The Dissertation Committee for Ching-Pei Lin
certifies that this is the approved version of the following dissertation:

Mitigating Bank Conflicts in Main Memory via Selective Data Duplication and Migration

Committee:

Yale Patt, Supervisor

Derek Chiou

Mattan Erez

Emmett Witchel

Chris Wilkerson

Mitigating Bank Conflicts in Main Memory via Selective Data Duplication and Migration

by

Ching-Pei Lin

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2021

Acknowledgments

There are many people to thank.

First and foremost, my family. My father, who has been a bedrock in my life through all the ups and downs. My brother, with whom there are so many things that only he can understand. My stepmom, who has been nothing but loving and nurturing ever since she joined our family. My late mother and grandparents, who all shaped me into who I am today. All my aunts, uncles, and cousins, with whom I have shared so many joyous and sad moments.

Next, I wish to thank all HPS members with whom I overlapped.

Jose, for his encyclopedic knowledge of all things computer-related and always calm demeanor. Khubaib, for many great conversations, and a very memorable reunion at Derek's house. Veynu, for always beating me in squash, and for teaching me about GPUs. Carlos, for many social outings, squash matches, heartfelt advice, and an incredible lunch atop Google Manhattan. Rustam, for many dinners together, and for being the first to show me what research really entails.

Milad, for always being helpful, insightful and confident. Faruk, for your astute intuitions, and for countless priceless moments throughout the years.

Stephen, for daily conversations and explanations on just about every topic conceivable, and for teaching each other how to do research. Siavash, for co-inventing an entire genre of humor with me, for educating me on machine learning and C++, and for letting me vent to you. Ali, for putting up with my nonsense, and for being someone I can talk DRAM with within the group. Aniket, for taking part in the heroic push to get Scarab working with 64 bit executables right after joining the group. Chester, for taking on the mantle of 460N TAing guru.

Thank you to the other architecture students that I overlapped with: Nick, Haishan, Esha, Sangkug, Yongkee, Majid, Wenqi, Austin, Chirag, Dan, Tian, Alex, Sabine, Mochamad, Kishore, and Kamyar. I always felt a special fellowship among us.

Thank you to the many amazing friends I have met since coming to Austin. There are too many to name, but I will try: Bill, Sharon, Joe, Janet, Vincent, Wei-Jin, Hudson, Joanne, Vera, Clemens, Dara, Dominique, Leah, Yina, Weiyi, Henry, Fiona, Ben, Hope, and Jordy. Thank you for many wonderful moments.

Thank you to all my friends from Canada for still keeping in touch: Fraser, Mark, Derek, Tereza, Robyn, Julia, Josh, Jonny, Will, Leanne, Allison, Ken, Michael, Ivan, Yatin, Heidi, Yifan, Marguerite, James, Aric, and Matt. Special thanks to the Thurns, the Hills, the Browns, the deBruyns, and the Tardibuonos for always treating me like family. And a special note to Yatin: who would have thought back in 2006 that we would both end up doing PhDs in computer architecture? Thank you for being such a great friend all these years, and I look forward to visiting you in Ann Arbor!

Thank you to Rob Chappell, Philip Emma, Hillery Hunter, Doug Carmean, and Mike Shebanow for making my summer internships happen. I learned so much on those internships.

Thank you to all the members of my committee, whose feedback helped strengthen this dissertation. I especially wish to thank Mattan and Derek for helping out in the aftermath of Winter Storm Uri.

Finally, none of this would have been possible without the support and encouragement of my advisor, Dr. Yale Patt. Thank you for teaching me the importance of sound fundamentals and effective communication, and for being incredibly patient with me the entire PhD.

Mitigating Bank Conflicts in Main Memory via Selective Data Duplication and Migration

by

Ching-Pei Lin, Ph.D.
The University of Texas at Austin, 2021

SUPERVISOR: Yale Patt

Main memory is organized as a hierarchy of banks, rows, and columns. Only data from a single row can be accessed from each bank at any given time. Switching between different rows of the same bank requires serializing long latency operations to the bank. Consequently, memory performance suffers on bank conflicts when concurrent requests access different rows of the same bank.

Many prior solutions to the bank conflict problem required modifications to the memory device and/or the memory access protocol. Such modifications create hurdles for adoption due to the commodity nature of the memory business. Instead, I propose two new runtime solutions that work with unmodified memory devices and access protocols. The first, Duplicon Cache, duplicates select data to multiple banks, allowing duplicated data to be sourced from either the original bank or the alternate bank, whichever is more lightly loaded. The second, Continuous Row Compaction, identifies data that are frequently accessed together, then migrates them to non-conflicting rows across different banks.

To limit the data transfer overhead from data duplication and migration, only select data are duplicated/migrated. The key is to identify large working sets of the running applications that remain stable over very long time intervals, and slowly duplicate/migrate them over time, amortizing the cost of duplication/migration. In effect, the set of duplicated/migrated data form a cache within main memory that captures large stable working sets of the application.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Main memory performance remains the principal bottleneck for many important applications. Main memory is organized as a hierarchy of banks, rows, and columns. Only data from a single row can be accessed from each bank at any given time. Switching between different rows of the same bank requires serializing long latency operations to that bank. Consequently, memory performance suffers on bank conflicts when concurrent requests access different rows of the same bank.

1.2 What Has Been Done

Prior work addressed the bank conflict problem in four ways:

- by changing the memory device itself to reduce the latencies associated with switching rows of a bank. However, such modifications create hurdles for adoption because they increase the cost of the device, and/or require changes to the memory access protocol that need to be agreed upon by all memory and processor manufacturers.

- by increasing the effective number of banks. Straight up increasing the number of banks gives the most benefit, but is also the most expensive option. Other techniques subdivide existing channels and banks into smaller independent modules (e.g., subranks and subarrays) to mimic the effect of having more banks. However, such techniques still require modifications to the memory device and/or memory access protocol, making adoption more difficult.

- by partitioning memory banks among different co-running applications to reduce bank conflicts caused by interference between different applications. However, scarcity in memory banks fundamentally limits the effectiveness of such partitioning.

- by optimizing the scheduling of memory requests to maximize utility from rows currently active in each bank. Such techniques need to strike a balance prioritizing memory requests to currently active rows in each bank while ensuring requests that conflict with the currently active rows still get serviced in a timely manner.

1.3 Contribution

The contribution of this thesis is two new runtime microarchitecture-only solutions to the bank conflict problem that are fully compatible with unmodified:

- Software (including OS, runtime, applications)

- Memory devices
- Memory access protocols

The first solution, Duplicon Cache, identifies frequently accessed and latency critical data that suffer from bank conflicts, then duplicates them to an alternate bank. Duplicated data can later be sourced early from its alternate bank when the original bank is currently busy, and vice versa. This substantially cuts down on memory request latency for requests that previously needed to wait for conflicts to resolve at the original bank.

The second, Continuous Row Compaction, identifies temporally correlated data that are frequently accessed together, then migrates them to a set of non-conflicting rows across different channels/ranks/banks. The migrated data can then later be accessed without bank conflicts.

All data duplications and migrations are managed at the memory controller, completely transparent to software, and performed with unmodified memory devices via existing memory access protocols.

In effect, the set of duplicated/migrated data form a cache within main memory. As this cache is resident in main memory itself, it can have considerably higher capacity than the conventional on-chip SRAM caches. Many of the challenges associated with Duplicon Cache and Continuous Row Compaction are in essence challenges associated with managing a large capacity cache in main memory. The two biggest challenges are:

(1): How do we minimize the data duplication/migration overhead associated with filling this cache?

(2): How do we efficiently track what has been cached (i.e., duplicated/migrated), given the large size of the cache? That is, how do we maintain an efficient tag store?

To minimize the data duplication/migration overhead, we take advantage of the enlarged capacity of this main memory cache to capture large working sets that are stable over very long time intervals, enabling one to achieve good reuse while replacing cache content very infrequently.

Efficient tag stores are maintained by using large line sizes and cache sectoring when appropriate. For Duplicon Cache, the tag store tracks what has been duplicated at 8KB granularity, but uses sectoring to allow individual 64B pieces within each 8KB line to be duplicated independently. For Continuous Row Compaction, the tag store tracks what has been migrated at 4KB granularity.

Both Duplicon Cache and Continuous Row Compaction have been evaluated on DDR4 SDRAM. They can be applied equally to other main memory technologies organized into banks and rows, such as non-volatile random-access memory (NVRAM), as well as other future memory technologies. I leave this as future work.

1.4 Thesis Statement

Memory bank conflicts can be mitigated without modifying the memory device or access protocol by transparently duplicating frequently accessed and latency critical data to multiple banks, and/or migrating temporally correlated data that are frequently accessed together to non-conflicting rows across multiple banks.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 goes over background information on DRAM and motivates how data duplication and migration help mitigate bank conflicts. It also describes how prior work, in contrast, needs to modify the DRAM device itself in order to mitigate bank conflicts, and why this is undesirable. Chapter 3 discusses managing the overhead of extra data movement associated with data duplication/migration and tracking what has been duplicated/migrated. Chapter 4 presents and evaluates the Duplicon Cache. Chapter 5 presents and evaluates Continuous Row Compaction. Chapter 6 discusses related work. Chapter 7 presents conclusions and future work.

Chapter 2

Background and Motivation

This chapter goes over basic DRAM organization and operations, explains what bank conflicts are, why they are detrimental to performance, and motivates how data duplication across different banks (Duplicon Cache) and data migration to non-conflicting rows across different banks (Continuous Row Compaction) help mitigate bank conflicts without requiring any modifications to the DRAM device or access protocol. This chapter also describes how prior work, in contrast, modified the DRAM device in order to mitigate bank conflicts, and why this is undesirable.

The discussion is specific to DDR4 SDRAM memory, as it is the current predominant main memory technology on which the thesis evaluations are based. However, bank conflicts are also problematic for other memory technologies like non-volatile random-access memory (NVRAM), and in general the proposed solutions are sufficiently generic that they can be applied to any memory technology where bank conflicts are problematic.

2.1 DRAM Organization and Operations

DDR4 SDRAM is organized as a hierarchy of channels, ranks, bank groups, banks, and columns. Figure 2.1 shows an example of this hierarchy with two channels and two ranks per channel.

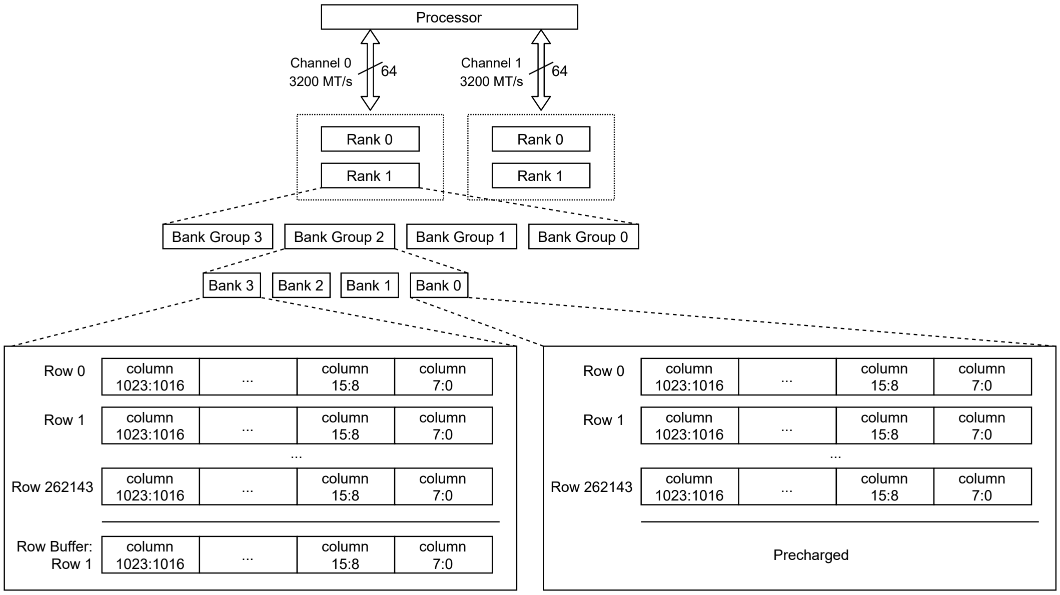


Figure 2.1: DRAM hierarchical organization.

2.1.1 Channel Bandwidth

The top level entity in the hierarchy is the channel. Each channel encompasses a subset of memory, along with the physical connections (pins,wires) that connect the subset of memory to the processor. Figure 2.1 shows two channels, labelled channels 0 and 1. Each channel is 64 bit wide. In this example, we assume each channel can transfer data at a rate of 3200 megatransfers per second (MT/s), which gives a maximum bandwidth of 23.8 GB/s.[1] The maximum aggregate bandwidth across both channels is thus 47.7 GB/s. This aggregate bandwidth is an upper limit on the maximum memory bandwidth available to the processor.

This upper limit can be increased by increasing:

- the number of channels
- the width of each channel
- the data rate of each channel

However, doing so comes at a cost. Increasing the data rate of each channel, as is done for graphics GDDR memory, increases the power consumption. Increasing the number of channels or the width of each channel increases the total number of physical connections between the processor and the memory, increasing the manufacturing cost. This is the most common differentiator between cost-optimized memory technology such as DDR4 that have fewer connections to the processor, and performance-optimized memory technology such as High Bandwidth Memory (HBM) that have much more connections to the processor. To support the higher number of connections between memory and processor, performance-optimized memory like HBM need to be connected to the processor using more expensive technology like

[1] 3200000000 transfers/s * 8B/transfer = 23.8 GB/s

silicon interposers, while cost-optimized memory can use cheaper technology like printed circuit boards [2].

The aggregate channel bandwidth is an upper bound on the maximum memory bandwidth available to the processor. Yet oftentimes, due to bottlenecks in other levels of the DRAM hierarchical organization, the available channel bandwidth is underutilized, even for cost-optimized memory that have lower available channel bandwidth to begin with. The biggest of these bottlenecks are bank conflicts, described in the following sections.

2.1.2 Rank, Bank Group, and Bank Level Parallelism

After the channel, the next levels in the hierarchical organization are ranks, bank groups, and banks.

Banks are individual sub-modules of memory that can each independently process memory requests. Every memory request is ultimately processed by a particular bank. However, the latency for accessing data at each bank varies depending on the state of the bank, and can be quite long. In general, the rate at which requests can be processed at a single bank is much lower than the rate needed to saturate the available channel bandwidth. Thus it is necessary to interleave requests to different banks and overlap latencies from multiple requests in order to achieve good utilization of the channel.

Consequently, it is advantageous to have as many banks as possible in order to maximize the level of interleaving. However, additional banks incur additional hardware costs. Bank groups were introduced with DDR4 to allow for additional banks while keeping hardware costs low. In essence, a bank group is a group of banks that share some hardware resources. The shared resources then only need to be replicated once per bank group, rather than once per bank. With bank groups, DDR4 was able to double the number of banks per device from 8 to 16 from DDR3, organized into four bank groups of four banks each. However, because banks of the same bank group share certain hardware resources, memory requests to the same bank group now incur additional delay, even if they are to different banks.

Finally, a rank is a group of memory devices that operate in lockstep in response to the same memory commands. Ranks are usually added to increase the memory capacity per channel, although additional ranks also increase the total number of bank groups/banks, improving performance. Figure 2.1 shows each channel is attached to two ranks, labelled ranks 0 and 1, for a total of four ranks in the system. Each rank is in turn made up of four bank groups, and each bank group made up of four banks. In total, there are 16 banks per rank, and 32 banks per channel in the example in Figure 2.1.

Each of the 32 banks belonging to the same channel share the same channel data pins. At any given time, only data from one of the 32 banks can be transferred over the channel, and interleaving is required to overlap the latencies from different requests in order to achieve good utilization of the channel. In particular, it is most advantageous to interleave requests across different bank groups. In fact, it is impossible to fully saturate the channel bandwidth without interleaving requests across different bank groups.

My first proposal, Duplicon Cache, leverages this property of memory by duplicating select data across bank groups, increasing the probability that interleaving can occur, thereby increasing performance.

2.1.3 Row Buffer Locality and Bank Conflicts

Data in each bank is organized into rows and columns. In Figure 2.1 there are 256K rows per bank, and 1K columns per row.

Although there are 1K columns per row, the DDR4 protocol specifies that columns must be accessed in aligned units of 8 or 4, with 8 being the most common configuration. On a read or write, the 8 columns in the aligned unit are read/written in 8 consecutive data bursts. Consequently, there are only 1K/8 = 128 addressable aligned units of data in each row.

Only data from a single row can be accessed at any given time from a bank. This is because data in DRAM is encoded via the presence of electric charge on capacitors, and before such data can be accessed, it first needs to be read out and amplified via a process called sense amplification. This involves two steps. The first is Precharge, which prepares the bank for sense amplification. The second is Activate, which reads out and amplifies an entire row's worth of data.[2] Once a row has been activated at a bank, it stays activated until the next Precharge operation, and subsequent reads and writes to columns in that row can be processed with low latency. The row currently

[2] the Activate operation is also interchangeably referred to as sensing the row, activating the row, or opening the row

activated in the bank forms the row buffer (also called the DRAM page). Additional accesses to the row buffer are called row buffer hits, and the tendency for future accesses to hit in the row buffer is called row buffer locality.

Figure 2.1 shows that row 1 has already been activated in rank 1, bank group 2, bank 3, indicated by row 1 being shown again in the area below all the others rows. In contrast, rank 1, bank group 2, bank 0 is in the precharged state, where no rows are currently activated. In general, each bank either has a particular row activated, or is in the precharged state.

The Precharge the Activate operations themselves incur long latencies and are subject to various timing constraints, resulting from physical limitations of the memory device. From a performance point of view, the worst case scenario is a row or bank conflict when data being requested belong to different rows of the same bank.[3] In this case the accesses are serialized, as only data from a single row of the bank can be accessed at a time. Switching between the two rows requires a long latency Precharge operation, followed by another long latency Activate operation; the Precharge and Activate operations are completely serialized with no parallelism.

Duplicon Cache mitigates the effects of bank conflicts by duplicating the data to another bank group, allowing the Precharge and Activate latencies to be overlapped to different bank groups.

[3] I use the terms row conflict, row buffer conflict, and bank conflict interchangeably in the rest of the thesis

Continuous Row Compaction reduces the number of Precharge and Activate operations needed by migrating data frequently accessed together to the same row address across different channels, ranks, bank groups, and banks. For example, if data items A, B, C,..., Z were frequently accessed together, Continuous Row Compaction would migrate (i.e., compact) them to regions in memory that share the same row address (e.g. row 262143) across all channels, ranks, bank groups, and banks. Future accesses to A, B, C,..., Z are then guaranteed to not cause row/bank conflicts, and will be interleaved across all channels, ranks, bank groups, and banks.

2.1.4 Memory Timing

Long serialized Precharge and Activate latencies during bank conflicts substantially hurt memory performance. The exact latencies required for Precharge and Activate operations depends on the type, order, and location of preceding memory operations.

There are four main DRAM operations: Precharge, Activate, Read, and Write. The Precharge operation prepares the bank for sense amplification, and the Activate operation activates (i.e., perform sense implication) on the desired row. Once the row with the data has been activated, Read/Write operations read/write the data at the column specified. Timing constraints, imposed due to physical limitations of the memory device, limit when Precharge, Activate, Read, or Write operations can be issued. Table 2.1 shows some relevant timing constraints that affect memory performance.

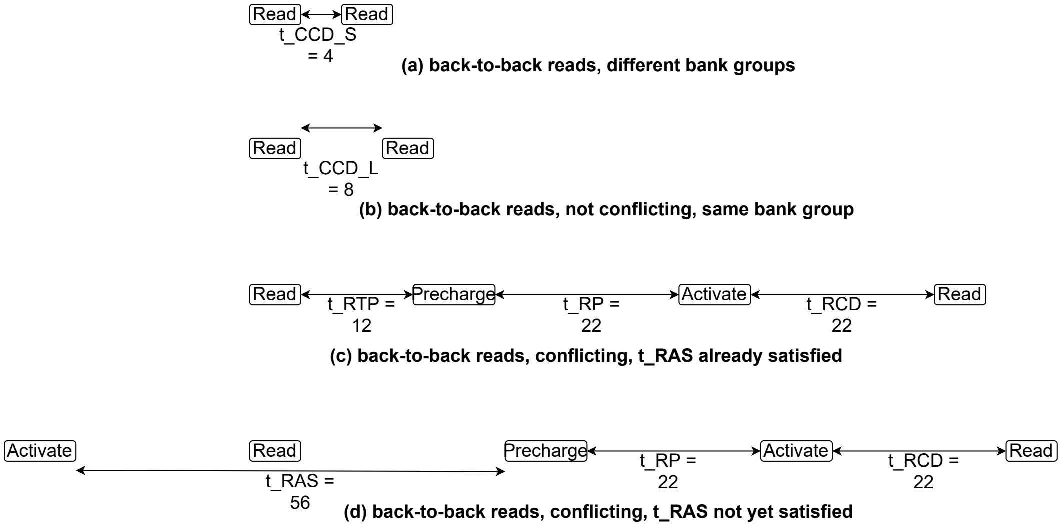
Table 2.1: DRAM timing constraints for DDR4-3200 22-22-22 memory.

Constraint	Description	Delay (cycles)
t_{CCD_S}	delay between consecutive reads (or consecutive writes) to different bank groups of the same rank	4
t_{CCD_L}	delay between consecutive reads (or consecutive writes) to the same bank group of the same rank	8
t_{RTP}	delay between read and subsequent precharge to the bank	12
t_{RAS}	delay between activate and subsequent precharge to the bank	56
t_{RP}	delay between precharge and subsequent activate to the bank	22
t_{RCD}	delay between activate and subsequent read or write to the bank	22

We now consider how the timing for two back-to-back reads to the same device (i.e., rank) changes under various conditions. We consider reads for simplicity, but the analysis is equally applicable for back-to-back writes. However, mixing of reads and writes involves other timing constraints that I have omitted for brevity.

If the reads are not conflicting (i.e., they are to different banks, or they access the same row in the same bank), then t_{CCD} is the limiting timing constraint. t_{CCD} specifies the minimum delay required between back-to-back reads to the same rank, and comes in two variants. If the reads are to different bank groups, then the short variant, t_{CCD_S}, which is 4 cycles, applies. As the actual data transfer over the channel takes 4 cycles itself, being able to issue a read every 4 cycles is sufficient to saturate the channel bandwidth. The timing for this scenario is shown in Figure 2.2(a).

If the back-to-back reads are not conflicting but to the same bank group,



(a) back-to-back reads, different bank groups

(b) back-to-back reads, not conflicting, same bank group

(c) back-to-back reads, conflicting, t_RAS already satisfied

(d) back-to-back reads, conflicting, t_RAS not yet satisfied

Figure 2.2: Back-to-back read latency for row hits and conflicts.

then long variant of t_{CCD}, t_{CCD_L}, which is 8 cycles, applies.[4] This means that even if all accesses are row buffer hits, if they all access the same bank group (i.e., no bank group interleaving), then one can at most achieve 50% channel utilization (4 cycles to transfer data for each request, but only able to issue read operations once every 8 cycles). This shows the importance of having high levels of bank group interleaving. The timing for back-to-back reads to the same bank group is shown in Figure 2.2(b).

Note that it is not necessary, in the two scenario described above, for both reads to be row buffer hits at their respective banks. The reads may instead themselves require Precharge and/or Activate at their respective banks,

[4] the longer delay is required because, as explained in section 2.1.2, banks of the same bank group share certain hardware resources

but because the Precharges/Activates are to different banks, their latencies can be overlapped.

In contrast, for back-to-back conflicting reads (i.e., reads that access different rows of the same bank), the Precharge and Activate latencies become serialized and exposed. After the first read is issued, a Precharge is required to prepare the bank for sense amplification, followed by an Activate to activate the row for the second read, followed by the second read itself. The timing for the Precharge depends on how long ago the previous Activate (i.e., the Activate that activated the row for the first read) was issued. If the previous Activate was issued sufficiently long ago, then t_{RTP}, which specifies the minimum delay required between a read and subsequent Precharge to the same bank (12 cycles), determines how soon the Precharge can be issued. This case is shown in Figure 2.2(c). On the other hand, if the previous Activate was issued recently, then t_{RAS}, which specifies the minimum delay between an Activate and subsequent Precharge to the same bank (56 cycles), determines how soon the Precharge can be issued. This second case is shown in Figure 2.2(d).

After the Precharge, the remaining timing are the same for both cases (c) and (d). The Activate for the row of the second read can be issued t_{RP} (22) cycles after the Precharge, and then the second read can be issued t_{RCD} (22) cycles after the Activate.

For the case in Figure 2.2(c), the latency between two reads ends up being $12 + 22 + 22 = 56$ cycles. If all memory accesses were like this, then we could only achieve $4/56 = 7.1\%$ channel utilization, as we would only be able to issue read requests once every 56 cycles, which would occupy the channel for 4 cycles, while the other 52 cycles remain idle. For the case in Figure 2.2(d), the performance limiter would be the latency between the two Activates, which is $56 + 22 = 78$ cycles. If all memory accesses were like this, then we could only achieve $4/78 = 5.1\%$ channel utilization.

The takeaway from this example is that conflicting accesses to different rows of the same bank is particularly harmful to memory performance, as Precharge and Activate latencies, which are significant, become fully exposed. Memory controllers try to minimize such conflicting accesses through better memory scheduling and physical-to-DRAM addressing mapping. I discuss these mechanisms below, and motivate how my proposals, Duplicon Cache and Continuous Row Compaction, enhance these two existing mechanisms.

2.1.5 Memory Scheduling and Duplicon Cache Motivating Example

Memory controllers buffer multiple memory requests and reorders them to better exploit bank level parallelism and row buffer locality. First Ready - First Come First Serve (FR-FCFS) [52] is a common strategy for reordering memory requests. FR-FCFS prioritizes ready memory requests (i.e., requests whose timing constraints have been satisfied) over non-ready requests, and then prioritizes older requests over younger requests. Since row buffer hits have timing constraints that are much more easily satisfied than row buffer conflicts, FR-FCFS essentially prioritizes row buffer hits over other requests.

FR-FCFS tends to maximize the overall memory request throughput, but introduces unfairness into the system because row conflict requests end up waiting a long time while other row buffer hits are getting processed.

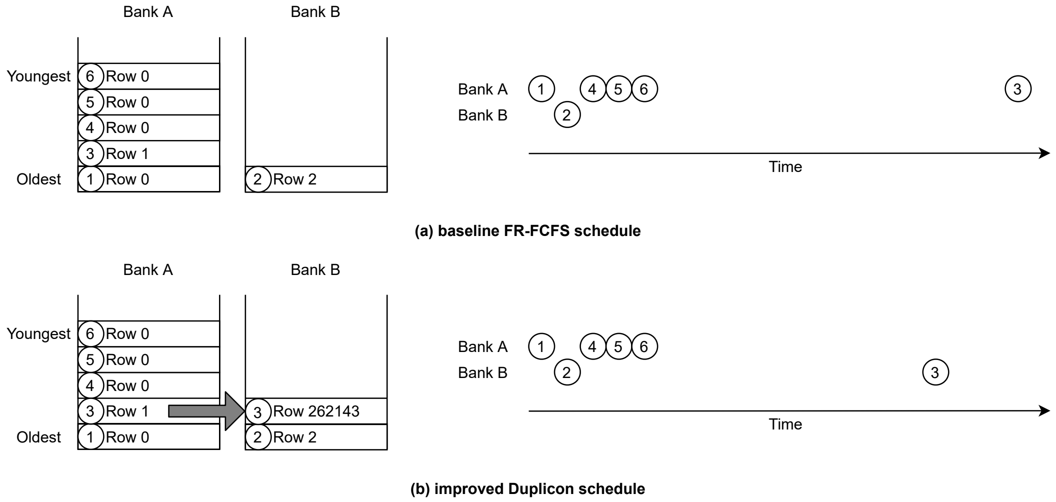


Figure 2.3: Baseline FR-FCFS vs. improved Duplicon Schedule.

Figure 2.3(a) shows how this can happen. We have six requests, ① - ⑥, from oldest to youngest. Requests ①, ③, ④, ⑤, and ⑥ are queued in bank A, while request ② is queued in bank B. Requests ①, ④, ⑤, and ⑥ access row 0 of bank A, while request ③ accesses row 1 of bank A. Since FR-FCFS favors row buffer hits over row conflicts, it will schedule ①, ④, ⑤, and ⑥ first, as all four can be serviced from (bank A, row 0). Request ③, which accesses (bank A, row 1), creates a bank conflict at bank A that requires a separate Precharge and Activate, and gets scheduled last, after ⑥.

Meanwhile, bank B happens to be lightly loaded. Request ② gets scheduled (due to its age) between ① and ④. After request ② is serviced, bank B becomes idle, and remains idle even while bank A is servicing requests ④, ⑤, ⑥, and ③.

One of the main insights behind Duplicon Cache is that these moments of load imbalance between banks can be exploited if data was duplicated to both banks. In Figure 2.3(b), we assume the data for request ③, in (bank A, row 1), was previously duplicated to (bank B, row 262143). This allows us to service request ③ from either bank A or B. Since bank B becomes idle in our example after servicing servicing request ②, we can now begin servicing of request ③ earlier from bank B, shortly after request ② has been serviced. Consequently, request ③ gets serviced earlier compared to the FR-FCFS baseline.

In this example bank B happened to be less loaded than bank A; in another instance, bank A may be less loaded than bank B. In either case, the duplicated data can be sourced earlier from the less loaded bank.

Duplicon Cache complements FR-FCFS nicely, because it alleviates the unfairness introduced by FR-FCFS. Normally, FR-FCFS introduces unfairness into the system because row conflict requests end up waiting a long time while other row buffer hits are getting processed. With Duplicon Cache, FR-FCFS can continue to prioritize row buffer hits to maximize overall request throughput, while Duplicon Cache identifies data that are suffering the most from row conflicts and enable them to be serviced earlier via duplication, thus reducing unfairness.

Note that Duplicon Cache does not actually reduce the number of row conflicts. In fact, it can increase the number row conflicts, since requests that were previously concentrated in the same row of the same bank now get diffused across multiple banks. But Duplicon Cache allows conflicting requests to be initiated earlier in another less loaded bank.

2.1.6 Physical-to-DRAM Address Mapping and Continuous Row Compaction Motivating Example

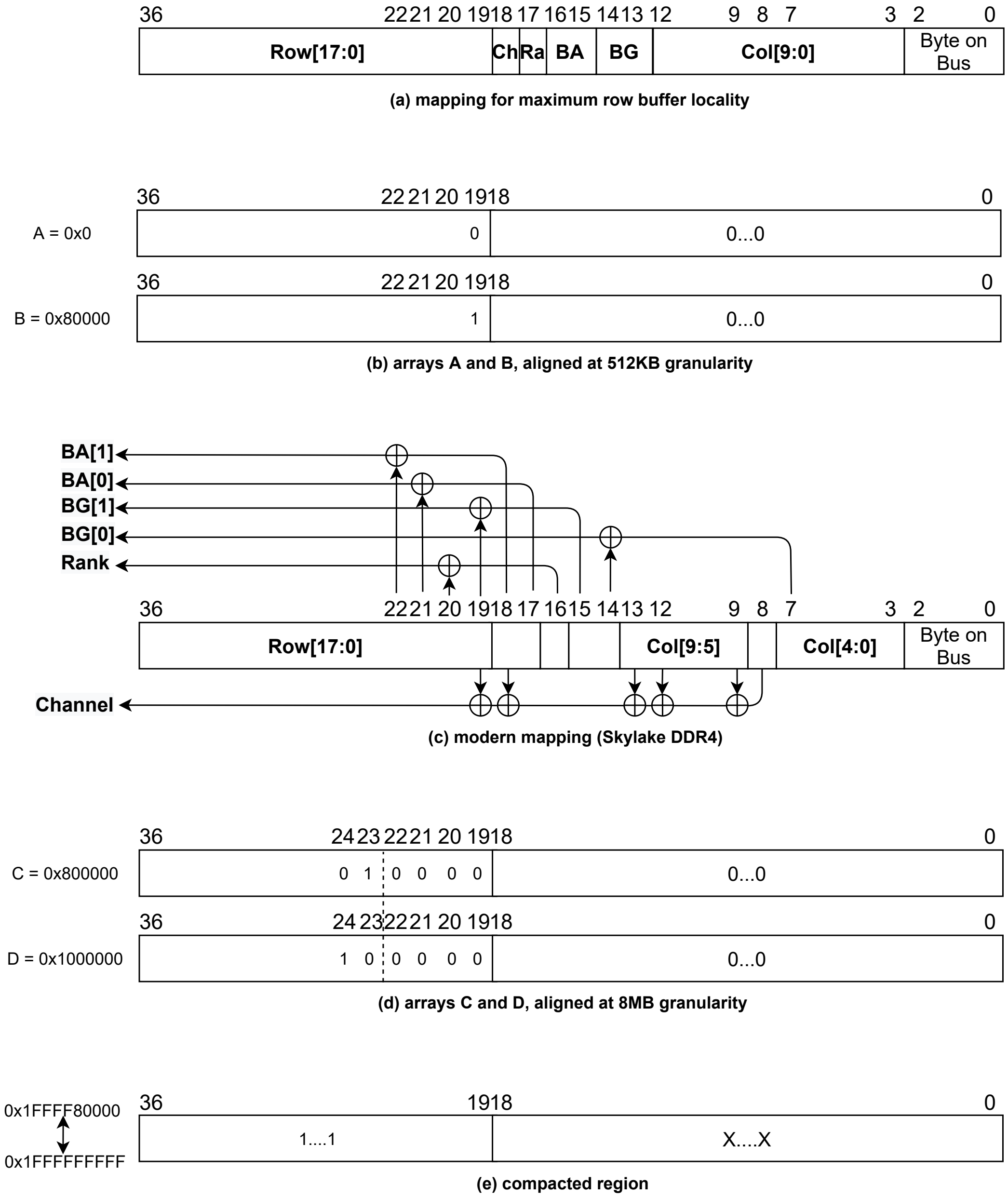
Bank conflicts arise when two concurrent requests share the same channel, rank, bank group, and bank address, while differing in the row address. The channel, rank, bank group, bank, row, and column addresses collectively form the DRAM address, and are determined from the physical address. In general, they are either taken directly from a subset of the physical address bits, or are computed as hashes (usually XOR) of various physical address bits. This mapping from physical address to DRAM address directly affects the frequency of row conflicts and memory performance.

In general, the lower order physical address bits toggle the most, while the higher order bits toggle the least. Thus taking the column address directly from the lowest order physical address bits produces the highest level of row buffer locality, as this maximizes the probability for DRAM addresses of concurrent accesses to only differ in the column address, resulting in row buffer hits to the same row of the same bank. Similarly, it is advantageous to place the row address bits as high as possible to minimize row address toggling, as two accesses with the same row address cannot conflict - they will either access the same row in the same bank, or access different banks.

Figure 2.4(a) shows such a mapping. As the DDR4 channel data width is 8 bytes, the lowest 3 bits of the physical address specify the byte on the channel. Next come the column address bits, which are taken from bits[12:3] of the physical address, followed by the bank group (denoted as BG), bank (BA), rank (Ra), channel (Ch), and Row bits.

This mapping maximizes the row buffer locality in each bank, but can actually be harmful to performance. To see this, suppose an application makes sequential accesses to two arrays, A and B, which are aligned to 512KB boundaries (i.e., the low 19 address bits are 0) at addresses 0x0 and 0x80000, respectively, as shown in Figure 2.4(b). The first problem with the mapping in Figure 2.4(a) is that, if A and B are sequentially accessed concurrently with the same incrementing index i and stride (for example, A and B are both accessed in a for loop with the same incrementing loop variable i), then every pair of accesses will be row conflicts, because the channel, rank, bank, and bank group address of A[i] and B[i], taken from bits [18:13] of the physical address, will always be the same for all indices i, while their row addresses (taken from bits 19 and above) will always be different.

The second problem with the mapping is that it provides poor channel interleaving and parallelism. Each pair of accesses to A[i] and B[i] will always go to the same channel, and as i increments, the accesses will stay in the same channel until the channel bit (bit 18) toggles. Assuming a stride of 64 bytes,



(a) mapping for maximum row buffer locality

(b) arrays A and B, aligned at 512KB granularity

(c) modern mapping (Skylake DDR4)

(d) arrays C and D, aligned at 8MB granularity

(e) compacted region

Figure 2.4: Physical-to-DRAM address mappings

this will only happen once every 4096 loop iterations.

The third problem is the mapping provides poor bank group interleaving, as A[i] and B[i] will always go to the same bank group, and the bank group bits (14 and 13) only toggle infrequently. Recall from Section 2.1.4 that bank group interleaving is advantageous because many DRAM timing constraints have more favorable (i.e., shorter delay) variants when the accesses are to different bank groups.

Figure 2.4(c) shows an improved state-of-the-art address mapping that fixes these issues [48]. The new mapping, in order to maximize row buffer locality, still takes the column address from the lowest order address bits, and the row address from the highest order address bits. However, the channel, rank, bank group, and bank addresses are now computed as XORs of several address bits, as opposed to being taken directly from the address [80]. For the channel address, the XOR-ed bits come from throughout the physical address, including one that is placed in the middle of the column bits (bit 8) [20], resulting in the channel bit toggling very frequently for both sequential and random accesses. This addresses the issue of poor channel interleaving. Likewise, one of the XOR-ed bits for the bank group comes from the low address bits (bit 7), ensuring the bank group address will also toggle frequently, providing bank group interleaving. The low 4 bits of the row address (bits [22:19]) are also now incorporated in the channel, rank, bank group, and bank computations via XORing. This fixes the problem of A[i] and B[i] always being row conflicts - since A and B differ in bit 19, and since bit 19 is now XORed as part of the channel and bank group computation, A[i] and B[i] will now access different bank groups in different channels.

However, the improved mapping does not fundamentally solve the problem of concurrent accesses with the same index and stride to aligned arrays resulting in row conflicts. If we replace A and B with arrays C and D, which are now aligned to 8MB boundaries at addresses 0x800000 and 0x1000000 respectively (shown in Figure 2.4(d), where the low 23 address bits all 0 for both C and D), then we still have the exact same issue as before, as the channel/rank/bank group/bank computations of the new mapping only consider bits 22 and below of the physical address, which would be the same between C[i] and D[i] for all indices i.

My second proposal, Continuous Row Compaction, does in fact fundamentally resolve this problem of concurrent accesses with the same index and stride to aligned arrays resulting in row conflicts. The fundamental idea behind Continuous Row Compaction is to migrate (i.e., compact) concurrently accessed data to regions in memory with the same row address. For example, suppose we reserved the row address 0x3FFFF, the largest row address possible, for exclusive use of the hardware, as shown in Figure 2.4(e). Since the row address comes from the highest order bits of the physical address, this essentially reserves the highest 512KB of the physical address space, comprised of addresses 0x1FFFF80000 through 0x1FFFFFFFFF, for the hardware (i.e., the software is made to think 0x1FFFF7FFFF is the last byte of the physical address space). The memory controller can then use this reserved region however it sees fit.

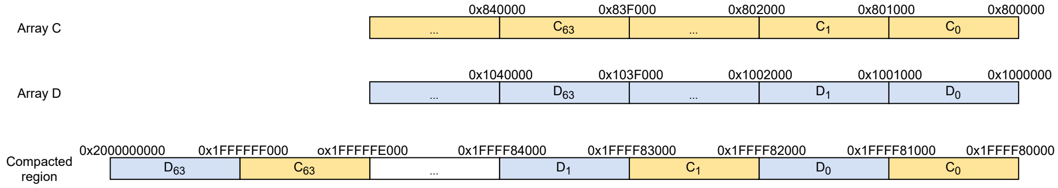


Figure 2.5: Compacting Arrays C and D into the Compacted Region

Continuous Row Compaction records the order in which 4KB memory regions are accessed by the application, and then migrates (i.e., compacts) those regions in the order they were accessed to a reserved row address. I show this in Figure 2.5. Suppose the application sequentially accesses the arrays C and D in the order C[0], D[0], C[1], D[1], C[2], D[2], and so on. Continuous Row Compaction would detect that the first 4KB region of array C (labelled C_0 in Figure 2.5, 0x800000 to 0x800FFF) was accessed first, followed by the first 4KB region of array D (D_0, 0x1000000 to 0x1000FFF), followed by the second 4KB region of array C (C_1, 0x801000 to 0x801FFF), then the second 4KB region of array D (D_1, 0x1001000 to 0x1001FFF), and so on. Once enough 4KB memory regions have been recorded to fill a reserved row address, then the recorded 4KB regions can be migrated, in the order they were recorded in, to the reserved row. In our example, since the reserved row 0x3FFFF spans 512KB between the addresses 0x1FFFF80000 and 0x1FFFFFFFFF, we would require 512KB/4KB = 128 different 4KB regions to fill the reserved row address. Consequently, we would migrate the first 64 4KB regions of array C, along with the first 64 4KB regions of array D, into the 512KB reserved region between 0x1FFFF80000 and 0x1FFFFFFFFF, with the individual 4KB regions of C and D interleaved in the order in which they were accessed. Thus, C_0, the first 4KB of array C, would be migrated to the first 4KB of the reserved region, 0x1FFFF80000 to 0x1FFFF80FFF; D_0, the first 4KB of array D, would be migrated to the second 4KB of the reserved region, 0x1FFFF81000 to 0x1FFFF81FFF; C_1, the second 4KB of array C, would be migrated to the third 4KB of the reserved region, 0x1FFFF82000 to 0x1FFFF82FFF; ....

Essentially, Continuous Row Compaction divides the arrays C and D into 4KB segments, then interleave the segments, in the order they were accessed, to the reserved region. Future accesses to the same 4KB segments can later be serviced from the reserved region without any row conflicts, as the interleaved 4KB segments all share the same row address and cannot conflict with one another.

In our example, the arrays being accessed shared the same alignment and were accessed with the same index and stride. This was chosen deliberately to emphasize the negative impact from row conflicts. However, in general any concurrent streaming accesses to arrays with different row addresses (i.e., different high order address bits) can result in row conflicts, even if they do not necessarily share the exact same alignment, index, or stride. In particular, multiple concurrent streaming accesses are very common in stencil computations (section 5.2). In these cases, Continuous Row Compaction is able to splice together 4KB segments from different streams into the same row address, removing row conflicts.

In addition, note that Continuous Row Compaction can give benefit even if the order in which 4KB segments were recorded and migrated differs from the exact order in which the segments are later accessed, as long as there is enough overlap between the set of 4KB segments that were migrated, and the set of 4KB segments that were later concurrently accessed.

2.1.7 The Alternative: Tweaking the DRAM Device

I have so far motivated how memory bank/row conflicts can be mitigated via data duplication (Duplicon Cache) and data migration (Continuous Row Compaction), all without changing the memory device or memory access protocol. In contrast, prior work advocated for tweaking the design of the DRAM device in order to reduce the latency of switching rows at a bank. In particular, they focused on reducing the latency of DRAM Precharge and Activate operations by reducing the effective capacitance of the shared piece of wire, called the bitline, which connects to individual DRAM storage cells.

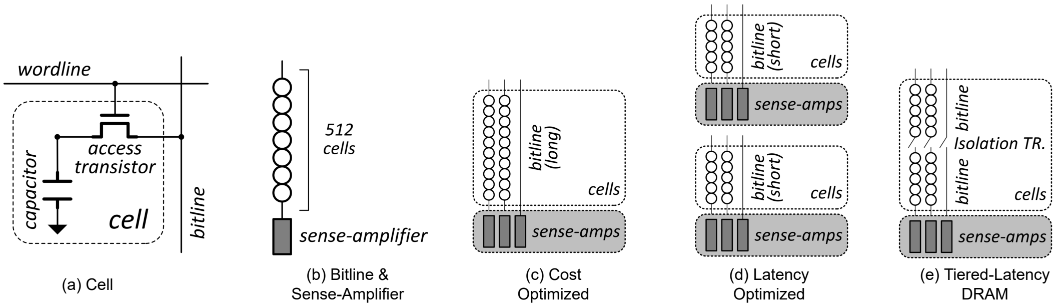


Figure 2.6: Tweaking the DRAM device to shorten the bitlines. (from [29])

The latencies for Precharge and Activate depend on the capacitance of the bitlines, which can be reduced in certain regions of the DRAM device via device level tweaks. This is shown in Figure 2.6. Figure 2.6(a) shows a DRAM cell, consisting of a capacitor connected to a bitline via an access transistor. A single bit is stored in the cell, via the presence (or lack) of charge on the capacitor. To access the bit, the bitline first needs to be brought (via the Precharge operation) to a reference voltage. Then the access transistor for the cell is turned on by raising the associated wordline (via the Activate operation), which connects the cell capacitor to the bitline. The presence (or lack) of charge on the capacitor will then slightly raise (or lower) the voltage on the bitline, producing a signal. However, this signal is very weak, and needs to be amplified by a sense-amplifier.

To increase the cell density, several cells share a single bitline and sense-amplifier. At any time, at most only one of the cells can connect to the shared bitline. Figure 2.6(b) shows 512 cells sharing a single bitline and sense-amplifier. This forms a single column of a two-dimensional matrix of storage cells, called a mat, which is shown in Figure 2.6(c).

Increasing the number of cells sharing a bitline increases the cell density and lowers the cost of the memory device. However, this also increases the bitline capacitance, which increases both the Precharge latency (latency to bring bitlines to the reference voltage, t_{RP} in Table 2.1) and Activate latency (latency to sense-amplify, t_{RCD} and t_{RAS} in Table 2.1). Cost-optimized DRAMs opt for more cells sharing the same bitline (e.g. 512 cells per bitline), like in the mat shown in Figure 2.6(c), while performance-optimized DRAM like Reduced Latency DRAM (RLDRAM) opt for fewer cells sharing the same bitline (Figure 2.6(d)).

Prior work proposed reducing the number of cells sharing a bitline in certain regions of the DRAM device. For example, Center High-Aspect-Ratio Mats (CHARM) [60] proposed replacing normal cost-optimized mats at the center of the chip with latency-optimized mats that have fewer cells per bitline, like those of Figure 2.6(d). Tier-Latency DRAM proposed adding isolation transistors that break the bitline into near and far segments (Figure 2.6(e)). When the isolation transistors are turned off, there are effectively fewer cells sharing the bitline in the near segment.

2.1.7.1 Drawbacks of Modifying DRAM

The main drawback with these proposals is that they require modifications to the DRAM device, which can increase the manufacturing cost and/or reduce the yield. For example, adding the isolation transistors for Tiered-Latency DRAM can be challenging because the process technology used for DRAM is specifically optimized to allow for greater cell density, and building isolation transistors with the necessary characteristics (e.g., good isolation when off, very low resistance when on) in this process may be challenging. Cost-optimized DRAM is a commodity business with very low margins and very large volumes. Thus DRAM manufacturers have thus far not adopted these proposals.

Additionally, since these proposals also require changes to the DRAM access protocol, consensus from different DRAM and processor manufacturers are required before these proposals can be adopted. Such consensus is often slow and difficult to reach among companies with different competing interests.

In contrast, my proposals, Duplicon Cache and Continuous Row Compaction, avoids these issues entirely, as I do not require modifications to the DRAM device or access protocol. Instead, I am able mitigate the effects of bank conflicts via transparent data duplication and migration.

Chapter 3

Main Memory Caches

As data are duplicated/migrated, the set of duplicated/migrated data essentially form a cache within main memory. Many of the issues that arise with data duplication/migration are in essence issues that arise when managing a large capacity cache in unmodified main memory. This chapter examines these issues.

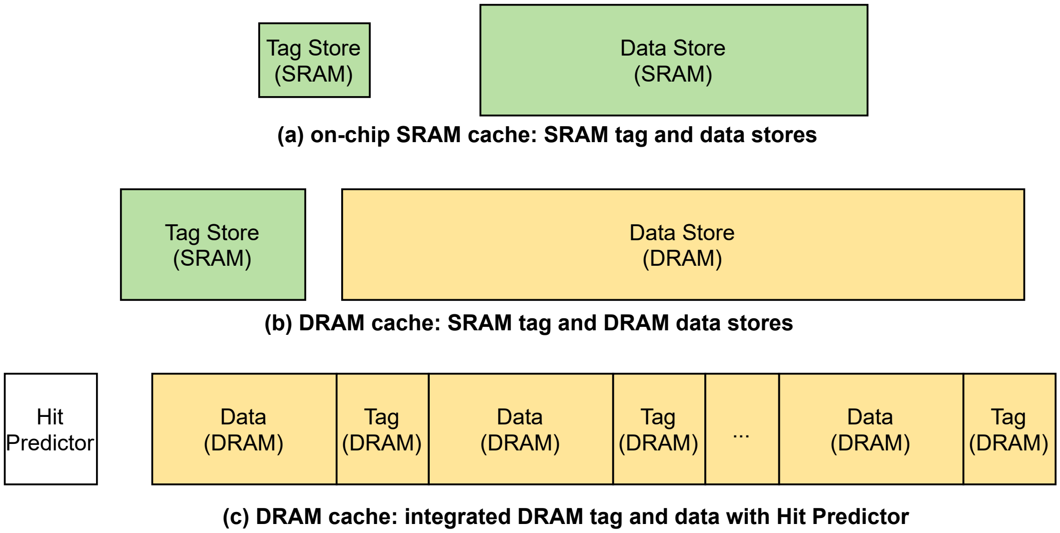
3.1 Tag and Data Store

A cache needs to store both tag and data. In the context of Duplicon Cache and Continuous Row Compaction, the data is the actual data being duplicated/migrated, while the tag encompasses all the information needed to track:

- which data have been duplicated/migrated
- where the data have been duplicated/migrated to
- other information needed for the maintenance of the cache (e.g. replacement info, dirty status)

Conventional SRAM caches store tag and data in two separate structures, tag store and data store. This is shown in Figure 3.1(a).

For larger capacity caches, storing the data entirely in SRAM becomes prohibitively expensive. Instead, data is stored in less expensive technology, such as DRAM. This is shown in Figure 3.1(b).



(a) on-chip SRAM cache: SRAM tag and data stores

(b) DRAM cache: SRAM tag and DRAM data stores

(c) DRAM cache: integrated DRAM tag and data with Hit Predictor

Figure 3.1: Tag and Data Store.

The tags, on the other hand, only require a fraction of the storage capacity of data, and can still fit in SRAM up to a point. However, for even larger capacity caches, new techniques are needed to reduce the size requirement of the tag store. One can, for example, increase the granularity at which tag information is tracked (i.e., increase the line/block size). We discuss this more in Section 3.3.2.

For the largest capacity caches, even the tags become too big to fit in

SRAM, and are instead kept in DRAM. This is shown in Figure 3.1(c). Rather than continuing to store tag and data in two separate structures, now both tag and data are integrated in the same DRAM memory, where the corresponding tag for each line/block is placed contiguous to the data. This allows both the data and tag to be accessed in a single DRAM access [51, 59]. Such caches need to be direct mapped so that one can know exactly where in DRAM to look for a particular piece of data [51]. Note that one cannot for sure know whether the data exists until one has read both the data and tag from DRAM. A hit predictor is employed to decide whether or not to parallelize the access to the DRAM cache with the access to the uncached copy of the data.

The two main memory caches I propose in this thesis, Duplicon Cache and Continuous Row Compaction, both employ SRAM tag stores and DRAM data stores. The SRAM tag store is maintained at the memory controller, giving the memory controller knowledge of which data have been duplicated/migrated, and to where. I leave adoption of DRAM tag stores as future work (Section 7.2.1).

3.2 Data Store

3.2.1 Reserving Physical Memory for Cache Data Store

Both Duplicon Cache and Continuous Row Compaction reserve memory at the top of the physical address space as the cache data store to store duplicated/migrated data. This is shown in Figure 3.2. Suppose the total physical address space is 2^m bytes, and we wish to reserve the top 2^k bytes for the data store. This reservation is done by under-reporting the amount of physical memory available to the software at boot time; that is, rather than reporting there are 2^m bytes of physical memory available, the hardware would report that there are only $2^m - 2^k$ bytes of physical memory available. This prevents the software from using physical memory between the addresses $2^m - 2^k$ and $2^m - 1$, . This is shown in Figure 3.2(a).

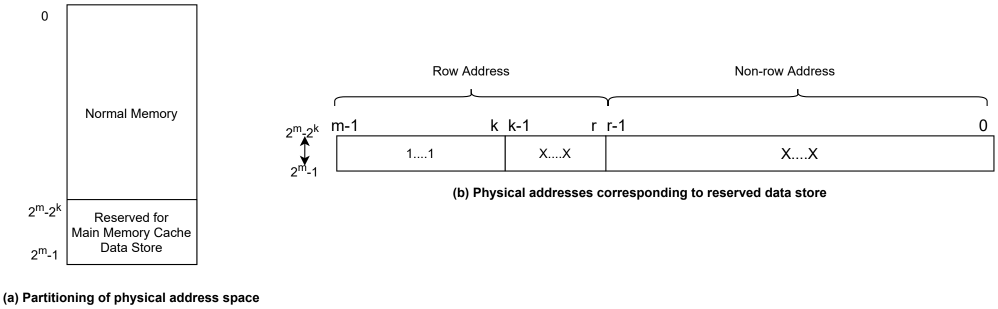

0
Normal Memory
2^m-2^k
Reserved for Main Memory Cache Data Store
2^m-1
(a) Partitioning of physical address space
Row Address
Non-row Address
2^m-2^k
2^m-1
m-1
k
k-1
r
r-1
0
1....1
X....X
X....X
(b) Physical addresses corresponding to reserved data store

Figure 3.2: Reserving the top 2^k bytes of an 2^m physical address space.

Figure 3.2(b) shows this reserved data store maps to physical addresses where the high bits $m-1$ to k are 1s. Since the row address bits are always placed at the highest order end of the physical address (see Figure 2.4(a) and (c)), we are essentially reserving the highest 2^{k-r} row addresses in each DRAM device for the main memory cache data store, where r is the position of the lowest row address bit. Our example in Figure 2.4(e), where we reserved the top $2^9 = 512$KB at the top of the physical address space for row compaction, was an example of reserving memory for the main memory cache data store where $m = 36$, $k = 19$, and $r = 19$.

3.3 Tag Store

We now consider issues relating to the design of a main memory cache tag store.

3.3.1 Serial vs. Parallel Tag and Data Accesses

The first consideration is whether to access the tag and data stores serially or in parallel. Accessing them serially is more efficient, as one will definitively know after accessing the tag store whether the data is in the data store, and where in the data store it is. Thus the data store access only happens if the data exists, and only the location with the data needs to be accessed. In contrast, accessing tag and data in parallel allows the two access latencies to be overlapped, but results in wasteful accesses to the data store.

For Duplicon Cache and Continuous Row Compaction, I assume the SRAM tag store and DRAM data store are accessed serially. In general, as DRAM is more expensive to access compared to SRAM, both in terms of performance and energy, one should minimize wasteful speculative accesses to DRAM. Since there is usually some queuing delay between the memory request arriving at the memory controller and when the request can actually be serviced, accessing the SRAM tag store can be overlapped with this delay.

3.3.2 Line Size and Sectoring

Another fundamental design parameter is the line size, also known as the block size. In general, the trade-offs associated with larger line sizes are:

- Advantages:
  - Smaller tag stores, as there becomes fewer lines to track
  - Increased spatial locality within each line
- Disadvantages:
  - Increased memory traffic from filling and writing back large lines
  - Potentially poorer utilization of cache capacity if the intra-line access pattern is sparse

Main memory caches have very large capacities and the tag store needs to track a lot of data. Adopting larger line sizes will reduce the tag store size, but increase the memory traffic and potentially result in poorer utilization of the cache capacity. Cache sectoring is a technique that allows for large line sizes without increasing the memory traffic. With sectoring, each line is divided into smaller, aligned sectors. Data is then filled and written back to/from the cache at the finer granularity of sectors, rather than the entire line. Separate coherence (e.g. valid, dirty) bits need to be maintained per sector, but only a single tag needs to be maintained per line. Figure 3.3 illustrates the distinction between small line size, large line size, and large line size with sectoring.

Sectoring eliminates the memory traffic overhead incurred by large line sizes, as only accessed sectors need to be transferred, rather than the entire line. However, sectoring does not eliminate the other drawback of large line

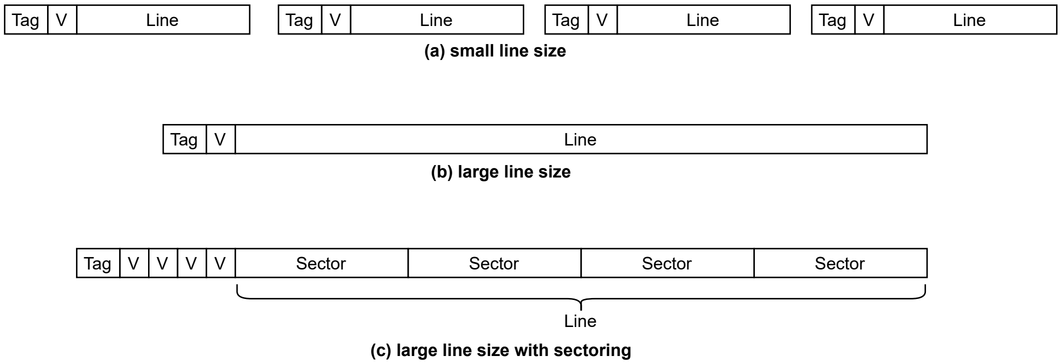


Figure 3.3: Sectored Caches

sizes, which is poor utilization of the cache when access patterns are sparse (i.e., when there is poor spatial locality). This is because all sectors of a line share the same tag, as shown in Figure 3.3(c). Thus different sectors from different lines cannot be mixed together.

Moreover, the additional coherence bit(s) per sector do increase the size of the tag store. At the very least, a single valid bit is required per sector. While still substantially cheaper than requiring a full tag, the additional bits per sector do add up for very large capacity caches.

Both Duplicon Cache and Continuous Row Compaction make use of sectoring to some degree. Duplicon Cache is organized as a sectored cache in its entirety, with duplication performed at 64B sector granularity and individual valid bits per sector tracking whether the sector has been duplicated, while tags are maintained at the 8KB line granularity. In contrast, the Continuous Row Compaction tag store is organized as a conventional non-sectored cache with 4KB line sizes (i.e., one tag per 4KB line, no sectoring). However, when 4KB lines are being filled or written back, they are migrated from/to their original locations in main memory at the granularity of 64B sectors. Thus I maintain valid and dirty bits for individual 64B sectors for 4KB lines currently being filled or written back.

Incidentally, there appears to be disagreement on the meaning of the terms sector and line/block. For example, André Seznec in 1994 reverses the terminology, calling the larger unit associated with the tag the sector, while calling the smaller sub-units within lines [54]. However, the oldest reference I could find on the matter, the PowerPC™ 601 RISC Microprocessor User's Manual [39] (which Seznec cites), calls the larger unit the line, and the smaller sub-units sectors. This is the terminology used in the rest of the thesis.

3.4 Mitigating Cache Data Transfer Overhead

Main memory caches also need to figure out how to move data to and from the cache without the additional memory traffic crippling performance.

Normally, in the absence of a main memory cache, regular data would be transferred to and from main memory via the memory channel, as shown in Figure 3.4(a). The yellow circled Ⓡs in the channel in the figure represent regular Read or Write requests (R stands for Regular) that are transferred over the channel.

With the addition of a main memory cache, additional data transfers

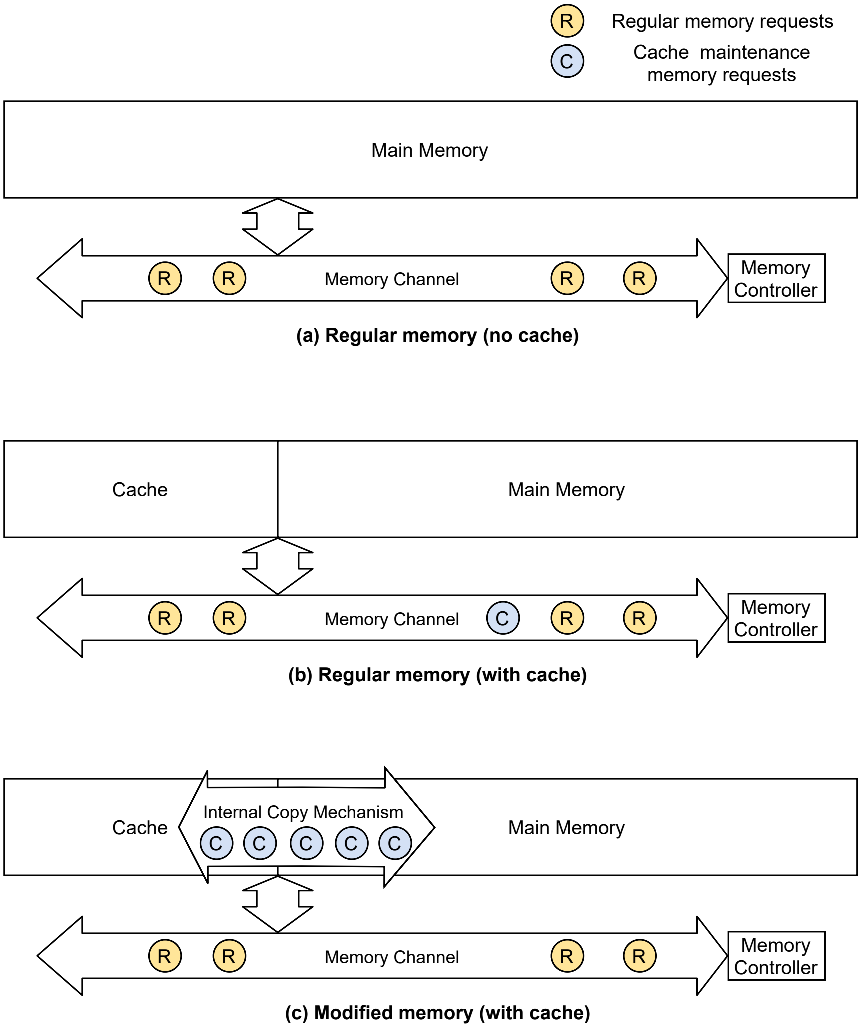


Figure 3.4: Handling extra data traffic from main memory cache fills and writebacks.

are needed to fill the cache and write back from the cache. I denote these additional cache maintenance data transfers with blue circled circled Ⓒs in the figure (C stands for Cache).

In the absence of any modifications to the memory device, the additional cache maintenance fills and writebacks are transferred over the existing channel. This is shown in Figure 3.4(b). This increases the contention for the memory channel and other memory resources, such as banks. Granted, the cache itself is supposed to help with the contention. However, too much cache induced traffic will negate the benefits of the cache. This is one of the main challenges of building a main memory cache out of unmodified memory devices.

3.4.1 Device Modifications to Support High Bandwidth Internal Copying

Prior work, again, modify the DRAM device to get around this problem. The DRAM device/protocol is modified to create internal copy mechanisms that allows high bandwidth copying of data entirely within the DRAM device, rather than over the regular memory channel. RowClone [53] modifies the Activate operation to allow an entire row to be copied to another row during sense-amplification. Low-Cost Inter-Linked Subarrays (LISA) [5] and Dynamic Asymmetric Subarray DRAM (DAS-DRAM)[34] proposed device and protocol modifications to allow bulk copying of data between adjacent subarrays (i.e., mats) within the DRAM device . FIGARO [74] proposed modifications to the DRAM device and protocol to allow for fine-grained and unaligned copying of data within the DRAM device using existing connections within the device.

These high bandwidth internal copying mechanisms allows main memory cache fills and writebacks to be transferred without creating contention on the regular channel. This is shown in Figure 3.4(c). Consequently, a lot more main memory cache fills and writebacks can be performed without hurting performance. The downside, of course, is the need for modifying the DRAM device and protocol.

3.4.2 Infrequent Replacements and Resulting Challenges

Since Duplicon Cache and Continuous Row Compaction do not modify the DRAM device and protocol, they do not have access to the large internal copy bandwidth that prior work assumed. Thus they do not have the ability to quickly fill and replace data in the cache. Instead, once data have been duplicated or migrated, they remain in the cache for a very long time.

For example, assume we have two DDR4-3200 channels. This provides a maximum aggregate channel bandwidth of 47.7 GB/s. Suppose we wish to limit overhead from main memory cache fills and writebacks to 2% of the maximum total bandwidth. This means only 47.7 * 0.02 = 0.954 GB/s of bandwidth can be used for fills and writebacks. Furthermore, each fill and writeback request requires two transfers over the channel: the first to read out the data to be filled/written back, followed by the actual write that performs the fill/writeback. If we further assume that every cache replacement results in a writeback (i.e., the replaced data is always dirty, and we have a writeback cache), then every unit of data replaced in the cache results in 4 units of data being transferred over the channel:

- 1 transfer to read the victim data from the cache location
- 1 transfer to write the victim data back to its original location
- 1 transfer to read the new data from its original location
- 1 transfer to write the new data to the cached location

Having so little memory bandwidth available for cache replacements creates three major challenges:

(A) How do we minimize the bandwidth requirement for each cache replacement? (Section 3.4.3)

(B) How do we enforce limits on aggregate cache replacement bandwidth utilization in order to limit the cache replacement bandwidth overhead? (Section 3.4.4)

(C) Given that cache replacement will occur very infrequently, how do we select which data to cache in order to maximize performance? (Section 3.4.5)

3.4.3 Using Less Bandwidth Per Replacement

3.4.3.1 Lazy vs. Eager Copying

So far we have assumed that replacing an unit of data in the cache results in 4 units of data being transferred over the channel: 1 to read out the old (i.e., victim) data from the cached location, 1 to write the old data back to its original location, 1 to read the new data from its original location, and 1 to write the new data to the cached location. I call this eager copying where explicit read and write requests are issued for the cache writeback/fill. However, this is not the only way to perform writebacks and fills. Instead, one can perform writebacks and fills lazily, by waiting until the application naturally, through running, reads or writes the data to be written back or filled. If application reads the data naturally, then the explicit read transfer for the writeback/fill can be omitted, as the data is already available at the memory controller. If the application is about to write the data naturally, then the writeback/fill can happen for free, as the writeback/fill simply entails redirecting the application write to the original/cached location in memory.

This is shown in Figure 3.5. Figure 3.5(a) shows the previously assumed eager writeback/fill case, where both the writeback/fill read (Figure 3.5(a)(1)) and write (Figure 3.5(a)(2)) are explicit memory requests created in addition to the regular application requests. This creates extra contention for the memory channel. Figure 3.5(b) shows a lazy writeback/fill on a natural application read. Now the explicit read can be omitted, because the data has already been read naturally; however, the explicit write is still required. Figure 3.5(c) shows a lazy writeback/fill on a natural application write. Now both the explicit read and write can be omitted, and the natural application write simply needs to be redirected to the original(writeback)/cached(fill) memory location.

While technically possible, lazy writebacks are somewhat silly, because if the victim data being written back was accessed by the application, then it

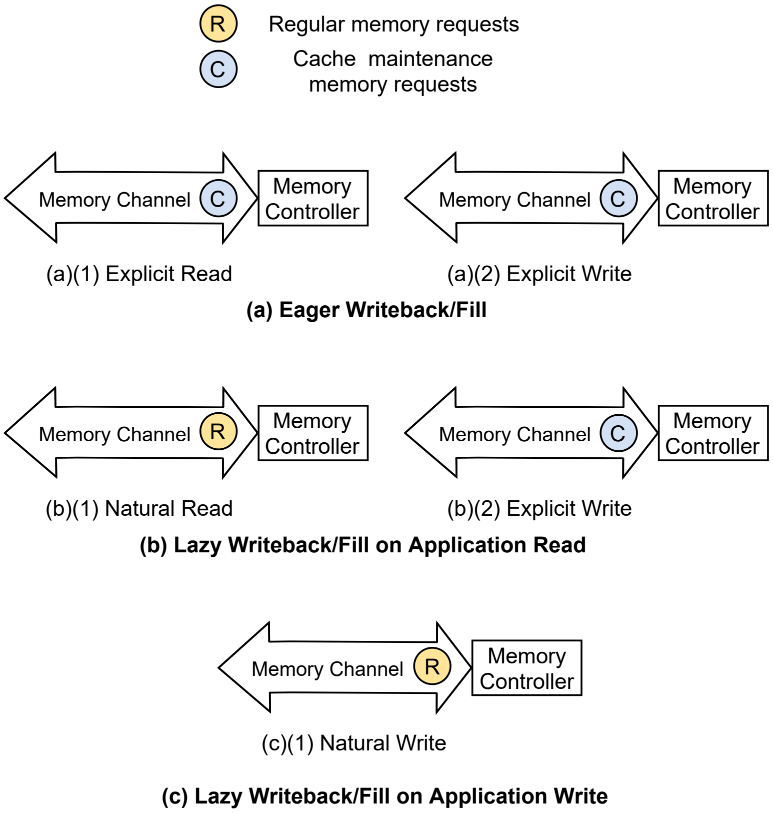


Figure 3.5: Eager vs. Lazy writebacks/fills.

probably should remain in the cache. Hence I primarily focus on lazy fills.

While lazy fills can help reduce the channel overhead for main memory caches, they have two main limitations:

Limitation 1: Need for Fine-Grained Tracking (Sectoring) The granularity of access with DDR4 SDRAM is 64B.[1] However, as stated in Section

[1] assuming burst length of 8

3.3.2, main memory caches employ larger line sizes (e.g., KBs) in order to reduce the size of the tag store. For example, Duplicon Cache adopts a line size of 8KB. As the application naturally accesses data at 64B granularity, lazy filling can only be done at 64B granularity. Hence some sort of fine-grained 64B tracking mechanism (e.g., cache sectoring) must be employed to keep track of which 64B pieces have been lazily filled. This increases the size of the tag store.

Limitation 2: Lack of Control The other limitation for lazy fills is that one has no control over when and what data the application will access next. This is problematic because frequently, one would monitor the application access pattern for a while to determine the most profitable piece of data to cache next (see Section 3.4.5). Yet once this piece of data is identified, there is no knowing when it will be accessed again.

This is especially true for lazy fills on application writes. While such fills have the least overhead (no explicit read nor write required), they are also the rarest and most unreliable. While the application will eventually read its entire working set, it will not necessarily write to its entire working set.

Table 3.1 summarizes the trade-offs for eager writebacks/fills, lazy writebacks/fills on application reads, and lazy writebacks/fills on application writes.

Table 3.1: Eager vs. lazy writeback/fill characteristics.

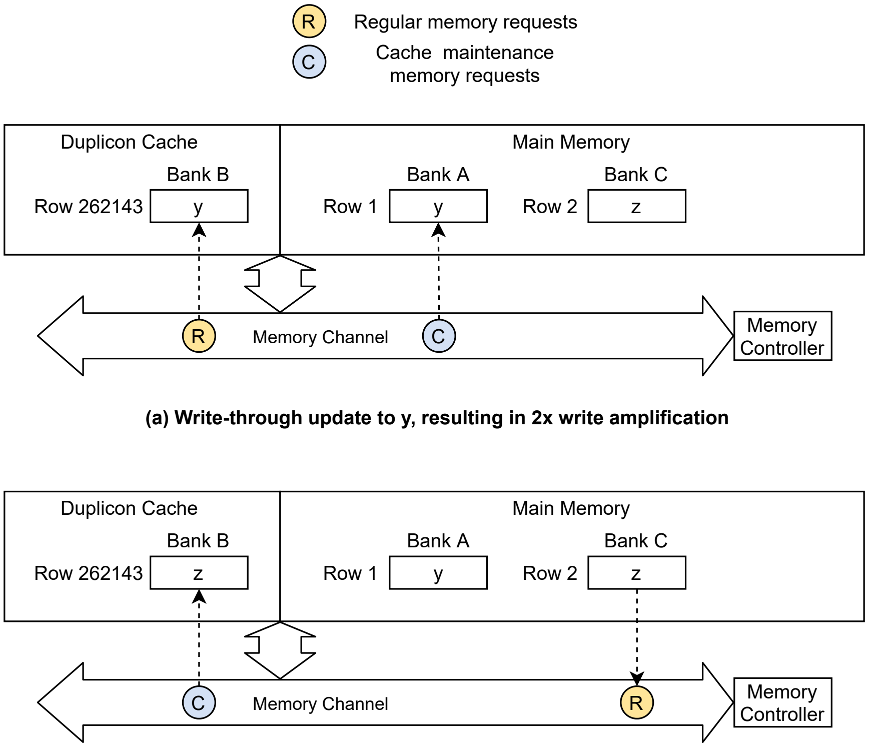
Type	Explicit Read	Explicit Write	Sectoring	Degree of Control
Eager Writeback/Fill	Required	Required	Not Required	High
Lazy Writeback/Fill on Application Read	Omitted	Required	Required	Medium
Lazy Writeback/Fill on Application Write	Omitted	Omitted	Required	Low

3.4.3.2 Write-Through vs. Writeback Cache

So far we have also assumed that every cache replacement results in a writeback. However, in general writebacks are only required when evicting dirty lines from writeback caches.

It is instead possible to have a write-through main memory cache that, upon every write to cached data, also updates the data in the original uncached memory location. Such a write-through main memory cache has the same trade-offs as a conventional write-through SRAM cache: writeback traffic is eliminated, but additional write-through traffic is added.

The Duplicon Cache is a write-through cache. In fact, it has to be a write-through cache as a consequence of it being a data duplication based cache. This is shown in Figure 3.6. Recall with Duplicon Cache, data is cached by virtue of it being duplicated to multiple banks. In Figure 3.6(a), the data item y, originally in (Bank A, Row 1), has also been duplicated to (Bank B, Row 262143). This allows y to be accessed with lower latency, because it can be sourced from either Bank A or B, depending on which bank is less heavily loaded.



(a) Write-through update to y, resulting in 2x write amplification

(b) Replacing y with z via a lazy fill on a natural read to z

Figure 3.6: Duplicon Cache as a Write-Through Cache.

Now suppose y is written to. If we only update one of the copies of y, then y ceases to be in the Duplicon Cache, because it is no longer duplicated. Hence, if we want to keep y in the cache after it is written to, we need to update the duplicated copy of y in Bank B, and also update the original copy of y in Bank A, as shown in Figure 3.6(a). This is exactly what would happen in a write-through cache, as each write to cached data results in an additional write to the original uncached data. Therefore Duplicon Cache is a write-through cache, and suffers from write-through traffic overhead. On the plus side, it means writebacks are not required during cache replacement. This is shown in Figure 3.6(b), where we replace y in the Duplicon Cache with a new piece of data, z. As Duplicon Cache is a write-through cache, y does not need to be written back before it is replaced. Furthermore, if we assume we are performing a lazy fill on a read to z (i.e., the application naturally read z while running), then only a single additional memory request, an explicit write of z to (Bank B, Row 262143), needs to be performed. In contrast, a regular write-back cache with eager writebacks and fills would have required four explicit memory requests to perform the same replacement.

3.4.3.3 Tracking Dirty Bits

Unlike Duplicon Cache, Continuous Row Compaction is organized as a writeback cache. Writeback caches in theory only need to write back dirty data, assuming one can track the dirty status of every 64B piece of data in the cache.[2] However, tracking a dirty bit for every 64B of migrated data substantially increases the size of the Continuous Row Compaction tag store. I thus do not track dirty status, and instead always write back the entire line (size 4KB) upon eviction.

One possible optimization is to track dirty status at some larger granularity between 64B and 4KB. I leave this as future work.

[2] DDR4 granularity of access is 64B, assuming burst length of 8

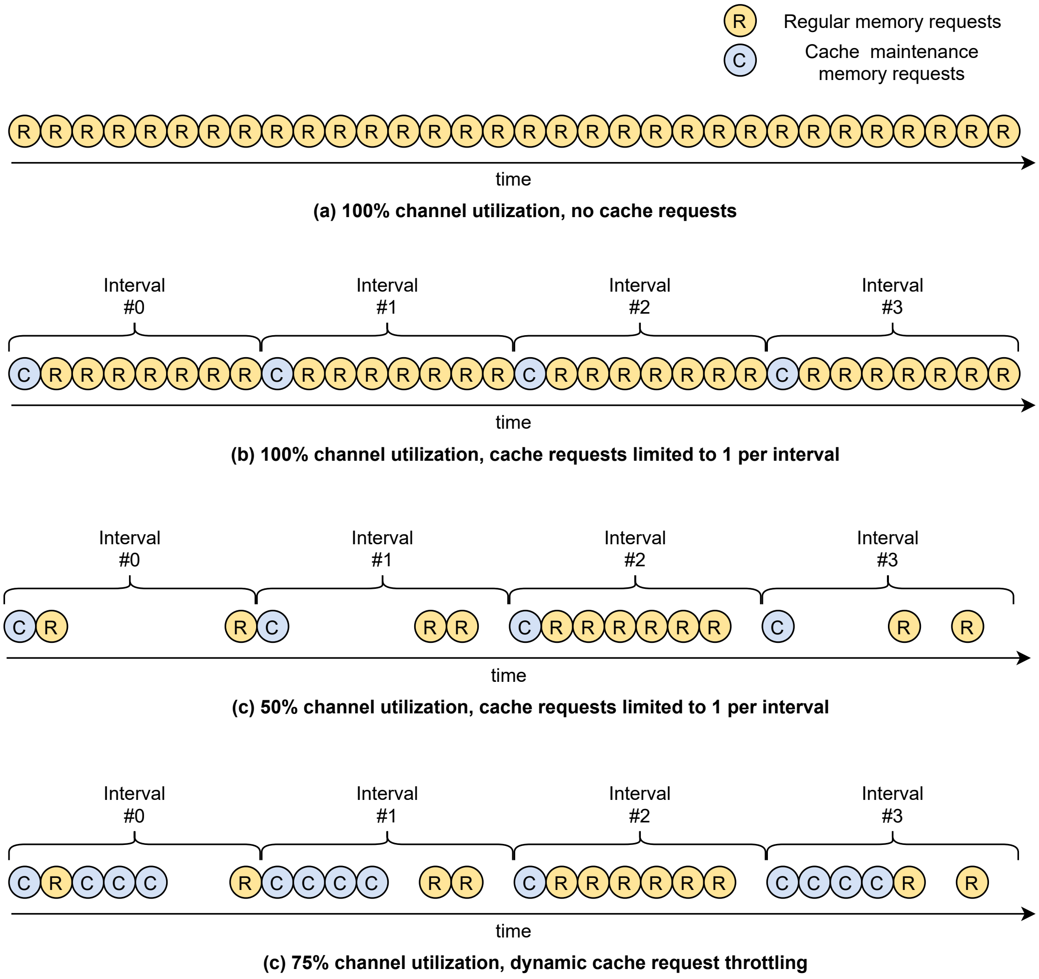
3.4.4 Limiting Replacement Frequency

Besides limiting the channel bandwidth utilization per replacement, we also need a mechanism that regulates the frequency of cache replacements in order to limit the total replacement bandwidth overhead. I present two such schemes. The first, Explicit Copying Throttling, explicitly limits the frequency of cache replacements by specifying a minimum time interval between cache replacements. The second, Implicit Usefulness Tracking, implicitly limits cache replacements by locking lines in the cache that have been deemed “useful” in some way (e.g., mitigated a bank conflict). Lines that are marked as useful are then protected from replacement and locked in the cache. The useful designation is sticky, so over time most of the cache will be marked useful, making replacements less likely.

3.4.4.1 Explicit Copying Throttling

The most straightforward way to limit the cache fill/writeback channel overhead is to explicitly limit how often fill and writebacks can occur. This is shown in Figure 3.7.

Suppose we wish to explicitly limit the channel overhead of cache fills and writebacks to $1/8 = 12.5\%$. One can proceed as follows. First, compute the total number of channel transfer cycles needed to replace one unit of data in the main memory cache. Suppose this is X cycles. Then, if one explicitly limited cache replacements to once every 8X cycles, then the channel overhead of fills and writebacks is bounded at $1/8 = 12.5\%$. This is shown in Figure



(a) 100% channel utilization, no cache requests

(b) 100% channel utilization, cache requests limited to 1 per interval

(c) 50% channel utilization, cache requests limited to 1 per interval

(c) 75% channel utilization, dynamic cache request throttling

Figure 3.7: Explicit throttling of cache fills and writebacks.

3.7 (b). Every 8X cycle interval we perform one single cache replacement, occupying the channel for X cycles, while the other 7X cycles of the interval are used for regular memory accesses.

Continuous Row Compaction adopts this form of explicit copying throttling. Recall that Continuous Row Compaction identifies and migrates data that are accessed sequentially in time to regions of memory that share the same row address. The unit of cache replacement in this case is the size of each region that shares the same row address. In our example in Chapter 2, this size is 512KB (see Figure 2.1(e)). Hence the unit of cache replacement is 512KB. Recall that, to replace 1 unit of data, 4 units of data transfers are required: 1 to read out the old data being replaced, 1 to write back the old data being replaced, 1 to read the new data being filled, and 1 to write the new data being filled. Hence every replacement of a 512KB region requires 4 × 512KB = 2MB to be transferred. Each DDR4 channel is 8 bytes wide, and can transfer 16 bytes every DRAM cycle (i.e., double data rate). Over two channels, the aggregate bandwidth is 32 bytes/DRAM cycle. Thus transferring 2MB would take 2M/32 = 64K DRAM cycles. This is the parameter X described at the start of the section, the total number of channel transfer cycles needed to replace one unit of data in the main memory cache. To limit the cache fill/writeback channel overhead to 2%, we would then need to limit cache replacement to once every 50X = 50(64K) = 3200K DRAM cycles.

Hence every 3200K DRAM cycles a 512KB region is replaced. I maintain a circular pointer that points to the next 512KB region to be replaced in the cache. Assuming a 512MB cache, then there are 512MB/512KB = 1K total such regions in the cache. Thus it would take 3200K * 1K = 3200M DRAM cycles for the circular pointer to completely cycle through the cache. This is the significance of the word "Continuous" in Continuous Row Compaction, as we continuously cycle through the cache and replace stale data with new data. While the rate of replacement is kept low to minimize the overhead of replacement, over time all stale data eventually gets replaced from the cache.

Increasing Replacement Frequency during Low Utilization Intervals

In the example of Figure 3.7, suppose the channel is only 50% utilized. This is shown in Figure 3.7 (c). In this case, our explicit throttling would still limit cache replacements to 1 every 8X cycles. However, since the channel is only 50% utilized, we could potentially make use of some of the idle channel cycles to perform additional cache fills and writebacks. This can be advantageous because newly accessed data can then be brought into the cache sooner.

Suppose a scheme existed that can dynamically vary the number of cache replacements performed. For example, we can have a scheme where we are limited to a single cache replacement for the interval if the channel utilization within the interval is greater than 75%, while additional cache replacements are allowed if the interval channel utilization is less than 75%. This dynamic scheme is shown in Figure 3.7 (d). Since the intensities of regular memory requests are relatively sparse in intervals #0, #1, and #3, the dynamic scheme is able to squeeze in extra cache replacements in these three intervals while keeping the interval channel utilization $\leq 75\%$. In interval #2, which has a long burst of regular memory requests, the dynamic scheme still limits the cache replacement channel overhead to 12.5%.

Such a dynamic scheme requires the ability to monitor and predict the regular request intensity for the current interval. I believe this is possible via a history based predictor that predicts the regular request intensity for the current interval based on observed intensity in the previous interval. However, I leave this as future work, and in my evaluations I only perform one cache replacement per interval, regardless of channel utilization.

Analogy to DRAM Refresh This concept of periodically cycling through and replacing small portions of the cache was inspired by the DRAM refresh mechanism.

DRAM cells store information in the form of charge on capacitors. Over time the capacitors leak, and periodically the charges on the capacitors need to be restored via an operation called Refresh. Refreshing essentially entails precharging the bank, then activating the row with the cell to be refreshed. The act of sense-amplification during activation restores the charge on the capacitor.

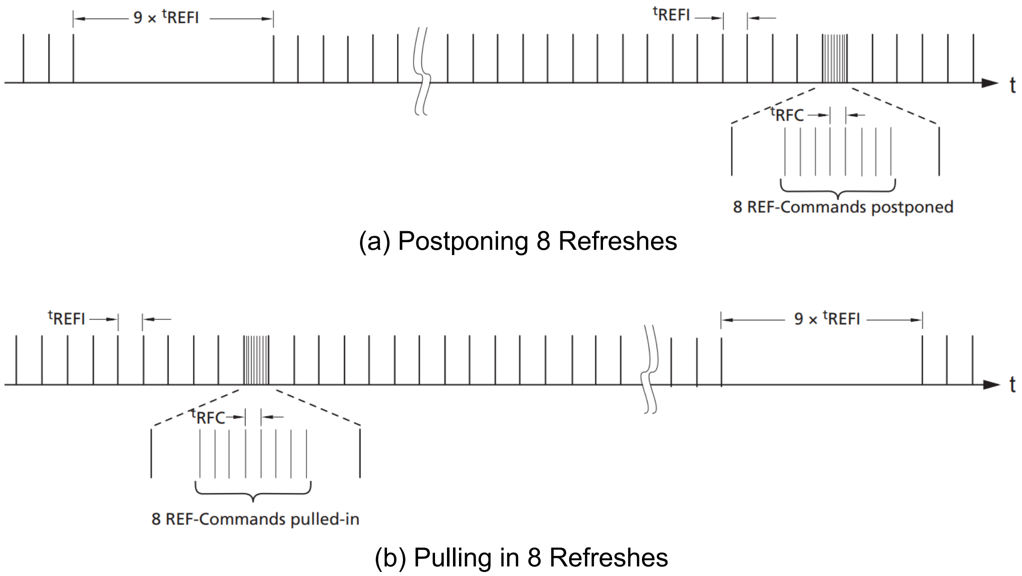
Each Refresh operation refreshes several rows at a time. Internally, each DRAM device maintains a circular pointer which tracks which rows need to be refreshed next. This is exactly analogous to the circular pointer maintained by Continuous Row Compaction which points to the next region of the cache to be replaced.

All banks in the device (i.e., in the rank) become unavailable during refresh for a number of cycles defined by the Refresh Cycle timing parameter, t_{RFC}. For the DRAM device evaluated (DDR4-3200 16Gb), $t_{RFC} = 350$ ns. This is analogous to the parameter X discussed earlier, which represented the total number of channel transfer cycles needed to replace one unit of data in the main memory cache. In our Continuous Row Compaction example, this is the number of channel cycles needed to replace a 512KB region, which was 64K DRAM cycles. In both the refresh example (t_{RFC} = 350 ns) and the Continuous Row Compaction example (X = 64K DRAM cycles), the quantity represents the length of time during which the memory is unavailable for regular application requests.

The Refresh Interval timing parameter, t_{REFI}, defines the time interval between Refresh operations. This is analogous to the chosen length of the interval between cache replacements, which was 8X in the example in Figure 3.7, and 3200K DRAM cycles in our Continuous Row Compaction example.

For the DRAM device evaluated, t_{REFI} = 7.8125 us = 7812.5 ns. One can then compute the performance overhead of refresh by computing the ratio between t_{RFC} (the amount of time the device is unavailable during each refresh operation) and t_{REFI} (the amount of time between refresh operations). In this case, the overhead is 350 ns/ 7812.5 ns = 4.48%. This is analogous to the channel overhead ratio X/8X = 1/8 = 12.5% in Figure 3.7(b), or the channel overhead ratio 64K/3200K = 2% in our Continuous Row Compaction example.

Additionally, if we know that there are 256K rows in the DRAM device, and that 32 rows are refreshed at a time during each Refresh operation, then we can compute the retention time, which is the amount of time each cell can retain charge for before requiring refresh. We know the retention time should equal the amount of time it takes for refreshes to wrap around all the rows once. Since there are 256K rows total, and 32 rows refreshed at a time, it would take 256K/32 = 8K = 8192 Refresh operations to completely refresh every row in the device. As the time interval between each Refresh operation is t_{REFI} = 7.8125 us, the cell retention time is 8192 * 7.8125 us = 64000 us, or 64 ms. Similarly, in our Continuous Row compaction example, the entire cache is 512MB, made up of 1K × 512KB regions. Thus 1K cache replacements are needed to completely renew the cache contents. Since the time interval between each cache replacement is 3200K DRAM cycles, the retention time for each 512KB region in the cache is 1K * 3200K = 3200M DRAM cycles.



(a) Postponing 8 Refreshes

(b) Pulling in 8 Refreshes

Figure 3.8: Postponing and pulling in Refreshes(from [38])

There is one last similarity between DRAM refresh and Continuous Row Compaction. On average, one Refresh operation should be performed every t_{REFI}, but the device allows for some flexibility and permits Refresh operations to be postponed or pulled in as necessary. This is shown in Figure 3.8. The horizontal axis denotes time. Normally, Refresh operations, denoted by vertical bars, occur at regular t_{REFI} intervals. However, during stretches where the device is busy with many requests queued up, it becomes advantageous to postpone Refresh operations until later, preventing the device from being unavailable while requests are waiting to be serviced. This is shown in Figure 3.8(a). 8 Refreshes are postponed, indicated by the large $9 \times t_{REFI}$ gap in time with no vertical bars. However, the eight postponed Refreshes need to be made up later. This is shown by the subsequent dense t_{REFI} interval with ten Refreshes - the two that normally bound a t_{REFI} interval, plus the eight postponed Refreshes that are now being made up.

Similarly, Refreshes can be pulled in during stretches when the device is idle. This is shown in 3.8(b). This allows future Refreshes to be skipped when the device is heavily loaded.

This flexibility of being able to postpone or pull in Refreshes based on the load is analogous to the dynamic throttling mechanism I proposed in Figure 3.7 (d). Again, I leave this as future work.

3.4.4.2 Implicit Usefulness Tracking

The alternative to an explicit scheme which specifies the minimum time interval between replacements is an implicit scheme which regulates the frequency of replacements by probabilistically making replacements less likely.

This is a scheme borrowed from the TAGE branch predictor [56], which uses it to determine which branch predictor table entries should be replaced. The idea is to associate a sticky Useful Bit with every line in the cache. Once the line has been deemed useful in some sense, then the Useful Bit is set, and the line is locked in the cache and protected from future replacement. Over time, most of the lines in the cache will be marked useful, making replacements less likely. Periodically, all the Useful Bits are cleared, which allows stale data to be eventually replaced.

Duplicon Cache adopts this implicit Usefulness Tracking scheme. There are two main design considerations for such a scheme. First, how exactly does one determine when a line should be marked as useful and be locked in the cache? Second, how often should one clear the Useful Bits? I address these issues in Section 4.5.

3.4.5 Choosing the Right Data To Cache

The last remaining challenge with infrequent cache replacement is that, unlike conventional SRAM caches, we cannot afford to cache every single piece of data that is accessed. Thus we cannot react to changes in the application working set over short time intervals, as a conventional SRAM cache would. Instead, our goal is to capture the larger long term working set of the application over very long time intervals.

This is shown in Figure 3.9. The application accesses data a through j in cyclic order in subsets of three: a, b, c are accessed repeatedly first, then

d, e, f, then g, h, and i, and then back to a, b, c, and so on.

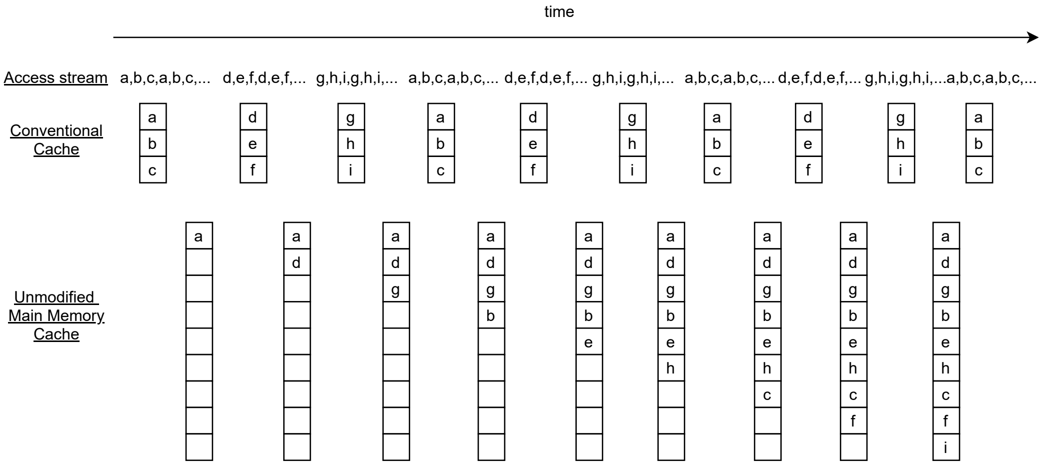

time
Access stream
a,b,c,a,b,c,... d,e,f,d,e,f,... g,h,i,g,h,i,... a,b,c,a,b,c,... d,e,f,d,e,f,... g,h,i,g,h,i,... a,b,c,a,b,c,... d,e,f,d,e,f,... g,h,i,g,h,i,...a,b,c,a,b,c,...
Conventional Cache
a b c
d e f
g h i
a b c
d e f
g h i
a b c
d e f
g h i
a b c
Unmodified Main Memory Cache
a
a d
a d g
a d g b
a d g b e
a d g b e h
a d g b e h c
a d g b e h c f
a d g b e h c f i

Figure 3.9: Caching for very long time intervals with infrequent fills.

A conventional SRAM cache, or a modified main memory cache with a high bandwidth internal copying mechanism, will be able to cache each new data item as it is accessed. Hence they can track the hot working set within the shorter time interval: $\{abc\}$, $\{def\}$, or $\{ghi\}$. In contrast, Duplicon Cache and Continuous Row Compaction have very low fill bandwidth and caches new data slowly. The goal is to instead try to eventually capture the working set within the longer time interval $\{abcdefghi\}$ over time, and exploit reuses over much longer time intervals.

The following sections describe the access characteristics considered when deciding what to duplicate/migrate for Duplicon Cache and Continuous Row Compaction.

3.4.5.1 Access Frequency

The foremost consideration is access frequency. Because cache replacement is so infrequent, we can monitor memory accesses over long time intervals before deciding on which piece of data to cache. Duplicon Cache only caches (i.e., duplicates) data after the number of monitored accesses exceeds a threshold. Continuous Row Compaction only migrates the data with the most accesses during the monitoring interval.

3.4.5.2 Access Cost/Benefit

The other consideration is what the cost of an uncached access would be, compared to that of a cached access. For example, Duplicon Cache derives benefit from being able to mitigate bank conflicts via duplication of data to another bank. Hence data that frequently suffer from bank conflicts would benefit most from duplication. Conversely, data that rarely suffer from bank conflicts, such as streaming accesses with good row buffer locality, would not benefit from duplication.

Note that for prior work that modified the DRAM device to create fast and slow regions within the device (e.g., TL-DRAM[29], FIGARO[74]), since they know the precise DRAM timing characteristics of both the fast and slow regions, they can precisely compute how many cycles a cached access would save compared to an uncached access. On the other hand, the cost/benefit computation for Duplicon Cache is a lot less exact, because whether or not a bank conflict occurs, and how much duplication can help, all depend on dynamic access patterns.

3.4.5.3 Criticality

For Duplicon Cache, I also consider whether the data is mostly accessed via demand accesses, or prefetch accesses. Because Duplicon Cache mostly helps with reducing individual request latency, but does not significantly increase the overall request throughput (since it in fact slightly lowers row buffer hit rates), it makes most sense to target demand accesses that are very likely to be on the application critical path. Prefetch accesses, on the other hand, can have some latency slack if they have been issued early enough. Admittedly, late prefetches can also end up on the application critical path, but as a first cut Duplicon Cache prioritizes data frequently accessed via demand accesses for duplication.

Continuous Row Compaction, on the other hand, does increase the overall request throughput by significantly increasing the row buffer hit rate. Hence Continuous Row Compaction does not distinguish between demand and prefetch accesses, as both can benefit from increased overall request throughput.

3.4.5.4 Temporal Correlation

For Continuous Row Compaction, the whole point is to identify data that are frequently accessed together in time (i.e., temporally correlated), then migrate them to non-conflicting row buffers. Thus the algorithm for identifying which data to migrate specifically tries to identify temporally correlated data. Duplicon Cache, in contrast, does not target nor rely on data temporal correlation, and does not consider temporal correlation when selecting which data to duplicate.

3.5 Coherence

Lastly, caches need to maintain coherence. This is actually relatively straightforward with both Duplicon Cache and Continuous Row Compaction because the memory controller has a centralized and authoritative view of all cached data via the SRAM tag store. In cases where there are multiple independent memory controllers, data duplication/migration can be managed separately at each of the controllers, since data belonging to different controllers are disjoint. Special care needs to be taken when data that is currently being duplicated/migrated gets modified by the application. This is addressed in more detail in sections 4.6 and 5.6 for Duplicon Cache and Continuous Row compaction, respectively.

3.6 Summary

Table 3.2 summarizes the differences between Duplicon Cache and Continuous Row compaction.

Table 3.2: Duplicon Cache vs. Continuous Row Compaction.

		Duplicon Cache	Continuous Row Compaction
Tag Store	Line size	4KB	8KB
	Sector size	64B	n/a (not sectored)
Replacement	Fills	Lazy on natural read/write	Eager
	Writebacks	n/a (write-through cache)	Eager
	Mechanism for reducing replacement frequency	Implicit Usefulness Tracking	Explicit Copying Throttling
Criteria for caching	Access Frequency	Favors frequently accessed data	Favors frequently accessed data
	Row conflict vs. hits	Favors data experiencing frequent row conflicts	Don't care
	Demand vs. Prefetch	Favors data accessed via demand requests	Don't care
	Temporal Correlation	Don't care	Specifically tries to identify temporally correlated data

Chapter 4

Duplicon Cache

This chapter presents the Duplicon Cache[31]. Many of the high level motivations, insights, and trade-offs for the Duplicon Cache have been steadily introduced throughout the thesis up to this point. This chapter summarizes these motivations, insights, and trade-offs, and provides additional implementation details and evaluation results.

4.1 Overview

DRAM devices are hierarchically organized into bank groups, banks, rows, and columns. Section 2.1 made the case that the key to achieving good performance with modern DRAM devices is to interleave accesses to different banks and, if possible, to different bank groups, while avoiding row conflicts that access data from different rows of the same bank. Duplicon Cache accomplishes this by duplicating select data to an alternate bank in another bank group, allowing duplicated data to always be interleaved to a different bank group. In cases of row conflicts, duplication allows the data to be serviced earlier by allowing the option to source data from the alternate bank if the original bank is heavily loaded, and vice versa.

This chapter begins with a potential study in section 4.2 that shows the potential performance benefit from removing bank and/or bank group conflicts, and motivates how data duplication across bank groups can achieve most of this benefit. The four subsequent sections then address the four main challenges to making Duplicon Cache work:

(I) How do we efficiently track what has been duplicated, and to where?

(II) How do we identify the most suitable data for duplication?

(III) How do we minimize the extra data movement overhead from data duplication?

(IV) How do we ensure coherence and correctness of data?

For challenge (I), efficiently tracking what has been duplicated (and to where), Duplicon Cache is organized as 4-way set-associative sectored cache with 8KB lines and 64B sectors. The detailed design of the Duplicon Cache tag store is presented in section 4.3.

To identify the most suitable data for duplication (challenge (II)), Duplicon Cache employs a mechanism called Demand Activates Filtering, which examines the data access frequency (section 3.4.5.1), whether or not the data suffers from row conflicts (section 3.4.5.2), and whether the data is accessed via demand or prefetch accesses (section 3.4.5.3). This is section 4.4.

To minimize the extra data movement overhead from data duplication (challenge (III)), Duplicon Cache employs techniques both to reduce the data movement overhead from each cache replacement (section 3.4.3), and to reduce the frequency of cache replacements (section 3.4.4). This is section 4.5.

Section 4.6 goes over how Duplicon Cache ensures coherence and correctness of data (challenge (IV)) while performing data duplication.

Section 4.7 explains the evaluation methodology and presents the evaluation result.

4.2 Motivation

Recall Figure 2.2 of chapter 2 described four different possible cases for back-to-back reads:

Case (a): back-to-back reads to different bank groups (no conflict)

Case (b): back-to-back reads to the same bank group, but to different banks, or to the same row of the same bank (bank group conflict, no bank conflict)

Case (c): back-to-back reads to different rows of the same bank, t_{RAS} already satisfied (bank conflict)

Case (d): back-to-back reads to different rows of the same bank, t_{RAS} not yet satisfied (bank conflict)

Case (a) resulted in the best performance with a 4 cycle latency between the back-to-back reads. Case (b) resulted in the next best performance with an 8 cycle latency between the reads. Case (c) latency is 56 cycles, and Case (d) latency is 78 cycles.

I devised a series of idealized experiments to measure the impact of such bank and bank group conflicts on real workloads. The first idealized experiment (i) approximates converting all bank conflicts into bank group conflicts by relaxing the bank mapping within a bank group, allowing requests to be serviced by any bank in the same bank group. Any request mapped to bank n of bank group m, which I denote as (m,n) can now alternatively be serviced by banks (m,0), (m,1), (m,2), or (m,3).[1] In essence, the first experiment (i) converts all case (c) and (d) accesses to case (b) accesses.

The second experiment (ii) approximates removing all bank group conflicts, but keeping bank conflicts. Recall that many DRAM timing constraints have long and short variants, where the long (i.e., worse performance) variant applies when the back-to-back operations are to the same bank group, while the short variant applies when the operations are to different bank groups. Experiment (ii) sets long variant of each timing constraint to the same value as the short variant, in essence converting all (b) accesses to (a) accesses. (c) and (d) accesses, however, remain row conflicts and still suffer from serialized Precharges/Activates to the same bank.

The third experiment (iii) approximates removing both bank and bank group conflicts by relaxing all bank mapping constraints, allowing any request

[1] the request still accesses the same row and columns at the new bank

to be alternatively serviced by any other bank/bank group[1], essentially converting (b), (c), and (d) accesses into (a) accesses.

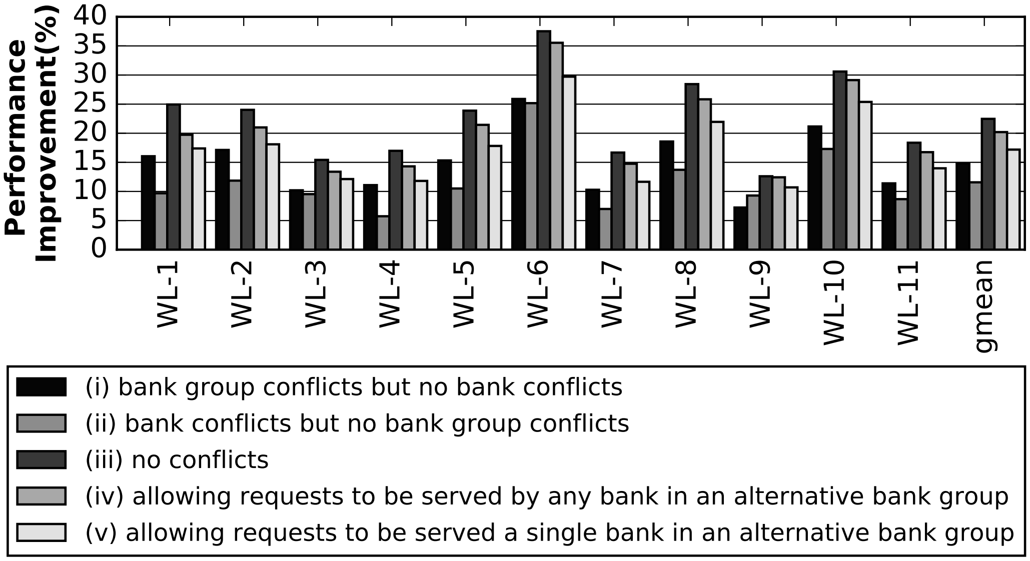


Figure 4.1: Performance improvement when bank and/or bank group conflicts are removed or mitigated.

Figure 4.1 shows the results of the three experiments across a set of eleven 4-core multi-programmed workloads formed from the memory intensive SPEC 2006 benchmarks and Graph 500. The workloads are listed in Table 4.2. Removing bank conflicts in (i) improved performance by 7.2% - 25.9% across the workloads, and by 14.8% on average; removing bank group conflicts in (ii) improved performance by 5.7% to 25.2% across the workloads, and by 11.6% on average; removing both bank and bank group conflicts in (iii) improved performance by 12.6% to 37.5% across the workloads, and by 22.5% on average. The results show removing bank and bank groups significantly improve performance in real workloads, and removing both improved performance far more than removing either one in isolation.

Experiment (iii) approximated removing all bank and bank conflicts by allowing any request to be serviced by any other bank/bank group. Such relaxation is possible if all data are fully duplicated to all bank/bank groups; unfortunately, full duplication has unacceptable storage and coherence overheads.

While full duplication is infeasible, we find limited duplication is sufficient to remove most bank/bank group conflict penalties. In experiment (iv), only the bank mapping to the next bank group is relaxed, so requests to banks in bank group m can only be alternatively serviced by banks in bank group m+1 (mod 4).[2] Figure 4.1 shows that (iv) retains most of the benefit of (iii), improving performance by 12.4% - 35.5% across workloads, and by 20.2% on average. Experiment (v) further restricts duplication such that requests to bank (m,n) can only alternatively be serviced by bank (m+1 (mod 4), n). This improves performance by 10.7% - 29.7% across workloads, and by 17.2% on average, which is worse than (iii) and (iv), but still substantial.

(iv) and (v) only require duplication of data between pairs of bank groups, as opposed to all-to-all duplication. We can further reduce the level of duplication required by applying the caching principle and only duplicate a select subset of data from each bank group to the next bank group. To this

[2] there are 4 bank groups in a rank in DDR4

end we propose the Duplicon Cache, a technique that mitigates the penalties of bank and bank group conflicts by duplicating select lines of data to an alternate bank group.

There are four main challenges to making Duplicon Cache work:

4.3 Challenge (I): Minimizing Tag Store Overhead

The first challenge is coming up with an efficient tag store design that can effectively track what data has been duplicated, and to where.

4.3.1 Set-Associativity

I first describe the set-associative architecture of Duplicon Cache. The Duplicon Cache data store is created by reserving space in different bank groups for storing duplicated data. This is done by reserving a small region at the end of the physical memory address space at boot time, as shown in Figure 4.2(a). I call this reserved physical address space the Reserved Storage. With 2^m bytes of total physical memory capacity, we reserve 2^k bytes of space at the end of the address space to store duplicate data. The software is then led to believe there are only $2^m - 2^k$ bytes of physical memory available, and will not allocate memory in the Reserved Storage. In our evaluated configuration $m = 34$ and $k = 27$; that is, we reserve $2^{27}B = 128MB$ of storage out of $2^{34}B = 16GB$ of total physical memory, for a storage overhead of 1/128.

We assume a 16GB DDR4 configuration with 2 channels, 1 rank, 4

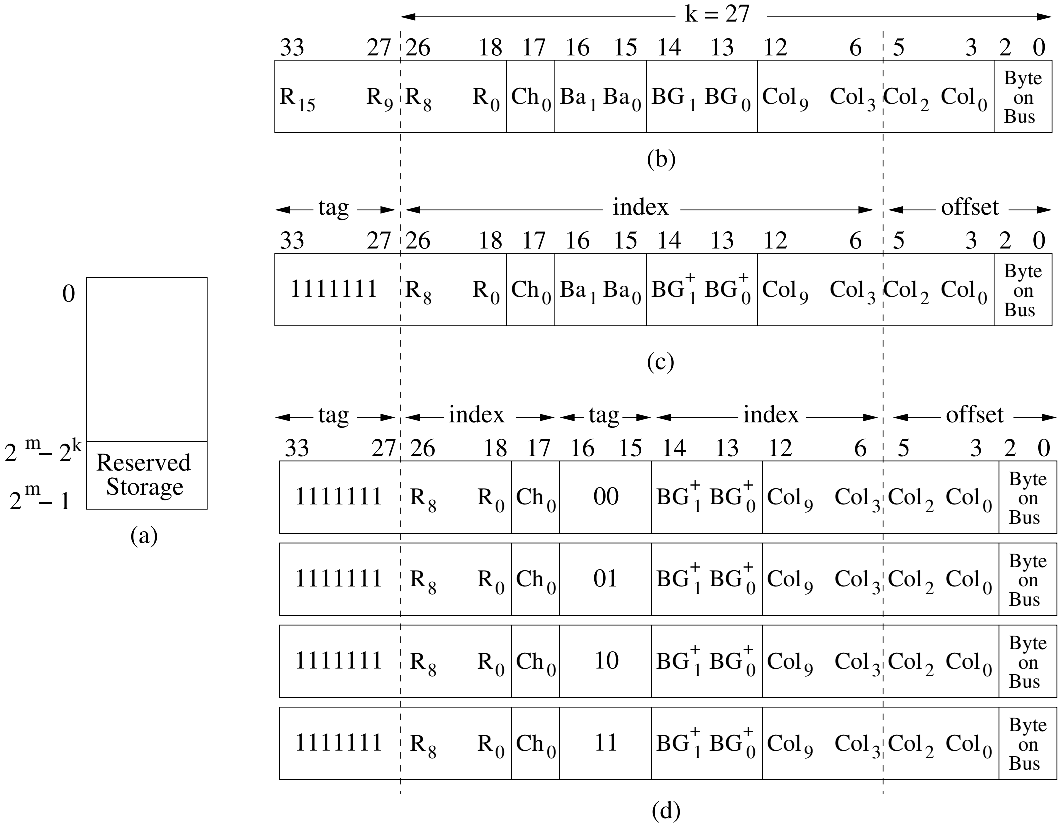

k = 27
33 27 26 18 17 16 15 14 13 12 6 5 3 2 0
R15 R9 R8 R0 Ch0 Ba1 Ba0 BG1 BG0 Col9 Col3 Col2 Col0 Byte on Bus
(b)
tag index offset
33 27 26 18 17 16 15 14 13 12 6 5 3 2 0
1111111 R8 R0 Ch0 Ba1 Ba0 BG+1 BG+0 Col9 Col3 Col2 Col0 Byte on Bus
(c)
0
2^m – 2^k
2^m – 1
Reserved Storage
(a)
tag index tag index offset
33 27 26 18 17 16 15 14 13 12 6 5 3 2 0
1111111 R8 R0 Ch0 00 BG+1 BG+0 Col9 Col3 Col2 Col0 Byte on Bus
1111111 R8 R0 Ch0 01 BG+1 BG+0 Col9 Col3 Col2 Col0 Byte on Bus
1111111 R8 R0 Ch0 10 BG+1 BG+0 Col9 Col3 Col2 Col0 Byte on Bus
1111111 R8 R0 Ch0 11 BG+1 BG+0 Col9 Col3 Col2 Col0 Byte on Bus
(d)

Figure 4.2: (a) reserving a region in physical memory for duplicates, (b) the original physical address, (c) single duplication destination way in a direct-mapped Duplicon Cache (d) set of possible duplication destination ways in a 4-way set-associative Duplicon Cache.

bank groups, 4 banks per bank group, 64K rows per bank, and 1K columns per row. Furthermore, I assume a plain Physical-to-DRAM address mapping with no XOR-ing, as shown in Figure 4.2(b).

All data in memory can be found in its original location, pointed to by its original address (Figure 4.2(b)). Data may then be duplicated from its original location to a Reserved Storage. To access the Reserved Storage, the high bits$[m-1{:}k]$ of the address are set to 1, while the low k bits dictate where in the Reserved Storage we are accessing. Where we duplicate data to in the Reserved Storage is a design decision that has implications for how we design the tag store. Recall the whole point of Duplicon Cache is to duplicate data to a different bank group. A direct-mapped scheme, shown in Figure 4.2(c), is one where the low k bits of the duplication destination address are identical to the low k bits of the original address, except that the bank groups bits (bits 14 and 13) BG_1BG_0 are replaced by $BG_1^+BG_0^+$, where $BG^+ = BG + 1 \pmod 4$. In this scheme each piece of data can only be duplicated to a single location in bank group BG^+. Alternatively, Duplicon can be organized as a 4-way set-associative cache, shown in Figure 4.2(d). Here, in addition to the bank group bits BG_1BG_0 being replaced by $BG_1^+BG_0^+$, the bank bits (16 and 15) are now completely free in the duplication destination address, meaning data can be duplicated to any of 4 banks in bank group BG^+. Note the analogy to traditional caches: with traditional caches, a direct-mapped scheme is one in which the data can only be cached in a single location, while an j-way set-associative scheme is one in which the data may be cached in any one of j ways in a given set, and all of them need to be searched for a cache hit. In our case, the 4 banks of bank group BG^+ form the 4 ways of our 4-way set-associative Duplicon Cache. As with traditional caches, the original address bits can be divided into (i) offset bits that dictate the byte offset within a line, (ii) index bits that dictate the set of locations to which the data may be cached, and (iii) tag bits that need to be tracked in order to differentiate between different data that map to the same set (i.e., have the same index bits). The breakdown of offset, index, and tag bits for both the direct mapped and 4-way set-associative schemes are shown in Figure 4.2.

The direct-mapped cache corresponds to experiment (v) in Figure 4.1, where data in bank (m,n) can only be alternatively serviced by bank (m+1 (mod 4), n); the 4-way set-associative cache corresponds to experiment (iv), where requests to bank group m can be alternatively serviced by any bank in bank group m+1 (mod 4). Figure 4.1 shows the 4-way set-associative cache performs better, so the 4-way set-associative configuration is assumed in the rest of the chapter.

4.3.2 The Tag Store

Duplicon maintains a tag store in a dedicated SRAM table at the memory controller to track which data have been duplicated. The tag store is searched upon each memory request to check if a duplicated copy exists.

Duplicon reduces the storage cost of tags via a sectored cache design (section 3.3.2) where the cache lines are 8KB DRAM rows, while the sectors are formed from aligned units of 8 columns each within the DRAM row, with each sector totalling 64B. All sectors in a line share a single Address Tag, reducing the size of the tag store. The tag store additionally maintains one valid bit per sector to mark which columns have been duplicated. As the Duplicon sector size is 8 columns (64B), one valid bit is needed for every 8 columns; 128 valid bits are needed for the 1K columns of each line. Collectively the valid bits

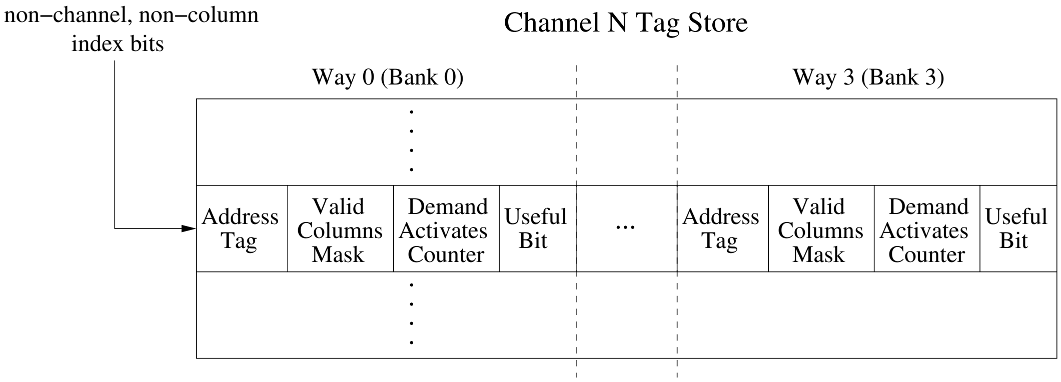


Figure 4.3: Duplicon Cache tag store.

form the Valid Columns Mask.

Since different channels may have different memory controllers, a separate tag store is maintained for each channel to track duplicated data within that channel. At each channel, we use the non-column (because all columns of the same row belong to the same line and share the same Address Tag) index bits from the original address to index into a set in the tag store. Each way in the set requires:

1. an Address Tag to identify the line (i.e., which DRAM row the line contains)

2. a Valid Columns Mask to identify which columns have valid data

3. a Demand Activates Counter (DAC), saturating counter used for the Duplicon Cache insertion policy (Section 4.4)

4. an Useful Bit, used for the Duplicon Cache replacement policy (Section 4.5.1)

Figure 4.3 shows the tag store for a particular channel. For an access to hit in the Duplicon cache, the Address Tag for the line needs to match, and the corresponding bit in the Valid Columns Mask needs to be set. On a hit, the memory controller then has the option to service the request from the alternate bank if it will be available sooner.

Section 4.7.3 addresses the tag store area cost. The Tag Store requires 142KB/channel in our evaluated configuration, for a total of 284KB with two channels. In section 4.7.2.3 we also evaluate whether this extra area cost is justified by comparing Duplicon performance against that of the baseline with additional on-chip SRAM cache added, and show that Duplicon substantially outperform the baseline with additional on-chip SRAM cache.

4.4 Challenge (II): Identifying the Most Suitable Data for Duplication

Duplication incurs non-trivial costs in terms of storage and extra memory write traffic. Thus it is important to only duplicate data that are likely to impact program performance. Duplicon uses three criteria to determine whether data should be duplicated. First, Duplicon looks at the overall access frequency (section 3.4.5.1). Second, Duplicon tracks how often the data suffers from DRAM row misses/conflicts, as such accesses incur longer latencies and are more likely to benefit from duplication (section 3.4.5.2). Third, Duplicon tracks how often the data is accessed via demand (as opposed to prefetch) read requests, as such requests are likely to be on the program critical path (section 3.4.5.3).

4.4.1 Demand Activates Filtering

We can measure all three criteria with a single metric, the number of Demand Activates to the data. A Demand Activate is an Activate to a row for a demand read request. The number of Demand Activates identifies the number of demand non-row buffer hit accesses to the data, as row buffer hits do not require an Activate.

Duplicon tracks the number of Demand Activates in the tag store, maintaining a saturating Demand Activates Counter(DAC) for each cache line. We allocate a line in one of the ways of the tag store on the first Demand Activate to the row, and increment the DAC for each subsequent Demand Activate. Duplication of individual sectors(i.e., 64B columns) in the row only proceed after the DAC surpasses a threshold (Thrsh), but once the threshold is reached we duplicate on all accesses, not just Demand Activates. We swept over a large range of Thrsh values and found 15 to be a sweet spot for our evaluated configuration.

4.5 Challenge (III): Minimizing Data Movement Overhead

To reduce the overall data movement overhead, we need to both reduce the amount of data movement per cache replacement, and reduce the frequency of cache replacements.

To reduce the data movement overhead from each cache replacement, Duplicon Cache employs lazy fills on natural application reads and writes (section 3.4.3.1). This means cache fills (i.e. duplications) only occur in the aftermath of a normal application read or write to the data to be duplicated. While the data is available at the memory controller from the normal application read/write, an explicit duplication write request for the data is created to the appropriate location in the Reserved Storage. This explicit write request is then buffered and serviced by the memory controller like an ordinary memory write request.

In addition, Duplicon Cache is organized as a write-through cache (section 3.4.3.2), which removes the need for cache writebacks upon replacement.

To reduce the frequency of cache replacement, Duplicon Cache implicitly controls the rate of cache replacement via two mechanisms, Usefulness Tracking and Probabilistic Replacement.

4.5.1 Usefulness Tracking

Duplicon tracks the usefulness of each duplicated cache line (i.e., DRAM row) via the Useful Bit in the tag store (Section 4.3.2). Lines are initially marked not useful, but become useful when a duplicated sector in the line gets used (i.e., when the duplicated sector gets sourced by a read request because of a conflict at the original bank/bank group). Lines that are marked as useful cannot be replaced. Periodically, all the Useful Bits are cleared. We swept over a range of Useful Bit reset periods and found resetting the Useful Bits every million memory requests worked well, although the sensitivity to the reset period is quite low provided the period is large enough.

Based on the values of the Useful Bit and the Demand Activates Counter(DAC), each cache line in Duplicon Cache is in one of four states:

1. **Invalid** (DAC = 0): no DRAM row has been allocated to the cache line yet

2. **Monitoring** ($1 \leq$ DAC $<$ Thrsh): a DRAM row has been allocated to the cache line, and we are tracking Demand Activates to increment the DAC, but not yet duplicating

3. **Duplicating-not useful** (DAC $\geq$ Thrsh, Useful=0): a DRAM row has been allocated to the cache line and we are duplicating on all accesses (even writes and prefetches), but no duplicated data has been used; the line may be replaced

4. **Duplicating-useful** (DAC $\geq$ Thrsh, Useful=1): a DRAM row has been allocated to the cache line and we are duplicating on all accesses (even writes and prefetches), and duplicated data has been used; the line may not be replaced

4.5.2 Probabilistic Replacement

The Useful Bit protects lines in the Duplicating-useful state from being overwritten, but does not protect lines in other states. To give lines in the

Monitoring and Duplicating-not useful states time to reach the Duplicating, useful state, we introduce a parameter ϵ which controls the probability that a line in states Monitoring or Duplicating-not useful can be replaced. A properly chosen ϵ parameter should give time for beneficial lines in Monitoring and Duplicating-not useful to become useful, while still eventually replacing the lines that never do. We performed a sweep of the ϵ parameter and used $\epsilon = 1/256$ in our experiments.

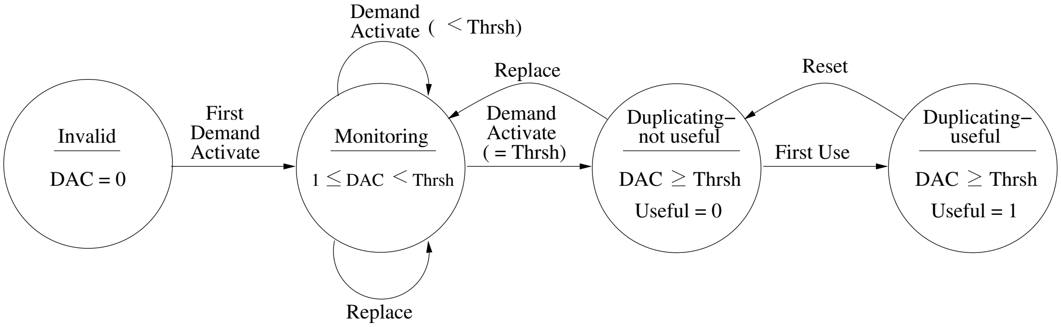


Figure 4.4: Cache line state diagram.

The state machine for each cache line is shown in Figure 4.4. All cache lines start in the Invalid state, and transition to the Monitoring state on the first Demand Activate, at which point the line is allocated to the DRAM row. Lines in the Monitoring state have their DAC incremented on each subsequent Demand Activate to the row. When the DAC reaches Thrsh, the line transitions to the Duplicating-not useful state. From this state data is duplicated on each subsequent access (including prefetches and writes). If all the lines in the set are allocated, then lines in the Monitoring and Duplicating-not useful states may be replaced with probability ϵ each time another row wishes to allocate into the set. If a replacement occurs, then the Address Tag is updated to the new row, the Valid Columns Mask and Useful Bit are cleared, and the Demand Activates Counter is set to 1, putting the new row/line in the Monitoring state. Lines in Duplicating-not useful are promoted to Duplicating, useful when a duplicated sector is used (i.e., the memory controller decided to source the duplicated data). Lines in Duplicating-useful are protected from replacement. On an Useful Bit reset all lines in the Duplicating-useful state are demoted to the Duplicating-not useful state.

Figure 4.5 is a flowchart that summarizes this sequence of events on read requests. Actions (allocate cache line, replace cache line, increment DAC, etc.) in the flowchart are enclosed in boldface boxes.

4.6 Challenge (IV): Ensuring Data Coherence and Correctness

To ensure coherent and correct data in light of data duplication, two things need to happen upon a write to duplicated data. First, the existing duplicate cache sector corresponding to the data must be invalidated by clearing the appropriate bit of the Valid Columns Mask in the tag store entry. Second, the memory controller write buffer must be searched to remove any pending duplication write requests to the sector. Figure 4.6 summarizes the sequence of events on write requests. Again, actions in the flowchart are enclosed in boldface boxes.

These requirements does not increase hardware complexity of the write

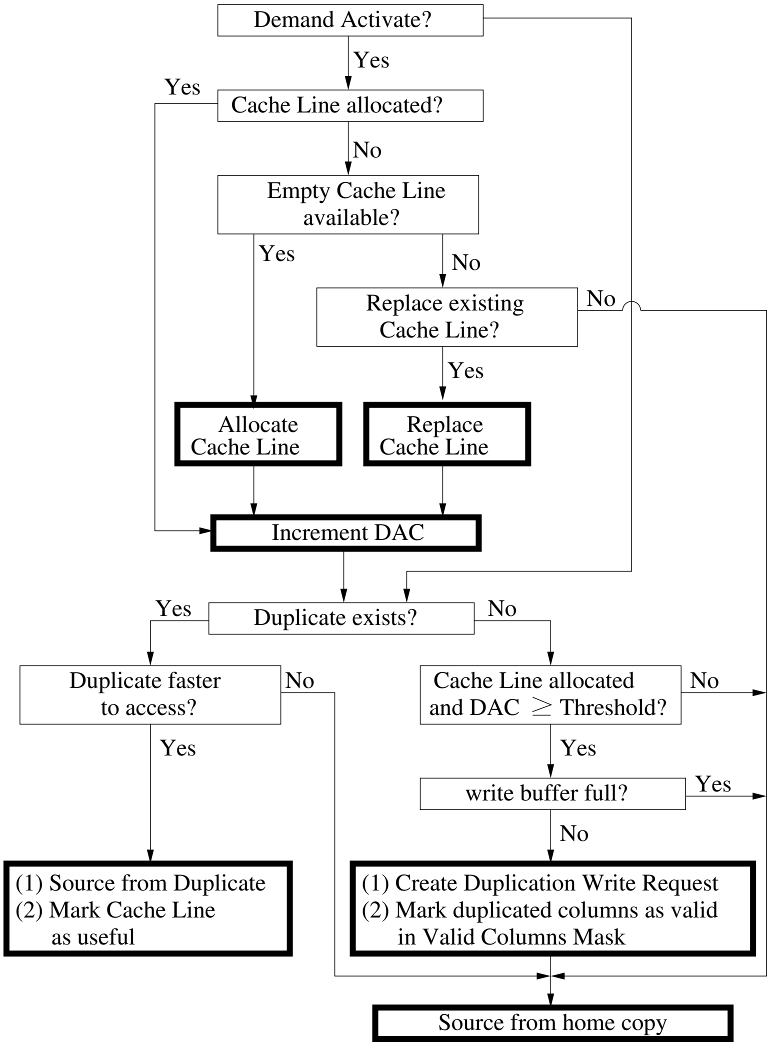

Demand Activate?
Yes
Yes
Cache Line allocated?
No
Empty Cache Line available?
Yes
No
Replace existing Cache Line?
No
Yes
Allocate Cache Line
Replace Cache Line
Increment DAC
Yes
Duplicate exists?
No
Duplicate faster to access?
No
Cache Line allocated and DAC ≥ Threshold?
No
Yes
Yes
write buffer full?
Yes
No
(1) Source from Duplicate
(2) Mark Cache Line as useful
(1) Create Duplication Write Request
(2) Mark duplicated columns as valid in Valid Columns Mask
Source from home copy

Figure 4.5: Flowchart for read.

buffer, as it already needs to support searches for a particular line to allow data from ordinary pending write requests to be forwarded to subsequent matching read requests.

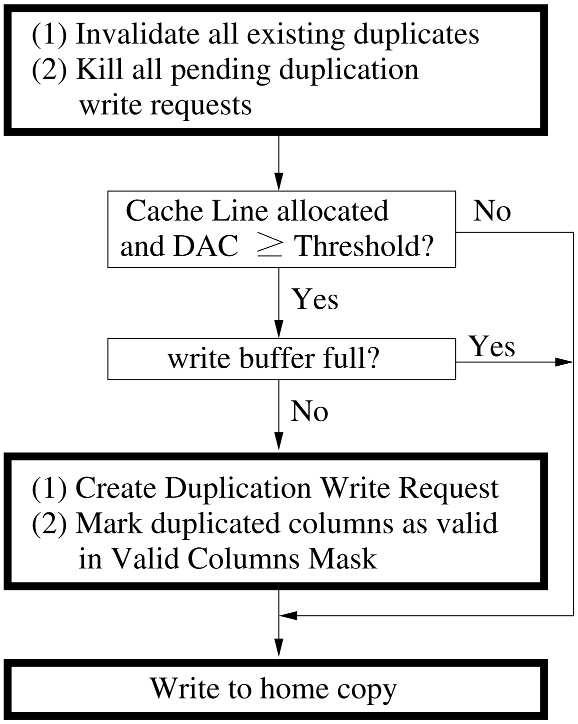

(1) Invalidate all existing duplicates
(2) Kill all pending duplication write requests
Cache Line allocated and DAC ≥ Threshold?
No
Yes
write buffer full?
Yes
No
(1) Create Duplication Write Request
(2) Mark duplicated columns as valid in Valid Columns Mask
Write to home copy

Figure 4.6: Flowchart for write requests.

4.7 Evaluation

4.7.1 Methodology

We evaluated our mechanism on an execution-driven, cycle-accurate simulator for a 4-core out-of-order x86 processor. The frontend of the simulator is based on Multi2Sim [71]. The simulator models port contention, queuing effects, and bank conflicts throughout the cache hierarchy and includes a detailed DDR4 SDRAM model which models t_{CL}, t_{CWL}, t_{RP}, t_{RCD}, t_{RAS}, t_{RTP}, $t_{CCD(L/S)}$, $t_{RRD(L/S)}$, t_{FAW}, $t_{WTR(L/S)}$, and t_{WR}. Table 4.1 describes our baseline configuration. Chip power and energy are modeled using McPAT [30], and DRAM power and energy are modeled using CACTI [41].

To mimic the effects of virtual-to-physical address translation in our simulation, we pass the Virtual Page Number (VPN) concatenated with the processor ID through a hash function (Paul Hsieh's SuperFastHash[15]) to generate the Physical Frame Number, which is then combined with the page offset to form the physical address. The DRAM channel/bank group/bank/row/column addresses are then computed using the mapping function in Fig. 4.2(b) from this generated physical address. This virtual-to-physical hashing in our simulation maximizes the entropy in the channel/bank group/bank addresses.

The ten most memory-intensive SPEC 2006 benchmarks with the highest single-threaded Last-Level-Cache (LLC) misses per kilo instructions (MPKI) without prefetching, along with the Graph 500 benchmark, were used to form 11 randomized 4-core multi-programmed workloads such that each benchmark appears in four workloads. Table 4.2 shows the workloads. We represent each application with the highest weight representative SimPoint. Each workload is simulated until every application in the workload has completed at least 800 million instructions from the representative SimPoint [46]. Static power of shared structures is dissipated until the completion of the entire workload. Dynamic counters stop updating upon each benchmark's completion.

We report the Harmonic Mean of Weighted-IPCs [35] for Chip-Multiprocessor (CMP) performance. The Harmonic Mean of Weighted-IPCs is the reciprocal of the Average Normalized Turnaround Time(ANTT)[9], and is a measure of both fairness and system throughput [9, 8]. All performance graphs re-

Table 4.1: Baseline Configuration.

Core	4-Wide Issue, 128 Entry ROB, RS size 48, Hybrid Branch Predictor, 3.2 GHz Clock
L1 Caches	32KB I-Cache, 32KB D-Cache, 64 Byte Cache Lines, 2 Reads Ports, 1 Write Port, 3 Cycle Latency, 4-way Set-Associative, Write-back
Last Level Cache	Shared 4MB, 64 Byte Cache Lines, 12 Cycle Latency, 8-way Set-Associative, Write-back, Inclusive
Memory Controller	128 Entry Memory Queue, FR-FCFS[52] Open-Page Policy, precharge oldest idle bank when memory controller is idle
Prefetcher	Stream Prefetcher [70]: Streams 64, Distance 64, Queue 128, Degree 4 with Feedback Directed Prefetching(FDP) [62] to throttle prefetcher
DRAM	2 Channels, 1 Rank/Channel, 16 Banks/Rank, 8Gb DDR4-3200 x8 chips Bus Frequency 1.6GHz (DDR 3.2GHz) t_{RCD}-t_{RP}-t_{CL}: 22-22-22 8KB Row Buffer (1KB x8)

Table 4.2: Evaluated multi-programmed workloads.

Name	**Workloads**
WL-1	bwaves + Graph500 + lbm + mcf
WL-2	bwaves + lbm + mcf + sphinx3
WL-3	bwaves + lbm + omnetpp + milc
WL-4	GemsFDTD + bwaves + Graph500 + leslie3d
WL-5	GemsFDTD + Graph500 + milc + soplex
WL-6	GemsFDTD + lbm + mcf + libquantum
WL-7	GemsFDTD + leslie3d + omnetpp + soplex
WL-8	Graph500 + leslie3d + libquantum + omnetpp
WL-9	leslie3d + libquantum + soplex + sphinx3
WL-10	libquantum + mcf + milc + sphinx3
WL-11	milc + omnetpp + soplex + sphinx3

port Harmonic Mean of Weighted-IPCs unless otherwise stated. We additionally report the Weighted Speedup [58] and Unfairness[8, 11, 44] for our best performing configuration. Weighted Speedup differs from Harmonic Mean of

Weighted-IPCs in that Weighted Speedup is only a measure of system throughput. The equations for Harmonic Mean of Weighted-IPCs(HMWI), Weighted Speedup (WS), and Unfairness are given below:

$$HMWI = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{shared}}}, WS = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Unfairness = \frac{MAX(\frac{T_0^{shared}}{T_0^{alone}}, \frac{T_1^{shared}}{T_1^{alone}}, \ldots, \frac{T_{N-1}^{shared}}{T_{N-1}^{alone}})}{MIN(\frac{T_0^{shared}}{T_0^{alone}}, \frac{T_1^{shared}}{T_1^{alone}}, \ldots, \frac{T_{N-1}^{shared}}{T_{N-1}^{alone}})}$$

Where N is the number of cores, IPC_i^{alone} is the IPC of application i running alone on one core in the CMP system while other cores are idle, IPC_i^{shared} is the IPC of application i running on one core while other applications are concurrently running on other cores, T_i^{alone} is the number of cycles it takes application i to run alone, and T_i^{shared} is the number of cycles it takes application i to run with other applications.

4.7.2 Performance

4.7.2.1 Ideal vs. Realized Performance

Fig. 4.7 compares the idealized performance potential of the Duplicon Cache against actual realized performance. We start with the idealized experiment (iv) in Fig. 4.1. Recall experiment (iv) allowed any requests to bank group m to be alternatively serviced by banks in bank group m+1 (mod 4). Effectively this represents an idealized Duplicon Cache where:

1. there are no cold misses (i.e., everything is already duplicated)

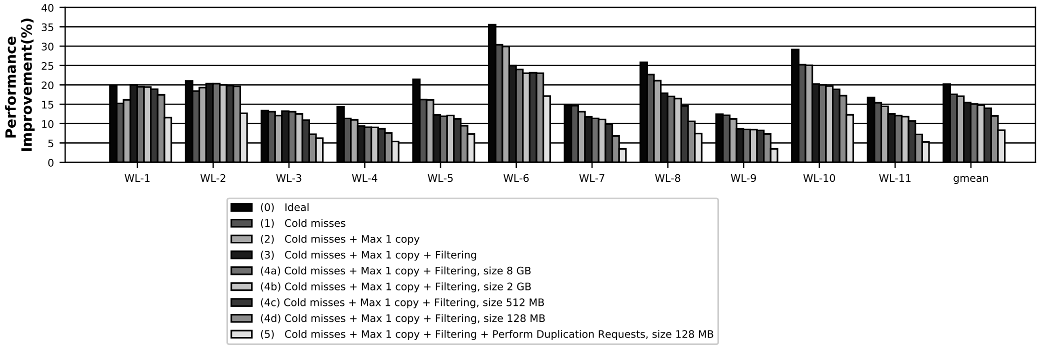


Figure 4.7: Ideal vs. realized performance improvement.

2. data are duplicated to all banks in the alternate bank group (i.e., four duplicate copies are available)

3. all rows can be duplicated (i.e., no Demand Activates Filtering)

4. the tag store and reserved storage are infinitely sized

5. duplication write requests are free and do not cause interference

These idealized assumption are removed one by one until we end up with a realistic Duplicon Cache implementation.

Cold misses are added in experiment (1); now only data that have been encountered before can be serviced by the alternate bank group. Introducing cold misses removes some of the potential, reducing the average potential performance gain from 20.2% to 17.6%.

Experiment (2) limits the maximum number of duplicates to 1; now each piece of data is assigned to a single bank in the alternate bank group the first time it is encountered, and subsequent accesses can only alternatively source the data from that bank (previously the request can be serviced by any bank in the alternate bank group). Adding this constraint had little effect, changing the potential performance gain from 17.6% to 17.1%, showing that allowing further duplication of the same data yields little benefit, In fact, in some cases (WL-1 and WL-2) limiting the maximum number of duplicates to 1 actually improved performance. This is because sourcing from an alternate bank can actually reduce row buffer locality in the alternate bank. In general row buffer locality improves as we limit duplication, because requests are more likely to stay in their original bank and less likely to interfere with rows in other banks. However when bank conflicts do occur the penalty is lower if the data has been duplicated to another bank.

Experiment (3) considers the effects of Demand Activates Filtering (Section 4.5.1). Demand Activates Filtering reduces the number of useless duplications, but also limits which rows can be duplicated, decreasing the performance gain potential. We modeled an infinite sized tag store and tracked Demand Activates for each DRAM row encountered. Recall duplication is only allowed after the row reaches the Demand Activates threshold (Thrsh), so data has to be seen one more time after the row reaches the threshold before we allow it to be sourced from the alternate bank group. On average Demand Activates Filtering reduces the potential from 17.1% to 15.4%. Again on select workloads (WL-1, WL-2, WL-3) adding Demand Activates Filtering, which limits duplication, results in better row buffer locality and can improve the performance.

Experiments (4a), (4b), (4c), and (4d) consider the effect of sizing the Duplicon Cache from infinite sized to 8GB, 2GB , 512MB, and 128MB. The average performance gain drops from 15.4% to 15.1%, 14.8%, 14.0%, and 12.0%, respectively, showing bigger Duplicon Cache sizes can provide additional gains, but at the cost of both additional main memory and tag store storage.

So far we have assumed duplication to be free. Experiment (5) considers the cost of actually performing duplication write requests. This drops the performance gain from 12.0% to 8.3%. 8.3% is the final performance gain realized after considering all costs and constraints. This shows the overhead from duplication writes is considerable, and can negate all performance gains if left unchecked. This demonstrates the importance of techniques like lazy fills on reads/writes, usefulness tracking, and probabilistic replacement that reduce the overall overhead from duplication. In the next section we consider what happens when these techniques are removed.

4.7.2.2 Effectiveness of Demand Activates Filtering and Usefulness Tracking

Duplicon employs Demand Activates Filtering (Section 4.4) to reduce the number of useless duplications, and Usefulness Tracking (Section 4.5.1) to protect useful duplicated lines. Fig. 4.8 shows the importance of both mechanisms. (0) is the performance gain with both mechanisms; (1) is when Demand Activates Filtering is removed (i.e., the Monitoring state is removed from the

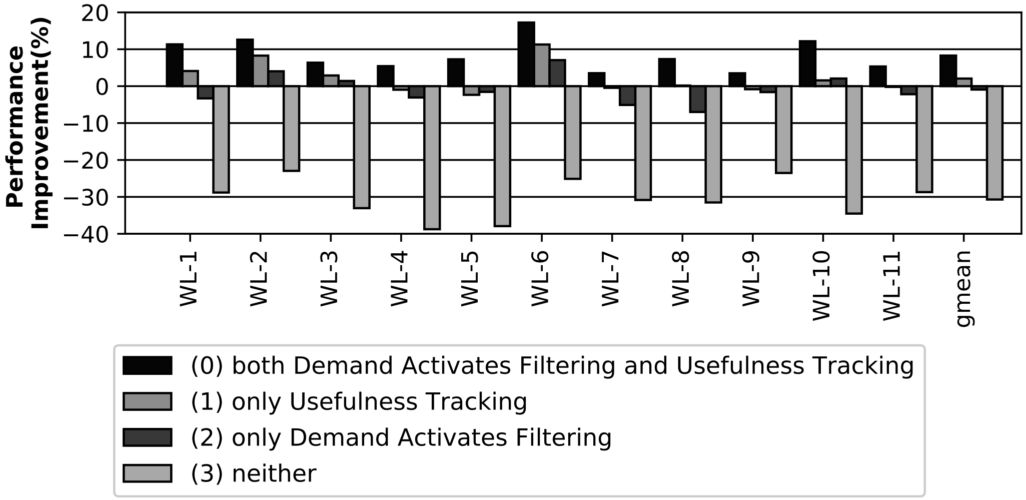


Figure 4.8: Performance without Demand Activates Filtering/Usefulness Tracking.

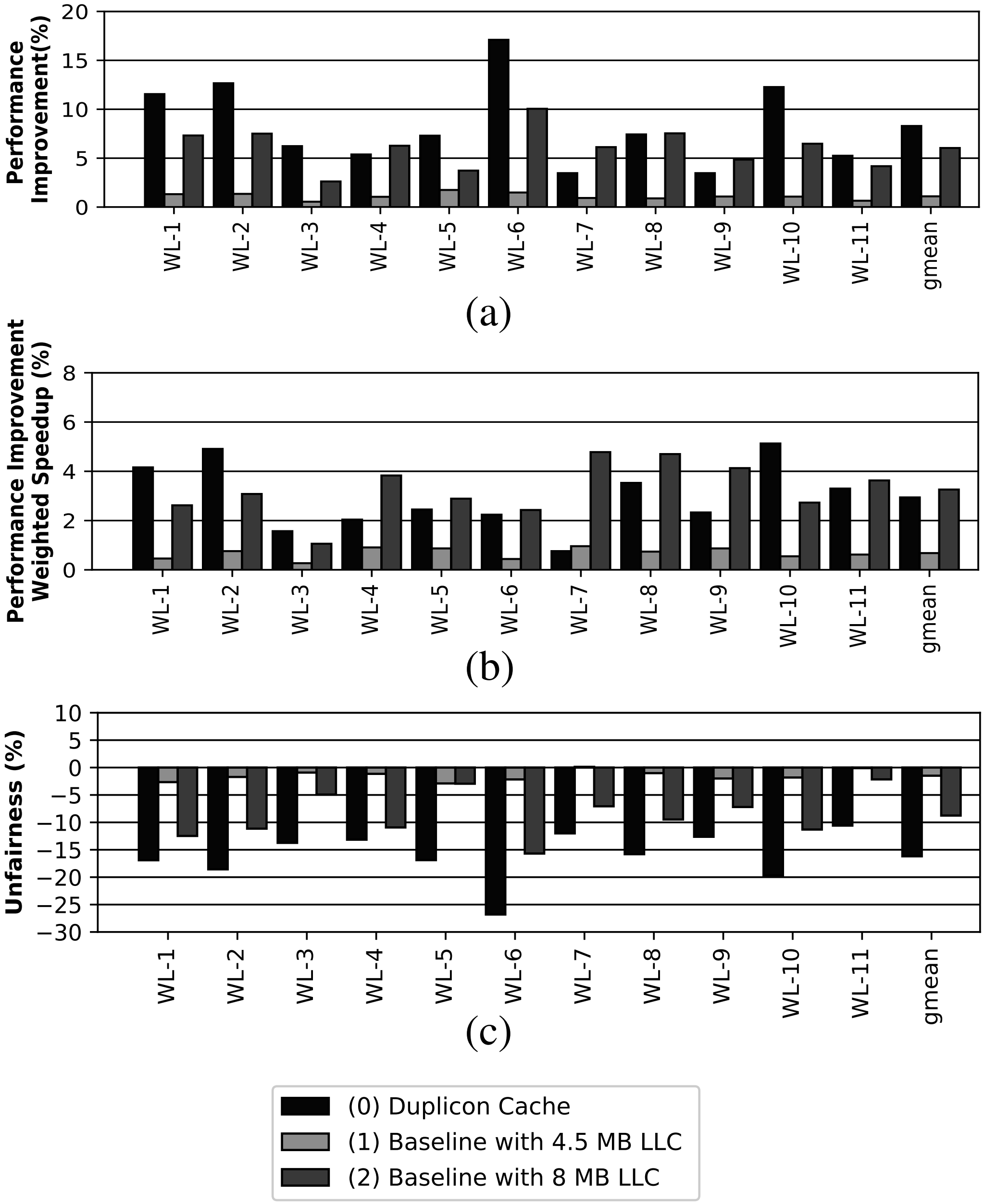
state diagram in Fig. 4.4); (2) is when Usefulness Tracking is removed (i.e., the Duplicating-useful state is removed); (3) is when both Demand Activates Filtering and Usefulness Tracking are removed. The results clearly show both mechanisms are required: removing Demand Activates Filtering drops the average performance gain from 8.3% to 2.1%; removing Usefulness Filtering drops the average performance gain to -0.9%; removing both drops the average performance gain to -30.7%. There is synergy between the two mechanisms: Usefulness Tracking is based on actual duplication outcome (i.e., something duplicated in this row was actually later used), whereas Demand Activates Filtering is heuristics based. We use the heuristics based Demand Activates Filtering to first determine what might be suitable for duplication, then use Usefulness Tracking to more rigorously evaluate if the row should have been duplicated.

4.7.2.3 Comparison to Area-Equivalent Baseline

The Duplicon Cache tag store takes up a non-trivial amount of storage - 284KB in total for our evaluated configuration (see Section 4.7.3 for details). This additional storage alternatively could have been used elsewhere on chip to improve performance - by increasing the on-chip Last-Level-Cache (LLC), for example. Since we evaluated with a 4MB 8-way set-associative LLC, the smallest increment at which the LLC can be increased is by an extra way, or 512KB, which is nearly double the amount of storage we added. Nonetheless we conservatively compare the Duplicon Cache with a 4MB LLC (config 0) against the baseline with a 4.5MB LLC (config 1). We in addition compare against the baseline with an 8MB LLC (i.e., doubling the LLC) (config 2). In addition to reporting performance in terms of the Harmonic Mean of Weighted-IPCs as in the rest of the paper, we also report the Weighted Speedup and Unfairness. For the Unfairness metric, lower is better. Fig. 4.9 shows the comparison.

Duplicon outperforms the baseline with 4.5MB LLC in all workloads and by all metrics. Duplicon also outperforms or matches the baseline with 8MB LLC in most cases, so while Duplicon requires additional on-chip area for the tag store, the extra area cost is well justified.

We also note that Duplicon Cache substantially improves fairness in the system, by 16.2% on average. This is because in the baseline, FR-FCFS memory scheduling introduces unfairness in the system by penalizing row conflict



(a)

(b)

(c)

Figure 4.9: Performance comparison to baseline with added LLC using different metrics: (a) Harmonic Mean of Weighted-IPCs, (b) Weighted Speedup, (c) Unfairness

accesses behind row buffer hits. Duplicon reduces this unfairness by allowing the conflicting request to be serviced sooner from the alternate bank.

4.7.2.4 Effect on Request Latency

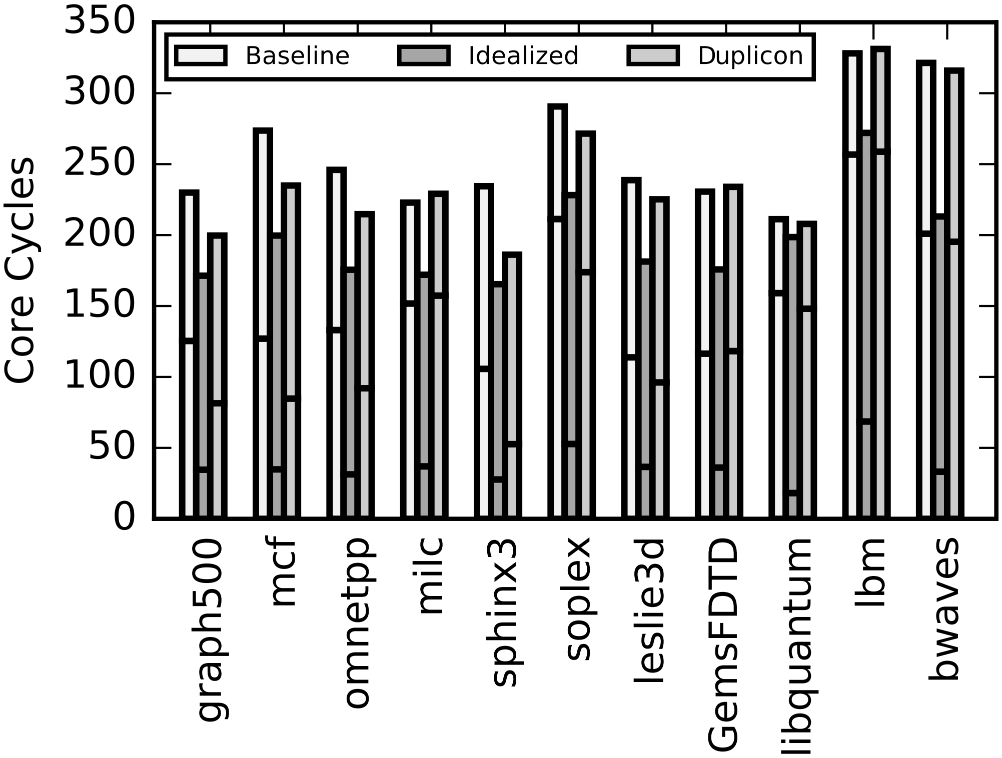


Figure 4.10: Breakdown of average request latency.

Figure 4.10 shows the average memory request latency breakdown for each of our eleven benchmarks. For each benchmark, the average memory request latency is computed across all the 4-core multi-programmed workloads that the benchmark appears in.

There are three stacked bar for each benchmark, representing the average memory request latency of that benchmark in:

1) the baseline configuration,

2) the idealized experiment (iii) from Figure 4.1 that removed all bank and bank group conflicts by allowing any request to be serviced by any bank in the rank, and

3) our final Duplicon Cache configuration.

Each stacked bar is made up of a bottom component, representing the queuing delay, and a top component, representing the access delay. The queuing delay is the delay between the request arriving at the memory controller, and the first DRAM operation for the request being issued. In the case of a row hit, this first DRAM operation would be a read or a write; for a row miss (i.e., the bank was precharged), the first operation is an Activate; for a row conflict (i.e., the bank had the wrong row activated), the first operation is a Precharge. The access delay is the delay between the issuing of the first DRAM operation and when data is actually transferred over the channel.

We first note that, for the idealized experiment with minimal bank/bank group conflicts, represented by the middle stacked bar corresponding to each benchmark, the overall request latency is reduced for all benchmarks compared to the baseline. In particular, the queuing delay (bottom component of each stacked bar) is substantially reduced, while the access delay (top component) is increased. This makes sense, because the idealized experiment allowed any request to be serviced by any available bank, reducing the time each request needed to wait before issuing its first operation to a bank. But allowing requests to be serviced by any bank also diluted the row buffer locality at the original bank, resulting in more row conflicts and an increased access delay.

The average request latency with Duplicon Cache is represented by the rightmost stacked bar of each benchmark. For the benchmarks graph500, mcf, omnetpp, sphinx3, soplex, and leslie3d, the Duplicon request latency follows the same general trend as the idealized experiment, but to a lesser degree, reducing the overall request latency by substantially reducing the queuing delay, while increasing the access delay. The benchmarks milc, GemsFDTD, libquantum, lbm, and bwaves see slightly worse or only marginally better average latencies with Duplicon, because these benchmarks have very large working sets that exceed the Duplicon Cache capacity.

4.7.2.5 8-Core Performance

Duplicon Cache was also evaluated on 8 cores, keeping the number of channels the same (2 channels), while doubling the overall on-chip LLC and Duplicon Cache to keep the amount of LLC and Duplicon Cache capacity per core the same (8MB LLC total, 256MB Duplicon Cache storage total, 1MB/core LLC, 32MB/core Duplicon Cache). Table 4.3 shows the evaluated 8-core multi-programmed workloads. As before, I took the ten most memory-intensive SPEC 2006 benchmarks, plus the Graph 500 benchmark, then formed

22 randomized 8-core multi-programmed workloads such that each benchmark appears 16 times across all 22 8-core workloads.

Table 4.3: 8-core mixes.

WL-1	bwaves + graph500 + lbm + leslie3d + libquantum + mcf + milc + soplex
WL-2	GemsFDTD + omnetpp + sphinx3 + GemsFDTD + libquantum + mcf + soplex + sphinx3
WL-3	leslie3d + bwaves + milc + omnetpp + lbm + graph500 + graph500 + libquantum
WL-4	GemsFDTD + bwaves + lbm + leslie3d + mcf + milc + omnetpp + sphinx3
WL-5	soplex + bwaves + graph500 + lbm + leslie3d + milc + soplex + sphinx3
WL-6	GemsFDTD + mcf + omnetpp + libquantum + graph500 + leslie3d + mcf + sphinx3
WL-7	GemsFDTD + bwaves + milc + omnetpp + lbm + libquantum + soplex + sphinx3
WL-8	GemsFDTD + bwaves + lbm + leslie3d + mcf + milc + omnetpp + soplex
WL-9	libquantum + graph500 + GemsFDTD + lbm + mcf + milc + omnetpp + soplex
WL-10	leslie3d + bwaves + libquantum + graph500 + sphinx3 + graph500 + leslie3d + milc
WL-11	GemsFDTD + bwaves + lbm + libquantum + mcf + omnetpp + soplex + sphinx3
WL-12	bwaves + graph500 + lbm + leslie3d + libquantum + milc + omnetpp + sphinx3
WL-13	GemsFDTD + mcf + soplex + bwaves + graph500 + leslie3d + milc + soplex
WL-14	GemsFDTD + mcf + omnetpp + lbm + libquantum + sphinx3 + bwaves + milc
WL-15	GemsFDTD + graph500 + lbm + leslie3d + libquantum + mcf + omnetpp + sphinx3
WL-16	soplex + GemsFDTD + bwaves + graph500 + leslie3d + mcf + omnetpp + soplex
WL-17	milc + lbm + libquantum + sphinx3 + GemsFDTD + bwaves + libquantum + sphinx3
WL-18	leslie3d + mcf + milc + omnetpp + lbm + graph500 + soplex + milc
WL-19	GemsFDTD + graph500 + leslie3d + libquantum + mcf + omnetpp + soplex + sphinx3
WL-20	bwaves + lbm + bwaves + graph500 + leslie3d + mcf + omnetpp + soplex
WL-21	GemsFDTD + milc + lbm + libquantum + sphinx3 + bwaves + lbm + soplex
WL-22	GemsFDTD + graph500 + leslie3d + libquantum + mcf + milc + omnetpp + sphinx3

Figure 4.11 shows the performance improvement from baseline for Duplicon Cache for 8 cores. We see that Duplicon Cache scales very well to 8 cores. In fact, Duplicon Cache gives even more benefit with 8 cores than with 4 cores (13.3% gmean performance improvement for 8 cores, versus 8.3% for 4

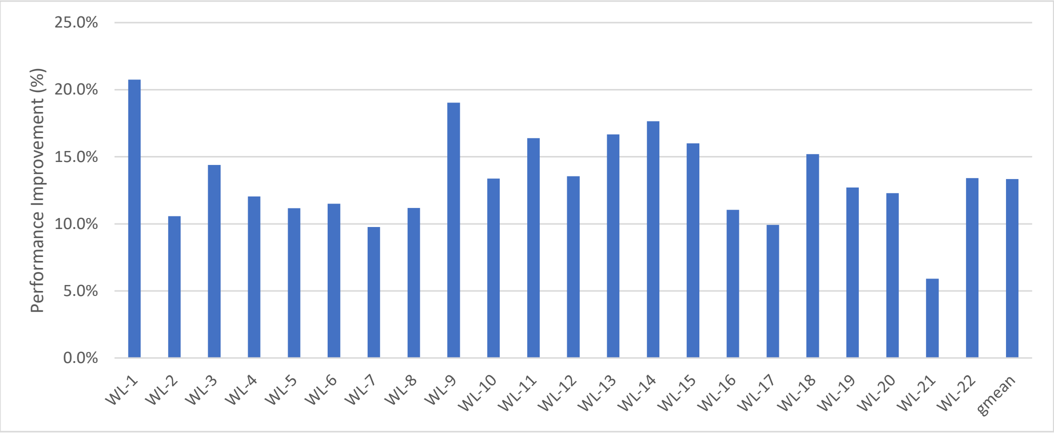


Figure 4.11: Performance improvement on 8 cores.

cores). This is because bank conflicts become even more prominent with extra contention from 8 cores.

4.7.3 Area

The tag store is in dedicated SRAM tables at the memory controller. Each tag store entry is made up of four fields: Address Tag, Valid Columns Mask, Demand Activates Counter, and the Useful bit. The width of each field is computed below:

1. **Address Tag (9 bits)** : Figure 4.2 shows that there are 9 tag bits in the 4-way set-associative scheme (d). The 9 bits are: bits 33 to 27 of the physical address (row bits 15 to 9), and bits 16 and 15 of the physical address (bank bits 1 and 0)

2. **Valid Columns Mask (128 bits)**: as the Duplicon Cache sector size is 8 columns (64B), one valid bit is needed for every 8 columns; there

are 1K columns in each line, so 128 total valid bits are required

3. **Demand Activates Counter (4 bits)**: we empirically found 15 to be a good value for the Demand Activates Counter threshold; thus we use a 4-bit saturating counter

4. **Useful Bit (1 bit)**

Each tag store entry has $9 + 128 + 4 + 1 = 142$ bits. Each tag store set has 4 ways, so there are $4 \times 142 = 568$ bits per set. Recall we index into a tag store set using the non-channel, non-column index bits. Figure 4.2 shows that are 11 such bits: bits 26 to 18 of the physical address (row bits 8 to 0), and bits 14 and 13 of the physical address (bank group bits 14 and 13). Thus there are $2^{11} = 2048$ sets in each tag store, and $2048 \times 568 = 1163264$ bits = 142KB per tag store table - i.e., 142 KB/channel (recall we maintain a tag store table per channel). Our configuration has two channels, so the total storage cost is $2 \times 142\text{KB} = 284$ KB.

4.7.4 Power/Energy

Duplicon introduces extra power in two ways:

(a) Extra leakage from the tag store, and extra dynamic power due to tag store accesses

(b) Extra DRAM power from duplication write requests

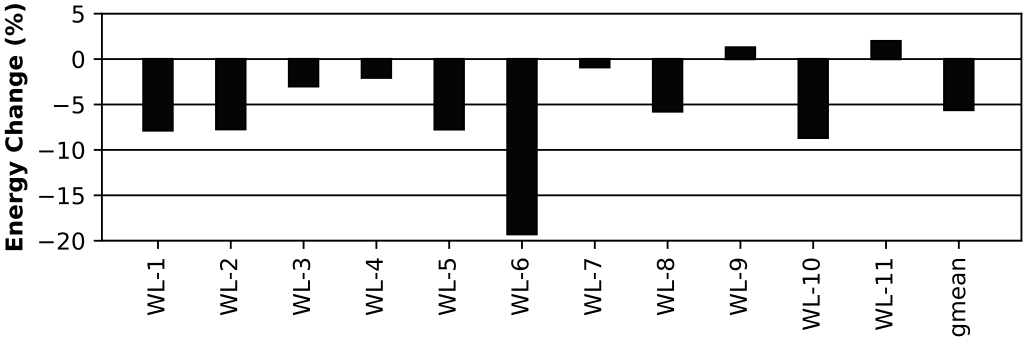


Figure 4.12: Duplicon Cache energy evaluation.

For (a), we model the tag store as a cache in McPAT, model read and write accesses to the tag store as read and write accesses to the cache, and add the power/energy contribution from the cache to the total power/energy. For (b), we account for the duplication write requests in our CACTI DRAM power and energy model.

Fig. 4.12 shows the energy results. Duplicon reduces the total energy on 9 out of 11 workloads, reducing the total energy by 5.6% on average. The energy savings come from reducing the workload execution time.

Chapter 5

Continuous Row Compaction

This chapter presents Continuous Row Compaction. Many of the high level motivations, insights, and trade-offs for Continuous Row Compaction have been steadily introduced in chapters 1, 2, and 3. This chapter summarizes these motivations, insights, and trade-offs, and provides additional implementation details and evaluation results.

5.1 Overview

Section 2.1 made the case that the key to achieving good performance with modern DRAM devices is to interleave accesses to different banks and, if possible, to different bank groups, while avoiding row conflicts that access data from different rows of the same bank. Duplicon Cache dealt with increasing interleaving across different banks and bank groups. Continuous Row Compaction now deals with avoiding row conflicts.

This chapter begins with a motivational example in section 5.2 showing how a class of computations, called stencil computations, creates multiple concurrent memory access streams that can conflict with one another, especially when crossing virtual-to-physical address translation boundaries. I then motivate how Continuous Row Compaction helps eliminate such conflicts by migrating data accessed together to non-conflicting row buffers.

The subsequent four sections then address the four main challenges to making Continuous Row Compaction work:

(I) How do we minimize the extra data movement overhead from data migration?

(II) How do we identify the most suitable data for migration? In particular, how do we identify temporally correlated data that should be migrated/compacted together?

(III) How do we efficiently track what has been migrated/compacted, and to where?

(IV) How do we ensure coherence and correctness of data in light of migration?

To minimize the extra data movement overhead from data migration, Continuous Row Compaction explicitly controls the rate at which migrations can occur. This mechanism, Explicit Copying Throttling, was extensively discussed in section 3.4.4.1, and briefly summarized again in section 5.3.

To identify temporally correlated and frequently accessed data most suitable for migration, Continuous Row Compaction records sequences of data accesses at 4KB granularity, monitors the sequences to see which one is most frequently accessed, then selects the most frequently accessed sequence for migration. This process, Candidate Sequence Identification, is explained in section 5.4.

To efficiently tracking what has been migrated/compacted (and to where), the Continuous Row Compaction tag store, called the Remap Table, is organized as a set-associative non-sectored tag store with 4KB line size. By removing the need for sectoring bits, the Continuous Row Compaction tag store is significantly more storage efficient than the Duplicon Cache tag store, even while employing a smaller line size. This is shown in section 5.5.

Section 5.6 goes over details on how Continuous Row Compaction ensures coherence and correctness of data while performing data migrations.

Section 5.7 explains the evaluation methodology and presents the evaluation results.

5.2 Motivation

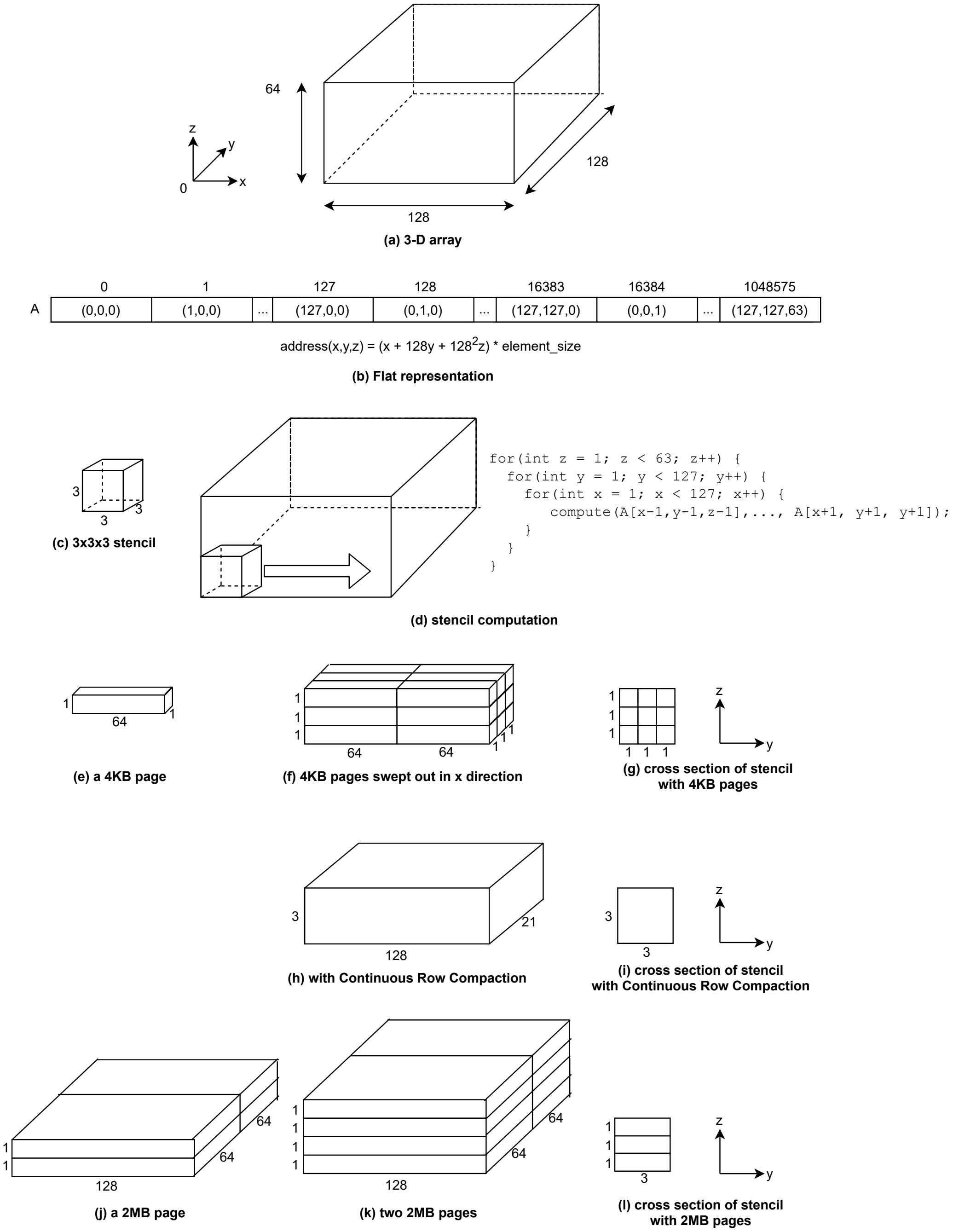
I motivate Continuous Row Compaction with a stencil computation example that shows how Continuous Row Compaction can substantially reduce the number of memory row conflicts in the presence of multiple concurrent memory access streams.

Stencil computations are frequently found in scientific workloads. For example, Lattice Boltzmann Method (lbm) [50] is a stencil based fluid dynamics simulation algorithm that is part of SPEC 2017.

Lbm performs an iterative stencil computation on a 3-D array. The array, with dimensions 128 x 128 x 64 for our example, is shown in Figure 5.1(a). In physical memory the 3-D array is stored as a flat 1-D array, which I label A in Figure 5.1(b). Elements of the 3-D array are stored in the flat array first sorted by their z index, then by y, then by x, as shown in Figure 5.1(b).

The stencil computation iterates through each element in the 3-D array, examining the neighboring elements of the current element in each iteration to perform a computation. The shape of the stencil specifies which neighbors are examined during each iteration. In our example we have a 3x3x3 stencil, shown in Figure 5.1(c). Thus for each element in the 3-D array, we examine its 26 neighboring elements: 9 neighbors above, 9 neighbors below, and then 8 additional neighbors on the same plane. Figure 5.1(d) shows the stencil computation, which is a triple-nested for loop that iterates through the x, y, and z dimensions. The innermost loop body examines the 26 neighbors plus the current element, and then performs some computation. Visually, the 3x3x3 stencil sweeps across the 3-D array in the x dimension first, and then in the y dimension, then in the z dimension, as shown in Figure 5.1(d).

As the stencil moves across the 3-D array, it creates many concurrent memory access streams that potentially access different rows of the same bank, creating row conflicts. To see this, assume virtual-to-physical translation occurs at 4KB granularity. Assuming the array is 4KB aligned, and assuming each array element is 64B, then the array can be divided into 4KB pages, each with 64 elements. In the 3-D array, every aligned 64x1x1 block of data, shown in Figure 5.1(e), represents an unique 4KB page. Accesses from within the



64
z
y
0
x
128
128
(a) 3-D array

	0	1		127	128		16383	16384		1048575
A	(0,0,0)	(1,0,0)	...	(127,0,0)	(0,1,0)	...	(127,127,0)	(0,0,1)	...	(127,127,63)

address(x,y,z) = (x + 128y + 128^2z) * element_size

(b) Flat representation
3
3
3
(c) 3x3x3 stencil

```
for(int z = 1; z < 63; z++) {
  for(int y = 1; y < 127; y++) {
    for(int x = 1; x < 127; x++) {
       compute(A[x-1,y-1,z-1],..., A[x+1, y+1, y+1]);
    }
  }
}
```

(d) stencil computation
1
64
1
(e) a 4KB page
1
1
1
64
64
1
1
1
(f) 4KB pages swept out in x direction
1
1
1
1 1 1
z
y
(g) cross section of stencil with 4KB pages
3
128
21
(h) with Continuous Row Compaction
3
3
z
y
(i) cross section of stencil with Continuous Row Compaction
1
1
128
64
64
(j) a 2MB page
1
1
1
1
128
64
64
(k) two 2MB pages
1
1
1
3
z
y
(l) cross section of stencil with 2MB pages

Figure 5.1: Motivating stencil computation.

same 4KB page are guaranteed to not cause row conflicts, because such accesses share the same high order physical address bits that correspond to the DRAM row address, assuming physical-to-DRAM mappings like those in Figure 2.4. Accesses with the same DRAM row access cannot cause row conflicts - they either access the same row of a bank, or go to different banks.

However, accesses from different 4KB pages are not guaranteed to share the same DRAM row address, and as the 3x3x3 stencil sweeps across the 3-D array, it will touch data from many different 4KB pages. For example, Figure 5.1(f) shows that, in the course of performing a full sweep in the x direction (i.e., performing a full execution of the innermost for loop), the stencil touches data from 18 different unique 4KB pages. Even a single iteration in the x-direction (i.e. each execution of the function compute(A[x-1,y-1,z-1],..., A[x+1, y+1, y+1]) in Figure 5.1(d)) accesses data from at least 9 different 4KB pages, shown by the cross section of the stencil in Figure 5.1(g). The more streams concurrently accessing different 4KB pages, the more likely accesses with different row addresses will end up colliding at the same bank.

In contrast, with Continuous Row Compaction, as different 4KB regions are encountered, they get grouped together and migrated/compacted to reserved compacted regions of physical memory under the same row address. This is shown in Figure 5.1(h). As the stencil sweeps across the x-direction in the 3-D array, different 4KB regions encountered get migrated to a compacted region with the same row address. This results in a 128x21x3 compacted region formed by migrating the 126 encountered 4KB regions to the same reserved

DRAM row address. Array accesses generated when the stencil is within this compacted region are then guaranteed to not conflict with one another. This is shown in Figure 5.1(i), which shows that as the stencil performs a full sweep in the x-direction within the compacted region, it will access data from only a single row address, in contrast with 9 in Figure 5.1(g).

In essence, Continuous Row Compaction, based on the dynamic memory access order, rearranges the data layout in physical memory to try and fit data that is frequently accessed together into non-conflicting row buffers across different channels/ranks/banks under the same row address. It is essentially, at runtime, performing data blocking for row buffers distributed across different channels/ranks/banks.

While smaller virtual-to-physical translation granularities (e.g. 4KB) exacerbate the problem of concurrent memory accesses to different memory regions with different row addresses, the problem still exists with larger translation granularities, or even if there was no translation. This is because fundamentally, assuming a physical-to-DRAM mapping like that of Figure 2.4, at most 512KB can fit under the same row address. Thus a 2MB page will still straddle four row addresses, each spanning 512KB. We show this in Figure 5.1(j). Due to the ordering in which the 3-D array is stored into flat memory, a 2MB page, assuming the array is 2MB aligned, represents a 128x128x2 block in the 3-D array (i.e., two complete x-y planes stacked on top of another in the z dimension). Each 2MB page spans 4 row addresses, with each row address occupying a 512KB 128x64x1 block. Figure 5.1(k) shows two 2MB pages stacked on top of one another. As the stencil sweeps across these two 2MB pages in the x dimension, it will concurrently access data from, in the best case, 3 different row addresses, shown in Figure 5.1(l). Hence the problem with concurrent memory accesses to regions with different row addresses still exists.

We note that on-chip SRAM caching can reduce the actual number of concurrent DRAM access streams to different row addresses, since part of the array might already be cached in the on-chip SRAM caches, removing the need for DRAM accesses. Nonetheless, on-chip SRAM caching will not eliminate all concurrent DRAM access streams to different row addresses, so Continuous Row Compaction can still provide benefit. We also note that the compacted regions created by Continuous Row Compaction are not necessarily ideal, and one can still end up with concurrent streams to different row addresses even with after row compaction. However, the number of such cases will be reduced compared to the baseline.

There are four main challenges to making Continuous Row Compaction work:

5.3 Challenge (I): Minimizing Data Movement Overhead

5.3.1 Explicit Copying Throttling

Migrating and compacting data to reserved compacted regions introduces additional memory traffic overhead. By carefully choosing the granularity and frequency of compaction, we can keep this overhead in check. This process, which I call Explicit Copying Throttling, was explained in detail in section 3.4.4.1, and briefly summarized here again.

The natural choice for compaction granularity is the amount of data that can be compacted under a single DRAM row address. If we assume the memory setup from Figure 2.4, since the row address starts at bit 19 of the physical address, this means 2^{19}B = 512KB of data can be compacted under the same row address.

Compacted data can become stale over time. We periodically examine previously compacted data and, when appropriate, replace them with newly identified temporally correlated data. This is the continuous aspect of Continuous Row Compaction. We propose a simple scheme where a Replacement Pointer points to the next set of compacted data to be potentially replaced. Once the data has been examined, it will either be replaced with newly identified temporally correlated data, or it will remain. In either case, the pointer then advances to the next set of compacted data. Over time, the pointer wraps around. In this manner, any stale compacted data will eventually be replaced.

Each 512KB row compaction requires transferring up to 2MB (4 × 512KB) of data:

1. reading previously compacted data from the reserved compacted region (512KB)
2. writing previously compacted data back to their original frames (512KB)

3. reading newly identified temporally correlated data from their original frames (512KB)

4. writing newly identified temporally correlated data to the reserved compacted region (512KB)

Transferring 2MB of data over two DDR4 channels requires 64K bus cycles. We can cap this overhead by choosing an appropriate compaction frequency. For example, if we perform one compaction every 3200K bus cycles, then the bandwidth overhead from compaction will be capped at 64K / 3200K = 2%. Increasing the interval between compactions reduces the compaction bandwidth overhead, but also increases the time which stale compacted data remains in the compacted region.

5.4 Challenge (II): Identifying the Most Suitable Data for Duplication

5.4.1 Candidate Sequence Identification

To identify temporally correlated data for compaction, we record the sequences in which uncompacted 4KB frames are accessed in time. Each captured sequence, which we call a Candidate Sequence, contains up to 128 unique uncompacted 4KB frames. The number 128 comes from the granularity of row compaction, which is 128 × 4KB = 512KB, the amount of data that can fit under a single row address, assuming the memory setup in Figure 2.4. Multiple Candidate Sequences are captured during the interval between row compactions (i.e., cache replacements). At the end of the interval, we pick the Candidate Sequence with the most recorded accesses, and compact its constituent 128 4KB frames.

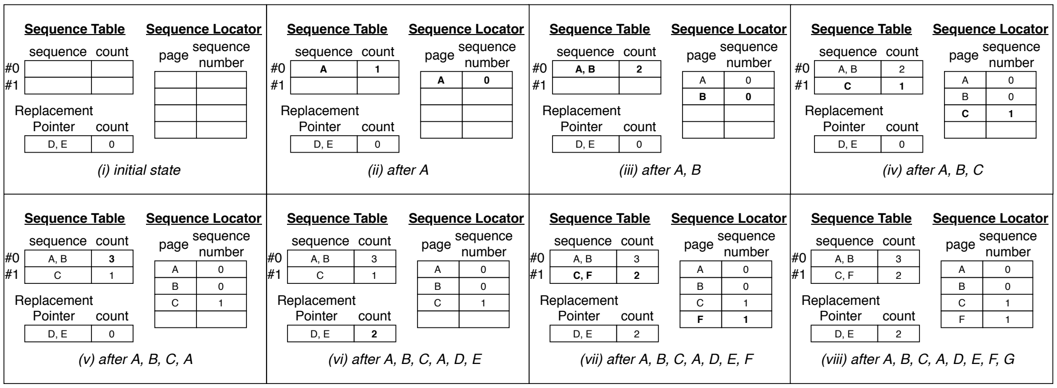


Figure 5.2: Candidate Sequence Identification.

Figure 5.2 illustrates the Candidate Sequence Identification process. To simplify the example, the figure assumes a row compaction granularity of 2 × 4KB frames, rather than 128 × 4KB frames. Three data structures are needed during Candidate Sequence Identification:

1. The Sequence Table, which records the actual Candidate Sequences, along with how many times each Candidate Sequence has been accessed.
2. The Sequence Locator, which records, for each uncompacted 4KB frame encountered, which Candidate Sequence it has been included in.
3. The Replacement Pointer, which points to the reserved row address to be replaced next. Previously compacted frames under that reserved row

address will then be overwritten if replacement occurs.

In this example we assume the 4KB frames D and E have previously been compacted, and that our Replacement Pointer is pointing to D and E. Our goal is to identify two new 4KB frames to replace D and E with.

Initially, both the Sequence Table and Sequence Locator are empty. Figure 5.2(i) shows the initial state of all relevant data structures. Over the course of this example, the processor issues requests to 4KB frames in the order A, B, C, A, D, E, F, G, where each upper case letter denotes an unique 4KB frame. Note that the actual offset within each accessed 4KB frame is unimportant for Candidate Sequence Identification.

Upon receiving the first request to frame A, we see that A has not been previously compacted,[1] and has not been included in any Candidate Sequence thus far - i.e., the Sequence Locator has no entry for A. Furthermore, the Sequence Table is empty, so we start a new sequence with A in entry 0 of the Sequence Table, set the access counter to 1, and update the Sequence Locator to show that A is now part of Candidate Sequence 0. This is shown in Figure 5.2(ii).

Next, we receive a request to frame B. We again see that B has not been previously compacted, and is not part of any existing Candidate Sequence. We then append B to Candidate Sequence 0, increment its access count, and

[1] we can know A has not been previously compacted by looking it up in the Remap Table, which we discuss in section 5.5.1

update the Sequence Locator. This is shown in Figure 5.2(iii).

Next, we receive a request to frame C. C has not been previously compacted and is not part of any existing Candidate Sequence. However, in this case Candidate Sequence 0 is already full (recall the granularity of compaction is $2 \times$ 4KB frames in this example, so each Candidate Sequence only tracks up to 2 unique 4KB frames), so the request to C starts a new sequence, Candidate Sequence 1, and we increment its access count and update the Sequence Locator accordingly, shown in Figure 5.2(iv).

The next request is to frame A. We can detect that A already belongs to Candidate Sequence 0 by checking the Sequence Locator table and finding an entry for frame A. Thus we simply increment the access count for Candidate Sequence 0 from 2 to 3. No other updates are necessary. This is shown in Figure 5.2(v).

The next two requests, D and E, are to previously compacted frames pointed to by the Replacement Pointer. We increment the corresponding access count for these previously compacted frames, and perform no other updates. This is shown in Figure 5.2(vi). At the end of each Candidate Sequence Identification phase, we will compare the highest access count among all the identified Candidate Sequences against the access count of the previously compacted frames pointed to by the Replacement Pointer, and only replace the previously compacted frames if the access count is greater.

Next, we receive a request to frame F. It is uncompacted and not part of an existing Candidate Sequence, so we append it to Candidate Sequence 1, increment its access count, and update the Sequence Locator. This is shown in Figure 5.2(vii).

At this point, both the Sequence Table and Sequence Locator are full, and have no more space for new Candidate Sequences. When we receive the next request to frame G, we do nothing, as shown in Figure 5.2(viii). However, additional accesses to previously compacted frames pointed to by the Replacement Pointer (e.g., D and E), or frames that are part of an existing Candidate Sequence (e.g., A, B, C, and F), will continue to increment the appropriate access counter. In Section 5.7.8 we discuss sizing the Sequence Table and Sequence Locator appropriately to capture the best Candidate Sequence while maintaining reasonable hardware costs.

At the conclusion of the Candidate Sequence Identification phase, we take the Candidate Sequence with the most accesses (in this case, Candidate Sequence 0 with 3 accesses), and compare it to the previously compacted sequence pointed to by the Replacement Pointer (D and E, with 2 accesses). If this Candidate Sequence has more accesses, as is the case in this example, then we will replace the previously compacted sequence with the new Candidate Sequence.

In a multi-core environment, in the absence of data sharing, all true temporal correlation exists solely within individual processes each running on its own core. Conversely, many false temporal correlations will show up between accesses from completely unrelated processes running on different cores. Hence we partition the Candidate Sequence Identification process by core. The Sequence Table and Sequence Locator are replicated, with one copy per core, and DRAM requests are processed by the appropriate Sequence Table/Sequence Locator based on the request's core of origin. Candidate Sequences are formed and tracked independently for each core. At the conclusion of Candidate Sequence Identification, the Candidate Sequence with the most accesses across all the cores is compared against the previously compacted data pointed to by the Replacement Pointer, and replacement occurs if the candidate access count is greater.

While per-core Candidate Sequence Identification is optimized for cases where there is minimal data sharing between cores, it will not break in the presence of data sharing. For example, if a frame is shared and accessed by multiple cores, it will potentially be included in multiple Candidate Sequences across multiple cores, but ultimately only one of those Candidate Sequences will be chosen for compaction. Thus per-core Candidate Sequence Identification will function correctly and still provide benefit in the presence of data sharing across cores (e.g., with multi-threaded applications). We leave the optimization of Candidate Sequence Identification for multi-threaded applications as future work.

An OS context switch may occur in the middle of Candidate Sequence Identification. In this case, we may end up capturing Candidate Sequences with accesses from the original running process, the context switch handler, and the new running process. This is functionally allowed and will not cause any errors. Furthermore, such Candidate Sequences will likely have low access counts compared to others, because they capture data accesses from processes that only ran for a fraction of the Candidate Sequence Identification phase. As such, they are unlikely to be selected for compaction.

Finally, with SMT, the access stream from each individual core may itself be composed of accesses from multiple unrelated processes. We leave the optimization of Candidate Sequence Identification in the presence of SMT as future work.

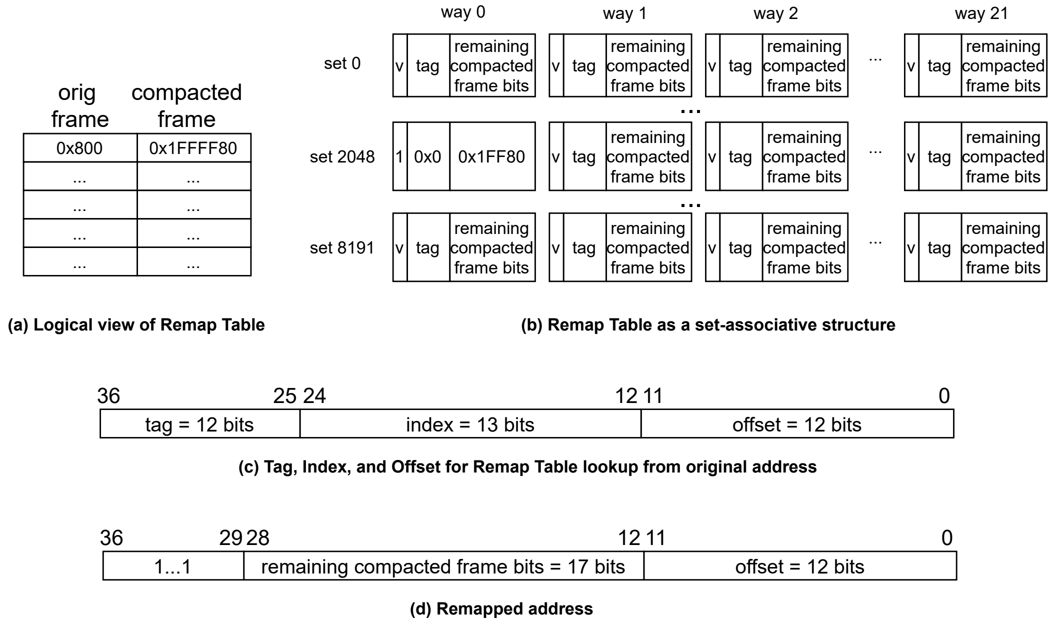
5.5 Challenge (III): Minimizing Tag Store Overhead

5.5.1 Remap Table Organization

After a frame has been compacted (i.e., migrated), we require an additional layer of address translation to map the compacted frame from its original location to its compacted location. This is done via the Remap Table. Logically, the Remap Table is a lookup table accessed with the original frame address, and outputs the address of the corresponding compacted frame, if it exists. This is shown in Figure 5.3(a). Here, frame 0x800 has been remapped (i.e., migrated) to reserved frame 0x1FFFF80.

The Remap Table can be physically implemented as a set-associative structure. This is shown in Figure 5.3(b), as a structure with 8K sets and 22 ways.

To access the Remap Table structure, the tag, index, and offset are



(a) Logical view of Remap Table

(b) Remap Table as a set-associative structure

(c) Tag, Index, and Offset for Remap Table lookup from original address

(d) Remapped address

Figure 5.3: Remap Table.

taken from the original memory address, as shown in Figure 5.3(c). The low 12 bits are the offset into the frame, which remain unchanged even after the frame has been compacted/migrated. The next 13 bits are used to index into one of the 8K sets of the Remap Table structure. The remaining 12 bits are tag. For example, if the original memory address was 0x800140, then the tag would be 0x0, the index 0x800, and the offset x140. With this address, we find a hit in set 2048, way 0.

Inside way 0, the value 0x1FF80 is stored. This is the corresponding compacted frame address, but with the high order 1s removed. Recall that Continuous Row Compaction, like Duplicon Cache, reserves physical memory from the top of the physical address (section 3.2.1), and all compacted frames come from this top end of the physical address space. If the top 2^k bytes of physical memory is reserved out of 2^m bytes of total memory, then the address bits $[m-1{:}k]$ of the reserved region will always be 1. Hence it is unnecessary to store those bits of the compacted frame address. In this example, I assume $k = 29$ and $2^{29} = 512$MB of memory are reserved for Continuous Row Compaction.

Figure 5.3(d) shows the final remapped address after the Remap Table has been consulted. The high 8 bits are set to 1, then concatenated with the remaining compacted frame bits read out from the Remap Table (0x1FF80 in our example), then concatenated with the original offset (0x140). The end result is that the original address 0x800140 is remapped to 0x1FFFF80140.

Each way in the set-associative Remap Table structure requires:

- 1 valid bit
- 12 bits of tag
- 17 bits for the remaining compacted frame bits

for a total of 30 bits. Hence each compacted/migrated 4KB frame only requires 30 bits of storage to track. In contrast, if we had gone for a sectored approach like with Duplicon Cache, then, assuming 64B sectors, we would require at least 64 sector valid bits to track a 4KB frame. This is more than double the storage requirement.

Since we reserved 512MB of memory for compaction, there are a total of 512MB/4KB = 128K possible 4KB compacted frames. Hence, the Remap Table needs to be sized to at least be able to hold 128K entries. With 8K sets, this can be achieved with an associativity of 16. However, in practice the structure should have a little higher associativity than necessary to prevent set conflicts. I empirically found that 22 ways virtually removed all set conflicts in my evaluation.

Hence the total Remap Table storage cost is: 8K sets × 22 ways/set × 30 bits/way = 660KB. In section 5.7.8 I compare against the baseline with an equivalent amount of cache added.

5.6 Challenge (IV): Ensuring Data Coherence and Correctness

5.6.1 Tracking partially written back/filled frames

The main challenge for ensuring data coherence and correctness with Continuous Row Compaction arises when frames are replaced, as frames may be partially written back or filled. To address this, we track fine grained valid and dirty status for 64B sectors of frames in the midst of being written back or filled. 64B is the granularity of tracking, as this is the default transaction size for DDR4.

This is done via two new structures, the Current Writeback Tracker and the Current Fill Tracker. Each Tracker has four components:

- orig frame: the original frame address

- compacted frame: the compacted frame address
- status: an array that tracks the status of each 64B sector within the frame. Each sector is in one of three states:
  - Uninitialized (U): the 64B sector has not been read.
  - Read (R): the 64B sector has been read, but not written to its final destination yet.
  - Written (W): the 64B sector has been read, and written to its final destination.
- data buffer: an array that holds the current value of 64B sectors that have been read, but not yet written to its final destination.

The two Trackers are shown in Figure 5.4. For brevity, we only show eight status and data buffer entries per Tracker, when in reality there will be 4KB / 64B = 64 such entries.

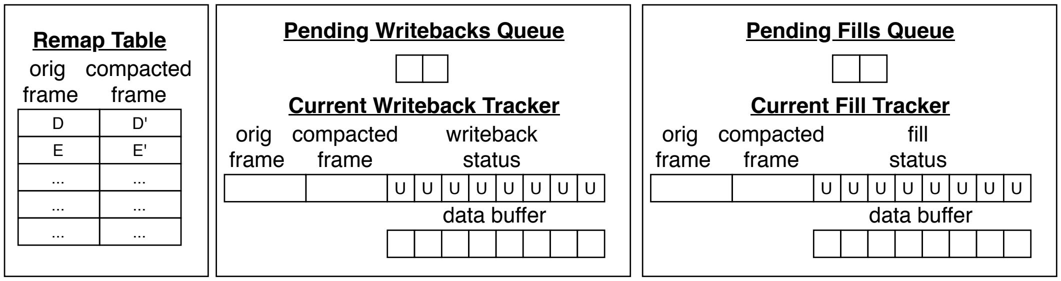


Figure 5.4: Initial state of tracking structures, before replacement.

Two additional structures are needed. The Pending Writebacks Queue is a queue that identifies pending frames that need to be written back to their original locations. Similarly, the Pending Fills Queue identifies pending frames that need to be migrated from their original locations to compacted locations.

Initially, the Pending Writebacks Queue, Current Writeback Tracker, Pending Fills Queue, and Current Fill Tracker are all empty. The Remap Table holds two entries, remapping frames D to D', and E to E', respectively. This is shown in Figure 5.4.

At the end of the Candidate Sequence Identification example (section 5.4), we identified that frames D and E should be replaced with A and B. This requires writing the contents of frames D and E back from their compacted locations to their original locations, and then copying the contents of A and B from their original locations to their compacted locations. We now show how these writeback and fill operations are performed.

The first step in replacing D and E with A and B is to add D and E to the Pending Writebacks Queue, and A and B to the Pending Fills Queue. This is shown in Figure 5.5. Note at this point, nothing has yet been moved in memory, so requests to D and E will still be remapped to D' and E', and requests to A and B will still go to A and B.

Next, we dequeue D from the Pending Writebacks Queue, and begin writing back D to its original location. The Current Writeback Tracker is updated to reflect that we are in the midst of writing D back. Initially, all of

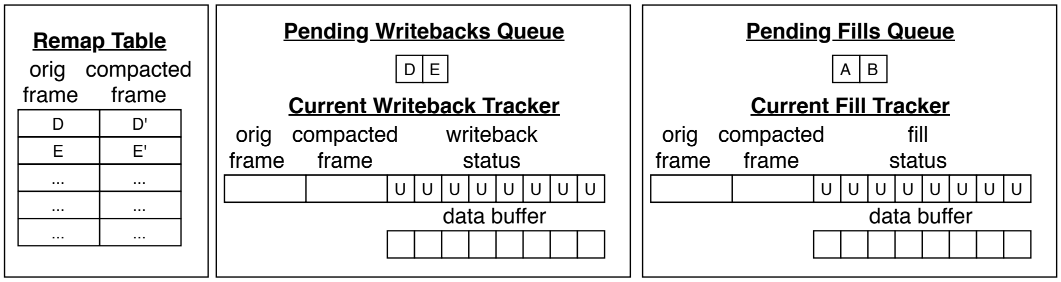


Figure 5.5: After enqueuing into the Pending Writebacks Queue and Pending Fills Queue

the 64B sectors of D are in the Uninitialized state, as denoted by U. At this point we remove the entry for D from the Remap Table. From this point on, how requests to D are serviced depends on the status of the individual 64B sector being requested. If the requested sector is in Uninitialized state, then it is serviced from the compacted location (i.e., D'). If the sector is in the Read (R) state, it is serviced from the corresponding data buffer in the Current Writeback Tracker. If the sector is in the Written state (W), it is serviced from its original location (i.e., D). The state of the tracking structures as we begin writeback of D is shown in Figure 5.6.

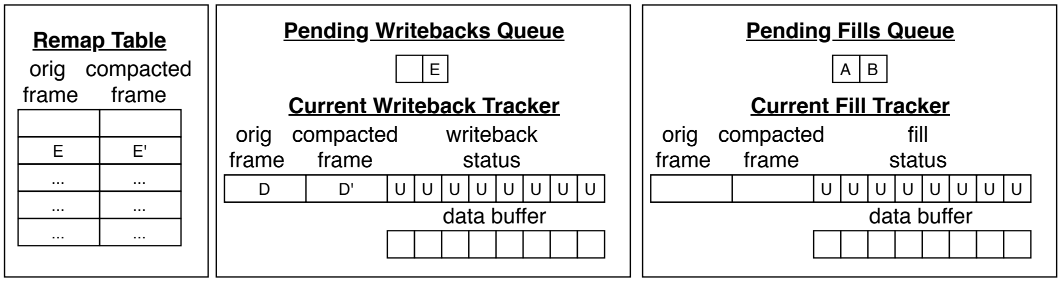


Figure 5.6: Beginning the writeback of D.

We go ahead and issue fetch requests for all the constituent 64B sectors of D from their compacted locations (i.e., from D'). As we do so, we update the status of the sectors to Read (R). As individual fetch requests complete, the data read is placed in the corresponding data buffer. Each sector data buffer is guarded by a valid bit, which is initially cleared to denote that the buffer is empty. As the fetch requests for different sectors complete, they set the valid bit of the corresponding data buffer. Fetch requests are processed by the memory controller no differently than from ordinary read requests, with the only distinction being that fetch requests originate at the memory controller instead of the cores.

While a 64B sector is in the Read state, other ordinary (i.e., non-fetch) requests to the sector must be serviced through the data buffer. For read requests, the data can be immediately serviced from the data buffer if it is valid. If the data buffer is not valid, the read request must wait until the pending fetch request completes.[2] Write requests to sectors in Read state write directly into the corresponding data buffer, and set the valid bit. If a fetch request completes after another write request has already filled the corresponding data buffer, then the fetch request is dropped, as the data read is already stale.

Figure 5.7 shows the state of the Current Writeback Tracker after all fetch requests have been issued. Some fetch requests have completed, as ev-

[2] If the sector is in Read state but the data buffer is not valid, there must be a pending fetch request

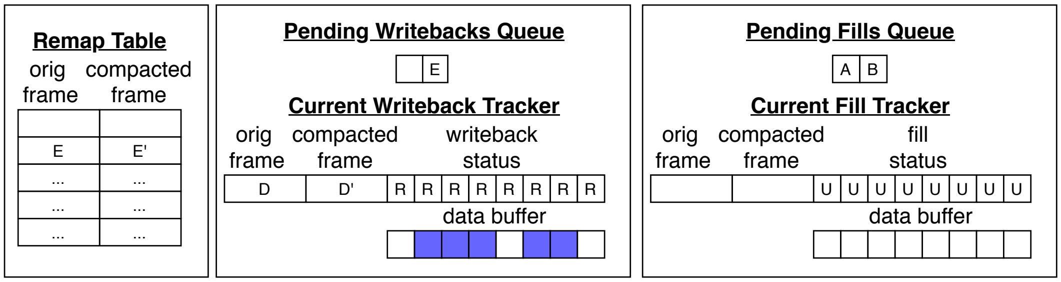


Figure 5.7: All fetch requests issued, and some have returned.

idenced by the corresponding data buffers being valid, shown in the figure with the corresponding data buffer box being filled in (alternatively, write requests could have made the data buffers valid). 64B sectors in Read state with a ready data buffer are ready to be written back to their original locations. We go ahead and issue writeback requests for these 64B sectors. Writeback requests are no different from ordinary write requests, except that they are generated at the memory controller for writing back previously compacted data. Upon issuing a writeback request, the sector immediately changes state to Written (W). Once the sector is in Written state, any future access to the sector will be directed to the original location (i.e., D). Note it is possible for a writeback request to still be pending when another read request for the same 64B sector arrives. However, this sequence of events, where we issue a write request to a location, and later receive a read request to the same location before the write request completes, can just as easily happen during normal DRAM operations, and we assume the baseline memory controller is able to handle it. Figure 5.8 shows the state where some sectors have updated to

Written state. Note as a sector gets updated to Written state, its corresponding data buffer becomes invalid, because now the data will be served by the sector's original location in DRAM.

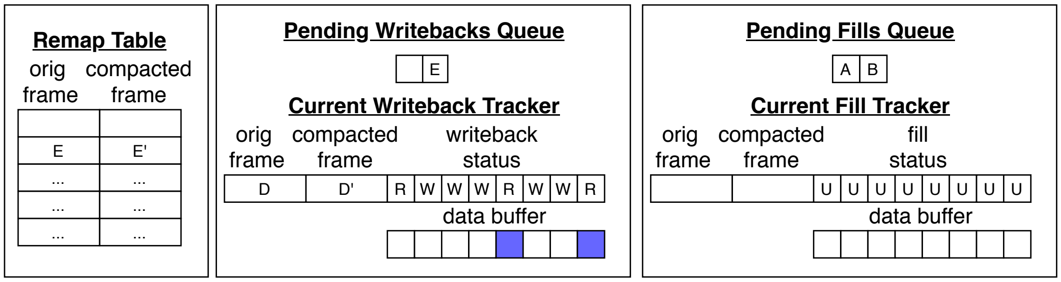


Figure 5.8: Some writeback have completed.

Once all the 64B sectors reach the Written state, the writeback of D is complete. At this point, two things happen:

1. The compacted frame D' is now available for another frame (A) to be compacted into. At this point, A is dequeued from the Pending Fills Queue, and populates the Current Fill Tracker.

2. We can start writeback of the next frame (E). E is dequeued from the Pending Writebacks Queue, and populates the Current Writeback Tracker. The entry for E, previously mapping E to E', is removed from the Remap Table.

These changes are reflected in Figure 5.9. At this point, we are ready to start issuing fetch requests both for the writeback of E, and the filling (i.e., copying into a compacted frame) of A. While fetch requests for a frame being

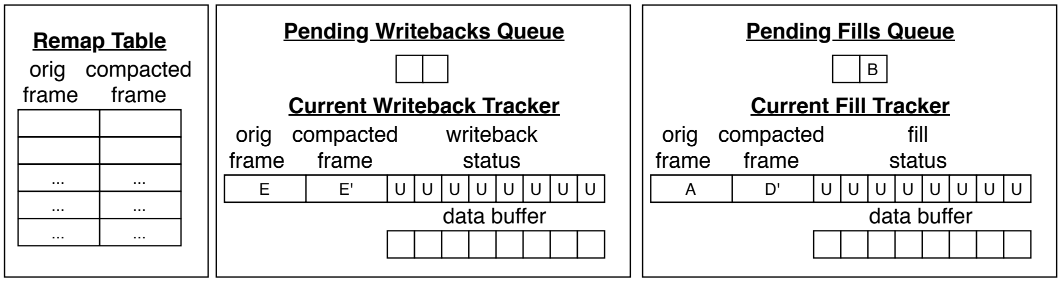


Figure 5.9: Beginning to fill A into compacted frame D', and writing back of E from compacted frame E'.

written back fetch from the compacted frame (E' in this case), fetch requests for a frame being filled fetch from the frame's original location (A in this case). Similarly, the location from/to which requests to a frame in the midst of being filled gets serviced depends on the status of the 64B sector being requested. If the sector is in Uninitiated (U) state, then the sector is served from the original location (A in this case). If the sector is in Read (R) state, it is serviced from the corresponding data buffer in the Current Fill Tracker. As before, read requests to 64B sectors in Read state can only be serviced once the data buffer is valid. A sector transitions to the Read state as soon as a fetch request is issued, but the corresponding data buffer remains invalid until either (i) the fetch request completes reading the data, or (ii) another write request to the sector writes the data. In the latter case, if a write request populates the data buffer before the fetch request completes, the data read by the fetch request is dropped, since it is already stale.

The writeback of E and filling of A proceed concurrently. When the filling of A completes, the entry mapping A to D' is added to the Remap Table. When the writeback of E completes, the filling of B into E' can begin. This is shown in Figure 5.10. When the filling of B into E' completes, the Remap Table is updated with a new B to E' mapping, and the process is complete. This is shown in Figure 5.11.

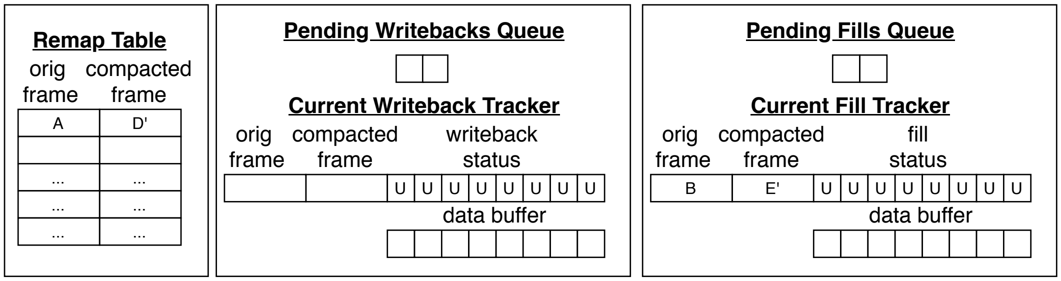


Figure 5.10: A is remapped to D' in the Remap Table, and the filling of B into E' begins.

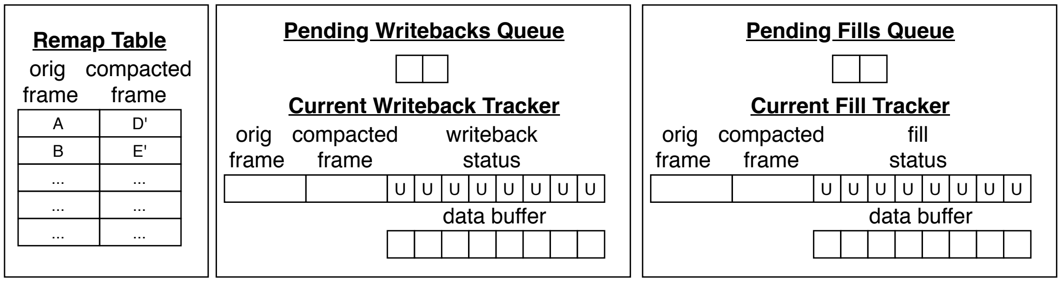


Figure 5.11: Replacement complete.

We note that the time interval between row replacements is on the order of millions of DRAM cycles (e.g., 3200K DRAM cycles), so the writeback and filling of frames can happen gradually in concurrence with other regular DRAM requests.

We also note the initial step of populating the Pending Writebacks Queue (with D and E) requires knowing which frames are pointed to by the Replacement Pointer. One way to obtain this information is to have an additional data structure that maps reserved row addresses to the original frame addresses of compacted frames. However, such a structure would be quite expensive. Instead, since the time interval between row replacements is very long, we can slowly examine every Remap Table entry over the course of many cycles, and insert frames whose compacted location row address match the Replacement Pointer into the Pending Writebacks Queue.

5.7 Evaluation

We implemented and evaluated Continuous Row Compaction on the execution-driven, cycle-level core simulator Scarab[36] and modelled DRAM via Ramulator[25]. We warmup for 1 billion (4/8 core evaluations) or 8 billion (single core evaluations) instructions, then perform detailed simulation for 200 million instructions on representative SimPoints.

5.7.1 Benchmarks

We evaluated on the SPECrate 2017 benchmark suite. For each benchmark, we used SimPoint[46] to generate representative regions. We included all representative regions with a weight greater than 5%.

We then selected the eleven most memory intensive benchmarks from the SPECrate 2017 suite. These eleven benchmarks, along with their single core DRAM accesses per kilo instructions on the baseline configuration,[3] are shown in Table 5.1.

Table 5.1: Eleven most memory intensive SPECrate 2017 benchmarks.

benchmark	DRAM accesses per kilo instruction
roms	102.37
bwaves	56.50
lbm	48.98
parest	31.92
wrf	24.57
cam4	23.76
mcf	23.26
omnetpp	18.69
gcc	14.93
xalancbmk	12.68
cactuBSSN	7.27

5.7.2 Baseline Configuration

Table 5.2 details the baseline configuration.

5.7.3 Warmup Methodology

A key challenge in accurately modeling Continuous Row Compaction is the long time period required to completely fill the reserved memory with compacted data. Assuming that we only perform 1 compaction every 3.2 million DRAM cycles , then with 1K reserved row addresses, it would take 1K × 3.2 million = 3.28 billion DRAM cycles to completely fill the reserved memory. Since the DRAM runs at half the frequency as the CPU in our evaluation,[4] this is equivalent to 6.55 billion CPU cycles. Assuming an IPC of

[3] computed as a weighted average across all the SimPoints of the benchmark

[4] 3200MT/s DRAM runs at 1.6GHz, and we have a 3.2GHz CPU

Table 5.2: Evaluated configuration.

Core	1/4/8 Cores, 6-Wide, 224 Entry ROB, 97 Entry Unified RS, TAGE-SC-L Branch Predictor [56], 3.2 GHz Clock
L1	32KB Icache, 32KB Dcache, 64B Line, 2 Read Ports, 1 Write Port, 2 Cycle Latency, 4-way
L2	Private, 1MB/core, 64B Line, 1 Read Port, 1 Write Port, 14 Cycle Latency, 16-way Writeback, Non-Inclusive
Prefetcher	Stream Prefetcher [69], 16 Streams per core, Distance 64 Queue 128, Degree 4, with Feedback Directed Prefetching (FDP) [62]
Memory System	32 MSHRs per core 32 read queue, 32 write queue per channel FR-FCFS [52] with cap of max 16 row hits precharge oldest idle bank when memory controller is idle
DRAM	2 Channels, 2 Ranks/Channel, DDR4-3200, 22-22-22, 16Gb device, 256K rows, 128 GB total capacity

1, then this means it would require 6.55 billion instructions to completely fill the reserved memory. In our single core evaluations, we warmup for 8 billion instructions to completely fill the reserved memory before starting detailed simulation. During warmup, we filter right-path memory requests through the L1 and L2 cache, perform Candidate Sequence Identification based on the L2 misses, then perform compaction every 10 million instructions. 10 million instructions was chosen as the interval between compactions during warmup, as this equals the 3.2 million DRAM cycles compaction interval, assuming 1.56 IPC.

For 4-core and 8-core evaluations, we only warm up for 1 billion instructions due to simulation time constraints. In this case, since there are only 100 compaction intervals over a billion instruction warmup period (1 billion instructions/10 million instructions), but 4K(4 core)/8K(8 core) compactions required to completely fill the reserved memory, we need to perform 41/82 compactions every compaction interval to completely fill the reserved memory by the end of warmup. Thus during warmup Candidate Sequence Identification, we compact the top 41/82 Candidate Sequences every 10 million instructions, rather than just the one, as is normally done.

In section 5.7.5.8, we compare our 8 billion instructions warmup single core results with our 1 billion instructions warmup single core results. In general, we find the results between 8 billion warmup and 1 billion warmup are similar. This confirms our intuition that, given a large enough cache capacity, we can identify working sets that are stable over very long time intervals (section 3.4.5).

In addition, we always start the warmup early enough so that the detailed simulation is over the instruction interval specified by the SimPoint. For example, if the representative SimPoint is supposed to begin at 10 billion instructions into the execution, and we are performing 8 billion instruction warmup, then we will begin the warmup at 2 billion instructions into execution.

5.7.4 Modelling Virtual-to-Physical Address Translation

Unless otherwise stated, our evaluation assumes 4KB page size, and mimics virtual-to-physical address translation by computing the physical frame number as a hash of the virtual page number. This effectively models a randomized virtual-to-physical address mapping.

In section 5.7.5.4, we examine the effect when we have larger page sizes. In general, the benefit from Continuous Row Compaction diminishes with larger page sizes, because much of our benefit comes from compacting together adjacent 4KB frames that differ in DRAM row address. With larger page sizes, accesses within the larger frames are much less likely to cross DRAM row address boundaries, making row conflicts less likely. Nonetheless, Continuous Row Compaction does continue to provide benefit with larger pages on stencil based benchmarks like lbm and cactuBSSN.

We assume no virtual pages share the same frame. We assume memory requests from different cores in the same multi-programmed workload access completely disjoint sets of frames in physical memory.

5.7.5 Single Core Performance

We now present our single core performance results. In this section, the system has 2 ranks per channel, 2 channels, and Continuous Row Compaction is configured with 512MB of reserved memory (1K reserved rows) and 3.2 million DRAM cycles between compactions.

Figure 5.12 shows the single core IPC improvement of Continuous Row Compaction over the baseline. In general we improve performance for most benchmarks. Lbm is our best performing benchmark by far, and as we showed in our motivational example in section 5.2, this is due to Continuous Row Compaction being able to create regions within the 3-D array that share the same row address, allowing the stencil to sweep through the region without

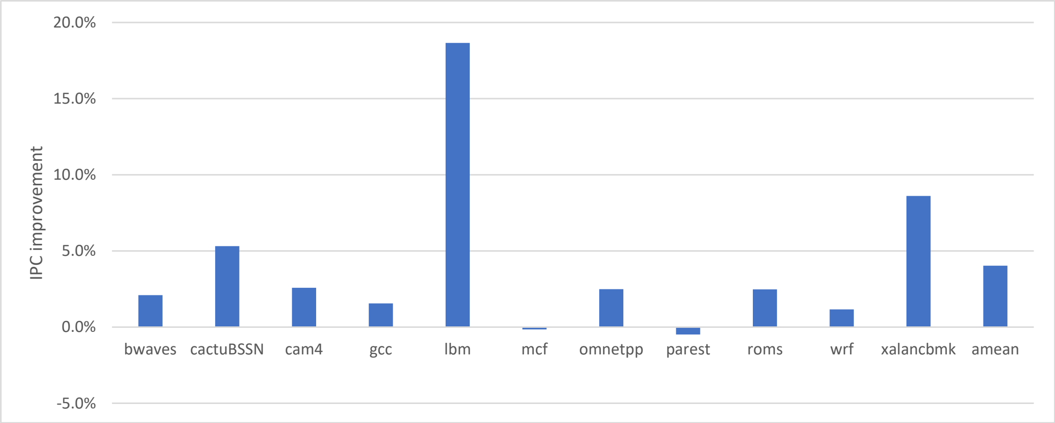


Figure 5.12: Single core IPC improvement.

mixing together accesses with different row addresses. CactuBSSN is another similar stencil based computation.

We lose performance on mcf (0.3% IPC loss) and parest (0.7% IPC loss). This is because in performing Continuous Row Compaction, we also change the order in which data is accessed by the core, and this resulted in a slight increase in the on-chip L2(which is our last level cache) eviction frequency. We discuss this in more detail in section 5.7.5.2.

5.7.5.1 Row Buffer Hit Rate

Figure 5.13 shows the DRAM Activates per kilo instructions for both the baseline and Continuous Row Compaction. We see that Continuous Row Compaction is able to improve row buffer locality and reduce the number of Activate operations required across the board. However, in some cases other bottlenecks kick in once row buffer locality is improved, limiting the performance gain. For example, both bwaves and roms were already close to saturating the channel bandwidth in the baseline; hence the performance gains from improved row buffer locality were limited. For mcf, parest, and wrf (plus cactusBSSN and cam4 to a lesser extent) the improved row buffer locality changed the order in which data was filled into the on-chip L2 (LLC) in such a way that the L2 eviction rate actually increased, leading to worse performance. We discuss this more in the next section.

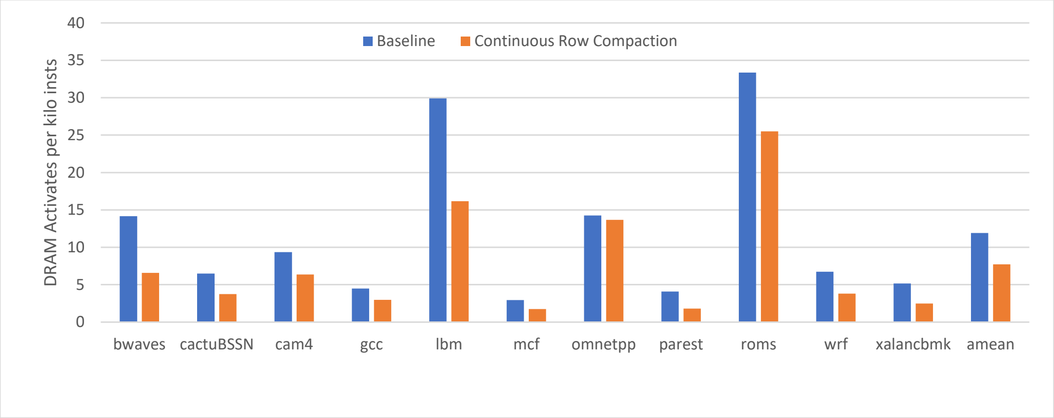


Figure 5.13: Single core DRAM Activates per kilo instructions.

5.7.5.2 Effect on On-Chip L2 Evictions

Despite improved row buffer locality (i.e., decreased frequency of DRAM Activates), two benchmarks, mcf and parest, still lose performance. I narrowed the cause to a higher eviction rate from the on-chip L2 (LLC) cache for these two benchmarks after Continuous Row Compaction. Continuous Row Compaction changes the order in which DRAM requests are serviced (by producing more row buffer hits), which can change the cache replacement behavior in the on-chip caches.

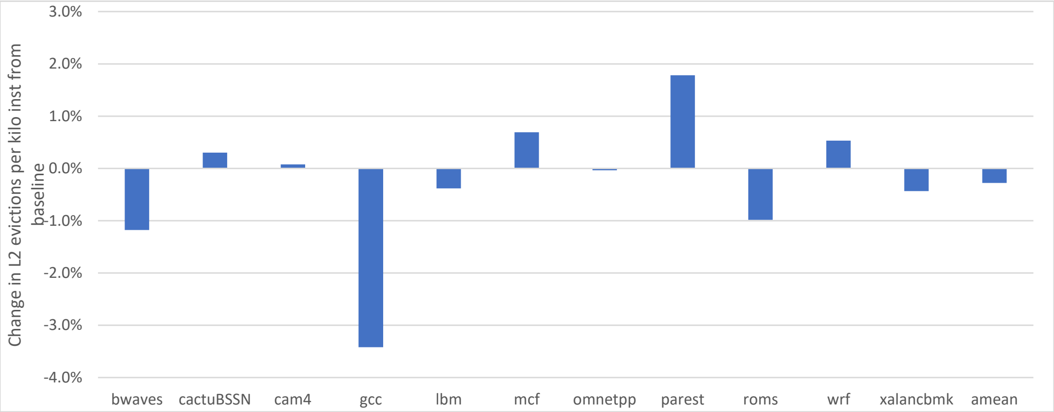


Figure 5.14: Change in L2(LLC) evictions per kilo instructions from baseline.

Figure 5.14 plots the change in L2 (LLC) evictions per kilo instructions between Continuous Row Compaction and the baseline. The L2 evictions per kilo instructions increased slightly by 0.7% and 1.8% for mcf and parest, respectively, and this was enough to negate the performance gains from better row buffer locality. The L2 eviction rate also increased for wrf (0.5%), cactuBSSN (0.3%), and cam4 (0.1%), but for these three benchmarks the improvement in performance from better row buffer locality was enough to offset the higher L2 eviction rate.

5.7.5.3 Compaction Overhead

Our Explicit Copying Throttling mechanism (sections 3.4.4.1 and 5.3), which only allows one row compaction every 3.2 million DRAM cycles, is suppose to limit the channel overhead from writebacks and fills. Figure 5.15 demonstrates its effectiveness, by examining what would happen if we made all compaction writebacks/fills free.

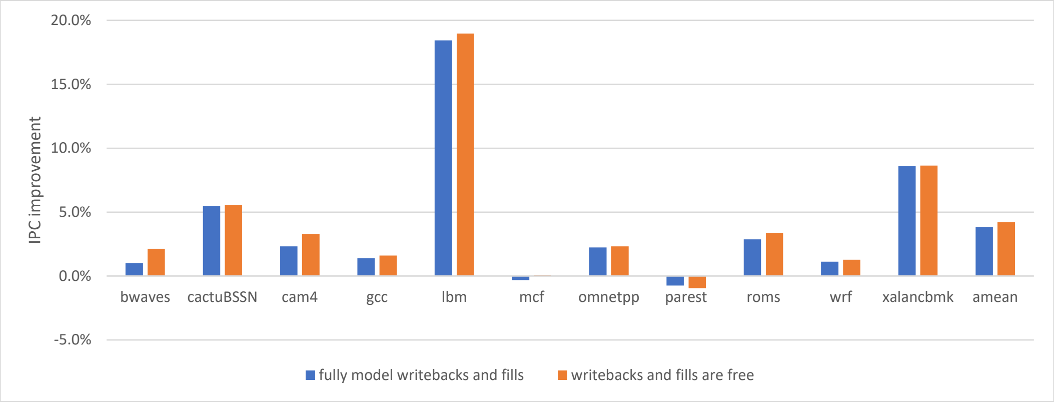


Figure 5.15: Effect on performance when writebacks and fills are made free.

We see that the performance attained with our Explicit Copying Throttling mechanism while modelling all writeback and fill overhead comes very close to the performance when we make all writebacks and fills free (i.e., do not require any DRAM operations). Thus the Explicit Copying Throttling mechanism is able to effectively limit the writeback and fill overhead, at least in the single core case. However, in section 5.7.6.2, we see that with increased channel contention in multi-core environments, writeback and fill overheads become more significant, even with Explicit Copying Throttling.

5.7.5.4 Huge Pages and Consecutive Frames

As explained in the motivational example of section 5.2 and in section 5.7.4, Continuous Row Compaction derives most of its benefit from compacting concurrently accessed 4KB frames with different row addresses into the same row address. Concurrently accessed 4KB frames are very likely to differ in row address in our simulation methodology because we model virtual-to-physical address translation by computing the physical frame address as a hash of the virtual page address (section 5.7.4), meaning contiguous virtual pages are likely to be mapped to non-contiguous physical frames that differ in the row address. However, prior work like CoLT[49] observed that on real systems contiguous virtual pages also exhibit some contiguity in physical address space, due to the effects of OS memory allocations mechanisms such as buddy allocators and memory compaction. In addition, the use of huge pages (e.g., 2MB and 1GB) would also make concurrent accesses to regions with different row addresses less likely. I thus ran additional experiments evaluating Continuous Row Compaction with 8KB, 16KB, and 2MB page sizes. In these experiments, I still migrate data at 4KB granularity, but the virtual-to-physical address hashing is done at 8KB, 16KB and 2MB granularity, respectively, for both the baseline and Continuously Row Compaction.

The results are shown in Figure 5.16. In the cases of bwaves, cam4, gcc, omnetpp, roms, and xalancbmk, the benefit from Continuous Row Compaction diminishes with each doubling of the page size from 4KB to 8KB to 16KB, and is further reduced with 2MB pages. As the benefit from Continu-

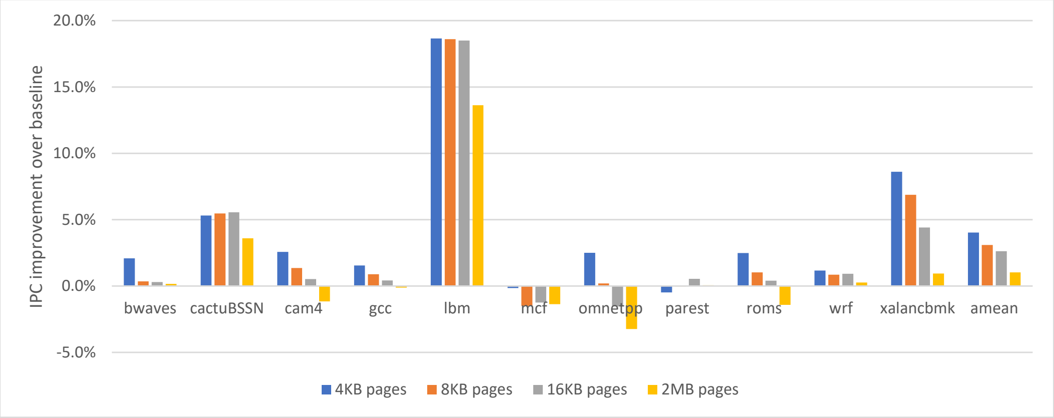


Figure 5.16: Effect of page size on benefit.

ous Row Compaction diminishes with bigger page sizes, the cost of performing data migrations remains the same, and in some cases, we end up with a performance loss with bigger page sizes (cam4 with 2MB pages, gcc with 2MB pages, omnetpp with 16KB and 2MB pages, roms with 2MB pages).

lbm and cactuBSSN continue to benefit from Continuous Row Compaction, even with larger page sizes. Both these benchmarks perform stencil based computations like the motivational example of section 5.2. As was explained in that example, concurrent accesses to memory regions with different row addresses remain a problem even with huge pages, because the stencil is multi-dimensional and is likely to span multiple regions with different row addresses. Continuous Row Compaction can still help in these cases by rearranging and migrating data in the array to minimize the number of regions with different row addresses encountered by the stencil as it traverses across the array.

5.7.5.5 Prefetching

One might wonder why Continuous Row Compaction gave such substantial benefit on highly prefetchable streaming benchmarks such as lbm. If all memory requests can be prefetched far enough in advance, then request latency improvements from increased row buffer hit rate should not matter. Furthermore, with enough outstanding prefetch requests buffered, high channel bandwidth utilization can be maintained by exploiting bank level parallelism, even if row buffer hit rate is low.

Additional experiments show this is indeed the case, provided that:

- there are enough Miss Status Holding Registers (MSHRs) and memory controller queue capacity to hold enough outstanding requests to achieve high bank level parallelism, and
- the prefetcher is able to track enough concurrent streams to issue enough prefetches

The original baseline configuration (Table 5.2) had 32 MSHRs per core, and was able to track 16 concurrent prefetching streams per core. However, this turned out to be insufficient for allowing lbm to prefetch well enough to become insensitive to row buffer hit rate. I thus performed additional experiments that:

- double the number of MSHRs per core to 64

- increase the number of tracked prefetch streams from 16 to 32, then 64

This is shown in Figure 5.17. Each bar in Figure 5.17 represents the single core IPC improvement of Continuous Row Compaction over the baseline with the specified number of MSHRs and tracked prefetcher streams. The left three bars for each benchmark are with the original MSHR capacity of 32, while the right three are with 64 MSHRs. Within each set of three bars, the individual bars represent, from left to right, 16 prefetch streams, 32 prefetch streams, and 64 prefetch streams.

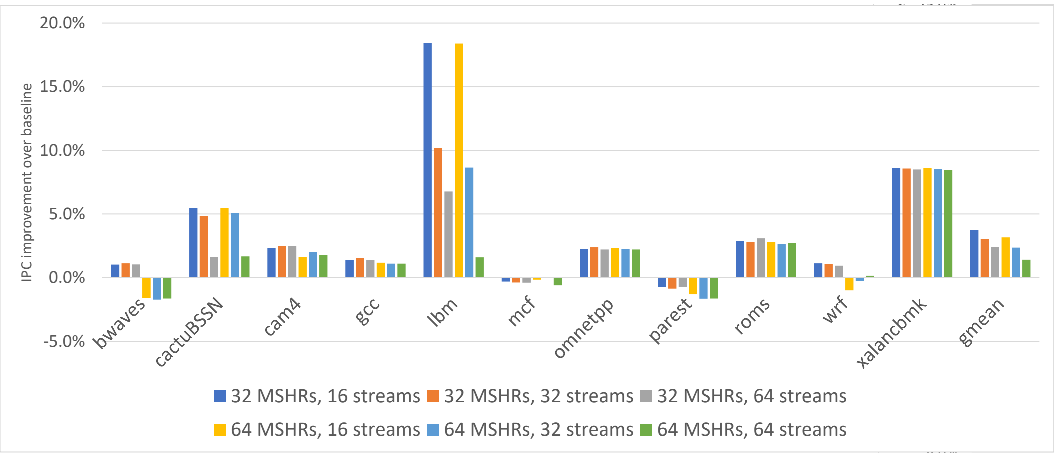


Figure 5.17: Effect of increasing the number of MSHRs and prefetcher stream buffers.

We see that for bwaves, lbm, parest, roms, and wrf, the benefit from Continuous Row Compaction decreases as the number of MSHRs is doubled from 32 to 64 because more requests can be concurrently held in the memory controller queues, exposing additional bank level parallelism and making row buffer hit rates less important.

For CactuBSSN and lbm, the default 16 prefetch streams was insufficient to fully maximize the number of prefetches. As the number of tracked streams is increased to 32 and 64, prefetching improves substantially for these two benchmarks, making them less sensitive to row buffer hit rates and reducing the benefit from Continuous Row Compaction.

For lbm, the IPC improvement from Continuous Row Compaction decreased from 18.4% to 1.6% when both the number of MSHRs and tracked prefetch streams are increased to 64. However, both needed to be increased to make lbm insensitive to row buffer hit rate. Simply increasing the MSHRs or number of prefetch streams alone decreases the benefit from Continuous Row Compaction, but does not eliminate it. When both the number of MSHRs and tracked streams are increased to 64, only xalancbkmk continues to give substantial benefit (8.5%), since it is not stream prefetchable.

5.7.5.6 Different Candidate Sequence Identification and Selection Algorithms

The Candidate Sequence identification and selection algorithm outlined in section 5.4 records multiple Candidate Sequences during a compaction interval, then picks the Candidate Sequence with the most accesses to compact. In my evaluation I allowed for up to 14 Candidate Sequences to be recorded per core (Table 5.4). 14 was chosen because this was the most number of Candidates Sequences on average encountered per compaction interval among all benchmarks. However, since the storage for tracking Candidate Sequences is relatively cheap, one can easily track more Candidate Sequences if necessary.

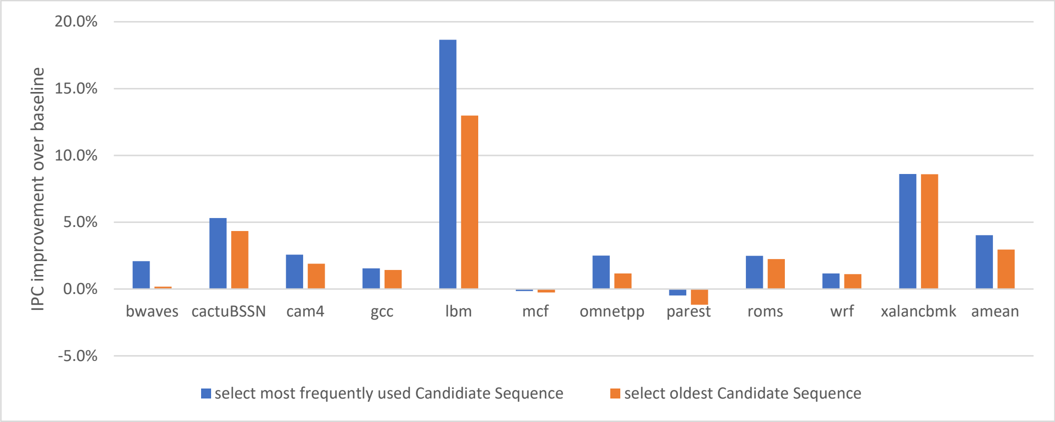


Figure 5.18: Selecting most frequently accessed vs. oldest candidate sequence for compaction.

I in addition examined a simpler algorithm that always picks the oldest Candidate Sequence for compaction. Figure 5.18 shows that we do lose some performance with this simpler algorithm. In particular, the stencil based benchmarks lbm and cactuBSSN perform worse with simply choosing the oldest Candidate Sequence. This makes sense, as each Candidate Sequence essentially defines the shape of a region within the 3-D array to compact to the same row address. By recording multiple Candidate Sequences and then choosing the one with the most accesses, we are essentially trying out different possible shapes of regions to compact, and then picking the best one.

One might wonder whether the sequential Candidate Sequence identification algorithm was over-simplistic and can result in sub-optimal candidates for compaction. For example, assume 4 frames can fit in a compacted row address. If the very first access was to frame A, but the remaining accesses were repeatedly to frames B, C, D, and E, then the sequential algorithm would identify the two Candidate Sequences:

- A,B,C,D
- E

This is clearly sub-optimal, as the algorithm fails to identify the best compaction candidate B, C, D, E.

To quantify how much of a problem this is, I developed an alternative Most Frequently Seen Together algorithm that always tracks the last N (where N is the number of frames that can fit in a compacted row address) unique frames that were accessed. A counter is associated with each set of N unique frames that has been encountered. On every DRAM access, the counter for the set containing the N most recently accessed unique frames is incremented. This is done during an entire compaction interval. At the end of the interval, the set of frames with the highest counter value, denoting the set of frames that were most frequently accessed together, is chosen for compaction for the next interval. Not that unlike the simple sequential algorithm, the Most Frequently Seen Together algorithm can always identify the set of most frequently accessed together frames regardless of where they appear in the sequence of accesses.

Figure 5.19 shows the IPC improvement of Continuous Row Compaction with both the original sequential Candidate Sequence identification

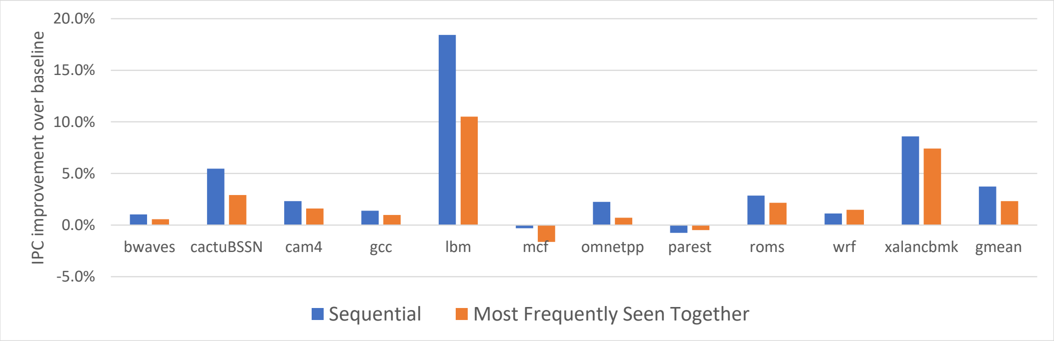


Figure 5.19: IPC improvement over baseline with sequential vs. Most Frequently Seen Together compaction candidate identification

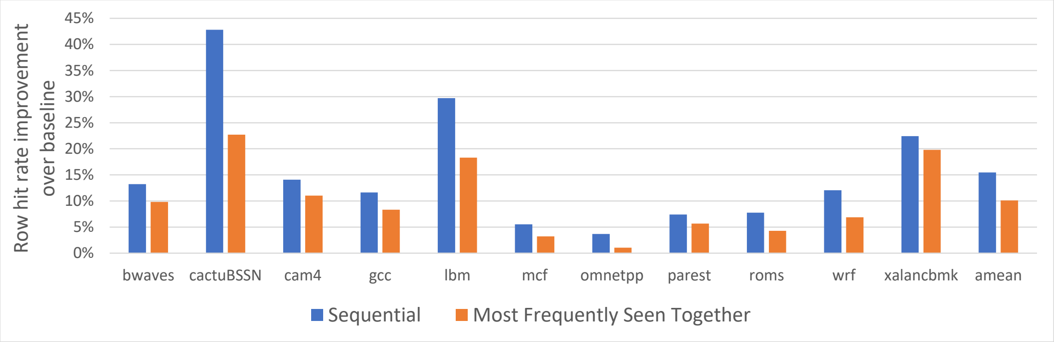


Figure 5.20: Row buffer hit rate improvement over baseline with sequential vs. Most Frequently Seen Together compaction candidate identification

algorithm, and the new Most Frequently Seen Together algorithm. Surprisingly, the Most Frequently Seen Together algorithm actually performs worse on all benchmarks except parest and wrf (and even then, only with very modest performance improvements). The reduced IPC improvement is due to reduced row buffer hit rate improvement, as shown in Figure 5.20.

It is not entirely clear what an optimal algorithm for compaction candidate identification is, as the more sophisticated Most Frequently Seen Together algorithm actually performs worse than the sequential algorithm. However, since Continuous Row Compaction mainly benefits workloads with repeating access patterns, it seems the simple sequential algorithm is able to capture good enough candidates for compaction.

5.7.5.7 Unified vs. Distributed Compaction Across Channels

We have thus far assumed a single unified Remap Table, capable of maintaining all frame remappings across all channels. In practice, this may not be feasible on large chips, because different channels may be controlled by separate memory controllers that are physically apart. Functionally, it is possible to duplicate all necessary structures (Remap Table, Current Writeback/Fill Tracker, Pending Writebacks/Fills Queue, Sequence Table, Sequence Locator) and perform distributed row compaction independently per channel, or on any arbitrary division of channels, as long as one ensures that each frame is always remapped to a compacted frame with the same XOR output among the frame index bits during channel computation. For example, in the physical-to-DRAM address mapping of Figure 2.4(e), the channel is computed as the XOR of bits 19, 18, 13, 12, 9, and 8 of the physical address. Among these, bits 9 and 8 are part of the page offset (i.e., not part of the frame index), and will remain unchanged under frame remappings. However, bits 19, 18, 13, and 12 will potentially change during frame remapping. If row compaction is performed independently per channel, or on some arbitrary division of channels, then we need to make sure the new compacted frame that we remap to has the same XOR value across address bits 19, 18, 13, and 12. This way, we are guaranteed that the original frame and the compacted frame will be interleaved to different channels in the same manner.

Distributed row compaction creates two issues. First, concurrently accessed regions with different row addresses might be compacted in one channel, but not the other. This can lead to worse performance. Figure 5.21 shows that, indeed, separate compaction across two channels performs worse than unified compaction.

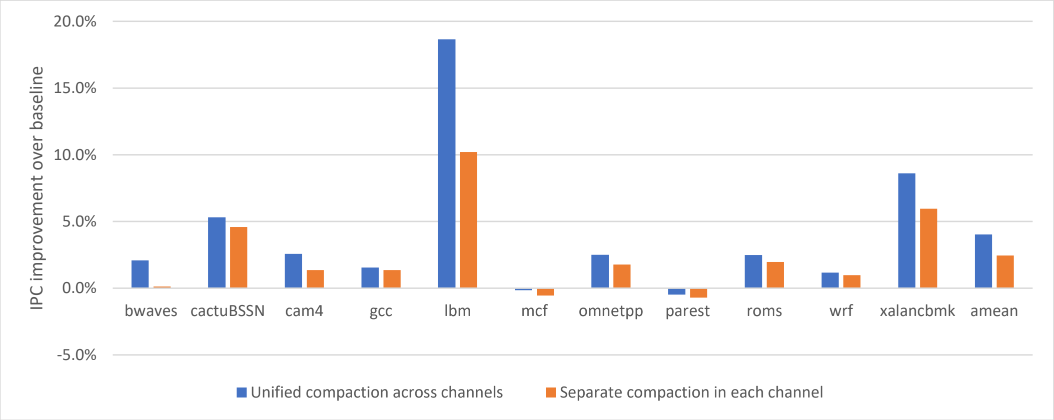


Figure 5.21: Unified vs. Separate compaction with two channels.

Second, duplication of the Remap Table increases the hardware storage cost of Continuous Row Compaction. For these reasons, it is preferred to have as many channels share the same row compaction data structures as possible.

5.7.5.8 Warmup Length

In section 5.7.3, I explained that it takes billions of instructions to completely fill up the Continuous Row Compaction reserved storage. For single core simulations, we are able to warmup for the necessary number of instructions (8 billion) to completely fill the reserved storage. However, for 4-core and 8-core simulations, it would be very slow to warmup for 8 billion instructions. I instead only warmup for 1 billion instructions, but allow data to be filled into the Continuous Row Compaction reserved storage at a greater rate during warmup, so that by the end of the 1 billion warmup the entire reserved storage has potentially been filled. Once warmup ends and detailed simulation starts, we replace data in the reserved storage at the normal rate (e.g., once every 3.2 million DRAM cycles).

To verify whether this methodology is sound, I compared the single core results between running the full 8 billion warmup and running the 1 billion abbreviate warmup.

The results are shown in Figure in 5.22. We see that, in general, the results between 8 billion and 1 billion instructions warmup are very close. The only notable exception is cactuBSSN, where 1 billion instructions warmup performs worse compared to 8 billion instructions. Thus I believe using 1 billion warmup for 4-core and 8-core evaluations is a reasonable methodology.

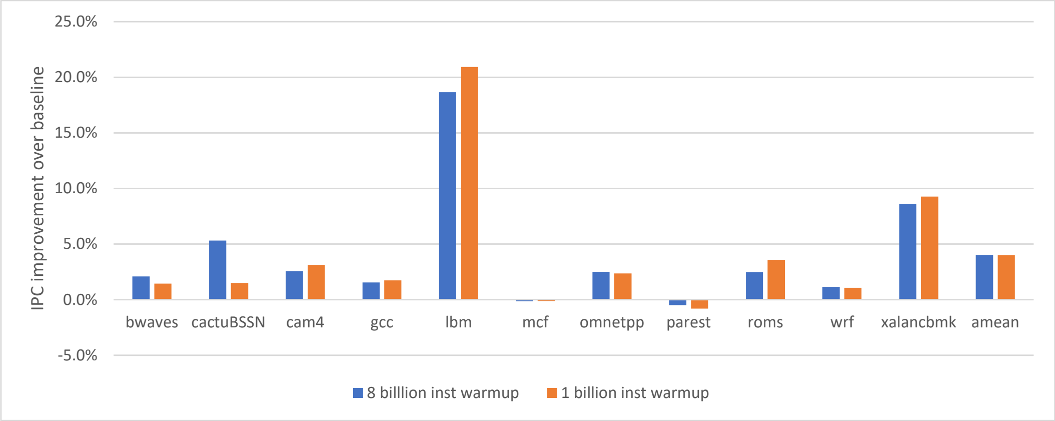


Figure 5.22: 8 billion vs. 1 billion instructions warmup.

5.7.5.9 DRAM Reserved Storage Overhead

This section discusses what the reserved DRAM storage overhead was as a fraction of the working set of the application, and how the benefit from compaction changes as this fraction is varied.

For all SimPoints with $\geq$ 5% IPC improvement from row compaction in the default single core configuration, I performed additional sweeps on the amount of DRAM storage reserved for compaction. I then plotted the IPC improvement against the amount of DRAM reserved as a fraction of the application working set, which I define as the total amount of memory that was accessed during the 8 billion instructions warmup (see section 5.7.3) and 200 million instructions detailed simulation.

Figure 5.23 shows the result of the sweeps. Each curve represents a different SimPoint. For each point on a curve:

- the x-axis value represents the amount of DRAM reserved for row compaction, as a fraction of the amount of total memory accessed during the 8 billion instructions warmup and 200 million instructions detailed simulation. For example, if 200 MB of DRAM was reserved for row compaction, and the SimPoint accessed 400 MB worth of data during warmup and detailed simulation, then the corresponding x-value would be 50%

- the y-axis value represents the fraction of the maximum IPC improvement attained. For example, in sweeping the amount of DRAM storage reserved for row compaction, if the maximum IPC improvement attained across the entire sweep was 10%, while the IPC improvement for this particular data point in the sweep is 2%, then the y-value would be 20%

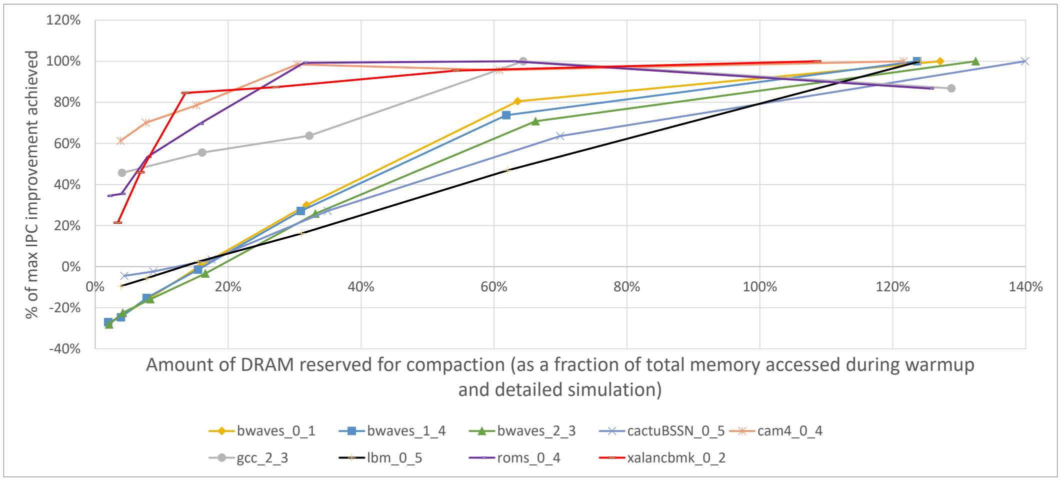


Figure 5.23: Fraction of max IPC improvement achieved as a function of fraction of DRAM reserved for row compaction.

We see that the benchmarks (i.e., SimPoints) can be roughly divided into two groups. The first group, comprising of cam4_0_4, roms_0_4, xalancbmk_0_2, and to a lesser extent, gcc_2_3, is able to achieve most of the maximum IPC improvement with only around 20% to 40% of its working set compacted. The second group, made up of all the remaining benchmarks, require the entire working set to be compacted in order to achieve the maximum IPC improvement. This gives us some indication on what will happen with future workloads with even larger working sets - some will be able to get most of the benefit of compaction with about 20% to 40% of its working set compacted, while others will require the entire working set to be compacted in order to see full benefit.

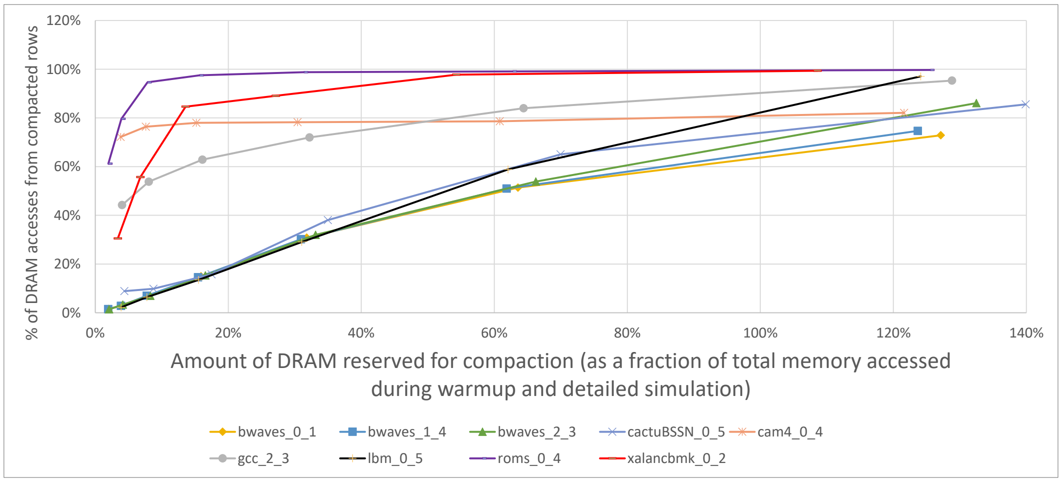


Figure 5.24: Fraction of DRAM accesses from compacted rows as a function of fraction of DRAM reserved for row compaction.

Figure 5.24 is similar to Figure 5.23, but instead plots the fraction of DRAM accesses serviced from compacted rows on the y-axis. We see the same trend where the first group, comprising of cam4_0_4, roms_0_4, xalancbmk_0_2, and to a lesser extent, gcc_2_3, is able to have most of its DRAM requests serviced from compacted rows with only around 20% to 40% of its working set compacted. With the second group, made up of all the remaining benchmarks, the fraction of accesses serviced from compacted rows scales roughly linearly with the fraction of the working set compacted.

5.7.5.10 Remap Table Latency

Another consideration is the latency through the Remap Table, which must be accessed on every DRAM access to determine if the location has been compacted.

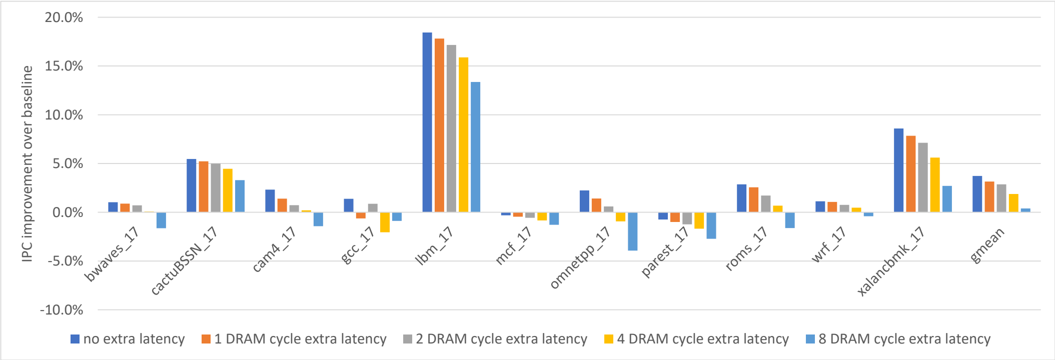


Figure 5.25: Effect of Remap Table latency (in DRAM cycles) on performance.

Figure 5.25 shows the effect of Remap Table latency on Continuous Row Compaction performance benefit as the latency is varied from 0 to 8

DRAM cycles.[5] Continuous Row Compaction is configured as before with 1K reserved row addresses, allowing 512 MB of memory to be compacted. Given this configuration, the Remap Table would be 660 KB (see section 5.7.8).

Unsurprisingly, the benefit degrades as the Remap Table latency increases. Hence Remap Table latency is one of the biggest challenges to realizing Continuous Row Compaction.

5.7.6 4-core Performance

We now present our 4-core performance results. We report weighted speedup [58] (section 4.7) as a measure of multi-core performance. Each workload in the 4-core mix is simulated until every benchmark in the 4-core mix has retired at least 200 million instructions. Benchmarks that exceed 200 million instructions continue to execute to provide interference for the other unfinished benchmarks, but the IPC is recorded from the first 200 million instructions corresponding to the representative SimPoint region.

For our 4-core evaluations, we increased the amount of reserved storage for compaction by 4x from 512MB to 2GB.

5.7.6.1 Benchmark Selection

We formed 22 randomized 4-core multi-programmed workloads from our set of eleven memory intensive SPECrate 2017 benchmarks, such that each benchmark appears 8 times across all 22 multi-programmed mixes. For

[5] each DRAM cycle is two CPU cycles in our configuration; see Table 5.2

each benchmark, the composition of the 8 included SimPoints depends on the SimPoint weights. For example, the benchmark lbm has only one representative SimPoint of weight 100% that is included 8 times across all 22 randomized 4-core mixes. On the other hand, cactuBSSN, with three representative SimPoints of weights 59%, 29%, and 12%, has the first SimPoint included 5 times, the second SimPoint included 2 times, and the third SimPoint included 1 time across all 22 4-core mixes. Table 5.3 shows the 22 4-core mixes.

Table 5.3: 4-core mixes.

Mix-1	cac0_2	lbm0_5	mcf0_4	wrf0_4
Mix-2	bwa0_3	lbm0_5	mcf0_4	rom0_4
Mix-3	cac0_2	omn0_2	rom0_4	xal0_3
Mix-4	bwa3_5	gcc2_1	rom0_3	rom0_5
Mix-5	omn0_2	par0_4	par0_4	wrf0_4
Mix-6	bwa3_4	omn0_2	rom0_5	wrf0_1
Mix-7	lbm0_5	omn0_2	xal0_3	xal0_3
Mix-8	cac0_4	lbm0_5	par0_4	wrf0_2
Mix-9	bwa1_2	par0_4	rom0_4	xal0_3
Mix-10	gcc1_1	mcf0_1	mcf0_4	wrf0_2
Mix-11	cam0_2	cam0_4	cam0_4	gcc4_2
Mix-12	cac0_5	omn0_2	xal0_2	xal0_3
Mix-13	cac0_2	cam0_4	gcc0_1	lbm0_5
Mix-14	cac0_4	cam0_2	lbm0_5	mcf0_1
Mix-15	cam0_4	gcc3_3	rom0_4	xal0_4
Mix-16	bwa2_1	bwa3_5	cam0_1	par0_2
Mix-17	omn0_2	omn0_2	par0_1	wrf0_1
Mix-18	cac0_2	gcc4_1	lbm0_5	xal0_2
Mix-19	cam0_4	gcc1_1	lbm0_5	wrf0_3
Mix-20	mcf0_4	mcf0_5	par0_4	rom0_3
Mix-21	bwa1_2	bwa2_1	cac0_2	wrf0_3
Mix-22	gcc4_1	mcf0_4	omn0_2	par0_4

5.7.6.2 Compaction Overhead

With 4 cores, there is now more contention for the two memory channels. We again perform the experiment in section 5.7.5.3, where we make writebacks and fills free, and see whether or not our Explicit Copying Throttling mechanism is still able to minimize writeback and fill overhead in the multi-core case.

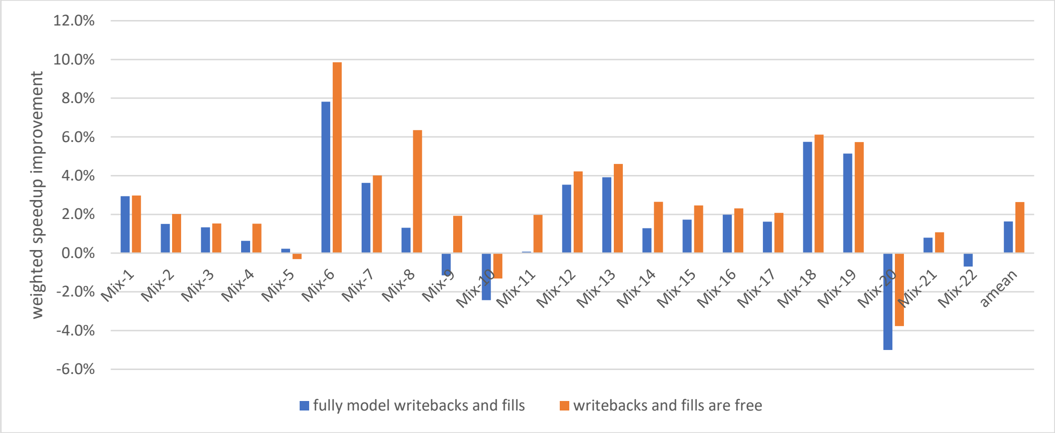


Figure 5.26: Effect on performance when writebacks and fills are made free for 4-cores.

We see that with 4-cores, the performance gap between fully modelling writebacks/fills vs. making writebacks/fills free is larger. This is expected, as there is more contention with 4-cores, and writebacks and fills become relatively more expensive.

We note that in mixes 5, 10, 20, and 22, we have almost zero benefit, or end up with performance losses, even when writebacks and fills are free. This can be explained by the benchmark composition of those three mixes. As we saw in section 5.7.5, mcf and parest are our two worse performing benchmarks, due to memory access reorderings that end up producing more L2 (LLC) evictions. Mix-5 has two copies of parest, mix-10 has two copies of mcf, mix-20 has two copies of mcf and 1 copy of parest, and mix-22 has a copy of mcf and a copy of parest. No other mixes have more than 1 copy of mcf or parest (Table 5.3).

We also note that the performance detriments from increased L2 (LLC) evictions will be more pronounced in a multi-core environment compared to the single core case, because with multi-core, not only do we lose performance because we are getting fewer L2 hits, but we are also creating more memory traffic to an already contended DRAM system. Hence the performance losses suffered by mcf and parest are more drastic in the 4-core case than in the single core case.

5.7.6.3 Row Buffer Hit Rate

Figure 5.27 shows that, like in the single core case, Continuous Row Compaction is able to improve row buffer locality and reduce the number of DRAM Activate operations required across the board.

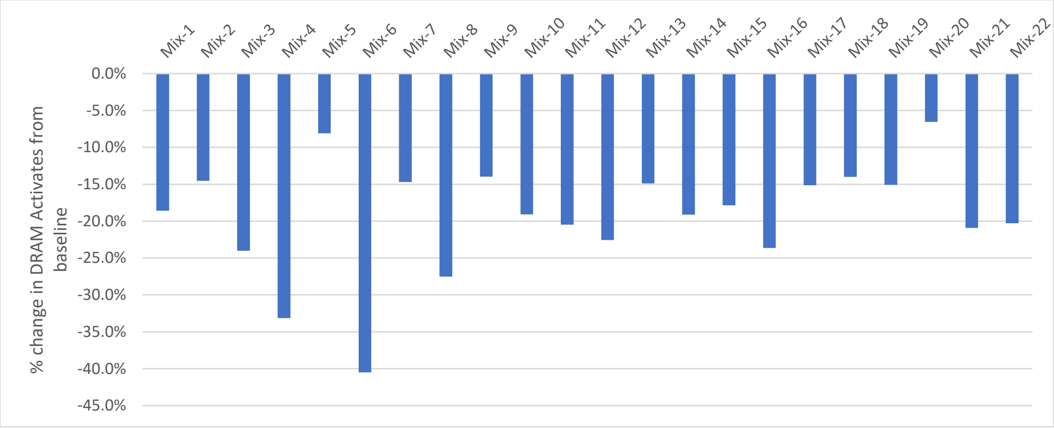


Figure 5.27: Change in number of DRAM Activates for 4-core mixes

5.7.7 Effect of Varying Bank and Core Counts

In this section we examine the effect of varying bank (i.e., rank) and core count on Continuous Row Compaction.

Our starting point is a single core evaluation on a system with 2 channels and 1 rank per channel (16 DDR4 banks per rank). We take the SimPoints on which Continuous Row Compaction performed the best ($\geq 5\%$ IPC improvement) in this single core setup, then vary the bank and/or core count. Bank count is varied by doubling the number of ranks to 2 per channel. Core count is varied by taking each SimPoint and creating 2-core, 4-core, and 8-core multi-programmed workloads with 2, 4, or 8 copies of the same SimPoint. For the multi-programmed workloads, the amount of DRAM reserved for compaction is scaled with the number of cores. For example, if X units of DRAM was reserved for compaction in the single core case, then 2X, 4X, and 8X DRAM are reserved for the 2-core, 4-core, and 8-core multi-programmed workloads, respectively.

This is shown in Figure 5.28. For each benchmark (i.e., SimPoint), the left four bars are with 1 rank per channel, while the right four are with 2 ranks per channel. Within each set of four bars, each individual bar represents, from left to right, the performance benefit of Continuous Row Compaction over the baseline with 1, 2, 4, and 8 cores, respectively.[6] The number of channels

[6] IPC improvement in the 1-core case, Weighted Speedup improvement in the 2,4, and 8 core cases

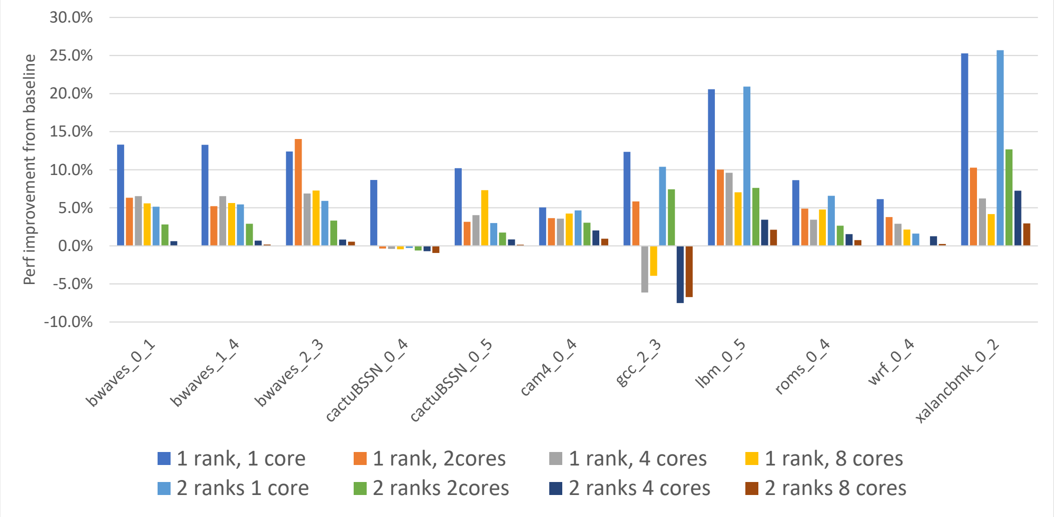


Figure 5.28: Performance improvement with varying rank and core counts.

remains at 2 regardless of the number of ranks per channel and the number of cores.

In general, the benefit from Continuous Row Compaction diminishes with more ranks(i.e., banks), as more bank level parallelism becomes available and the importance of row buffer hit rate decreases. Similarly, the benefit from Continuous Row Compaction diminishes with more cores, because it becomes easier to saturate the channel bandwidth with traffic from multiple cores, even at low row hit rates.

However, there are two cases, cactuBSSN_0_5 and cam4_0_4, where the benefit of Continuous Row Compaction continues to scale with increasing core counts at low bank counts (i.e., 1 rank per channel). While in both cases the performance benefit initially decreased from 1-core to 2-core, it then improved with each additional core count doubling afterwards.

This is further supported in Figure 5.29, which shows the improvement in DRAM channel utilization from baseline with varying rank and core counts. We see that for cactuBSSN_0_5 and cam4_0_4, channel utilization improvement scales with increasing core count at 1 rank per channel.

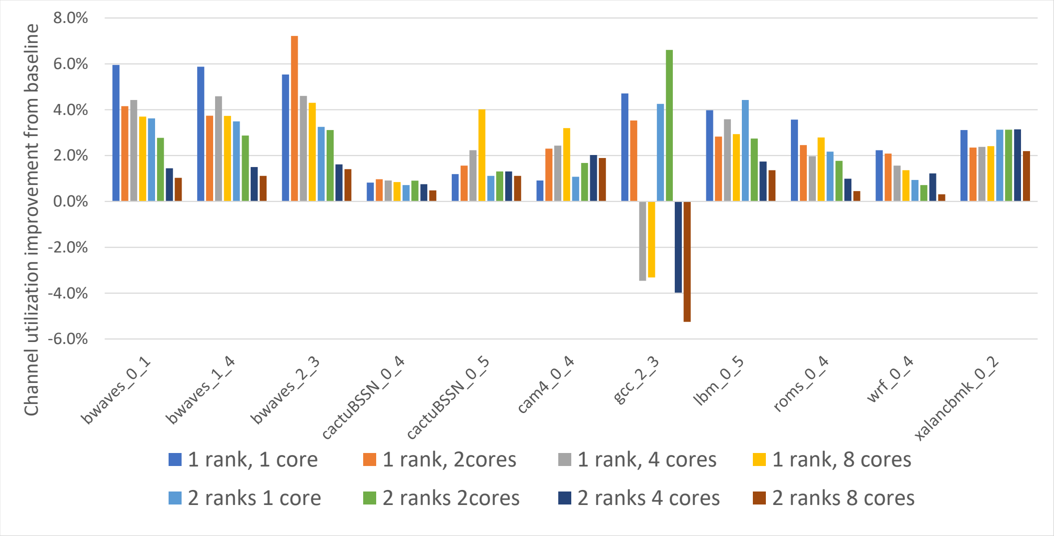


Figure 5.29: DRAM channel utilization improvement with varying rank and core counts.

5.7.8 Area

We now examine the on-chip storage budget required for Continuous Row Compaction. We assume the configurations listed in Table 5.4 for our storage budget calculations.

Table 5.5 shows the sizes of data structures required for Continuous

Table 5.4: Memory and Continuous Row Compaction configuration for area calculation.

DRAM	2 Channels, 2 Rank/Channel, DDR4-3200, 22-22-22, 16Gb x4 device, 256K rows, 128GB total capacity
Reserved Memory	1K reserved row addresses (out of 256K) = 512MB out of 128GB reserved
Sequence Table	14 Candidate Sequences per core, 128 tracked frames and 13 bit saturating Access Counter per Candidate Sequence
Sequence Locator	1K sets, 3-way set-associative table per core
Remap Table	8K sets, 22-way set-associative

Row Compaction. Overall, the storage cost is dominated by the Remap Table. Total storage required is 681.4KB.

Table 5.5: Required data structures and cost

Structure	Cost
Remap Table	660 KB
Pending Writebacks+Fills Queues	800 B
Current Writebacks+Fill Trackers	8.0 KB
Replacement Pointer	10 bits
Access Counter for frames to be replaced	13 bits
Sequence Table	5.5 KB
Sequence Locator	7.1 KB
Total	681.4 KB

We compare the single core performance of Continuous Row Compaction in this configuration against the performance of the baseline with 512KB and 1MB of added L2 (LLC). This is shown in Figure 5.30. Continuous Row Compaction outperforms the baseline with additional cache in bwaves, cactuBSSN, lbm, and xzlancbmk, while the baseline with additional cache is better in the other benchmarks. Overall, Continuous Row Compaction improves performance by 3.9%, compared to 3.3% and 1.9% for the baseline with 1MB and 512KB of additional cache added.

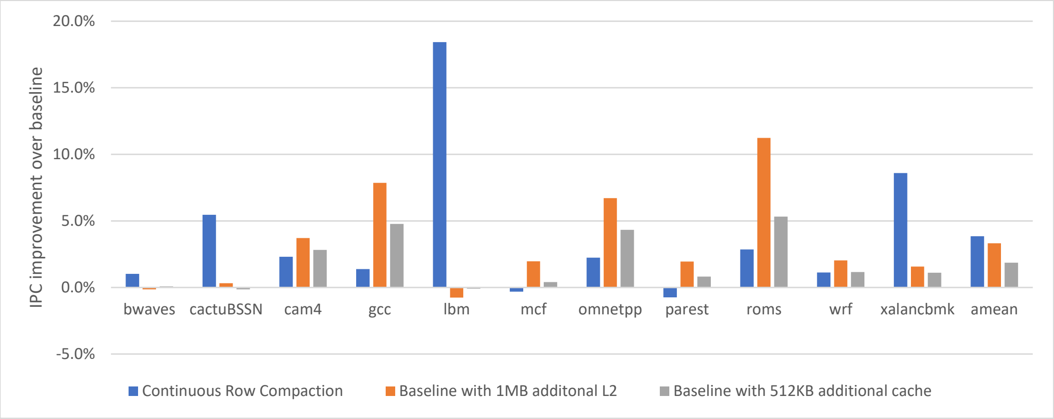


Figure 5.30: Performance comparison to baseline with additional cache.

Chapter 6

Related Work

6.1 Caching using Performance-Optimized DRAM

Caching using performance-optimized DRAM such as eDRAM, HBM, MCDRAM, RLDRAM, and GDDR DRAM can significantly increase performance, but comes with high cost. eDRAM requires special fabrication technology to integrate DRAM to the same die as the processor. HBM and MCDRAM require special packaging to connect to the processor. RLDRAM has reduced density and is much more expensive than cost-optimized DRAM. GDDR DRAM transfers data at very high data rates and consumes considerable power and energy. In contrast, my work seeks to improve performance while employing only cheap, widely available commodity DRAM memory.

6.2 Caching using Tweaked Cost-Optimized DRAM

Prior work like TL-DRAM[29] and CHARM[60] attempted to make small tweaks to commodity cost-optimized DRAM to create fast and slow regions within the same device, then use the fast region as a cache. However, there are two main problems with this approach. First, some of the changes proposed, like breaking the bitline with isolation transistor, are not trivial in density-optimized DRAM technology. Second, due to the commodity nature of the DRAM industry, adoption of device level optimizations is often slow and difficult. In contrast, my work does not require any modifications to the DRAM device or access protocol, is completely transparent to software, and can be unilaterally adopted by the processor vendor.

Multiple Clone Row DRAM (MCR-DRAM) [6] sacrifices memory capacity for latency by using multiple DRAM rows (called a Multiple Clone Row, MCR) to represent a single logical row of data. Latency is reduced for data represented with MCRs due to:

- increased sensing strength from the increased number of sensed cells, effectively reducing t_{RCD}, the latency between a row Activate and a subsequent Read/Write
- more frequent refreshes to the data without increasing the frequency of Refresh commands. This allows:
  - t_{RAS} to be reduced without compromising data integrity, meaning Precharges can be issued earlier, and
  - t_{RFC} to be reduced, reducing the overhead from Refreshes

MCR-DRAM still requires changes to the DRAM device, but changes are limited to the peripherals outside of mat cell-arrays. This is desirable because the mat cell-arrays are highly optimized for cell density, and DRAM vendors are often hesitant to muddle with this part of the design. By contrast, Continuous Row Compaction requires no modifications to the DRAM device at all.

A row of compacted data in Continuous Row Compaction requires the same amount of memory storage as a 2x MCR that uses two DRAM rows to represent a single logical row. One can have a mix of MCRs and normal rows. The MCRs can be used as a cache of data in the normal rows. If this were the case, the two main challenges discussed in this thesis:

1. minimizing the data movement overhead associated with filling the cache
2. efficiently tracking what has been cached

apply equally to a MCR-DRAM based cache, and many of the same solutions discussed in chapter 3 can be employed.

6.3 Caching using Unmodified Cost-Optimized DRAM

Micro-pages [66] tries to leverage intra-page access non-uniformity within individual 4KB OS page by identifying and combining the most frequently accessed 1KB regions (micro-pages) from different 4KB OS pages into the same row buffer. However, our experiments show that first, meaningful intra-page access non-uniformity only exists for a small handful of benchmarks (omnetpp, xz), while for most other benchmarks, different 1KB micro-pages within a 4KB OS page all get accessed frequently enough that there are no clear discernible hot vs. cold 1KB micro-pages. Second, it is insufficient to use access frequency as the sole criterion for determining which micro-pages to combine into the same row buffer, as this leads to frequently accessed micro-pages with little to no temporal correlation being placed in the same row buffer, which does not provide benefit. In contrast, Continuous Row Compaction records the order in which unique 4KB regions are accessed in during Candidate Sequence Identification, thereby capturing temporal correlation between different OS pages.

6.3.1 Reducing DRAM Latency using Unmodified Cost-Optimized DRAM

Non-Uniform Access Time memory controller (NUAT) [57], ChargeCache[13], and Charge-Level-Aware Look-Ahead Partial Restoration (CAL)[73] track which rows of data have been recently activated or refreshed, or are likely to be activated/refreshed in the future, and loosen the Activate timing constraints for these rows. This is possible because the DRAM cell capacitors in these rows have charge levels that have either been recently restored from a recent Activate/Refresh, or will soon be restored by another future Activate/Refresh. However, these techniques only reduce the Activate latency, not the Precharge latency. In contrast, Duplicon Cache can potentially hide both the Precharge and Activate latencies by allowing requests to be serviced earlier at a less loaded bank, while Continuous Row Compaction can eliminate Precharges and Activates altogether.

6.4 Fast In-Memory Copying

One of the main challenges in making Duplicon Cache and Continuous Row Compaction work is the limited channel bandwidth of cheap commodity memory, as additional data traffic from data duplication/migration worsen contention for this limited channel bandwidth (section 3.4). Prior work [53, 5, 74] again modify the DRAM device to create connections that allow bulk copying of data entirely within the DRAM device (setion 3.4.1). In contrast, I developed techniques in this thesis that identify and capture stable working sets of the application over very long time intervals, enabling the benefits of caching while minimizing data movement overhead by performing cache writebacks and fills very infrequently.

6.5 Partitioning Memory Resources

Duplicon Cache derives benefit from mitigating bank conflicts that arise under contention in multicore systems. There have been proposals to limit memory contention in multicore systems by partitioning memory channels [40, 32] or banks[32, 14, 16, 33, 79, 37, 19] among different co-running applications/threads. However, scarcity in memory channels and banks fundamentally limit the effectiveness of partitioning techniques.

For example, as of January 2021, the processor with the most cores from Intel supports 112 threads and 12 DDR4 channels. This scarcity of channels to threads creates a fundamental pigeonhole problem for channel partitioning. Furthermore, if we assume 4 DDR4 ranks per channel, then there are 768 total banks in the system, and on average 6.9 banks per thread. However, Bank-level Partitioning Mechanism (BPM [32]) showed that the vast majority of SPEC 2006 applications benefited when the number of banks allocated was doubled from 8 to 16. Hence there is a fundamental scarcity of banks that cannot be solved through partitioning.

In addition, Continuous Row Compaction is able to mitigate bank conflicts even in single core scenarios without interference from other cores.

6.6 Subrank and Subarray Parallelism

A straightforward way to decrease memory contention is to increase the number of channels and/or banks; however, this is expensive. Alternatively, researchers have proposed subdividing existing channels and banks into smaller independent modules.

A DRAM bank is implemented as a collection of subarrays that are largely independent but share certain global structures. Subarray-level parallelism (SALP) [24] proposes DRAM circuitry and protocol changes that allow subarray-level parallelism to be exposed to memory scheduling. However, the need for DRAM circuitry and/or DRAM protocol changes is problematic. In contrast, Duplicon works with mainstream commodity memory and does not require any protocol changes.

Other proposals partition the devices of a rank into smaller and narrower subranks that can be controlled independently, increasing the memory-level-parallelism[75, 3, 81, 4]. Subranking has two main drawbacks compared to Duplicon. First, subranking requires adding additional chip select (CS) pins to control individual devices within a rank, requiring additional pins, wires, and decoders. Second, the same amount of data now takes longer to transfer across the narrower data interface of a subrank. Duplicon, in contrast, has none of these drawbacks.

6.7 Memory Scheduling and Throttling

FR-FCFS memory scheduling introduces unfairness in the system because row buffer misses/conflicts get unfairly penalized behind row buffer hits (section 2.1.5). To address this problem, numerous algorithms have been proposed to identify which requests should be prioritized because they are being unfairly penalized [21, 42, 22, 43, 23, 64, 63, 45, 65, 65, 8]. However such algorithms usually trade off some memory throughput (by de-prioritizing row buffer hits) to improve fairness. Duplicon gives these algorithms a better throughput/fairness tradeoff by giving the system additional leeway for handling bank conflicts.

BLP-Preserving Multi-core Request Issue (BPMRI) [28] tries to preserve the bank-level-parallelism of requests from individual cores by loading the memory controller request buffers in batches from each individual core. However, if the request scheduling window is large enough, then BPMRI will provide minimal benefit.

6.8 Data Blocking

Blocking [27, 77, 7, 12, 17] is a well known technique to improve data locality across the memory hierarchy. Blocking entails modifying the data layout and/or the order in which data is accessed in order to maximize locality. Continuous Row Compaction essentially performs blocking for DRAM, but at runtime, and in a way that is completely transparent to the application. In fact, from the application's point of view, both the data layout, in virtual address space, and the memory access order, specified by the program order, remain unchanged. Instead, Continuous Row Compaction modifies the data layout in physical memory via an added layer of indirection, the Remap Table, which leads to a more optimized reordering of physical memory accesses.

6.9 Memory Compaction

Continuous Row Compaction also shares similarities to memory compaction performed by the operating system. With OS memory compaction, fragmented pages are migrated to create contiguous aligned free regions that can later be allocated for future huge pages. With Continuous Row Compaction, contiguous regions are reserved in physical memory, and the hardware migrates pages that are frequently accessed together to reserved contiguous regions. In both cases page migration is made possible via a layer of indirection in address translation that the system controls - the page table in the case of OS memory compactions, and the Remap Table in the case of hardware Continuous Row Compaction.

6.10 Temporal Prefetching

Continuous Row Compaction also shares similarities to temporal prefetching (e.g., [78, 76, 18]), as both rely on memory being accessed repeatedly in roughly the same order. The Candidate Sequence Identification process is similar to processes in other temporal prefetchers that record and learn correlation between data addresses. However, Candidate Sequence Identification is far more lightweight, as we only need to detect sets of 4KB pages that are roughly accessed at around the same time, while temporal prefetchers need to learn far more precise correlation between finer granularity (e.g., 64B) blocks.

Chapter 7

Conclusion

7.1 Summary

In this thesis I showed, through Duplicon Cache and Continuous Row Compaction, that one can mitigate the effects of memory bank/row conflicts via duplication of data across banks and migration of data to non-conflicting row buffers. I also showed we can perform the necessary data duplications and migrations entirely with unmodified DRAM using unmodified DRAM access protocols, without the additional memory traffic eroding away the performance gains. The key was to identify large working sets that remain stable over 100s of millions to billions of instructions and duplicate/migrate those working sets at low intensity over long time intervals.

The main drawback of the work is that the size of the required SRAM tag store becomes prohibitively large for larger working sets. This is particularly problematic because the SRAM tag store needs to be accessed on every DRAM access (to determine if the data has been duplicated/compacted), and this access latency cannot be hidden. With Duplicon Cache, which employs sectoring, the tag store overhead is at least 1 bit per 64B of duplicated data. Continuous Row Compaction halved this overhead to 0.47 bit per 64B of compacted data.[1] For duplication/migration of up to 100s of MBs of data, the corresponding SRAM tag store can be kept to within 100s of KB, and I showed for both Duplicon Cache and Continuous Row Compaction that using this SRAM storage budget as tag store for Duplicon Cache and Continuous Row Compaction gives more benefit than simply adding the same area to the on-chip SRAM cache. Thus for applications whose working sets are on the order of 100s of MBs, both Duplicon Cache and Continuous Row Compaction are feasible solutions. However, to extend my work to even larger larger working sets - 10s/100s of GBs, even TBs, a couple of things need to happen:

7.2 Future Work

7.2.1 Integrated DRAM Tag and Data

First, we need a way to remove the need for expensive SRAM tag stores and directly store the tag in DRAM itself. For example, the Intel Xeon Phi processors employ a 16GB MCDRAM cache in which the tag is stored in the memory error-correcting code (ECC) bits, eliminating the need for SRAM tag stores [59]. Upon accesses to data in MCDRAM, the tags are also read out and transferred to the memory controller, via the same mechanism through which the ECC bits are read out and transferred. The tag can then be checked at the memory controller to determine is the data is correct or not.

However, such schemes only work for direct mapped caches, because

[1] 30 bits per 4KB frame; see section 5.5 for details

one needs to exactly know where in DRAM the data and tag would be before making the access. Duplicon Cache was configured as a 4-way set associative cache, but it can also be configured as a direct mapped cache (section 4.3.1). I believe it is possible to implement Duplicon Cache with commodity ECC DRAM and store the Duplicon tags in the ECC bits, eliminating the need for a SRAM tag store. All data in memory would thus have a single direct mapped alternate location in which the duplicate data copy would reside, if it exists. In this configuration, there would no longer be a need for cache sectoring. Instead, we can maintain fine-grained line sizes (e.g., 64B, or whatever the minimum granularity of memory access is), and store the tag for each fine-grained line in the ECC bits. On a read access, a hit predictor would predict whether or not the data has been duplicated, and if the predictor predicts a hit, the data in the alternate location can be fetched in lieu of the original location, with the tag from the alternate location ECC bits needing to be checked after fetching the data. Additional care is required to ensure coherence of duplicated data on writes. For example, on a write to location A, one cannot for sure know whether or not location A has been previously duplicated without making a DRAM access. To ensure coherence, one could, if the hit predictor predicts a hit, always overwrite the alternate location of A with the new value of A, and update the tag in the ECC bits for the alternate location to A.

Continuous Row Compaction, on the other hand, is not a direct mapped cache where one knows exactly where in DRAM the cached data is; in fact, it is the opposite of a direct mapped cache, because cached data could have been migrated to anywhere within the reserved memory. I do not yet have a good solution on how the SRAM tag store (i.e., Remap Table) can be eliminated.

7.2.2 Non-Volatile Memory

If we want to enable Duplicon Cache and Continuous Row Compaction on the order of 100s of GBs or TBs (or even greater), then we need to move to an even cheaper memory technology, such as non-volatile memory. Fundamentally, non-volatile memory is also organized with banks and row buffers, so Duplicon Cache and Continuous Row Compaction can in theory be applied to them.

Bibliography

[1] Skylake (server) - microarchitectures - intel.

[2] Ultra-bandwidth solutions. `https://www.micron.com/products/ultra-bandwidth-solutions`. Accessed: 2020-12-25.

[3] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. IEEE Computer Architecture Letters, 8(1):5–8, Jan 2009.

[4] T. M. Brewer. Instruction set innovations for the convey hc-1 computer. IEEE Micro, 30(2):70–79, March 2010.

[5] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 568–580, 2016.

[6] Jungwhan Choi, Wongyu Shin, Jaemin Jang, Jinwoong Suh, Yongkee Kwon, Youngsuk Moon, and Lee-Sup Kim. Multiple clone row dram: A low latency and area optimized dram. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), pages 223–234, 2015.

[7] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw., 16(1):1–17, March 1990.

[8] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pages 335–346, New York, NY, USA, 2010. ACM.

[9] S. Eyerman and L. Eeckhout. Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance. IEEE Computer Architecture Letters, 13(2):93–96, July 2014.

[10] S. Eyerman and L. Eeckhout. Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance. IEEE Computer Architecture Letters, 13(2):93–96, 2014.

[11] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on, pages 149–160, Dec 2006.

[12] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. Journal of Parallel and Distributed Computing, 5(5):587 – 616, 1988.

[13] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. Chargecache: Reducing dram latency by exploiting row access locality. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 581–593, 2016.

[14] E. Herrero, J. González, R. Canal, and D. Tullsen. Thread row buffers: Improving memory performance isolation and throughput in multiprogrammed environments. IEEE Transactions on Computers, 62(9):1879–1892, Sept 2013.

[15] Paul Hsieh. Hash functions., 2008.

[16] T. Ikeda and K. Kise. Application aware dram bank partitioning in cmp. In 2013 International Conference on Parallel and Distributed Systems, pages 349–356, Dec 2013.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, page 319–329, New York, NY, USA, 1988. Association for Computing Machinery.

[18] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, page 247–259, New York, NY, USA, 2013. Association for Computing Machinery.

[19] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In IEEE International Symposium on High-Performance Comp Architecture, pages 1–12, Feb 2012.

[20] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: A dram page-mode scheduling policy for the many-core era. In 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 24–35, 2011.

[21] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A dram page-mode scheduling policy for the many-core era. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pages 24–35, New York, NY, USA, 2011. ACM.

[22] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pages 1–12, Jan 2010.

[23] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 65–76, Dec 2010.

[24] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In 2012 39th Annual International Symposium on Computer Architecture (ISCA), pages 368–379, June 2012.

[25] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. IEEE Comput. Archit. Lett., 15(1):45–49, January 2016.

[26] Kenneth C. Knowlton. A fast storage allocator. Commun. ACM, 8(10):623–624, October 1965.

[27] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, page 63–74, New York, NY, USA, 1991. Association for Computing Machinery.

[28] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving memory bank-level parallelism in the presence of prefetching. In 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 327–336, Dec 2009.

[29] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pages 615–626, 2013.

[30] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[31] B. Lin, M. B. Healy, R. Miftakhutdinov, P. G. Emma, and Y. Patt. Duplicon cache: Mitigating off-chip memory bank and bank group conflicts via data duplication. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 285–297, 2018.

[32] Lei Liu, Zehan Cui, Yong Li, Yungang Bao, Mingyu Chen, and Chengyong Wu. Bpm/bpm+: Software-based dynamic memory partitioning mechanisms for mitigating dram bank-/channel-level interferences in multicore systems. *ACM Trans. Archit. Code Optim.*, 11(1):5:1–5:28, February 2014.

[33] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM.

[34] S. Lu, Y. Lin, and C. Yang. Improving dram latency with dynamic asymmetric subarray. In *2015 48th Annual IEEE/ACM International*

Symposium on Microarchitecture (MICRO), pages 255–266, 2015.

[35] Kun Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on, pages 164–171, 2001.

[36] HPS/SAFARI Research Group. Scarab. `https://github.com/hpsresearchgroup/scarab`.

[37] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In Chen Ding, Zhiyuan Shao, and Ran Zheng, editors, Network and Parallel Computing, pages 329–343, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[38] Micron Technology, Inc. MT40A4G4 DDR4 SDRAM Datasheet Rev. G, August 2020.

[39] Motorola Inc., Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona 85036. PowerPC™ 601 RISC Microprocessor User's Manual, mpc601/d edition. retrieved from `http://pdf.datasheetcatalog.com/datasheet/motorola/MPC601.pdf`.

[40] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In

Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pages 374–385, New York, NY, USA, 2011. ACM.

[41] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to understand large caches.

[42] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pages 146–160, Dec 2007.

[43] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In 2008 International Symposium on Computer Architecture, pages 63–74, June 2008.

[44] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, December 2007.

[45] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. SIGMETRICS Perform. Eval. Rev., 31(1):318–319, June 2003.

[47] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[48] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In 25th USENIX Security Symposium (USENIX Security 16), pages 565–581, Austin, TX, August 2016. USENIX Association.

[49] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. Colt: Coalesced large-reach tlbs. In 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pages 258–269, 2012.

[50] Y. H Qian, D D'Humières, and P Lallemand. Lattice BGK models for navier-stokes equation. Europhysics Letters (EPL), 17(6):479–484, feb 1992.

[51] 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 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pages 235–246, 2012.

[52] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. SIGARCH Comput. Archit. News, 28(2):128–138, May 2000.

[53] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 185–197, 2013.

[54] A. Seznec. Decoupled sectored caches. IEEE Transactions on Computers, 46(2):210–215, 1997.

[55] André Seznec. A 256 kbits l-tage branch predictor. Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2), 9, 2007.

[56] André Seznec. TAGE-SC-L Branch Predictors Again. In 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), Seoul, South Korea, June 2016.

[57] Wongyu Shin, Jeongmin Yang, Jungwhan Choi, and L. Kim. Nuat: A non-uniform access time memory controller. 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pages 464–475, 2014.

[58] Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

[59] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. Knights landing: Second-generation intel xeon phi product. IEEE Micro, 36(2):34–46, 2016.

[60] Young Hoon Son, O. Seongil, Yuhwan Ro, Jae W. Lee, and Jung Ho Ahn. Reducing memory access latency with asymmetric dram bank organizations. SIGARCH Comput. Archit. News, 41(3):380–391, June 2013.

[61] Young Hoon Son, O. Seongil, Yuhwan Ro, Jae W. Lee, and Jung Ho Ahn. Reducing memory access latency with asymmetric dram bank organizations. In Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, page 380–391, New York, NY, USA, 2013. Association for Computing Machinery.

[62] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture,

HPCA '07, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society.

[63] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 62–75, Dec 2015.

[64] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pages 639–650, Feb 2013.

[65] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. IEEE Trans. Parallel Distrib. Syst., 27(10):3071–3087, October 2016.

[66] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. SIGPLAN Not., 45(3):219–230, March 2010.

[67] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram

efficiency with locality-aware data placement. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, page 219–230, New York, NY, USA, 2010. Association for Computing Machinery.

[68] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. SIGARCH Comput. Archit. News, 38(1):219–230, March 2010.

[69] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. IBM Journal of Research and Development, 46(1):5–25, 2002.

[70] Joel Tendler, J. Steve Dodson, J. S. Fields Jr., Le Hung, and Balaram Sinharoy. POWER4 system microarchitecture. 46:5–25, October 2001.

[71] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[72] David Tawei Wang. Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm. PhD thesis, University of Maryland, University of Maryland, College Park, MD 20742, 4 2005.

[73] Y. Wang, A. Tavakkol, L. Orosa, S. Ghose, N. Mansouri Ghiasi, M. Patel, J. S. Kim, H. Hassan, M. Sadrosadati, and O. Mutlu. Reducing dram latency via charge-level-aware look-ahead partial restoration. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 298–311, 2018.

[74] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S. Kim, Juan Gómez-Luna, Mohammad Sadrosadati, Nika Mansouri-Ghiasi, and Onur Mutlu. FIGARO: improving system performance via fine-grained in-dram data relocation and caching. In 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020, pages 313–328. IEEE, 2020.

[75] F. A. Ware and C. Hampel. Improving power and data efficiency with threaded memory modules. In 2006 International Conference on Computer Design, pages 417–424, Oct 2006.

[76] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In 2009 IEEE 15th International Symposium on High Performance Computer Architecture, pages 79–90, 2009.

[77] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN 1991 Conference on

Programming Language Design and Implementation, PLDI ’91, page 30–44, New York, NY, USA, 1991. Association for Computing Machinery.

[78] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’52, page 996–1008, New York, NY, USA, 2019. Association for Computing Machinery.

[79] M. Xie, D. Tong, K. Huang, and X. Cheng. Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning. In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pages 344–355, Feb 2014.

[80] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000, pages 32–41, 2000.

[81] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Minirank: Adaptive dram architecture for improving memory power efficiency. In 2008 41st IEEE/ACM International Symposium on Microarchitecture, pages 210–221, Nov 2008.