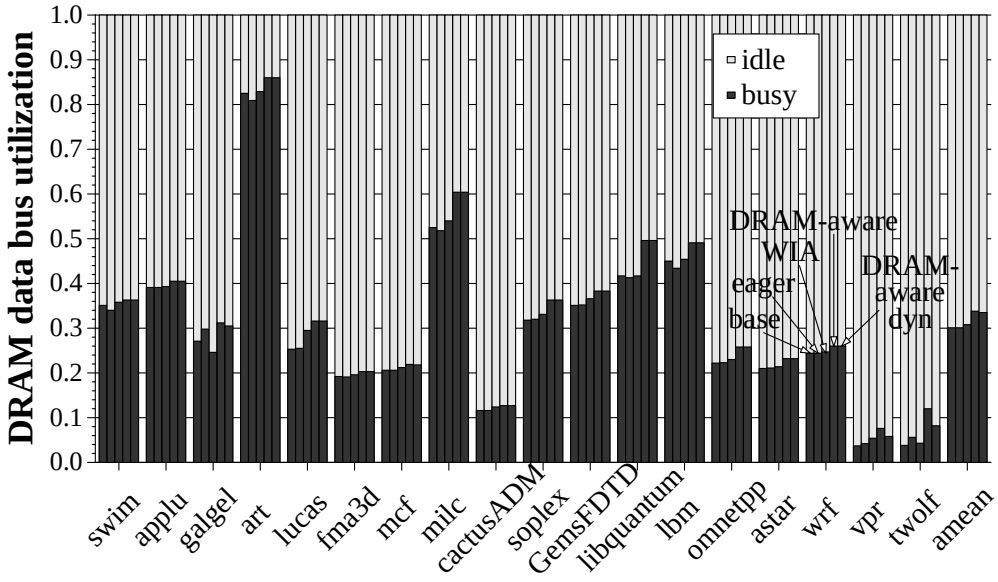
mization described in Section 8.3.2. The optimization dynamically adjusts the dirty line LRU positions which are considered for writeback based on the rewrite rate estimation. When the rewrite rate is less than 50%, we allow any LRU position which generates a row-hit to be written back. If the rewrite rate is between 50% and 90%, only the least recently used half of the LRU stack can be sent out. If the rewrite rate is more than 90%, only writebacks in the LRU position can be sent out. Note that the eager writeback mechanism uses a write buffer policy that sends writes when the bus is idle as discussed in Section 8.4. In Section 8.7.1, we showed that sending out writes when the bus is idle is inferior to draining the write buffer only when it is full (*drain_when_full*). As such, for fair comparison we use an improved version of eager writeback that uses the baseline *drain_when_full* policy. First we make the following major performance-related observations from Figure 8.7, and then provide more insights and supporting data using other DRAM and last-level cache statistics in subsections.

First, the eager writeback technique degrades performance by 1.1% compared to the baseline. This is mainly because it is not aware of DRAM characteristics. Filling the write buffers with writebacks for dirty lines which are in the LRU position of their respective sets does not guarantee fast service time of writes since servicing row-conflict writes must pay the large write-to-precharge penalties. As shown in Figure 8.7(b), eager writeback suffers as many idle cycles as the baseline on average.

Second, DRAM-aware writeback improves performance for all benchmarks except for *vpr* and *twolf*. It improves performance by more than 10% for *lucas, milc, cactusADM, libquantum* and *omnetpp*. This is because our mechanism sends many row-hit writes that are serviced quickly by the DRAM controller, which in turn reduces write-to-read switching penalties. As shown in Figure 8.7(b), our mechanism improves DRAM bus utilization by 12.3% on average across all 18 benchmarks. Increased bus utilization translates to high performance. On average, the mechanism improves performance by 7.1%. However, the increased bus utilization does not increase performance for *vpr* and *twolf*. In fact, the mechanism degrades per-




(a) Performance




(b) Data bus utilization

Figure 8.7: Performance and DRAM bus utilization of DRAM-aware writeback on single-core system

formance for these two benchmarks by 2.4% and 3.8% respectively. This is due to the large number of writebacks that are generated by the DRAM-aware writeback mechanism for these two benchmarks. We developed a dynamic optimization as presented in Section 8.3.2 to mitigate this degradation, which we refer to as dynamic DRAM-aware writeback.

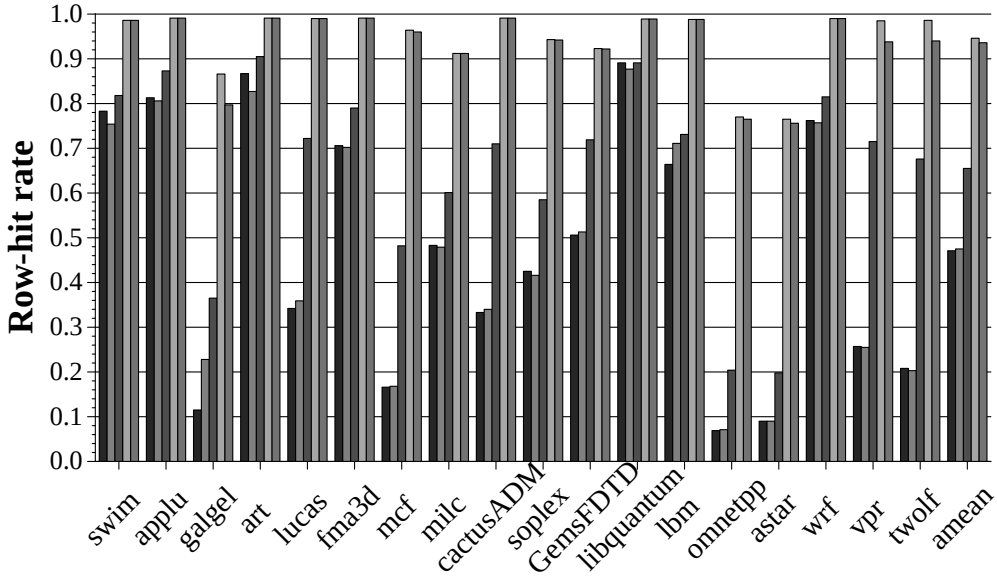
Third, dynamic DRAM-aware writeback mitigates the performance degradation for *vpr* and *twolf* by selectively sending writebacks based on the rewrite rate of DRAM-aware writebacks. By doing so, the performance degradation of *vpr* and *twolf* becomes 1.2% and 1.8% respectively, which results in 7.2% average performance improvement for all 18 benchmarks. Note that the dynamic mechanism still achieves almost all of the performance benefits of non-dynamic DRAM-aware writeback for the other 16 benchmarks. As we discussed in Section 8.3.2, the gain from this optimization is small compared to design effort and hardware cost.

Finally, our DRAM-aware writeback policies significantly outperform the WIA replacement policy. WIA improves the performance of the baseline only by 2.2%. This is mainly because WIA looses opportunities that more row-hit dirty lines can be written back fast, since it writebacks only when a replacement occurs. Our mechanism reduces more write-caused interference in the DRAM system thereby better utilizing DRAM bus.

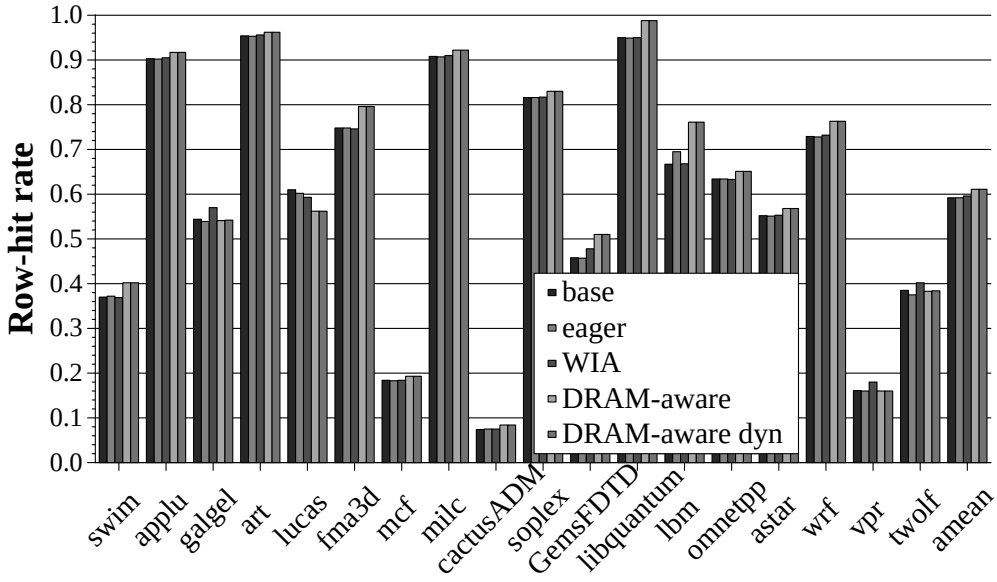
#### 8.7.2.1 Why Does Eager Writeback Not Perform Well?

As discussed above, eager writeback degrades performance compared to the baseline in today's DDR DRAM systems since it generates writebacks in a DRAM-unaware manner. In other words, it can fill the write buffer with many row-conflict writes. Figure 8.8 shows the row-hit rate for write and read requests serviced by DRAM for the 18 benchmarks. Because we use the open-row policy (that does not use either auto precharge or manual precharge after each access), row-conflict rate can be calculated by subtracting row-hit rate from one.

While eager writeback does not change row-hit rates for reads as shown




(a) Writes




(b) Reads

Figure 8.8: Row hit rate of DRAM writes and reads for DRAM-aware writeback

in Figure 8.8(b), it generates more row-conflict writes (fewer row-hits) for *swim, art, milc,* and *libquantum* compared to the baseline as shown in Figure 8.8(a). For these benchmarks, these row-conflict writes introduce many idle cycles during the servicing of writes with the baseline *drain_when_full* write buffer policy as shown in Figure 8.7(b). This increases the time to drain the write buffer which in turn delays the service of critical reads required for an applications' progress.

#### 8.7.2.2 Why Does DRAM-Aware Writeback Perform Better?

In contrast to eager writeback, DRAM-aware writeback selectively sends many row-hit writes that are serviced quickly by the DRAM controller. Therefore the row-hit rate for writes significantly increases (to 94.6% on average) as shown in Figure 8.8(a). Note that it also increases the row-hit rate for reads (by 3.3% on average) as shown in Figure 8.8(b). This is mainly because DRAM-aware writeback reduces row-conflicts between reads and writes as well by reducing write-to-read switching occurrences. We found that due to the last-level cache and row locality of programs, it is very unlikely that while servicing reads to a row, a dirty cache line to that row is evicted from the cache. Therefore decreased write-to-read switching frequency reduces row-conflicts between writes and reads for the entire run of an application.

DRAM-aware writeback leverages the benefits of the write buffer and the *drain_when_full* write buffer policy as discussed in Section 8.3. Once the mechanism starts sending all possible row-hit writebacks for a row, the write buffer becomes full very quickly. The *drain_when_full* write buffer policy continues to expose writes until the buffer becomes empty. This makes it possible for the DRAM controller to service all possible writes to a row very quickly. Therefore our mechanism reduces the total number of write buffer drains over the entire run of an application. Table 8.5 provides the evidence of such behavior. It shows the total number of write buffer drains and the average number of writes per write buffer drain for each benchmark. The number of writes per write buffer drain for DRAM-aware writeback is increased significantly compared to the baseline, eager writeback, and

WIA. Therefore the total number of drains is significantly reduced, which indicates that DRAM-aware writeback reduces write-to-read switching frequency thereby increasing row hit rate for reads as well. The increased row hits (i.e., reduced row conflicts) lead to high data bus utilization for both reads and writes and performance improvement as shown in Figure 8.7.

| | Benchmark | swim | applu | galgel | art | lucas | fma3d | mcf |
|---|---|---|---|---|---|---|---|---|
| drains | base | 64960 | 24784 | 2891 | 83870 | 19890 | 24625 | 62521 |
| | eager | 76660 | 26367 | 4264 | 90020 | 22096 | 25263 | 62938 |
| | WIA | 95356 | 26758 | 5681 | 104271 | 50783 | 24106 | 69574 |
| | DRAM-aware | 13642 | 2927 | 8043 | 16754 | 7677 | 2995 | 49915 |
| writes/drain | base | 25.38 | 14.36 | 80.11 | 23.34 | 23.93 | 14.79 | 34.19 |
| | eager | 21.52 | 13.51 | 97.86 | 24.29 | 22.47 | 14.43 | 34.09 |
| | WIA | 17.34 | 13.33 | 40.16 | 19.86 | 12.27 | 15.15 | 31.37 |
| | DRAM-aware | 121.90 | 121.97 | 50.19 | 128.26 | 96.24 | 122.09 | 45.05 |

| | Benchmark | milc | cactusADM | soplex | GemsFDTD | libquantum |
|---|---|---|---|---|---|---|
| drains | base | 50764 | 15264 | 43967 | 49027 | 115563 |
| | eager | 52581 | 15243 | 43033 | 50805 | 114461 |
| | WIA | 63305 | 18093 | 46992 | 63179 | 115561 |
| | DRAM-aware | 47982 | 2142 | 17611 | 14023 | 12535 |
| writes/drain | base | 20.43 | 15.99 | 17.07 | 28.21 | 10.16 |
| | eager | 19.75 | 16.05 | 17.53 | 27.34 | 10.26 |
| | WIA | 16.51 | 13.52 | 16.09 | 21.95 | 10.16 |
| | DRAM-aware | 21.83 | 114.27 | 44.32 | 99.49 | 93.66 |

| | Benchmark | lbm | omnetpp | astar | wrf | vpr | twolf |
|---|---|---|---|---|---|---|---|
| drains | base | 92310 | 35902 | 26377 | 38353 | 1961 | 4785 |
| | eager | 94396 | 36425 | 26859 | 38622 | 2732 | 8080 |
| | WIA | 94519 | 37455 | 27119 | 42492 | 4165 | 5493 |
| | DRAM-aware | 24630 | 44413 | 29836 | 4921 | 4346 | 9030 |
| writes/drain | base | 22.57 | 23.22 | 28.78 | 13.16 | 27.54 | 21.18 |
| | eager | 22.19 | 23.24 | 28.48 | 13.08 | 29.72 | 27.15 |
| | WIA | 22.04 | 22.54 | 28.15 | 11.97 | 27.97 | 20.99 |
| | DRAM-aware | 85.08 | 20.50 | 27.05 | 103.26 | 69.91 | 71.88 |

Table 8.5: Number of write buffer drains and number of writes per drain for various policies

#### 8.7.2.3 When is Dynamic DRAM-Aware Writeback Required?

Recall that DRAM-aware writeback degrades performance for *vpr* and *twolf*. Figure 8.9 shows the total number of DRAM read and write requests serviced by DRAM for the 18 benchmarks. While DRAM-aware writeback does not increase

the total number of reads and writes significantly for the other 16 benchmarks as the baseline and eager writeback do, it does increase the number of writes significantly for *vpr* and *twolf*.

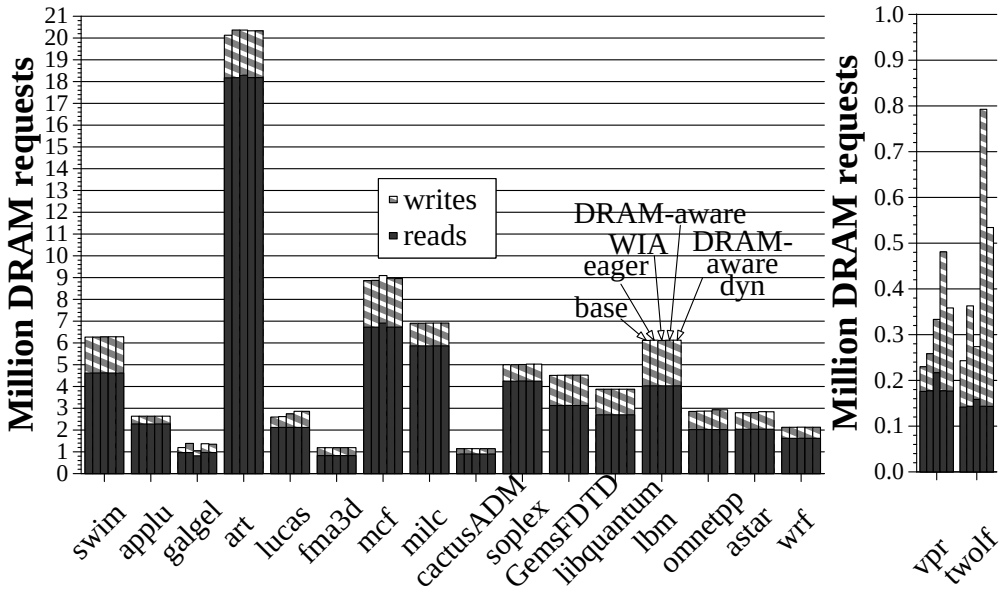



Figure 8.9: Number of DRAM requests for DRAM-aware writeback

Table 8.6 shows the total number of writebacks generated by DRAM-aware writeback, cache lines that were cleaned but reread, and cache lines that were cleaned but rewritten. It also shows the number of rewrites per cache line written back (referred to as rewrite rate). For *vpr* and *twolf*, rewrites to cache lines cleaned by the mechanism happen very frequently (82% and 85% respectively). These rewritten lines' writebacks are sent again by the mechanism thereby increasing the number of writes significantly. The increased writes make the write buffer full frequently, therefore aggregate write-to-read switching penalty becomes larger, which degrades performance. However, the performance degradation is not significant for these two benchmarks, because the total number of requests is not large (i.e., memory non-intensive) as shown in Figure 8.9. .

The dynamic DRAM-aware writeback mechanism discussed in Section 8.3.2 mitigates this problem by adaptively limiting writebacks based on rewrite rate estimation. Since the rewrite rate is high most of the time for *vpr* and *twolf*, the dynamic

| Benchmark | swim | applu | galgel | art | lucas | fma3d | mcf |
|---|---|---|---|---|---|---|---|
| Writebacks | 1640260 | 350641 | 346550 | 2061007 | 731063 | 361590 | 2167616 |
| Reread | 42 | 183 | 23741 | 70931 | 0 | 0 | 122290 |
| Rewritten | 20 | 0 | 166871 | 191596 | 0 | 501 | 108871 |
| Rewrite Rate | 0.00 | 0.00 | 0.48 | 0.09 | 0.00 | 0.00 | 0.05 |

| Benchmark | milc | cactusADM | soplex | GemsFDTD | libquantum | lbm |
|---|---|---|---|---|---|---|
| Writebacks | 947328 | 242377 | 732556 | 1251832 | 1161287 | 2069208 |
| Reread | 0 | 16 | 1599 | 1905 | 0 | 0 |
| Rewritten | 0 | 55 | 28593 | 13474 | 0 | 0 |
| Rewrite Rate | 0.00 | 0.00 | 0.04 | 0.01 | 0.00 | 0.00 |

| Benchmark | omnetpp | astar | wrf | vpr | twolf |
|---|---|---|---|---|---|
| Writebacks | 698896 | 612423 | 500963 | 299262 | 639582 |
| Reread | 21982 | 6012 | 746 | 12479 | 24230 |
| Rewritten | 73667 | 37075 | 2588 | 245645 | 540604 |
| Rewrite Rate | 0.11 | 0.06 | 0.01 | 0.82 | 0.85 |

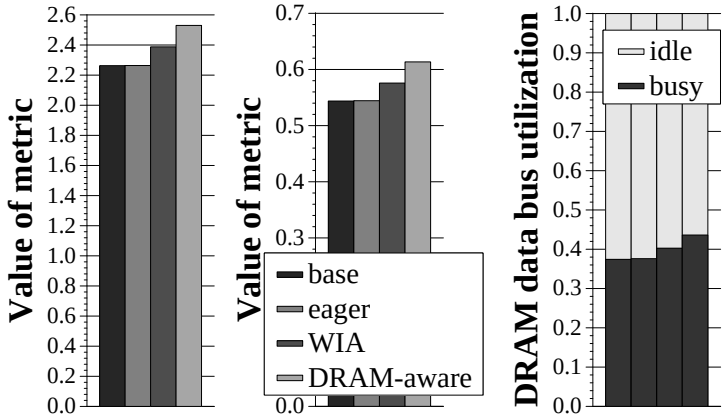
Table 8.6: Number of DRAM-aware writebacks generated, reread cache lines and rewritten cache lines, and rewrite rate

mechanism allows writebacks only for row-hit dirty lines which are in the LRU position of their respective sets. Therefore, it reduces the number of writebacks as shown in Figure 8.9. In this way, it mitigates the performance degradation for these two benchmarks as shown in Figure 8.7. Note that the dynamic mechanism does not change the benefits of DRAM-aware writeback for the other 16 benchmarks since it adapts itself to the rewrite behavior of the applications.

### 8.7.3 Multi-Core Results

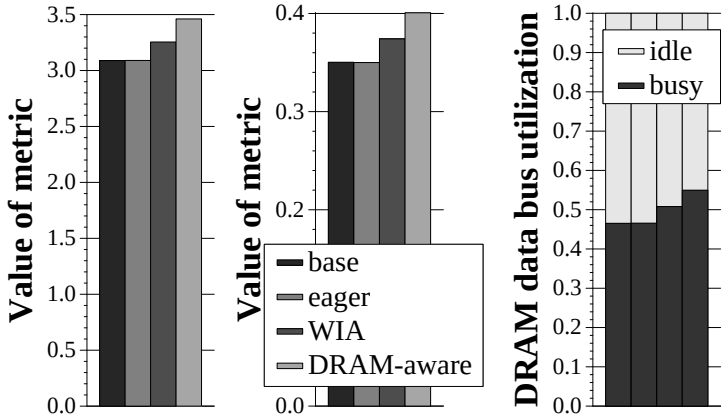
We also evaluate the DRAM-aware writeback mechanism on multi-core systems. Figures 8.10 and 8.11 show average system performance and bus utilization for the 4 and 8-core systems described in Section 8.5. In multi-core systems, write-caused interference is more severe since there is greater contention between reads and writes from multiple cores in the DRAM system. Furthermore, writes can delay critical reads of all cores. As such, reducing write-caused interference is even more important in multi-core systems. Our DRAM-aware writeback mechanism increases bus utilization by 16.5% and 18.1% for the 4 and 8-core systems respectively. This leads to an increase in weighted speedup (WS) and harmonic mean of

speedups (HS) by 11.8% and 12.8% for the 4-core system and by 12.0% and 14.4% for the 8-core system. We conclude that DRAM-aware writeback is effective for multi-core systems.




(a) WS (b) HS (c) Bus utilization

Figure 8.10: Performance of DRAM-aware writeback on 4-core system



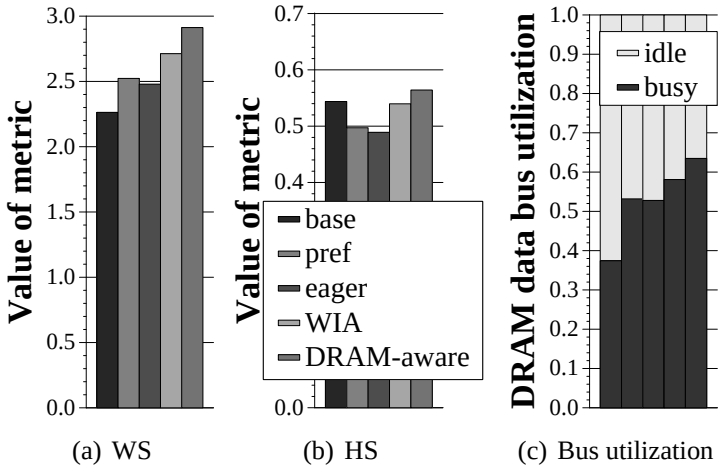

(a) WS (b) HS (c) Bus utilization

Figure 8.11: Performance of DRAM-aware writeback on 8-core system

### 8.7.4 Effect on Systems with Prefetching

We evaluate DRAM-aware writeback when it is employed in a 4-core system with the aggressive stream prefetcher described in Section 8.5. Figure 8.12 shows average system performance and bus utilization for the baseline with the

baseline with no prefetching, the baseline with prefetching, eager writeback, Write-caused Interference-Aware (WIA) replacement and DRAM-aware writeback for our 30 4-core workloads. All three mechanisms are employed on the baseline with prefetching.




(a) WS (b) HS (c) Bus utilization

Figure 8.12: Performance of DRAM-aware writeback on 4-core system with prefetching

Prefetching increases write-caused interference severely. Prefetch requests, which are essentially reads, put more pressure on the DRAM system. Prefetching improves weighted speedup by 11.5% by utilizing idle DRAM bus cycles while it degrades harmonic speedup by 8.6% compared to the baseline with no prefetching. Eager writeback suffers performance degradation (WS and HS by 1.7% and 1.6% ) compared to prefetching alone mainly due to its DRAM-unawareness. In contrast, WIA improves WS and HS by 7.5% and 8.5% compared to prefetching by reducing write-caused interference (increasing DRAM bus utilization by 9.3%). Using DRAM-aware writeback significantly improves DRAM bus utilization (by 19.4% compared to prefetching) by further reducing write-caused interference. The increased bus utilization turns into higher performance. DRAM-aware writeback performs best by improving WS and HS by 15.4% and 13.5%. We conclude that DRAM-aware writeback is also effective in multi-core systems that employ prefetching.

## 8.8 Summary

This chapter describes the problem of write-caused interference in today's DRAM systems, and shows it has significant performance impact in modern processors. Write-caused interference will continue to be a performance bottleneck in the future because the memory system's bus clock frequency continues to increase in order to provide more memory bandwidth. To reduce write-caused interference, we propose a new writeback policy for the last-level cache, called DRAM-aware writeback, which aggressively sends out writebacks for dirty lines that can be quickly written back to DRAM by exploiting row buffer locality. We demonstrate that the proposed mechanism and the previous best write buffer management policy are synergistic in that they work together to reduce write-caused interference by allowing the DRAM controller to service many writes quickly together. This reduces the delays incurred by read requests and therefore increases performance significantly in both single-core and multi-core systems. We also show that the performance benefits of the mechanism increases in multi-core systems and systems with prefetching where there is higher contention between reads and writes in the DRAM system. We conclude that DRAM-aware writeback can be a simple solution to reduce write-caused interference.

# Chapter 9

# Combining All DRAM-Aware Mechanisms

This chapter discusses and evaluates the performance of all proposed DRAM-aware mechanisms when they are employed together on single, 4, and 8-core systems.

## 9.1 DRAM-Aware Mechanisms Are Complementary

The Prefetch-Aware DRAM Controller (PADC) in Chapter 5 manages the DRAM request buffers to maximize row buffer locality for useful prefetches and demand requests. The BLP-aware request issue policies in Chapter 6 manage the issue order to Miss Status/Information Holding Registers (MSHRs) and to DRAM requests buffers to maximize the BLP of each application (or core).

On the other hand, the DRAM-aware replacement policy in Chapter 7 changes the mixture of memory read and write requests from the last-level cache to improve all three DRAM characteristics. It consists of Latency and Parallelism-Aware Replacement (LPA) and Write-caused Interference-Aware (WIA) replacement policies. LPA evicts lines that would take advantage of row buffer locality or BLP when they are refetched later. WIA evicts dirty lines that can be written back quickly by exploiting row buffer locality. The DRAM-aware writeback in Chapter 8 also changes the mixture of write requests to further reduce write-caused interference in the DRAM system.

Since each of the four mechanisms manages a different on-chip memory structure/policy to improve DRAM performance, they are orthogonal to one another except for the WIA replacement and DRAM-aware writeback. The objectives of these two mechanisms are identical, which is to reduce write-caused inter-

ference. In Chapter 8, we compared these two mechanisms and showed that the DRAM-aware writeback outperforms WIA by reducing more write-caused interference. However, these two mechanisms are partially complementary when they are combined. This is because WIA can be helpful for sending more row-hit writes quickly. If a replacement occurs while the writebacks for a row are sent out by DRAM-aware writeback, it is likely that WIA will evict a dirty line that is mapped to the same row if found. As a result, writes can be sent out faster than the DRAM-aware writeback alone. Therefore the combined mechanisms we evaluate in this chapter include both the WIA replacement and DRAM-aware writeback mechanisms.

## 9.2 Methodology

### 9.2.1 System Model

Table 9.1 shows the baseline system configuration used for performance evaluations when all DRAM-aware mechanisms are combined together on the same system. The DRAM timing constraints we modeled are identical to the DDR3-1600 constraints presented in Section 7.2.

### 9.2.2 Workloads

We use the same methodology for compiling and running the SPEC workloads as in Section 5.3.3. We evaluated the 20 most memory intensive or prefetch-sensitive (either prefetch-friendly or prefetch-unfriendly) SPEC 2000/2006 benchmarks on the single-core system. To evaluate our mechanism on CMP systems, we formed new combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran 30 and 20 pseudo-randomly chosen workload combinations for our 4 and 8-core CMP configurations respectively. We imposed the requirement that each of the multiprogrammed workloads has at least one memory intensive application since these applications are most relevant to our study.

| | |
|---|---|
| Execution Core | 4.8 GHz, out of order, decode/retire up to 4 instructions,<br>issue/execute up to 8 microinstructions; 15 stages<br>256-entry reorder buffer; |
| Front End | Fetch up to 2 branches; 4K-entry BTB;<br>64-entry return address stack;<br>64K-entry gshare/PAs hybrid branch predictor |
| Caches and on-chip buffers | L1 I/D-cache: 32KB, 4-way, 2-cycle, 64B line size;<br>Shared last-level cache: 16-way, 8-bank, 15-cycle,<br>1 read/write port per bank, LRU replacement<br>writeback, 64B line size, 1, 2, 4MB for 1, 4 and 8-core systems;<br>32-entry MSHRs per core & LLC access/miss/fill buffers,<br>for 1, 4 and 8-core systems |
| Prefetcher | Stream prefetcher per core: 32 stream entries,<br>prefetch degree of 4, prefetch distance of 64 [77, 73],<br>128-entry prefetch request buffer per core |
| DRAM and bus | 1, 2, 2 channels (DRAM controllers) for 1, 4, 8-core systems;<br>800MHz DRAM bus cycle,<br>Double Data Rate (DDR3 1600MHz) [49];<br>8B-wide data bus per channel, BL = 8;<br>1 rank, 8 banks per channel, 8KB row buffer per bank; |
| DRAM controllers | On-chip, open-row, demand-first FR-FCFS scheduling policy [66];<br>64-entry ($8 \times 8$ banks) DRAM read and write buffers per channel<br>drain_when_full write buffer policy |

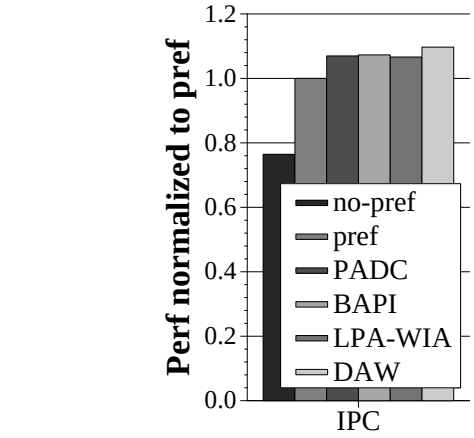
Table 9.1: Baseline configuration for all combined DRAM-aware mechanisms

## 9.3 Experimental Evaluation

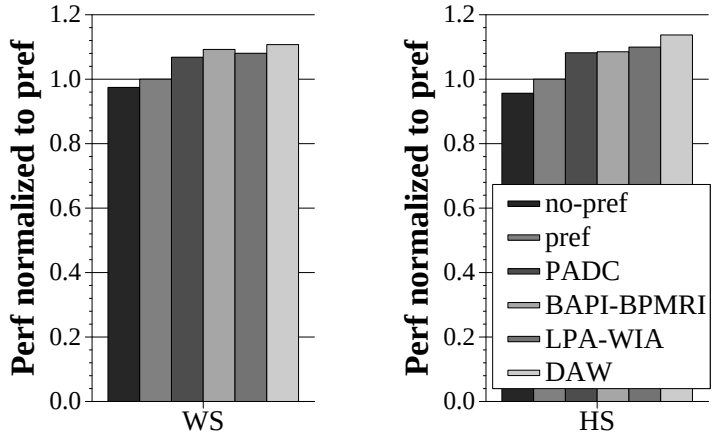
To show that our mechanisms are complementary, we first evaluate performance when each mechanism is employed alone on the systems shown in Section 9.2.1. Figure 9.1 shows the performance of no prefetching, prefetching, PADC, BLP-aware issue policies (BAPI-BPMRI), DRAM-aware replacement (LPA-WIA), and DRAM-Aware Writeback (DAW) on the single, 4, and 8-core systems. The performance numbers are normalized to the baseline prefetching.

As shown in Figures 9.1(a), (b), and (c), each of the mechanisms alone significantly improves performance. Each mechanism improves all performance metrics more than 6.0% on all systems.

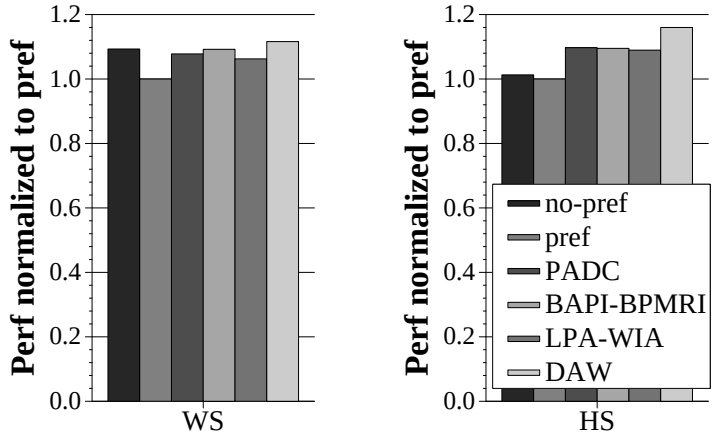
Figure 9.2 shows the average performance and DRAM data utilization of no




(a) Single-core processor

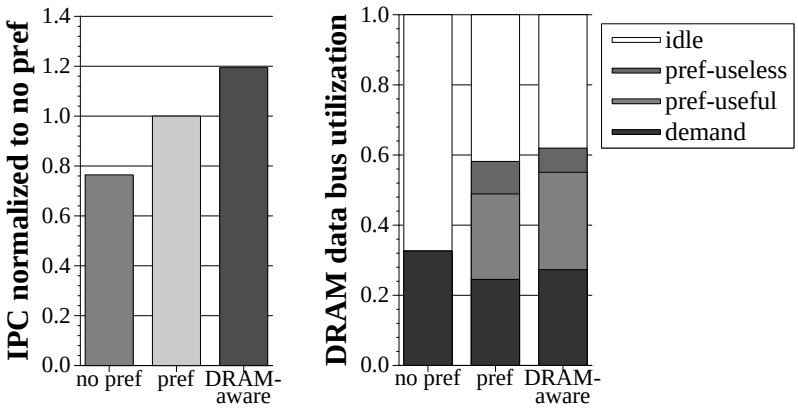



(b) 4-core chip multiprocessor

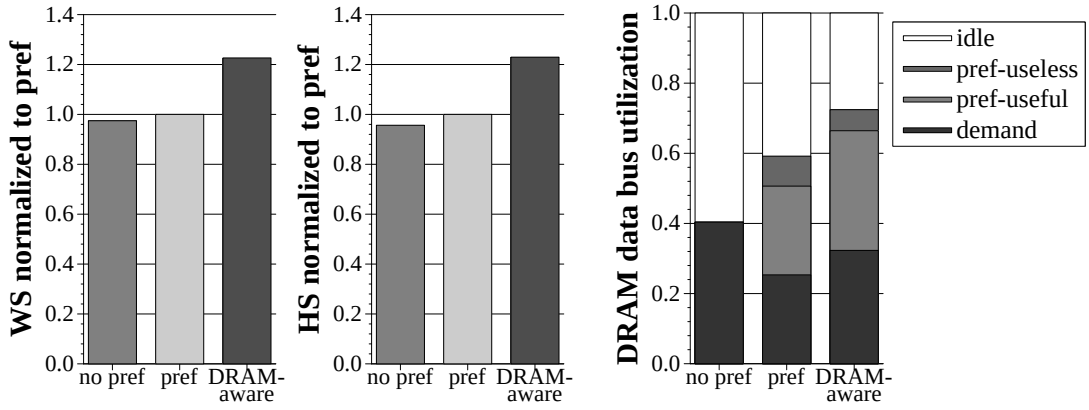



(c) 8-core chip multiprocessor

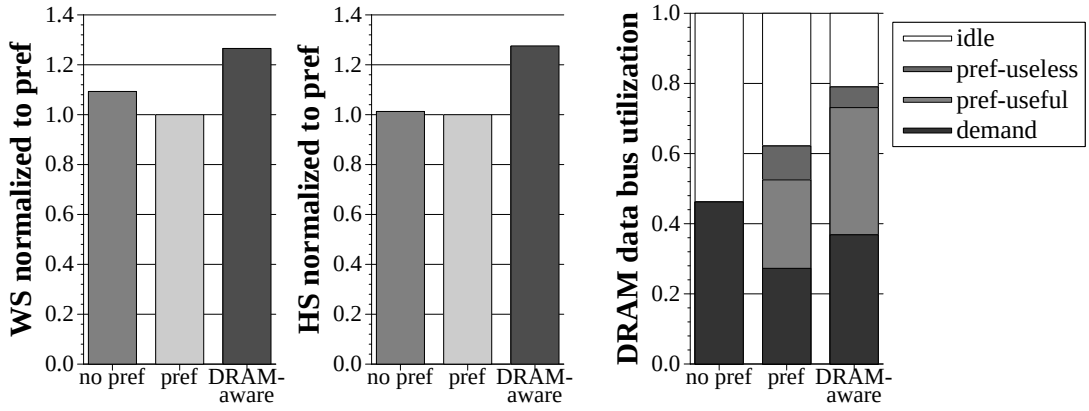
Figure 9.1: Performance of individual DRAM-aware mechanisms on single, 4, and 8-core systems




(a) Single-core processor




(b) 4-core chip multiprocessor




(c) 8-core chip multiprocessor

Figure 9.2: Performance and DRAM bus utilization of combined DRAM-aware mechanisms

prefetching, the baseline with prefetching, and all DRAM-aware polices combined on the single, 4 and 8-core systems. The performance numbers are normalized to the baseline systems with stream prefetching.

The DRAM-aware mechanisms significantly increase DRAM bus utilization by 6.5% for the 20 memory intensive benchmarks on the single-core system as shown in Figure 9.2(a). This is because the DRAM-aware mechanisms allow the DRAM controllers to increase row buffer locality and BLP while reducing write-caused interference. They reduce bus utilization of useless prefetches by 25.5% compared to the baseline prefetching mainly due to Adaptive Prefetch Dropping (APD) and BLP-Aware Prefetch Issue (BAPI). As discussed in Chapters 5 and 6, APD cancels prefetches and BAPI limits the issue of prefetches when the estimated prefetch accuracy is low. The increased DRAM utilization for useful requests improves performance by 19.4% compared to the baseline prefetching.

The DRAM-aware mechanisms are also very effective on the 4 and 8-core systems as shown in Figures 9.2(b) and (c). The mechanisms together increase DRAM utilization by 22.4% (from 59.1% to 72.4% utilization of the peak bandwidth) while reducing useless utilization by 29.9% compared to the baseline for the 30 4-core workloads. This in turn improves weighted speedup and harmonic mean of speedups by 22.6% and 23.0% respectively.

On the 8-core system, they improve WS and HS by 26.5% and 27.6% by increasing DRAM bus utilization by 27.1% (from 62.2% to 79.0% utilization of the peak bandwidth). This reduces useless utilization by 38.9% as well. The benefits become larger as the number of cores increases.

We conclude that our DRAM-aware policies work synergistically and significantly improve system performance by better utilizing the DRAM system for useful requests on single, 4, and 8-core systems.

# Chapter 10

# Conclusion and Future Research Directions

## 10.1 Conclusion

DRAM performance is one of the most important contributing factors to the overall performance of computer systems. However, DRAM performance is severely limited if a microprocessor's on-chip memory system management policies do not take into account DRAM characteristics: row buffer locality, bank-level parallelism (BLP), and write-caused interference.

This dissertation identified conventional on-chip memory system management policies that can limit DRAM performance, and proposed new low-cost policies that allow higher performance of the DRAM system. We proposed and evaluated four low-cost DRAM characteristic-aware mechanisms, each of which works on a different on-chip memory resource management policy.

To maximize DRAM row buffer locality for useful memory requests and minimize the negative effect of useless prefetches, this dissertation proposed Prefetch-Aware DRAM Controllers (PADC). PADC treats likely-useful prefetches and demands equally so that the DRAM controller can exploit row buffer locality for useful requests. It also delays and drops likely-useless prefetches. We show in Chapter 5 that PADC significantly outperforms the existing rigid DRAM scheduling policies and requires low-cost hardware and design support.

To maximize DRAM bank-level parallelism in the presence of prefetching, we proposed in Chapter 6 two BLP-aware memory request issue policies. They determine the order in which memory requests are sent from one on-chip buffer to another. The BLP-aware prefetch issue policy sends prefetches that can be serviced in parallel with requests to other DRAM banks. The BLP-preserving memory

request issue policy does the actual loading of the DRAM request buffers so that requests from the same core can be serviced in parallel. This reduces the serialization of each core's otherwise parallel requests. The proposed request issue policies increase and preserve BLP, thereby significantly reducing memory stall time on both single and multi-core systems.

To maximize row buffer locality and bank-level parallelism and minimize write-caused interference in the DRAM system, this dissertation proposed a DRAM-aware last-level cache replacement policy. The DRAM-aware replacement policy replaces cache lines that would incur low-cost (serviced quickly or in parallel) rather than high-cost (serviced slowly or serially) in terms of refetching and writeback due to the three DRAM characteristics. We showed in Chapter 7 that the DRAM-aware replacement policy significantly outperforms DRAM-unaware replacement policies.

To further reduce write-caused interference, this dissertation proposed an aggressive DRAM-aware last-level cache writeback policy. In contrast to the proposed DRAM replacement policy that takes action only when a replacement is necessary, DRAM-aware writeback proactively cleans dirty lines that can be written back quickly to DRAM due to row buffer locality. We showed in Chapter 8 that this policy significantly reduces write-caused interference because it allows more writes to be written back quickly.

Each of the four mechanisms manages different on-chip memory resources to improve DRAM utilization. We showed in Chapter 9 that the four mechanisms work synergistically when employed together. They significantly increase DRAM bus utilization for useful data and significantly improve performance beyond what can be achieved by each one alone on both single and multi-core systems. We conclude that DRAM-aware on-chip memory system design can significantly improve DRAM performance, enabling higher performance for the entire system.

Our proposals are not limited to DRAM-based main memory systems. Other memory technologies in the future are also likely to employ multiple banks and row

buffers (i.e., sense amplifiers which can serve as row buffers) to provide high bandwidth and low latency. They will also likely have large write-caused interference due to high bus clock frequency. As such, the key ideas of the proposed mechanisms in this dissertation should be able to be seamlessly applied to on-chip memory systems that employ other main memory technologies.

## 10.2 Future Research Directions

This dissertation introduced the notion of main memory system-aware design in on-chip microarchitectures. There are several possible future research directions in improving main memory performance.

- As discussed above, the key ideas of the proposed mechanisms in this dissertation are not limited to today's DRAM-based systems. Other more scalable main memory technologies in the future will likely present characteristics similar to DRAM in order to provide high peak bandwidth. For example, a recently developed technology, phase change memory [35, 64] exhibits longer latency of writes, which likely increases write-caused interference. The key ideas presented in this dissertation could be extended to processors that employ new main memory technologies.

- DRAM performance varies depending on the memory address mapping and the applications' memory behavior as well as on-chip memory system management policies. DRAM characteristic-aware memory allocators, compilers, and profilers can increase DRAM utilization efficiency by changing address mapping in a DRAM-aware manner and giving hints about an application's memory behavior to the on-chip memory resource management policies.

- The concept of DRAM-awareness can also be applied to other on-chip memory resources for many-core systems in the future. An example is on-chip interconnect. A DRAM-aware interconnect could prioritize requests to better

exploit row buffer locality and bank-level parallelism so that effective memory stall time could be minimized.

- Last-level cache management that takes into account both temporal locality and DRAM characteristics can further improve efficiency. An example is to switch between temporal locality-aware replacement and DRAM-aware policies based on runtime behavior of an application. This will result in both cache and DRAM efficiency.

# Bibliography


[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[2] 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*, 1991.

[3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, 1999.

[4] M. Charney and T. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 31(3), 1997.

[5] M. Charney and A. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.

[6] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, 2004.

[7] J. Doweck. Inside Intel Core microarchitecture and smart memory access. *Intel Technical White Paper*, 2006.

[8] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing (ICS)*, 1997.

[9] J. D. Dundas. *Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream*. PhD thesis, University of Michigan, 1998.

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, 2009.

[11] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA-15)*, 2009.

[12] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[13] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3), 2008.

[14] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

[15] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.

[16] E. G. Hallnor and S. K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Workshop on Memory Performance Issues*, 2004.

[17] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001. Q1 2001 Issue.

[18] I. Hur and C. Lin. Adaptive history-based memory scheduler. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[19] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO-39)*, 2006.

[20] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA-35)*, 2008.

[21] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[22] JEDEC. *JEDEC Standard: DDR3 SDRAM STANDARD (JESD79-3D).* http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[23] J. Jeong and M. Dubois. Optimal replacements in caches with two miss costs. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, 1999.

[24] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[25] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[26] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, 1997.

[27] 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 International Symposium on Computer Architecture (ISCA-17)*, 1990.

[28] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40, 2004.

[29] S. Kim. Area-efficient error protection for caches. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2006.

[30] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA-16)*, 2010.

[31] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO-43)*, 2010.

[32] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-18)*, 1991.

[33] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA-8)*, 1981.

[34] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM power6 microarchitecture. *IBM Journal of Research and Development*, 51, 2007.

[35] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, 2009.

[36] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *Proceedings of the 41st International Symposium on Microarchitecture (MICRO-41)*, 2008.

[37] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. Technical Report TR-HPS-2008-002, University of Texas at Autin, 2008.

[38] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-002, The University of Texas at Autin, Apr. 2010.

[39] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving memory bank-level parallelism in the presence of prefetching. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, 2009.

[40] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, 2000.

[41] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[42] W.-F. Lin, S. K. 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 (HPCA-7)*, 2001.

[43] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001.

[44] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[45] S. A. McKee. Hardware support for dynamic access ordering: Performance of some design options. Technical Report CS-93-08, University of Virginia, Aug. 1993.

[46] S. A. McKee, R. H. Klenke, A. J. Schwab, W. A. Wulf, S. A. Moyer, J. H. Aylor, and C. Y. Hitchcock. Experimental implementation of dynamic access ordering. Technical Report CS-93-42, University of Virginia, Aug. 1993.

[47] S. A. McKee, R. H. Klenke, A. J. Schwab, W. A. Wulf, S. A. Moyer, C. Y. Hitchcock, and J. H. Aylor. Experimental implementation of dynamic access ordering. In *Proceedings of IEEE 27th Hawaii International Conference on Systems Sciences (HICSS-27)*, 1994.

[48] S. A. McKee, W. A. Wulf, J. H. Aylor, R. H. Klenke, M. H. Salinas, S. I. Hong, and D. A. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49, Nov. 2000.

[49] Micron. *2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*, 2002. http://download.micron.com/pdf/datasheets/dram/ddr3/.

[50] 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*, 1992.

[51] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5), October 2005.

[52] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005.

[53] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-40)*, 2007.

[54] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA-35)*, 2008.

[55] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[56] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6), 2003.

[57] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Workshop on Memory Performance Issues*, 2004.

[58] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO-39)*, 2006.

[59] K. J. Nesbit, A. S. Dhodapkar, J. Laudon, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[60] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, 1999.

[61] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic

instrumentation. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[62] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high-performance caching. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA-34)*, 2007.

[63] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA-33)*, 2006.

[64] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, 2009.

[65] S. Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[66] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[67] Samsung. *Application Note: tWR (Write Recovery Time)*, 2002. http://www.samsung.com/global/business/semiconductor/products/dram/.

[68] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[69] A. J. Smith. Cache memories. *Computing Surveys*, 14(4), 1982.

[70] W. E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3, 1956.

[71] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, 2000.

[72] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*.

[73] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[74] V. Srinivasan, G. S. Tyson, and E. S. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan Technical Report, 1999.

[75] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating DRAM and last-level cache policies. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[76] Sun Microsystems, Inc. *OpenSPARC$^{(TM)}$ T1 Microarchitecture Specification*.

[77] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[78] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11, Jan. 1967.

[79] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO-39)*, 2006.

[80] O. Wechsler. Inside Intel Core microarchitecture. *Intel Technical White Paper*, 2006.

[81] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2), 1965.

[82] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th International Symposium on Microarchitecture (MICRO-24)*, 1991.

[83] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, 1992.

[84] D. H. Yoon and M. Erez. Memory mapped ECC: low-cost error protection for last level caches. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, 2009.

[85] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, 2009.

[86] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 2000 International Conference on Supercomputing (ICS-14)*, 2000.

[87] 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 27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[88] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th International Conference on Supercomputing (ICS-17)*, 2003.

[89] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[90] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 32nd International Conference on Parallel Processing*, 2003.

[91] X. Zhuang and H.-H. S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1), Jan. 2007.

[92] W. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. Patent Number 5,630,096, 1997.

# Vita

Chang Joo Lee was born in Seoul, South Korea on 12 September 1975. He finished Seoul High School, Seoul, Korea in February 1994. He completed his B.S. degree in Electrical Engineering in February 2001 at Seoul National University, Seoul, Korea. He earned his M.S. degree in Electrical and Computer Engineering from the University of Texas at Austin, Texas, USA in May 2004.

Chang Joo was a recipient of the scholarship from Ministry of Information and Communication in Korea during 2002-2006, the IBM PhD fellowship in 2007, and the IBM scholarship in 2008. He served as a teaching assistant for EE382N Microarchitecture in Spring 2006, EE360N Computer Architecture in Spring 2007, and EE306 Introduction to Computing in Fall 2008. Chang Joo worked as a summer intern at Freescale Semiconductor in 2004 and 2005, and IBM T.J. Watson Research in 2006 and 2007.

Permanent address: 622-13 Yeoksam-Dong Kangnam-Gu, Seoul
135-080, Republic of Korea

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.