THE UNIVERSITY
of EDINBURGH

Optimizing Cache Utilization in Modern Cache Hierarchies

Cheng-Chieh Huang

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2015

Abstract

Memory wall is one of the major performance bottlenecks in modern computer systems. SRAM caches have been used to successfully bridge the performance gap between the processor and the memory. However, SRAM cache's latency is inversely proportional to its size. Therefore, simply increasing the size of caches could result in negative impact on performance. To solve this problem, modern processors employ multiple levels of caches, each of a different size, forming the so called memory hierarchy. Upon a miss, the processor will start to lookup the data from the highest level (L1 cache) to the lowest level (main memory). Such a design can effectively reduce the negative performance impact of simply using a large cache. However, because SRAM has lower storage density compared to other volatile storage, the size of an SRAM cache is restricted by the available on-chip area. With modern applications requiring more and more memory, researchers are continuing to look at techniques for increasing the effective cache capacity. In general, researchers are approaching this problem from two angles: maximizing the utilization of current SRAM caches or exploiting new technology to support larger capacity in cache hierarchies.

The first part of this thesis focuses on how to maximize the utilization of existing SRAM cache. In our first work, we observe that not all words belonging to a cache block are accessed around the same time. In fact, a subset of words are consistently accessed sooner than others. We call this subset of words as *critical words*. In our study, we found these critical words can be predicted by using access footprint. Based on this observation, we propose *critical-words-only cache (co cache)*. Unlike the conventional cache which stores all words that belongs to a block, co-cache only stores the words that we predict as critical. In this work, we convert an L2 cache to a co-cache and use L1s access footprint information to predict critical words. Our experiments show the co-cache can outperform a conventional L2 cache in the workloads whose working-set-sizes are greater than the L2 cache size. To handle the workloads whose working-set-sizes fit in the conventional L2, we propose the adaptive co-cache (aco-cache) which allows the co-cache to be configured back to the conventional cache.

The second part of this thesis focuses on how to efficiently enable a large capacity on-chip cache. In the near future, 3D stacking technology will allow us to stack one or multiple DRAM chip(s) onto the processor. The total size of these chips is expected to be on the order of hundreds of megabytes or even few gigabytes. Recent works have proposed to use this space as an on-chip DRAM cache. However, the tags of the DRAM cache have created a classic space/time trade-off issue. On the one hand, we

would like the latency of a tag access to be small as it would contribute to both hit and miss latencies. Accordingly, we would like to store these tags in a faster media such as SRAM. However, with hundreds of megabytes of die-stacked DRAM cache, the space overhead of the tags would be huge. For example, it would cost around 12 MB of SRAM space to store all the tags of a 256MB DRAM cache (if we used conventional 64B blocks). Clearly this is too large, considering that some of the current chip multiprocessors have an L3 that is smaller. Prior works have proposed to store these tags along with the data in the stacked DRAM array (tags-in-DRAM). However, this scheme increases the access latency of the DRAM cache. To optimize access latency in the DRAM cache, we propose *aggressive tag cache (ATCache)*. Similar to a conventional cache, the ATCache caches recently accessed tags to exploit temporal locality; it exploits spatial locality by prefetching tags from nearby cache sets. In addition, we also address the high miss latency issue and cache pollution caused by excessive prefetching. To reduce this overhead, we propose a cost-effective prefetching, which is a combination of dynamic prefetching granularity tunning and hit-prefetching, to throttle the number of sets prefetched. Our proposed ATCache (which consumes 0.4% of overall tag size) can satisfy over 60% of DRAM cache tag accesses on average.

The last proposed work in this thesis is a *DRAM-Cache-Aware (DCA) DRAM controller*. In this work, we first address the challenge of scheduling requests in the DRAM cache. While many recent DRAM works have built their techniques based on a tags-in-DRAM scheme, storing these tags in the DRAM array, however, increases the complexity of a DRAM cache request. In contrast to a conventional request to DRAM main memory, a request to the DRAM cache will now translate into multiple DRAM cache accesses (tag and data). In this work, we address challenges of how to schedule these DRAM cache accesses. We start by exploring whether or not a conventional DRAM controller will work well in this scenario. We introduce two potential designs and study their limitations. From this study, we derive a set of design principles that an ideal DRAM cache controller must satisfy. We then propose a DRAM-cache-aware (DCA) DRAM controller that is based on these design principles. Our experimental results show that DCA can outperform the baseline over 14%.

Lay Summary

Memory is an essential component of today's computing systems. Computer programs use memory to store the input, output, and also the intermediate values during computation. However, because the speed of today's main memory is significantly slower than the processor speed, modern processors use hardware caches to bridge the performance gap between processor and main memory. Typically, caches are made using SRAM, which is a fast but a low density technology. The low-density of SRAM limits the size of caches due to the limited available chip area.Meanwhile, modern applications such as big data analytics or high-performance computing workloads require more and more memory to complete their tasks. Because the conventional cache cannot scale as the growing memory requirement in these applications, memory becomes the bottleneck in developing a high performance/efficient computing system. Therefore, how to increase cache capacity for applications with larger memory requirement is an important task for developing next-generation computers. In this thesis, we propose techniques to mitigate capacity issue in memory systems. In the first part of the thesis, we try to improve the (SRAM) cache utilization in current memory systems. In the second part of this thesis, we study on how to efficiently enable a larger (DRAM) cache by exploiting emerging die-stacking technology.

Acknowledgements

I would like to thank my supervisor Vijay Nagarajan for all the help and guidance that he has provided. Coming from an industry background, without a postgraduate degree, I had no idea how to do research at the beginning of my PhD. Vijay helped me build up all the essential skills for doing research work.

I would also like to thank everybody at Institute of Computing Systems Architecture (ICSA) for their support and help. I am thankful to my second supervisor Nigel Topham for reviewing my works in the annual review panel. I would also like to thank Michael O'Boyle for giving me opportunity of being an organizing member in PLDI 2014. Many thanks to my fellow students, Jose Cano, Marco Elver, Priyank Faldu, Arpit Joshi, Rakesh Kumar, Thibaut Lutz, Alberto Magni, Andrew McPherson, Bharghava Rajaram, Erik Tomusk, and Yuan Wen. As an non-native English speaker, they have provided a great support in both improving my writing and research skills.

Finally, and most importantly, I would like to thank my parents for their absolutely unconditional support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- C. Huang and V. Nagarajan, "Increasing Cache Capacity via Critical-words-Only Cache", The 32nd IEEE International Conference on Computer Design (ICCD), Seoul, Korea, October 2014 [1].
- C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache", The 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT), Edmonton, Canada, August 2014 [2].

(*Cheng-Chieh Huang*)

Table of Contents

List of Figures

List of Tables

Part I

Preamble

Chapter 1

Introduction

1.1 Overview

In modern computer systems, memory has become an overwhelming bottleneck in system performance due to the disparity of speed between the processor and the off-chip memory, otherwise called the *memory wall*. To bridge the performance gap between processor and off-chip memory, modern processors use hardware caches to avoid the penalty of accessing off-chip memory. A hardware cache is a small memory buffer that is usually made of static random access memory (SRAM) and placed on the processor chip. While the access latency of off-chip main memory is around 10 to 100 nanoseconds, the on-chip cache allows the access latency to be in the order of few nanoseconds or less. However, this latency is inversely proportional to its size. Larger SRAM size will result in slower access latency. Therefore, modern processors use multiple levels of caches (each of a different size) in its memory system. Upon a miss, the processor will start to look up the data from the highest level (L1 cache) to the lowest level (main memory). This is known as memory hierarchy.

In addition to multiple levels of caches, most modern processors, ranging from mobile phones to supercomputers, have multiple processing units (cores) in a single physical package, known as the chip multiprocessor (CMP). A CMP chip will typically have multiple small caches for each core (private caches) and a large cache shared among all cores (shared cache). In a typical CMP memory hierarchy, the L1 is usually small and private to match the speed of the processor core. On the other hand, lower level caches such as L2 or L3 are are relatively larger and slower access latency compared to the L1. Furthermore, to enhance the performance of multi-thread programs, the last level cache is usually shared among all cores.

Computer programs nowadays require more memory to complete their jobs due to various reasons. For example, huge data sets in big data applications increase the memory footprint in modern computers. On the other hand, total cache size in today's processor chips is restricted by various reasons including the price, power budget, latency, or available die area. For example, we can see an Intel Nehalem processor from 2007 has similar cache settings as Intel Haswell processor from 2013 [5]. With cache sizes not able to match the growing memory requirement in these modern workloads, caches will suffer from excessive cache misses (called *capacity misses*) and lose the purpose of bridging the latency gap between the processor and the off-chip memory. Given that memory is one of the major performance bottlenecks in computer systems, how to maximize cache efficiency is a major challenge for computer scientists and therefore draws a significant amount of research works from the computer architecture community.

In this dissertation, we study how to support higher cache capacity in the CMP processor. In modern processors, most on-chip caches use SRAM as storage. However, SRAM's low packing density limit the available on-chip storage. Therefore, how to use existing SRAM space effectively becomes a fundamental problem for developing a high performance/efficiency processor.

On the other hand, many computer programs in HPC [6] or servers [7] today have working-set-sizes that are over hundreds of megabytes. Clearly, the conventional SRAM cache will not be able to provide enough capacity for such workloads. Recently, 3D-stacking technology has enabled the option of embedding hundreds of megabyte or even few gigabytes DRAM chips onto the processor. Prior studies have proposed using this stacked DRAM as an L4 DRAM cache [8]. Given the huge size, many design questions, such as how to organize data and tag in the DRAM array [3, 4] arise in DRAM cache. Therefore, how to efficiently enable this DRAM cache is another important research problem.

To summarize, we focus on two architectural questions in this dissertation:

- How to effectively use existing SRAM space in memory hierarchy?
- How to effectively enable a large DRAM cache?

1.2 Contributions

1.2.1 Critical-words-Only Cache

In the first work, we target the first architectural question that we arise in introduction — *how to effectively use existing SRAM space in memory hierarchy?* Today, most processors have multiple levels of caches. In multi-level cache hierarchies, choosing the right cache size for each level is critical for performance. As we mentioned in the previous section, the first-level cache (L1) is typically small, in order to match the speed of the processor. The lower level caches, on the other hand, are typically large, in order to reduce capacity misses. Ideally, the lower level cache should be large enough to fit workloads' working-set-sizes, but situations may arise in which the size of the cache is lesser than the working-set-sizes of the workload. In that case, this cache could adversely affect the overall system performance.

The first work of this thesis proposes a cache design called emphcritical-words-only cache (co-cache) for increasing the effective cache capacity. Our approach involves rethinking the notion of cache blocks; instead of storing all the words that belong to a cache block, we only store the critical words, where the critical words are the words that are generally accessed before the others. Our experiments show that with our design a 256 KB L2 performs as well as a 512 KB conventional L2 cache on average.

1.2.2 Aggressive Tag Caching for DRAM cache

As 3D-stacking technology has enabled the chance to support a very large size DRAM cache in the processor chip, *how to effectively enable this DRAM cache* is a major challenge in research works. Because of its large size (a DRAM cache can be on the order of hundreds of megabytes), the total size of the tags associated with it can also be quite large (on the order of tens of megabytes). The large size of the tags has created a problem [3]. Should we maintain the tags in the DRAM and pay the cost of a costly tag access in the critical path? Or should we maintain the tags in the faster SRAM by paying the area cost of a large SRAM for this purpose? Prior works have primarily chosen the former and proposed a variety of techniques for reducing the cost of a DRAM tag access. In this work, we first establish (with the help of a study) that maintaining the tags in SRAM, because of its smaller access latency, leads to overall better performance. Motivated by this study, we ask if it is possible to maintain tags in SRAM

without incurring high area overhead. Our key idea is simple. We propose to cache the tags in a small SRAM tag cache; we show that there is enough spatial and temporal locality amongst tag accesses to merit this idea. We propose the aggressive tag cache (ATCache) which is a small SRAM tag cache. Similar to a conventional cache, the ATCache caches recently accessed tags to exploit temporal locality; it exploits spatial locality by prefetching tags from nearby cache sets. In addition, we also address the high miss latency issue and cache pollution caused by excessive prefetching. To reduce this overhead, we propose a cost-effective prefetching, which is a combination of dynamic prefetching granularity tunning and hit-prefetching, to throttle the number of sets prefetched. Our proposed ATCache (which consumes 0.4% of overall tag area) can satisfy over 60% of DRAM cache tag accesses on average.

1.2.3 DRAM-Cache-Aware DRAM Controller

We also observed many recent DRAM works [3, 4, 9, 10, 11] have built their technique based on a tags-in-DRAM scheme, which means cache tags are stored in the DRAM array itself. Storing these tags in the DRAM array, however, increases the complexity of a DRAM cache request. In contrast to a conventional request to DRAM main memory, a request to the DRAM cache will now translate into multiple DRAM cache accesses (tag and data). In this work, we address challenges of how to schedule these DRAM cache accesses. We start by exploring whether or not a conventional DRAM controller will work well in this scenario. We introduce two potential designs and study their limitations. From this study, we derive a set of design principles that an ideal DRAM cache controller must satisfy. We then propose a DRAM-cache-aware (DCA) DRAM controller that is based on these design principles. Our experimental results show that DCA can outperform the baseline over 14%.

1.3 Thesis Structure

Chapter 2 provides background on memory systems. In this chapter, we first explain the conventional memory hierarchy and the basic structure of a hardware-managed cache. We then show the DRAM terminologies and access protocols which will be used in our DRAM cache works. In the rest of this chapter, we summarize the current state-of-the-art in the DRAM cache.

Chapter 2, 3, and 4 are the main contributions of this thesis. **Critical-words-Only Cache** (co-cache, chapter 2) provides an opportunity to increase effective cache capacity when workloads' working-set-sizes are not fitted in the cache size. We describe the problem and evaluation our idea in this chapter. **Aggressive Tag Cache** (ATCache, chapter 3) is used to enhance the access latency of a DRAM cache with the tags-in-DRAM design. ATCache uses a small SRAM cache to the buffer of DRAM cache's tags. Also, we study and use a cost-effective prefetching to avoid prefetching overhead. The last contribution of this thesis is the **DRAM-Cache-Aware (DCA) DRAM controller** (chapter 2). In this work, we address the problem of how to schedule a DRAM cache's request and propose a controller design that can enhance the scheduling efficiency in DRAM cache.

In chapter 5, we first summarize the key findings and results presented in this thesis. Finally, we describe several potential future works and/or possible further studies related to this thesis.

Chapter 2

Background

2.1 Memory Hierarchy Overview

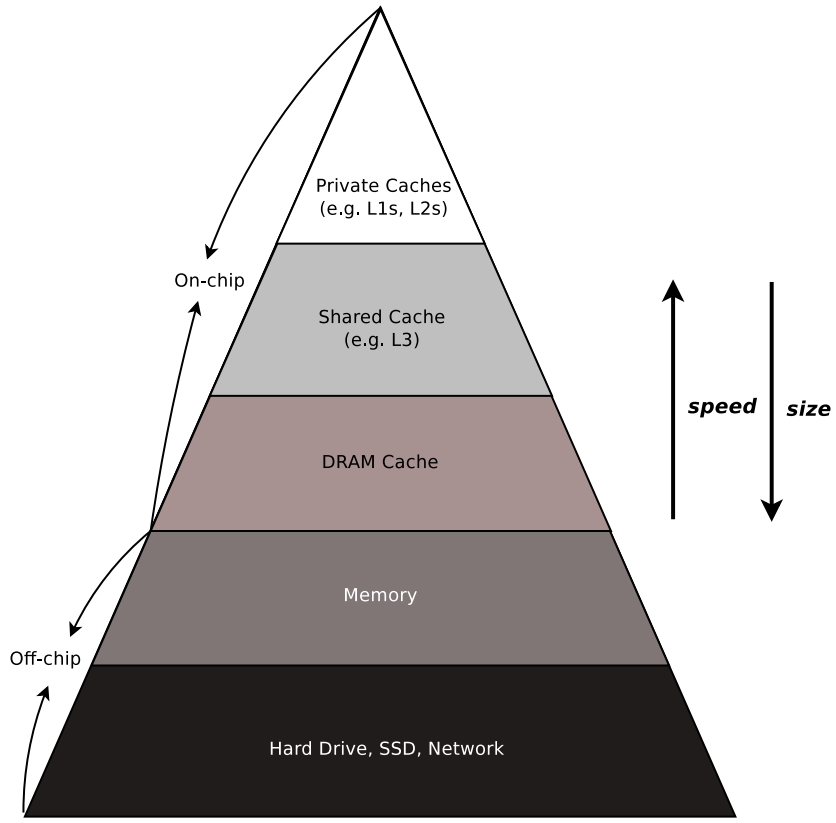

Figure 2.1: Memory Hierarchy

Memory wall is one of major performance bottlenecks in modern computer systems, which is caused by the disparity of speed between the processor and the off-chip memory. To avoid the access of memory, *cache* is used to bridge this latency gap. A

cache is a small memory region that stores partial contents of the main memory. These caches are usually made of static random access memory (SRAM). SRAM provides an access latency that can match the speed of processors. However, as the SRAM speed (latency) is inversely proportional to its size, to match the processor speed, the cache needs to be on the order of tens of kilobytes [12].

To overcome this problem, modern processors use multiple layers of caches. The level 1 (L1) cache, which typically small (e.g. 32KB), can complete the access in a few processor cycles. Lower level caches (level 2 or level 3) are typically larger, ranging from hundreds of kilobytes to few megabytes. Although L2 and L3 caches are usually slower (about 10 to 40 processor cycles) than L1 caches, the larger capacity allows them to further reduce the number of accesses to the off-chip memory, which can incur a penalty of over 100 processor cycles. This multiple layer design is known as *memory hierarchy*.

One disadvantage of SRAM cache is its area cost. SRAM has lower area density compared to other storage media such as dynamic random access memory (DRAM). Because of this, the total size of all on-chip caches is usually under 30 MB. Therefore, to support higher cache capacity, many recent works [3, 13, 14, 15, 4, 9] have proposed using die-stacked DRAM as an L4 cache. This allows the cache size to be over hundreds of megabytes. Figure 2.1 shows an overview of the modern memory hierarchy.

2.2 Cache Overview

The *cache* mentioned in this paper is a transparent hardware-managed cache, which means they work independently and the software is not aware of their existence.[1]. In this section, we will show how a conventional cache is organized and some of their optimization techniques.

2.2.1 Organization

Address Mapping and Set-associativity. In the cache, if a given address can be placed in any cache blocks, this is called *fully-associative cache* as shown in Figure 2.2 (a). However, even an L1 cache, which has the smallest size in the cache

[1] On the other hand, a software-managed cache is a cache that requires the programmer/OS to decide when to evict the data and where to store the data.

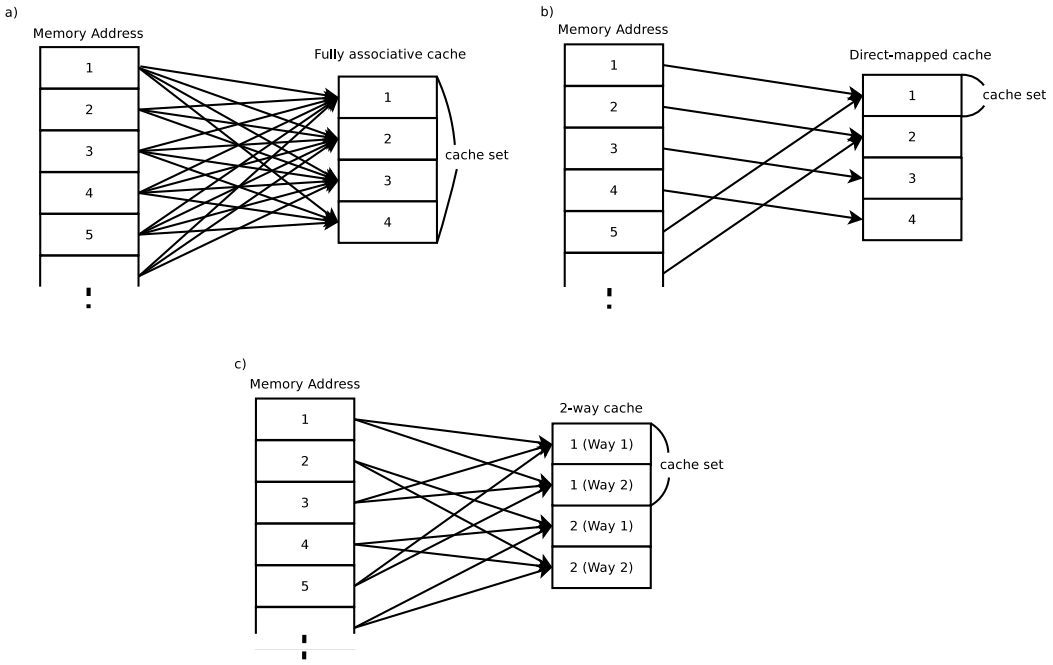

Figure 2.2: Memory Mapping in Different Cache Structure

hierarchy, will have about 1000 of blocks (assumed a 64KB L1 cache, 64B per block). In other words, a single cache access will need to search the given address in about 1000 blocks. The access overhead in a fully-associative cache is huge in both energy and latency perspectives. An extreme solution to this problem is the direct-mapped design, in which a memory address can only be mapped to one cache block (as shown in Figure 2.2(b)). In this case, only one location (block) needs to be checked for a cache access. However, the direct-mapped design gives rise to rise another potential problem. Considering an access sequence that is repeatedly accessing two blocks (address 1 and 5) in Figure 2.2(b), although our cache has space to store 4 data blocks, the accesses will always miss in the direct-mapped cache. These are known as *conflicts misses*[2]. As a trade-off, most caches use a set-associative design. In this design, a memory address can be mapped to multiple blocks (ways) but not all cache blocks. An example of a two-way design is shown in Figure 2.2(c).

Accessing a set-associative cache. Figure 2.3 shows the flow of accessing a set-associative cache. When a processor sends a load request to the cache, the cache controller will use the part of the load address to locate tag/data (1) in the SRAM array. Later, the tag in the SRAM array will be checked (2) with the tag part of the load address. If the check of the tag results in a match, it is a cache hit and the data located

[2]On the other hand, cache misses caused by insufficient cache blocks is called *capacity misses*

in (1) will be sent back to the processor (4a). Otherwise, a cache miss will fetch the data from the lower-level memory (4b).

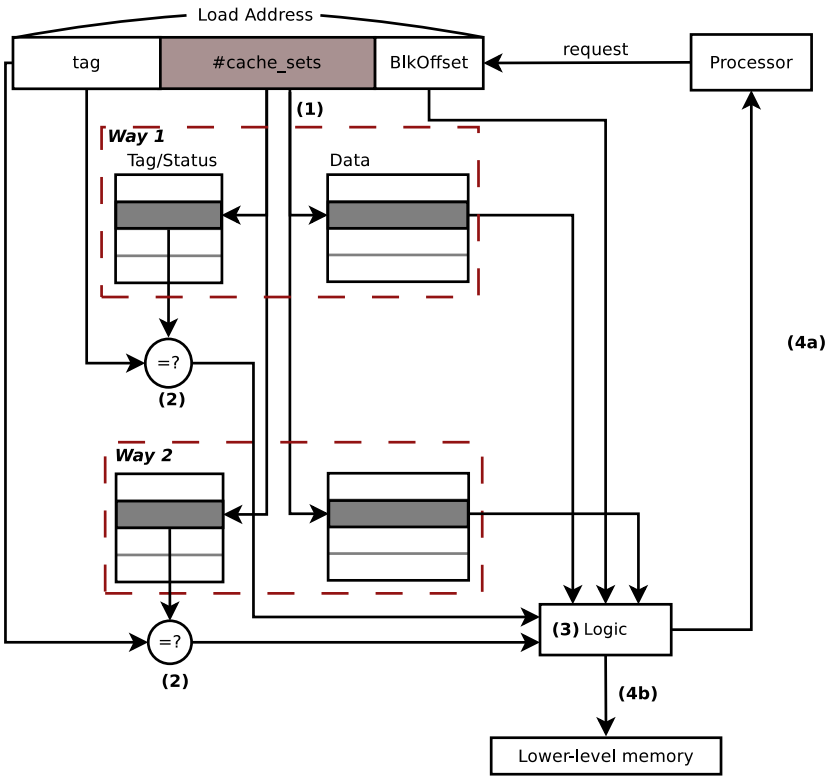

Figure 2.3: Cache Organization

Cache Replacement. In the cache, a miss will fetch requested block from lower memory. To store this block, the controller will need to decide a victim block in the cache and use it to replace with requested block. The algorithm of choosing the victim block is called *replacement policy*. One of the popular algorithms is called *Least Recently Used (LRU)* [16, 17, 18]. LRU will try to discard the least recently used item inside a cache set. In addition, many replacement policies [19, 20, 21] are proposed to enhance the replacement policy in multi-core systems.

2.2.2 Cache Optimizations

As cache hierarchy is a critical factor to the system performance, many optimization techniques have proposed to improve cache efficiency. In this section, we discuss the optimization techniques that are related to our work.

Victim Cache As mentioned in the prior discussion, the multiple associativity is used to reduce the number of conflict misses. However, the increase of cache associativity means the increase of both energy and access latency. This is because set-associative

cache needs to search N items (in parallel) in the same cache set. Jouppi[22] proposed using *victim cache* to improve performance of a direct-mapped cache. This work is based on the observation that conflict misses only happen in a small number of cache sets. The proposed work uses a small fully-associativity cache (victim cache) to hold blocks that are evicted from the direct-mapped cache.

Hardware Prefetcher Researchers keep looking for different kinds of heuristics in the cache accesses and exploit them to improve cache efficiency. One of the observed heuristics is that some of cache access sequences show a specific pattern in a running program. *Hardware data prefetching* is a technique that exploits this heuristic. A hardware data prefetcher captures on-line access heuristic and fetches the data before they are actually demanded by the processor. One famous prefetcher is the stride prefecther [23]. Baer and Chen observed that cache accesses show a stride pattern in a given PC (program counter) address. They use a Reference Prediction Table (RPT) to capture this pattern and prefetch the data. In addition to the stride accesses, many other works [24, 25, 26, 27] also work on capturing non-stride access pattern.

2.3 Dynamic Random Access Memory (DRAM)

2.3.1 Introduction

Dynamic random access memory (DRAM) is the storage media which is widely used as the main memory in today's computer system. A typical DRAM cell requires one transistor and one capacitor (1T1C)[3] to store one bit of data. Compared to SRAM, which normally use six transistors for one bit of data, DRAM provides higher density and has less power leakage.

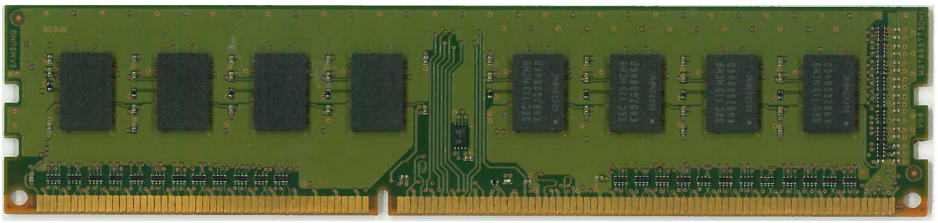

Figure 2.4: A dual in-line memory module

[3] 1T1C is not the only DRAM cell structure. e.g. Intel's 1103 DRAM device use 3T1C cell structure.

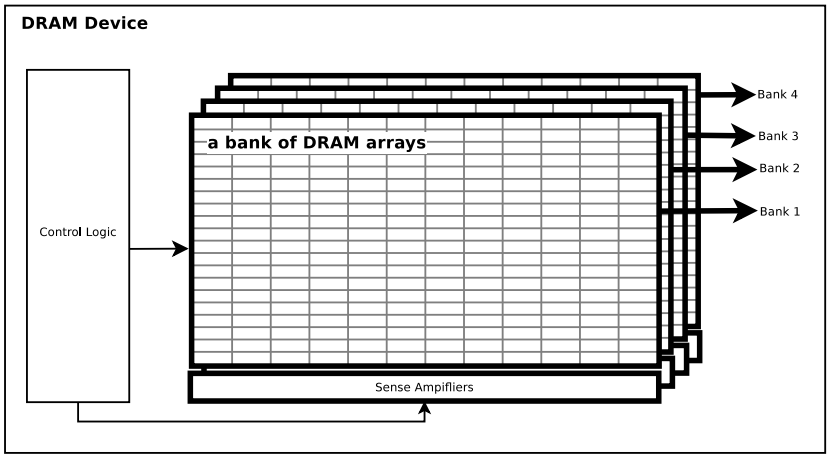

Figure 2.5: A DRAM Device in memory system

2.3.2 Terminologies

In this section, we will explain some DRAM terminologies that we will use in this thesis.

DRAM Module. The DRAM module is a module that contains multiple DRAM chips on a small circuit board with pins, and this module can be plugged or embedded (for portable devices) into a computer motherboard. Figure 2.4 shows a 184-pin dual in-line memory module (DIMM), which is widely used in today's desktop computers. As we can see, this module contains 8 DDR3 (double data rate type three) x8 DRAM chips which constitute a 64-bit data width.

DRAM Device. In most cases[4], a DRAM device is the chip that you can see on your DRAM module. For example, we can see *8 devices* in Figure 2.4. A typically DRAM device consists of decode/access/IO logic, DRAM arrays, and sense amplifiers (also known as row buffer) as shown in Figure 2.5.

Channel. The channel in DRAM systems refers to the bus that connects DRAM controller to DRAM modules (Figure 2.6). Today, high-performance computers commonly support multiple channels in their IMC (integrated memory controller). For example, Intel Haswell platform [5] supports two channels of DDR3 memory.

Rank/Bank/Row/Column. In DRAM systems, a **rank**[5] refers to one or more DRAM devices that operate together in response to a given command. In general, the number of devices per rank is determined by the width of the DRAM module and DRAM device. For example, a rank for a 64-bits DRAM module (e.g. DDR3 x64) can be com-

[4]This is not always true because some of DRAM vendors encapsulate multiple DRAM devices to the same package. For example, a Micron TwinDie^{TM} chip [28] has two DRAM devices.

[5]Depends on manufacturers, definition of the rank and bank might vary in their datasheet.

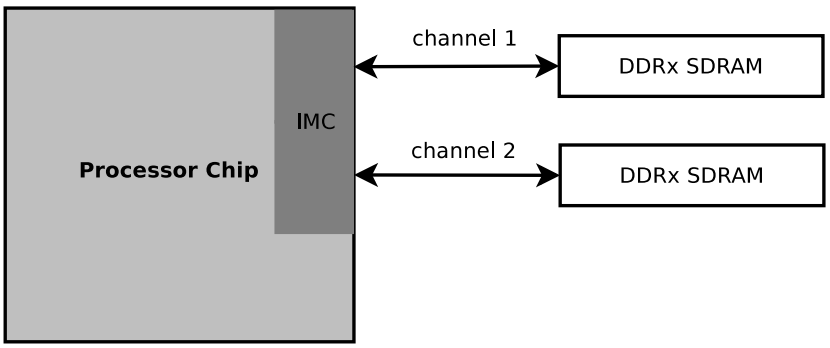

Figure 2.6: A DRAM memory system with two channels. (IMC: Integrated Memory Controller.)

posed of 2 32-bits DRAM devices or 4 16-bits DRAM devices in parallel connection. Figure 2.7 shows the a 64-bit DRAM module that has two ranks of memory and each rank is consists of two x32 devices. Also, as we can see from the figure, a (logical) **bank** in a DRAM module refers to banks in DRAM devices that belong to the same rank. Similarly, a **row** will be spanned in multiple DRAM devices that are in the same rank. A **column** of data is the minimum addressable memory in the DRAM memory systems. The size of a column of data usually equals to the width of the data bus that is 64-bit in our example.

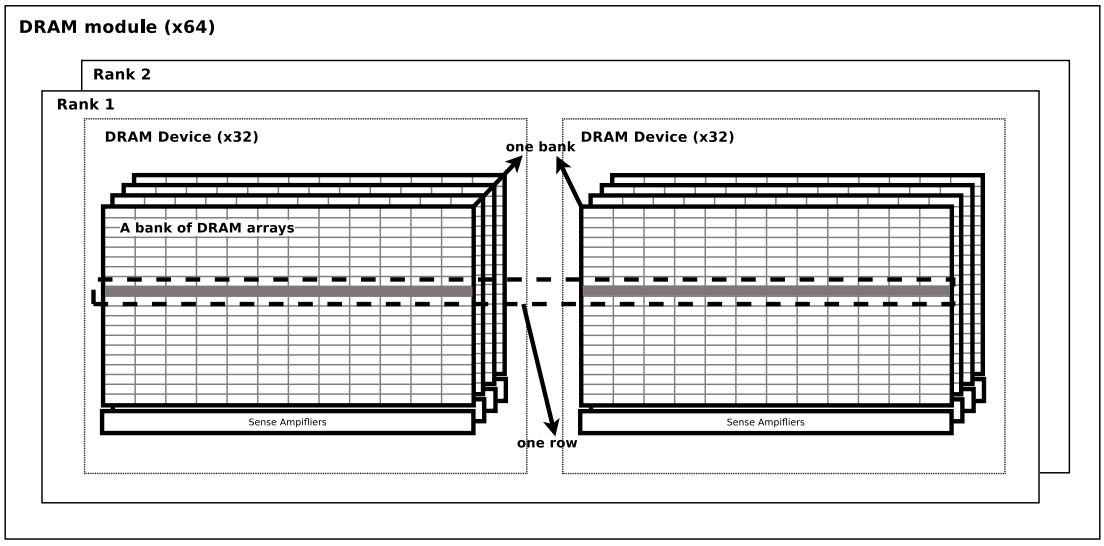

Figure 2.7: The diagram of a x64 DRAM module (with 2 x32 DRAM devices)

Timing	Description
t_{RAS}	Row Address Strobe
t_{CAS}	Column Address Strobe
t_{CCD}	Column to Column Delay
t_{RCD}	Row to Column Delay
t_{RP}	Row Precharge Delay
t_{RTW}	Read to Write Delay
t_{WTR}	Write to Read Delay
t_{CWD}	Column Write Delay
t_{BURST}	Data burst
t_{RFC}	Refresh Cycle

Table 2.1: DRAM Timing Parameters

2.3.3 DRAM Accesses

In DRAM systems, a complete DRAM access will consist of multiple steps/commands. In this section, we first introduce the several basic DRAM commands, and the associated timing parameters[6]. Then, we explain a complete DRAM read/write cycle in detail.

2.3.3.1 Commands

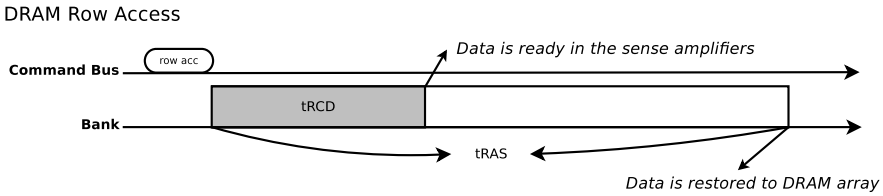

Figure 2.8: A row access in DRAM

Row command As shown in Figure 2.8, a DRAM row access will first move the data from a DRAM row to the sense amplifiers (row buffer) and then, restore them back to the DRAM arrays. There are two primary timing parameters associated with this command. The first one is called *row to column delay (t_{RCD})*. After t_{RCD}, the data in the

[6]DRAM timing parameters are commonly used for representing DRAM access protocols. Table2.1 summarizes the timing parameters that we will use in this thesis.

row will be available by the column command. Therefore, in many academic research papers [3, 4], t_{RCD} is also called t_{ACT} (row activation). Another timing parameter for row access command is the t_{RAS} (row address strobe). t_{RAS} includes the row activation time (t_{RCD}) and the time to restore the data from sense amplifiers to the DRAM array.

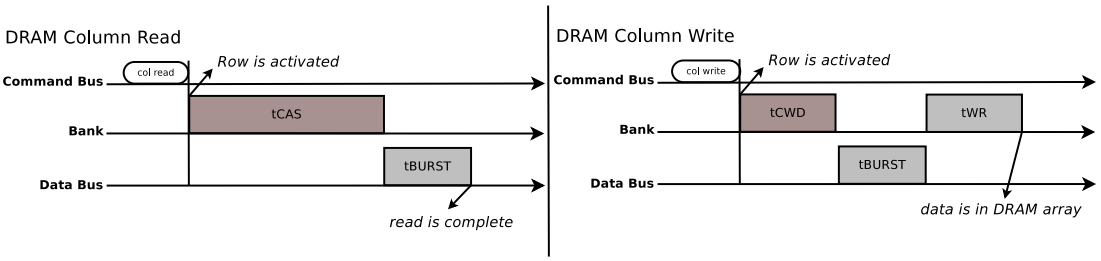

Figure 2.9: Column Read/Write in DRAM

Column Read/Write/BURST commands As explained in row command, a row data will be first moved to the row buffer after t_{RCD}. When the row buffer is ready, the column command can be used to read/write the data into/from the row buffer. Figure 2.9 shows the process of column read/write in column access. In this figure, t_{BURST} is representing the time to transfer a 64-byte block[7] to/from the memory controller completely. For a column read command, the data will be ready for transferring to the data bus after t_{CAS} (Column Address Strobe) and the read request can be satisfied after t_{BURST}. For the column write, data will start to transmit to row buffer after t_{CWD} (Column Write Delay). t_{CWD} specify the timing between the assertion of column write command on the command bus and the start of transmitting write data on the data bus. After the completion of data transferring (t_{BURST}), the data will need t_{WR} before it writes to DRAM arrays.

Precharge Command As we described in Figure 2.8, a typical row access will first load the data into sense amplifiers (row buffer) and then restore the data to the DRAM arrays (after column read/write commands). After the row access, the sense amplifiers (row buffer) will need to be reset before processing another access. This process is called *precharging* and the delay/time for this charging is t_{RP}.

Refresh command As mentioned in §2.3.1, the DRAM uses capacitor to store bits. Due to the nature of the capacitor's characteristics, the electrical charge stored in the capacitor decays over time, and this can lead to the loss of the data. To avoid this problem, DRAM will periodically refresh itself to prevent the loss of the data. In a refresh process, row data in all banks will be read out and restored. After that, row

[7]Generally, the data width of a modern DRAM system equals to the block size of processor caches which is normally eight words (64 bytes).

buffers in all banks will be precharged. This process will take the refresh cycle time (t_{RFC}).

2.3.3.2 Full Access Cycle

Read Cycle A complete DRAM read cycle consists of multiple commands. When a DRAM bank is accessed, a row access command will move the data from DRAM arrays to the sense amplifiers. After the entire row is loaded to sense amplifiers (t_{RCD}), the subsequent column command will read/write the requested column data ($t_{CAS} + t_{BURST}$) and move them to the DRAM controller.

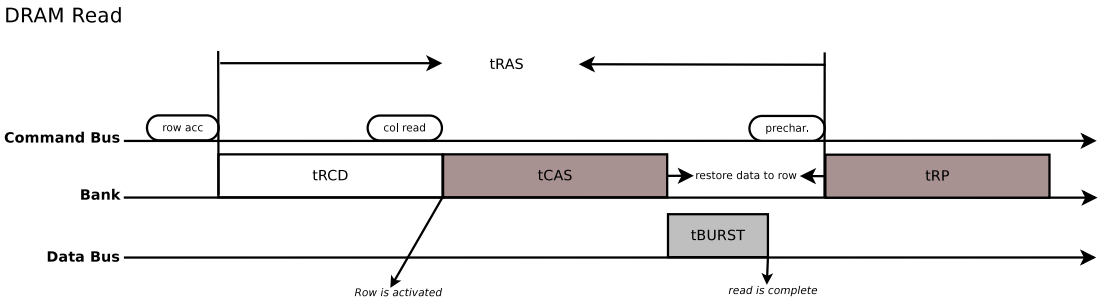

Figure 2.10: A complete read access in DRAM

Write Cycle A DRAM write cycle in DRAM system is similar to the read cycle. However, there are few differences. As shown in Figure 2.11, instead of a t_{CAS} after t_{RCD}, a column write delay (t_{CWD}) is required before transferring data (t_{BURST}) to the row buffer. After data transfer, t_{WR} is representing for the time that writes data back to the DRAM arrays.

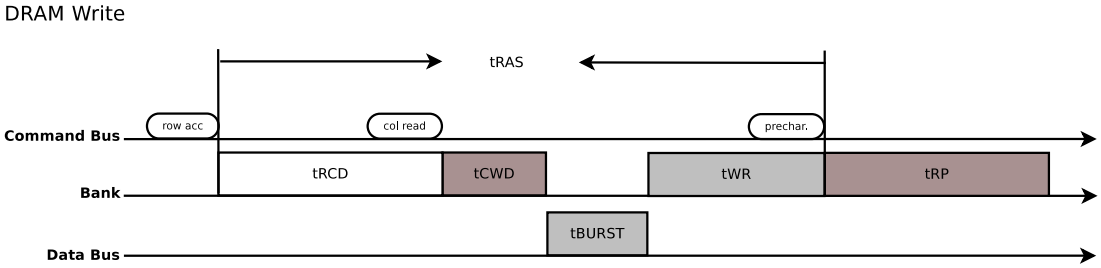

Figure 2.11: A complete write access in DRAM

2.3.4 Bus Turnaround

In a DRAM system, the DRAM bus can be used to service a read or a write request at any given time. Switching the bus between read and write modes is known as

turnaround, which incurs a latency known as turnaround delay. Typically, write to read turnaround delay is called t_{WTR} and read to write turnaround delay is called t_{RTW}.

Write caching. To avoid turnaround overheads, conventional DRAM schedulers commonly store read and write requests in separate queues — namely, read queue and write queue. Read queue will be served with a higher priority since read requests are usually in the critical path of system performance. On the other hand, writeback requests are handled by a passive flushing scheme. The simplest scheme is only to service the write queue when the write queue is close to full. By prioritizing reads over writebacks, the DRAM controller can avoid turnaround overheads and enhance performance.

2.4 DRAM Cache

As mentioned in §2.1, using die-stacking technology to implement a DRAM cache is widely studied in recent works. Due to the larger capacity, prior works [3, 8, 15] have proposed to use stacked DRAM as a large cache inside the processor chip. By exploiting the high density of stacked DRAM, it is possible to have multiple hundreds of megabytes of on-chip DRAM cache. Ideally, we want to architect this large cache with same cache block size as conventional L1 or L2 (which typically use a block size of 32/64 bytes). Consequently, with multiple hundreds of megabytes of DRAM cache, the overhead of storing tags of conventional blocks requires tens of megabytes. In this case, because SRAM has low density characteristics, it is unlikely [3] to use SRAM, which provides faster access latency, as storage media. On the other hand, tag latency is a critical factor to cache design, as this latency would be added to the critical path of the total latency, no matter a hit or miss. Therefore, recent DRAM works are divided into two directions — block-based and page-based DRAM caches.

Page-based DRAM cache: A page-based cache (i.e. a much larger cache block size of 2KB to 4KB) has been proposed in several works [14, 15]. With a larger block size, the tag overhead is reduced to hundreds of kilobytes or few megabytes depending on the cache size. With this cache design, there are two major challenges that prior works have addressed. First, a larger cache line size means more data to fetch on a miss. A DRAM cache miss needs to fetch data from a low-bandwidth off-chip memory. This downside becomes one of the performance bottlenecks of a page-based design. Second, a larger cache line might fetch a significant amount of unused data to the cache. This decreases the effective capacity of the DRAM cache compared to a conventional block size design. To solve the bandwidth problem, Jevdjic et al. [14] proposed a foot-

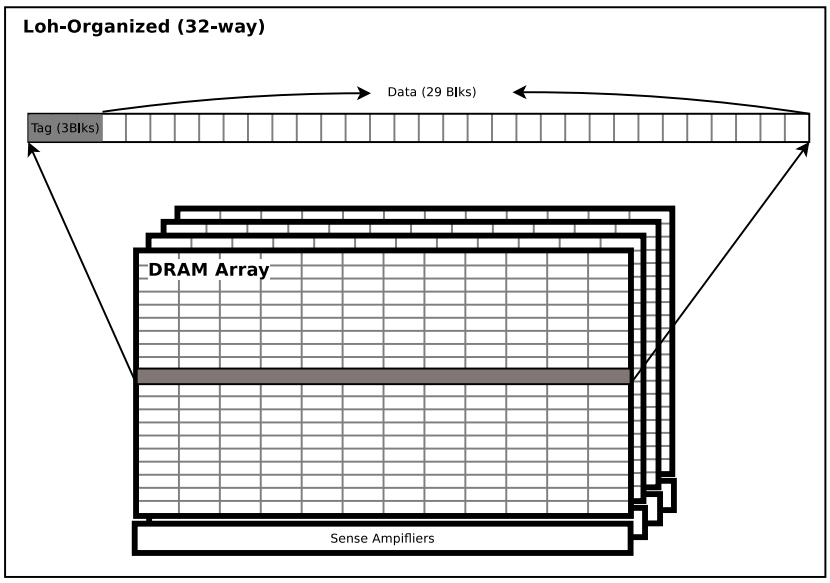

Figure 2.12: Block-based DRAM cache in Loh and Hill's work [3]

print cache which removes the unused blocks from the (page-based) cache line and shows an effective reduction in bandwidth. Their design enables paged-based DRAM cache to outperform block-based caches (tags-in-DRAM, with MissMap) in server workloads [7]. However, in their study, desktop workloads (SPEC 2006) shows about 25% slowdown compared to block-based DRAM cache (with MissMap, 256MB). This is because the spatial footprint in desktop workloads is generally lower than server workloads.

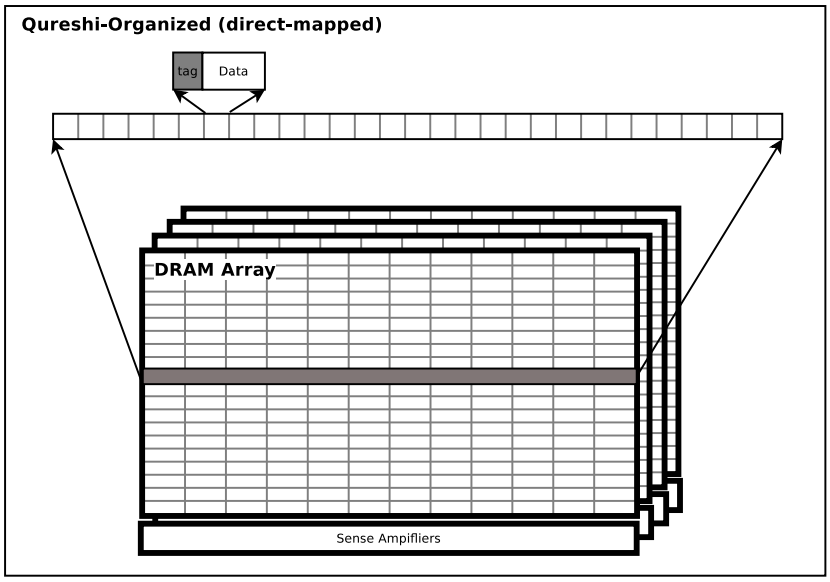

Figure 2.13: Block-based DRAM cache in Qureshi and Loh's work [4]

Block-based DRAM cache: Loh and Hill [3, 13] proposed a DRAM cache with conventional block size (64 bytes). In their work, they embedded tags along with their data in the same row buffer (tags-in-DRAM). They architect a 29-way DRAM cache with a 2KB row buffer (29 tags and 29 blocks) as shown in Figure 2.12 (a). One issue with this design is that DRAM cache misses have to pay the high cost of accessing the tags from the DRAM; consequently they also proposed a MissMap, which is an SRAM structure that tracks the contents of DRAM cache to skip miss accesses. Since MissMap consumes a reasonably large amount of SRAM (in the order of few megabytes) to maintain the required information, subsequent works [4, 9] proposed miss predictors to predict cache misses with a lower SRAM overhead (in the order of few kilobytes). Subsequently, Qureshi and Loh [4] pointed out that instead of a sequential access of the data and tag, one could use a wider data width (72 bytes) to read tag and data in parallel as shown in Figure 2.13. In this design, the hit latency of a direct-mapped DRAM cache is close to the latency of accessing only the data. Whereas this approach works great for direct-mapped caches, the downside of this design, however, is its inflexibility in scaling to set-associative caches. To support a 4-way cache, for example, every access requires additional three tBURSTs for reading three more blocks from DRAM cache; assuming a 2.5 ns tBURST for reading a 72-byte block, the additional latency overhead of supporting a 4-way cache is 23 cycles for a 3 GHz processor.

Part II

SRAM Cache Optimization

Chapter 3

Critical-words-Only Cache

3.1 Introduction

As we have shown in the background (§2.1), current processors have multiple levels of caches. Considering that SRAM is a limited resource inside the processor chip, choosing the right cache size for each level is critical for performance. In general, the first-level cache (L1) is typically small, in order to match the speed of the processor. The lower level caches, on the other hand, are typically large, in order to reduce capacity misses.

How well a lower level cache is able to reduce capacity misses depends on how well the working set fits into the cache. Ideally, the cache should be large enough to fit the working set, but situations may arise in which the size of the lower level cache is significantly lesser than the working set. For instance, several modern multi-core processors (including Intel Nehalem and Sandy Bridge) choose to support L2 caches that are private to each core; although it would be desirable to have a large private L2, this might not be possible, as increasing the size of per core L2 has a multiplicative effect on the overall area. Indeed, the above processors support only 256 KB of L2 cache per core.

Does the L2 provide any performance benefit to the overall system? In general, we would like the L2 cache to reduce capacity misses from the L1 and hence, improve the overall performance. Therefore, the answer to this question will lead to "is the L2 large enough to avoid capacity misses?" Figure 3.1 shows the miss-rates [1] incurred by SPEC benchmarks as the size of the L2 cache is varied from 128 KB through 4 MB.

[1] For clarity, only six programs are shown, but the average is across the 33 programs from the SPEC2000 and SPEC2006 benchmarks we were able to run. Refer to §3.4 and §3.5 for comprehensive results.

As we can see, with a 256 KB L2 cache, the average miss-rate is greater than 50%. In fact, for programs such as *galgel* and *omnetpp* which have a miss-rate of over 90%, we found that bypassing the L2 cache and directly accessing the LLC can be more performance-efficient (§3.5) than accessing the L2 cache. This is because accessing the L2 in such situations ceases to provide any performance benefit since the L2 miss-rate is high; to make matters worse, it also adds the latency of a wasteful L2 access to the critical path. As the size of L2 is increased, however, we observe that the miss-rate significantly decreases in some benchmarks.

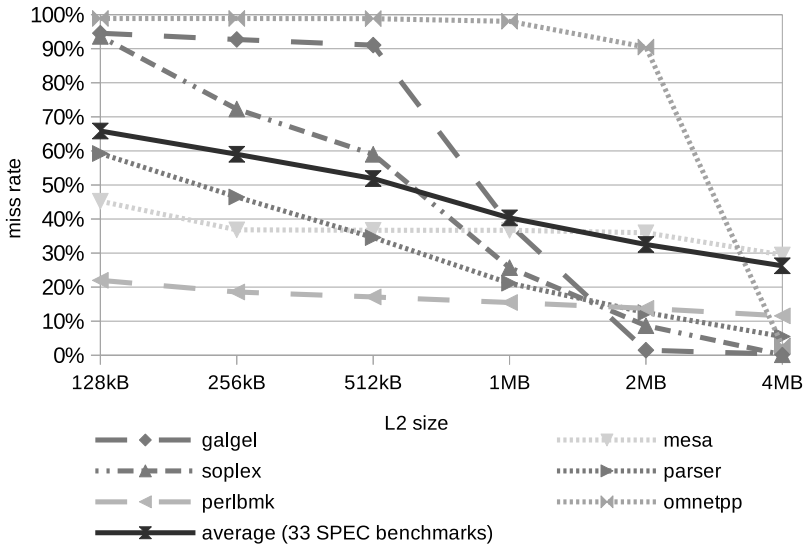

Figure 3.1: Miss rate versus L2 cache size: as the size of the L2 cache (8-way) is increased from 128 KB through 4 MB, the local miss-rate significantly decreases. For this experiment 32 KB (4-way) L1 cache was used.

From the above example, it is clear that a lower level cache that is too small compared to the working-set size could potentially hurt performance. In such situations, is there a way in which the cache can be used more effectively? In this work, we propose a cache design for increasing the effective cache capacity. Although our design is applicable for any lower level cache, i.e. any cache other than L1, in this work we restrict ourselves to L2 caches.

We observe that not all words belonging to an L2 cache block are accessed around the same time – a subset of the words, in fact, are consistently accessed sooner than the others. We refer to the former as *critical words* and the latter as *non-critical words*. Our key idea stems from the realization that, if the time interval between the critical word accesses and the non-critical word accesses is at least as high as the latency of

accessing the lower level cache (or memory), then an L2 that caches *only the critical words* of each block could potentially perform as well as a conventional L2 that caches the full block. This is because, whenever any of the critical words in L2 is accessed, the L2 cache could additionally request the full cache block from the lower level and have it sent to the L1 cache – in time before any of the non-critical words are accessed by the processor. We call such a cache design that stores only the critical words for every block, *critical-words-only cache (co-cache)*, and we denote the number of critical words per block as the *depth* of the co-cache. In the ideal case, a co-cache of depth 1 will perform as well as a conventional cache with full-sized blocks.

```
1  int X[8];
   ...
3  while(exit)
   {
5    ...
     for (int i=0;i<8;i+=2)
7    {
       in1 = X[i];          //load X[0],X[2],X[4],X[6]
9      in2 = X[i+1];        //load X[1],X[3],X[5],X[7]
       Compute(in1,in2);    //100 cycles of execution
11   }
     ...                    //X is evicted from L1!
13 }
```

Code 3.1: Example to illustrate co-cache idea.

An Example. We illustrate our idea with a simple example is shown in Code 3.1. For this example, let us assume a two-level cache organization with a block size of 8 words. Furthermore, let us assume that the cache block containing array X is cached in L2 throughout, but is only cached in the L1 when executing the inner *for* loop (as it is evicted from the L1 soon after). Consequently, every time $X[0]$ is loaded within the inner *for* loop, it causes an L1 miss, which in turn causes block X to be transferred from L2 to L1 as shown in Figure 3.2 (left). Then, $X[0]$ and $X[1]$ are loaded, and the loaded values are used to perform some computation that takes 100 cycles. Subsequently, $X[2]$ and $X[3]$ are loaded (and so on).

In this example, $X[0]$ and $X[1]$ constitute the *critical words*, as these are accessed sooner than the other words in the block. This example sheds light on the reason behind this observation of criticality: typically not all words within a block are accessed

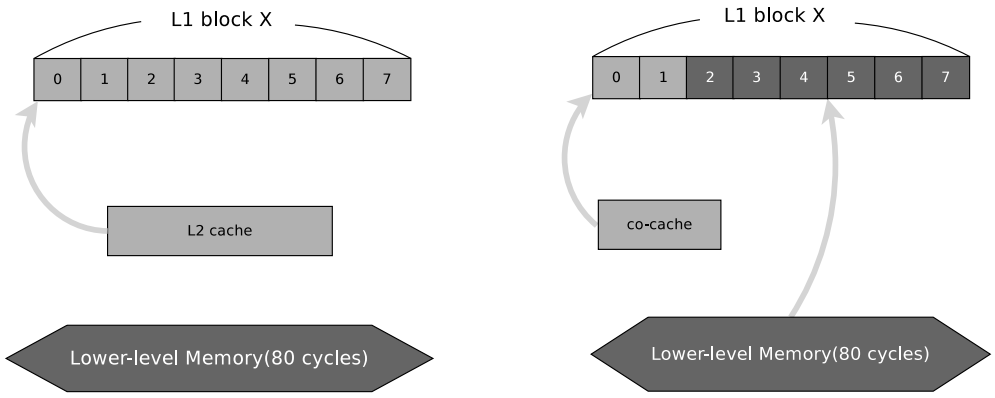

Figure 3.2: **(left)** L1 + conventional L2: (1) $X[0]$ causes a miss in L1 (2) the block is fetched from L2 (3) $X[0], X[1]...X[7]$ hit in L1. **(right)** L1 + L2 as co-cache of depth 2: (1) $X[0]$ causes an L1 miss, accessed in both co-cache and lower-level (2) The first 2 critical words are fetched from co-cache (3) $X[0]$ and $X[1]$ is accessed from L1 (4) The memory returns the full block into L1 in time for second iteration of for loop(5) $X[2]$ ... $X[7]$ hit in L1.

together; often one or two words are loaded and the loaded values are used to perform some computation before loading the subsequent values.

It is also important to note the above pattern repeats itself across each iteration of the outer *while* loop. The example also sheds light on the reason for this repeatability: often, like in this example, the same piece of code is responsible for causing misses to a given block in the L1 cache. Consequently, critical words for a particular block are often accessed in regular patterns.

Figure 3.2 (right) illustrates how an L2 that is organized as a co-cache of depth 2 achieves the same performance as that of the conventional L2 with eight words per block. When $X[0]$ is accessed in the co-cache, it triggers a request to fetch full block X from memory. By the time the processor requests $X[3]$ after performing the computation (which takes 100 cycles), the block would have been fetched from the memory, as the memory latency is 80 cycles.

Contributions and Chapter Organization. In this chapter, we consider the problem of how best to use a lower-level cache whose size is much smaller than the working-set size. Our contributions are as follows:

- We observe that a subset of words from a cache block – the critical words – is consistently accessed sooner than others. We demonstrate this with the help of an empirical study (§3.2), in which we also demonstrate that the above critical words are predictable.

- We exploit this observation by proposing a cache design called co-cache (§3.3), which is a lower-level cache comprising only the critical words of every cache block, and thus uses the available cache size more effectively.
- We show how the L2 can be engineered as a co-cache (§3.3.2). For predicting critical words, we have a simple *critical words predictor* (§3.3.1), in which we add additional tag bits to the L1 cache (less than 500 bytes overhead) for remembering the order in which words are accessed; thus our design requires no complex prediction table.
- Whereas the co-cache is effective in situations where the size of the L2 is significantly smaller than the working-set size, a conventional cache might perform better in situations where the working-set fits in the cache. For this reason, we propose the adaptive co-cache (aco-cache), a scheme for reconfiguring back to a conventional cache if the working set is determined to fit within L2.
- We describe our evaluation methodology in §3.4. Among other things, we discuss how we model the co-cache access latency (§3.4.1) and the space overhead of the co-cache (§3.3.4).
- Our experiments (§3.5) with SPEC2000 and SPEC2006 benchmarks show that a 256 KB L2 used as a co-cache (aco-cache) of depth 2 can achieve up to 36.8% (36.8%) performance improvement and on average 5.3% (6.1%) improvement compared to a conventional 256 KB L2 cache. As for multi-core workloads, the co-cache (aco-cache) in a 4-core system shows up to 29.3% (29.3%) improvement and on average 7.3% (8.1%) improvement across 32 randomly generated workload groups.

3.2 Study and motivation

Clearly, the effectiveness of our idea hinges on whether or not each L2 cache block can be split into critical and non-critical components. In addition to this, we should be able to predict the critical words for each cache block – only then would we be able to know what to cache in the proposed co-cache.

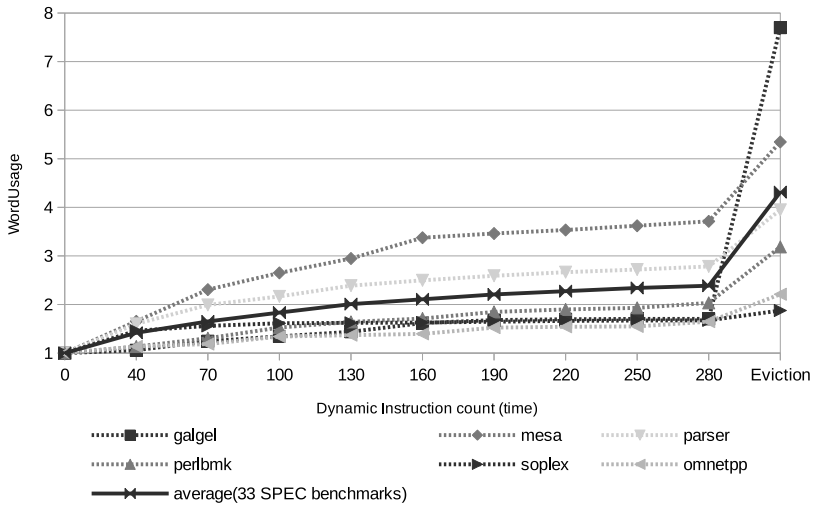

Figure 3.3: The word usage in different time intervals (dynamic instruction count); the time at which the first word is accessed is taken to be 0. Eviction in the graph refers to the time in which the block is replaced from L1.

3.2.1 Critical Words vs Non-critical Words

We conducted two experiments[2] to ascertain whether, for each L1D cache block, a subset of words (the critical words) are consistently accessed before the others (as opposed to all words being accessed around the same time).

Word Usage. We first perform a study on the number of words are used (word usage) in L1 data cache blocks. In this study (Figure 3.3), we measure the word usage of a block in different time intervals (between the insertion of this block and N dynamic instructions afterwards). We report the average word usage across all blocks. The "Eviction" in the figure means the word usage at the time where blocks are replaced/evicted. As we can see from Figure 3.3, the average word usage across all benchmarks is only 2 at 130 dynamic instructions, while 4.3 words end up being accessed at the time of eviction. In particular, galgel has a word usage of 7.8 at the time of eviction, but the usage is only 1.4 at 130 dynamic instructions. This result provides a good evidence that words in a block are accessed in different time intervals which are quite apart from each other.

Study for N critical words. We run another experiment to study the potential of the critical words idea. Similar to the previous study, we timestamp the cache block at its insertion into the L1D cache. Then at every subsequent word access in the same cache

[2]In both experiments, we simulate a 4-way 32KB L1 with 64-byte blocks and run across 33 SPEC benchmarks. We use the dynamic instruction count as the measure of time.

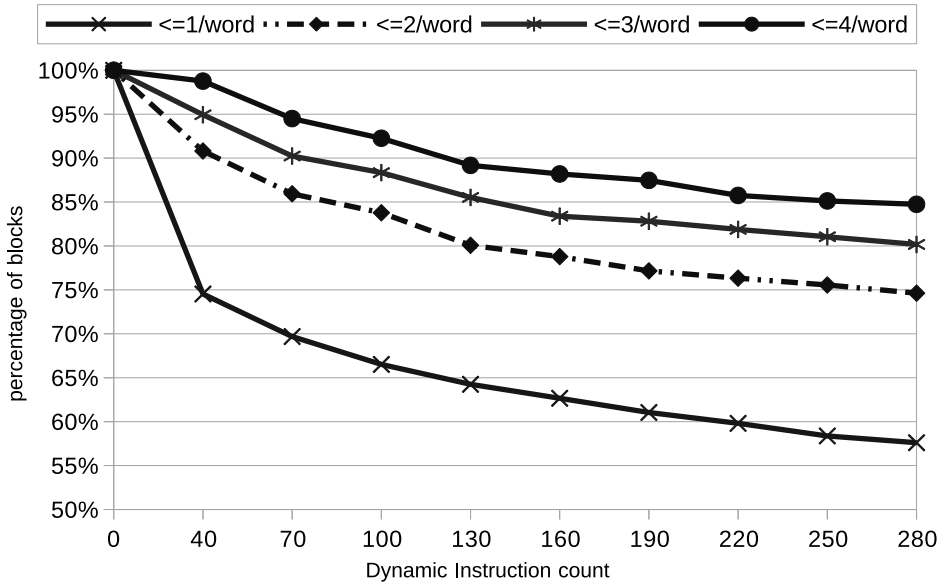

Figure 3.4: The percentage of blocks that has accessed less than or equal to N words with time.

block we measure the time (in dynamic instruction count) elapsed from the initial timestamp. Each line in Figure 3.4 shows the fraction of blocks with N number of words touched and its variation over time. As shown in the figure, we found that 75% (80%) of L1D blocks has only touched up to two words (<= 2/word) at the 280th (130th) instruction. Assuming a CPI of 1, this would mean, 75% blocks only use the first two words within the 280 cycles. It is worth noting that, "<=1/word" decrease 25% after 40 instructions and more than 35% of the block already touch more than one word at the 130th instruction. We can infer from this study that the first 2 words are generally accessed much earlier than the rest of the words (non-critical words); in other words, there is very good evidence for the first 2 words being the critical words. The reason for this behavior can be explained by the fact that applications typically read a couple of words, after which they would likely use the above words that are read to perform some computation, before reading the next set of memory words, and so on.

3.2.2 Critical Words Predictability

To check if the critical words are predictable, we remember for every memory block, the word address that caused the L1 miss (the 1st critical word) and also the word subsequently accessed (the 2nd critical word). When the same memory block results

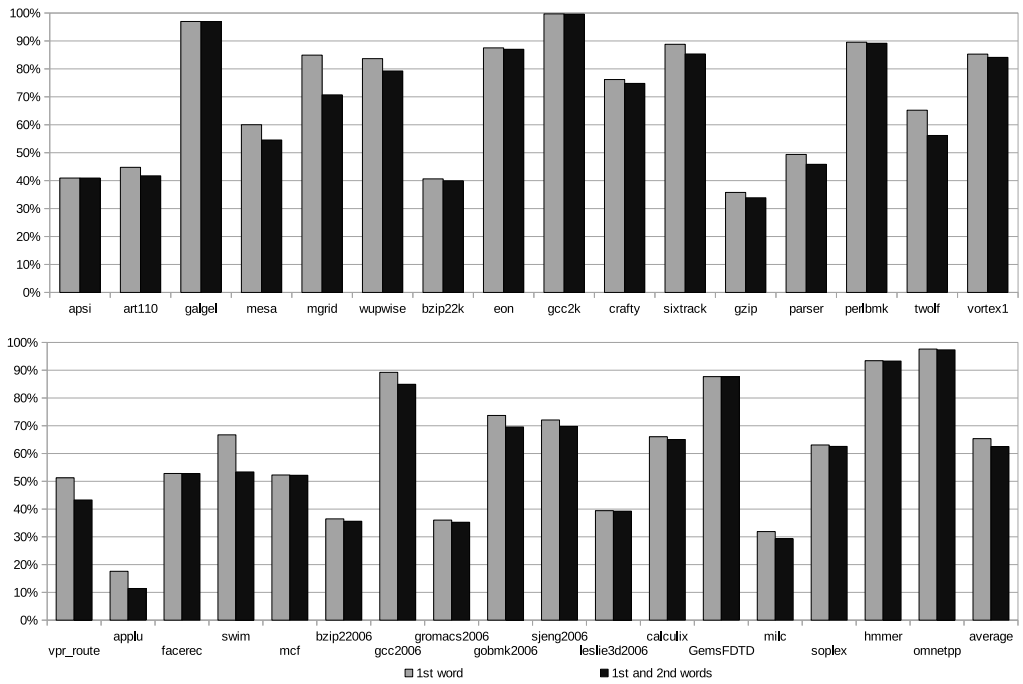

Figure 3.5: Predictabilities of the first critical word, and both first and second combined.

in an L1 miss and is brought back into the L1 again, we compare the recorded addresses with the actual addresses of the 1st and 2nd critical words.

As we can see from Figure 3.5, the average predictability of the 1st critical word is 65.4% across all benchmarks. For some benchmarks, such as *galgel* and *omnetpp* the critical words are extremely predictable with close to 100% predictability. On the other hand, for benchmarks such as *mesa* and *parser* the predictabilities are in the order of 60.0% and 49.4% respectively. It is important to note, however, that even for benchmarks with relatively poor predictabilities, they are still significantly better than the expected predictability of 12.5% if the critical words were uniformly distributed. We also measure the percentage of times *both* critical words are predicted correctly – as we can see this is about 62.5% across all benchmarks. From this, we can infer that, in general, when the first critical word is predicted correctly, the prediction for the second critical word is also correct. The reason for this behavior can be explained by the fact that in most cases, the same piece of code is responsible for causing L1 misses to a given memory block. Consequently, the critical words are accessed in a regular pattern and hence predictable.

3.2.3 Useful Words vs Critical Words

Despite the presence of spatial locality, not always do all the words belonging to a cache line end up being used; a number of techniques [29, 30, 31, 32, 33] leverage this observation to improve cache performance. For instance, Line Distillation [31] attempts to discard such unused words in a cache line to improve cache capacity. While our idea of exploiting critical words is closely related to the idea of exploiting useful words, there is one crucial difference. Techniques that exploit useful words benefit from cache lines in which not all the words in the cache line are used, i.e. the more unused words, the better. In contrast, our idea exploits the factor of time; for example, even if all the words from a cache line are used, we can still benefit as long as some of the words in this cache line are accessed sooner than others. We conduct an experiment to find out, on average, how many words are accessed in a block before its eviction (i.e. word usage of the block). As shown in Figure 3.6, the average word usage is 4.3 words per block across all benchmarks, which amounts to 53.8% of the original block size. Intuitively, this ratio corresponds to the maximum benefit that is possible by exploiting useful words (the cache that is about half the size of the original cache can perform as well as the original cache). On the other hand, the benefit that can be derived by exploiting critical words is dependent on the latency of accessing the lower level memory: the faster lower level access latency, the less number of critical words is required. Figure 3.6 shows the word usage for various latencies. As we can see, even with as high latency as 160 cycles, the number of critical words (and thus the average word usage) is 2.1 words, which amounts to 26.25% of the original block size. From the above study, it is clear that exploiting critical words has a potential for greater savings in comparison to useful words.

3.3 Methodology and Design

A critical component of our design is the *critical words predictor*, which we introduce first. Next, we show how to engineer the L2 cache as a *critical-words-only cache (co-cache)*. A conventional cache might perform better than the co-cache if the cache size is large enough to fit the working-set; for this reason we introduce a reconfiguration scheme called *adaptive co-cache* (aco-cache), for dynamically choosing between co-cache or conventional cache based on the workload's miss-rate. Finally, we discuss the area overhead of our design.

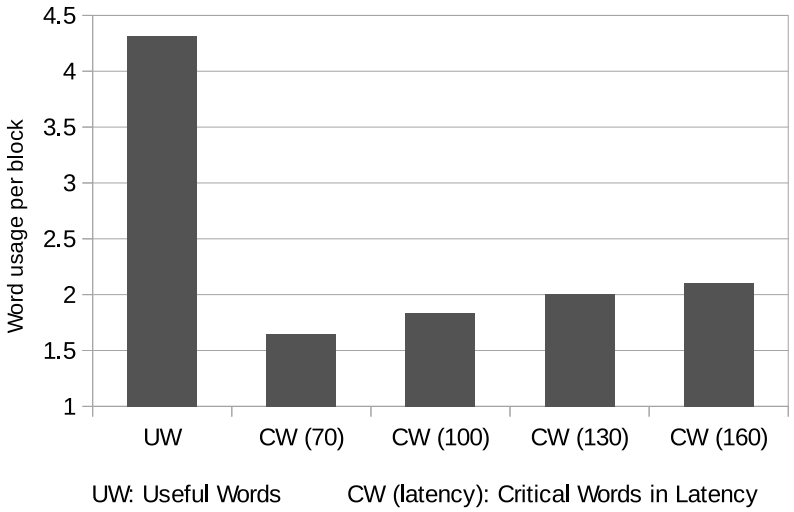

Figure 3.6: Useful words filtering versus critical words filtering

We want to emphasize that, considered in itself, our hardware design is neither sophisticated nor does it propose anything new with regard to prediction or reconfiguration techniques. Our design is fairly straightforward using well-known ideas. However, we consider this as a positive – the fact that our critical word caching approach can be realized using relatively simple hardware.

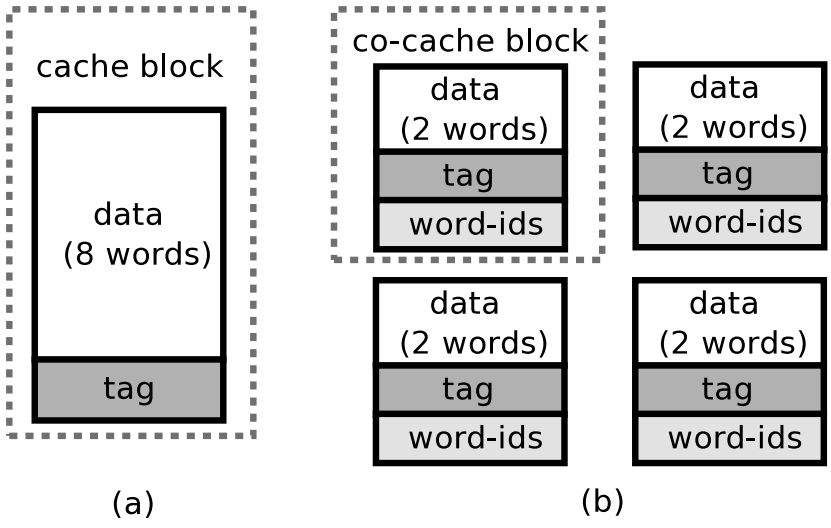

Figure 3.7: (a) cache block in a conventional cache (b) 4 blocks in a co-cache

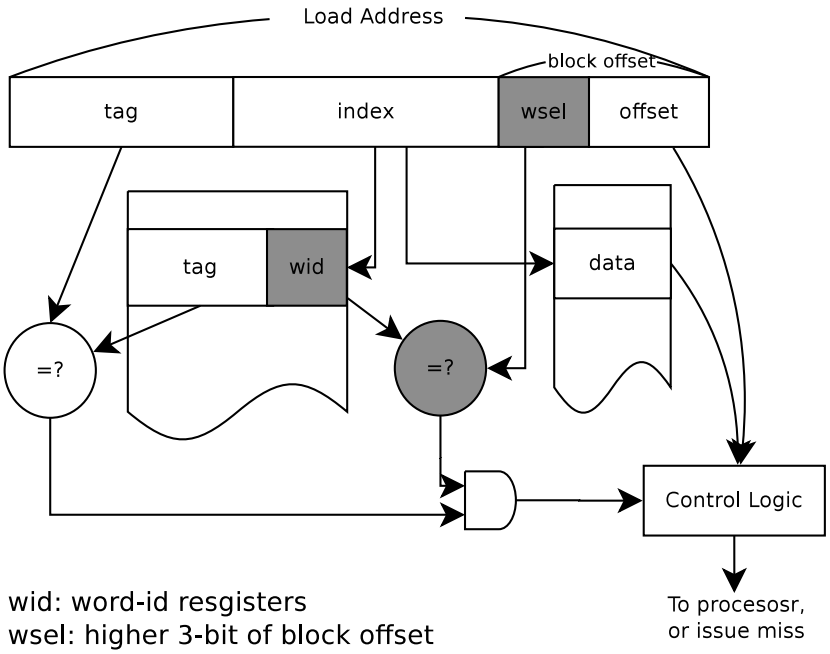

Figure 3.8: Accessing a co-cache block

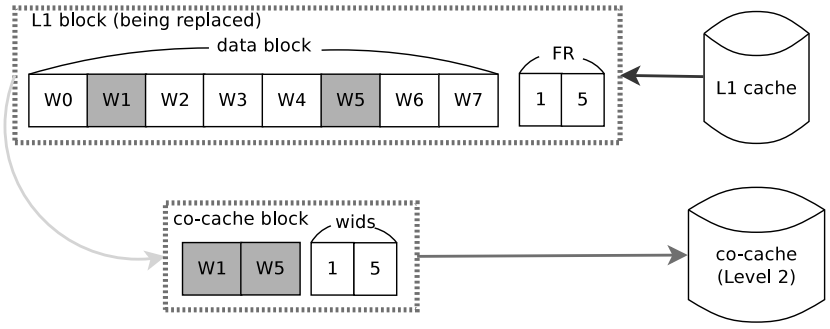

Figure 3.9: Block placement – FR: Footprint registers (§ 3.3.1)

3.3.1 Critical Words Predictor

We use a simple predictor, in which we simply remember the order in which the words of a cache block are accessed in the L1 and use it as our prediction – our approach is similar to the predictor used in [31]. To remember the identities of the critical words, we add additional tag bits – the *footprint registers* – to each L1 cache block. Assuming the processor word is 64-bit, and the cache block size is 64-byte (i.e. each block contains eight words), each block will need 3-bit (footprint register) for remembering one critical word. To support the depth of 2 co-cache (which means we need to remember two critical words per block), we need two footprint registers per L1 block – which only amounts to 384 bytes overhead for a 32 KB L1 cache. In general, our pre-

dictor design does not require any prediction table, and its area overhead is relatively minimal.

3.3.2 Co-cache Structure and Implementation

The co-cache is a cache organization for caching only the critical words for every cache block. Its organization is similar to a traditional cache, except in the following respects. For the purpose of this discussion let us assume all caches use write-back policy:

- **Checking for hit/miss**. Since the co-cache caches only the critical words from each cache block, we need to be able to identify what words are cached. For this purpose, each block in the co-cache is associated with additional *word-ids* which stores the identities of the words that are currently cached, as shown in Figure 3.7. Note that the word-ids would also need to be compared as part of tag checking as shown in Figure 3.8. In other words, adding the word-ids has the same effect of increasing the tag size of each cache block. We model the delay of this increased tag size in our experiments (§3.4) and find that this is negligible.
- **Accessing the co-cache**. It is worth noting that even if the requested word is found in the co-cache, the request has to be forwarded to the lower level in order to bring the remaining (non-critical) words to the L1. Consequently, upon an L1 miss, both the L2 (functioning as a co-cache) and the lower-level are accessed. If the requested word is found in the co-cache, the requested word, and the other critical words are sent to the L1; later, the rest of the words would arrive from the lower-level. On the other hand, if the requested word is not found in the co-cache, the processor will have to wait until the whole block arrives from the lower-level. Thus, accessing the co-cache proceeds as detailed in Table 3.1.
- **Updating the co-cache (block placement)**. When *a cache block is replaced from the L1*, we need to decide what words from the block need to be cached in the L2 co-cache. We use the output of our critical words predictor (*footprint registers*) and only cache those words that are marked as critical (as shown in Figure 3.9). It is worth noting that all blocks in the L1 should have at least one access that caused the block to be fetched, so the *first* footprint register cannot be empty.
- **Mis-prediction handling**. It is worth noting that since the processor forwards the request to lower level memory for the full block data, this would automatically handle the case in which the critical words are mispredicted.

1	If there is a miss in L1, the fetching request is sent to both co-cache (for partial block containing the critical words) and the L3 (for the whole block).
2	If the co-cache access is a hit, the partial block in the co-cache is brought back to L1 and satisfies processor's request. If the co-cache access is a miss, the processor waits for the whole block to be brought from L3.
3	When the requested (whole) block returns from L3, it will be placed in L1, but not in the co-cache. If the co-cache access had resulted in a hit (in step 2), the whole block will replace the previously present partial block in L1.

Table 3.1: Accessing the co-cache

- **Handling instruction accesses.** In the co-cache, we bypass the L2 for the instruction accesses and directly access the lower level cache (i.e. L3). This is because, for an instruction cache, all words in a cache block are usually accessed in a short period[3]. Alternatively, all words are critical. Therefore, caching a partial instruction block will not provide much benefit. It is worth noting that, in this work, we focus on the desktop workloads (33 SPEC benchmarks) whose data working-set size does not fit in the L2. In such workloads, we found the L2 is thrashed by the data accesses and incurs high miss-rate for instructions. Therefore, bypassing the instruction accesses in co-cache will not result in much performance loss because it would have been a miss in the L2.
- **Handling write backs**. When the L1 writes back dirty data to the L2 co-cache, since the co-cache cannot hold the full block, the dirty block will also have to be written to the L3.
- **Cache coherence**. In contrast to conventional cache coherence (with local L1s/L2s and a shared L3), ensuring cache coherence with an L2 co-cache involves one minor change. When a block with read-write permissions is evicted from the L1 since the co-cache cannot hold the full block, the co-cache will in turn have to write the block back to the L3. However, in doing so, the block in L3 will have to be maintained in the read-only state as the co-cache continues to hold part of the block.

[3]Unless the instruction block contains a taken branch, the fetch unit of the processor will prefetch instructions sequentially.

3.3.3 Adaptive Co-cache

A co-cache targets situations in which the size of L2 is significantly smaller than the working set size. However, when the working-sets size does fit in the cache, a conventional cache might perform better. Thus, it would be beneficial to have a scheme for switching back to a conventional cache if the working-sets fit in the L2. We call this *adaptive co-cache (aco-cache)*.

Obtaining miss-rates. This design requires that we obtain the miss-rate of a conventional cache while we are operating in the co-cache mode. To achieve this, we make use of shadow tags [34, 35] to monitor the cache miss-rate. In contrast to a conventional cache, the shadow tags only perform tag operations – i.e., they only keep the information of blocks but not their data.

Scheme. In our proposed aco-cache scheme, we start in co-cache mode but keep recording the miss-rate of the conventional cache with the shadow tags. When the program encounters a reconfiguration point, we compare the miss-rate with a pre-defined threshold; if the measured miss-rate is lower than a threshold, we reconfigure the cache back to conventional cache mode (in §3.5.2, we perform detailed sensitivity studies to determine this threshold). Regarding the reconfiguration points, these can be placed either by the software (operating system or compiler) or the hardware. In aco-cache, we assume the latter – specifically, we assume *a hardware monitoring unit* [35, 36] which simply checks the miss rate every fixed number of instructions. It is worth noting that the OS will take care of reverting L2 to co-cache again if an application is terminated or encounter the context-switch.

Reconfiguration: hardware. In principle, performing this reconfiguration is as simple as coalescing the (partial) blocks of the co-cache and turning off [37] the additional tags and word-ids; for instance, four blocks of the co-cache can be combined into one conventional block in the example shown in Figure 3.7 (b). There are two ways to do this: set-based reconfiguration [37, 38, 39] or way-based reconfiguration [40]. While way-based reconfiguration will require higher associativity(e.g. an 8-way cache will convert to a 32-way co-cache), it will also increase the tag access latency. To avoid this, in this work, we use the set-based reconfiguration that configures four co-cache sets to 1 conventional cache set.

Reconfiguration: single vs multiple transitions. Our reconfiguration scheme allows only a single transition: from the co-cache to the conventional cache (if the working set fits L2). Another alternative is to enable multiple back-and-forth transitions between

L1 Cache	33.5 KB
L2 cache	267 KB
L1 overhead (predictor+word-ids)	0.75 KB (2.2%)
L2 overhead (w/o shadow tags)	36 KB (13.4%)
L2 overhead (with shadow tags)	44 KB (16.4%)
L1 + L2 overhead (w/o shadow tags)	36.75 KB (12.2%)
L1 + L2 overhead (with shadow tags)	44.75 KB (14.8%)

Table 3.2: Area Overhead

the co-cache and the conventional cache depending on the program phase behavior. However, we chose the former, primarily because of the cost associated with a single reconfiguration transition.

Reconfiguration involves the following costs. First, the cost of flushing all dirty L2 blocks [4]. Second, invalidating the L2 blocks that were flushed. Third, updating the L3 coherence directories. Fourth, performing the circuit changes associated with the reconfiguration [38]. Since the L2 cache is larger than L1, the reconfiguration cost of the above steps could be more expensive than Yang's work [38]. For a 256 KB L2 cache, the conventional cache and co-cache have 4,096 and 16,384 blocks respectively. Assuming it costs 100 cycles to flush an L2 block and 30 cycles to update an L3 directory, the overheads to flush sets and update directory would be in the order of 400000 cycles. In addition to that, a single transition incurs the cost of associated compulsory misses. Therefore, frequent transitions between the co-cache and the conventional cache could hurt performance. In contrast, if we perform only a single transition, we find in our experiments that the cost of reconfiguration pales into insignificance.

3.3.4 Space Overhead

Each cache block in the co-cache requires additional space for the word-ids. Also, with the same data array size, a co-cache has more sets compared to a conventional cache – consequently, more tag and status registers are required for those extra sets. On the L1 side, word-ids are again needed to identify the words brought from the co-cache, in the transition period before the full block is back from the lower-level. Besides, footprint registers are also required to remember the critical words for prediction. Thus, for a

[4] Since all blocks are read-only in co-cache (§3.3.2), flushing operation only applies when transitioning from conventional cache to co-cache

Processor	ALPHA, 3GHz, 4-core, 4-wide, inorder
L1 I/D caches	each 32 KB/4way, 2-cycle
L2 cache	256 KB (512 sets/8 way), 12-cycle
Co-cache	256 KB (*2048 SETS/8 WAY*), 12-cycle (3.4.1)
Co-cache depth	2
Reconfiguration threshold	40%
Reconfiguration point	every 5 million instructions
ToL3 bus	12 GB/s, single core 48 GB/s, four cores
L3 cache	16MB/16way, 4MB per core, 40-cycle
Memory	DDR3-2000, 9-9-9
Memory Bus	1GHz, 8-byte, single core 2GHz, 8-byte, 4 cores
Benchmarks	1.apsi 2.applu 3.art 4.bzip22k 5.crafty 6.eon 7.facerec 8.galgel 9.gcc2k 10.gzip 11.mcf 12.mesa 13.mgrid 14.parser 15.perlbmk 16.sixtrack 17.swim 18.twolf 19.vortex1 20.vpr 21.wupwise 22.bzip2 23.gcc 24.gromacs 25.gobmk 26.sjeng 27.leslie3d 28.calculix 29.GemsFDTD 30.milc 31.soplex 32.hmmer 33.omnetpp

Table 3.3: Architectural Parameters

co-cache with a depth of 2, every block in L1 cache needs two word-id registers and two footprint registers. If we consider a 32 KB L1 (4-way, 64B block), a 256 KB L2 (8-way, 64B block), and 5-bit status registers, as shown in Table 3.2 the additional overhead of using the L2 as a co-cache amounts to 36.75 KB (including the predictor). An aco-cache additionally requires shadow tags to monitor the cache miss-rate, which requires around 8 KB space, bringing the overall overhead to 44.75 KB overall. The co-cache (aco-cache) only adds 2.2% (2.2%) overhead to L1 and 13.4% (16.4%) overhead to L2. Thus, the overall (L1 + L2) overhead is 12.2% (14.8%).

miss-rate	256 KB	512 KB
A (>70%)	sixtrack wupwise mcf galgel mgrid applu milc omnetpp facerec GemsFDTD gcc2k art soplex swim	sixtrack wupwise mcf galgel mgrid applu milc omnetpp facerec GemsFDTD art swim
B (40 - 70%)	bzip22k hmmer vpr twolf gcc apsi leslie3d bzip2 parser	soplex hmmer vpr twolf gcc apsi leslie3d
C (<40%)	eon gzip crafty calculix gobmk vortex sjeng perlbmk gromacs mesa	gcc2k bzip22k parser bzip2 eon gzip crafty calculix gobmk vortex sjeng perlbmk gromacs mesa

Table 3.4: L2 miss-rates in different workloads

3.4 Evaluation Methodology

We implemented our technique in the *gem5* simulator [41] and architectural parameters are shown in Table 3.3. We evaluated our technique across the SPEC benchmark suite (21 from SPEC2000 and 12 from SPEC2006) with ref input. We used all programs except those we could not get to compile and run correctly in our infrastructure. For the simulation, we first fast-forward 1 billion instructions (4 billion instructions for SPEC2006) and warm-up for 100 million instructions. Then, we measure the data for the next 1 billion instructions. Recall that the co-cache is most effective for programs without sufficient L2 cache capacity to accommodate their working sets (i.e., L2 miss-rate is high). Accordingly, we show L2 miss-rates for these benchmarks in Table 3.4. As we can see, with a 256 (512) KB L2 cache, 14 (12) of the 33 programs have an L2 miss-rate of greater than 70%.

Depth of co-cache. In our study, depth 2 of co-cache has about 2% better performance than depth 1 across our 21 SPEC2000 benchmarks. This is because the second request of the block is usually closed to first request (within 40 cycles). However, the third request to the same block is usually taking more than 100 cycles. In other words, depth of 2 gives us the chance to hide 100 cycles miss latency, Although this cannot hide the latency of an LLC miss entirely, it can still avoid more than half of the stall

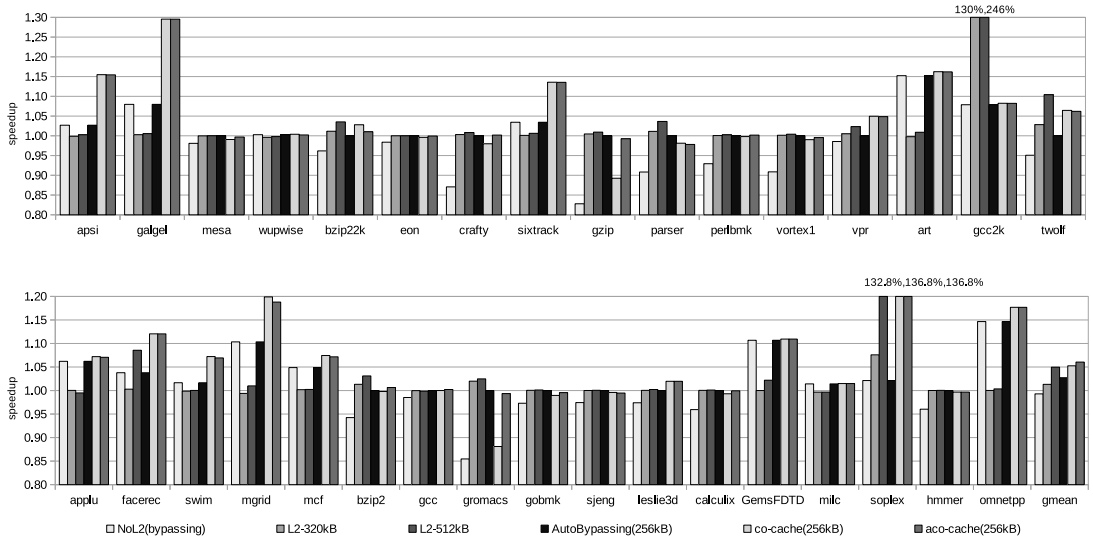

Figure 3.10: Performance Speedup

cycles caused by LLC miss (around 160 cycles in our configuration). Also, depth 2 has lower additional space overhead on L2 cache tags. Compared to depth 2 design, depth 1 requires additional 36KB (i.e. 72 KB in total, which is 26.8% of L2 size) space overhead to L2. Therefore, we used a depth of 2 for the co-cache experiments as we found that this was the depth that gave the best performance/cost results. However, the depth 1 design point might be preferred because it requires no modifications to the L1 (i.e., the critical word is returned to the core while the full-block fill happens later). In this work, we consider this design space exploration as future work.

3.4.1 Co-cache's access latency

The co-cache requires additional tag bits: 6 bits for identifying the critical words, and also the extra tags bits due to the increase in the number of sets. We use CACTI to model the effect of the increased tag size on the access latency. The result shows that the increased tag access latency is negligible (less than 0.1 ns), and can be overlapped with data access like in the conventional cache.

3.5 Results

3.5.1 Performance Evaluation

The primary goal of our evaluation is to compare the performance of the L2 used as a co-cache (aco-cache) with the conventional cache. We evaluate the performance of the following configurations:

- baseline. Conventional cache, 256 KB/8 way/64B.
- L2-320 KB / L2-512 KB. Conventional cache, 320 KB/10 way/64B (resembles the aco-cache area) and 512 KB/8 way/64B.
- bypassing / auto-bypassing. In the bypassing scheme, the L2 cache is bypassed to reduce the access latency to the lower level memory (i.e. No L2 cache). Auto-bypassing automatically selects the better-performing alternative between accessing L2 and bypassing L2 for that benchmark.
- co-cache / aco-cache. The proposed design, 256 KB/8 way/16B.

The performance speedup (compared to the baseline) are shown in Figure 3.10. From the result, we can see that bypassing (No L2) provides an improvement for some benchmarks with high miss-rate (as shown in Table 3.4) such as soplex (2.1%) and galgel (8.0%). In comparison, we observe that co-cache provides much better improvement – for example 36.8% and 29.5% for soplex and galgel respectively. On average, co-cache provides an improvement of 5.3% whereas bypassing incurs a slight slowdown (about 1% slower).

However, in some benchmarks, co-cache is slower than the baseline – for example, gzip is 10.7% slower (17.2% slower for bypassing). This is because the 256 KB L2 in gzip incurs low miss-rate and hence already provides good performance. This is one of the reasons we proposed aco-cache; recall that aco-cache can switch between the co-cache design and the conventional design depending on the miss-rate. With aco-cache, the average improvement is 6.1% that is better than the improvement provided by doubling the conventional cache size (5.0%). Most of the benchmarks that incur a performance loss with the co-cache have now improved significantly with the aco-cache – for example, gzip which was 10.7% slower now performs as well as a conventional cache. Also, note that the improvement incurred with auto-bypassing is only 2.7% and is significantly lesser than the improvement in the co-cache/aco-cache.

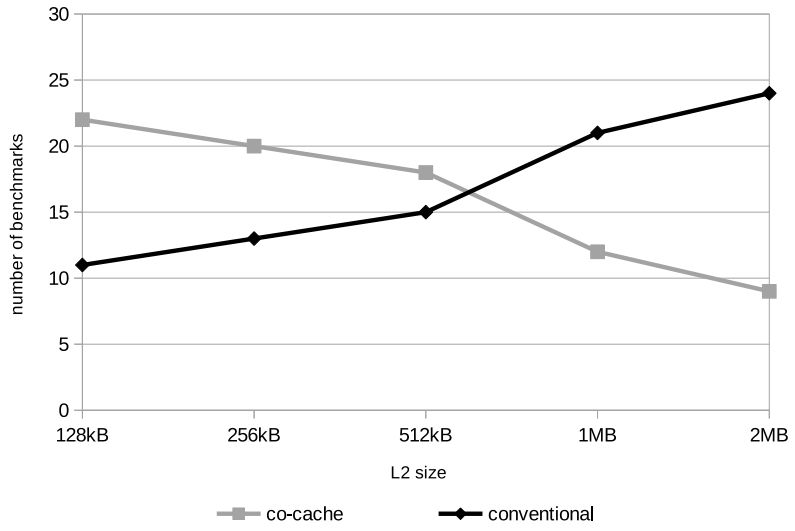

Figure 3.11: Benchmark preference

3.5.2 Adaptive Co-cache

In this section, we motivate why adaptive co-cache (aco-cache) is required by studying the conditions under which a benchmark benefits from a co-cache in comparison to the conventional cache. We also perform sensitivity studies to determine the threshold (§3.3.3) for when the reconfiguration from co-cache to conventional cache must take place.

Benchmark preference. In the performance evaluation section, we observed that the performance in several benchmarks decreased compared to the baseline when we configured the L2 as the co-cache. In Figure 3.11, we show the number of benchmarks that perform better with co-cache than with conventional cache. We call these benchmarks *co-cache-preferring* and the others as *conventional-preferring*. As we can see in Figure 3.11, the number of co-cache-preferring benchmarks reduces when the cache size is increased. This is not surprising as the co-cache is designed primarily for situations in which the working-set size is greater than the size of the L2. From this we can conclude that when most of the benchmarks stop benefiting from the co-cache design (as the cache size is increased), it is crucial to have a way to reconfigure the cache back to conventional design. This conclusion serves as the motivation for our aco-cache approach which is the needing to support the workloads whose working-set sizes are already fit in the conventional cache.

Reconfiguration threshold. In aco-cache, we use shadow tags to monitor the L2's miss-rate and when the miss-rate falls below a threshold we reconfigure it back to a

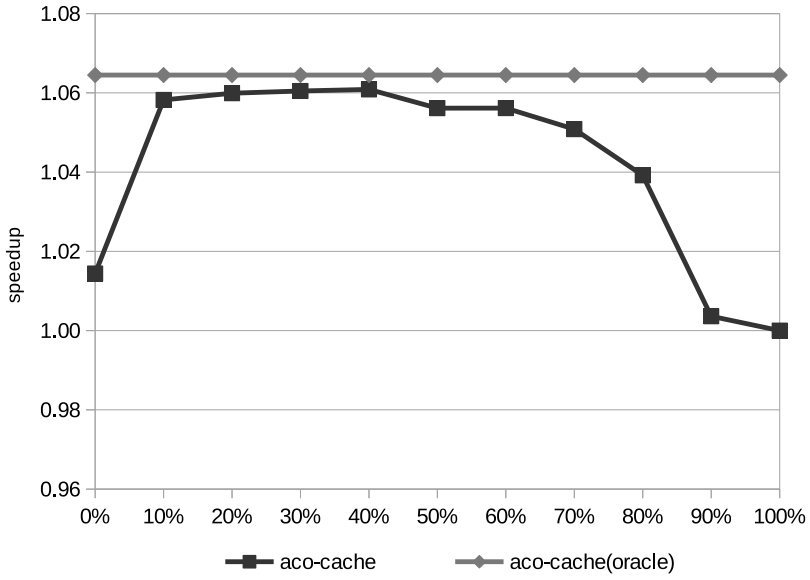

Figure 3.12: Sensitivity to reconfiguration threshold – 0% threshold means always use co-cache; 100% means always use conventional cache.

conventional cache. To obtain the correct threshold, we conduct a sensitivity study by varying the above threshold. In this experiment, we start by using the L2 as a co-cache and use 5 million instructions interval for the monitoring. We choose a *512 KB* cache size in this experiment because the number of co-cache-preferring benchmarks is about 50% (Figure 3.11). The performance results with different thresholds are shown in Figure 3.12. All results are normalized to *512 KB* conventional cache. As we can see, a threshold of 40% gives 6.1% improvement which is very close to the oracle reconfiguration's 6.5%. For this reason, we choose 40% as the default threshold.

3.5.3 Sensitivity to L2 Size

Figure 3.13 shows the performance as the size of L2 is varied. As we can see, for the smaller L2 sizes (128 KB, 256 KB, 512 KB), the co-cache performs better than the conventional cache. However, for the larger sizes (1 MB, 2 MB), the conventional cache outperforms the co-cache. As we mentioned in the previous section, this is because, as the size of L2 cache is increased, conventional cache starts becoming more and more efficient, and its miss-rate drops. On the other hand, the co-cache performance will be dependent on whether or not the particular benchmark displays criticality. Motivated by this observation, we show the performance results of aco-cache that could be configured into a conventional L2 or a co-cache depending on benchmark's miss-rate. As

we can see, this performs consistently better than the other two, especially when the size is neither too small nor large.

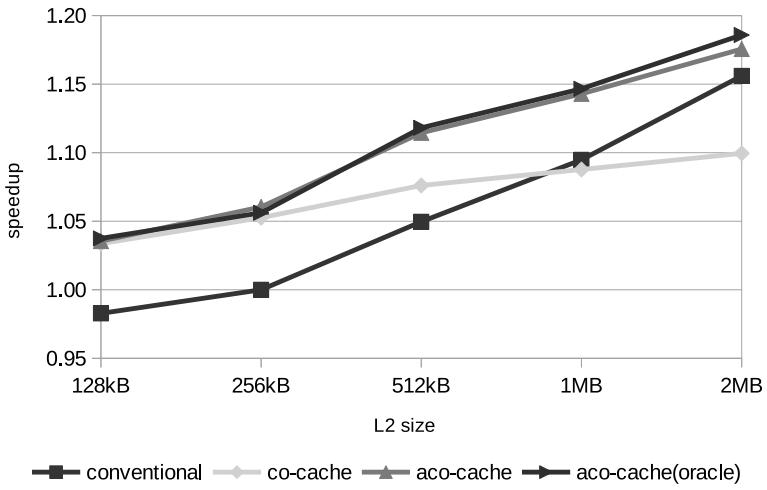

Figure 3.13: Sensitivity to cache size

3.5.4 Sensitivity to Lower Level Latency

In this experiment, we want to examine our technique's sensitivity to lower level latency. Therefore, instead of a 3-level cache hierarchy, we *remove the L3 cache* and vary the memory latency. In our 3-level cache hierarchy, the average latency of *L3 + memory* (i.e.L2 miss latency) is about 100 cycles. Therefore, we vary the latency from 30 cycles through 240 cycles.

In this experiment, all results are normalized to the baseline (256 KB conventional cache). In addition to overall performance results, we also show the improvement of co-cache under different L2 miss-rate groups that are shown in Table 3.4. As we can see from Figure 3.14, the improvement of group A benchmarks (high miss-rate) increase with increase in latency. Since these benchmarks have high miss-rate, a co-cache hit becomes all the more important with increasing latency. However, for benchmarks from group C (low miss-rate), co-cache would lose more performance because most of accesses are already hits in the baseline configuration (as we discussed in §3.5.2).

Overall, the improvement of the co-cache first increases from 5.2% to 5.9% and then decreases to 1.1% when the latency is varied from 30 cycles to 240 cycles. Since the aco-cache enables the L2 to choose between the co-cache and the conventional

designs based on L2 miss-rate, it provides a relatively consistent improvement (from 5.5% to 7.4%). This shows that aco-cache can continue to work even if the access latency to lower level memory is high such as a no-L3 system.

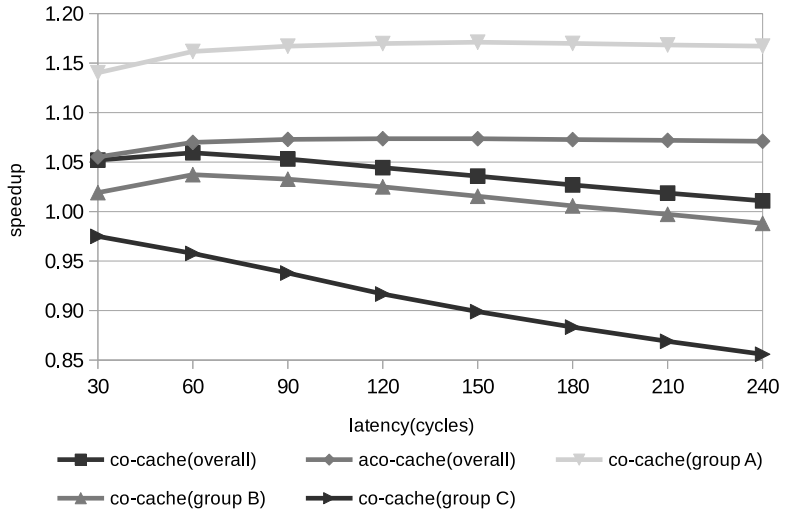

Figure 3.14: Performance speedups with different latencies – Group A, B, C are defined in Table 3.4.

3.5.5 Sensitivity to Bandwidth

The major side effect of our technique is the increased L3 accesses – recall that even for an L2 co-cache hit, we need to access the L3 to fetch the non-critical words. Consequently, the traffic between on-core caches and L3 (we call it *toL3 bus*) could become a potential bottleneck to performance. To quantify this, we conduct an experiment to examine the effect of varying the toL3 bus bandwidth. Figure 3.15 shows the performance results as the bandwidth is varied from 4 GB/s through 24 GB/s; all results are normalized to the performance of baseline (256 KB L2) for the corresponding bandwidth. As we can see, the improvement of co-cache decreases gracefully with decreasing bandwidth. In very low bandwidth setting (4GB/s), we can even see a small slowdown of 3.0% in co-cache. However, even for this restrictive setting, aco-cache does relatively well with a 2.3% improvement. This is because aco-cache would only use the co-cache mode in those benchmarks whose miss-rate is high; in such benchmarks, most of the L2 accesses would miss in the L2, that would cause the requests to go to the L3 anyway. Therefore, we believe aco-cache can provide performance improvement even when the bus bandwidth is limited.

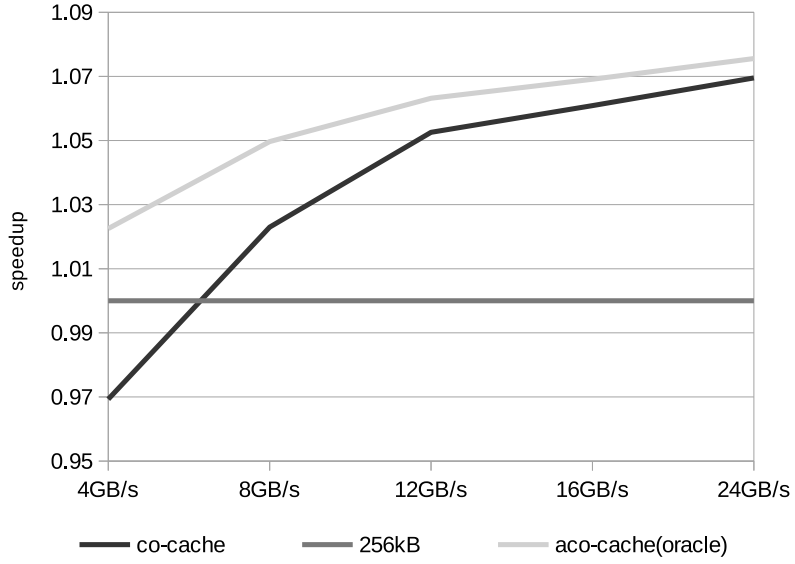

Figure 3.15: Speedup versus Bandwidth

3.5.6 Prefetcher Effects

Hardware data prefetching is a technique to predict miss stream and fetch blocks before they are accessed. If the prefetcher brings data into the L3, then our technique is orthogonal to it. However, if the prefetcher brings data into the L2, this cannot directly apply to our L2 (as a co-cache) as co-cache only holds partial blocks (critical words). As an alternative, we can have a prefetcher that brings data into the L1 instead, with a potential downside that the prefetcher may induce L1 cache pollution. In this experiment, we apply a stride prefetcher[5] to the L2 in our baseline system to see if our alternative (co-cache + L1 prefetcher) can still improve over the baseline. We experiment on the following configurations: baseline (prefetcher in L2), L1PF (prefetcher in L1), and aco-cache[6]. With the prefetcher included, some of the benchmarks no longer experience significant capacity misses. In other words, for such benchmarks there is no problem to solve. Nonetheless, despite using a prefetcher, as shown in figure 3.16, a significant number of benchmarks (19 out of the 33) continue to suffer from capacity misses. We are able to get 6.3% (up to 36.7%) performance improvement if we consider these 19 benchmarks (3.4% improvement if we consider all 33 benchmarks).

[5]In this stride prefetcher, the prefetching distance is 8 and it will prefetch across the OS page boundaries.

[6]In this experiment, the aco-cache is an oracle design which chooses the best-performing alternative among the co-cache/L1 prefetcher and the baseline.

From this result, we believe that the co-cache/aco-cache can continue to boost performance even under the presence of a prefetcher.

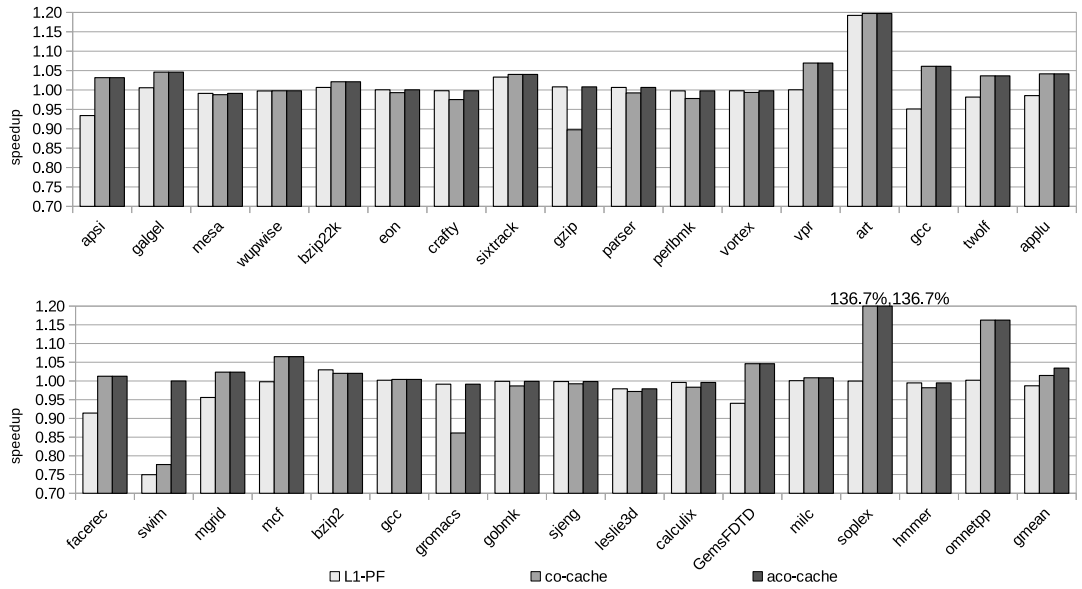

Figure 3.16: Performance speedups with a stride prefetcher

3.5.7 Energy Impact

In this section, we discuss the energy impact of our technique. We use CACTI [12] to model co-cache's power consumption. In general, the co-cache design potentially has both static and dynamic energy overheads compared to conventional cache. In our experiments, we show energy impact of co-cache in all on-chip caches (L1s + L2s + L3). In CACTI, we use *itrs-hp* cell and 32 nm technology. The access mode is set to *fast* in L1 and *normal* for L2 and L3.

Static energy. The aco-cache induces additional space overhead of 44.75KB as described in §3.3.4. On a 4-core system, this amounts to 179KB overhead, which is a mere 1% space overhead if we consider the overall cache hierarchy (L1s+L2s+L3). Therefore, the increase in static power is about 1%. However, since the co-cache (aco-cache) improves the performance by 5.3% (6.1%), there is an overall reduction in static energy of 4.2% (5.3%) with the co-cache (aco-cache).

If we only consider L1 + L2 static energy, as we can see from Figure 3.17, the co-cache in group A (high miss-rate benchmarks as shown in Table 3.4) only increases the static energy by 3.9%. This is because co-cache provides performance improvement for these benchmarks. For aco-cache, although the shadow tags require additional

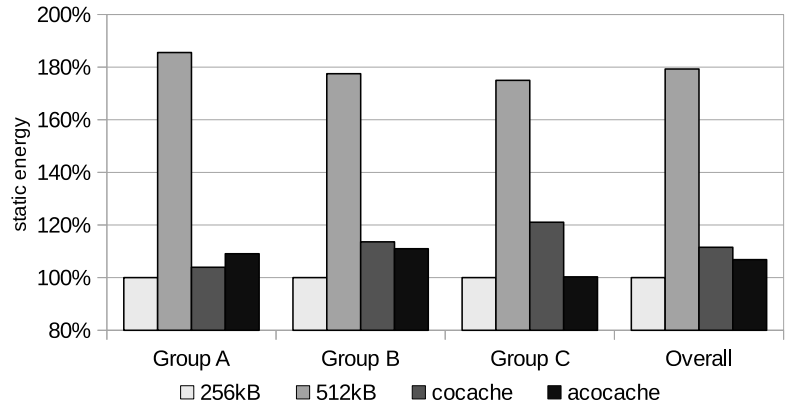

Figure 3.17: Static energy comparison (without L3, the lower the better)

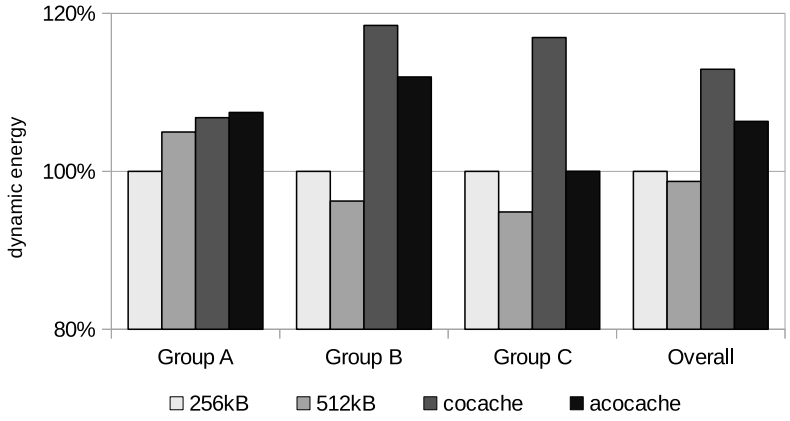

Figure 3.18: Dynamic energy comparison (the lower the better)

overhead (8 KB) and hence increase the energy cost to 9.1% in group A, the reconfiguration capability enables low miss-rate benchmarks to run in conventional mode, which leads to almost no increase in static energy in group C.

Dynamic energy. In our proposed designs, the co-cache requires accessing the L3 even for a co-cache hit and aco-cache requires additional accesses to shadow tags for obtaining miss-rate (as in§3.3.3). Therefore, we examine the dynamic energy (L1+L2+L3) overheads in our techniques. As we can see from Figure 3.18, because most of the accesses in group A (high miss-rate benchmarks) miss in the L2 and go to the L3, the co-cache (aco-cache) only causes a moderate increase of 6.8% (7.5%) in dynamic energy. In group B and group C, the dynamic energy overhead due to co-cache increases to 18.5% and 16.9% respectively. However, the aco-cache improves the energy overhead (compared to co-cache) in group C by reconfiguring back to the conventional mode and shows almost no energy overhead. It is worth noting that the

monitoring will be turned off [37] after the decision to reconfigure it as a conventional cache. Overall, the dynamic energy overhead of co-cache (aco-cache) is 12.9% (6.3%).

Total energy. If we consider the total energy impact (static + dynamic), there is a 3.8% (5.0%) improvement with the co-cache (aco-cache), as the static energy dominates dynamic energy (the average static/dynamic ratio is 3.0 to 1).

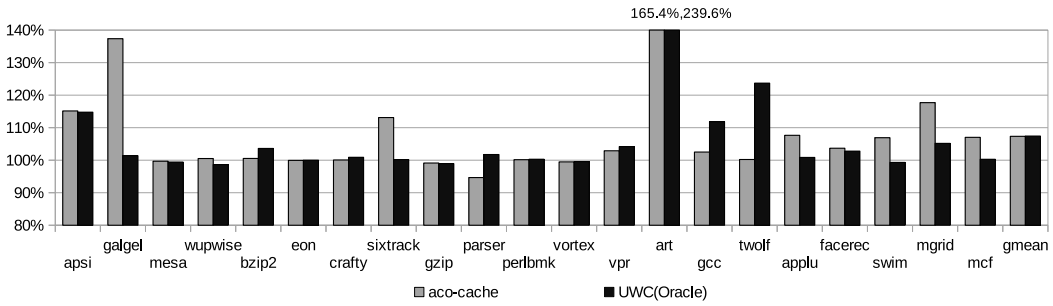

Figure 3.19: Performance comparison with oracle useful words filtering

3.5.8 Comparison with Oracle Useful Words Technique

To quantify our advantage over the useful words [31, 30], we implemented an enhanced version of WOC and called it *useful words cache (UWC)*. The UWC we implemented is a cache that can *dynamically allocate a variable sized block from 8 to 64 bytes* and *reuse the saved space for additional blocks*; for example, a cache set in UWC can contain eight 64-byte blocks, 64 8-byte blocks, or any combinations that equal 512 bytes. Furthermore, to predict useful words in the UWC, we use an *oracle predictor* (which always predicts correctly). Thus, the performance results we obtain from UWC is expected *to be better* than the actual performance of Qureshi' work[31]. Note that, for our aco-cache results, we only use the simple critical words predictor described in §3.3.1 (we *do not* use oracle predictor in our results).

In this experiment, we compare aco-cache and UWC for an L2 size of 512 KB. We ran this with benchmarks from SPEC2000 suite. As we can see from Figure 3.19, even under this configuration that is a disadvantage to us, aco-cache (7.35%) is still as good as an oracle UWC (7.43%) on average. Despite using much simpler hardware (fixed block size design, simpler predictor), we were able to match the performance of UWC. Thus, this experiment reinforces the conclusion of our earlier limit study – that our idea of critical words filtering is inherently more powerful than the idea of useful words filtering.

Groups	Workloads
G1 - G4	27-23-10-31, 1-7-9-24, 9-24-33-13, 23-1-20-17
G5 - G8	34-14-20-30, 24-5-5-30, 30-22-20-32, 5-5-5-30
G9 - G12	24-9-8-3, 5-20-9-22, 13-22-7-13, 17-18-4-5
G13 - G16	18-6-27-22, 7-34-19-16, 12-7-10-32, 8-1-22-26
G17 - G20	18-9-1-4, 32-29-20-7, 7-32-8-10, 11-5-32-3
G21 - G24	9-14-25-10, 13-4-32-29, 23-4-1-1, 22-28-5-15
G25 - G28	7-6-12-11, 11-25-7-15, 30-2-33-4, 9-18-9-33
G29 - G32	27-7-33-1, 11-31-11-5, 4-29-27-24, 28-30-3-2

Table 3.5: Workload groupings

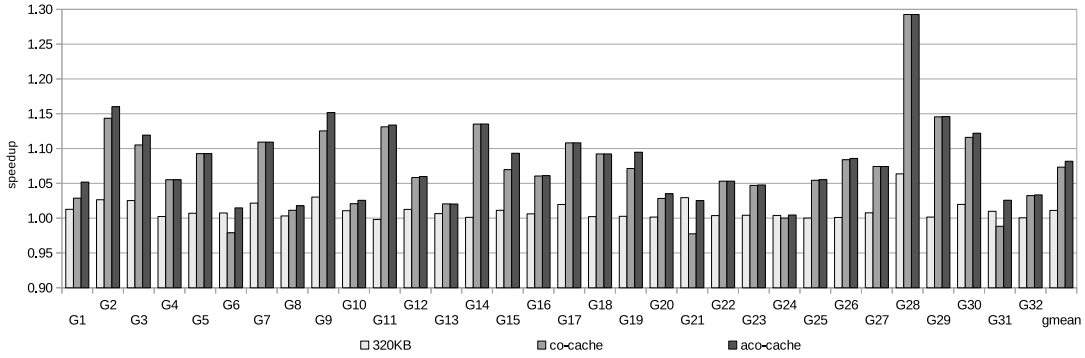

Figure 3.20: Performance improvement in multiprogrammed workloads – Please refer to Table 3.3 and Table 3.5 for workloads' information.

3.5.9 Multi-core Workloads

In this section, we study how our co-cache design performs under multiprogrammed workloads. We randomly generate 32 workload groups as shown in Table 3.5, with each group consisting of 4 workloads (the numbers identify the benchmark programs as shown in Table 3.3). The performance results are shown in Figure 3.20. We use the sum of each workload's IPC[7] and normalize them to the 256 KB baseline. As we can see, the co-cache is up to 29.3% faster than the baseline and on average 7.3% (8.1% for aco-cache) faster than the baseline, whereas the 320 KB size cache (that resembles aco-cache in the area) only provides 1.1% improvement. Since it is comparable to the improvements we obtained with our uniprocessor workloads, this shows that the co-cache continues to work well under multi-programmed workloads. One interesting

[7]Instead, if we simply use the gmean to average four workloads' improvements, the co-cache (aco-cache) would have an 8.7% (9.7%) improvement compared to the baseline.

observation here is that aco-cache performs consistently better than the co-cache with multiprogrammed workloads for most cases. We believe this is because aco-cache benefits here from two reasons. First, like before, aco-cache avoids performance loss for conventional-preferring benchmarks (as in §3.5.2). Second, since conventional caches cause lesser traffic on the L3 bus, it frees up bandwidth which can be potentially exploited by co-cache-preferring benchmarks (as in §3.5.5). For example, G21 has two conventional-preferring workloads (gobmk and gzip) and two co-cache-preferring workload (gcc2k and parser); by freeing up bandwidth due to conventional-preferring workloads, aco-cache provides better bus throughput to gcc2k and parser.

3.6 Related Work

Our idea is inspired by the classic *critical word first* [18] technique for reducing miss penalty, in which, instead of providing the full cache block, the word causing the miss (the critical word) is provided first. Gieske [42] adapted this idea to reduce the miss-penalty of lower level caches by proposing a cache organization that is able to service the critical words faster. In contrast to the above works which seek to reduce the miss-penalty, we seek to increase the cache capacity of lower level caches.

Useful words fetching. A number of works exploit the fact that not all words belonging to a cache line end up being used; only a subset of the words – the *useful words* – end up being used. Several prior works [32, 30, 29, 33] focus on predicting these useful words and fetching only these. Kumar et al. [29] first observed that not all words belonging to a cache block end up being used, i.e spatial locality is not always high in all cache blocks. They exploit this observation to improve the miss-rate of a *sectored cache* [43] – which is a design to optimize bandwidth by fetching only sub-blocks. However, the performance of this technique is still worse than the conventional cache because of the low useful words predictability. To improve that, Chen et al. [32] and Pujara et al. [33] proposed more accurate PC-based predictors to provide better predictability. While the above techniques focus on primarily reducing bandwidth, the recently proposed *amoeba cache* [30] also focus on the performance. They do this by allocating a suitable number of entries for each cache block depending on the number of consecutive useful words in that block. We consider these techniques to be orthogonal to our work. While our co-cache works by retaining the critical words when a block is replaced from the higher level, the above techniques work by only fetching

useful words. In other words, it would be possible to integrate our method with any of the above, which we leave for future work.

Useful words filtering. Qureshi et al. [31] proposed *Line Distillation* to achieve better performance. They use a separate *word-organized cache* (WOC) as a victim cache that caches only the useful words. When a block is replaced from the higher level, they filter the useful words and put it in the WOC. In doing so, the cache is utilized more effectively which in turn leads to better performance. The downside to this approach is the complexity of supporting WOC. WOC is a highly associative cache – compared to a cache with 64-byte blocks, a WOC with 8-byte words increases associativity eight times. Our approach, like Line Distillation, is a filtering approach; however, we exploit critical words rather than useful words.

Even if all words in a cache block end up being used (i.e., spatial locality is high), we can still benefit as long as the critical words are accessed before the non-critical words, as we illustrated in the limit study in §3.2.3. We found that for most benchmarks, the number of critical words is only 2 – which is why caching only these is sufficient. In other words, we do not need the complexity of having to deal with blocks of various degrees of spatial locality. Thus, unlike WOC used in Line Distillation, our approach does not increase associativity.

3.7 Conclusion

In this chapter, we proposed a cache design called *critical-words-only cache* for increasing cache capacity of an L2 cache. Our design is based on the observation that for every L2 cache block, a subset of words (the critical words) are accessed sooner than the others. In contrast to a conventional L2, a co-cache only caches the critical words for each cache block. Our experimental results provide evidence to support the hypothesis that a co-cache is able to utilize the cache space more effectively, especially in situations in which the cache size is significantly less than the working-set size. However, in situations in which the cache size is larger than the working-set size, a conventional cache could perform better. For this reason, we also proposed adaptive co-cache (aco-cache) that can dynamically choose to behave like a co-cache or a conventional cache. In our experiments, a 256 KB L2 organized as an aco-cache performed as well as a 512 KB conventional L2 cache on average.

Although the co-cache can increase effective cache capacity in a private L2 cache, many modern applications have working-set sizes [6, 7] on the order of tens of megabyte

to hundreds of megabytes. Clearly, a conventional last-level cache is not sufficient for these applications. Therefore, in the next chapter, we will study how to support a larger last-level cache based on a recent proposed die-stacked DRAM cache.

Part III

DRAM Cache Optimization

Chapter 4

Aggressive Tag Caching for DRAM Cache

4.1 Introduction

The size of the conventional last-level cache may not be sufficient for many modern applications' working-set-size [6, 7]. Recently, 3D-stacking technology has enabled the option of embedding DRAM chip(s) onto the processor die. Based on that, DRAM cache [8, 3] have proposed to use exploited this stacked DRAM as a gigantic DRAM cache. However, because of the larger size[1], the size of the tags associated with it can also be quite large (on the order of tens of megabytes).

The large size of the tags has created a classic space/time trade-off issue. On the one hand, we would like the latency of a tag access to be small as it would contribute to both hit latency and miss latency. Accordingly, we would like to store these tags in a faster media such as SRAM (*tags-in-SRAM*). However, with hundreds of megabytes of die-stacked DRAM cache, the space overhead of the tags would be huge. For example, it would cost around 12 MB of SRAM space to store all the tags of a 256MB DRAM cache (if we used conventional 64B blocks). Clearly this is too large, considering that some of the current chip multiprocessors have an L3 that is smaller [5].

To solve the above problem, Loh and Hill [3] proposed an approach for storing the tags within the DRAM itself (*tags-in-DRAM*). To make this approach practical, they proposed a scheme for embedding the tag and data in the same row buffer, which would enable both tag and data to be accessed in a single *compound access*. Compared to a

[1]The size of the DRAM cache proposed in prior works are be from hundreds of megabytes [3] to few gigabytes [11].

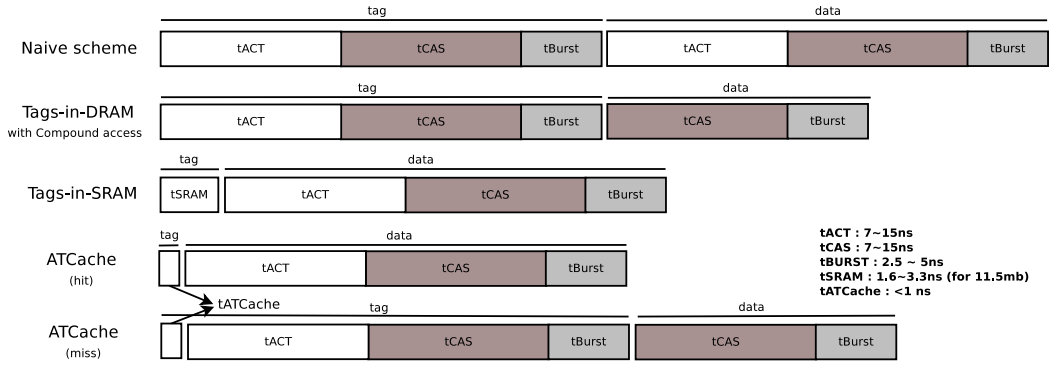

Figure 4.1: Access latency of different types of DRAM cache.

naive scheme, performing a compound access optimizes the hit latency by obviating the need to reopen a row to access data as shown in Figure 4.1 (saving on *tACT*, the time to activate a row). However, it does not optimize the miss latency, since only the tag is accessed on a miss. To reduce the miss penalty, Loh and Hill also proposed a technique for tracking the contents of the DRAM cache in a structure called the *MissMap*. This design enables them to avoid a DRAM tag access for misses, with only a few megabytes of SRAM overhead. Subsequently other works [4, 9] proposed miss predictors to achieve the same effect as a MissMap, but with significantly lesser SRAM overhead (in the order of a few kilobytes).

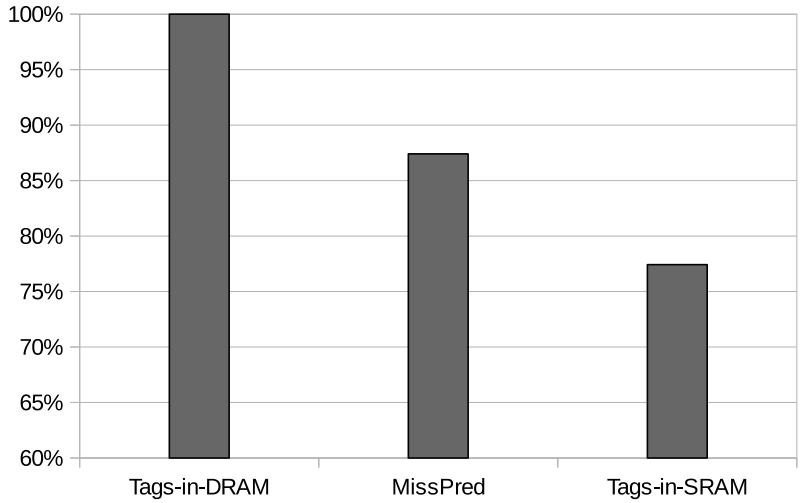

Figure 4.2: Average latency of the DRAM cache (including main memory access latency). The architectural parameters and workloads are shown in Table 4.5.

The impracticality of tags-in-SRAM has led to the above tags-in-DRAM proposals. However, how does the performance of recently proposed tags-in-DRAM techniques compare with tags-in-SRAM? We conduct a study to measure the average DRAM ac-

cess latency for different configurations of DRAM cache, as shown in Figure 4.2. Here, tags-in-SRAM refers to a scheme in which tags of all blocks in the DRAM are stored in the SRAM; tags-in-DRAM refers to a scheme that uses Loh and Hill's compound access; MissPred refers to the above, augmented with an oracle miss predictor (100% accuracy and zero latency). For this study, we use an SRAM access latency of 6 cycles which we modeled using CACTI (32nm, itrs-hp cell); further, we use 7ns for tACT and tCAS and 2.5ns for tBURST for stacked DRAM (other architectural parameters used in this study are shown in Table 4.5). All results are normalized to the access latency of tags-in-DRAM. As we can see, using the tags-in-SRAM configuration results in a 23.7% reduction in DRAM cache access latency. Furthermore, this latency is 10.2% lesser than MissPred (which is an oracle miss predictor with zero access latency).

Given that there is a significant gap in performance between tags-in-SRAM and tags-in-DRAM, we want to benefit from the tags-in-SRAM approach, but without incurring the cost of a high SRAM overhead. Our key idea is to maintain the tags in the DRAM cache, *but also cache a small amount of tags in SRAM in a dedicated cache*. We call this *Aggressive Tag Cache (ATCache)*, as we will later show that we can achieve excellent performance with a small tag cache. In addition to this, having a small cache is beneficial as this means that ATCache can be accessed faster than a larger tags-in-SRAM design.

Like a conventional cache, ATCache exploits temporal locality by only caching the tags of recently accessed DRAM cache sets. In a similar vein, it exploits spatial locality by prefetching the tags of adjacent DRAM cache sets. One potential problem is to know how many sets' tags to prefetch, as prefetching too much can lead to cache pollution and also increased miss penalty. To deal with this, we propose a *cost-effective prefetching* in which we prefetch tags of the adjacent sets only if there is evidence of spatial locality amongst the sets.

We evaluate our approach on the gem5 cycle-accurate simulator [41]. On memory-intensive single-threaded SPEC 2006 benchmarks, our ATCache can achieve 10.3% performance improvement on average compared to a tags-in-DRAM (with compound access) baseline. This compares favorably with the improvement of 6.8% provided by adding a high-accuracy miss predictor (MAP-I) to tags-in-DRAM. Our ATCache can also be integrated with miss predictors. Indeed, on multiprogrammed workloads, our ATCache is 9.3% faster than the tags-in-DRAM baseline, but 10.9% faster when we integrate a miss predictor (MAP-I) to ATCache. Finally, our results are almost as good as a full tags-in-SRAM cache, which provides a performance improvement of 11.9%.

Our ATCache only uses 47.375KB space (including the overheads) and around 50KB if we include the MAP-I as part of our design. Therefore, the overall SRAM space consumed is only around 0.5% of a tags-in-SRAM design.

4.2 Motivation

In this work, we motivate our idea by studying on the tag size and its access latency. In this section, we first show how we measure the tag sizes of different DRAM cache configurations and also their latencies. Later, we show our perspective on the study.

Tag size. A *tag* for a block refers to several SRAM bits of storage to accommodate not only the tag for that cache block but also status bits (dirty/valid and cache coherence states) and state associated with the replacement policy. In our system model, the DRAM cache is located a level below the cache coherent shared cache. The minimum requirement of the status register is 2 bits (valid and dirty bits). The number of tag bits depends on processor's addressing capability and the associativity of cache. For a processor with 40-bit address space and 64-byte cache line, a 256MB/16-way cache requires 16 bits for the tag. Let us assume the state associated with the replacement policy requires 5 bits. The overall space requirement for each cache block then is 22 bits, which amounts to around 11.5 MB in total.

Latency. We model the latency of tag access using CACTI 6.5 [44]. Specifically, we use 32nm technology and two types of SRAM cell (itrs-hp and itrs-lstp, which represent high-performance configuration and low standby power configuration respectively). We report tag latencies for a 3GHz processor cycle. The results are shown in Table 4.1[2].

Our perspective. As we can see from Table 4.1, the latency result shows that even with a low standby power cell, an 11.5MB SRAM tag access (10 processor cycles) can still be faster than minimum tBURST latency (which is about four memory cycles which are about 5ns or 15 processor cycles) in DDR3-1600 in DRAM. Substituting the values for tag latencies computed above, it is clear (as shown in Figure 4.1) that tags-in-SRAM can provide both better hit (tag + data) and miss (tag) latency than tags-in-DRAM with compound access. In this work, we are interested in how to benefit from the low latency that SRAM provided, but without incurring the cost of a high SRAM overhead.

[2]In this work, we will use these modeled parameters in Table 4.1 for the tags-in-SRAM approach.

Cache size	128MB	256MB	512MB	1024MB
Tag size (per block)	17 bits	16 bits	15 bits	14 bits
Status size (per block)	7 bits			
Total tag size	6MB	11.5MB	22MB	42MB
Latency/hp (cycles)	5	6	7	8
Latency/lstp (cycles)	9	10	12	13

Table 4.1: Tag sizes/latencies for different cache sizes.

4.3 Methodology and Design

As our study in Figure 4.2 shows, a tags-in-SRAM design provides an opportunity to improve both hit and miss latency. The SRAM overhead, however, is a major impediment that limits its practicality. In this chapter, we propose a hybrid method in which we maintain full tags in DRAM like Loh and Hill's work [3] but also cache a small number of tags in our proposed cache structure called *ATCache*. To put it simply, similar to a conventional cache which *caches data from memory*, our ATCache *caches the tags of the DRAM cache*.

4.3.1 Terminology

Before describing our approach, we first define several terminologies that can help explain our design.

SetTag: A SetTag refers to the set of tags of the blocks which belong to the same cache set of a DRAM cache. For example, if the DRAM cache is 16-way associativity, then SetTag refers to all the 16 tags in each set. It is worth noting that all 16 tags are necessary to check for a hit or miss in DRAM cache, which is why they are cached together in the ATCache. In other words, a SetTag of an ATCache is analogous to a word of a conventional cache.

SetID: A SetID is an identifier for ATCache to identify if the SetTag for the correct set exists in the ATCache. In other words, a SetID of an ATCache is analogous to a tag of a conventional cache.

Caching ratio: The Caching ratio of an AT cache is the ratio of the size of ATCache to the total size of the tags required by the DRAM cache. For example, a 256MB/16-way DRAM cache requires 11.5 MB for tags. A caching ratio of 256 corresponds to an ATCache of size 46KB.

Prefetching granularity (PG): In a conventional cache, whenever a word is accessed we also (pre)fetch additional adjacent words belonging to a block to exploit spatial locality. In a similar vein, the prefetching granularity (PG) refers to the number of adjacent SetTags that the ATCache will fetch on a miss. In other words, the PG of an ATCache is analogous to a block size of a conventional cache.

4.3.2 Locality of tag accesses

The idea of caching is founded on the principles of spatial and temporal locality, which a conventional cache exploits. In this section, we want to examine if spatial and temporal localities exist for tag data accesses also. To this end, we conduct a quick study in which we use a 46KB ATCache (1/256 of total tag size) to store recently accessed tags. Because tags of the same cache set would be accessed together, we store these tags in units of a DRAM cache set (and call it SetTag). In addition to this, to exploit spatial locality we fetch the SetTag of adjacent sets on an ATCache miss; we use the term *prefetching granularity* (PG) to refer to the number of adjacent cache sets that the ATCache will fetch on a miss.

Figure 4.3 shows the average miss ratio of ATCache for different PGs across all single-thread benchmarks that we studied (§ 4.4) in SPEC 2006. From the figure, the miss ratio of PG/2 is around 63.4%. This indicates that more than 30% of the tag accesses can be satisfied (hit) by our ATCache for a PG of 2, which is significantly better than a uniformly distributed hit ratio ($1/256 \simeq 0.4\%$). Furthermore, as the PG is increased from 2 through 64, the miss ratio decreases from 63.4% to 20.5%. These all indicate the existence of locality amongst tag accesses.

4.3.3 Cost-effective Prefetching

In our prior study, we found that the prefetching granularity (PG) is critical to the performance of ATCache. Prefetching nearby SetTags to the ATCache, generally speaking, is beneficial to hit ratio; however, a large PG can potentially pollute the ATCache, which could result in a hit ratio drop, in addition to increasing miss-penalty. Besides,

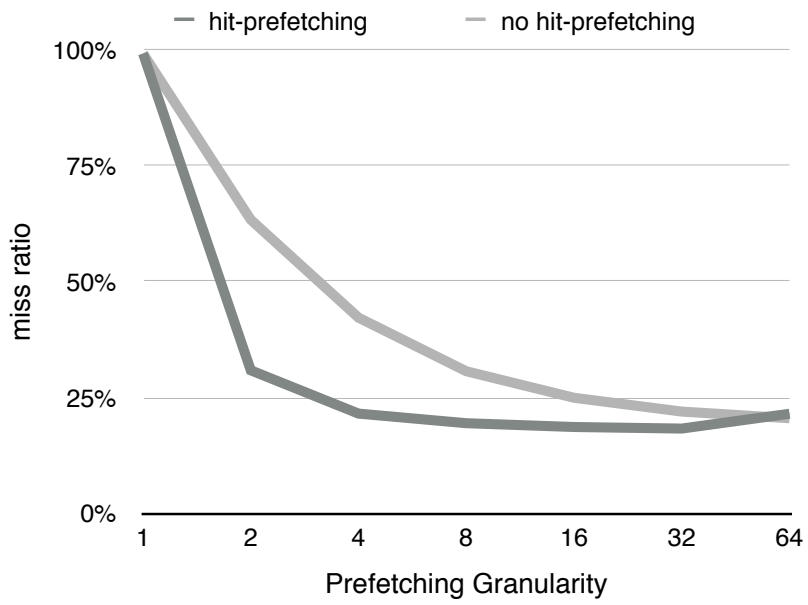

Figure 4.3: Miss ratio with various PG.

prefetching a SetTag that is not going to be eventually used, is wasteful in terms of energy. Therefore, it is also desirable to avoid such redundant prefetches. In this section, we propose two techniques for performing prefetching in a cost-effective fashion.

Hit-prefetching We provide a simple solution to achieve a balance between cache pollution and spatial locality. We start by fetching the SetTags for a relatively small number of cache sets (say PG/4). Corresponding to the SetTags of each fetched cache set, we maintain a flag that tracks whether or not the SetTags of the current cache set is accessed. Even if one of the prefetched SetTags is accessed, we prefetch the SetTags of the next contiguous four cache sets – in doing so, we achieve the effect of a larger PG. In case none of the prefetched SetTags are accessed, we do not prefetch additional SetTags – in doing so, we avoid paying the space and time costs of a larger PG when a larger PG is not beneficial. It is worth noting that the space overhead of this technique is small. It only requires *one additional bit* in ATCache's SetID. As shown in Figure 4.3, we conducted a simple experiment to estimate the benefit of hit prefetching; as shown in Figure 4.3, we find that with hit prefetching the miss ratio of PG/4 (21.5%) can match the miss ratio of PG/32 (22%) without hit prefetching. Since hit prefetching provides a significant benefit, we include it in our baseline system.

Dynamic PG tuning

With hit prefetching, we achieve the effect of a larger PG with a smaller PG. However, what exact PG value should we use? Intuitively, the PG should be a function

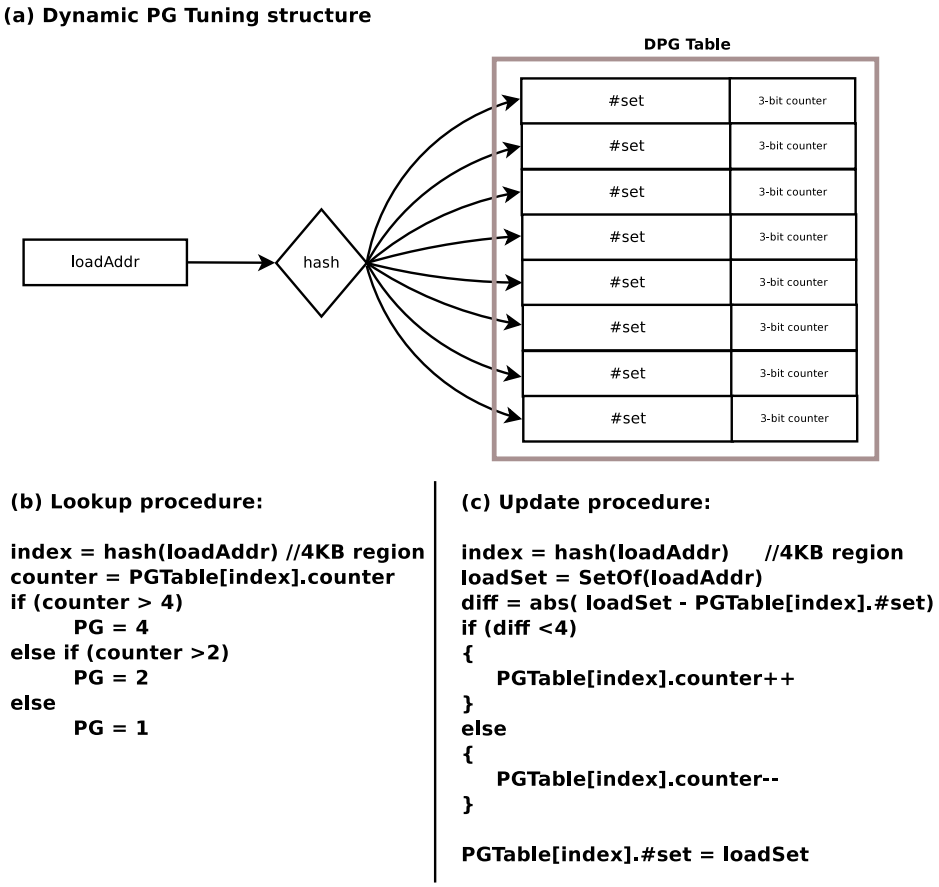

Figure 4.4: Hardware structure of DPGTable

of the spatial locality of the workload; furthermore, the spatial locality of a workload can vary: different memory regions can have different spatial localities. Therefore, we need to have a way to configure the PG dynamically, depending on both the current workload and the spatial locality of the current memory region. To achieve this, we divide the physical address space into fixed-size regions (4KB in our implementation). We use a small hardware structure (DPGTable) to track the spatial locality in different memory regions as shown in Figure 4.4(a). Each entry in DPGTable consists of (i) set number (#set), which stores the most recently accessed set in the memory region and (ii) a counter, which is a measure of the spatial locality of that region. Upon an ATCache miss, we perform a DPGTable lookup to determine the PG depending on the counter value as shown in Figure 4.4(b). At the same time, we also update the counter value as shown in Figure 4.4(c): if the accessed set is close to the last accessed set, we increment the counter; otherwise, we decrement the counter. In our experimental evaluation, we find that with dynamic PG tuning, we can remove about 70% additional tag accesses with negligible performance impact (less than 0.3%).

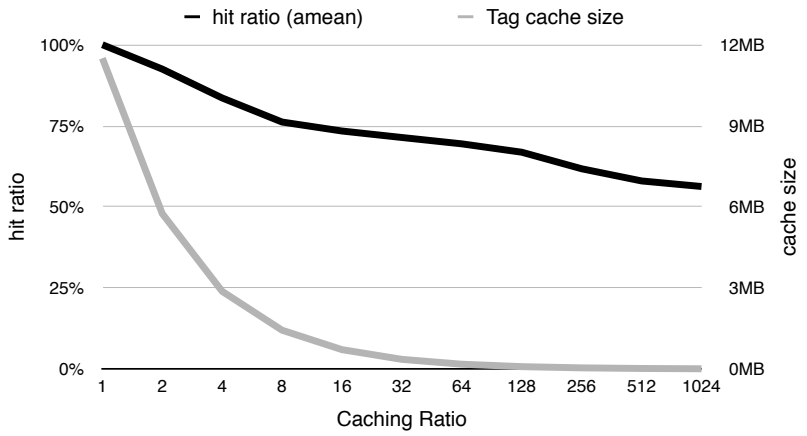

Figure 4.5: Hit ratio, size for different caching ratios.

4.3.4 Size, hit ratio and latency

Another important factor to consider in cache design is the interplay between size and the access latency. In most cases, a larger cache size can provide a better hit ratio, but would also mean a higher access latency. In this section, we want to study how the cache capacity affects the ATCache's hit ratio and its latency. Figure 4.5 shows the hit ratio for different caching ratios. For this experiment, we use PG/4 with hit prefetching turned on. As we can see, the miss ratio degrades gracefully as the caching ratio is increased. Even with an ATCache size of 11.2KB (caching ratio of 1024), the ATCache can still satisfy over 50% of tag accesses. On the other hand, an ATCache of 11.2KB will enable it to have a faster access latency of close to 1 cycle in comparison to about six cycles for a full tags-in-SRAM design (the latency values are computed using CACTI as discussed in the earlier section). The reduced tag access latency contributes to better DRAM cache performance as illustrated in Figure 4.1.

Virtualizing the ATCache In this work, we use a dedicated SRAM structure to store cache tags. Another ATCache design point is to virtualize the storage in the shared cache [45]. Compared to using a dedicated structure, virtualization gives us the opportunity to adjust the size of ATCache dynamically and utilize on-chip SRAM space more efficiently. However, if we virtualized the ATCache in the L2 (§4.5), the ATCache access latency will increase from dedicated structure's 2-cycle to L2's 20-cycle. This trade-off could be acceptable for the workload that requires larger space to provide the better hit rate. In this work, while most of the workloads that we used does not require a large ATCache, we focused on a dedicated ATCache design and left the virtualized design for the future study.

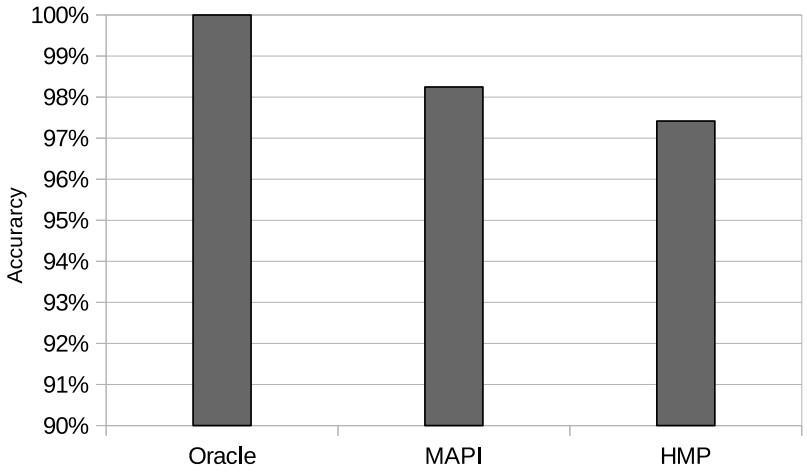

Figure 4.6: Accuracy of prior proposed predictors.

4.3.5 Integration with miss predictor

Prior works have proposed miss predictors [3, 4, 9] which help to improve the performance of a tags-in-DRAM design by reducing the miss penalty. More specifically, they help avoid a DRAM tag access for (what is predicted to be) a DRAM cache miss. Although the latency to access a tag in our design is not as high as the tags-in-DRAM design (since the tags are now stored in the ATCache, which is in the SRAM), we can still benefit from a miss predictor. In addition to this, because miss accesses can be handled by the high accuracy predictor, they can skip the ATCache. This means that the ATCache does not need to store the tags corresponding to misses anymore, which in turn increases the effective cache capacity of the ATCache. Therefore, we implement two predictors proposed in prior works – HMP [9] and MAP-I [4]. In our study, we found that both predictors can provide a very high prediction accuracy (Figure 4.6) with very small space overhead as prior works have observed. In our design we choose to integrate with MAP-I as it provides marginally better predictability.

4.3.6 Design of ATCache

The design of our ATCache follows the design principles of a conventional cache: the full tags in the DRAM array represent the “main memory” and tags in the ATCache represents the “cached data”. The procedure to access the ATCache is illustrated in Table 4.2 and the access logic is shown in Figure 4.7. As we can see, the ATCache requires an additional SetID checking (Step 2A) – which is similar to a tag check for a conventional cache. Now, this represents an additional check compared to a

Steps	Description
Step 1	Locate SetID and SetTag in ATCache by a subset of #cache_set (#sub_cache_sets).
Step 2A	Check SetID to determine if ATCache set contains SetTag.
Step 2B	Check SetTag to determine if DRAM cache contains requested data (DRAM cache hit/miss).
Step 3	If step 2A is hit, use step 2B's result to determine hit/miss in DRAM cache. Otherwise, issue compound access to DRAM cache.

Table 4.2: ATCache access procedure (refer to Figure 4.7).

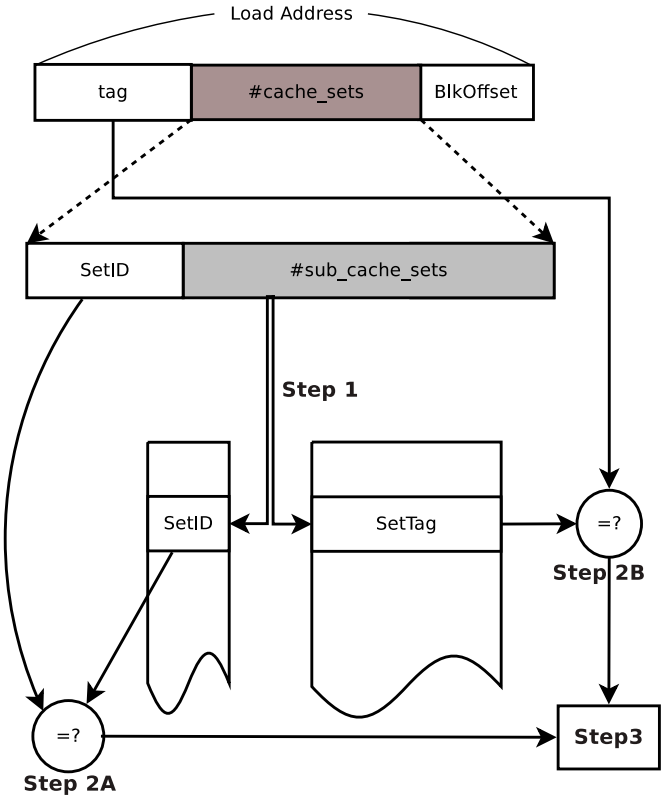

Figure 4.7: The access logic (also refer to Table 4.2).

#access	#set	MAP-I	ATCache	description
1	0x1	hit	miss	DRAM compound access and prefetch set 0,2,3 due to prefetching granularity (PG)
2	0x53	miss	X	predicted miss from MAP-I, skip the access
3	0x2	hit	hit	prefetch set 4-8 (hit prefetching)
4	0x3	hit	hit	access data in DRAM cache
5	0x10	hit	miss	DRAM compound access and prefetch set 0x11,0x12,0x13 (same as 1st access)

Table 4.3: Example showing 5 DRAM cache accesses.

conventional full tags-in-SRAM design. However, it is worth noting that this (step 2A) can be overlapped with step 2B (which is the tag check for DRAM cache). Therefore, the ATCache does not need additional access cycles for an ATCache hit in comparison to a full tags-in-SRAM design.

However, on an ATCache miss, the tags in DRAM have to be accessed and fetched back into the ATCache. So miss processing for ATCache is comparable to a tags-in-DRAM cache, with one small difference. Since step 2A is still required to identify if ATCache contains the correct SetTag, we incur one cycle penalty (step 2A is assumed to take one cycle) to the total access latency when there is an ATCache miss.

4.3.7 Putting it all together

We illustrate how ATCache works with a spatial predictor and a miss predictor (MAP-I), with the help of an example (Table 4.3). This example consists of 5 consecutive DRAM cache accesses, and each of them is accessing different cache sets (#sets). In the 1st access, the outcome of MAP-I is a hit which means DRAM cache might contain this data. Then, the system accesses the ATCache but it does not contain the corresponding SetTag. Therefore, similar to the tags-in-DRAM approach, an ATCache

issues a DRAM compound access for tag and data. After the compound access, the ATCache will also fetch the adjacent cache sets. Later, we can see these prefetched adjacent cache sets are accessed in the 3rd access. It is worth noting that a hit prefetching event (§4.3.3) also is generated in the 3rd access. In the 2nd access, we show a situation in which MAP-I is predicting the access as a miss. In this case, the system will skip the access of DRAM cache because of the prediction result and directly fetch data from main memory. In the 4th access, a prefetched set (set 3) is accessed but the sets 4-8 is already brought back by the 3rd access. Therefore, there is no additional access required for our design. Finally, the 5th access is a MAP-I hit and an ATCache miss. Similar to first access, the system will issue a compound access and fetch adjacent sets.

Cache Size	128MB	256MB	512MB	1024MB
Total tag size	6MB	11.5MB	22MB	42MB
ATCache space	24KB	46KB	88KB	168KB
ATCache overhead	704B	1.375KB	2.75KB	5.5KB
DPG Table	2.5KB	2.625KB	2.75KB	2.875KB
Latency (ns)	1.33	1.65	1.85	2.41

Table 4.4: Overhead for different cache sizes.

4.3.8 Area overhead

In this work, we use an ATCache with a caching ratio of 256. In other words, we use a 46KB cache for caching total tags amounting to 11.5MB (256MB/16-way DRAM cache). As shown in Figure 4.7, the additional overhead of an ATCache is a bunch of SetIDs. The size for each SetID actually depends on caching ratio and the associativity of ATCache. Each ATCache block with the caching ratio of 256 and associativity of 4 needs $8+2$ bits for storing SetID. In addition to SetID, to implement hit prefetching (§4.3.3, 1 bit per set), the overhead of SetID for each ATCache block slightly increases to 11 bits. In a 256MB/16-way cache, there are 256k cache sets. An ATCache with a caching ratio of 256 would cache only 1k ($256k/256$) sets. Therefore, the overhead of ATCache is around 1.375KB ($11*1024/8$ bytes). To support the dynamic PG tuning, DPGTable is required to track the spatial footprint in different memory regions. In this work, we use a DPGtable of 1024 entry. Each entry in DPGTable requires a set number (n-bit, depends on a total number of sets in DRAM cache) and a 3-bit counter. In total, the ATCache requires about 50KB SRAM space that is less than 0.5%

SRAM space compared to tags-in-SRAM's 11.5MB. Table 4.4 shows the overhead for different cache sizes.

4.4 Experimental Methodology

4.4.1 Baseline system

We use the gem5 cycle-accurate simulator [41] in which we consider the L3 to be the DRAM cache; accordingly, we implement the DRAM timing model for the L3. For scheduling DRAM cache's requests, we use the First-Come-First-Serve-based (FCFS-based) approach[3]. The system parameters that we used are shown in Figure 4.5[4]. We show single-threaded results for 11 benchmarks from SPEC 2006 which are considered to be memory-intensive in prior works [4, 9]. In addition to single-threaded benchmarks, we also use the same 11 benchmarks to generate 25 multiprogrammed workloads (4-core, as shown in Table 4.6) and evaluate their performance.

4.4.2 DRAM cache organizations

In this work, we evaluate the following DRAM cache designs:

Baseline (Tags-in-DRAM): The DRAM cache design we used as baseline follows Loh and Hill's work [3]. The SetTag and their data are stored in the same row. With compound scheduling, a delay of opening a row ($\simeq$ tRCD) can be saved compared to storing them in separate rows. Our baseline system uses a 4KB row buffer that is close to a realistic DRAM organization (1KB per device, four devices per rank). In our system, a 4KB row buffer can store four 15-way cache sets (4 x 15 x 64 bytes) and 4 SetTags (4 x 45 bytes). We use an open page policy to manage DRAM banks because we found open page policy gives better performance for our baseline.

NoDRAM (No DRAM cache): In this setting, we remove the DRAM cache from our cache hierarchy. The purpose of this setting is to examine if our system has any performance benefit from the DRAM cache in the first place.

MAP-I: As mentioned in §4.3.5, we use a MAP-I predictor for predicting misses. We implement An MAP-I predictor with 256 memory access counter (MAC) entries as suggested in prior work [4].

[3] It is worth noting that we will study DRAM cache's scheduling challenges in the next chapter.

[4] In the configuration, we use a relatively faster L2 cache access latency (4MB for 9 cycles) compared to the previous chapter (16MB for 40 cycles). This is because we try to be consistent with the optimal tag latency that we assumed in tags-in-SRAM design (12MB for 6 cycles).

Processor	3GHz, 4-core, 4-issue OoO, 64 ROB
L1 I/D caches	each 32KB/2way, LRU, 2-cycle, private
L2 cache	4MB/8way, 9-cycle, LRU, shared
Stacked DRAM	256MB, 2-bit SRRIP [20] tRCD-tCAS-tRAS 7-7-25 (ns) tBURST: 2.5ns 16 banks per rank, 1 rank per channel, 4 channel 4KB row buffer, open-page, FCFS
Off-chip DRAM	800MHz (DDR3-1600), x64 interface tRCD-tCAS-tRAS 13.5-13.5-40.5 (ns) tBURST: 5ns 8 banks per rank, 2 ranks per channel, 1 channel 8KB row buffer, open-page
On-chip bus	3GHz, 256-bit width
System Bus	1.5GHz, 64-bit width
Miss Predictor	MAP-I [4], 256 entries, 1-cycle latency
ATCache	Caching Ratio:256 (47.375KB incl. overhead) 4-way, 2-cycle latency, hit prefetching and DPGTable of 1024 entries (§4.3.3)

Table 4.5: System parameters

1-2	soplex-astar-lbm-mcf	lbm-omnetpp-leslie3d-bwaves
3-4	milc-leslie3d-leslie3d-gcc	milc-libquantum-bwaves-gcc
5-6	libquantum-lbm-soplex-libquantum	libquantum-GemsFDTD-soplex-milc
7-8	gcc-milc-libquantum-astar	milc-soplex-bwaves-libquantum
9-10	leslie3d-omnetpp-leslie3d-mcf	lbm-astar-leslie3d-libquantum
11-12	leslie3d-leslie3d-libquantum-milc	mcf-gcc-milc-astar
13-14	omnetpp-libquantum-milc-soplex	gcc-libquantum-libquantum-soplex
15-16	soplex-GemsFDTD-omnetpp-milc	milc-soplex-leslie3d-libquantum
17-18	lbm-libquantum-omnetpp-bwaves	gcc-milc-leslie3d-milc
19-20	omnetpp-omnetpp-libquantum-leslie3d	soplex-mcf-gcc-libquantum
21-22	astar-omnetpp-astar-gcc	mcf-soplex-astar-leslie3d
23-24	bwaves-lbm-libquantum-leslie3d	astar-leslie3d-lbm-mcf
25	bwaves-soplex-bwaves-GemsFDTD	

Table 4.6: Workload groupings

SRAM (Tags-in-SRAM) We use a tags-in-SRAM design which requires 11.5MB (§4.2) SRAM space to store all tags for 256MB DRAM cache. The access latency of the SRAM array is 6 cycles (high performance cell) as we modeled in §4.2. It is worth noting that the high SRAM overhead makes this design impractical.

ATCache: In our proposed design, we use an ATCache with *caching ratio of 256, prefetching granularity (PG) of 4, and associativity of 4*. This means we use $11.5MB/256 =$ 46 KB SRAM tag with 1.375KB overhead. We assume a single cycle latency to identify if the ATCache contains a correct cache set (step 2A in §4.3.6). A single cycle latency is reasonable considering that the SetID array is only 1.375KB in our design. Since the SRAM array of the ATCache is only 46 KB and step 2A and step 2B in §4.3.6 can be overlapped, we use a two cycles for the ATCache hit latency. For a miss in the ATCache, we add a *a one cycle lookup latency* (Step 2A) to the total access latency. It is worth noting that similar to miss-predictors in prior works [4], the *ATCache does not cache SetTags for DRAM cache writes*. This is because such writes are not in the critical path of the performance. Besides, as introduced in §4.3.3, we use dynamic PG tuning to optimize the tag accesses with a DPG table of 1024 entries (2.675KB overhead). In summary, in this design we use less than 0.5% of the SRAM space used by tags-in-SRAM.

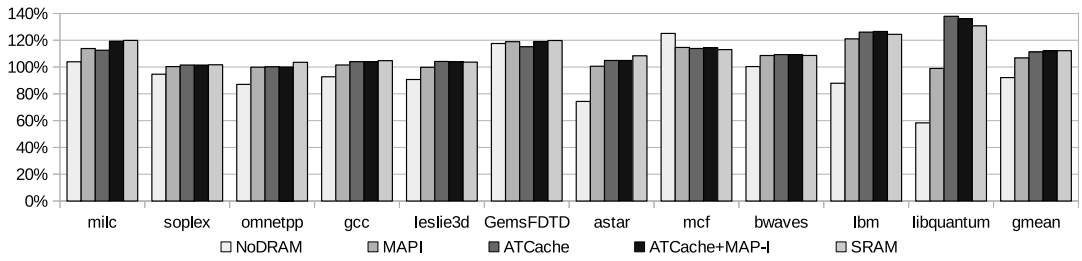

Figure 4.8: Performance results.

4.5 Results

4.5.1 Performance

The IPC improvements (normalized to the baseline) are shown in Figure 4.8. From the results, we can see that a system without DRAM cache is 8.6% slower than the tags-in-DRAM baseline; from this we can conclude that the DRAM cache is effective for our system configuration.

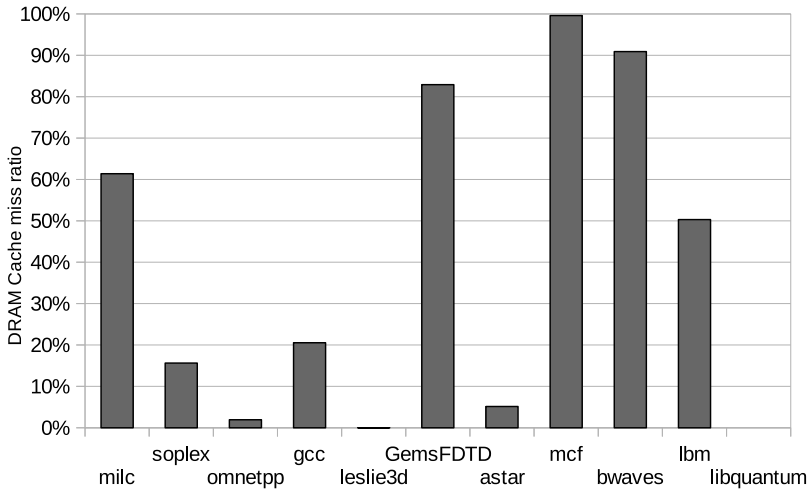

Figure 4.9: L3 miss ratio.

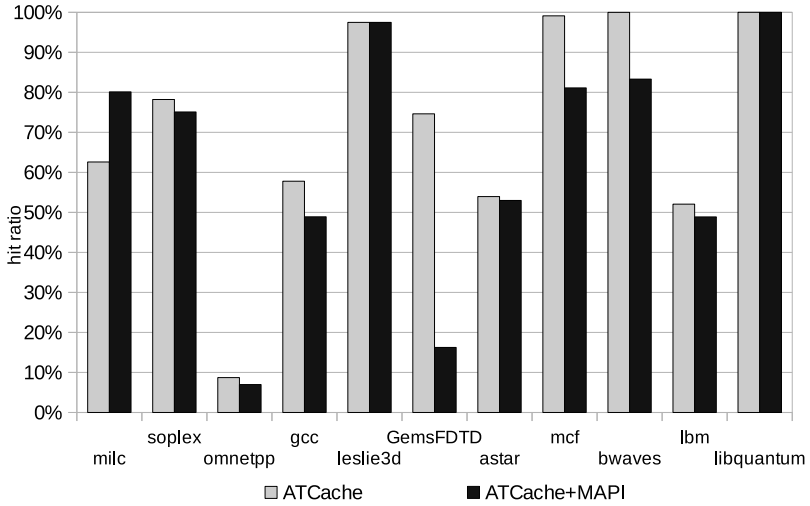

Figure 4.10: ATCache hit ratio.

We can also observe that a prior proposed miss predictor (MAP-I) improves over the baseline by 6.8%. In contrast, a tags-in-SRAM approach which consumes an impractically large SRAM space (11.5MB) provides an improvement of 12.2%. It is worth noting that this is the gap that ATCache tries to bridge while consuming much smaller SRAM space.

We can see that ATCache achieves 10.3% improvement without MAP-I predictor and 11.9% with the predictor – while consuming only 47.375KB (without MAP-I) and 48KB with MAP-I (256 entries). In other words, we are able to do almost as well as tags-in-SRAM while consuming 0.5% SRAM overhead compared to a tags-in-SRAM design.

To understand the speedups in individual benchmarks, we show the miss ratio of each benchmark program in Figure 4.9. One interesting aspect to note here is that, by skipping miss accesses, MAP-I can provide a good speedup and even outperforms a tags-in-SRAM design for high miss ratio workloads such as *mcf* and *bwaves*. This is because the latency of accessing MAP-I structure is only one cycle (in comparison to 6 cycles access latency we used for a full tags-in-SRAM cache). However, for benchmarks with a low miss ratio (such as *libquantum*), MAP-I is not as effective and even shows a small slowdown due to its cycle lookup penalty.

On the other hand, with our ATCache, we can observe benefit on both high and low miss ratio benchmarks. This is because ATCache reduces both hit latency and miss penalty. For example, for *libquantum* which has a low miss ratio, we are able to achieve an improvement of 38% over the baseline.

When we integrate MAP-I and ATCache together (ATCache + MAP-I), we can see a speedup boost in moderate miss ratio workloads such as lbm and milc. This is because ATCache's average hit ratio is around 60% (as shown in Figure 4.10), but the high accuracy of MAP-I predictor (with a predictability of close to 98%) can help us further in reducing the DRAM miss penalty. In addition to this, with MAP-I the ATCache can be more effectively used for caching only hit tags. This is exemplified by *milc* benchmark where using an ATCache with MAP-I results in an improvement of 28.9% in comparison with an improvement of 18.9% (MAP-I only) and 11.8% (ATCache only).

However, for low miss ratio benchmarks such as *libquantum*, ATCache+MAP-I shows a small 1% slowdown compared to only ATCache design. This can be attributed to the increased hit access latency (extra cycle) for looking up the MAP-I table.

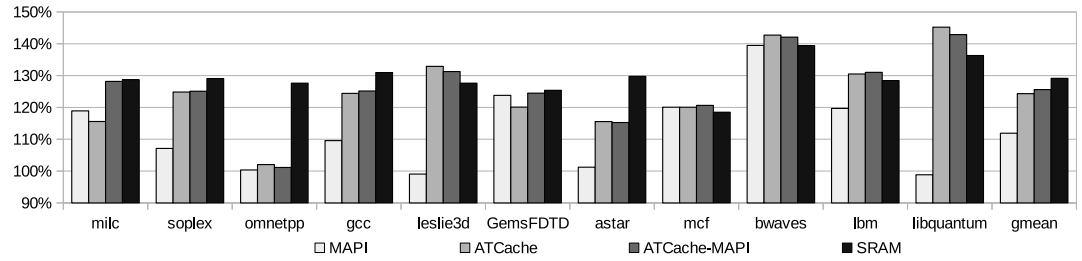

Figure 4.11: L2 miss latency reduction (higher the better).

4.5.2 L2 miss latency

Figure 4.11 shows the L2 miss latency reduction (normalized to the baseline) for all benchmarks. The L2 miss latency here refers to the time it takes for a cache block to be transferred to the L2 from the lower levels upon an L2 miss. In other words, this only includes the L3 (DRAM cache) latency and possibly the main memory latency (in the case of a DRAM cache miss). This latency provides us with a more transparent indicator of the benefit of our proposal as it avoids the obfuscating effects of other parameters (such as the effects of the our-of-order processor). As we can see from Figure 4.11, with our ATCache, we can see up to 45.3% (21.2% on average) reduction in L2 miss latency compared to the baseline. It is worth noting that this is approximately double the reduction provided by MAP-I, which provides a reduction of 11.9% on average. Finally, ATCache+MAP-I is able to provide the maximum reduction over the baseline of 24.5%. From these results, we believe that our technique can effectively improve the performance of the DRAM cache.

Caching Ratio	2	4	8	16	32
latency (cycles)	5	5	4	4	3
Caching Ratio	64	128	256	512	1024
latency (cycles)	3	2	2	1	1

Table 4.7: Latency in different caching ratios – caching ratio of 1 equals tags-in-SRAM (6 cycles).

4.5.3 Sensitivity towards caching ratio

In this section, we want to understand the effect of varying the caching ratio. As introduced in §4.3.1, caching ratio represents the area gain of our technique in comparison to tags-in-SRAM. A caching ratio of 1 refers to a tags-in-SRAM design. Generally,

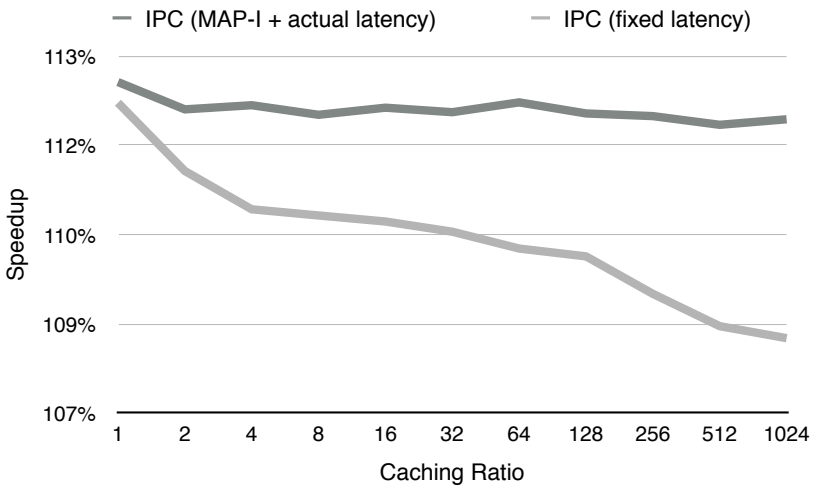

Figure 4.12: Performance improvement for different caching ratios. *Caching Ratio: 1* means a *tags-in-SRAM* design.

the miss ratio of ATCache will increase when the caching ratio is increased (smaller cache). Figure 4.12 shows IPC improvement (average of all benchmarks) for different caching ratios. Here, "IPC (no MAP-I + fixed latency)" refers to a configuration where we fixed the access latency of ATCache to 6 cycles. The improvement in terms of IPC reduces from 12.2% to 9.0% when caching ratio is increased from 1 (tags-in SRAM, 11.5MB) to 256 ($\simeq$48KB). Even without accounting for (a) faster access latency that a smaller cache could provide and (b) the miss predictor, we believe this trade-off is still interesting – a reduction in 99.5% space for 3.2% reduction in performance. "IPC (MAP-I + fixed latency)" refers to the configuration that we consider the effect of the miss predictor. As we can see from the figure, the performance gap between caching ratio 1 and 256 further reduces to 1.5%. This is because the miss predictor can help to avoid the slowdown caused by increased miss-rate in ATCache. Finally, "IPC (MAP-I + actual latency)" refers to the realistic configuration in which we consider the effect of a faster SRAM latency due to a much smaller cache (Table 4.7 shows latencies for different caching ratios) and also the MAP-I predictor. As we can see, the gap between caching ratio 1 and 256 is only less than 1% in this configuration.

4.5.4 Sensitivity towards DRAM cache size

Figure 4.13 shows the effect of varying DRAM cache size. We consider 3 different sizes (128MB, 256MB, and 512MB). In this experiment, we use the latencies that we modeled in Table 4.1 for tags-in-SRAM. For ATCache (we fix caching ratio at 256),

we use ATCache latency of 1, 2, and 3 cycles for ATCache size of 24KB (for 128MB DRAM cache), 46KB (for 256MB DRAM cache), and 88KB (for 512MB DRAM cache). From the results, we can see that ATCache has a 9% to 10% performance improvement over its baseline (tags-in-DRAM) for all sizes. With MAP-I predictor, ATCache+MAP-I is 4% to 5% better than the MAP-I-only.

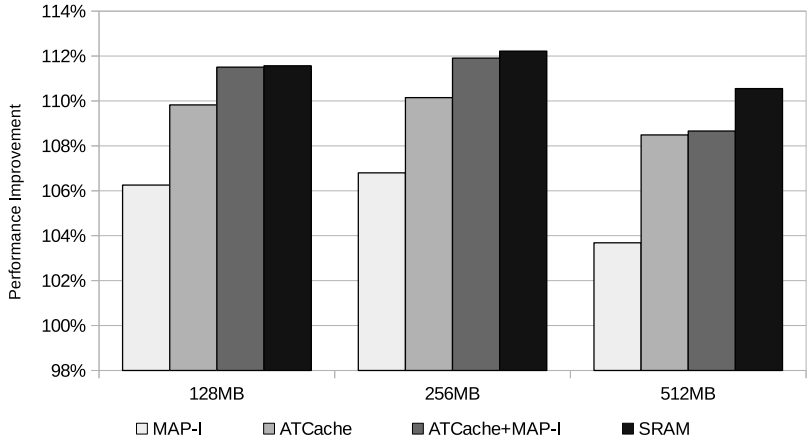

Figure 4.13: Sensitivity study of DRAM cache sizes.

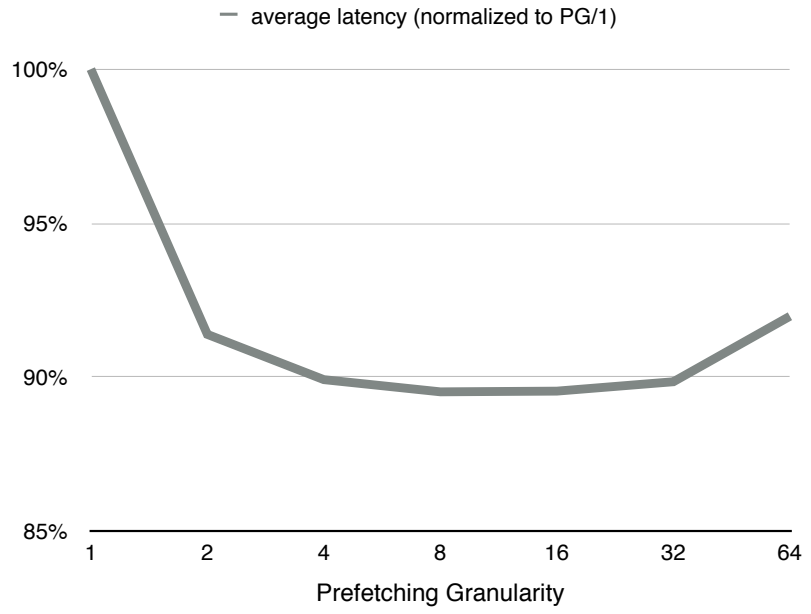

Figure 4.14: Sensitivity study to PG (with MAP-I).

4.5.5 Sensitivity towards PG

As our prior studies (§4.3.2 and §4.3.3) show, prefetching granularity (PG) is a key parameter in our design. In this section, we study the effect of varying the PG. In this study, we represent the performance of DRAM cache in terms of its average access latency, normalized to the access latency for PG/1. MAP-I is integrated with ATCache for this study. As we can see from Figure 4.14, increasing the PG from PG/1 to PG/2, reduces the average latency of DRAM cache by 8.7%. The average latency reaches a minimum for PG/4 (10.1% reduction) and PG/8 (10.5% reduction). However, when we keep increasing PG, ATCache starts to suffer from cache pollution and increased miss-penalty. The results show a degradation in performance in PG/64, whose average latency is 2.5% higher than PG/8. This study also vindicates our decision of choosing PG/4 for our baseline as it is close to the best performing PG.

4.5.6 Effect of Dynamic PG tuning

In this work, ATCache exploits spatial locality by prefetching nearby tags (for example PG/4 in our baseline). However, this could potentially incur additional wasteful tag accesses in the DRAM cache, when the fetched tag is not accessed. In this section, we study the number of tag accesses of our proposed technique. For this experiment, we use the same baseline configuration as our prior study. We compare ATCache in two configurations: with Dynamic PG tuning (DPG) and without DPG. We include a MAP-I in ATCache and normalize our results to tags-in-DRAM + MAP-I. As shown in Figure 4.15, we can see tag accesses increase by 36% without DPG. With DPG, the additional tag accesses reduce to 12%. It is important to note that if we consider overall DRAM cache accesses (tag + data), the additional accesses of our technique is about 6%. Since the DPG use training to obtain to the PG number of a specific memory region, this could potentially affect performance. However, we observe that the impact is minimal – indeed, the performance of ATCache+MAP-I + DPG (11.9%) is almost as good as ATCache+MAP-I (12.1%). This leads us to believe that including DPG is a reasonable trade-off for reduced additional accesses for a small drop in performance.

4.5.7 Multiprogrammed workloads

IPC: In this section, we study how our design performs under multiprogrammed workloads. We randomly generate 25 workload groups as shown in Table 4.6, with each

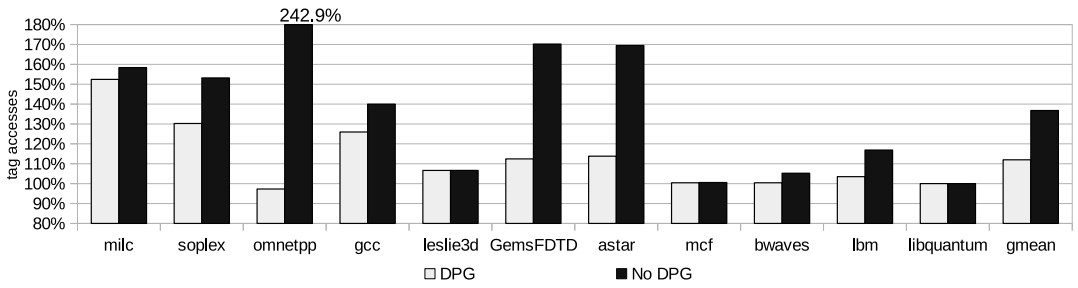

Figure 4.15: Tag Accesses

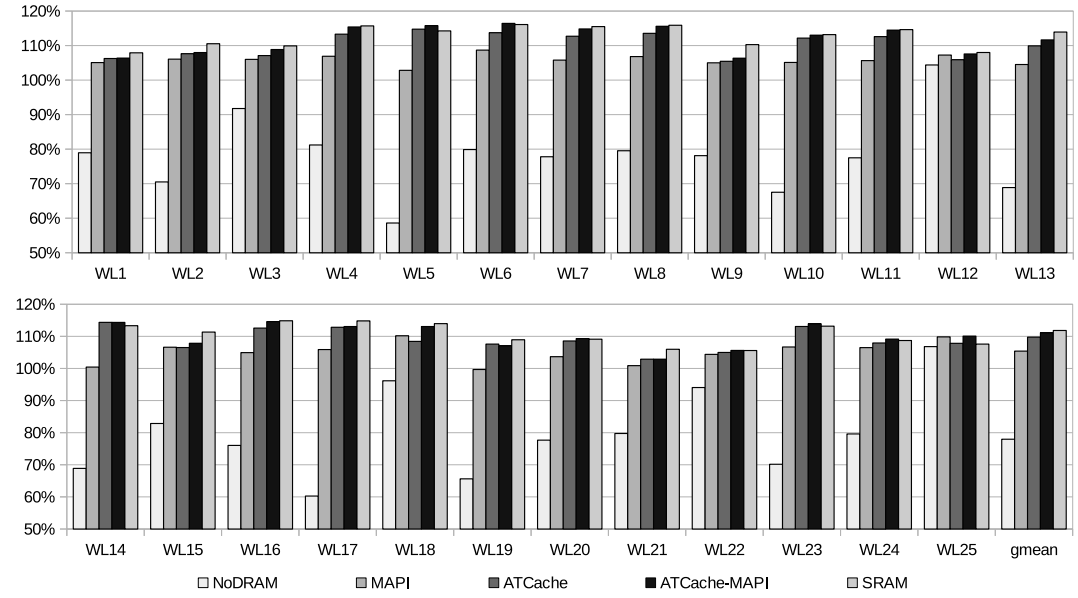

Figure 4.16: Performance improvement in multiprogrammed workloads.

group consisting of 4 workloads. The performance results compared to the baseline system are shown in Figure 4.16. Each workload group's result is a geometric mean of 4 workloads' IPC improvement. The average is geometric mean of 4 x 25 workloads. As we can see, MAP-I provides 5.4% improvement, and a full tags-in-SRAM design gives 11.8% improvement over the baseline. It is worth noting that a system without DRAM cache is 22.0% slower than our baseline. This is because multiprogrammed workloads have significantly more L2 misses. On the other hand, ATCache+MAP-I provides up to 12.2% improvement and on average 10.7% improvement over the baseline. Since this is comparable to the speedups we obtained with our uniprocessor workloads, this shows that ATCache continues to work well under multiprogrammed workloads.

L2 miss latency: To sidestep the effect of different speedup metrics that can be potentially used to summarize the performance of multiprogrammed workloads [46, 47],

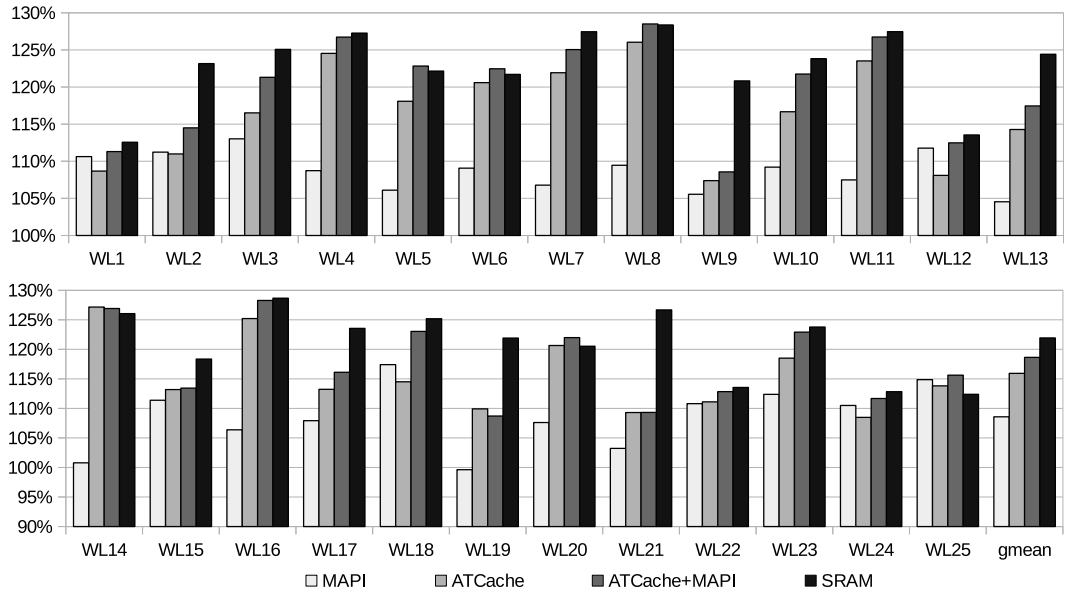

Figure 4.17: L2 miss latency reduction in multiprogrammed workloads (higher the better).

we compare the L2 miss latency in Figure 4.17, as it is arguably a more transparent indicator of the efficacy of ATCache. As we can see from Figure 4.17, ATCache with MAP-I predictor shows an 18.4% improvement in L2 miss latency over the baseline; in comparison, tags-in-SRAM shows 21.9% improvement. It is worth noting that a tags-in-SRAM design consumes about 256 times of SRAM space than our design. Again, from these results, we can conclude that our design continues to work effectively in a more memory-intensive scenario.

4.6 Related Work

The idea of caching tags has been explored previously [48, 49]. Mesa et al. proposed using a tag buffer (TIMBER) to improve the performance of hybrid memories. Compared to their work, the major difference is that ATCache further enhance the caching efficiency by exploiting cost-effective prefetching and the miss predictor. Wang et al. [48] also proposed CAT cache for reducing the area overhead of storing tags in the SRAM cache. CAT cache exploits tag value locality by removing tag value duplication – in doing so, they are able to reduce the area overhead of the tag array with a small performance overhead. However, this work is not applicable in the DRAM cache con-

text since CAT involves accessing the data array first before accessing the tag; this is precisely what works on DRAM cache seek to avoid.

Other works [50, 51] have proposed caching tags to improve cache access latency in the generic setting of a multilevel SRAM cache hierarchy. The key difference between this and our work is the new context (DRAM cache) in which target this idea. By targeting the idea to a novel and concrete context of DRAM caches, we are able to specialize our technique in ways over and beyond the prior works. For example, hit prefetching, through which we gain a significant chunk of our performance, has not been discussed in any of the prior works.

4.7 Conclusion

The advent of the DRAM cache has posed a problem of how to efficiently manage the tags associated with the DRAM cache. One naive option is to store all the tags in SRAM; while this would ensure fast access of the tags, the associated storage cost would render this approach impractical. Consequently, prior works have proposed innovative techniques to manage the tags efficiently in DRAM. Nonetheless, we observe, with a help of a study, that it is more performance efficient to manage the tags in SRAM.

Having established this, we propose a simple idea to cache the tags in SRAM so that we can achieve the effect of maintaining all tags in SRAM, without paying the prohibitive cost; we show that there is enough spatial and temporal locality amongst DRAM cache tag accesses to merit caching the tags. Our experimental results show that we achieve similar performance (within 2%) to a very fast tags-in-SRAM design (6 cycles access latency for 11.5MB), while consuming less than 1% of the SRAM space. If we integrate our caching idea with prior proposed miss prediction, we show that we can come within 0.5% of performance achieved with the tags-in-SRAM approach.

In the next chapter, we will target on a different problem in DRAM caches – the design of the DRAM cache controller. While many tags-in-DRAM designs are proposed in recent works, we noticed that the study of how to design a DRAM cache controller is missing. Therefore, our next work will address problems in the DRAM cache controller and propose a design that adapts to DRAM caches.

Chapter 5

DRAM-cache-aware DRAM controller

5.1 Introduction

While many recent works [10, 11, 3, 4, 52], including our previous work, have proposed storing these cache tags in the stacked DRAM array, storing these tags in the DRAM array, however, increases the complexity of a DRAM cache request. In contrast to a conventional request to DRAM main memory, a request to the DRAM cache will now translate into multiple DRAM cache accesses (tag and data). In this work, we address the question of how to schedule these DRAM cache accesses. We start by exploring whether or not a conventional DRAM controller will work well in this scenario.

A conventional DRAM controller usually consists of two queues — a read queue and a write queue. Each read (writeback) request to DRAM memory requires only one read (write) access to DRAM array; these read (write) requests will be stored in the read (write) queue respectively. Switching between read and write modes on the DRAM bus is known as a *turnaround*, which incurs a latency known as turnaround delay. Consequently, DRAM controllers strive to avoid frequent turnarounds, as that can be detrimental to overall performance. Furthermore, DRAM controllers typically prioritize reads over writes. This is because, read requests tend to be in the critical path of system performance, whereas write requests are usually not.

Unlike requests to conventional DRAM memory, requests to a DRAM cache will translate into multiple DRAM accesses. For example, a read request to a set-associative DRAM cache can translate into three accesses: 1) tag read; 2) data read; and 3) tag

write[1]. Similarly, a write request to the DRAM cache[2] could translate into one tag read and two writes (tag and data). In addition, the DRAM cache also needs to handle refill requests from lower level memory. This complexity increases the challenge of designing a DRAM cache controller. Therefore, a study on how to design DRAM cache controllers is a critical step in the adoption of DRAM caches.

Study: Baseline. We start by exploring a baseline design, which is a natural extension of a conventional DRAM controller. In this design, the DRAM cache controller simply pushes accesses into the read or write queue depending on the DRAM access type (read or write). For example, if a DRAM cache read request translates into one read and two write accesses, the read access will be placed in the read queue and write accesses will be placed in the write queue. We call this *Baseline Design* (BD). Unfortunately, BD suffers from two limitations. First, we observe that a read access from a read request can be blocked by those read accesses coming from a writeback request. This is not desirable since read accesses from a read request are usually in the critical path of system performance. Second, we observe that BD suffers from a large number of row conflicts, because of interference between a read and writeback requests to the DRAM cache. We refer to this problem as Read-Write Interference (RWI) and discuss this in more detail in §5.2.4. It is worth noting that a similar problem has also been observed in DRAM memory previously [53], but we show how the problem is more severe in this context.

Study: Enhanced Design. To avoid these limitations, we consider an enhanced design based on BD, in which the controller pushes accesses into the read or write queue depending on the request type (and not the access type). Specifically, all accesses associated with a read (writeback) request will be placed on the read (write) queue. We call this *Request-Oriented Design* (ROD). Unfortunately, we observe that ROD suffers from increased turnaround delays. Furthermore, we observe that ROD suffers from longer write queue servicing latency compared to BD. This is because ROD will not schedule read tag accesses for write requests, even when the DRAM cache bus is idle. These read accesses will be scheduled only when the controller starts servicing the write queue. This increased servicing time for write queue could eventually hurt overall system performance.

[1] Accesses 2 and 3 will only happen if the read request hits in the DRAM cache. It is worth noting that this translation could vary for different DRAM cache settings. Please see §5.2.2 for details.

[2] A write to the DRAM cache is actually the result of a writeback from the higher level, since we assume a write-allocate policy in our cache hierarchy. In this work, DRAM cache "write" and "writeback" are used interchangeably.

Proposed Design. Based on the above observations, we come up with a set of principles that a DRAM cache controller should ideally satisfy. The DRAM cache controller should: 1) take into account both the access type as well as request type in prioritizing accesses; 2) minimize the number of turnarounds; 3) avoid RWI; and 4) avoid increasing the write queue servicing latency.

We propose DRAM-Cache-Aware (DCA) DRAM controller that is based on the above principles. Similarly to BD, DCA holds write accesses in a write queue; the read accesses, however, are held in two distinct read queues: the conventional read queue (RQ) which serves as a high priority read queue and a low priority read queue (LPRQ). The RQ holds read accesses that are in the critical path — i.e., tag and data reads from read requests. The LPRQ allows us to schedule read accesses that are not in the critical path — i.e. tag reads from DRAM cache writeback requests and cache refill requests.

Similarly to the write queue, the LPRQ is scheduled passively. Unlike the write queue, however, servicing requests of the LPRQ is not restricted by turnaround delay considerations. In the case of the former, we need to ensure that the write queue services at least a minimum number of writes before returning to the read queue, to amortize the cost of a turnaround. The LPRQ has no such constraints because bus turnaround is not required to switch between RQ and LPRQ. This allows us to use an opportunistic flushing scheme (OFS) for LPRQ. So, instead of using a completely passive queue servicing scheme which only service requests when the queue is nearly full, OFS will seek to service/flush LPRQ whenever: 1) the read queue is empty, and 2) RWI cannot happen.

In comparison with BD, DCA ensures that non-critical read accesses do not block critical read accesses (from read requests); in addition, RWI is explicitly minimized. In comparison with ROD, DCA experiences reduced servicing times, as part of the write queue from ROD is moved into the LPRQ in DCA and flushed opportunistically; also, turnaround delays are explicitly minimized.

Results and Contributions Our experiments on multiprogrammed workloads show that DCA is 15.6% (14.7%) faster on average in comparison with the baseline when we use a 16-way (direct-mapped) DRAM cache setting.

The contributions of this work are as follows:

- To our knowledge, this is the first study on the impact of the DRAM cache controller on the performance of the DRAM cache.

- We first establish a baseline DRAM cache controller (§5.2.3.1) that is based on a conventional DRAM controller. We also introduce an enhanced version of the baseline design (§5.2.3.2). We study the limitations of these designs.
- We propose DRAM-Cache-Aware (DCA) DRAM controller that addresses the limitations of the above two designs.
- Our work is orthogonal to existing DRAM cache works which strive on improving DRAM cache's structure. Therefore, our observations and design can apply to any of the recently proposed tags-in-DRAM designs [10, 11, 3, 4] (§5.6).

5.2 Motivation

To motivate this work, we first briefly discuss the basics of a DRAM controller (§5.2.1) and accesses in DRAM cache (§5.2.2). We then study the limitations (§5.2.3 and §5.2.4) of adapting a conventional DRAM controller for the DRAM cache.

5.2.1 Basics of DRAM Controller Design

In a DRAM system, the DRAM bus can be used to service a read or a write request at any given time. Switching the bus between read and write modes is known as turnaround, which incurs a latency known as turnaround delay. Typically, in DDR3-1600, a write to read turnaround (t_{WTR}) will cost 7.5 ns and a read to write turnaround (t_{RTW}) will cost 2.5 ns. Frequent bus turnarounds will result in a performance loss due to these extra latencies. To avoid these turnaround overheads, conventional DRAM schedulers commonly store read and write requests in separate queues — namely, read queue and write queue. Read queue will be served with a higher priority since read requests are usually in the critical path of system performance. On the other hand, write requests are handled by a passive queue management scheme. The simplest scheme is to service the write queue only when the write queue is close to full.

In our work, we consider an optimized scheme which uses two thresholds — a high threshold and a low threshold to determine the queue servicing point. On reaching the high threshold, the DRAM controller will trigger a forced flush of the write queue. In addition to that, if there are no pending read requests and the occupancy of write queue is greater than the low threshold, the DRAM controller will also service the write queue. The pseudo code of how to control these queues is shown in Code 5.1[3]. By

[3]Is is worth noting that this scheme is implemented in both gem5 [41] and zsim [54]

prioritizing reads over writes, the DRAM controller can avoid turnaround overheads and enhance performance.

```
if (busState == READ) {
  if (readQueue.empty()) {
    //no pending read request
    //try to schedule write request if ...
    if (writeQueue.size() > writeLowThreshold) {
    //turn busState to WRITE for DRAM writes
      busState = WRITE;
    } else {
      //nothing to do
      return;
    }
  } else {
    serviceRequest(readQueue);
    if (writeQueue.size() > writeHighThreshold) {
      busState = WRITE;
    }
  }
} else {            //if busState = WRITE
  serviceRequest(writeQueue);
  //switch to read status if ...
  if (writeQueue.empty() ||
    writeQueue.size() < writeLowThreshold ||
    !readQueue.empty()) {
    busState = READ;
  }
}

```

Code 5.1: Pseudo code for the write queue servicing scheme used in this work

5.2.2 Accesses in DRAM Cache

In conventional DRAM memory, mapping between request and DRAM access is quite simple — a read request translates to a read access and a write request translates to a write access. Compared to conventional DRAM memory, a set-associative DRAM cache requires a more complex mapping. One request typically translates to multiple DRAM accesses. In this section, we describe mappings for three main type of requests in the cache:

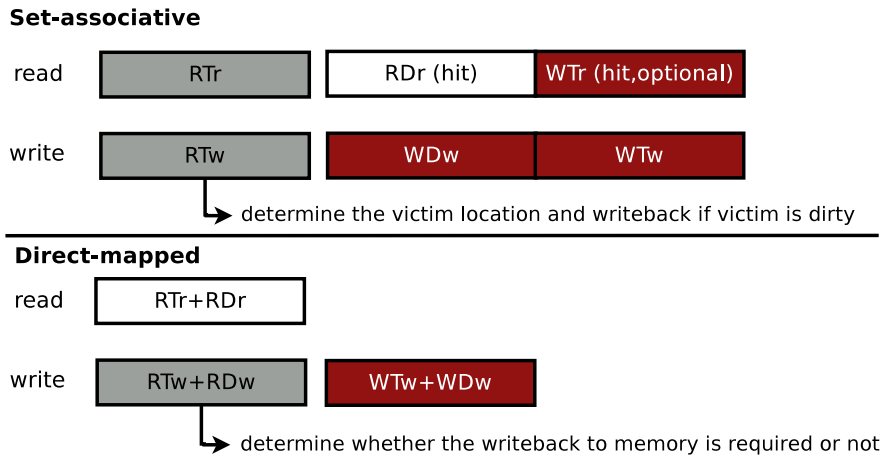

Figure 5.1: Accesses in a cache read and a cache writeback.

Read Request. A read request in a cache will translate to several DRAM accesses as shown in Figure 5.1: 1) a read tag (RTr) to determine hit/miss and identify the location of the data, 2) a read data to satisfy processor request (RDr), and 3) a write tag to update replacement bits (WTr). Step 2 and 3 will only happen if request hits in the cache. It is worth noting that *WTr is not required if replacement bits are stored in the SRAM, or if the DRAM cache is organized as a direct-mapped cache. In this work, we assume replacement bits are stored in the SRAM so that our study is independent of the replacement policy.*

Write/WriteBack Request. A cache write request translates to at least three accesses as shown in Figure 5.1: 1) a read tag to obtain the current tag status (RTw); 2) a write data (WDw) and 3) a write tag (WTw). It is worth noting that if the dirty flag of the replaced block is set, a read data is also required (RDw).

Refill Request. A cache refill means a block is brought back from the lower level memory and it is waiting to be written into the DRAM array. This translation is identical to write request.

In addition to above requests, there are a small number of other requests to maintain the coherence. Because the DRAM cache is usually located a level below the coherent shared cache, these requests are infrequent in the DRAM cache. Most of coherence overhead is handled at the shared cache level.

5.2.3 Potential Designs

In this section, we propose two adaptations of a conventional DRAM controller. These designs can be differentiated based on how they classify DRAM accesses correspond-

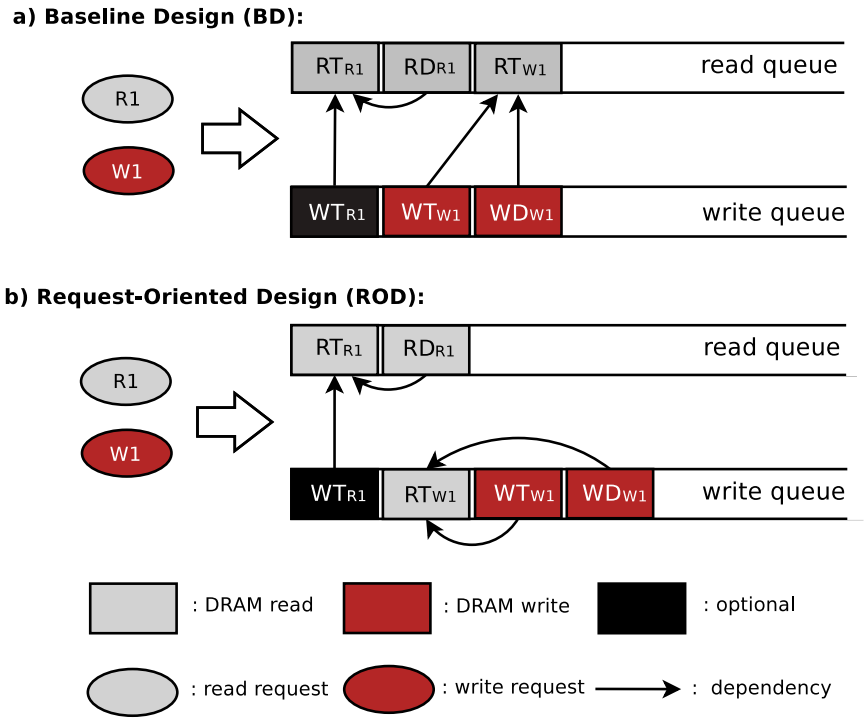

Figure 5.2: How the translated accesses map to queues in BD and ROD.

ing to DRAM cache requests. We then illustrate the limitations of these designs using examples.

5.2.3.1 Baseline Design (BD)

We first introduced a potential design which is based on DRAM memory controller, we call it *Baseline Design (BD)*. As described in §5.2.2, a DRAM cache request requires both read and write accesses to the DRAM array. In BD, DRAM accesses corresponding to a request are classified based on access type. Read accesses will go the the read queue and write accesses will go to the write queue, irrespective of their request type, as shown in Figure 5.2 (a). This design is very similar to a conventional DRAM controller design as described in §5.2.1. Using this design, the DRAM cache controller can minimize the frequency of turnarounds.

However, this design has potential performance issues. Since the read queue contains accesses from both read and write requests, read accesses corresponding to read requests compete with read accesses corresponding to write requests. This interference can potentially delay the completion of read requests. In fact, this delay can result in an overall performance degradation, because read requests are usually in the critical path of processor execution.

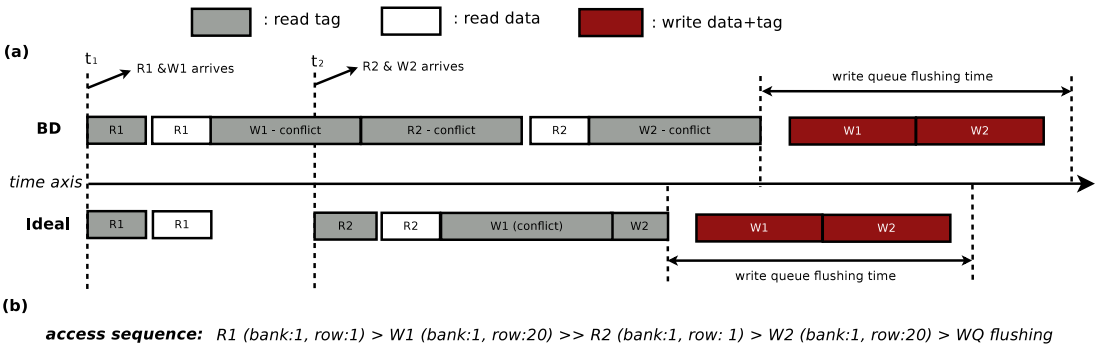

Figure 5.3: A case study in baseline design (BD).

Figure 5.3 shows an example of how write interference can delay read requests in BD. Consider the sequence of cache requests shown in Figure 5.3 (b) (where R* are read requests and W* are write requests). The DRAM cache first receives requests R1 and W1 at time t_1. These requests create three read access entries in the read queue — read tag for R1 (RT_{R1}), read data for R1 (RD_{R1}) and read tag for W1 (RT_{W1}) — and two write access entries in write queue — write data for W1 (WD_{W1}) and write tag for W1 (WT_{W1}). BD will first schedule RT_{R1}. On completion of RT_{R1} it will schedule RD_{R1}. After request R1 is completed, RT_{W1} will be scheduled. As shown in Figure 5.3(b), because R1 and W1 are accessing a different row in the same bank, RT_{W1} will cause a row conflict and the controller will have to close the previously opened row and re-open another row for request W1. Requests R2 and W2 arrive while the controller is busy handling RT_{W1}. This results in delaying request R2 because it needs to wait for RT_{W1} to complete. After completing RT_{W1}, BD will schedule RT_{R2} and cause another row conflict.

As we can see from this example, BD will trigger three row conflicts in four accesses. Moreover, request R2 is delayed because it has to wait for RT_{W1}'s completion. In the figure, we also show an ideal case in which we avoid scheduling RT_{W1} between R1 and R2. Thus, when request R2 arrives it will be scheduled immediately. Also, the ideal case will trigger only one row conflict in this entire request sequence.

5.2.3.2 Enhanced design: Request-Oriented Design (ROD)

To avoid the issues seen in BD, we show an enhanced alternate design. In this design, DRAM accesses are classified based on their corresponding request type. Accesses corresponding to read requests will go to the read queue and accesses corresponding to

write requests will go to the write queue [4] as shown in Figure 5.2. We call this design *Request-Oriented Design (ROD)*.Thus both the read and write queues can contain a mixture of read and write accesses to the DRAM array. The advantage of this design is that it will eliminate write interference from delaying read requests.

However, this design also has its own set of limitations. Since the write queue contains both read and write accesses, the frequency of bus turnarounds will increase. This in turn will increase the write queue servicing time. Moreover, the increase in this write queue servicing time can potentially degrade the overall system performance because it can delay subsequent read requests.

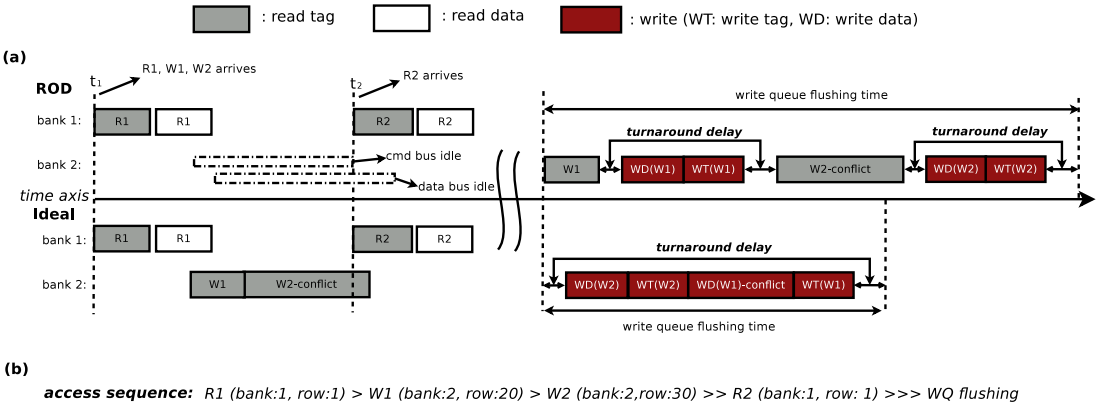

Figure 5.4: A case study in Request-Oriented Design (ROD) — please note that bank 1 and bank 2 are in the same rank/channel and all queues are using FR-FCFS policy.

Figure 5.4 shows an example highlighting this problem. Consider the sequence of cache requests shown in Figure 5.4 (b) (where R* are read requests and W* are write requests). The DRAM cache first receives requests R1, W1 and W2 at time t_1. These requests create two read access entries in read queue — read tag for R1 (RT_{R1}) and read data for R1 (RD_{R1}) — and two read and four write access entries in write queue — read tag for W1 (RT_{W1}), write data for W1 (WD_{W1}), write tag for W1 (WT_{W1}), read tag for W2 (RT_{W2}), write data for W2 (WD_{W2}) and write tag for W2 (WT_{W2}). ROD will first schedule RT_{R1}. On completion of RT_{R1} it will schedule RD_{R1}. After request R1 is completed, ROD will not schedule RT_{W1} until the write queue reaches its flushing (servicing) condition. Request R2 arrives at time t_2. Now, ROD will schedule RT_{R2} and RD_{R2} one after the other. Eventually, ROD starts servicing the write queue. In addition to the writes, RT_{W1} and RT_{W2} also need to be processed. This results in a longer write queue servicing time.

[4] With one exception: the write tag for a read request (if present) would go to write queue for performance reasons.

As we can see from this example, ROD fails to utilize the idle time between processing requests R1 and R2 to schedule RT_{W1} and RT_{W2}. Whereas in the ideal case shown in the figure, read tags for write requests are scheduled opportunistically and the overall completion time of the sequence is lower.

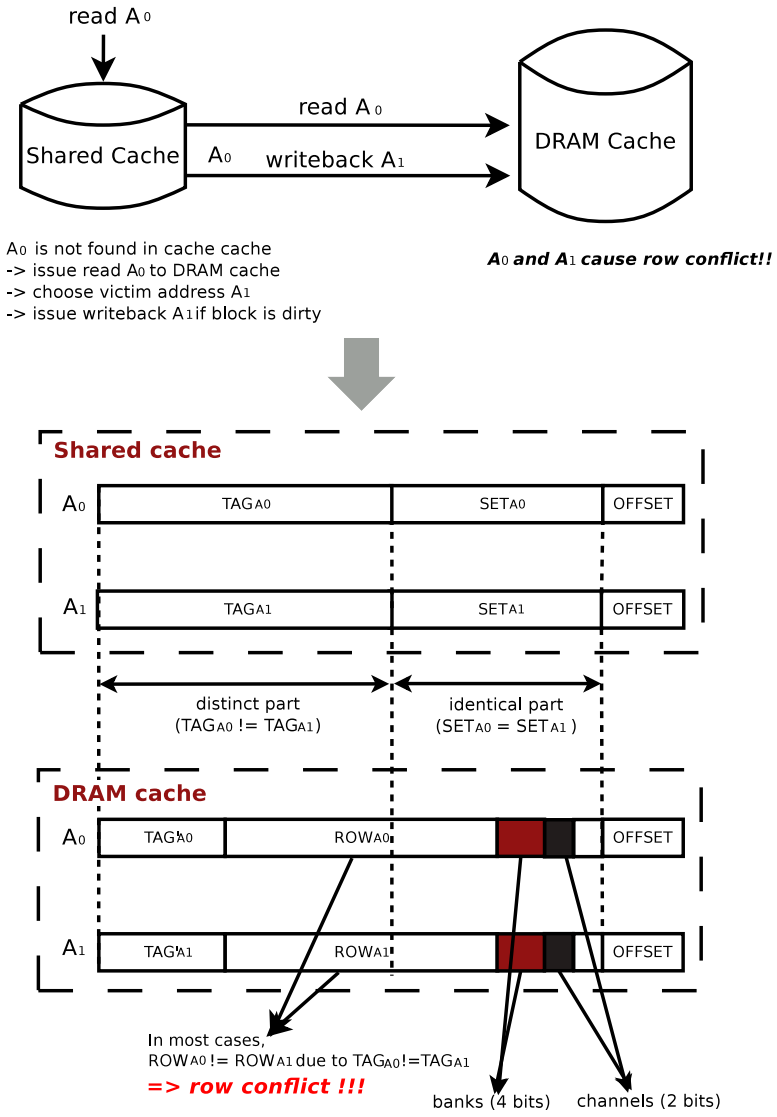

Figure 5.5: Row conflict in DRAM cache due to shared cache replacement

5.2.4 Read-Write Interference

In this section, we want to introduce a memory access pattern that could potentially result in many row conflicts in the DRAM cache. It is worth noting that that a similar access pattern has also been observed in the memory side [53]. In this work, we refer to this observation as *Read-Write Interference (RWI)*.

To explain RWI, we fist consider a cache sub-system with three-levels of cache hierarchy: private L1s, a shared L2 and an L3 DRAM cache (see Table 5.1 for more information). Due to set-associativity, a part of any two addresses (say A_0 and A_1) mapping to the same set in L2 will be identical as shown in Figure 5.5. Also, considering a DRAM cache organization with 4 cache sets in a row, 16 banks, 1 rank per channel, and 4 channels, addresses A_0 and A_1 will be located in the same bank/rank/channel. If we accesses them consecutively, it will lead to a potential row conflict.

In this configuration, when a request (for some address A_0) misses in the L2, the request will be forwarded to DRAM cache. In addition to that, L2 cache will also find a victim block (say address A_1) in the same cache set. If this victim block A_1 is dirty, L2 will issue a writeback request for A_1 to DRAM cache. There is a high probability that these two requests (request for A_0 and writeback for A_1) will arrive in L3 together (or within a short interval). And since they arrive within a short interval, it is also likely that they will be processed consecutively. As described in previous paragraph this will cause a row conflict.

On the conventional DRAM memory side, many DRAM controllers employ a strategy called *write caching* [17], which simply buffers the write requests in a write queue. In our study, this strategy[5] can effectively avoid RWI on the memory side.

In the DRAM cache, RWI issue becomes more complex. Every writeback in DRAM cache requires a read tag to check if the victim (replaced) block is dirty or not. It is worth noting that this read tag operation is required in the direct-mapped design [4] also. Therefore, the access sequence in Figure 5.3 will significantly hurt the performance if we choose BD.

5.3 DRAM-Cache-Aware Controller

In this section, we first propose a set of design principles for designing a DRAM cache controller. We then propose DRAM-Cache-Aware (DCA) DRAM controller which is designed based on those principles

5.3.1 Design Principles

Based on our observations in §5.2, we propose a set of principles for designing a DRAM cache controller as follows:

[5]For the write queue, we employ a two-threshold write queue servicing scheme (as mentioned in §5.2.1).

- Take into account both the access type as well as request type in prioritizing accesses (§5.2.1) — this can avoid the scenario where priority reads are delayed by low-priority reads.
- Minimize turnarounds (§5.2.1) — frequent turnarounds result in a performance degradation because of adding extra latencies.
- Avoid RWI (§5.2.4 & §5.2.3.1) — RWI will potentially cause row conflicts and degrade the performance.
- Avoid increasing the write queue servicing latency (§5.2.3.2) — if the time of servicing the write queue is increased, subsequent read requests might be delayed.

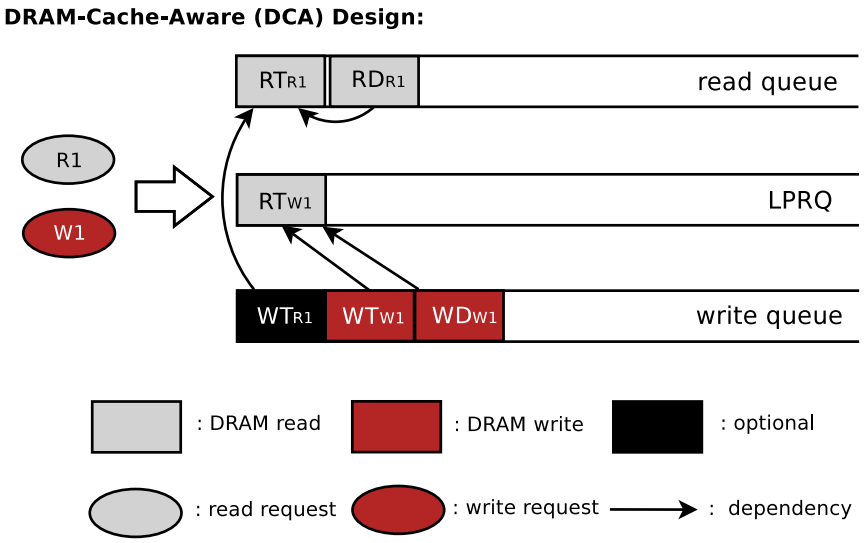

Figure 5.6: How the translated accesses map to queues in DRAM-Cache-Aware design

5.3.2 Low Priority Read Queue (LPRQ)

In our DRAM-Cache-Aware DRAM controller, we first classify read accesses in two categories — priority reads (PR) and low priority reads (LPR). *Read accesses corresponding to read requests are classified as PR* and *read accesses corresponding to write requests (including cache refill requests) are classified as LPR.* We use the existing read queue to store PR and propose a new low-priority read queue (LPRQ) to store LPRs. Figure 5.6 shows how accesses corresponding to various requests are stored in DCA's queues. LPRQ's requests will only be scheduled in two conditions — either LPRQ is full (or close to full) or on a pre-condition specified in §5.3.3.

Difference between LPRQ and ROD. Compared to ROD which simply places writes in the write queue, LPRQ is like splitting read queue into two parts. Servicing request

in the LPRQ is not restricted by turnaround delay considerations. In the case of the write queue, to amortize the cost of a turnaround, we need to ensure that it services at least a minimum number of writes before returning to read queue. The LPRQ has no such constraints because bus turnaround is not required to switch between read queue and LPRQ.

5.3.3 LPRQ Servicing Scheme

When to Service? When to service the LPRQ is an important design decision. In theory, if LPRQ will be scheduled together with the write queue, our design will be quite similar to ROD as we described in §5.2.3. On the other hand, if we schedule LPRQ together with the read queue, LPRQ design will be similar to BD and suffer from RWI issue as we described in §5.2.4. Therefore, in this work, we strive to strike a balance in the servicing scheme for LPRQ. In principle, we would like to schedule LPRQ with the read queue but avoid RWI conflicts. In our observation, RWI conflicts actually appear due to the spatial locality of DRAM accesses. This can be avoided by not scheduling those low-priority reads which are accessing recently accessed banks.

Obtain the access footprint for each bank To identify recently accessed banks, we use a technique similar to the replacement policy to let scheduler determine the recency of used banks. We use a 3-bit bank re-reference prediction counter (RRPC) to keep track of the bank re-reference history and use it to determine (predict) whether we should schedule a particular low priority request or not. This RRPC design is similar to a cache replacement technique called *re-reference interval prediction (RRIP)* [20]. In our proposed design, each bank will have a 3-bit RRPC counter which amounts to 24 bytes overhead for a DRAM organization with 64 banks. This counter will only change when a priority read is accessed. Initially, the counter is set to 0. When there is a priority read X to the DRAM controller, the system will first decrease the counter in all banks by 1 (0 will stay at 0) and set the most recently accessed bank's counter to 7. By doing this, we can determine the recency of each bank. It is worth noting that the RRPC will only be modified for priority reads (i.e. requests from read queue).

Opportunistic Flushing Scheme (OFS) With regard to the queue servicing scheme, we first define that requests in the LPRQ will only be scheduled when read queue (priority reads) is empty. When the scheduler is ready for scheduling a low priority read (assumed request A_0), we first check if there is a row conflict in corresponding bank. If there is no row conflict (either row buffer hit or a closed row), A_0 will be

scheduled. On the other hand, if there is a row conflict in that bank, we will check the corresponding bank's RRPC. If the RRPC is lower than a pre-defined threshold that we call *flushing factor* (FF), our controller will schedule A_0. However, if the request can not meet any of above criteria, it will not be scheduled and will have to wait for next available scheduling slot for LPRQ. In our study, we found that the design is not very sensitive to FF when it is smaller than 5 (FF/5). In our multiprogrammed workloads, the average performance difference from FF/4 to FF/0 is less than 1%. Therefore, in this work, *we use FF/4 as default value*.

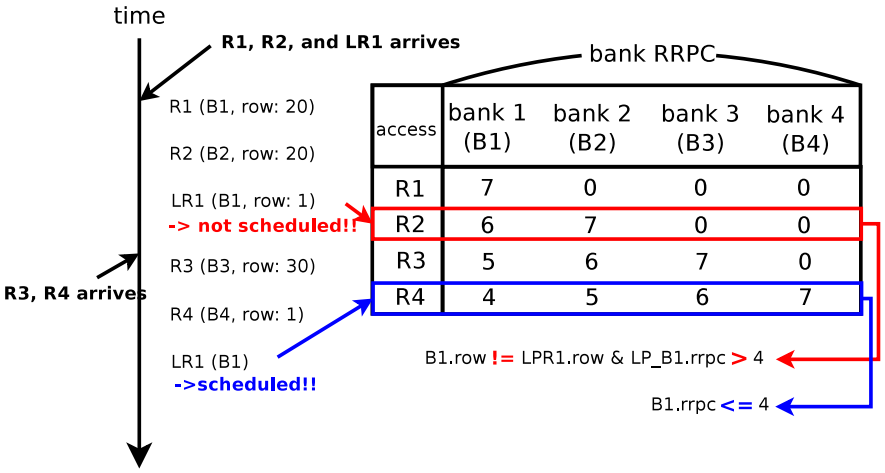

Figure 5.7: Working case for Re-Reference Prediction Counter

5.3.4 An example

In this example (as shown in Figure 5.7), we assume three read requests — R1 (bank: 1, row: 20), R2 (bank: 1, row: 20), and LR1 (low priority, bank: 1, row: 1) — come to the DRAM cache in a short period. R1 and R2 will be stored in the read queue and LR1 will be stored in LPRQ. Since read queue has higher priority than LPRQ, R1 and R2 will be scheduled first and will change B1's RRPC value to 7. After that, when the bus is available for scheduling another request, we try to schedule the request from LPRQ. The scheduler first checks if LR1 will cause a row conflict in B1. Since the last opened row is row 20, serving LR1 will need to close that row and re-open another row. In this case, we need to check if the B1.RRPC is less than or equal to the FF. Because B1's RRPC is 6, the scheduler will not schedule LR1. Later, we received two more requests R3 (bank:2, row:30) and R4 (bank:3, row:1). Since DRAM bus is idle at this moment, R3 will be scheduled immediately and R4 will be scheduled after R3.

When R4 is finished and read queue is empty, we try to schedule LR1 again. This time we can see that B1's RRPC (4) is less than or equal to the FF (4). Therefore, LR1 is scheduled.

Processor ROB	4GHz, x86, 8-wide OoO 192 Entries
L1 I/D caches	each 32KB/2way, 2 cycles, private
L2 cache	8MB, 20 cycles, shared
L3 cache	DRAM Cache, 256MB, 1/16 way
On-chip bus	4GHz, 256-bit width
Memory latency	50ns

Table 5.1: System parameters

Timing Params	tRCD-tCAS-tRP-tRAS 8-8-8-30 (ns) tWTR-tRTP-tRTW 5-7.5-1.67 (ns) tWR-tBURST 15-3.33 (ns)
Organization	16 banks/rank, 1 rank/channel 4 channels, 4KB row buffer, RoBaRaChCo, open-page
Read Queue	32 (64 for BD) entries per channel FR-FCFS
Write Queue	64 (96 for ROD) entries per channel low/high flush thres.: 50%/85% FR-FCFS
LPRQ	32 entries per channel, FF/4 FR-FCFS

Table 5.2: Stacked DRAM parameters

5.4 Experimental Methodology

We use gem5 [41], a cycle-accurate simulator for our evaluations. We model an OoO x86 core based CMP system with private L1 and shared L2 caches. A 256*MB* L3 DRAM cache is modeled using a detailed DRAM timing model. The detailed system parameters and stacked DRAM parameters that we use are shown in Table 5.1 and

1-2	soplex-mcf-gcc-libquantum	astar-omnetpp-astar-gcc
3-4	mcf-soplex-astar-leslie3d	bwaves-lbm-libquantum-leslie3d
5-6	bwaves-soplex-bwaves-GemsFDTD	omnetpp-milc-leslie3d-astar
7-8	soplex-astar-lbm-mcf	lbm-omnetpp-leslie3d-bwaves
9-10	milc-leslie3d-omnetpp-gcc	bwaves-astar-gcc-leslie3d
11-12	omnetpp-libquantum-mcf-gcc	gcc-libquantum-lbm-soplex
13-14	gcc-leslie3d-GemsFDTD-soplex	lbm-libquantum-omnetpp-bwaves
15-16	gcc-milc-leslie3d-milc	omnetpp-mcf-leslie3d-lbm

Table 5.3: Workload groupings

Table 5.2. As the DRAM cache could be organized as either direct-mapped [4] or set-associative cache [3], our work evaluates both organizations in §5.5.1. Recall that the major difference between these two designs is the number of accesses per DRAM cache request (as shown in Figure 5.1). To simplify simulation results, we assume additional row buffer space for tags. Assuming a cache tag requires 4 bytes, the actual row buffer size would be $4KB + 4 * 64bytes = 4.25KB$.

We evaluate our proposed DCA DRAM controller (LPRQ+OFS) and compare it with the baseline BD, the enhanced baseline ROD, and LPRQ-only. For the LPRQ in the DCA design, we use a queue of 32 entries. To compensate for additional LPRQ entries, we increase the read queue size to 64 entries in BD and the write queue size to 96 entries in ROD. We use benchmarks categorized as memory-intensive [9] from SPEC 2006. Using these memory-intensive benchmarks, we generated 16 4-core workloads (§5.5.1) and 8 8-core workloads (§5.5.5) for evaluating the controller designs. The details about these workloads are shown in Table 5.3 and Table 5.4.

5.5 Results

In this section, we first analyze the overall performance of the proposed DCA design in comparison with BD, and ROD (in both 16-way and direct-mapped organizations) using multiprogrammed workloads (§5.5.1). We also analyze the magnitude of turnaround overhead (§5.5.2) for the different designs. This analysis of turnaround overhead helps in explaining the improvements in performance compared to ROD. We then conduct a sensitivity study on DRAM cache's associativity, which quantitatively highlights the detrimental impact of RWI on BD. We analyze single-core workloads in §5.5.4 to understand how each benchmark individually performs under the different designs. In addition, in §5.5.5 we study the effectiveness of different designs when we

increase the number of DRAM cache requests (using 8-core workloads). Finally, we show a summary of how all techniques compare in various configurations.

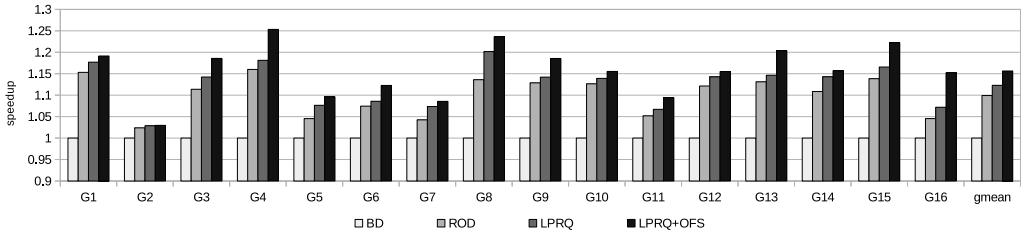

Figure 5.8: Performance speedup of all designs (16-way)

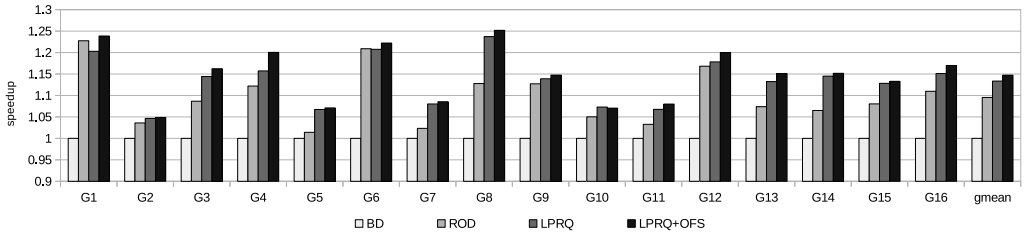

Figure 5.9: Performance speedup of all designs (direct-mapped)

5.5.1 Performance

16-way Figure 5.8 shows the speedups of various designs across all multiprogrammed workloads in 16-way setting, where results are normalized to BD (baseline). As the figure shows, BD has the worst performance amongst all designs. Furthermore, ROD has an average improvement of 9.9% and LPRQ has an average improvement of 11.2%. Among all the designs, our DCA design (LPRQ+OFS) provides the best performance and achieves an improvement of 15.6% over the baseline.

Considering individual workloads, LPRQ+OFS has an improvement of 25.3% in G4, whereas the ROD and LPRQ-only only improve 16.0% and 18.1% respectively.

Direct-mapped For the direct-mapped setting, as we can see from Figure 5.9, BD is still the worst in terms of performance amongst all designs. This is because the RWI issue (§5.2.4) also manifests in direct-mapped setting due to the additional tag/data access for writes (Figure 5.1). Similar to 16-way, in direct-mapped setting, ROD has an average improvement of 9.5% and LPRQ+OFS has an average improvement of 14.7%.

Considering individual workloads, G8 in LPRQ+OFS has the highest improvement (25.1%) and is 12.8% better than ROD. The figure also shows that LPRQ-only has a

small performance degradation compared to the ROD in G1. This is because ROD (and BD) has additional 32 entries in its write queue (read-queue) for compensating for LPRQ's space overhead. As a result, it is possible for ROD to achieve better performance improvement in situations when additional write-queue entries provide greater benefit than the benefit due to prioritizing low priority reads. This appears to be the case for the G1 workload, although it is important to note that this is not common (other workloads do not show this behavior). Note, however, that even in G1, LPRQ+OFS outperforms ROD by 1.1%.

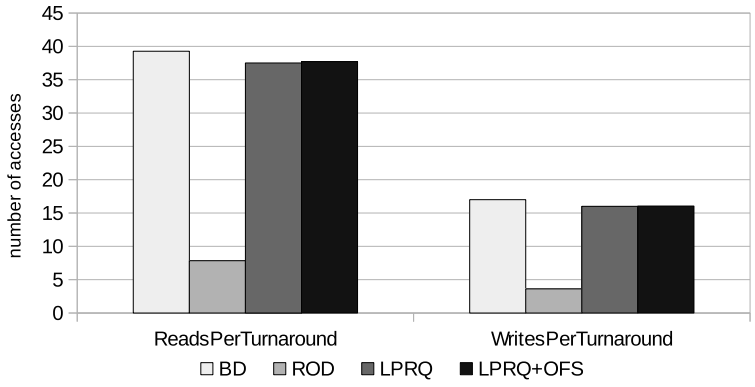

Figure 5.10: Read/Write accesses per turnaround (the higher the better) – 16-way

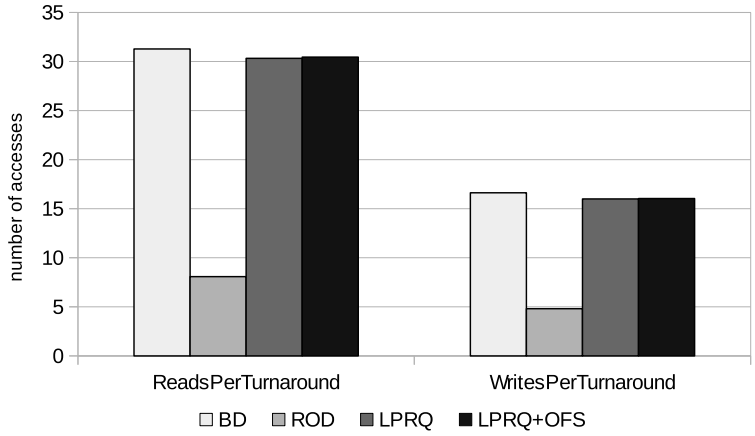

Figure 5.11: Read/Write accesses per turnaround (the higher the better) – direct-mapped

5.5.2 Turnarounds

As we discussed in §5.2.1, a DRAM bus can only operate in either read or write mode at a given point of time. An increase in the number of turnarounds will result in a

performance degradation which we want to minimize. In this experiment, we would like to analyze the number of accesses per turnaround for all designs. As we can see from Figure 5.10 (16-way) and Figure 5.11 (direct-mapped), because BD places all read accesses in the same read queue, the number of read/write accesses that it processes per turnaround is the best among all designs. On the other hand, ROD is only able to process about a third of read/write accesses per turnaround, as compared to BD. Finally, LPRQ is able to process almost the same number of read/write accesses as BD per turnaround. This is why LPRQ's (LPRQ+OFS) performance is better than ROD (Figure 5.8 and Figure 5.9).

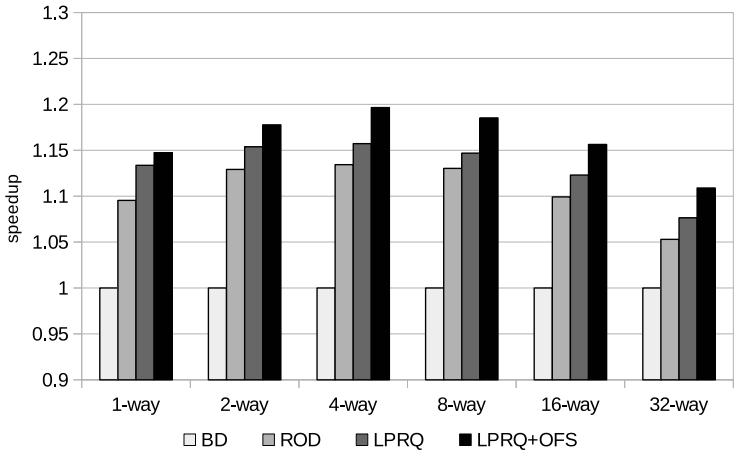

Figure 5.12: Average speedup in different associativity

5.5.3 Associativity

In §5.2.4, we discussed how the RWI issue could cause row conflicts. This problem will translate into a serious performance degradation if the access sequence shows spatial locality like the example shown in Figure 5.3. In our configuration, we use a $4KB$ row buffer which allows us to store 64 cache blocks in a row[6]. A 16-way cache would store 4 cache sets in a row (16 blocks per set * 4 cache set = 64). Therefore, it is necessary to perform a sensitivity study on the associativity of DRAM cache. In this section, we show and analyze how associativity affects the performance of each design.

Figure 5.12 shows the sensitivity study on associativity in 4-core workloads. we can see that ROD which can also avoid the RWI issue is actually performs well in low

[6]In our configuration, we assume we have additional row buffer space for storing tags. Therefore, the actual row buffer size is 4.25KB

associativity (1 or 2-way) cache designs. However, when the associativity gets larger, we can see the performance gap between BD and ROD is reduced to less than 5% (in a 32-way cache). On the other hand, LPRQ+OFS can provide over 10% improvement.

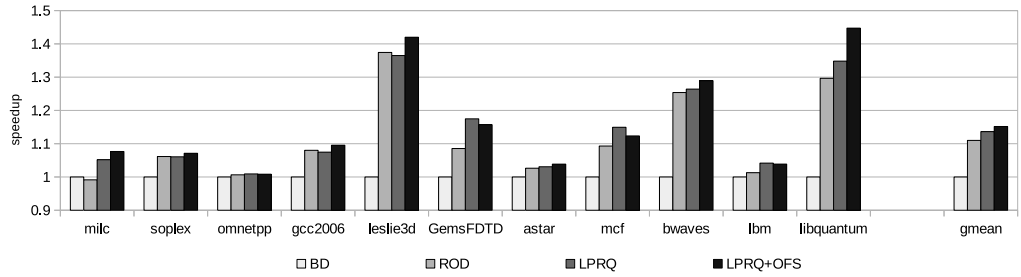

Figure 5.13: Performance speedup in single workload

5.5.4 Workload Analysis

Here we analyze how these designs will affect the performance in single workload. In this experiment, we reduce the number of stacked DRAM's channel to 1 and reduce L2 cache size to 4MB and use a 16-way setting as our default setting. As we can see from Figure 5.13 (results normalized to BD), ROD, LPRQ, and LPRQ+OFS show a very good speedup (over 29%) in libquantum and leslie3d. Based on the SPEC 2006's characteristics [6], we know leslie3D and libquantum have a large working set size which does not fit in L2 (but fits in the DRAM cache). In this case, the L2 suffers significant cache thrashing and creates a lot of cache replacement events. Therefore, RWI issue (§5.2.4) significantly affects BD's performance in these two benchmarks. On average, we can see LPRQ+OFS shows a 15.2% improvement whereas ROD has 11.1% improvement. The difference in performance improvement between them (4.1%) is reduced compared to multiprogrammed scenario where it was 5.7%. This is because some of the workloads do not have too many reads/writes and are less sensitive to the write queue servicing time. Furthermore, we can see LPRQ is actually performing better than LPRQ+OFS in mcf and GemsFDTD. As we can observe similar behavior from multiprogrammed workload results, because OFS reduces write queue servicing time by aggressively flushing low-priority reads, there is a possibility that these low-priority reads are blocking other priority reads. However, we can see the effect of this issue is not very significant and only causes a slowdown of 1% in mcf and GemsFDTD. On the other hand, for libquantum in which L2 cache is thrashing and sending a lot of

writebacks to DRAM cache, there is a 10% difference with and without OFS. Overall, we can see that the single core results are quite similar to multiprogrammed results.

soplex-mcf-gcc-libquantum-astar-omnetpp-astar-gcc
mcf-soplex-astar-leslie3d-bwaves-lbm-libquantum-leslie3d
bwaves-soplex-bwaves-GemsFDTD-libquantum-milc-leslie3d-astar
soplex-astar-lbm-mcf-lbm-omnetpp-leslie3d-bwaves
milc-leslie3d-omnetpp-gcc-milc-libquantum-bwaves-gcc
libquantum-lbm-soplex-libquantum-omnetpp-libquantum-mcf-gcc
soplex-astar-mcf-gcc-bwaves-astar-mcf-libquantum
lbm-libquantum-omnetpp-bwaves-gcc-milc-leslie3d-milc

Table 5.4: 8-core workload groupings

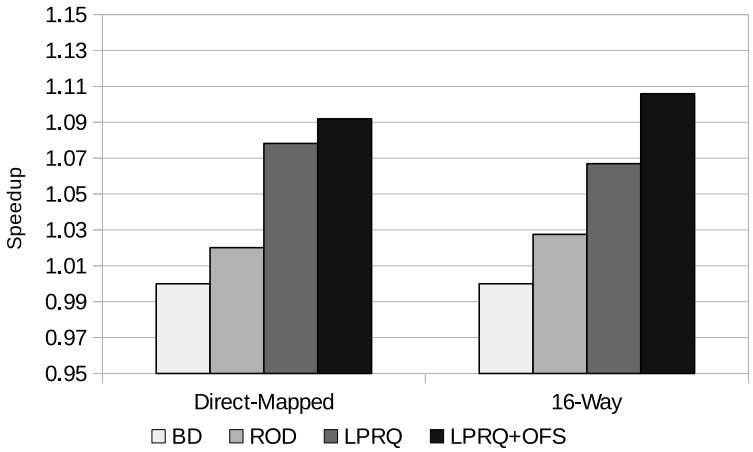

Figure 5.14: Performance speedup in 8-core workload

5.5.5 8-core Workloads

In this section, we analyze how our design will react to higher DRAM cache request frequency. We simulate this scenario via eight 8-core workloads as shown in Table 5.4 but keep the same setting as previously shown in Table 5.2. As we can see from Figure 5.14, in the 8-core workloads, the improvement of ROD reduce to 2.8% in 16-way and 2.0% in direct-mapped configuration. This is because high access pressure will reduce the benefits of buffering writes and also, the performance impact due to turnaround delays is increased. On the other hand, the DCA (LPRQ+OFS) can continue to provide improvement of 10.6% and 9.2% in direct-mapped and 16-way respectively. This because in addition to buffering writes, our DCA controller also optimize for turnarounds.

5.5.6 Summary

Based on our experiments, we show a comparison among our designs in Table 5.5[7]. In both single-core and 4-core results, ROD provides about 10% improvement in low associativity but the improvement drops to less than 5% in 32-way. Also, in high access pressure (8-core) workloads, LPRQ+OFS can continue to improve 9 to 10% when ROD can only provide 2 to 3% improvement.

	BD	ROD	LPRQ	LPRQ+OFS
8-core	*	**	**	***
Turnaround	****	*	****	****
Low assoc.	*	***	****	****
High assoc.	*	**	**	***

Table 5.5: Techniques comparisons

5.6 Related Work

There has been a number of recent works on die-stacked DRAMs. Prior works have shown that this technology is a promising step in the direction of bridging the latency of on-chip cache and off-chip memory. Some of them proposed using this die-stack DRAM as part of a main memory [55, 56, 57, 58] and many of them proposed to use it as a cache [10, 11, 14, 15, 3, 4, 9].

As we discussed in previous chapter, where to store cache tags is a non-trivial question in DRAM cache and has attracted significant attention [10, 11, 3, 4, 9, 52]. Unlike prior works, which are focused on the tag problem. In this work, we target a different problem pertaining to DRAM caches, one that has not been explored previously. More specifically, we address the problem of how to effectively schedule DRAM cache's requests. Although our work is based on the set-up used by Loh's organization [3] and Qureshi's organization [4], our work is orthogonal and can be applied to most of existing tags-in-DRAM designs. Among the recent proposed works, Chou et al. [52] have proposed a bandwidth-efficient DRAM cache design, which introduces a number of techniques to reduce the number of DRAM cache accesses for the direct-mapped DRAM cache. However, our work is still applicable for effectively scheduling the residual accesses.

[7] Higher the number of stars (*), better the design.

Zhang et al. [53] observed that L2 writebacks can cause interference with DRAM reads, which in turn could result in row conflicts in the DRAM (RWI in DRAM). Instead of using a write-buffer-based approach, they propose a permutation-based remapping scheme for mitigating the RWI issue. Based on Rau's work [59], the permutation-based scheme generate a new addressing address by XOR the *original bank index* with *k-bit* (first N-bit of page index). Ideally, this scheme can reduce row conflicts by remapping blocks in the same cache set to different memory banks. It is worth noting that remapping will not preclude the use of the write-buffer in the DRAM controller, and is therefore orthogonal to our technique. In other words, the remapping scheme can still apply to our DCA controller to further reduce the RWI problem.

5.7 Conclusion

Recent studies have proposed DRAM cache designs that maintain tags in the DRAM cache, which increases the complexity of a DRAM cache access. In this work, we addressed the problem of how to effectively schedule these DRAM cache accesses. A conventional DRAM controller only classifies accesses into two categories — read and write. This simple two way classification is not suitable for DRAM cache because different read accesses can have different priorities depending on the requests they correspond to.

In this work, we study two potential designs based on conventional DRAM controller and analyze their limitations. We then propose a DRAM-Cache-Aware (DCA) DRAM controller which uses an additional low-priority read queue. We also propose an opportunistic flushing scheme (OFS) for this low-priority read queue. In our study, we found that in a 16-way DRAM cache, LPRQ+OFS improves 15.6% compared to BD, which is a naive design derived from conventional DRAM controller. We also study the direct-mapped design, which allows tag and data to be read in one single DRAM burst. Our experiment shows that LPRQ+OFS achieves 14.7% improvement compared to BD.

Chapter 6

Summary of Contributions and Future Work

6.1 Summary of Contributions

In this thesis, we have proposed a number of techniques for enhancing the utilization of SRAM and DRAM caches. In the first part of this thesis, we proposed critical-words-only cache (co-cache) which allows us to improve the effective cache capacity of an L2 cache by only storing critical-words. Our design is based on the observation that for every L2 cache block, a subset of words (the critical words) are accessed sooner than the others. In contrast to a conventional L2, a co-cache only caches the critical words for each cache block. Our experimental results provide evidence to support the hypothesis that a co-cache is able to utilize the cache space more efficiently, especially in situations in which the cache size is significantly less than the working-set size. However, in situations in which the cache size is larger than the working-set size, a conventional cache could perform better. For this reason, we also proposed adaptive co-cache (aco-cache) that can dynamically choose to behave like a co-cache or a conventional cache. In our experiments, a 256 kB L2 organized as an aco-cache performed as well as a 512 kB conventional L2 cache on average.

In the second part of this thesis, we studied the tag issue in the DRAM cache. The advent of the DRAM cache has posed a problem of how to efficiently manage the tags associated with the DRAM cache. One naive option is to store all the tags in SRAM; while this would ensure fast access of the tags, the associated storage cost would render this approach impractical. Consequently, prior works have proposed innovative techniques to manage the tags efficiently in DRAM. Nonetheless, we observe, with a

help of a study, that it is more performance efficient to manage the tags in SRAM. We proposed aggressive tag caching so that we can achieve the effect of maintaining all tags in SRAM, without paying the prohibitive cost; we show that there is enough spatial and temporal locality amongst DRAM cache tag accesses to merit caching/prefetch the tags. In the result, we can see that the ATCache achieves 10.3% speedup on average across 11 memory-intensive workloads.

While prior works proposed a tags-in-DRAM design, a study on how to schedule DRAM cache requests is missing. In the last part of the thesis, we addressed the problem of how to effectively schedule these DRAM cache accesses. A conventional DRAM controller only classifies accesses into two categories — read and write. This simple two way classification is not suitable for DRAM cache because different read access can have different priorities depending on the request they correspond to. In this work, we study two potential naive designs based on conventional DRAM controller and analyze their limitations. We then propose a DRAM-Cache-Aware (DCA) DRAM controller that uses an additional low-priority read queue. We also propose an opportunistic flushing scheme (OFS) for this low-priority read queue. In our study, we found that in a 16-way DRAM cache, LPRQ+OFS improves 15.6% compared to BD, which is a naive design derived from conventional DRAM controller. We also study the direct-mapped design, which allows tag and data to be read in one single DRAM burst. Our experiment shows that LPRQ+OFS achieves 14.7% improvement compared to the baseline.

6.2 Future Work

Enhanced adaptive co-cache design. In the co-cache work, we proposed a simple adaptive scheme to switch between co-cache and conventional L2 cache based on the miss rate of L2. Prior works [60, 61] indicate that the accesses among all cache set are not uniformly distributed. To exploit this observation, instead of switching the whole L2 between conventional cache and co-cache, a reconfiguration scheme based on the cache sets' miss rate seems promising. Also, our proposed design require additional tags space (about 13% of L2 size) due to the increased number of cache sets. This SRAM overhead is acceptable as the L2 cache is normally four to eight times smaller than L3 cache. However, this overhead could restrict our application if we want to apply co-cache idea to a larger cache such as L3 cache. Instead of provide co-cache

support to all cache sets (i.e. all cache sets will have additional tags/wid), partial support could be an interesting direction for further study.

Bandwidth-aware tag caching. Our study in §4.3.2 shows that most of tag cache hits are coming from prefetching. Although our ATCache can satisfy 60% tag requests, we observed that most tag hits are coming from prefetched tags. If we remove all the prefetching mechanisms that we used in ATCache (i.e. PG/1 and no hit-prefetching), only about 4% tag requests are serviced in the ATCache. However, from our study, we found these prefetching mechanisms result in a 12% to 36% additional tag accesses compared to a tags-in-DRAM approach. In ATCache work, we try to minimize the SRAM requirement and gain the maximum performance. However, using tag caching to minimizing DRAM cache's tag accesses is also a promising direction.

Bibliography

[1] C. Huang and V. Nagarajan, "Increasing cache capacity via critical-words-only cache," in *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*, pp. 125–132, 2014.

[2] C. Huang and V. Nagarajan, "Atcache: reducing DRAM cache latency via a small SRAM tag cache," in *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pp. 51–60, 2014.

[3] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *MICRO*, pp. 454–464, 2011.

[4] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *MICRO*, pp. 235–246, 2012.

[5] Intel, "Desktop 4th generation intel core processor family, desktop intel pentium processor family, and desktop intel celeron processor family," 2012.

[6] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation."

[7] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, pp. 37–48, 2012.

[8] L. Zhao, R. R. Iyer, R. Illikkal, and D. Newell, "Exploring dram cache architectures for cmp server platforms," in *ICCD*, pp. 55–62, 2007.

[9] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO*, pp. 247–257, 2012.

[10] N. Gulur, G. R., R. Manikantan, and M. Mehendale, "Bi-modal dram cache: Improving hit rate, hit latency and bandwidth," in *MICRO*, 2014.

[11] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *MICRO*, 2014.

[12] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to model large caches," 2009.

[13] G. H. Loh and M. D. Hill, "Supporting very large dram caches with compound-access scheduling and missmap," *IEEE Micro*, vol. 32, no. 3, pp. 70–78, 2012.

[14] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA*, pp. 404–415, 2013.

[15] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *HPCA*, pp. 1–12, 2010.

[16] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pp. 439–450, 1994.

[17] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[19] A. Jaleel, E. Borch, M. Bhandaru, S. C. S. Jr., and J. S. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pp. 151–162, 2010.

[20] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ISCA*, pp. 60–71, 2010.

[21] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. S. Jr., and J. S. Emer, "CRUISE: cache replacement and utility-aware scheduling," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pp. 249–260, 2012.

[22] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*, pp. 364–373, 1990.

[23] J. Baer and T. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pp. 176–186, 1991.

[24] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[25] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: an adaptive data cache prefetcher," in *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pp. 135–145, 2004.

[26] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, 2005.

[27] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pp. 69–80, 2009.

[28] Micron, "8gb: x4, x8 1.5v twindie ddr3 sdram."

[29] S. Kumar and C. B. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *ISCA*, pp. 357–368, 1998.

[30] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pp. 376–388, 2012.

[31] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *HPCA*, pp. 250–259, 2007.

[32] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *HPCA*, pp. 276–287, 2004.

[33] P. Pujara and A. Aggarwal, "Increasing the cache efficiency by eliminating noise," in *HPCA*, pp. 145–154, 2006.

[34] R. K, T. Warrier, and M. Mutyam, "Skipcache: miss-rate aware cache management," in *PACT*, pp. 481–482, 2012.

[35] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, pp. 423–432, 2006.

[36] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Co-operative partitioning: Energy-efficient cache partitioning for high-performance cmps," in *HPCA*, pp. 311–322, 2012.

[37] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-v dd : A circuit technique to reduce leakage in deep-submicron cache memories," in *ISPLED*, pp. 90–95, 2000.

[38] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," in *HPCA*, pp. 147–157, 2001.

[39] S.-H. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *HPCA*, pp. 151–161, 2002.

[40] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO*, pp. 248–, 1999.

[41] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[42] E. J. Gieske, *Critical words cache memory: exploiting criticality within primary cache miss streams*. PhD thesis, Cincinnati, USA, 2008.

[43] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio," in *ISCA*, pp. 384–393, 1994.

[44] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Architecting efficient interconnects for large caches with cacti 6.0," *IEEE Micro*, vol. 28, no. 1, pp. 69–79, 2008.

[45] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi, "Predictor virtualization," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pp. 157–167, 2008.

[46] P. Michaud, "Demystifying multicore throughput metrics," *IEEE Computer Architecture Letters*, 2013.

[47] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, 2013.

[48] H. Wang, T. Sun, and Q. Yang, "Cat - caching address tags: A technique for reducing area cost of on-chip caches," in *ISCA*, pp. 381–390, 1995.

[49] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[50] W. Chou, Y. Nain, H. Wei, and C. Ma, "Caching tag for a large scale cache computer memory system," Sept. 22 1998. US Patent 5,813,031.

[51] T. Wicki, M. Kasinathan, and R. Hetherington, "Cache tag caching," Apr. 3 2001. US Patent 6,212,602.

[52] C. Chou, A. Jaleel, and M. K. Qureshi, "Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches," in *ISCA*, 2015.

[53] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, pp. 32–41, 2000.

[54] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pp. 475–486, 2013.

[55] C. Chou, A. Jaleel, and M. K. Quresh, "Cameo:a two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *MICRO*, 2014.

[56] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the processor-memory performance gapwith 3d IC technology," *IEEE Design & Test of Computers*, vol. 22, no. 6, pp. 556–564, 2005.

[57] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *MICRO*, 2014.

[58] D. H. Woo, N. H. Seong, D. L. Lewis, and H. S. Lee, "An optimized 3d-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, pp. 1–12, 2010.

[59] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*, pp. 74–83, 1991.

[60] F. Bodin and A. Seznec, "Skewed associativity enhances performance predictability," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, pp. 265–274, 1995.

[61] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pp. 187–198, 2010.