# INTELLIGENT CACHE MANAGEMENT FOR HETEROGENEOUS MEMORY SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Vinson Young

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2019

**INTELLIGENT CACHE MANAGEMENT FOR HETEROGENEOUS MEMORY SYSTEMS**

Approved by:

Dr. Moinuddin Qureshi, Advisor
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Dr. Hyesoon Kim
College of Computing
*Georgia Institute of Technology*

Dr. Aamer Jaleel
NVIDIA Research
*NVIDIA*

Dr. Milos Prvulovic
College of Computing
*Georgia Institute of Technology*

Date Approved: March 14, 2019

## ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Moinuddin Qureshi, for all his guidance and support throughout these years. I could not have asked for a better advisor, and I thank him from the bottom of my heart. I hope to carry on his relentless quest for simple and effective solutions, as well incorporate his focus on clear presentation of work. I would also like to thank Dr. Aamer Jaleel for his mentoring in both academic pursuits as well as personal-life pursuits – I thoroughly enjoyed our many spirited whiteboard discussions and collaborations, as well as appreciate support on non-research topics.

I would like to thank my committee members, the late Dr. Sudhakar Yalamanchili, Dr. Hyesoon Kim, Dr. Milos Prvulovic, for providing valuable feedback on my thesis and for inspiring me with their enthusiasm for the field. I would like to take this chance to express deep gratitude to Dr. Yalamanchili – his kind support was always very appreciated and he will be dearly missed. I always enjoyed interacting with Dr. Kim. I thank Dr. Prvulovic for his insightful comments and help.

I must also thank my labmates Swamit Tannu, Chiachen Chou, Prashant Nair, Gururaj Saileshwar, Sanjay Kariyappa, and Poulami Das, for our fun interactions in both lab and leisure setting. Thank you for always being there for me Swamit. Thanks for mentoring me in both problem solving and paper writing Chiachen and Prashant. Best of luck to the junior members! I hope everyone will do well in their future endeavors.

I would also like to thank my many collaborators. Thank you Dr. Rabin Sugumar and Dr. Giri Chukkapalli for your mentorship while at Cavium – I learned a lot about designing computer chips. Thank you Dr. Evgeny Bolotin for your support while I was interning at NVIDIA. I enjoyed the many interactions with my Intel collaborators – I always appreciate Dr. Alaa Alameldeen's positive attitude, and I enjoyed working hard together with Dr. Zeshan Chishti. I would like to express thanks to Dr. Sergey Blagodurov and Dr. David Roberts for their mentorship while I was doing my first research internship at AMD.

I would like to thank my parents Dr. Shanrong Chou and Dr. Kwo Young for raising me and supporting me. I would like to thank my twin sister Dr. Margaret Young for always being there for me – you are a good role model, healthy competition, friend, and support. I would also like to take this chance to say, I am no longer the least-educated person in my immediate family! ...There are so many Dr. Young in this family. I would also like to thank Melody Shao for her support and understanding throughout this PhD.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## SUMMARY

DRAM caches are important for enabling effective heterogeneous memory systems that can transparently provide the bandwidth of high-bandwidth memories and the capacity of high-capacity memories. This dissertation investigates enabling intelligent cache management for tag-inside-cacheline DRAM cache designs. Such a DRAM cache uses a direct-mapped design, co-locates the tag and data within the DRAM array, and streams out the tag and the data concurrently on an access. This direct-mapped design has been shown to be effective for enabling low latency and bandwidth-efficient tag access; however, such a direct-mapped design can have lower hit-rate and high bandwidth cost to confirm misses. This dissertation proposes simple architectural techniques to improve the hit-rate and bandwidth consumption of such DRAM caches to improve performance of heterogeneous memory systems.

Cache compression can improve hit-rate by fitting more lines in a set, provided the lines are compressible. This dissertation discusses how to enable cache compression for DRAM caches to improve associativity and capacity. Cache compression improves capacity to obtain 7% speedup. This dissertation then shows how to tune cache compression to also provide bandwidth benefits, by compressing adjacent lines together and using location prediction. The bandwidth benefits in addition to capacity benefits enable 19% speedup.

Cache associativity can improve hit-rate by allowing for more flexibility in line placement, and this flexibility can provide performance benefits so long it does not incur significant bandwidth costs. Way-prediction can be used to enable associativity for DRAM caches in a bandwidth-efficient manner; however, traditional way-prediction techniques typically require significant per-line SRAM storage. This dissertation shows that coordinating way-install and way-prediction can enable a storage-efficient ($<$1KB SRAM) and accurate way-prediction. This dissertation proposes three coordinated way-prediction methods: *Probabilistic Way-Steering*, *Ganged Way-Steering*, and *Skewed Way-Steering*. The three techniques enable associativity at low bandwidth cost to provide 11% speedup on average.

Cache replacement policy can improve hit-rate by predicting which lines will have reuse and keeping in the cache longer. This dissertation discusses how to enable replacement policies for direct-mapped caches with two techniques: formulating replacement policies as bypass policy with *Age-On-Bypass*, and exploiting spatial locality to reduce cost of maintaining per-line replacement state with *Efficient Tracking of Reuse*. The combination of the two reduce miss-rate by 10% and reduce bandwidth cost of maintaining replacement state by 70% to achieve 18% speedup on average.

The baseline *Tag-Inside-Cacheline (TIC)* organization is good for hits, but can suffer from needing bandwidth to check DRAM array to confirm misses. Alternatively, a *Tag-Outside-Cacheline (TOC)* organization that stores tags of multiple lines together can save on miss cost. This dissertation discusses how to intelligently combine both organizations in a *TicToc* organization that can provide both good hit and good miss path. Additional enhancements for dirty-bit tracking enables 10% speedup over baseline TIC organization.

# CHAPTER 1
# INTRODUCTION

As the processor industry steps into the many-core regime, the main memory system has become one of the key bottlenecks that limit performance and scalability. Memory systems must scale in both bandwidth and capacity to meet this ever growing data demand. System architects today have the choice of several memory technologies with varying bandwidths and capacities: conventional DRAM, high-bandwidth die-stacked 3D-DRAM, or high-capacity non-volatile 3D-XPoint. To meet both increased bandwidth and capacity needs simultaneously, system architects are now provisioning multiple memory technologies at a time, in what are called *heterogeneous memory systems* (e.g., Intel Knights Landing [1] or AMD Vega [2]). This dissertation focuses on improving the scenario where one memory technology is used as a hardware managed cache for another memory technology, since this configuration enables applications to see the latency and bandwidth of one memory technology and the capacity of another without relying on any software or OS support.

## 1.1 Difficulty: DRAM Caches are Direct-Mapped and Store Tag-Inside-Cacheline

Architecting one memory technology as a hardware managed cache for another technology (e.g., 3D-DRAM [3] in front of DRAM [4]) has a main challenge of organizing and accessing a tag storage of several megabytes. For example, a 2GB DRAM cache can fit 32 million lines, which would need 32-128MB of storage for tags. Practical designs for DRAM caches must therefore organize the tag-store such that the tags can be kept in DRAM (to reduce SRAM storage requirements) and yet the tags can also be obtained with low-latency and low-bandwidth overheads. Tag-stores for large SRAM-based caches are typically implemented by having a serial access for tags before a separate data access. However, in a memory system employing a DRAM-based cache where the cache [3] has similar latency

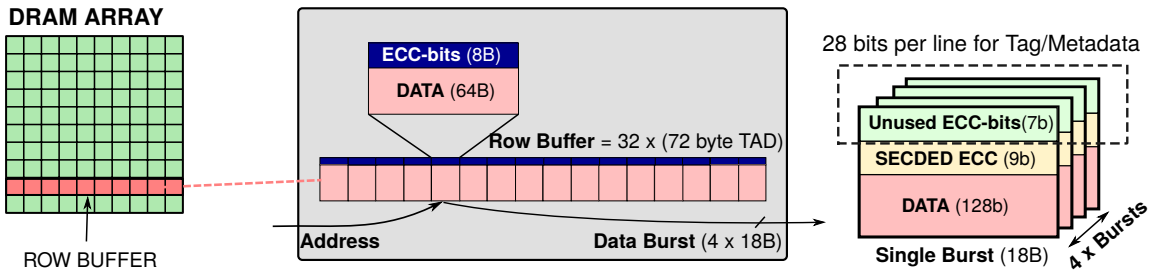



Figure 1.1: Organization of the Tag-Inside-Cacheline (TIC) DRAM cache used in Knights Landing. The DRAM cache is organized at a linesize of 64 bytes, is direct-mapped, and tags are kept with the data-line. On an access, the cache transfers 72 bytes using four bursts on an 18-byte bus (16-bytes for data + 2-bytes for ECC). We need only 9-bits for SECDED on 16-bytes of data, which leaves 7 unused ECC bits in each burst that can be used to store metadata (TIC utilizes these 28 unused ECC bits to store tags).

to memory, this serial tag lookup would incur 2x the bandwidth and latency and consequently perform poorly [5].[1] Instead, practical DRAM cache designs use a direct-mapped organization and store *Tag-Inside-Cacheline (TIC)* [1, 6], such that one access can retrieve both tag and data. A direct-mapped TIC DRAM cache *has good hit-latency*, as it can service cache hits in one DRAM access. However, TIC *incurs bandwidth overhead on cache misses* as such designs needs to probe the tag in DRAM in order to determine a miss. Figure 1.1 shows the tag organization for such a cache – tags are kept in unused ECC-bits and fetched along with data access. We perform our studies on TIC style DRAM caches as such caches perform well and are the approach industry has currently adopted [1].

The baseline direct-mapped TIC DRAM cache design optimizes for hit latency at the cost of lower hit-rate and some bandwidth overhead to confirm misses. This dissertation investigates architectural techniques to improve hit-rate via bandwidth-efficient set-associativity, to improve hit-rate via bandwidth-efficient replacement policies, and to reduce miss-probe bandwidth overhead via alternative tag storage methods. The following sections describe the three major problems.

[1] We show alternative DRAM cache organizations in Section 2.2.

### 1.1.1 Enabling Associativity for DRAM Caches, at Low Bandwidth Cost

While direct-mapped TIC DRAM caches provide low hit-latency, they can suffer from low hit-rate due to conflict misses. Ideally, we want to obtain the hit-rate of associative DRAM caches, without paying the latency and bandwidth costs of increasing associativity.

One way to enable associativity for TIC caches without changing direct-mapped organization is to utilize *cache compression*, to fit multiple compressible lines into a single set. This dissertation discusses how to design cache compression suitable for DRAM caches. DRAM cache compression can be implemented with minor changes local to the memory controller. And, additional tag and metadata bits, which are necessary for interpreting extra compressed lines, can be allocated as needed from the data bits of each line. This dissertation also discusses how to tune cache compression to additionally provide bandwidth benefits, with *Dynamic-Indexing Cache comprEssion (DICE)*. However, we note that these cache compression techniques enable associativity only when lines are compressible. Ideally, we would like to guarantee associativity without depending on compressibility.

Alternatively, we can use *way prediction* on associative TIC caches, to enable associativity at the latency and bandwidth of direct-mapped designs. Unfortunately, conventional way prediction policies typically require per-set storage, causing multi-MB storage overheads for gigascale DRAM caches. This dissertation proposes *Associativity via Coordinated Way-Install and Way-Prediction (ACCORD)*, a design that steers an incoming line to a "preferred way" based on the line address and uses the preferred way as the default way prediction. By coordinating way-install and way-prediction, ACCORD enables a scalable way-prediction framework that can achieve high way-prediction accuracy at low SRAM storage cost. ACCORD enables TIC style DRAM caches to get most of the benefits of associativity, without sacrificing the hit-latency of direct-mapped designs.

### 1.1.2 Enabling Replacement Policies for DRAM Caches, at Low Bandwidth Cost

The second part of the dissertation discusses how to enable intelligent replacement policies for TIC DRAM caches. Cache replacement policies are commonly used to improve the hit-rate of on-chip caches. The most effective replacement policies often require the cache to track and update per-line reuse state to inform their decision. However, there are two fundamental challenges to applying such replacement policies on direct-mapped TIC DRAM caches: (1) replacement policies rely on finding good eviction candidates from multiple ways and can be difficult to apply on direct-mapped caches, and (2) stateful policies would require significant bandwidth to maintain per-line DRAM cache state. To make replacement policies applicable to direct-mapped caches, we propose a *reuse-based bypass* policy called *RRIP-AOB* that can protect predicted-useful lines in direct-mapped caches by bypassing most other lines by default. To reduce the bandwidth cost of maintaining reuse state in DRAM caches, we propose *Efficient Tracking of Reuse (ETR)* that exploits spatial locality to reduce the bandwidth cost of state update. The combination of the two techniques improve hit-rate and reduce bandwidth consumption to enable substantial speedup.

### 1.1.3 Reducing Miss Confirmation Cost for DRAM Caches

The third part of the dissertation discusses how to reduce bandwidth cost to confirm DRAM cache misses. There are currently two major approaches for DRAM cache design: (1) a Tag-Inside-Cacheline (TIC) organization that optimizes for hits, by storing tag next to each line such that one access gets both tag and data, and (2) a Tag-Outside-Cacheline (TOC) organization that optimizes for misses, by storing tags from multiple data lines together in a tag-line such that one access to a tag-line gets information on several data-lines. Ideally, we would like to have the low hit-latency of TIC designs, and the low miss-bandwidth of TOC designs. To this end, this dissertation proposes a *TicToc* organization that provisions both TIC and TOC to get the hit and miss benefits of both. However, we find that naively combining both techniques actually performs worse than TIC individually, because one has

to pay the bandwidth cost of maintaining both metadata. The main contribution this part of the dissertation makes is developing architectural techniques to reduce the bandwidth cost of accessing and maintaining both TIC and TOC metadata. This dissertation proposes several techniques that reduce the bandwidth cost of maintaining dirty bit, maintaining tag bits, and installing lines.

## 1.2 Thesis Statement

DRAM caches suffer from conflict misses and need high bandwidth cost to confirm misses; simple architectural innovation that addresses the challenges of enabling associativity, replacement policy, and reduced miss probes at low bandwidth cost can improve the performance of heterogeneous memory systems.

## 1.3 Contributions

This dissertation makes the following contributions.

- **Associativity** The dissertation shows that cache compression can be used to increase DRAM cache set-associativity opportunistically, and that cache compression can also be used to improve effective DRAM cache bandwidth. Also, the dissertation shows that coordinating way-install and way-prediction policy can enable a scalable way prediction method suitable for gigascale associative DRAM caches.

- **Replacement Policy** The dissertation shows that intelligent replacement policies can be made suitable for direct-mapped caches by implementing them as bypassing policies. The dissertation also shows that spatial locality can be exploited to reduce the bandwidth cost of maintaining replacement state in the DRAM cache.

- **Miss Bandwidth Cost** The dissertation shows that utilizing both Tag-Inside-Cacheline and Tag-Outside-Cacheline tag storage methods simultaneously can improve both hit and miss bandwidth; however, such an approach can suffer from needing to maintain

both tags. The dissertation then proposes a technique to reduce repeated dirty bit updates and a technique to reduce even the initial dirty bit update, to reduce dirty bit tracking costs.

The thesis is organized as follows. Chapter 2 provides background on prior work and related studies. Chapter 3 and 4 presents techniques for enabling associativity in bandwidth-efficient manners via cache compression and way-prediction, respectively. Chapter 5 addresses the challenge of implementing replacement policies in gigascale direct-mapped DRAM caches. Chapter 6 shows presents techniques to reduce bandwidth costs of DRAM cache maintenance and Chapter 7 summarizes dissertation.

# CHAPTER 2
# BACKGROUND

As the main memory system is increasingly becoming a bottleneck, both industry and academia have increased their focus on bridging the gap between processors and memory. This chapter presents prior studies related to heterogeneous memory systems and is organized as follows: (1) available memory technologies of DRAM / 3D-DRAM / 3D-XPoint, (2) DRAM cache architectures, (3) cache compression, (4) associativity and way-prediction, and (5) replacement policies.

## 2.1 Memory Technologies

A system architect has many choices in memory technology today, and can use any combination of the technologies to build heterogeneous memory systems. We discuss the three memory technologies used in this dissertation.

### 2.1.1 Conventional DRAM

First, system architects can provision dynamic random access memories (*DRAM*) [4]. DRAM is a low-cost high-density memory technology in common use today. DRAM is built of a dense arrays of capacitors that store logical 0 and 1 depending on charge, and the capacitors needs periodic refresh to maintain their values. DRAM is typically provisioned in the DIMM form factor and is accessed across DDR channels. DRAM read and write latency usually depends on rowbuffer hit rate (8-16 buffers each 2-4 KB in size) – an access that hits in the rowbuffer will see low latency, whereas an access that misses will see higher latency to close the current row and open the requested row. DRAM bandwidth is typically limited by either bus bandwidth (number of channel pins and bus frequency) or internal bandwidth (number of banks or rowbuffers). DRAM capacities are the current standard for

memory technologies – a typical single-socket server node of 16-48 cores can provision up to 6-8 DDR4 channels each with 2 DIMMs of 4-32GB DRAM (e.g., ~384GB).

### 2.1.2 High Bandwidth 3D-DRAM

Second, system architects can provision emerging high-bandwidth die-stacked memories (*3D-DRAM*) [3, 7]. Improvements in die-stacking and interconnect technology have enabled 3D-stacking several DRAM chips together in small form factor and connecting them to processors with a higher number of pins [3]. Examples of such die-stacked memories, collectively referred to as *3D-DRAM*, include Hybrid Memory Cube [7], High Bandwidth Memory [3], and Multi-Channel DRAM [1]. The read and write latency to access 3D-DRAM are similar to conventional DRAM as the underlying DRAM technology is similar. The bandwidth of 3D-DRAM is on the order of ~4-8X higher bus bandwidth due to increased pin-count. However, 3D-DRAM is more expensive and has lower capacity than conventional DRAM (e.g., ~16GB).

Capacity is a critical need in many systems, so 3D-DRAM is unlikely to be able to outright replace conventional DIMM-based DRAM. However, heterogeneous memory systems employing both 3D-DRAM and conventional DRAM can potentially enable both the bandwidth benefits of 3D-DRAM and capacity benefits of DRAM.

### 2.1.3 High Capacity 3D-XPoint

Third, system architects can provision emerging high-capacity non-volatile memories (*3D-XPoint*) [8]. Non-volatile memories such as 3D-XPoint [9, 10, 8] and PCM [11, 12, 13, 14] are an even denser form of memory or storage technology that stores data based on the current phase/state of the material. Bits are written by physically changing the state of the material with electrical signals, and bits are read by measuring the electrical resistance of the material. Such technologies do not need power to maintain state, and hence constitute a *non-volatile* data storage technology. The read latency of these memories is roughly 4-6x

longer than DRAM, and the write latencies are roughly 20-50x longer than DRAM [9]. The bus bandwidth of such memories are similar to conventional DIMM-based DRAM as they use the same DDR channels [15]. The benefits of such memories are that they offer 4-8X higher capacity than conventional DRAM (e.g., ~1.5 TB).

The higher capacity of 3D-XPoint is attractive in many applications; however, the poor read and worse write latencies may cause performance degradation relative to DRAM main memory. Heterogeneous memory systems can employ 3D-XPoint in conjunction with 3D-DRAM or DRAM technologies [16] to potentially enable the capacity benefits of 3D-XPoint with the latency of DRAM technologies.

## 2.2 DRAM Cache Organizations

Operating one memory technology as a hardware-managed DRAM cache for another technology allows for applications to benefit from both technologies without relying on software or OS support. However, how one organizes the DRAM cache is critical to enabling a good hit-latency, hit-rate, and miss-bandwidth DRAM cache that can be implemented with minimal SRAM storage overhead. We explain prior studies on DRAM caches.

### 2.2.1 Tag-Inside-Cacheline DRAM Caches

Tag-Inside-Cacheline (TIC) designs [1, 6, 17, 18, 19, 20], shown in Figure 2.1(b), generally organize their cache as direct-mapped and store tag inside the cacheline, such that one access can retrieve both tag and data. TIC approaches are optimized for hit latency, at the cost of hit-rate and bandwidth overhead to confirm misses [6]. An enhancement, BEAR [17], proposes several optimizations to reduce bandwidth cost of TIC DRAM cache maintenance: we include its DRAM Cache Presence that targets reducing write probe in our baseline, we compare with Bandwidth-Aware Bypass with 90%-bypass that targets reducing install bandwidth in Chapter 5 and Chapter 6, but, however, we do not include Neighboring Tag Cache that targets reducing miss bandwidth as current implementations

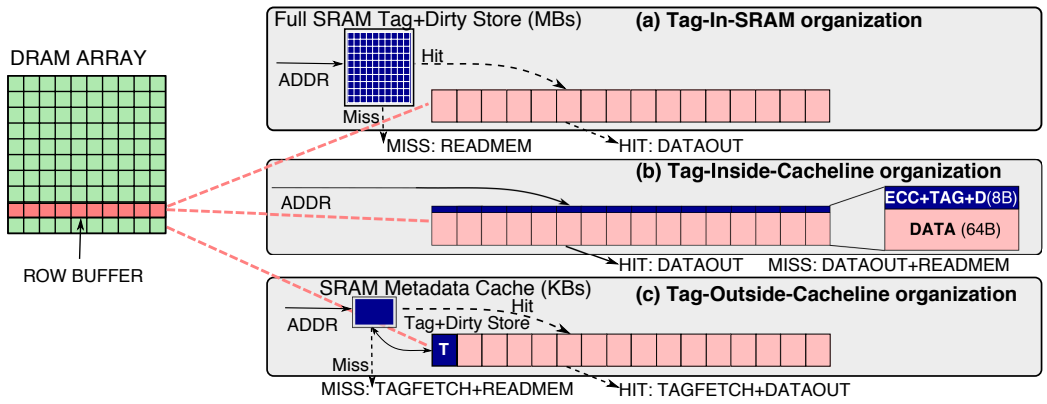



Figure 2.1: DRAM cache organization and flow for (a) idealized Tag-In-SRAM, (b) hit-latency-optimized Tag-Inside-Cacheline (TIC) [6], and (c) miss-bandwidth-optimized Tag-Outside-Cacheline (TOC) [24].

cannot obtain neighboring tag for free [1]. Such hit-latency optimized approaches have been proven effective in industrial application with Intel Knights Landing product [1]; as such, we perform all of our experiments with direct-mapped TIC DRAM cache (with BEAR optimizations) as our baseline.

### 2.2.2 Tag-Outside-Cacheline (TOC) DRAM Caches

Tag-Outside-Cacheline (TOC) designs [5, 21, 22, 23, 24, 25, 26], shown in Figure 2.1(c), store tags in separate tag-only locations in the DRAM cache and fetch them as needed. Such caches are not bound to direct-mapped organizations and can implement high-performance cache optimizations such associativity and intelligent replacement policies. However, they often need separate accesses for tag and data, which constitute significant bandwidth overheads and can limit overall performance. This dissertation perform comparisons to these caches throughout in Chapter 4, Chapter 5, and Chapter 6, and shows how one can enable their associativity, replacement policy, and miss-bandwidth benefits, respectively, for TIC DRAM caches without sacrificing TIC hit-latency. We discuss prior line-granularity and page-granularity implementations of TOC DRAM caches.

*Line Granularity Designs*

The earliest forms of line-granularity TOC DRAM caches were highly associative and would need a serial tag then data lookup (Loh-Hill [5]), which would have poor hit latency and performance. Some enhancements used tag-prefetching (ATCache [21]) or way-prediction (Buffered Way Predictor [22]) to avoid serialized tag lookup. Others used direct-mapped organization (Sim et. al [23]) to avoid serialized tag lookup, with one employing a tag cache (Timber [24]) to reduce the bandwidth of tag lookup as well. Such caches have the potential to amortize miss lookup as one access to a tag-only line can give residency information for multiple lines; however, they can suffer significant bandwidth cost to maintain tags and dirty bit. This dissertation attempts to obtain TOC's miss-bandwidth-reducing benefits while avoiding TOC's metadata maintenance costs in Chapter 6.

*Page Granularity Designs*

Large-granularity TOC DRAM caches have reduced tag and metadata storage overheads [25, 26], and have space to store associativity and replacement metadata. These caches can implement associativity with recency-based replacement [26, 27, 28] or frequency-based replacement [29, 30] that would otherwise be too expensive in TIC DRAM caches. We would like to enable associativity and replacement policies in our DRAM cache designs.

Tags for such caches are stored either in SRAM [25, 31] or in DRAM [26]. Tag-In-SRAM proposals, shown in Figure 2.1(a), typically use sector caching [31] to reduce the overall tag requirements, and fit them all in MegaBytes of SRAM (Footprint Cache [25]). However, the storage for these approaches are still typically quite large. On the other hand, Tag-In-DRAM proposals store metadata in DRAM, and fetch the tags as needed (Unison Cache [26]). These approaches often need to spend bandwidth to access and update metadata information in a separate area of DRAM cache. And, they have the penalty of poor cache utilization when not all lines in a page are used. We would like to avoid such metadata bandwidth overheads and cache-utilization difficulties in our DRAM cache design.

### 2.2.3 Software-supported Hybrid Memories

Software-supported DRAM cache approaches maintain mapping and metadata information inside page tables [27, 28, 29, 30], and use various heuristics to determine when to install pages. The benefit to such approaches is that they do not need to pay additional bandwidth to access tags. The shortcomings of such an approach are two-fold. First, the migration granularity is fixed to the size of OS page, which can cause overfetch problems, as well as poor cache DRAM utilization when not all of the page is useful. Second, such approaches require both hardware and software support, and can be difficult to deploy without cooperation between multiple vendors. This dissertation does not perform comparison with such works as these approaches are out of scope (i.e., breaks design goal of OS-transparency).

### 2.2.4 Swap-based Hybrid Memories

Other hybrid memory approaches attempt to get the capacity of both memories, and instead initiate hardware-managed line or page *swaps* to enable most data to be serviced at the lower-latency or higher-bandwidth memory [32, 33, 34, 35, 36]. These approaches have various tracking overheads and effectiveness. However, we note there is a fundamental difference from caching. On eviction of an unmodified line/page, caches can simply drop the clean line/page – whereas, swap-based approaches need to always write back the evicted line/page. Such swaps incur extra writes that could otherwise have been avoided.

These approaches can work well for 3D-DRAM + DRAM configurations because DRAM does not have constrained writes [2]. However, for a DRAM + 3D-XPoint configuration, these extra swapping-induced writes would cost performance, endurance, and power when writing to write-constrained 3D-XPoint. The added capacity benefits (3-12% increase in capacity) obtained from such swapping are unlikely to make up the difference. This dissertation does not perform comparison with such works as these approaches are out of scope (i.e., breaks design goal of write-efficiency when using 3D-XPoint memories).

### 2.2.5 Mostly-clean DRAM caches to Avoid Dirty-Bit Tracking

Tracking dirty-bit or most-recent-copy of cacheline efficiently with low SRAM storage costs is a known difficult problem. Many works limit the amount of lines that can be kept dirty [37, 38], to reduce SRAM storage costs needed to track dirty lines. Other approaches are more extreme and make the cache clean-only by always writing through [39, 40].

In Chapter 6, this dissertation targets DRAM caching for a DRAM + 3D-XPoint system, which is often constrained by 3D-XPoint write bandwidth. Such mostly-clean caching techniques, which limit the fraction of DRAM cache that can be dirty, hamper the ability for the DRAM cache to act as an effective write buffer for 3D-XPoint. This write limit can cause corresponding degradation in performance, endurance, and power. This dissertation does not perform comparison with such works as these approaches are out of scope (i.e., breaks design goal of write-efficiency when using 3D-XPoint memories).

## 2.3 Compression in Cache and Memory Systems

Compression has been used throughout the memory system to improve SRAM cache capacity and DRAM bandwidth. We show related work, and show how DRAM caches can enable both capacity and bandwidth benefits at minimal cost.

### 2.3.1 Compressing On-Chip SRAM Caches: Additional Capcity

Compression exploits redundancy in data values to increase effective capacity of a given substrate. Prior work has looked at compression for improving the capacity of SRAM caches [41, 42, 43, 44]. As decompression latency is in the critical path of cache access, these proposals use simple compression schemes such as Frequent Pattern Compression (FPC) [45], Base-Delta-Immediate (BDI) [46], CPACK [47], and ZCA [48], that can perform decompression within a minimal number of cycles. We would like to utilize learnings from SRAM cache compression to improve DRAM cache associativity and capacity.

These proposals focus solely on increasing the effective capacity of the cache as a means to improve performance. However, if the cache is large enough to hold the uncompressed working set of the applications, then these proposals would not provide any performance benefit. As the size of the DRAM cache is typically quite large compared to an SRAM cache, prior schemes [49, 50] that focus solely on increasing the capacity of DRAM cache can have limited performance benefit. As such, this dissertation investigates utilizing cache compression to improve DRAM cache bandwidth as well.

#### 2.3.2 Compressing Main Memory: Additional Bandwidth

Compression has also been applied to main memory for increasing memory capacity [51] and bandwidth [52, 53]. For example, Linearly Compressed Pages [52] proposes to spatially compress all lines in a page to one-fourth their size, such that one DRAM access can potentially get 4 lines worth of data at a time. If all 4 lines obtained are useful, this can result in 4x effective bandwidth improvement. This dissertation exploits the insight that such spatial compression can improve bandwidth, and proposes a cache compression scheme that can improve bandwidth as well.

### 2.4 Associativity at Low Bandwidth Cost

This dissertation discusses prior work on enabling associativity via way prediction or modified indexing, and show how such approaches fail to scale to gigascale DRAM caches.

#### 2.4.1 Conventional Way-Predictors: Challenge in Scaling to Gigascale DRAM Caches

Way Prediction has been proposed for SRAM caches (L1 or LLC) for reducing latency and power. If we can predict the location of the data-line and retrieve it in one access, we can avoid the tag serialization penalty.

For L1, tracking the most-recently-used way in each set (MRU Pred [54, 55]) provides high accuracy as the access stream of an L1 cache has high temporal locality. Unfortu-

nately, such locality is usually not visible to secondary caches (L2/L3/L4), so MRU prediction tends to be less effective.

Secondary caches can use a partial-tag (e.g., 4-bits) design for each line to avoid cache lookup [56, 57, 58, 59, 60, 61]. Unfortunately, both the MRU Pred and the Partial-Tag design require per-line or per-set storage. Therefore, these schemes would incur a large storage overhead of several megabytes for a 4GB DRAM cache.

If we are to implement way prediction effectively at the size of DRAM caches, we need a way-predictor that is both accurate and requires less storage overhead than prior designs.

### 2.4.2 Multiple-Indexed Caches

*Hash-Rehash* and *Column-Associative caches* (CA-cache) [62, 63] serially check two indices (preferred and alternate) in a direct-mapped cache and moves the line physically to the preferred index, if a cache hit happens at the alternate index. Thus, a hit to the preferred index can be serviced in one access, whereas a hit to the alternate index requires not only two accesses but also the bandwidth overhead of a swap between the preferred and the alternate index. The probability of CA-cache to obtain a hit with a single access is similar to having a two-way cache with MRU-based way predictor (see Table 4.1). However, CA-cache invokes significant bandwidth consumption in swaps, even when the cache is not sensitive to increased associativity, which limit its performance.

Prior work proposed *skewed associative cache* [64, 65] design that allows a cache line to reside in two possible locations based on a hash of memory address. If the first hash of two lines collide, it is unlikely the second hash location will also collide. Such caches uses parallel lookup of two locations, and can waste bandwidth when scaled to DRAM cache. Nonetheless, its concept of skew associativity is useful for increasing cache utilization.

## 2.5 Replacement Policies

Computer designers look for techniques that can improve performance, increase energy-efficiency, and reduce the off-chip bandwidth bottleneck. Quite often, a solution that improves one of these metrics ends up worsening the other metrics. However, an intelligent cache replacement policy is one such optimization that can simultaneously provide all three benefits: it can improve system performance by reducing the long-latency memory accesses, increase energy efficiency by servicing data on-chip at lower energy, and mitigate the pressure on the off-chip memory systems. We would like to enable such benefits for our DRAM caches. We discuss common replacement policies and their applications.

The commonly-used *Recency*-based replacement policies (Least Recently Used [66, 67, 68]) installs incoming lines with highest priority, based on the heuristic that recently-used lines are more likely to be re-used. Recency-based replacement policies work well for L1 caches, but they tend to be not as effective for the Last Level Cache (LLC). This is because the LLC receives an access stream that has filtered temporal locality.

Instead, LLC are often subject to significant thrashing patterns (e.g., cyclic ABCABC patterns on 2-way cache) that can cause recency-based policies to actually end up with 0% hit-rate. For such cases, global *Probabilistic* replacement policies [69, 70] can choose to install most lines at low priority to protect the current working set and maintain some hit-rate under thrashing conditions. However, such global policies are coarse-grain and can miss out on per-line behavior.

Over the last decade, there has been significant research work on intelligently managing the LLC by accurately predicting the re-reference interval of cache lines with *Frequency*-based replacement policies[71, 72, 73, 74, 75] or *Reuse*-based replacement policies[76, 77, 78, 79, 80]. Such works rely on maintaining reuse counters per each LLC line and protecting lines that are predicted to have high reuse. Such policies are resilient to thrashing and scans (an access pattern where a workload accesses a large number of lines and never

reuses them). Many of the most effective replacement policies are reuse based [76, 81, 82, 83, 84, 85]. We would like to implement such intelligent policies on DRAM caches; however, there are difficulties in implementing such policies on DRAM caches.

*Difficulty of applying such policies on DRAM caches*: We would like to obtain the hit-rate benefits of intelligent replacement policies for our DRAM caches. However, cache replacement policies are typically discussed in the context of a set associative caches, as the set contains multiple lines and there is a choice of the line to evict. Our baseline direct-mapped DRAM cache only has one choice in selecting victim, and can be difficult to apply such policies on. This dissertation needs to solve the dual problem of applying replacement policies on direct-mapped caches, as well as the problem of maintaining significant per-line reuse state for gigascale DRAM caches.

# CHAPTER 3

# ENABLING DRAM CACHE COMPRESSION WITH DICE

This chapter first discusses how to design cache compression suitable for DRAM caches to improve cache *associativity* and *capacity*, and then discusses how to enhance DRAM cache compression to improve *bandwidth* as well [20]. The baseline direct-mapped TIC DRAM caches is effective for hit-latency, but it can be subject to significant conflict misses. Cache compression is a promising method that can reduce conflict misses by opportunistically improving the associativity and capacity of caches when lines are compressible [41, 42, 43, 44]. However, cache compression techniques have primarily been developed for SRAM caches, and often need excessive tag management bandwidth if directly applied to DRAM caches designs. We would like to develop a practical and effective compressed cache architecture suitable for direct-mapped TIC DRAM caches that can improve cache associativity/capacity and bandwidth without too much added complexity.

## 3.1 Motivation: Potential of DRAM Cache Compression

### 3.1.1 DRAM Cache Performance Sensitive to Both Capacity and Bandwidth

Doubling the capacity of the DRAM cache could potentially provide an improvement of about 10%, on average. We note that DRAM caches are provided mainly to improve the system bandwidth. If we could use compression to increase the effective bandwidth of the DRAM cache as well, then we could get even higher performance benefit. For example, doubling both the capacity and bandwidth of the DRAM cache can provide 22% performance on average. As such, we aim to develop a cache compression architecture that can provide both capacity and bandwidth benefits.

### 3.1.2 DRAM Cache Compression is Nearly Free

We find that compressed DRAM caches are in a unique position to achieve both capacity and bandwidth benefits, with no software involvement, and with minimal cost / complexity, as shown in Table 3.1.2. Compressed SRAM caches can improve cache capacity/associativity without OS support, but often needs complicated tag access and management. Compressed main memory can improve both memory capacity and bandwidth, but current proposals need OS support to obtain both benefits. Compressed DRAM caches, on the other hand, can potentially improve both cache capacity and bandwidth, without the need for OS support, and without complicated tag management: they can utilize main memory's spatial compression [52, 53] to improve bandwidth, they inherently do not need OS support due to hardware-managed caching, and they do not need extra storage for storing extra tags (the extra tag entries needed for compression can be created dynamically within the DRAM array, as the memory controller has the freedom to interpret any bit as either a tag bit or a data bit). We discuss how to design DRAM cache compression to get capacity and bandwidth benefits with minimal complexity/cost next.

Table 3.1: Comparison of different forms of compression

| Module to Compress | On-Chip Cache | Main Memory | **DRAM Cache** |
|---|---|---|---|
| Improve Capacity Only | Yes | No | No |
| OS Support Needed | No | Yes | No |
| Tag Overhead | Yes | No | No |

## 3.2 Design: Compressing DRAM Cache for Capacity

### 3.2.1 Design Overview: Organization and Working

As practical DRAM caches use the same DRAM array for storing tags as well as data, we can simply architect a compressed DRAM cache by providing L4 cache controller with compression and decompression logic as shown in Figure 3.1. In our compressed DRAM cache design, only the L4 is compressed, and the data in other parts of the memory system

remain in normal uncompressed form; thus, cache compression can be implemented local to L4 cache controller without requiring changes to the other parts of the system.

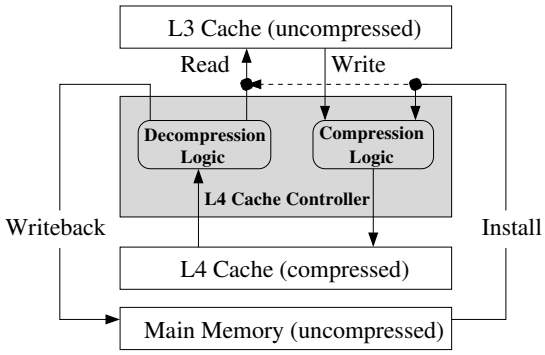



Figure 3.1: Design of compressed DRAM-cache. Changes are limited to L4 controller.

On an L3 read from the compressed TIC L4 cache, the L4 controller obtains a 72B compressed set (including ECC). Decompressing the set can provide multiple lines with a single access. The system can decide to install these lines in L3, or only requested lines.

The L4 cache controller compresses data before L3 writebacks to L4, and before L4 installs from main memory. For writes, the L4 cache controller compresses the data to see how much space is required to store the line, and reads from L4 cache to check what lines are resident. If the compressed line can be stored in the unused space of the set, it is appended and written. If the compressed line cannot fit, then resident entries are evicted (written back to memory if dirty) until enough space is made available for the line.

### 3.2.2 Flexible Tag Format:

To enable compression, we need not change the organization of the DRAM cache. The DRAM cache provides 72B per set (including ECC). It is up to the memory controller to interpret those 72B as either tag or data. For supporting compression, we use a format that allows the number of tag-store entries to increase dynamically to accommodate storing extra lines with the same set. We implement this by having one bit per tag denoting that the next 4B should be interpreted as tag or data.

Figure 3.2 shows the format of the tag and data entry used in our design. For compressed design, each tag entry has a *Next Tag Valid* bit to inform whether the next 4B is

tag or data. This allows us to store arbitrary number of tags. A *BAI* bit is added to distinguish the direct-mapped line vs. an adjacent line that is spatially compressed together with it (more on this in Section 5). A *Shared tag* bit is used to save tag space when spatially contiguous lines are compressed in the same index [43, 42]. We use up to 9 bits for compression algorithm metadata (FPC/BDI). Our design can accommodate up to 28 compressed lines per set.

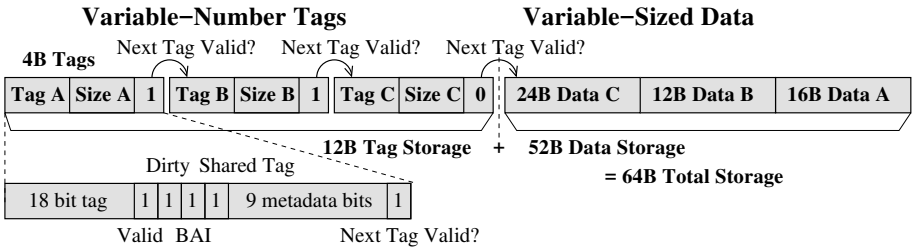



Figure 3.2: Accommodating multiple compressed lines within a DRAM cache set.

### 3.2.3 Speedup from Compression for Capacity

Our compressed cache design tries to accommodate more lines in each set of DRAM Cache, if data is compressible. For our baseline direct-mapped TIC DRAM Cache, we assume that each set is determined by the conventional cache indexing scheme that places consecutive lines in consecutive sets. We call this set selection as *Traditional Set Indexing (TSI)*. Lines that map to the same set under TSI are separated by several GB in physical memory. If these lines can be compressed, then they can reside in the same cache set, and an access to the set will obtain these lines with a single access. Unfortunately, lines that are spatially far away are unlikely to be accessed within a short period of each other, and should not be installed in the L3 cache. Thus, this form of compression is purely for capacity benefits, and not for bandwidth. Figure 3.3 shows performance improvement of a cache compressed with TSI. Unfortunately, compression for capacity alone has limited performance benefit, as it provides a speedup of 7%. We observe that compressing for both capacity and bandwidth has higher potential for speedup. Therefore, we seek a design that can potentially improve bandwidth as well.

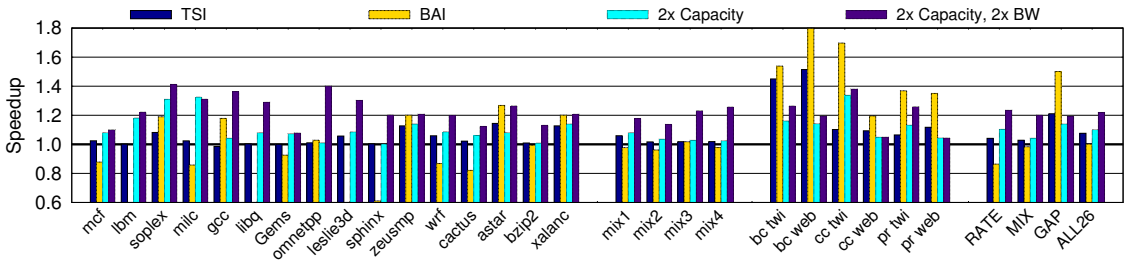



Figure 3.3: Speedup from Traditional Set Indexing and Bandwidth-Aware Indexing, compared to doubling the cache capacity and bandwidth. BAI improves bandwidth for compressible workloads but causes slowdown for others due to thrashing.

## 3.3 Design: Compressing DRAM Caches for Bandwidth

### 3.3.1 Insight: Cache Indexing Can Improve Associativity Or Bandwidth

**Indexing for Associativity:** Cache indexing plays an important role when compressing direct-mapped TIC DRAM caches. We explain the considerations in compressing DRAM caches with an example. Figure 3.4(a) shows the baseline uncompressed cache storing lines A-D. The baseline uses *Traditional Set Indexing (TSI)* that maps consecutive lines to consecutive sets. There are four more lines (W-Z) in the working set that are used less frequently than A-D. A straight-forward method to compress DRAM caches is to compress the lines that map to the same set together, if they both can be compressed to within the same set. If the data is compressible, we can expect all the eight lines (A-D and W-Z) to be resident in the cache, as shown in Figure 3.4(b). A single access to the cache can obtain two lines (e.g., A and W). However, even though we can get two lines with one access, such a design compresses purely for capacity, as two lines mapping to the same set (A and W) would spatially be GBs apart in main memory – and hence be unlikely to be used within a short period of each other.

**Indexing for Bandwidth:** To obtain higher effective bandwidth, it would be desirable to obtain two spatially adjacent lines in one access from the DRAM cache. We can implement this by changing the cache indexing such that two consecutive lines map into the same set. A simple method to have two consecutive lines map in the same set is to ignore

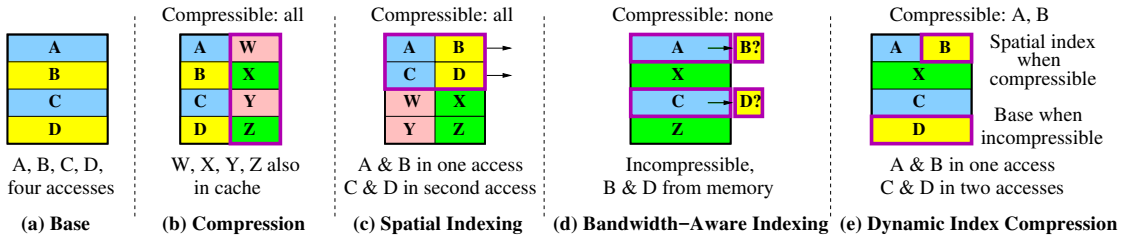
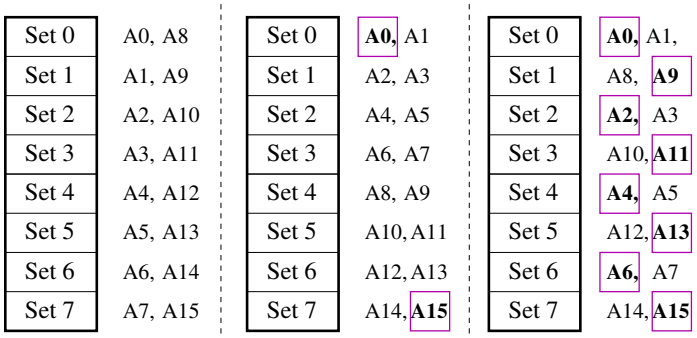



Figure 3.4: Considerations in compressing DRAM caches (a) Baseline system with four lines (A-D) (b) Compression for capacity (c) Spatial indexing for bandwidth (d) Slowdown from spatial indexing when data is incompressible (e) Dynamic index compression based on compressibility (A, B use spatial index)

the least significant bit of line address while indexing the cache, as in Figure 3.4(c) and Figure 3.5(b). We call such a method of cache indexing as *Naive Spatial Indexing (NSI)*. When lines are compressible, on an access to A, both line A and line B are received with one access, improving effective bandwidth.

However, when lines are incompressible, the spatially close lines fight for space in the same set, degrading performance (by as much as 63% in the studies). We explain shortcoming of NSI with an example. Figure 3.5 shows a cache with 8 sets, labeled Set 0 to Set 7. We have a workload with sixteen consecutive lines A0-A15, where lines A0-A7 are frequently accessed. Figure 3.5(a) shows the mapping with TSI and Figure 3.5(b) shows mapping with NSI. When lines are compressible, both TSI and NSI can fit all 8 frequently used lines (A0-A7), and NSI can stream out these lines in half the number of accesses. Unfortunately, if lines are incompressible, NSI can accommodate only four lines out of A0-A7 at any time, causing thrashing. This thrashing can degrade performance of NSI to worse than that of uncompressed cache, which is undesirable.

As such, we aim to have a dynamic policy to switch between TSI and NSI depending on compressibility, to get capacity and bandwidth when lines are compressible, but avoid slowdown when lines are incompressible.




(a) TSI (b) NSI (c) BAI

Figure 3.5: Mapping 16 consecutive lines A0-A15 in a cache with 8 sets under (a) TSI (b) NSI (c) BAI. Purple boxes indicate lines that remain in the same set as TSI.

### 3.3.2 Flexible Bandwidth-Aware Indexing (BAI)

However, switching between NSI and TSI is costly, as nearly all the lines are in different positions, as shown in Figure 3.5(b). To address this, we propose *Bandwidth-Aware Indexing (BAI)* that ensures consecutive lines map to the same set, while half of lines retain the same position as in TSI, as shown in Figure 3.5(c). BAI is calculated by using a tag bit as the least significant index bit. BAI retains capacity and bandwidth benefits of NSI when lines are compressible, as consecutive lines map to same set. And, it allows quick switching to TSI when lines are incompressible. Another key feature of BAI is that BAI is guaranteed to be either the same set, or neighboring set as under TSI. Thus, both locations of the line (under BAI or TSI) are guaranteed to be in the same row buffer.

### 3.3.3 Effectiveness of Bandwidth-Aware Index

BAI can get benefits of both capacity and bandwidth – as multiple lines obtained from a single cache access are likely to be useful, reducing accesses to the DRAM cache. Unfortunately, when lines are incompressible, BAI performs poorly compared to TSI. For example, if lines are incompressible, in Figure 3.4(d), only one of A and B can be resident at a time, and the other would be fetched from memory, degrading performance.

Figure 3.3 compares the speedup from BAI with TSI, and also to doubling the cache capacity and bandwidth. BAI improves performance significantly for compressible work-

loads such as *soplex*, *gcc*, *zeusmp*, and *astar*. This happens because compression allows the cache to have more effective capacity and bandwidth. Unfortunately, for workloads such as *mcf*, *lbm*, *libq*, and *sphinx*, there is significant performance degradation with BAI, as spatially contiguous lines end up fighting for the same set.

Ideally, we would like to use BAI as much as possible when lines are compressible to improve both bandwidth and capacity, but use TSI when lines are incompressible to avoid slowdown, as in Figure 3.4(e). To this end, we propose a dynamic indexing scheme for compressed caches that can adapt cache indexing based on data compressibility.

## 3.4 Design: Compressing DRAM Caches for Both Capacity and Bandwidth

### 3.4.1 Insight: Dynamic Indexing for Improving Bandwidth and Enabling Robust Performance

We develop a compressed cache indexing policy that maximizes both bandwidth and capacity while ensuring no performance degradation compared to baseline uncompressed cache. To do so, we propose a dynamic indexing scheme called *Dynamic-Indexing Cache Compression (DICE)* that switches between two indexing policies, Traditional Set Indexing (TSI), and Bandwidth-Aware Indexing (BAI) depending on data compressibility. We present overview and working of DICE, then discuss effectiveness of our solution.

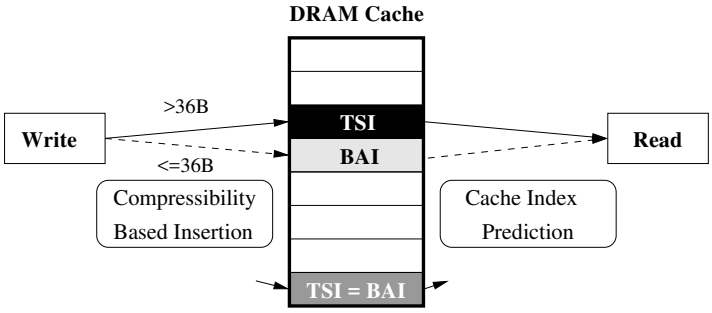



Figure 3.6: Design of DICE. DICE is implemented by deciding index policy on write, and predicting index policy on read.

### 3.4.2 DICE: Overview

DICE allows the cache to adapt its indexing scheme between TSI and BAI. Therefore, a given cache line can be present in either of the two locations, determined either by TSI or BAI. Fortunately, our BAI scheme is designed such that both of these sets would be either neighboring to each other, or be the same set, as shown in Figure 3.6. On a write access (due to install or writeback), we must decide which indexing policy to use. We develop a compressibility-based scheme to make this decision. Similarly, on a read access, we predict which indexing scheme is likely to have been used, and access that location. The effectiveness of DICE depends on developing simple and effective mechanism for deciding index policy on writes and predicting index policy on reads. Note that, given BAI is designed such that the set index of 50% of the lines remain invariant between BAI and TSI, we need to decide insertion index (on write) and predict index policy (on read) for only the remaining 50% of the lines.

### 3.4.3 Deciding Cache Index Policy on Insertion

To decide the index policy at insertion, we leverage the observation that lines within a page are usually compressible to similar sizes [52]. As BAI gets benefits of both capacity and bandwidth when lines are compressible, we want the insertion policy to favor BAI when two lines are likely to compress together. If a line compresses to $\leq$ 36B, its neighboring line is also likely to compress to $\leq$ 36B. In these cases, we insert into BAI. Conversely, if a line is incompressible (say it compresses to 60 bytes), then its neighboring line is unlikely to be able to be compressed with it, so we insert using TSI.

We propose a simple mechanism that selects insertion policy based on size of compressed line. If a line compresses to $\leq$*Threshold*, we insert the line using BAI. Otherwise, we insert it using TSI. We study different threshold values for deciding index policy and determine that a threshold of 36B provides the best performance. In our studies, we use a default threshold of 36B. Sensitivity to threshold is performed in Section 3.6.3.

### 3.4.4 Cache Index Prediction (CIP)

As DICE makes the decision to use either TSI or BAI at insertion time, a cache can become mixed with some lines using TSI and others using BAI. To retrieve a line in the cache, we may need to look up both the possible locations. Unfortunately, doing so would consume more bandwidth and incur high latency. We develop a predictor to determine which location to access first.

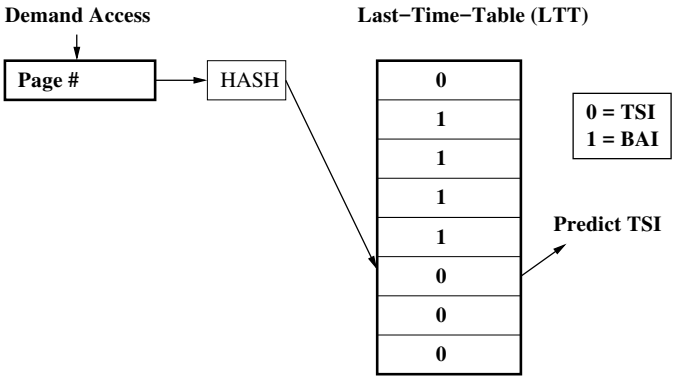



Figure 3.7: History-based Cache Index Predictor. CIP tracks history at page granularity.

As misprediction incurs extra latency and bandwidth, we would like rate of misprediction to be low. We develop *Cache Index Predictors (CIP)* for reads and writes that can accurately predict cache index with low storage overhead. For reads, we design a page-based CIP predictor that uses last-time information for predictions, as shown in Figure 3.7. CIP leverages observation that lines within a page have similar compressibility [52]. If a page is compressible, lines within that page will likely be in BAI. CIP contains a *Last Time Table (LTT)* that tracks last outcome for a page. Given LTT has limited entries, we hash page address to index LTT.

We vary the number of entries in LTT and find the accuracy increases from 93.2% (512 entries) to 94.1% (8192 entries). For reads, we use a default LTT of 2048 entries (256B), which has an average accuracy of 93.8%. For writes, we predict index based on compressibility of data (same as insertion policy), which has an accuracy of 95%.

## 3.5 Methodology

### 3.5.1 Configuration

We use USIMM [86], an x86 simulator with detailed memory system model. We modified USIMM to include a DRAM cache. Table 3.2 shows the configuration used in our study. We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache), with 64B line size. We model a virtual memory system to perform virtual to physical address translations.

Table 3.2: Baseline Configuration

| **Processors** | |
|---|---|
| Number of cores | 8 cores |
| Core type | 4-wide 3.2GHz out-of-order |
| L1/L2 (private) | 32KB/256KB (8-way each) |
| L3 cache (shared) | 8MB (1MB per core) |
| **DRAM Cache** | |
| Capacity | 1GB |
| Configuration | 4 channel, 128-bit bus |
| Bus Frequency | 800MHz (DDR 1.6GHz) |
| Banks | 16 banks per channel |
| tCAS-tRCD-tRP-tRAS | 44-44-44-112 CPU cycles |
| Read/Write Queue | 96 entries per channel |
| **Main Memory (DDR DRAM)** | |
| Capacity | 32GB |
| Configuration | 1 channel, 64-bit bus |
| Bus Frequency | 800MHz (DDR 1.6GHz) |
| Banks | 16 banks per channel |
| tCAS-tRCD-tRP-tRAS | 44-44-44-112 CPU cycles |
| Read/Write Queue | 96 entries |

We use direct-mapped TIC DRAM cache (Alloy Cache [6]) for the L4 cache, and results are normalized to direct-mapped TIC DRAM cache unless stated otherwise. Cache misses fill all levels of the hierarchy. We assume a heterogeneous memory system with DRAM cache using HBM technology [3] and main memory using conventional DDR-based DIMM technology, corresponding to $1/8^{th}$ scale of Knights Landing [87]. In accordance with stacked memory specifications, we assume same access latency for both DRAM technologies. However, the bandwidth of stacked-DRAM is 8x higher than main-memory, with 4x channels and 2x bus width.

### 3.5.2 Workloads

We use a representative slice of 4 billion instructions selected by PinPoints [88], for benchmarks from SPEC 2006 and GAP [89]. For SPEC, we perform studies on all the 16 benchmarks that have at least 2 Miss Per Thousand Instructions (MPKI) out of L3 cache. In addition, we run GAP suite (Graph Algorithm Platform) to show server workloads with real data sets (twitter, web sk-2005). We run all 30 suggested configurations, and present a sample where speedup is representative of GAP suite. We perform evaluations by executing benchmarks in rate mode, where all eight cores execute the same benchmark. In addition to rate-mode workloads, we also evaluate four 8-thread mixed workloads, which are created by randomly choosing 8 out of the 16 SPEC benchmarks. Table 3.3 shows the L3 miss rates and footprints of the 8-core rate-mode workloads used in our study.

Table 3.3: Workload Characteristics

| Suite | Workload (8-copies) | L3 MPKI | Footprint |
|---|---|---|---|
| SPEC | mcf | 53.6 | 13.2 GB |
| | lbm | 27.5 | 3.2 GB |
| | soplex | 26.8 | 1.9 GB |
| | milc | 25.7 | 2.9 GB |
| | gcc | 22.7 | 264 MB |
| | libq | 22.2 | 256 MB |
| | Gems | 17.2 | 6.4 GB |
| | omnetpp | 16.4 | 1.3 GB |
| | leslie3d | 14.6 | 624 MB |
| | sphinx | 12.9 | 128 MB |
| | zeusmp | 5.2 | 2.9 GB |
| | wrf | 5.1 | 1.4 GB |
| | cactus | 4.9 | 3.3 GB |
| | astar | 4.5 | 1.1 GB |
| | bzip2 | 3.6 | 2.5 GB |
| | xalanc | 2.2 | 1.9 GB |
| GAP | bc twitter | 69.7 | 19.7 GB |
| | bc web | 17.7 | 25.0 GB |
| | cc twitter | 93.9 | 14.3 GB |
| | cc web | 9.4 | 16.0 GB |
| | pr twitter | 112.9 | 23.1 GB |
| | pr web | 16.7 | 25.2 GB |

We perform timing simulation until all benchmarks in the workload execute at least 4 billion instructions each. We use weighted speedup to measure aggregate performance of

the workload normalized to baseline. We use geometric mean to report average speedup across workloads, and use RATE to denote average over 16 spec rate-mode workloads, MIX for the 4 mixed workloads, GAP for its 6 workloads, and ALL26 to denote average over all 26 workloads.

## 3.6 Results

### 3.6.1 Impact on Performance

Figure 3.11 shows the speedup of cache compression with TSI, BAI and DICE, and compare it with a cache that has double the capacity and bandwidth. Recall that compressing with TSI provides only capacity benefits and not bandwidth. Therefore, compression with TSI provides a small benefit of 7% on average. We observe that TSI always provides a hit rate that is better than or equal to an uncompressed cache, so no workloads have slowdown.

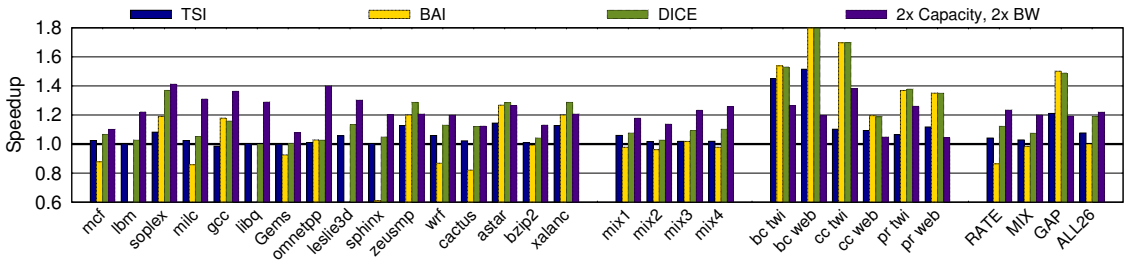



Figure 3.11: Speedup of compressing DRAM cache with (a) TSI, (b) BAI, and (c) DICE with respect to double capacity and bandwidth. DICE provides a speedup of 19.0%, whereas doubling the capacity and bandwidth provides 21.9%.

Compression using BAI tries to get both higher capacity and higher bandwidth. Therefore, there is significant performance improvement for workloads such as *gcc* and *cc twi*, where optimizing for capacity alone provided negligible benefits. Unfortunately, for workloads such as *lbm* and *libq*, the data is incompressible and the increased contention due to the indexing of BAI causes significant increase in cache misses, resulting in performance degradation. Overall, BAI performs similar to baseline (0.1% speedup), on average.

With DICE, the cache performs as well as BAI when BAI performs well, and similar

to TSI for incompressible workloads, causing no degradation compared to baseline. In addition, there are several standouts (such as *soplex*, *leslie3d*, *zeusmp*, *wrf*, and *cactus*) when DICE performs better than either BAI or TSI independently, as it is able to use BAI for compressible regions of memory, and TSI for incompressible regions of memory. The dynamic selection of DICE helps it to outperform the two static indexing schemes. Overall, our DICE design incurs an SRAM overhead of less than 1 kilobyte yet provides 19.0% speedup, nearing the performance of a double-capacity double-bandwidth DRAM cache.

### 3.6.2 Distribution of TSI and BAI with DICE

With DICE, the cache can use TSI, or BAI, or a combination of TSI and BAI across the cache sets. Figure 3.12 shows the distribution of BAI and TSI. We separate the cases where location of the line remains invariant between BAI and TSI (50% of lines). From the remaining lines, we see a skew of 52% towards TSI and 48% towards BAI. This is due to incompressible workloads such as *libq* that cause almost the entire cache to use TSI to avoid performance degradation.

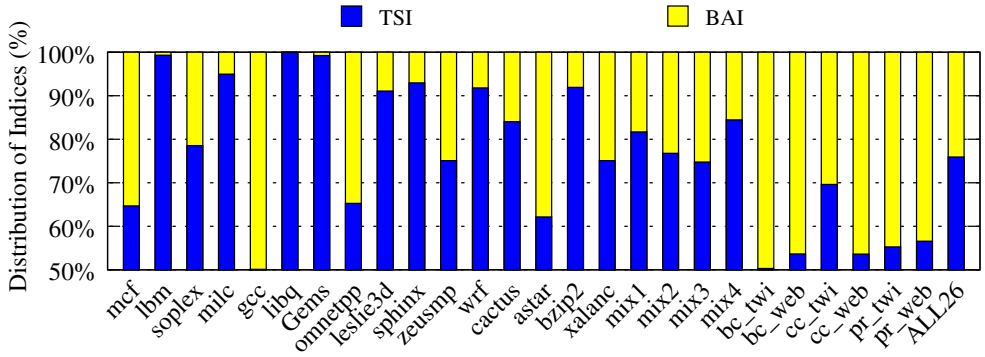



Figure 3.12: Distribution of BAI and TSI for a cache compressed with DICE. Note that for 50% of accesses, we do not need to make install decisions or do index prediction as TSI and BAI refer to the same set, hence the y-axis starts at 50%.

### 3.6.3 Sensitivity to Insertion Threshold

DICE uses compressibility of data to determine index policy on insertion. We use a default threshold of 36B to determine if BAI or TSI should be used. Table 3.4 shows the speedup

of DICE as the threshold is changed from 32B to 36B to 40B. Note that a threshold of 0 will degenerate DICE to always use TSI, and a threshold of 64 will degenerate DICE to always use BAI. We find that the performance is maximized for a threshold of 36B. This is because BDI often compresses a single line to 36B, but double-line compresses it to 68B, which can fit in BAI if the tags are shared.

Table 3.4: Sensitivity to DICE threshold

| | $\leq$ 32B | $\leq$ 36B | $\leq$ 40B |
|---|---|---|---|
| SPEC RATE | +10.6% | +12.2% | +11.1% |
| SPEC MIX | +6.4% | +7.5% | +7.4% |
| GAP | +47.6% | +48.9% | +49.1% |
| GMEAN26 | +17.5% | +19.0% | +18.3% |

### 3.6.4 Impact on DRAM Cache Capacity

Compression increases effective capacity of cache by storing more lines within the same physical space. Table 3.5 shows average capacity of DRAM cache when compressed with TSI, BAI, or DICE. We estimate effective capacity by checking number of valid lines in each set every 50M instructions.

Table 3.5: Effective Capacity of TSI/BAI/DICE

| | TSI | BAI | DICE |
|---|---|---|---|
| SPEC RATE | 1.07x | 1.16x | 1.13x |
| SPEC MIX | 1.12x | 1.28x | 1.24x |
| GAP | 2.00x | 5.57x | 5.06x |
| GMEAN26 | 1.24x | 1.69x | 1.62x |

DICE and BAI have higher compression ratios than TSI due to two reasons: First, TSI compresses lines from different pages within the same set. These lines are less likely to have similar compressibility. DICE and BAI, on the other hand, often compress together lines from the same page, which are likely to have similar compressibility and hence are more likely to fit within the same set. Second, DICE and BAI improve compression ratio due to tag and base-sharing (BDI), which amortizes tag and metadata overhead. DICE increases effective cache capacity by 62%, on average.

### 3.6.5 Impact of DICE on Hit-Rate of L3

If data is compressible, then BAI can provide two spatially-contiguous lines with a single access to L4 cache. As these lines are spatially close, we install both lines in the L3 cache as they are likely to be used within a short period of each other, improving L3 hit rate.

Table 3.6 shows the hit rate of the L3 cache for a baseline system (uncompressed L4), and a system using DICE. For the baseline system, the average L3 hit rate is 37.0%, and it is improved to 43.6% with DICE. Thus, the adjacent lines obtained due to compression with DICE are useful, and installing them in the L3 cache provides performance benefits.

Table 3.6: Effect of DICE on L3 hit rate

| | BASE | DICE |
|---|---|---|
| SPEC RATE | 34.7% | 43.0% |
| SPEC MIX | 61.6% | 67.2% |
| GAP | 26.9% | 29.4% |
| AVG26 | 37.0% | 43.6% |

### 3.6.6 Comparison to Larger Fetch for L3

DICE can send adjacent lines from the L4 cache proactively to the L3 cache. While this may have some resemblance to nextline prefetch or 2x-width line fetch, we note that there is a fundamental difference. DICE sends the adjacent line from L4 to L3, only when that line is obtained without any bandwidth overheads. However, prefetches result in an independent cache request which incurs extra bandwidth. We compare our proposal, with alternative designs for L3 cache that try to either get a wider granularity line in the L3 cache (128 bytes, with two separate 64 byte requests) or next line prefetching in the L3 cache (demand request is followed by a prefetch for the next line).

Table 3.7 shows the performance of wide-granularity fetch at L3 cache, next-line prefetch in L3 cache, and compare it with DICE (in L4) and a combination of DICE (in L4) plus next-line prefetch in L3 cache. We find that designs that simply try to get an extra line in the L3 cache (due to wider fetch or next-line prefetch) give marginal benefits of 1.9% and

1.6%, on average. DICE, which inherently provides an extra line to the L3 cache when such a line is obtained without bandwidth overheads, provides speedup of 19.0%. Nonetheless, the L3 optimizations are orthogonal to DICE and can be combined for greater benefit. For example, using DICE with next line prefetch increases speedup to 20.9%.

Table 3.7: Comparison of DICE to Prefetch

| | 128B-PF | Nextline-PF | DICE | DICE+NL |
|---|---|---|---|---|
| SPEC RATE | +3.2% | +2.6% | +12.2% | +16.7% |
| SPEC MIX | +1.2% | +1.9% | +7.5% | +7.7% |
| GAP | -1.1% | -1.1% | +48.9% | +43.4% |
| GMEAN26 | +1.9% | +1.6% | +19.0% | +20.9% |

### 3.6.7 Non Memory-Intensive Workloads

In our studies, we only considered benchmarks that had an L3 cache MPKI $\geq 2$, as they tend to be sensitive to optimizing memory system. Alternatively, if a workload fits in on-chip caches, then such workload would not benefit from improving off-chip memory.

Figure 3.13 shows performance impact of DICE on the non memory-intensive SPEC benchmarks excluded from detailed study. As many of these benchmarks fit in L3 cache, they do not see benefit. However, more importantly, DICE does not degrade performance for any of them. On average, DICE improves performance by 2% on these workloads.

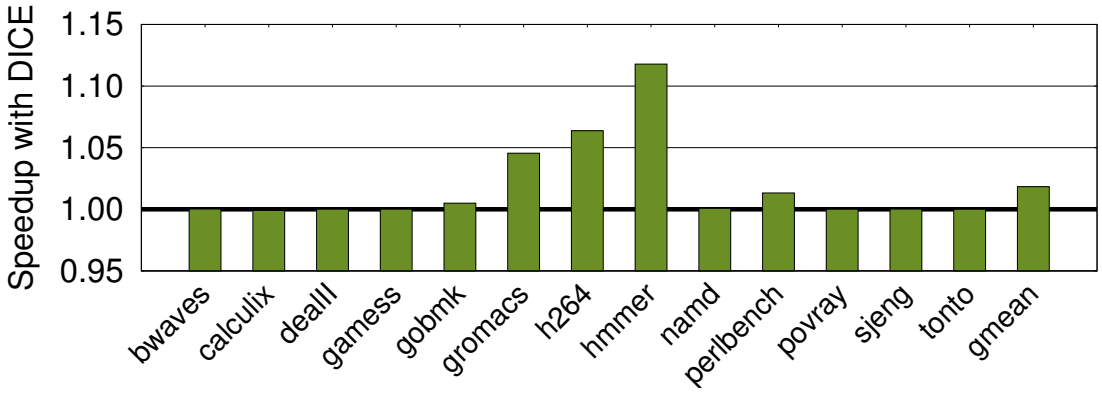



Figure 3.13: Speedup of DICE on non-memory-intensive applications (L3 MPKI $<2$). DICE does not degrade these workloads.

### 3.6.8 Sensitivity to Capacity, BW, and Latency

Table 3.8 shows sensitivity of DICE to varying the capacity, bandwidth, and latency of the DRAM cache, normalized to their respective uncompressed designs. For a 2GB DRAM cache, DICE retains its bandwidth benefits for a speedup of 13.2%. For a 2x-channel DRAM cache, denoted by 2x BW in Table 3.8, DICE performs well at 24.5% speedup. We note that specifications for stacked DRAM state that stacked DRAM latency remains same as DIMM-based counterparts. Nonetheless, we perform sensitivity study on a half-latency DRAM cache. For a half-latency DRAM cache, DICE is able to alleviate the increased memory pressure (by increasing L4 hit rate) caused by the lower-latency DRAM cache for a speedup of 24.4%. Overall, DICE is robust and benefits a wide range of DRAM configurations.

Table 3.8: Sensitivity of DICE on different caches

| | Base(1GB) | 2x Capacity | 2x BW | 50% Latency |
|---|---|---|---|---|
| SPEC RATE | +12.2% | +8.7% | +13.3% | +13.5% |
| SPEC MIX | +7.5% | +4.7% | +8.2% | +9.1% |
| GAP | +48.9% | +32.6% | +75.9% | +73.5% |
| GMEAN26 | +19.0% | +13.2% | +24.5% | +24.4% |

### 3.6.9 Impact of DICE on Energy

Figure 3.14 shows L4+Memory power, energy consumption, and energy-delay-product (EDP) of a system with TSI, BAI, and DICE, normalized to the baseline. TSI increases L4 hit rate, which reduces memory energy consumption. BAI improves performance and energy for compressible workloads, but hurts incompressible ones, making its performance similar to baseline but energy worse. DICE improves both L3 and L4 hit rate leading to a reduction in both stacked DRAM and memory energy consumption. Overall, DICE reduces energy consumption by 24% and EDP by 36%.

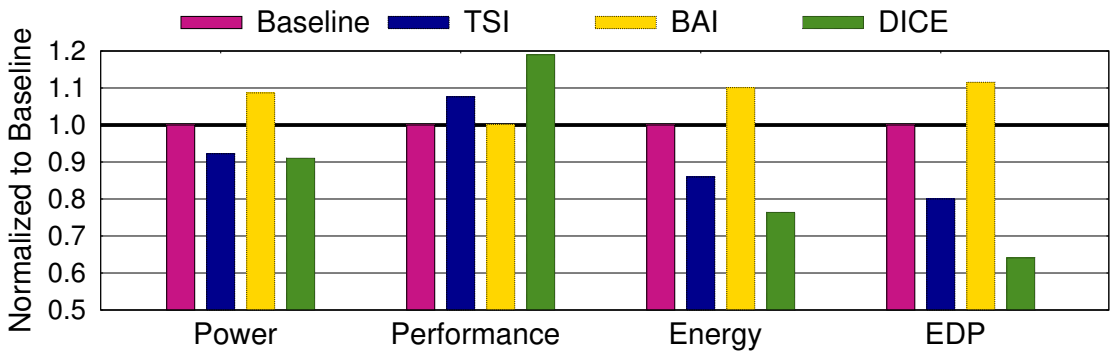



Figure 3.14: Impact of DICE on energy. DICE reduces DRAM cache and memory accesses, reducing off-chip energy by 24%.

### 3.6.10 Comparison to Compressed SRAM Cache Designs (Superblock)

Prior work has looked at using compression to increase capacity of on-chip SRAM caches. Cache compression is typically done by accommodating additional ways in a given cache set and statically allocating more tag-store entries [41, 90]. These proposals optimize purely for hit rate, while we find that DRAM caches are more sensitive to bandwidth. As such, 11.4% of our 19.0% speedup is from bandwidth benefits, not capacity (DICE over TSI). Recent proposals, such as Skewed Compressed Cache (SCC), investigate reducing SRAM tag overhead by sharing tags across spatially-contiguous sets in what are called *superblocks* [43, 42, 44, 91]. For a 4x-superblock 8-way physical cache, these proposals use 32 physical tags to address up to 128 compressed lines by sharing the tags of neighboring sets. Unfortunately, an access in SCC requires skewed associative lookup for different locations in the cache. While this may be practical to do in an SRAM cache, it incurs prohibitively high bandwidth overhead to lookup multiple locations in DRAM cache to service each request.

We evaluate SCC in the context of DRAM caches, to highlight need for bandwidth efficiency in compressing DRAM caches. Figure 3.15 shows the speedup from compressing the DRAM cache with SCC and DICE. Each request in SCC incurs four accesses to DRAM

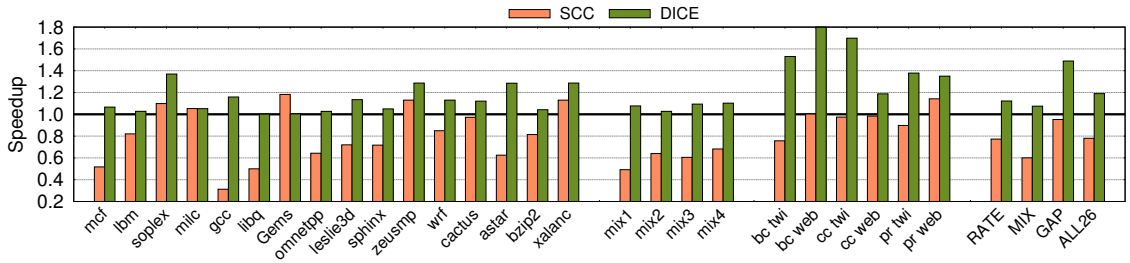



Figure 3.15: Skewed Compressed Cache (SCC) on DRAM cache. SCC causes 22% slowdown due to extra tag accesses.

cache (3 for tags and one for data), whereas a request in DICE requires only one access to the DRAM cache in the common case (second only in case of CIP misprediction). On average, SCC causes 22% slowdown, whereas DICE provides 19% speedup.

## 3.7 Summary

This work looked at compression as a means of increasing both the capacity and bandwidth of DRAM caches. We exploit the fact that practical DRAM caches are likely to store tags within the DRAM array, so they can support compression seamlessly, as the tag-store entries required for the additional capacity created due to compression can be accommodated within the DRAM substrate without the need for any SRAM overheads. Furthermore, as DRAM caches are managed entirely in hardware, we can do DRAM cache compression in a software-invisible manner and avoid the OS changes that are necessary for compressing main memory.

Our study showed that for maximizing performance it is important that DRAM caches perform compression for enhancing both the capacity and the bandwidth. We note that traditional methods to perform cache compression are aimed at solely increasing the cache capacity and provide only marginal benefits. To this end, our work proposes to change the cache indexing dynamically to a bandwidth-enhancing scheme called *Bandwidth-Aware Indexing (BAI)*. We show that while BAI can improve both capacity and bandwidth when data is compressible, it can degrade performance when data is not compressible. We propose

*Dynamic-Indexing Cache Compression (DICE)* that dynamically changes cache indexing depending on compressibility of line. To avoid looking up two locations for a line, we develop low-cost *Cache Index Predictors (CIP)* that can accurately predict index for the line using history information. Our evaluations show that DICE improves DRAM cache capacity and bandwidth to improve performance of a 1GB DRAM cache by 19.0% and reduce EDP by 36%, all while incurring storage overhead of less than 1 kilobyte and without requiring OS support.

These capacity and bandwidth benefits DICE offers are contingent on compressibility of data – it may be desirable to enable associativity benefits regardless of compressibility. We investigate an alternative solution based on way-prediction in the next chapter.

# CHAPTER 4
# ENABLING ASSOCIATIVITY WITH ACCORD

This chapter discusses how to enable associativity for baseline direct-mapped Tag-Inside-Cacheline (TIC) DRAM caches at minimal bandwidth cost [19].

## 4.1 Introduction

DRAM caches are important for enabling effective heterogeneous memory systems that can transparently provide the bandwidth of high-bandwidth memories [3], and the capacity of high-capacity memories [4, 8]. Current DRAM cache design [1, 6] stores tag next to data at a line granularity and utilize a direct-mapped organization (i.e., baseline direct-mapped TIC design), such that a single access can provide both data and the associated tag, which allows for quick determination of a hit or a miss. Unlike its set-associative counterpart [5, 37, 25, 26], the direct-mapped organization is attractive as it eliminates additional accesses to determine location of lines and optimizes for hit latency. However, such caches can suffer from a low cache hit rate due to significant conflict misses.Ideally, we would like the hit-latency of direct-mapped caches, but the hit-rate of set-associative caches. In this work, we focus on enabling associativity in a bandwidth-efficient and low hit-latency manner suitable for such DRAM caches that place Tag-Inside-Cacheline.

### 4.1.1 Challenges in Set-Associativity

Trading off the cache hit rate for low hit latency may be acceptable when the memory latency is similar to DRAM cache latency. However, this trade-off is unacceptable when the main memory latency is much longer than the DRAM cache latency (e.g., 2-4X longer in the case of NVM). Therefore, it is important to optimize for the DRAM cache hit rate while retaining low DRAM cache hit latency. A straightforward way of improving the cache hit

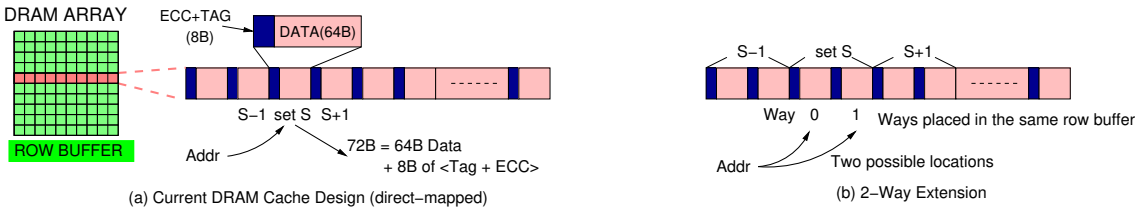



Figure 4.1: (a) Organization of practical direct-mapped TIC DRAM cache [1, 6]: Each access indexes a direct-mapped location and transfers 72 bytes that has tag and data (and ECC). (b) Extending to a 2-way cache. Ways in the same set are placed in the same row buffer. Each memory address has two possible locations (ways).

rate is to build a set-associative DRAM cache. For example, we extend the DRAM cache design to a 2-way cache, shown in Figure 4.1(b). The tag of a way is associated with the way, and ways in the same set are placed in one row buffer [5]; in this case, each memory address has two possible locations. However, we identify the following obstacles in designing set-associative DRAM caches:

1. **Determining Hit Location:** On an access, the line can be present in any way of a set in a set-associative DRAM cache. We may need to check all the ways to obtain the requested line, thus incurring the bandwidth and latency overheads associated with accessing multiple lines. Ideally, we want to implement a set associative DRAM cache while retaining the single lookup of a direct-mapped DRAM cache.

2. **Miss Confirmation:** On miss, we need to ensure that the line is not present in the DRAM cache. This typically requires checking all ways in a set, which incurs bandwidth overhead proportional to DRAM cache set associativity. For set-associative DRAM caches, we want to reduce the bandwidth overheads incurred by miss confirmation.

3. **Writeback Probe:** On a writeback operation to a set-associative DRAM cache, a *write-back probe* may be needed to determine the way in which the line is resident. For a direct-mapped DRAM cache, simply knowing that the line is resident is sufficient to avoid a writeback probe operation (by keeping the DRAM cache presence, or DCP bit, in the L3 cache [17]). However, for a set-associative DRAM cache, we additionally need to know "which way" to write back to. To determine the correct way to write back to on a writeback,

we extend the DCP scheme to store way information as well. We note that any associative DRAM cache will require this extension to enable write performance.

4. **Replacement Policy:** A set-associative cache relies on a replacement policy to choose a victim line. For any intelligent replacement policy, an update of the replacement state is performed on hits and insertions (e.g., counter update [76]). Since DRAM caches store the tags with the line, updating the replacement state incurs a DRAM write operation (to the line). Unfortunately, performance overhead because of extra bandwidth for such updates far outweighs performance benefits from the improved hit rate [6]. In fact, using an update-free replacement policy (e.g., random) provides better performance.[1] As such, we use random replacement for set-associative DRAM caches throughout this work.

### 4.1.2 Options for Implementing Set-Associativity

In an N-way set-associative cache, a line can be resident in any of the N locations. Ideally, we desire a low-latency and low-bandwidth implementation of a set-associative DRAM cache. We now discuss some options for implementing a set-associative DRAM cache.

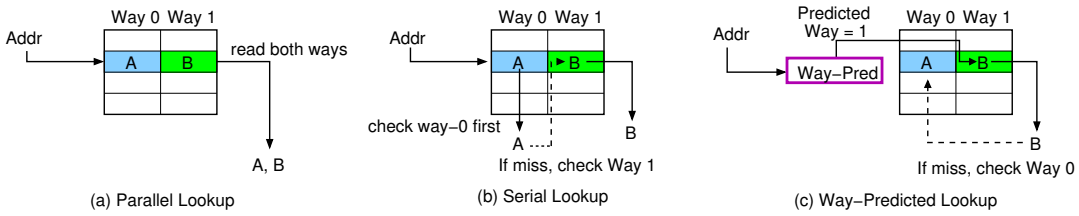



Figure 4.2: Comparison of accessing lines in a 2-way DRAM cache. (a) Parallel Lookup: One cache request reads all ways in the corresponding set. (b) Serial Lookup: A request first reads way-0; if miss, it checks way-1. If data is in way-1, this dependent way checking incurs serialization penalty. (c) Way-Predicted Lookup: A request first reads a predicted way; if miss, it checks the other way.

[1] For example, using LRU replacement for a 2-way DRAM cache degrades performance by 9% compared to random replacement.

*Parallel Lookup*

A straightforward way to locate data in an N-way DRAM cache is to read all N lines of a set on each access, shown in Figure 4.2(a). We refer to such a design as *parallel lookup*. While parallel lookup may be practical for SRAM caches, the bandwidth overhead of streaming N lines on each access (both hits and misses) becomes prohibitive for DRAM caches.

*Serial Lookup*

Alternatively, we can check the lines sequentially (see Figure 4.2(b)). We refer to this design as a *serial lookup* design, which reduces the bandwidth for transferring lines on a hit from N to ~(N+1)/2 on average. However, it introduces latency as the lookups are now serially dependent on each other (N accesses). If we can predict which way the line is likely to be in, we can reduce hit latency further.

*Way-Predicted Lookup*

We can reduce serialization penalty by intelligently choosing which way to check first using *Way-Prediction* (see Figure 4.2(c)). We refer to this design as a *Way-Predicted Lookup* design. On a hit, if we can predict the way accurately, we can service hits with just one DRAM access. This organization can achieve the hit latency of a direct-mapped cache while maintaining the hit rate of an N-way cache. However, achieving high way-prediction accuracy at low storage cost is difficult in practice. Furthermore, the bandwidth overheads of miss confirmation is still N lookups.

### 4.1.3 Conventional Way-Predictors: Challenge in Scaling to Gigascale DRAM Caches

Way Prediction has been proposed for SRAM caches (L1 or LLC) for reducing latency and power. For L1, tracking the most-recently-used way in each set (MRU Pred [54, 55]) provides high accuracy as the access stream of an L1 cache has high temporal locality. Unfortunately, such locality is typically not visible to secondary caches (L2/L3/L4), so

MRU prediction tends to be not as effective. Secondary caches can use a partial-tag (e.g., 4-bits) design for each line to avoid cache lookup [56, 57, 58, 59, 60, 61]. Unfortunately, both MRU Pred and Partial-Tag design require per-set or per-line storage. Such schemes would incur storage overhead of several MBs for a 4GB DRAM cache, in Table 4.1.

Table 4.1: Accuracy and Storage of way predictors for a 4GB cache

| | Rand Pred | MRU Pred | Partial-Tag |
|---|---|---|---|
| Storage | 0B overhead | 4MB overhead | 32MB overhead |
| 2-way | 50.0% | 85.7% | 97.3% |
| 4-way | 25.0% | 74.3% | 91.6% |
| 8-way | 12.5% | 63.2% | 81.2% |

Table 4.1 shows that simply predicting a random location has low accuracy (labeled as *Rand Pred*). Additionally, MRU prediction accuracy also degrades significantly at higher associativity. The partial-tag design has good accuracy but incurs an impractical storage overhead of 32MB. If we are to implement way prediction effectively at the size of DRAM caches, we need a way-predictor that is both accurate and requires lower storage overhead than conventional designs.

### 4.1.4 Insight: Way-Steering for Way-Prediction

In a conventional set-associative design, any way in a set can be replaced (determined by the replacement policy), thus complicating the way prediction. We observe that if we *steer* the incoming line to a "preferred way" based on the line address at install time, way prediction can simply use the preferred way as the default prediction at access time. The coordination between way install and way prediction can simultaneously improve way prediction accuracy and reduce way prediction hardware requirements. Based on this insight, we develop a scalable (low storage overhead), low-latency, and low-bandwidth way prediction for set-associative DRAM caches.

## 4.2 Design of Associativity via Coordinated Way-Install and Way-Prediction

Way-prediction can enable a set-associative cache to maintain the hit latency of a direct-mapped cache but the hit-rate of an associative cache. Conventional way predictors are designed independent of the way install policy of the cache and typically incur significant storage overhead. There tends to be no correlation between the line address and the install way. If we can instead coordinate way-install with way-prediction, we can obtain high accuracy for way prediction while incurring negligible storage overhead. To this end, we propose *Associativity via Coordinated Way-Install and Way-Prediction (ACCORD)*, a design that *steers* an incoming line to a "preferred way" based on line address and uses that preferred way as the default way prediction, in Figure 4.3. We propose and evaluate three low-cost and effective *way-steering* policies to enable associativity for DRAM caches at low bandwidth and SRAM storage cost.

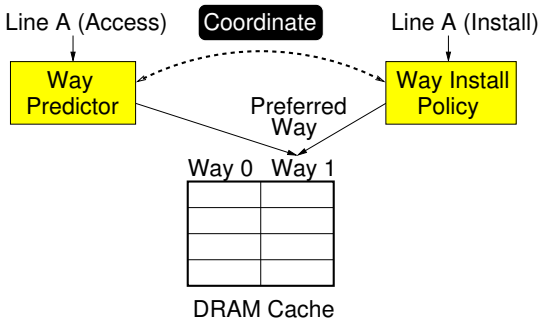



Figure 4.3: ACCORD: Coordinating Way-Install and Way-Prediction using Way Steering.

### 4.2.1 Probabilistic Way-Steering (PWS)

The install policy in practical set-associative DRAM cache is likely to employ random replacement, to avoid bandwidth for updating replacement state on a hit. Random replacement chooses any way in a set with equal probability. For example, in a 2-way cache, either way will be picked with 50% probability. However, using the line address, we can bias the install policy to select a particular way with a higher probability. Based on this insight, we propose our first way-steering policy, *Probabilistic Way Steering (PWS)*.

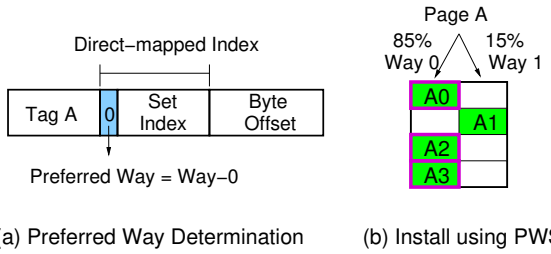



Figure 4.4: Probabilistic Way-Steering (PWS): (a) deciding preferred way based on tag, and (b) installing in the preferred way based on preferred-way install probability(e.g., 85%).

PWS determines the preferred way of a line based on the tag of the line, as shown in Figure 4.4(a). If the tag is even, Way-0 is preferred; if the tag is odd, Way-1 is preferred. On an install, PWS chooses the preferred way with a given probability, *Preferred-Way Install Probability (PIP)*. For example, if PIP=85%, then, PWS chooses the preferred way with 85% likelihood, as shown in Figure 4.4(b).[2] On an access, the way-prediction statically predicts the preferred way based on the tag. As we install in the preferred way 85% of the time, we will find the line in the preferred way ~85% of the time. As such, the prediction accuracy of PWS is approximately equal to PIP. Note that a high value of PIP can degrade cache hit-rate as it reduces the flexibility of an associative cache. Thus, tuning PIP effectively allows PWS to trade off a small amount of hit-rate (i.e., speed of learning to use both ways) for predictability.

Table 4.2: Average Hit-Rate and Speedup of PWS

| Organization | Hit-Rate | WP Acc. | Speedup |
|---|---|---|---|
| 2-way (Unbiased, PIP=50%) | 77.5% | 50.0% | 2.6% |
| 2-way PWS (PIP=60%) | 77.5% | 59.8% | 3.7% |
| 2-way PWS (PIP=70%) | 77.5% | 69.4% | 4.7% |
| 2-way PWS (PIP=80%) | 77.3% | 78.6% | 5.5% |
| 2-way PWS (PIP=85%) | 77.2% | 83.1% | 5.6% |
| 2-way PWS (PIP=90%) | 76.9% | 87.8% | 5.3% |
| Direct-Mapped (PIP=100%) | 74.2% | 100.0% | 0.0% |

PWS must balance between the dueling needs of high hit-rate and high way-prediction accuracy. Table 4.2 shows the hit-rate and performance of PWS as a function of PIP,

[2]For a two-way cache, PWS with PIP=50% is an unbiased policy that is identical to the baseline random replacement policy, whereas PIP=100% degenerates into a direct-mapped cache.

averaged over all our workloads. At PIP of 80% or below, PWS provides most of the hit-rate of a 2-way cache, and achieves high way-prediction accuracy (similar to PIP). Note that even at PIP=90%, PWS still provides most of the hit-rate benefit of a 2-way design, as lines that constantly thrash eventually use the other way. However, at PIP=100%, the design simply degenerates into a direct-mapped cache. We observe that, PIP between 80% and 85% provides the best trade off between hit rate and way-predictability. Performance of PWS is maximized (5.6% speedup) at PIP=85%. Thus, we use PIP=85% for the remainder of this work.

### 4.2.2 Ganged Way-Steering (GWS)

PWS guides most of the lines to the preferred way; however, some lines in the region can still go to different ways. This is because PWS makes the install decisions on each miss, and decides if the incoming line should be installed in the preferred way or the non-preferred way. In conventional cache management schemes, the install decisions of one set has no bearing on the install decisions on another set. We develop an insight that if we could coordinate the install decisions across sets, then we can obtain even higher way-prediction accuracy. For example, if multiple lines of a spatially contiguous region miss the cache, then rather than making independent install decisions for each line in the region, we could make the install decision for the first line in the region and install subsequent lines from the region in the same way as the earlier line from that region. Based on this insight, we propose *Ganged Way-Steering (GWS)*.

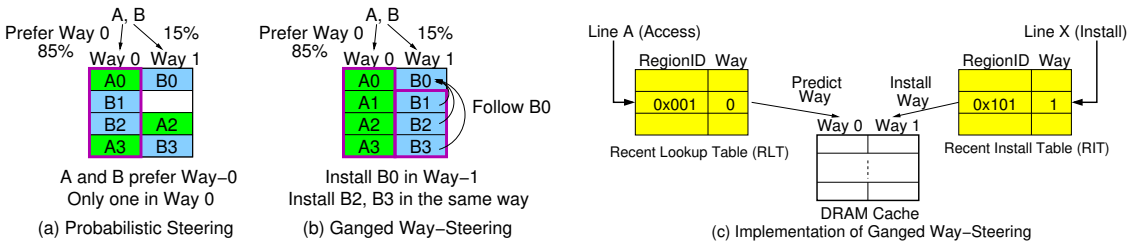



Figure 4.5: The benefit of coordinating install decisions across sets: (a) PWS install lines of each region (A and B) into either way. (b) GWS steers later lines from the region into the same way earlier line was installed. The purple boxes denote lines that can be predicted.

Figure 4.5(c) shows that GWS steers installs to last-way-installed (region-based) with Recent Install Table (RIT), and predicts way based on last-way-seen (region-based) with Recent Lookup Table (RLT). Ganged-Install and Ganged-Prediction provides high way-prediction accuracy for workloads with high spatial locality within a region.[3]

### 4.2.3 Extending to Higher Associativity with Skewed Way-Steering (SWS)

Thus far, we have analyzed ACCORD for 2-way caches. The next step is to scale to N-ways. However, there is a major obstacle in making DRAM cache highly set-associative: the prohibitive cost of miss confirmation. For an N-way cache, a cache request needs to do N lookups to confirm a miss. Consider a scenario where an 4-way cache receives 100 requests, and 60 of them hit. Even with perfect way prediction, the 40 misses still incur 160 lookups (40*4) for confirmation, shown in Figure 4.6(a). We found that extending ACCORD to four ways achieves only 3% speedup and extending it to eight ways causes 6% slowdown. Therefore, to enable high set associativity, we must address the cost of miss confirmation.

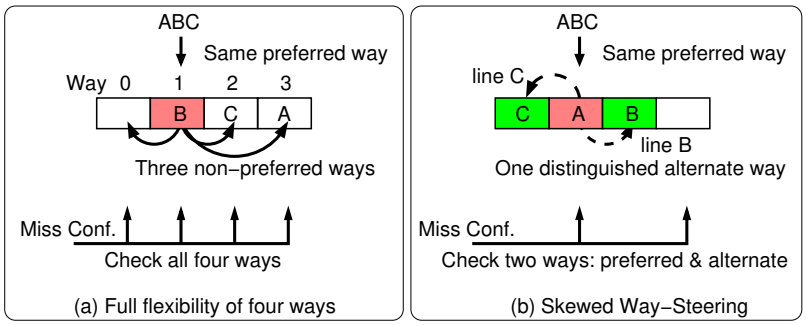



Figure 4.6: (a) Full flexibility of four ways. Miss confirmation checks all locations. (b) Skewed Way-Steering. Each line in the same set has one distinct alternate location. Miss confirmation checks only two locations.

If each line in a set can be restricted to only two possible locations, the miss confirmation needs to check only two ways, significantly reducing the bandwidth overhead of miss

[3]The region size is 4KB, and we use a 64-entry RIT and 64-entry RLT (total 160B storage overhead) as tracking 64 regions captures most of the benefit of GWS.

confirmation. Figure 4.6(b) shows an example. Lines $A$, $B$, and $C$ map to the same set and thrash in their preferred ways (Way-1); besides the preferred way, each line now can be in one and only one other location determined by the hash of tags: $A$ in 1 or 3, $B$ in 1 or 2, and $C$ in 1 or 0. On misses, $A$, $B$, and $C$ might first try to install in their preferred way. But on subsequent misses, $B$ could be installed in its alternate way Way-2, and $C$ in Way-0. Therefore, with this restriction, most of lines can still fit in the cache [64], but miss confirmation cost is reduced to two lookups. Based on this insight, we propose *Skewed Way-Steering (SWS)*, which restricts lines to having exactly one *Alternate* location, instead of allowing the line to have (N-1) non-preferred locations in a N-way set-associative cache. The Alternate way is selected using an intelligent hashing function such that the Alternate location is guaranteed to be different from the Preferred location. Overall, ACCORD with SWS provides improved hit-rate at direct-mapped bandwidth cost.

## 4.3 Methodology

### 4.3.1 Framework and Configuration

We use USIMM [86], an x86 simulator with detailed memory system model. We extend USIMM to include a DRAM cache. Table 4.3 shows the configuration used in our study. We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache). All caches use 64B line size. We model a virtual memory system to perform virtual to physical address translations. The baseline contains a 4GB direct-mapped DRAM cache that places tags with data in the unused ECC bits [87]. The parameters of our DRAM cache is based on HBM technology [3]. The main memory is based on non-volatile memory and assumed to have a latency similar to PCM [11, 12, 13, 14]: the read latency is 2-4X and the write latency is 4X as long as those of DRAM [4].

Table 4.3: System Configuration

| | |
|---|---|
| Processors | 16 cores; 3.0GHz, 2-wide OoO |
| Last-Level Cache | 8MB, 16-way |
| **DRAM Cache** | |
| Capacity | 4GB |
| Bus Frequency | 500MHz (DDR 1GHz) |
| Configuration | 8 channel, 128-bit bus |
| Aggregate Bandwidth | 128 GB/s |
| tCAS-tRCD-tRP-tRAS | 13-13-13-30 ns |
| **Main Memory (PCM)** | |
| Capacity | 128GB |
| Bus Frequency | 1000MHz (DDR 2GHz) |
| Configuration | 2 channel, 64-bit bus |
| Aggregate Bandwidth | 32 GB/s |
| tCAS-tRCD-tRP-tRAS-tWR | 13-128-8-143-160 ns |

### 4.3.2 Workloads

We run benchmark suites of SPEC 2006 [92], GAP [89], and HPC. For SPEC, we perform studies on all 9 benchmarks that have at least 5% speedup potential going from 1-way to 8-way, along with 2 workloads that are less sensitive to set associativity. GAP is graph analytics with real data sets (`twitter`, `web sk-2005`) [93]. The evaluations execute benchmarks in rate mode, where all cores execute the same benchmark. In addition to rate-mode workloads, we evaluate 10 mixed workloads, which are created by choosing 16 of the 16 SPEC workloads that have at least two miss per thousand instructions (MPKI). Table 4.4 shows L3 miss rates, memory footprints, and performance potential with ideal 8-way cache for the rate-mode workloads used in our study.

We perform timing simulation until each benchmark in a workload executes at least 2 billion instructions. We use weighted speedup to measure aggregate performance of the workload normalized to the baseline and report geometric mean for the average speedup across all the 21 workloads. For other workloads that are neither memory bound nor sensitive to set associativity of DRAM caches, we present performance of all 46 workloads evaluated (29 SPEC, 10 SPEC-mix, 6 GAP, and 1 HPC) in Section 4.4.3.

Table 4.4: Workload Characteristics

| Suite | Workload | L3 MPKI | Footprint | 8-Way Potential Speedup |
|---|---|---|---|---|
| SPEC | soplex | 29.8 | 3.6 GB | 2.43 |
| | leslie | 17.0 | 1.2 GB | 1.63 |
| | libq | 25.6 | 512 MB | 1.55 |
| | gcc | 5.8 | 2.9 GB | 1.27 |
| | zeusmp | 5.3 | 3.3 GB | 1.18 |
| | wrf | 7.4 | 2.2 GB | 1.18 |
| | omnet | 20.8 | 2.4 GB | 1.17 |
| | xalanc | 4.3 | 3.0 GB | 1.09 |
| | mcf | 83.1 | 26.1 GB | 1.06 |
| | sphinx | 13.7 | 293 MB | 1.01 |
| | milc | 25.5 | 9.0 GB | 0.99 |
| GAP | pr twitter | 121.7 | 30.5 GB | 1.15 |
| | cc twitter | 108.7 | 18.6 GB | 1.15 |
| | bc twitter | 81.1 | 26.9 GB | 1.11 |
| | pr web | 17.8 | 30.3 GB | 1.07 |
| | cc web | 9.2 | 18.6 GB | 1.05 |
| HPC | nekbone | 6.4 | 111 MB | 1.04 |

## 4.4 Results and Analysis

### 4.4.1 Effectiveness of ACCORD

The effectiveness of ACCORD is contingent on high way-prediction accuracy for low bandwidth overhead, and increase in hit-rate to enable more data serviced from high-bandwidth memory. We analyze ACCORD's impact on way-prediction accuracy and hit-rate.

*Impact on Way-Prediction Accuracy*

Figure 4.7 shows the way-prediction accuracy of PWS, GWS, and PWS+GWS. PWS has an accuracy close to 85% (since PIP=85%). GWS has near-ideal accuracy for workloads with high spatial locality (e.g., *nekbone* and *libq* have 99% accuracy). Note that GWS can only way-predict on RLT hits and defaults to random prediction on RLT miss. As such, GWS has limited accuracy when pages are sparsely accessed or the workload footprint is large (e.g., *mcf* and *pr twi*). GWS, combined with the way-prediction policy of PWS, improves way-prediction accuracy to 90%. This enables the DRAM cache to frequently locate the lines in one access.

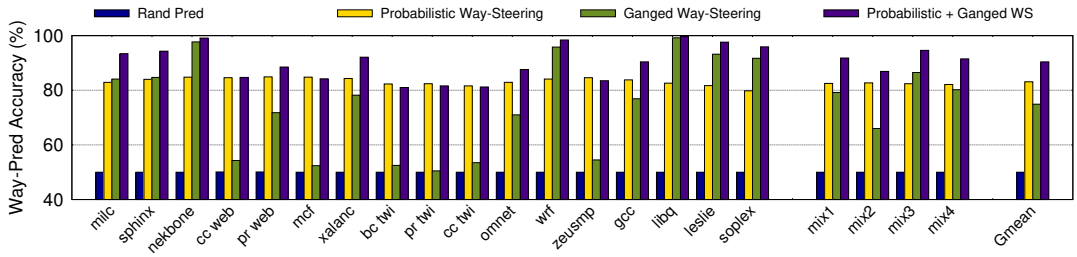



Figure 4.7: Accuracy of way-predictors for a 2-way cache. While probabilistic way-steering (PWS) provides a 83% accuracy, ganged way-steering (GWS) is 75% accurate. Combining GWS with PWS provides 90% accuracy.

*Impact on Cache Hit-Rate*

Table 4.5 shows the impact of Way-Steering on hit rate. Increasing associativity to 2 ways increases hit-rate from 74.2% to 77.5%. GWS retains the hit-rate of a 2-way cache, as it simply increases the granularity at which replacement happens. PWS, on the other hand, trades hit-rate for predictability, so there is small hit-rate degradation under PWS+GWS, to 77.3%. Overall, PWS+GWS achieves high way-prediction accuracy (>90%), while keeping most of the hit-rate of a 2-way cache (77.3%).

Table 4.5: Sensitivity of hit-rate to PWS and GWS

| | Direct-mapped | 2-Way Rand | PWS | GWS | PWS+GWS |
|---|---|---|---|---|---|
| Amean | 74.2% | 77.5% | 77.2% | 77.7% | 77.3% |

To further improve hit-rate without increasing miss confirmation cost, we employ Skewed Way-Steering. SWS provides flexibility of where to place lines, while simultaneously removing worst-case tag-lookup bandwidth. We show aggregate hit-rate for SWS(4,2) and SWS(8,2) and compare it with the hit-rate of direct-mapped, 2-way ACCORD (PWS+GWS), and 8-way cache. In Table 4.6, we can see that SWS(8,2) offers improved hit-rate over 2-way ACCORD, at similar miss-confirmation cost. Note that going from 2-way to 8-way provides little hit-rate improvement, and, SWS provides 1/3rd of the benefit for nearly free.

Table 4.6: Hit-Rate of different ACCORD designs

| | Direct-mapped | ACCORD (2-way) | SWS (4,2-way) | SWS (8,2-way) | 8-Way |
|---|---|---|---|---|---|
| Hit Rate | 74.2% | 77.3% | 77.7% | 77.9% | 79.7% |

### 4.4.2 Performance of ACCORD

ACCORD provides speedup due to enabling associativity at low bandwidth cost. Figure 4.8 shows the speedup for parallel tag-lookup, serial tag-lookup, PWS, GWS, PWS+GWS, and perfect way-prediction. Parallel Lookup wastes bandwidth, and Serial lookup incurs latency. Probabilistic Way-Steering provides a good baseline prediction accuracy for the cache. Ganged Way-Steering improves way-predictability for workloads with high spatial locality by steering ganged patterns to install in the same way. Combining Probabilistic Way-Steering and Ganged Way-Steering achieves 90% way-prediction accuracy (with <1KB SRAM) and provides 7.3% speedup, close to (10%) with perfect. Additional modifications for skewed-associativity bring final speedup to 11%.

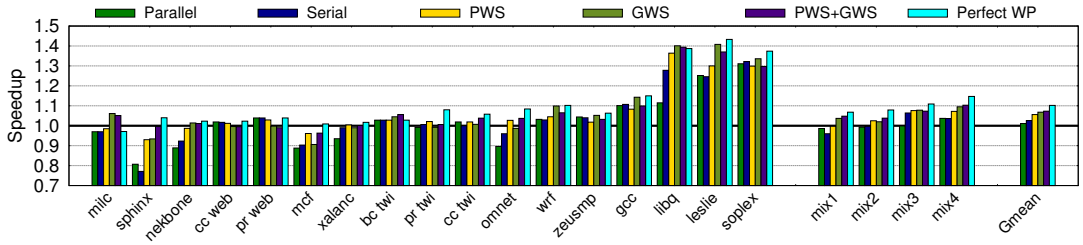



Figure 4.8: Speedup from 2-way DRAM Cache.

### 4.4.3 Speedup for Remaining Workloads and Mixes

Figure 4.9 shows speedup of ACCORD with 2-way and SWS(8,2) across all 46 workloads evaluated, including 29 SPEC, 10 SPEC-mix, 6 GAP, and 1 HPC workloads. On average, ACCORD improves performance by 4% and 6% for the 2-way and SWS(8,2) configurations. And, ACCORD improves the performance of the 10 Mix workloads by 7% and 11% on average. More importantly, Figure 4.9 shows that ACCORD maintains performance across all workloads, including ones that are not sensitive to increased associativity.

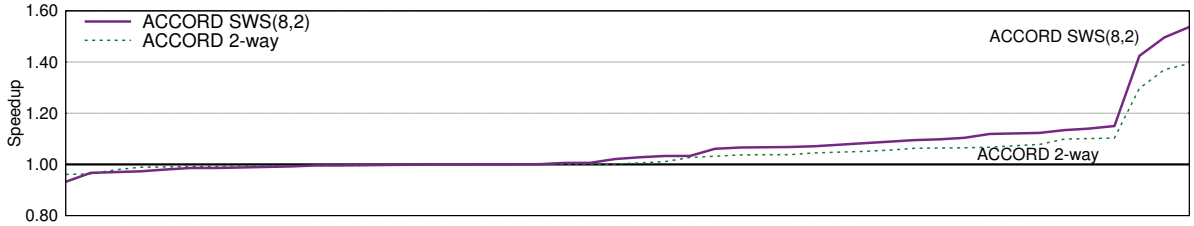



Figure 4.9: Speedup of ACCORD over 46 workloads (including ones not sensitive to memory or hit rate).

### 4.4.4 Impact of Cache Size

We use a default cache size of 4GB for our studies. Table 4.7 shows the speedup of ACCORD as the size of the DRAM cache is varied from 1GB to 8GB. ACCORD provides significant speedup across different cache sizes, ranging from 13.6% at 1GB to 8.6% at 8GB. As expected, when the cache size is increased, larger portions of the workload fit in, and there is reduced scope for improvement.

Table 4.7: Sensitivity to Cache Size

| Cache Size | Avg. Speedup from ACCORD |
|---|---|
| 1.0GB | 13.6% |
| 2.0GB | 12.0% |
| **4.0GB** | **10.6%** |
| 8.0GB | 8.6% |

### 4.4.5 Storage Requirements

We analyze the storage overheads of ACCORD in Table 4.8. Both PWS and SWS do not require any storage overheads. The only storage required is for GWS. Each entry in the Recent Lookup Table entry (RLT) and Recent Install Table (RIT) is 20 bits (1 valid bit + 19-bit tag). With a 64-entry RLT and 64-entry RIT, the total storage overheads would be 320 bytes. Thus, ACCORD enables associativity while incurring a total storage overhead of 320 bytes.

Table 4.8: Storage Requirements of ACCORD

| ACCORD Component | Storage |
|---|---|
| Probabilistic Way-Steering | 0 Bytes |
| Ganged Way-Steering | 320 Bytes |
| Skewed Way-Steering | 0 Bytes |
| ACCORD | 320 Bytes |

## 4.5 Summary

This work addresses the challenge of enabling set associativity for line-granularity DRAM caches at low bandwidth and low storage cost. We propose ACCORD (*Associativity via Coordinated Way Install and Way Prediction*), a framework that enables low-cost way prediction by steering lines to "preferred" ways on install time and predicting the preferred way on access time. We propose two policies: *probabilistic way-steering*, which steers cache line installs to a preferred way based on its tag and predicts the preferred way on an access, and *ganged way-steering*, which steers subsequent installs in a region to the same way and predicts accesses in that region with last observed way. These policies provide a 90% way-prediction accuracy at negligible storage overhead (320 bytes). To obtain higher levels of set associativity (e.g., 8-way) at reduced overheads of miss confirmation, we propose *skewed way-steering* policy that steers lines to at most two locations in an N-way set-associative cache. Our evaluations on a 4GB DRAM cache show that ACCORD outperforms direct-mapped organization by 11% on average. We develop ACCORD over the existing DRAM cache design used in industry, and we believe the simplicity of our solution will make it appealing for industrial adoption.

# CHAPTER 5

# ENABLING INTELLIGENT REPLACEMENT WITH ETR

This chapter discusses how to enable intelligent replacement policies for direct-mapped TIC DRAM caches at minimal bandwidth cost.

## 5.1 Introduction

DRAM caches are important for enabling effective heterogeneous memory systems that can transparently provide the bandwidth of high-bandwidth memories [3], and the capacity of high-capacity memories [4, 8]. For example, Intel's Knights Landing product organizes its DRAM cache as a direct-mapped cache with tags stored alongside each data-line, so that one access can retrieve both tag and data (i.e., our baseline direct-mapped TIC DRAM cache). This direct-mapped design has been shown to be effective for enabling low-latency and bandwidth-efficient tag access [6]; however, such a direct-mapped design can have significant conflict misses. Fortunately, cache bypassing [17, 80, 79] offers a way to both improve hit-rate and decrease bandwidth consumption, while still maintaining a direct-mapped organization. We investigate the extent to which an intelligent bypass policy can reduce conflict misses for direct-mapped TIC DRAM caches.

We would like to use the most effective replacement policies to improve DRAM cache hit-rate. However, intelligent replacement policies [76, 79, 81, 82] often require the cache to track per-line state that needs to be updated on cache events. On a DRAM cache, managing this per-line state is difficult as tracking even 2-bits of state per line would require multi-megabyte storage. As such, DRAM cache designs would need to keep this state in the DRAM array, and spend offchip bandwidth to update state. Prior replacement policies proposed for DRAM caches have avoided this per-line state with stateless policies [6, 1, 17]. KNL-Cache [6, 1], for example, employs an *Always-Install* policy. Along the same

lines, Chou et. al [17] propose a policy that bypasses the cache with 90% probability (we call this policy *90%-Bypass*). However, such stateless policies often fail to capture the reuse patterns commonly seen in large caches. We show how such policies are often inadequate with an example.


(a) Replacement Policies (b) Speedup Potential

Figure 5.1: (a) Always-Install, 90%-Bypass, and Desired replacement policies under mixed high-reuse low-reuse access pattern. (b) Potential for speedup: Probabilistic Bypass [17], and Ideal Reuse-based Bypass with no state-update cost.

Let us consider replacement policies for a common access pattern where the workload has repeated accesses to high-reuse data (labeled *A*) mixed with accesses to low-reuse data (labeled *B*), as shown in Figure 5.1(a). For the baseline *Always-Install* policy, accesses to A will install A and enable subsequent accesses to A to hit; however, accesses to low-reuse B will evict A and cause the subsequent access to A to miss. In this case, always-installing lines allows low-reuse B to evict high-reuse A, and this results in degraded hit-rate and wasted install bandwidth. For a *90%-Bypass* policy, references to A will install some A lines and marginally improve hit-rate, and references to B will install only a few lines and marginally degrade hit-rate. In this case, 90%-Bypass offers some working-set protection; however, it is indiscriminate in deciding which lines to protect and may not achieve high hit-rate. Figure 5.1(b) shows that such a probabilistic bypass policy has poor speedup potential of 3%. Ideally, we desire a bypass policy that can remember and protect individual lines that have high reuse (i.e., A), and bypass other lines (i.e., B). Figure 5.1(b) shows that if we formulate a *reuse-based bypass* policy while avoiding the bandwidth cost for state-update, we could achieve up to 20% speedup.

Our approach to improving DRAM cache performance is to (1) design a *reuse-based bypass* policy to improve DRAM-cache hit-rate, and to (2) reduce the bandwidth cost of state-update to further improve performance.

### 5.1.1 Adapting Replacement Policies for Direct-Mapped Caches

Typically, cache replacement policies are discussed in the context of a set associative cache, as the set contains multiple lines and there is a choice of the line to evict. For a direct-mapped cache, the set contains only one line, so if we want to install, there is exactly one place the line can go, and we do not have a choice in selecting the victim. However, we could choose to bypass the line, so the binary choice for a direct-mapped cache becomes, whether to evict the resident line or to bypass the incoming line. We can improve the hit-rate by making this binary decision intelligently. We explain different replacement strategies for a direct-mapped cache.

**Probabilistic Replacement:** The simplest policy is to bypass the incoming line with a certain probability. For example, Bandwidth-Aware Bypass (BAB) [17, 79] bypasses the incoming line with 90% probability to reduce install bandwidth, as long as hit-rate remains unaffected. Figure 5.5 shows that such global bypassing policies are coarse-grain and miss out on bypassing opportunities that exploit per-line information.

**Recency-Based Replacement:** LRU[66] installs incoming lines with the highest priority, based on the heuristic that recently-used lines are more likely to be re-used. On a direct-mapped cache, LRU degenerates into an *Always-Install* design, as the incoming line is the most recent. Enhancements of LRU, such as DIP [69], degenerate to probabilistic bypass.

**Reuse-Based Replacement:** Replacement policies that exploit reuse (also called re-reference or frequency) are resilient to thrashing and scans[76, 71, 72]. Such policies can protect the direct-mapped DRAM cache from thrashing when multiple pages are mapped to the same set of the DRAM cache. We discuss *Re-Reference Interval Prediction (RRIP)* [76] policy.

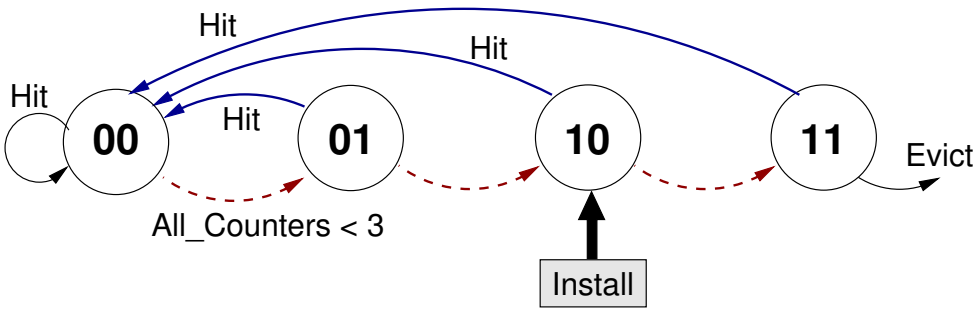



Figure 5.2: Re-Reference Interval Prediction (RRIP).

Re-Reference Interval Predictor[76] is a thrash-and-scan-resistant replacement policy often used in last-level caches. As shown in Figure 5.2, each line is equipped with a 2-bit counter to track the *Re-Reference Interval Prediction Value (RRPV)*. On a hit to the line, the RRPV is *Promoted* to 0. On a miss, the victim is found by searching from way 0 and finding the first line in the set with RRPV of 3. If no such line is found, the RRPV of all lines in the set is *Demoted* (i.e., incremented) and the search is repeated. Lines are installed in RRPV=2 to protect the lines that were re-used.

**Challenge in Using RRIP for Direct-Mapped Cache:** Just like other replacement policies based on reuse-information, RRIP operates by comparing the counter values of multiple candidates in the set. It becomes ill-defined for a direct-mapped cache, where there is only one counter, which means the resident line will always get evicted regardless of the past behavior. Thus, for a direct-mapped cache RRIP degenerates into always-install (or always-bypass if the incoming line is bypassed unless the RRPV of the resident line equals 3). We propose extensions that make reuse-based policies viable for direct-mapped and two-way caches, and implementations that reduce the cost of tracking the RRPV state for gigascale DRAM caches.

## 5.2 Design of Reuse-based Replacement on DRAM Cache with RRIP Age-On-Bypass

If we want to use RRIP on direct-mapped DRAM caches, we have to solve two issues: how do we formulate RRIP as a bypassing policy suitable for caches with limited associativity, and how can we mitigate the state-update cost of maintaining per-line reuse state in DRAM.

### 5.2.1 RRIP as a Bypassing Policy

We design a version of RRIP for limited-associativity caches, called *RRIP Age-On-Bypass (RRIP-AOB)*. The key insight in RRIP-AOB is to use the episode of cache bypassing to age / update the RRPV information associated with the line. Figure 5.3 shows the overview of our design. RRIP-AOB needs to similarly track lines that have reuse, so RRIP-AOB Promotes state (sets RRPV to 0) on hit. RRIP-AOB can protect these reused lines by bypassing when reuse has been seen (bypass when RRPV is 0, 1, or 2). However, reused lines can now stay stuck in high priority state. We need a different mechanism to age older lines so that new lines can eventually be installed. We choose to implement aging by Demoting (increment RRPV) state when an incoming line is bypassed. This allows lines to naturally age to RRPV of 3, and be evicted in favor of the incoming line. Similar to RRIP, RRIP-AOB needs 2-bits per line to track RRPV. A practical design must address where to store the RRPV bits and address the bandwidth needed to track the per-line RRPV.

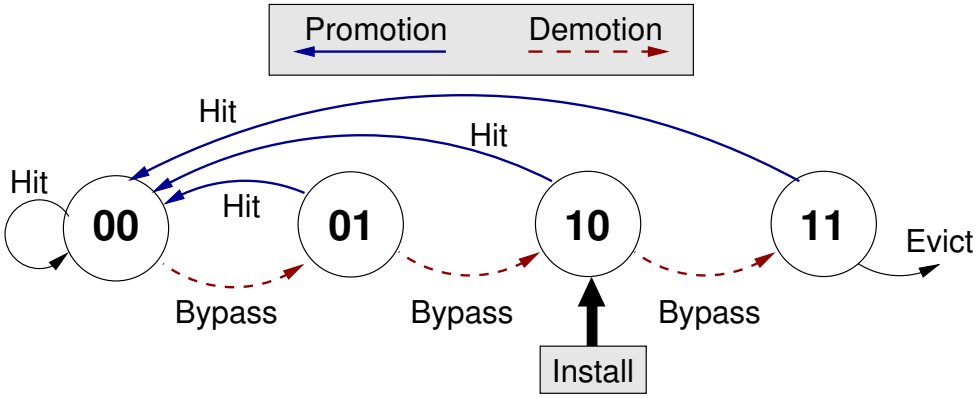



Figure 5.3: Overview of RRIP Age-On-Bypass (RRIP-AOB). The transition from one state to another is accomplished with replacement-state update operation. Such updates may consume significant bandwidth.

### 5.2.2 Storing RRPV in DRAM

A straight-forward way of incorporating RRIP into a DRAM-cache is to extend the tag-entry of the line to incorporate the RRPV bits. We refer to this design as simply *RRIP-AOB*. However, such a design incurs bandwidth overhead for performing update of the replace-

ment state. Note that these accesses for updating the replacement state are not present in the baseline and for designs that do bypassing without tracking per-line state.

Alternatively, we can avoid the bandwidth of replacement updates by storing the replacement state in a dedicated SRAM array. Unfortunately, for our 2GB DRAM cache, maintaining 2-bits of RRPV per line would need 8MB of SRAM, which is impractically large. We call this design *Ideal RRIP-AOB*.

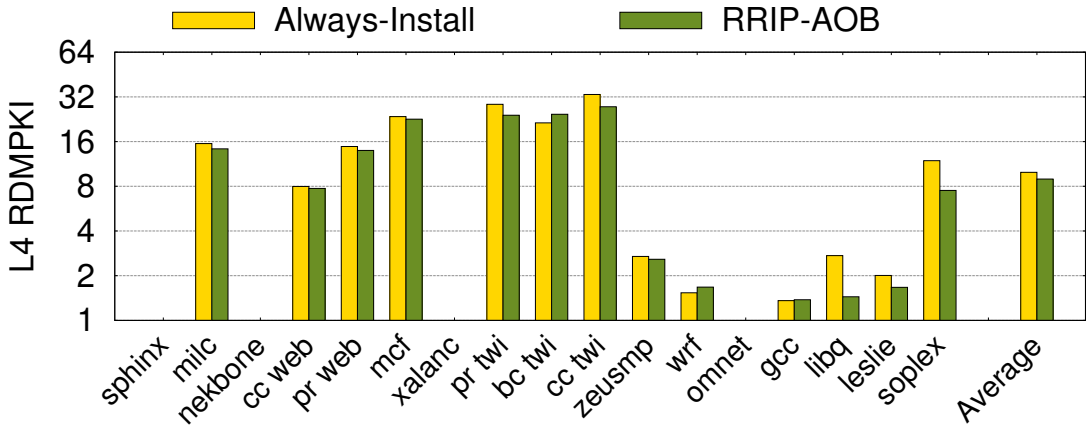



Figure 5.4: MPKI of baseline and RRIP-AOB. RRIP-AOB reduces misses by 10%.

### 5.2.3 Benefits from Reuse-Based Replacement

Intelligent replacement policies improve performance by reducing cache misses. Figure 5.4 shows the Misses Per Thousand Instructions (MPKI) for our baseline DRAM cache and with RRIP-AOB. RRIP-AOB reduces 10% of the misses on average. However, the speedup from RRIP-AOB also depends on bandwidth used in replacement-state updates.

Figure 5.5 shows the speedup from different bypassing policies implemented on our 2GB DRAM cache. Performance numbers are normalized to the always-install policy. Indiscriminately bypassing 90% of the lines (Bypass-90%) causes a degradation of 15%. The adaptiveness of Bandwidth-Aware-Bypass (BAB) [17, 79] avoids slowdowns; however, the average speedup is only 3%. With RRIP-AOB, the performance benefits is 13%, whereas with Ideal RRIP-AOB the speedup could be 20%. Thus, there is significant room for performance improvement with reuse-based replacement policies. Unfortunately, obtaining

this benefit in a practical manner is challenging as maintaining accurate per-line state in DRAM requires significant bandwidth for state-updates.

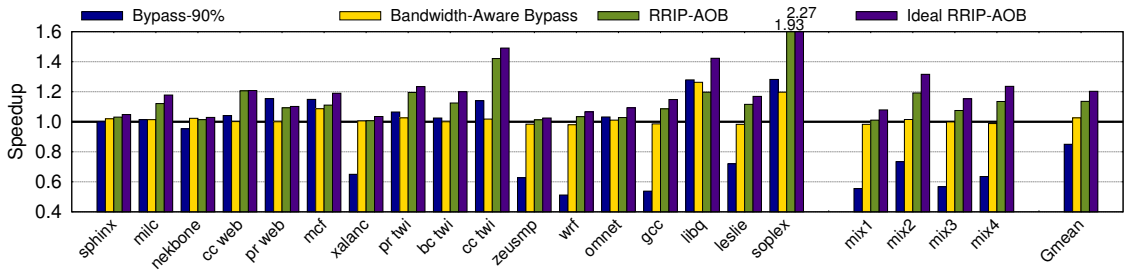



Figure 5.5: Speedup from different replacement policies over baseline always-install direct-mapped DRAM cache. (a) Bypass-90% causes 15% degradation, (b) Bandwidth-Aware Bypass provides 3% speedup, (c) RRIP-AOB that maintains state in DRAM provides 13% speedup, and (d) Ideal RRIP-AOB with no state-update cost provides 20% speedup

### 5.2.4 Dissecting the Bandwidth of Replacement-Updates

To highlight the bandwidth differences between Always-Install and RRIP-AOB, we show the bandwidth needed to implement replacement policy for Always-Install and RRIP-AOB. Always-Install simply has install bandwidth, whereas RRIP-AOB additionally needs bandwidth to *promote* and *demote* state. Figure 5.6 shows the replacement bandwidth consumption of RRIP-AOB, normalized to the replacement bandwidth of Always-Install. While RRIP-AOB reduces 76% of the install bandwidth (due to bypass), it has increased bandwidth consumption due to *promotion* and *demotion*. If we want to obtain most of the benefits of RRIP, we must develop methods to reduce this bandwidth overhead.

### 5.2.5 Potential for Improvement

RRIP-AOB with state in DRAM is a practical design as it does not require any SRAM overheads, and can be implemented without any changes to the DRAM cache (the extra bits for RRPV are taken from the unused ECC bits). However, it has two-thirds the speedup compared to the potential benefit of Ideal RRIP-AOB with no state-update costs. Ideally, we would like to get speedup similar to Ideal RRIP-AOB, and keep the implementation simplicity of RRIP-AOB. The goal of the next section is to develop such a solution.

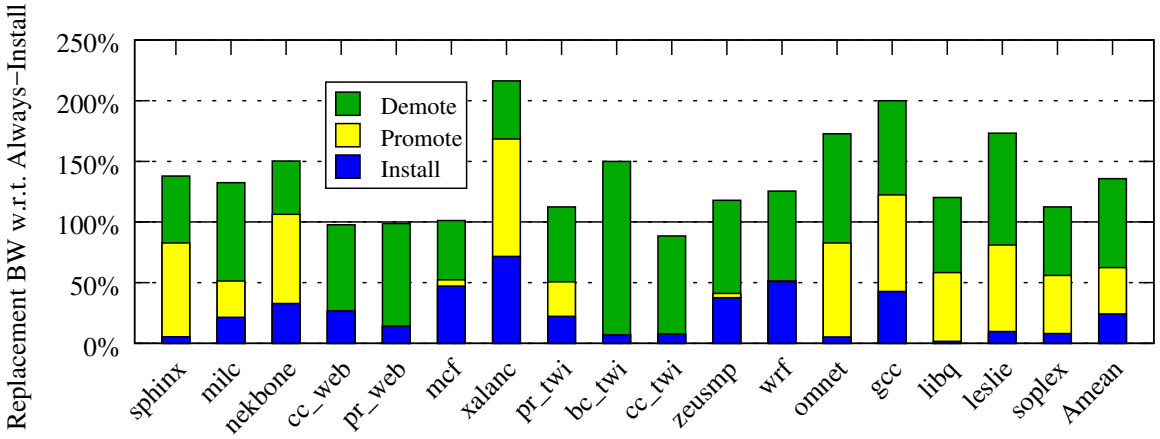



Figure 5.6: Replacement bandwidth (Install, Promote, Demote) of RRIP-AOB, normalized to replacement bandwidth (Install) of Always-Install. RRIP-AOB reduces install bandwidth but incurs state-update bandwidth.

RRIP-AOB simply suffers from high DRAM state-update cost. If we can find effective ways to mitigate this bandwidth overhead, we can get most of the benefits at little cost. We develop an insight that if we can do replacement updates in an efficient manner for only a subset of the lines, then we can reduce the bandwidth for replacement updates and still retain most of the benefits.

### 5.3 Design of Efficient Tracking of Reuse (ETR)

Demoting state on every cache bypass incurs significant bandwidth overheads–even if we choose to bypass the line, we still have to spend bandwidth to *demote* the replacement-state. We can avoid state-update costs if we have an effective way to infer an RRPV state. Our design reduces the bandwidth consumed in performing updates of the replacement state by doing the updates for only a subset of the lines and using their replacement state to infer the replacement state of the other lines. Our solution is based on two key properties, *Coresidency* and *Eviction-Locality*, which we describe next.

### 5.3.1 Insight of ETR: Coresident Lines Have Similar Reuse

*Coresidency* indicates that at any given time if a line is present, then several other lines belonging to that region are also present in the cache. Coresidency indicates that there is some amount of spatial locality in the reference stream, even if such spatial locality is not perfect. A 4KB region contains 64 lines each of 64 bytes. Therefore, the maximum number of coresident lines for a region would be 63. Figure 5.7 shows the level of coresidency for our workloads. In general, the workloads have between 16 to 45 coresident lines. We note that although this is lower than perfect spatial locality, there are still a large number of lines coresident (even 4 coresident lines can amortize 75% state update cost). This shows there is potential for using one line to infer replacement state of many coresident lines.

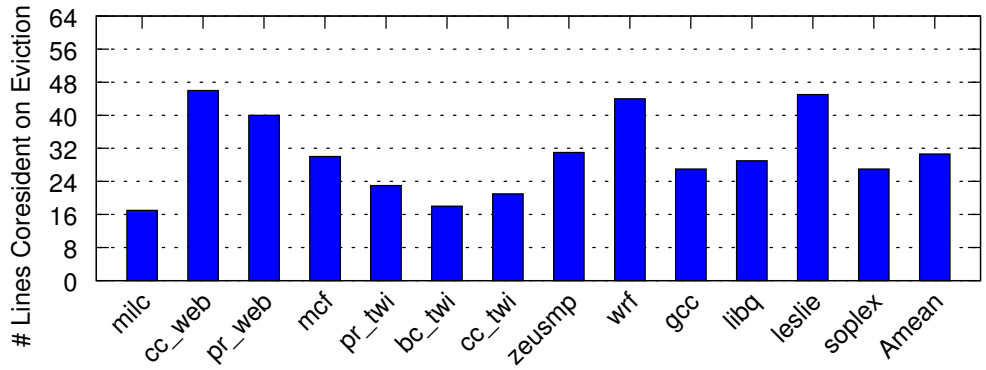



Figure 5.7: Coresidency in DRAM caches. Average number of coresident lines in a 4KB region on first line evicted from a region (workloads with L4 MPKI>1).

*Eviction-Locality* indicates that when a line gets evicted from the cache, then the replacement-state of the other coresident lines belonging to that region tend to have similar replacement-state as the line being evicted. Figure 5.8 shows the distribution of the RRPV of coresident lines, on an eviction from L4 (on the first line evicted from a region).

On an eviction, we typically observe 30 or more lines are coresident. In addition, we find the coresident lines generally have similar RRPV state (77% have RRPV$\geq$2). Together, this means that if we maintain accurate RRPV for just one of the coresident lines, then we can infer the RRPV state for the rest of the coresident lines in the region with reasonable accuracy. Our solution is based on exploiting this insight.

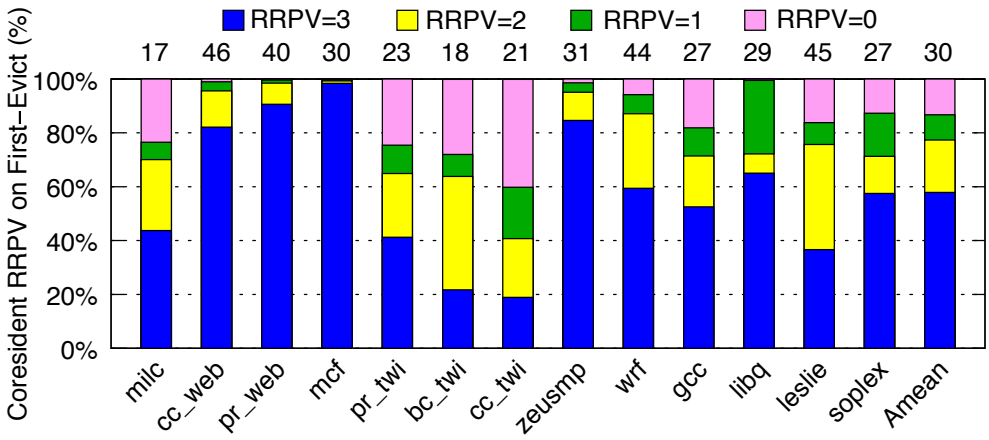



Figure 5.8: Distribution of RRPV of coresident lines on first line evicted from a 4KB region, for workloads with L4 MPKI>1. Average number of coresident lines shown above workloads. Eviction of one line indicates other lines in a region are likely to be evicted soon (RRPV≥2).

### 5.3.2 Design of ETR: Update Only the Representative-Line

We propose *Efficient Tracking of Reuse (ETR)* to reduce the bandwidth overheads of doing replacement updates in DRAM. We implement ETR on top of RRIP-AOB as an example. ETR is based on the insight of coresidency and eviction-locality. Instead of updating the replacement state of all of the lines in the cache, ETR updates the state of only one of the *Representative-Line* among all the coresident lines. The state of the Representative-Line is used in guiding the replacement states of the coresident lines. The design of ETR consists of three parts: (1) Selecting a Representative-Line in the region (2) Keeping accurate RRPV for *only* the Representative-Line, and (3) Using the representative's RRPV to infer coresident lines' RRPV to make bypass decisions.

To implement representative-update, we first need to pick a stable representative line. Prior work finds the first access to a region is relatively consistent [94]. If we maintain state for just the first conflicting set in a region, we can maintain good reuse information for the rest of the region without incurring extra bandwidth costs. Figure 5.9 shows an example of how ETR's RRPV-inference (i.e., representative-update and bypass-decision following) can be used to obtain similar install-policy and hit-rate at reduced update cost.

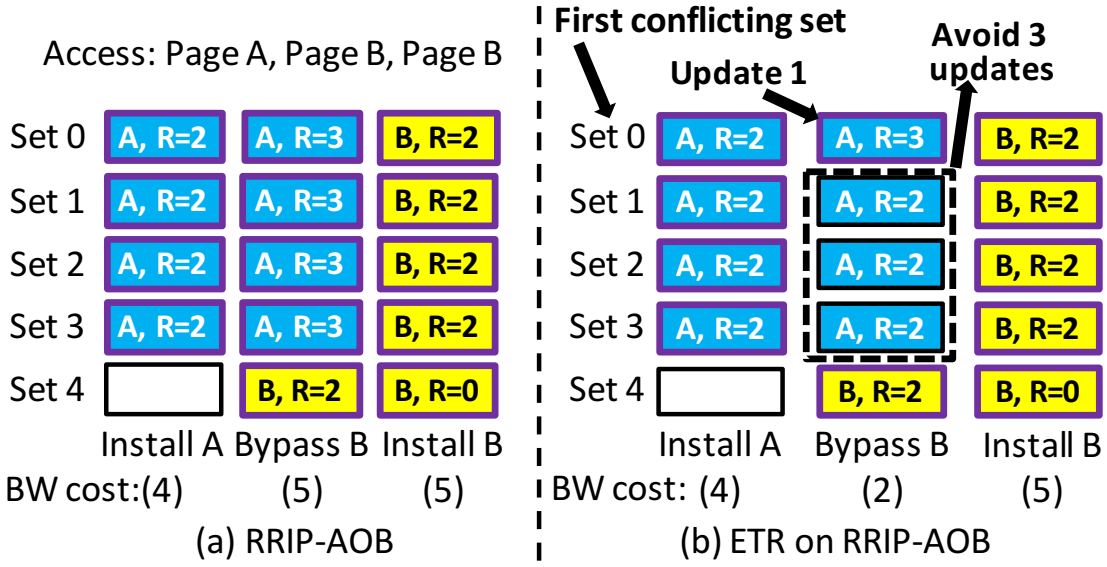



Figure 5.9: ETR's coordinated bypass-decision-following enables similar RRIP-AOB install policy, at reduced update-bandwidth. Dashed box denotes benefit.

If we first access 4 lines from region A at time 0, we install region A with RRPV=2. If 5 lines from region B are then accessed, Figure 5.9(b) shows that we can save bandwidth and *demote* only the state of the first conflicting set (being set 0). On second access to region B, set 0 with its RRPV of 3 will inform us that that region A was not used recently. This means that lines corresponding to region A have low reuse and should be evicted in favor of installing region B. We can then follow the region B install-decision for the rest of the lines. Such a policy will end up installing all of region B and result in an install policy similar to if we had maintained each state individually in Figure 5.9(a). As such, we can keep similar install policy and save update bandwidth with representative state update and bypass-decision following.

**Structures for ETR:** To implement representative state update and bypass-decision following, ETR maintains a *Recent-Bypass Table (RBT)*, in Figure 5.10. RBT tracks recently seen regions (*Region-ID*) and the bypass decision made for them (*Last-Bypass-Decision*). RBT enables us to find the representative first-conflicting-set in a region (as the first conflicting set would have miss in RBT), keep just that set's RRPV up-to-date, and remember the first-conflicting-set's bypass decision to inform bypass decision for the other lines in

the region (as the follower sets would hit in RBT and see previous decision made). We use a 128-entry RBT, which requires <512B of SRAM.

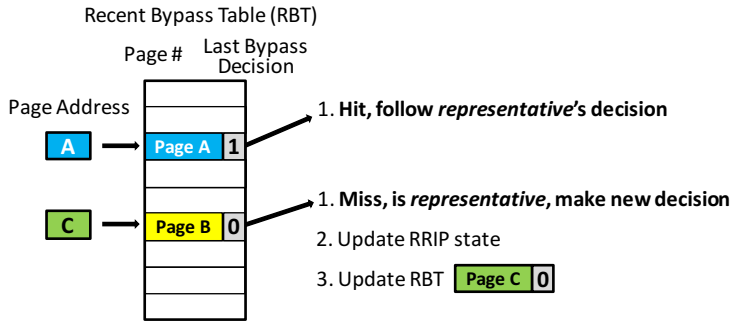



Figure 5.10: Design of Recent-Bypass-Table to enforce coordinated-bypass and coordinated-state-update. Demotions only occur on first miss to a region.

**Operation of ETR:** On cache miss, we index into RBT with Region-ID. If there is an RBT miss, we are currently accessing the representative first-conflicting-set in a region. In this case, we should make a bypass decision based on its RRPV, spend bandwidth to demote state if bypass was chosen, and update the RBT so later accesses can make an informed bypass decision. Otherwise, if there is an RBT hit, the region has been recently accessed and already had a bypass decision made, so we should follow the *Last Bypass Decision* to keep similar install policy and save on demotion bandwidth.

## 5.4 Methodology

### 5.4.1 Framework and Configuration

We use USIMM [86], an x86 simulator with detailed memory system model. We extend USIMM to include a DRAM cache. Table 5.1 shows the configuration used in our study. We model a configuration similar to a Intel Knights Landing (KNL) Sub-NUMA Cluster (one-eighth size). We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache). All caches use 64B line size. We model a virtual memory system to perform virtual to physical address translations. The L4 is a 2GB KNL DRAM-cache[87], which is direct-mapped and places tags with data in the

unused ECC bits. The parameters of our DRAM cache is based on HBM technology [3]. The main memory is based on non-volatile memory and assumed a latency similar to PCM and 3D-XPoint[9, 10, 8, 11, 12, 13, 14]: the read latency is 4X that of DRAM [4], and write bandwidth is worse than read bandwidth. We perform evaluations with DRAM-based memory in Section 5.5.7.

Table 5.1: System Config (KNL ⅛ Sub-NUMA Cluster)

| | |
|---|---|
| Processors | 8 cores; 3.0GHz, 2-wide OoO |
| Last-Level Cache | 8MB, 16-way |
| **DRAM Cache** | |
| Capacity | 2GB |
| Bus Frequency | 500MHz (DDR 1GHz) |
| Configuration | 4 channel, 128-bit bus |
| Aggregate Bandwidth | 64 GB/s |
| tCAS-tRCD-tRP-tRAS | 13-13-13-30 ns |
| **Main Memory (PCM)** | |
| Capacity | 64GB |
| Bus Frequency | 1000MHz (DDR 2GHz) |
| Configuration | 1 channel, 64-bit bus |
| Aggregate Bandwidth | 16 GB/s |
| tCAS-tRCD-tRP | 13-128-8 ns |
| tRAS-tWR | 143-160 ns |

### 5.4.2 Workloads

We use a representative slice of 2-billion instructions selected by PinPoints [88], from benchmarks suites that include SPEC 2006 [92], GAP [89], and HPC. For SPEC, we pick a sample of high intensity workloads that have at least two miss per thousand instructions (MPKI). The evaluations execute benchmarks in rate mode, where all eight cores execute the same benchmark. In addition to rate-mode workloads, we also evaluate 24 mixed workloads, which are created by randomly choosing 8 of the 15 SPEC workloads that have at least two MPKI. Table 5.2 shows L3 miss rates, and memory footprints for the 8-core rate-mode workloads in our study.

We perform timing simulation until each benchmark in a workload executes at least 2 billion instructions. We use weighted speedup to measure aggregate performance of the

workload normalized to the baseline and report geometric mean for the average speedup across all the 21 workloads (11 SPEC, 4 SPEC-mix, 5 GAP, 1 HPC). Note that to keep the graphs readable we only use 4 mixed workloads for all of our results. However, we provide key performance results for the set of the remaining 20 mixes in Section 5.5.4.

Table 5.2: Workload Characteristics

| Suite | Workload | L3 MPKI | Footprint |
|---|---|---|---|
| SPEC | soplex | 35.3 | 1.8 GB |
| | leslie | 22.1 | 623 MB |
| | libq | 30.1 | 256 MB |
| | gcc | 108.5 | 1.5 GB |
| | omnet | 29.1 | 1.2 GB |
| | wrf | 10.4 | 1.1 GB |
| | zeus | 7.0 | 1.6 GB |
| | xalanc | 7.4 | 1.5 GB |
| | mcf | 101.1 | 13 GB |
| | milc | 31.2 | 4.5 GB |
| | sphinx | 15.0 | 146 MB |
| GAP | cc twitter | 116.8 | 9.3 GB |
| | bc twitter | 101.2 | 13.5 GB |
| | pr twitter | 126.6 | 15.3 GB |
| | pr web | 24.8 | 15.1 GB |
| | cc web | 11.4 | 9.3 GB |
| HPC | nekbone | 13.71 | 44 MB |

## 5.5 Results

### 5.5.1 RRIP-AOB Impact on Misses

To isolate the impact of intelligent replacement without considering bandwidth costs specific to each DRAM cache organization, Table 5.3 shows the impact on average L4 misses when using different replacement policies for 1-way and 2-way caches, normalized to the baseline 1-way always-install policy. As expected, our RRIP-AOB achieves the highest miss reduction for 1-way caches as it enables intelligent reuse-based replacement policy for 1-way caches. Additionally, RRIP-AOB also achieves the highest miss reduction for 2-way caches. This is because RRIP-AOB can intelligently decide to bypass in the case the cache set is storing multiple useful lines.

Table 5.3: RRIP-AOB Impact on Misses for 1-2 Way L4

| Replacement Policy | Impact on Avg. L4 Misses |
|---|---|
| 1-way Always-Install | -0.0% |
| 1-way Probabilistic Bypass [17] | -1.6% |
| **1-way RRIP-AOB** | **-10.4%** |
| 2-way Random | -14.6% |
| 2-way LRU | -15.5% |
| 2-way RRIP | -19.7% |
| **2-way RRIP-AOB** | **-26.6%** |

### 5.5.2 ETR Impact on Bandwidth

ETR tries to reduce the bandwidth consumed in doing replacement updates (promotion and demotion of RRPV) to improve performance. To understand the effectiveness of ETR at reducing the bandwidth of state updates, we divide the bandwidth usage into three parts: installs, promote, and demote, and we normalize this consumption to the baseline design that uses bandwidth only for installs. Figure 5.11 shows the replacement bandwidth consumption of base RRIP-AOB and ETR on RRIP-AOB, normalized to Always-Install. ETR reduces 70% of bandwidth required for doing replacement state updates. These bandwidth savings result in speedup.

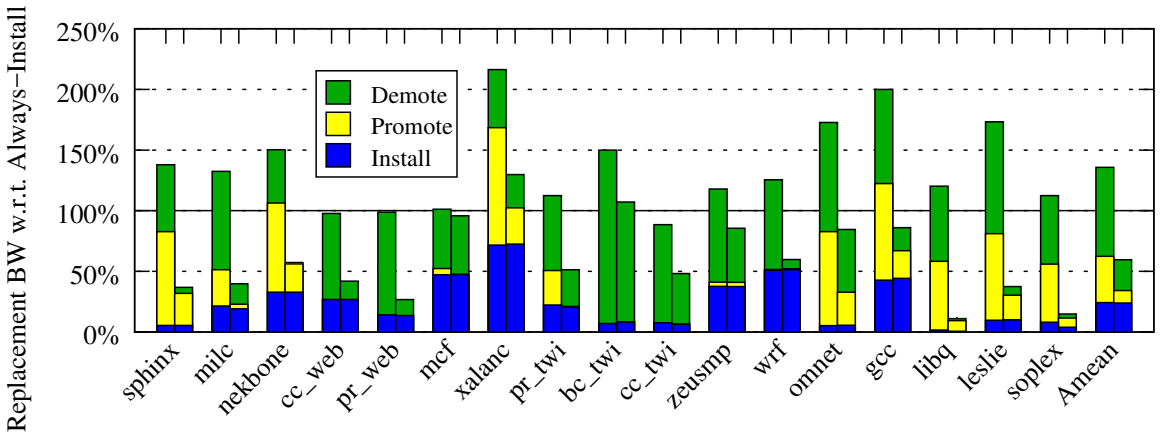



Figure 5.11: Replacement and Install bandwidth consumption of base RRIP-AOB [left] and ETR on RRIP-AOB [right], normalized to Always-Install. ETR reduces 70% of the bandwidth consumed in state-update.

### 5.5.3 ETR on RRIP-AOB Impact on Performance

Figure 5.12 shows the performance of RRIP-AOB, ETR on RRIP-AOB, and Ideal RRIP-AOB with no state-update costs. ETR on RRIP-AOB bridges 70% of the performance gap between RRIP-AOB and Ideal to achieve 18% speedup, while incurring negligible SRAM storage costs. Our proposed AOB and ETR make it practical to apply reuse-based policies to DRAM caches and get significant benefits while incurring negligible storage costs.

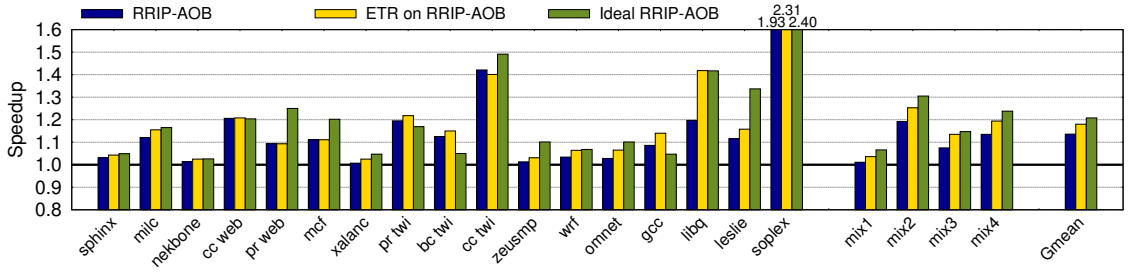



Figure 5.12: Performance of RRIP-AOB, ETR on RRIP-AOB, and an Ideal RRIP-AOB with no state-update costs. Coordinating bypass decisions with ETR reduces state-update needs, and enables RRIP-AOB to obtain 18% speedup.

### 5.5.4 Multi-programmed Workloads

To show robustness of our proposal to multi-programmed workloads, we evaluate over a larger set of 20 mix-application workloads. Figure 5.13 shows that ETR provides 19% speedup across 20 mixes, with no workloads experiencing slowdown.

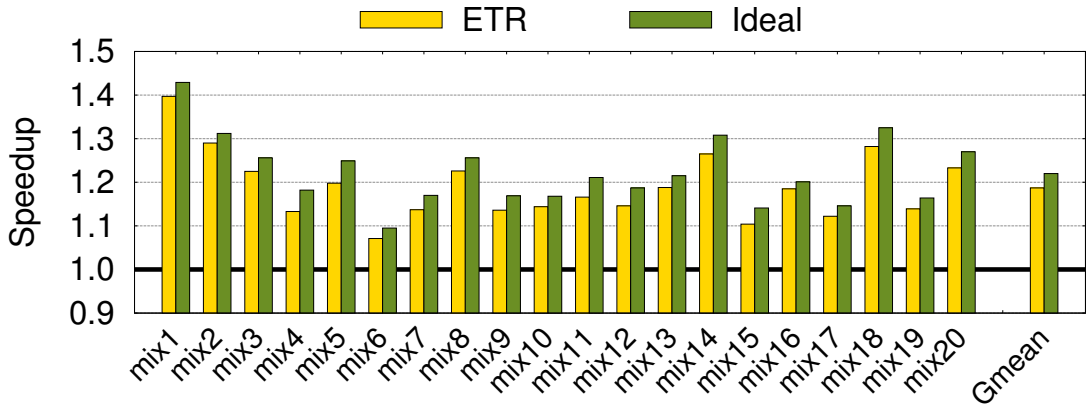



Figure 5.13: Speedup of ETR on RRIP-AOB and Ideal RRIP-AOB on multi-programmed workloads.

### 5.5.5 Storage Requirements

We analyze the SRAM storage overheads of ETR. ETR requires only a 128-entry 4B-per-entry Recent-Bypass Table, which needs 512B. Thus, our proposal can be easily built with negligible overheads within the memory controller.

For DRAM storage overheads required for RRIP-AOB, we use the fact that KNL-Cache has 28 unused bits in the ECC, which can be used for tag and metadata (see Figure 1.1). Baseline uses 8-10 bits for tag, valid, and dirty bit. RRIP requires just 2-bits for RRPV. Tag-entry becomes 12 bits, which fit in 28 available bits.

### 5.5.6 Impact of Cache Size

Table 5.4 shows the speedup of ETR as the size of the DRAM cache is varied from 1GB to 8GB. ETR on RRIP-AOB continues to provide significant speedup across different cache sizes, ranging from 16.4% at 1GB to 13.5% at 8GB. As expected, when the cache size is increased, larger portions of the workload fit, and there is reduced scope for improvement.

Table 5.4: ETR Sensitivity to Cache Size

| Cache Size | Avg. Speedup from ETR |
|---|---|
| 1.0GB | 16.4% |
| **2.0GB** | **18.0%** |
| 4.0GB | 17.4% |
| 8.0GB | 13.5% |

### 5.5.7 Impact of Memory Type

We use a non-volatile main memory for our studies, but our benefits are not limited to NVM-backed systems only. We compare BEAR's Bandwidth-Aware Bypass[17] with proposed ETR on DRAM-backed main memory in Table 5.5. ETR outperforms BEAR by intelligently bypassing lines and achieving better hit-rate and decent bandwidth benefits.

Table 5.5: ETR on DRAM-backed Memory

| | Bandwidth-Aware-Bypass | ETR |
|---|---|---|
| SPEC RATE | +7.4% | +17.0% |
| SPEC MIX | +1.7% | +16.0% |
| GAP | +4.8% | +26.6% |
| GMEAN26 | +5.7% | +19.0% |

### 5.5.8 Impact of Region Size

Table 5.6 shows the speedup of ETR as the region size is varied from 1KB to 4KB. Region size of 4KB (matching smallest OS page) provides best speedup of 18.0% as it amortizes the most replacement-update costs.

Table 5.6: ETR Sensitivity to Region Size

| Region Size | Avg. Speedup from ETR |
|---|---|
| 1KB | 17.8% |
| 2KB | 18.0% |
| **4KB** | **18.0%** |

### 5.5.9 Comparison to Other DRAM Cache Designs

In our study, we use the DRAM cache organization used in Intel's Knights-Landing (KNL-Cache)[1] that is direct-mapped and stores each tag next to its data. This organization is the commercial implementation of many research efforts that store Tag-With-Data[17, 6, 18] to improve latency and reduce bandwidth consumption. We compare with recent enhancements in Figure 5.5 (90%-Bypass and BAB[17]).

**Other Line-based DRAM Caches:** Alternative designs such as Sim et al.[23] take a different approach to storing tags via *tag grouping*. For such caches, a tag-only line is placed along with data in the same row buffer[5, 23, 24, 21, 22]. Timber is an enhancement that proposes to mitigate tag lookup by using a tag-cache and exploiting spatial locality (by co-locating tags and metadata from multiple sets)[24]. We compare with Timber as a representative of the grouped tag / metadata approach in Figure 5.14. Timber needs 2x lookup when workloads have large footprint and low spatial locality (e.g., *mcf* and *pr twi*), which limit its performance.

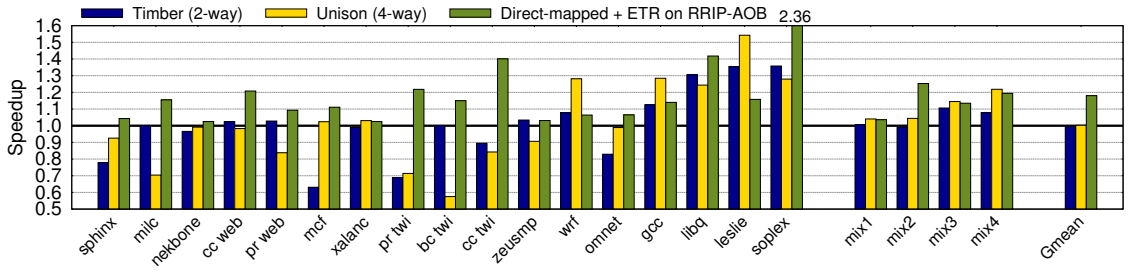



Figure 5.14: Speedup of line-based [24] and page-based [26] set-associative DRAM caches, rel. to baseline direct-mapped DRAM cache [1, 6]. Proposed ETR on RRIP-AOB enables direct-mapped DRAM caches to obtain the hit-rate benefits of intelligent cache replacement, without needing to pay additional bandwidth to maintain set-associative tags.

**Page-based DRAM Caches:** An alternate approach to designing DRAM-caches is to use large-granularity caches to reduce tag and metadata overhead, in hardware[25, 26] or software[27, 28, 29, 30]. The reduction in tag requirements enable more space for associativity and replacement metadata. Such large-granularity caches often implement recency-based replacement[26, 27, 28] or frequency-based replacement[29, 30] that would otherwise be too expensive in line-granularity caches. We compare with Unison cache[26] (hardware-managed, 4-way, page-based sectored cache with LRU replacement) as a representative of page-based designs, in Figure 5.14. The associativity and intelligent replacement Unison offers enable it to frequently outperform the baseline direct-mapped TIC cache. However, the large linesize of Unison often limits it from using a large portion of the cache (e.g., *pr twi* and *bc twi*). ETR on RRIP-AOB on the direct-mapped TIC cache, on the other hand, maintains the *cache utilization* benefits of fine-granularity caches while additionally offering *intelligent replacement*.

## 5.6 Extending to Enhanced Signature-Based Policies

Thus far, we have discussed AOB and ETR only in the context of RRIP. However, AOB and ETR are actually general techniques that enable formulating direct-mapped versions of replacement policies, as well as reducing the bandwidth needed to maintain replacement policy state. AOB and ETR can make even state-of-the-art signature-based policies [81, 82,

83, 84, 85] suitable for DRAM caches. We show how using *Signature-based Hit Predictor (SHiP)*[81] as an example.

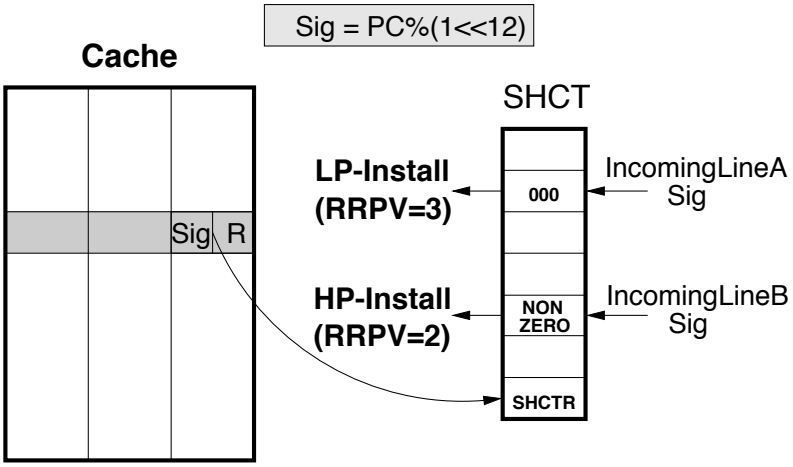



Figure 5.15: Operation and Organization of SHiP.

### 5.6.1 Operation of Conventional SHiP

SHiP works by observing and learning which signatures correspond to low-reuse lines, and installing lines accessed by those signatures at low priority, as shown in Figure 5.15. SHiP maintains signatures (PC) and reuse-bit (R-Bit) for tracking reuse of the signature. On eviction, the line increments or decrements a counter in the *Signature History Counter Table (SHCT)* based on the R-bit. On install, the SHCT decides if the incoming line is installed with High-Priority (RRPV=2) or Low-Priority (RRPV=3), based on signature.

### 5.6.2 Adapting SHiP to Direct-Mapped Cache

Conventional SHiP design always installs the incoming line, either with High-Priority or Low-Priority. Unfortunately, with a direct-mapped cache, doing so will degenerate into an always-install policy. We extend SHiP in the context of direct-mapped caches using the option of bypassing with *SHiP-AOB*. If the resident line has RRPV=3, then the incoming line is always installed. If the resident line has an RRPV of less than 3, then we bypass the incoming line. However, we then demote the RRPV of the resident line *only* if the incoming

line had a High-Priority install, and we skip the RRPV update for incoming lines with Low-Priority install. Thus, the advantage of SHiP-AOB is that it can reduce the bandwidth to perform *demotion* when the incoming line is predicted to have low reuse.

### 5.6.3 Implementing SHiP for DRAM Cache

To implement this bypassing version of SHiP for DRAM cache, we would need additional replacement metadata with each line: 12-bit signature + 1 R-Bit, in addition to the 2 bits for RRPV. Fortunately, these 15-bits of replacement metadata can still fit in the 18-20 unused bits available in ECC space of the DRAM Cache, so storage for the additional metadata for SHiP is not a concern. The SHCT table still needs to be implemented in SRAM (similar to conventional SHiP); however, the SRAM overhead of the SHCT is only 1.5 kilobytes.

Note that updating the R-Bit occurs concurrently with the Promote operation (setting RRPV to 0), so no additional bandwidth is required for tracking the R-Bit. We force install 2% of the time to get information on bypassed pages. We train write-reuse in a single SHCT entry as no PC is available.

### 5.6.4 Impact on Bandwidth

Figure 5.16 shows the bandwidth breakdown (in terms of install, promotion, and demotion operations) for ETR on RRIP-AOB, and ETR on SHiP-AOB, normalized to the bandwidth consumed in doing installs for the always-install design. ETR on SHiP-AOB achieves further reduction in state-update bandwidth compared to ETR on RRIP-AOB. ETR on SHiP-AOB is able to prevent the bandwidth for demotion operations by reducing update-cost when lines have predicted no-reuse, and this reduces the number of lines that need to be *promoted* back on hit. In particular, ETR on SHiP-AOB benefits workloads that had poor spatial locality (e.g., *mcf*).

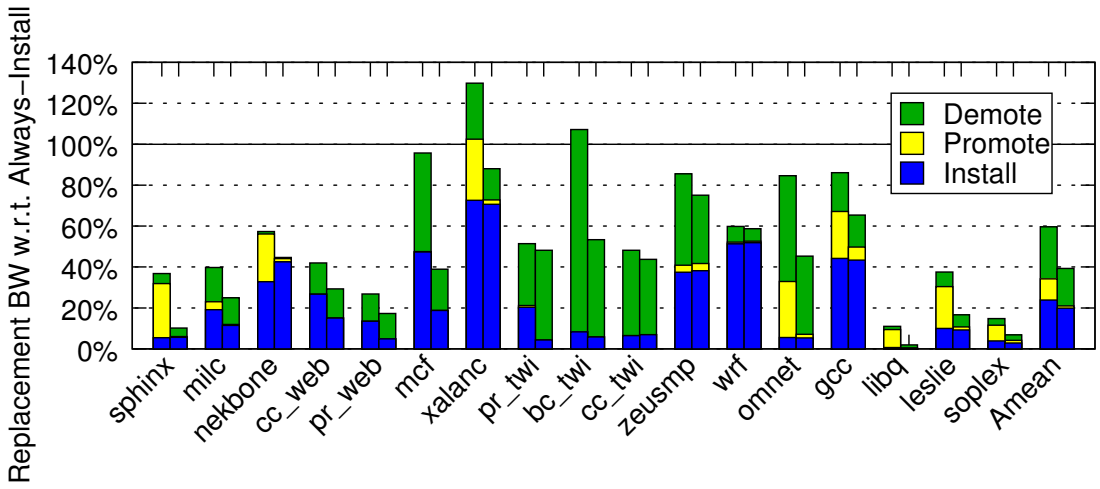



Figure 5.16: Bandwidth usage of ETR on RRIP-AOB [left] and ETR on SHiP-AOB [right], normalized to Always-Install. SHiP-AOB further reduces BW for state-update.

### 5.6.5 Impact on Performance

Figure 5.17 shows the speedup of ETR on RRIP-AOB, ETR on SHiP-AOB, and the Idealized version of SHiP-AOB with no state-update cost. Overall, ETR on SHiP-AOB achieves 21% speedup, achieving most of the 23% speedup with the Ideal design (which incurs impractical storage overheads).

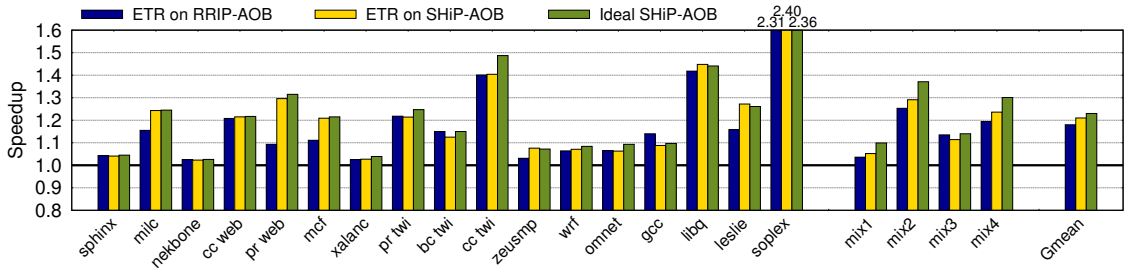



Figure 5.17: Performance of ETR on RRIP-AOB, ETR on SHiP-AOB, and Ideal SHiP-AOB with no state-update costs.

## 5.7 Towards Set-Associative Designs

We evaluate our solutions in the context of a direct-mapped DRAM cache. ACCORD [19] from Chapter 4 tries to make this DRAM cache set-associative (to improve hit rate albeit at a slight expense of bandwidth and latency [63, 54, 59]). We compare with the recently

proposed associative cache design, and show that ETR has higher potential as it improves both hit-rate *and* bandwidth. Nonetheless, ETR on RRIP-AOB can be used in conjunction with set-associative designs for DRAM caches for even better performance.

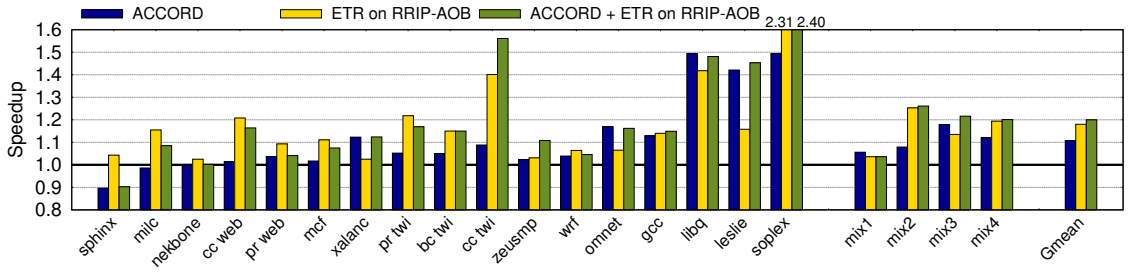



Figure 5.18: Performance of set-associative ACCORD, ETR on RRIP-AOB, and ACCORD with ETR on RRIP-AOB.

Figure 5.18 and Figure 5.20 shows ACCORD provides 10% speedup, as it reduces misses by 15% but can cost extra bandwidth consumption (due to way-mispredictions and looking up multiple ways on miss). Meanwhile, ETR on RRIP-AOB provides a higher 18% speedup, as it reduces misses by 10% while simultaneously reducing DRAM-cache bandwidth consumption. However, these ideas are not direct competitors. Associativity enables storing of multiple conflicting lines, and bypassing enables reducing install bandwidth while maintaining hit-rate. We design a solution that gets benefits of both.

### 5.7.1 Combining ACCORD and Bypassing

We want to obtain the hit-rate benefits associativity offers; however, we must do so while maintaining ACCORD's biased-install or we will face increased bandwidth cost to locate lines (due to frequent way-mispredictions).

To obtain benefits of both associativity and bypassing, we combine the two by viewing associativity as a *way-selection policy* and using a tiered decision. Figure 5.19 shows our tiered decision tree: ACCORD is first used to select the way, and then ETR is used to determine bypass policy.

For the first install to a region (Region Miss in ACCORD-RIT and ETR-RBT), we use ACCORD PWS to select which way to attempt install. This will have a biased probability

to install into a preferred-way, so we can maintain similar way-prediction accuracy. We subsequently use RRIP-AOB to decide install or bypass into this particular way, and correspondingly update the state. This enables us to keep ACCORD's flexibility to use both ways and maintain high way-prediction accuracy, as well as add RRIP's thrash-resistant replacement. This combination of the two enables high way-prediction accuracy (ACCORD) and bandwidth-efficient state-update and bypass (ETR on RRIP-AOB).

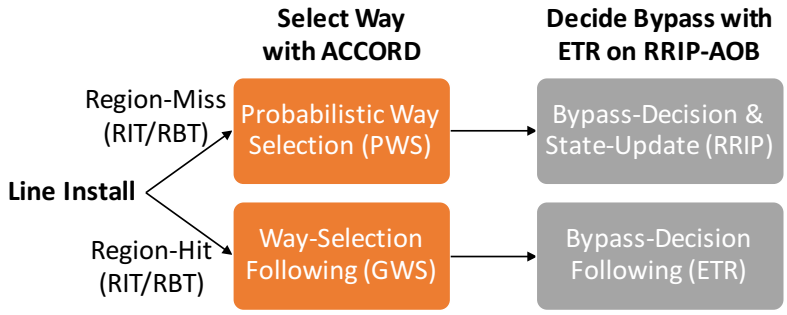



Figure 5.19: We first use ACCORD to select which way to attempt install, then use RRIP-AOB to decide bypass.

### 5.7.2 Effectiveness: Miss-rate and Speedup

Such a tiered decision process allows the bypass-policy ETR on RRIP-AOB to continue to obtain its thrash and scan-resistance, and bandwidth benefits. And, it enables ACCORD's associativity to utilize both ways to improve hit-rate, at minor cost to DRAM-cache bandwidth. Figure 5.20 shows the read MPKI of ACCORD, RRIP-AOB, and the combination of the two. ACCORD reduces miss-rate by 15%, RRIP-AOB reduces miss-rate by 10%, and the combination of the two reduces miss-rate by 20%.

Figure 5.18 shows the performance of ACCORD, ETR on RRIP-AOB, and the combination of the two. ETR on RRIP-AOB improves both hit-rate and bandwidth, whereas ACCORD improves hit-rate at the cost of bandwidth. The combination of ACCORD and ETR on RRIP-AOB enables 20% speedup, and show the concepts developed in ETR and RRIP-AOB are also applicable to set-associative DRAM cache designs. We discuss potential for improving other set-associative designs in Section 5.5.1.

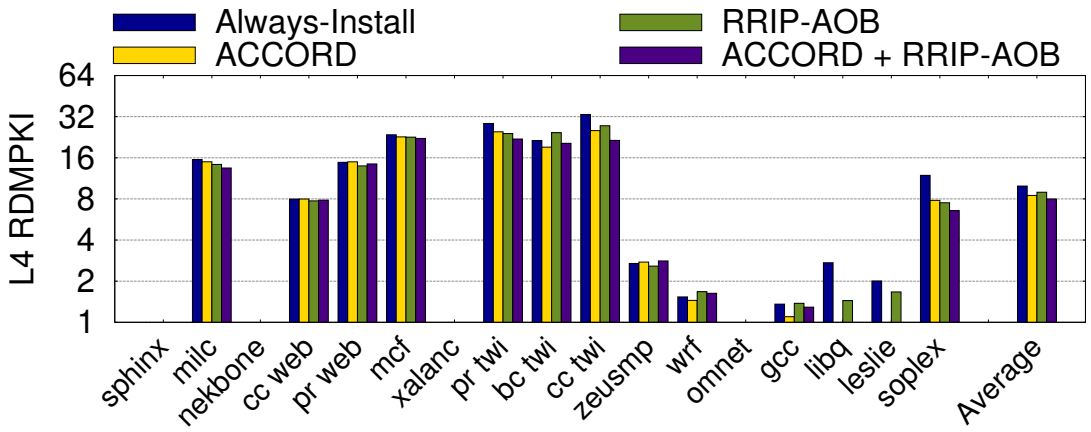



Figure 5.20: L4 Read-Miss-Per-Kilo-Instruction of Always-Install, ACCORD, RRIP-AOB, and ACCORD + RRIP-AOB. Combination enables 20% miss reduction.

## 5.8 Summary

This work investigates improving hit-rate for direct-mapped DRAM-caches by utilizing reuse-based replacement polices. We would like to use the most effective replacement policies to improve DRAM-cache hit-rate. Unfortunately, state-of-the-art policies based on reuse information are designed to compare multiple counter values within the set and then make the decision of a replacement victim. As such, these policies become ill-defined and inapplicable for direct mapped cache (they degenerate into either always-install or always-bypass policies). To make reuse-based policies, such as RRIP, applicable to direct-mapped DRAM caches, we propose a bypass formulation of RRIP called *RRIP Age-On-Bypass (RRIP-AOB)*. RRIP-AOB leverages the insight that the event of bypassing in a direct-mapped cache, should also age the reuse state of the resident line.

Similar to RRIP, RRIP-AOB requires per-line reuse counters. Maintaining such reuse state in DRAM costs significant bandwidth overheads (*promote* on hit, *demote* on bypass). We investigate methods to reduce bandwidth overheads of maintaining reuse state in DRAM. We propose *Efficient Tracking of Reuse (ETR)* to reduce state-update costs. ETR builds upon an observation that, at any given time, many lines from a 4KB region are *cores-ident* and *have similar reuse state*. If we select a representative (e.g., first-conflicting set in

a region) and maintain accurate reuse state for just that line, we can use that line's reuse to infer rest of the lines' reuse without update-cost. ETR reduces the bandwidth for tracking replacement state by 70% while maintaining similar hit-rate. Our evaluations with a 2GB DRAM-cache, show that ETR on RRIP-AOB provides a speedup of 18.0% while incurring an SRAM overhead of less than 1KB. ETR on RRIP-AOB performs within 2% of an idealized design that does not incur bandwidth for state-update.

# CHAPTER 6

# REDUCING MISS-BANDWIDTH COST WITH TICTOC DRAM CACHE

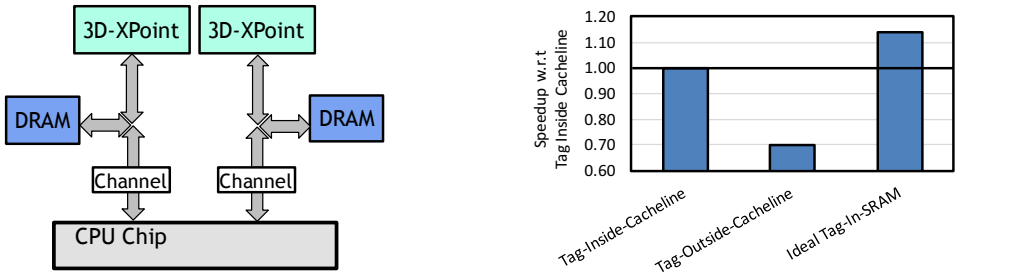
This chapter discusses how to reduce DRAM cache maintenance bandwidth costs for hits, misses, dirty-bit update, tag update, and installs by employing multiple tag organizations.

## 6.1 Introduction

This work investigates bandwidth-efficient DRAM caching for hybrid DRAM + 3D-XPoint memories. 3D-XPoint is becoming a viable alternative to DRAM as it enables high-capacity and non-volatile main memory systems. However, 3D-XPoint has several characteristics that limit it from outright replacing DRAM: 4-8x slower read, and even worse writes. As such, effective DRAM caching in front of 3D-XPoint is important to enable a high-capacity, low-latency, and high-write-bandwidth memory.

### 6.1.1 Target Configuration: Channel-Sharing Hybrid Memory

We would like to utilize insights learned from HBM+DDRx hybrid memories [5, 25, 6, 1, 24, 23, 26] to design effective DRAM caches in front of non-volatile memories, such as 3D-XPoint. However, we note that there are significant differences in setup and goals for a DRAM+3D-XPoint hybrid memory as compared to a HBM+DDRx hybrid memory. First, in a 3D-XPoint based hybrid memory, 3D-XPoint and DRAM will be sharing the same channel interfaces [15]. Second, as opposed to targeting bandwidth, DRAM caches in front of 3D-XPoint target *reducing read latency* and *improving write bandwidth and endurance* of 3D-XPoint, by servicing most data at the lower latency and higher write bandwidth of DRAM. An added complexity is that the DRAM cache and 3D-XPoint are likely to sit behind the same channel [95], as depicted in Figure 6.1(a). Such a set-up enables a balanced configuration where every channel has DRAM backing it. However, in such a channel-




(a) Channel-Sharing Hybrid Memory (b) Performance of DRAM Cache Organizations

Figure 6.1: (a) Channel-Sharing Hybrid Memory, and (b) Performance of hit-optimized Tag-Inside-Cacheline (TIC) [6], miss-optimized Tag-Oustide-Cacheline (TOC) [24], and idealized Tag-In-SRAM, normalized to TIC.

sharing set-up, bandwidth needed for maintaining DRAM cache state now comes directly at a cost to bus bandwidth available for memory. As such, there is a renewed need for bandwidth-efficient DRAM caches. We analyze prior DRAM caching approaches, highlight cases of bandwidth-inefficiency, and rigorously target remaining bandwidth overheads to develop a bandwidth-efficient DRAM cache suitable for DRAM + 3D-XPoint systems.

### 6.1.2 Available DRAM Cache Approaches

It is desirable to organize DRAM caches at the granularity of a cache line to efficiently utilize cache capacity, and to minimize the consumption of main memory bandwidth [25]. A key challenge in designing a large line-granularity cache is deciding where to store the tag and dirty-bit metadata. For a 4GB DRAM cache with 64B lines, there would be 64 million lines. Even if each metadata required 8 bits (6 tag, 1 dirty, 1 valid bit), this would result in 64MB storage for metadata. Next, we discuss various options for DRAM cache metadata management, and their implications on SRAM storage cost and bandwidth consumption.

*Tag In SRAM*

A costly method to design high performance DRAM caches is to simply maintain all of the tag and dirty bits in on-chip SRAM, and query the on-chip SRAM metadata to determine hit or miss, in a *Tag-In-SRAM* approach, shown in Figure 6.2(a). Assuming 1-byte

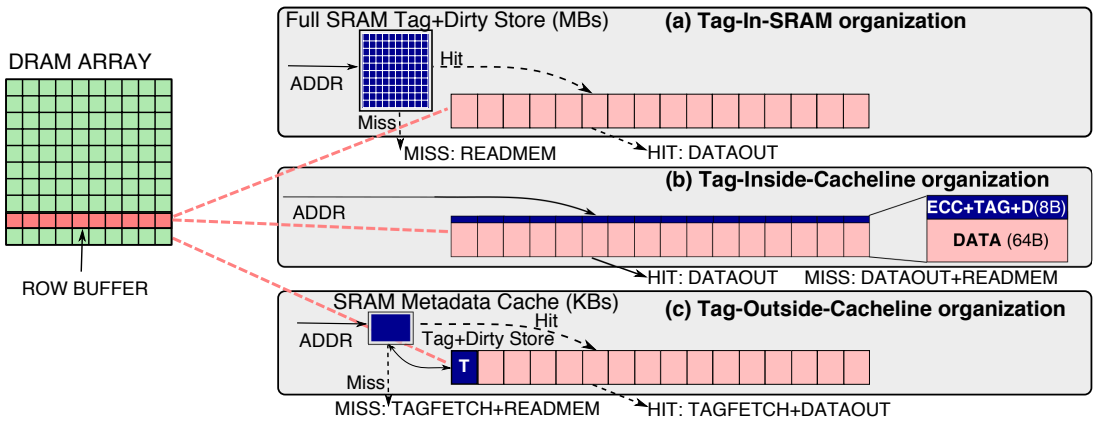



Figure 6.2: DRAM cache organization and flow for (a) idealized Tag-In-SRAM, (b) hit-latency-optimized Tag-Inside-Cacheline (TIC) [6], and (c) miss-bandwidth-optimized Tag-Outside-Cacheline (TOC) [24].

metadata per cache line, such an approach would require 64MB of SRAM for a 4GB cache (>20MB with sectoring [25, 31]). Table 6.1 shows the DRAM bandwidth consumption for such an approach. SRAM metadata is queried first to determine hit or miss. A hit can be serviced with one DRAM access. A miss can be serviced without a DRAM access. However, in preparation for installing the newly accessed line, the cache would need to perform an eviction of the resident line. If the resident line were clean, the location could be directly overwritten. However, if the resident line were dirty, the resident line would need to be read before writeback to memory. Hence, miss with eviction of clean line costs 0 bandwidth, and miss with eviction of dirty line costs 1 bandwidth. Such a design represents the minimum DRAM bandwidth needed for DRAM cache maintenance, and an upper-bound for performance. We aim to achieve Tag-In-SRAM performance at low SRAM cost.

*Tag Inside Cacheline (TIC)*

To reduce SRAM storage costs, one could store tags inside each line in DRAM [1, 6, 17] in a *Tag-Inside-Cacheline (TIC)* approach, shown in Figure 6.2(b). TIC optimizes for hit-latency by using a direct-mapped design and storing tag inside each data-line such that one access can retrieve both tag and data. Direct-mapped organization enables the controller to know which location to access, without waiting for tags.

Table 6.1: Bandwidth of DRAM Cache Approaches. $\rho$ is Metadata-Cache Miss Probability

| Organization (SRAM Cost) | SRAM (>20MB) | TIC (<1KB) | TOC (~32KB) |
|---|---|---|---|
| Hit | 1 | 1 | 1 + $\rho$ |
| Miss + Evict-Clean | 0 | **1** | 0 + $\rho$ |
| Miss + Evict-Dirty | 1 | 1 | 1 + $\rho$ |

Table 6.1 shows the bandwidth of such an approach. Hits are serviced with one DRAM access that retrieves both tag and data: in case of a tag match, the attached data can be used to service the request. However, misses also need to access tag in DRAM. As such, TIC is effective for hit-latency, but consumes extra bandwidth on misses. This approach of trading miss-bandwidth for hit-latency has been proven effective in commercial products [1], and, as such, we use the TIC organization [6] as our baseline.

Setup: We store metadata alongside data in unused ECC bits similar to Intel's Knights Landing [1]. TIC additionally employs a small hit-miss predictor to guide when to access cache+memory either in a parallel or serial manner (needs <1KB SRAM storage overhead). We additionally include bandwidth-reducing enhancements from Chou et al. [17], such as DCP to reduce writeback probe.

*Tag Outside Cacheline (TOC)*

Another option with reduced SRAM storage costs, is to store metadata lines in a separate area of DRAM and bring them in as needed in a *Tag-Outside-Cacheline (TOC)*[24, 23, 26] approach, shown in Figure 6.2(c). To determine hit or miss, TOC first accesses a metadata line to get tag+dirty information for the requested data line, then routes the request appropriately to DRAM cache or to memory. Of note, each of these metadata lines actually stores tag+dirty information of several adjacent data lines. An enhanced design [24] proposes to cache the metadata lines in a small metadata cache to avoid repeated accesses to the same metadata line, and would amortize metadata lookup if there is spatial locality. Table 6.1 shows the bandwidth consumption of such an approach. In case of a metadata-

cache hit, TOC performs similar to idealized Tag-In-SRAM. For a metadata-cache miss, TOC spends additional bandwidth to access the metadata. Overall, TOC has the potential for reducing miss bandwidth, but it can suffer from significant bandwidth overhead when the metadata-cache has poor hit rate (due to poor spatial locality).

Setup: We assume 1-byte metadata (6 tag, 1 dirty, 1 valid bits), and 64 tags stored in each metadata entry. The metadata are stored in a separate part of DRAM, consisting of 64MB out of the 4GB DRAM capacity. Recently accessed metadata are stored in a 512-entry metadata cache, which requires 32KB of SRAM. Note that the metadata cache is sized to capture only spatial locality and not the working set of the DRAM cache, which would need megabytes of SRAM.

Optimizing for Latency: In the case of metadata-miss in the metadata-cache, we want to avoid the latency for serialized tag + cache-data access, as well as the latency for serialized cache-data + memory access. We employ a direct-mapped organization and a hit-miss predictor [6] for latency and bandwidth considerations. If predicted hit, we access tag + cache-data in parallel to save latency (direct-mapped organization dictates only one possible location for data), and serially access memory only if prediction is wrong to save memory bandwidth. If predicted miss, we access tag + memory in parallel for latency, and serially access cache-data only if prediction is wrong to save DRAM cache bandwidth.

### 6.1.3 Insight: Combine Metadata Approaches

A TIC approach has good *hit-latency*, but suffers from extra miss bandwidth. Whereas, a TOC approach has good *miss bandwidth* but incurs extra hit bandwidth. We have an insight that if we could use TIC for hits and TOC for misses, then we could potentially achieve both good hit and miss bandwidth.

We note that provisioning metadata for both TIC and TOC is relatively inexpensive: TIC simply uses spare ECC bits [1], and TOC needs to dedicate only ~1.5% of DRAM cache capacity to store metadata lines and employs 32KB SRAM for its metadata cache [24].

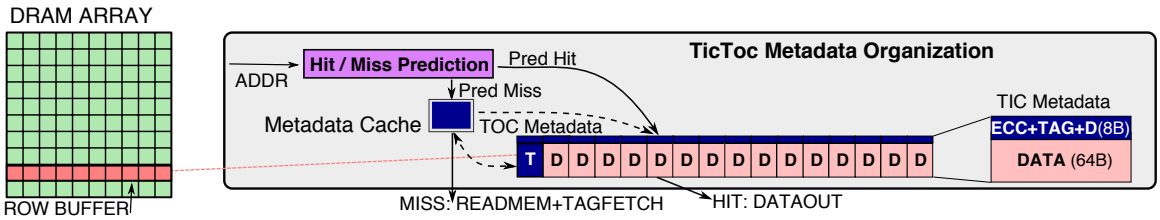



Figure 6.3: TicToc Metadata Organization queries hit/miss predictor to use TIC metadata for hits and TOC metadata for misses. TicToc enables good hit latency, and good hit/miss bandwidth.

Unfortunately, we find that naively combining both approaches actually leads to worse performance than TIC. This is because maintaining TOC tag and dirty bits consumes substantial bandwidth. To complete our design, we need to additionally develop effective solutions to reduce maintenance bandwidth for TOC. We discuss TicToc design next.

## 6.2 Design of TicToc

This section is organized is as follows: we describe how to provision and effectively use both TIC and TOC metadata in a TicToc organization, and describe how to reduce TOC metadata maintenance costs.

### 6.2.1 TicToc Metadata Organization

Figure 6.3 shows metadata organization of our *TicToc* design. TicToc provisions TIC metadata – tag-bits and dirty-bit are stored inside the cacheline in unused ECC bits, similar to commercial implementation [1]. In addition, TicToc provisions TOC metadata – metadata is stored in dedicated metadata lines corresponding to 1.5% of DRAM capacity, and cached as needed in a 32KB on-chip metadata cache. While provisioning both TIC and TOC metadata is relatively cheap, the complexity lies in utilizing TIC and TOC metadata appropriately to save on bandwidth for both hits and misses.

*TicToc Operation*

Figure 6.3 shows the operation of TicToc. Ideally, we want to use TIC metadata for hits and TOC metadata for misses. Our key insight is that one can use hit/miss prediction [6, 37]

to help guide when to use which metadata. Hit/miss predictors have been primarily used to hide the serialization latency that can occur from waiting on last-level cache response before sending memory access. They work by predicting which cache accesses are likely to miss, and sending both cache and memory requests in parallel to avoid serialization. We exploit an effective hit/miss predictor [6] to guide TicToc to use TIC metadata on likely-hit and TOC metadata on likely-miss. The common result: a hit is serviced in one cache access (TIC path), a miss with clean eviction directly goes to memory (TOC path), and a miss with dirty eviction goes to cache and memory (TIC path). An uncommon path of predict-hit actual-miss incurs serialization latency and bandwidth cost to access cache before memory. The other uncommon path of predict-miss actual-hit incurs extra memory access due to parallel lookup of cache and memory.

*TicToc Bandwidth Consumption*

To analyze effectiveness of TicToc, Figure 6.4 and Figure 6.5 shows the proportion of channel bandwidth being used for useful operations, install operations, and assorted maintenance operations, for baseline TIC and proposed TicToc, respectively. Useful operations include 3D-XPoint Read and Write, and DRAM Cache Hit and Writeback. Install operations refer to cache installs, which are important for improving hit-rate but cost bandwidth to write the line to DRAM. Lastly, Maintenance operations refer to bandwidth-wasting operations used to confirm a line is not resident: miss probes for TIC, and accessing and updating TOC metadata for TOC.

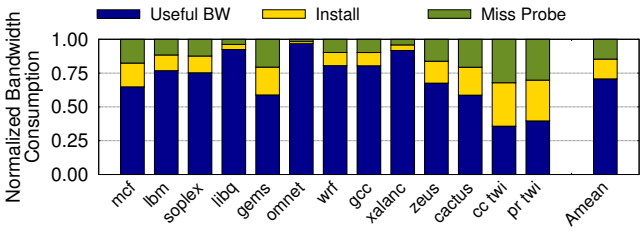



Figure 6.4: Breakdown of bus bandwidth consumption for TIC organization [6]. Workloads with low hit-rate waste significant bandwidth to confirm misses.

As expected, Figure 6.4 shows that TIC wastes bandwidth probing the DRAM cache to confirm misses. The proposed TicToc can utilize TOC to reduce such miss probes. However, Figure 6.5 shows that TicToc actually fares worse due to needing bandwidth to maintain TOC tag and TOC dirty-bit.

TOC tag-updates happen when the workload misses on a line and installs it. A large fraction of misses occur when a workload is accessing many lines in a new page, so misses generally have good spatial locality. In such cases, metadata-accesses/updates are amortized with the small metadata cache.

TOC dirty-bit-updates, on the other hand, occur upon eviction of a dirty line from an earlier level of cache. Eviction generally has poor spatial and temporal locality, so updating this information often takes significant bandwidth to read then update the TOC dirty-bit. We need effective methods that target reducing the cost of maintaining dirty information.

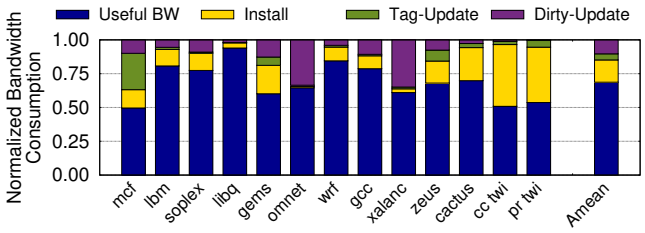



Figure 6.5: Breakdown of bus bandwidth consumption for proposed TicToc organization. Write-heavy workloads wastes significant bandwidth updating TOC dirty-bit.

### 6.2.2 Reducing TOC Dirty-Bit Tracking Costs

The main source of bandwidth overhead of TicToc is maintaining the dirty-bit for TOC metadata. We need effective methods to reduce the cost of tracking dirty information. We explain difficulty before describing solution.

*Understanding Dirty-bit Updates*

The dirty-bit update procedure starts upon an eviction of a dirty line from L3. First, we need to check the tag/dirty-bit line in the destination location of the DRAM cache to see if

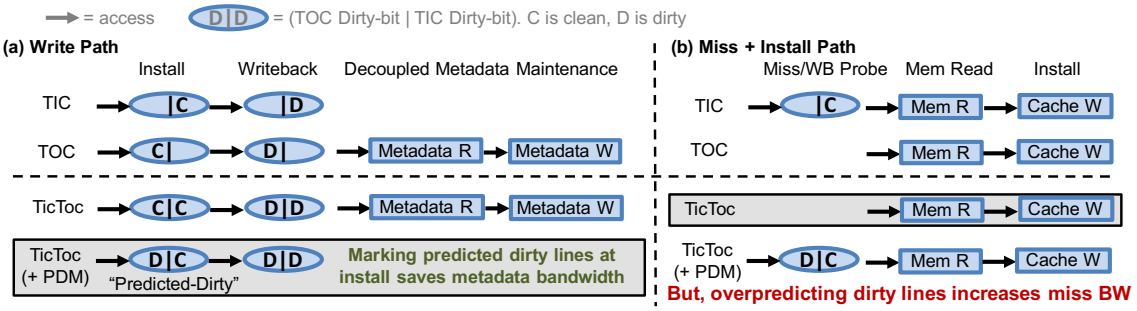



Figure 6.6: Bandwidth for a typical (a) write path and (b) miss+install path. TicToc+PDM adds "Predicted-Dirty" state, where TOC dirty-bit is installed as dirty but TIC dirty-bit is installed as clean. Installing lines in Pred-Dirty can (a) save TOC dirty-bit update, but (b) increase miss cost. Using Pred-Dirty only for write-likely lines can save bandwidth.

we can overwrite it (i.e., we must first evict a dirty tag-mismatched line). The common case is that the line evicted from L3 is resident in L4 in Figure 6.6(a). Chou et al. [17] proposes to eliminate the tag-check for this common case by maintaining a *DRAM Cache Presence bit (DCP)* alongside every line in L3. The DCP informs us that the same line is in both L3 and L4 – if the bit is set, the destination location has the same tag and can be directly overwritten (note that this optimization is included in our baseline). Second, the DRAM cache will then write the dirty line to the DRAM cache. Third, the cache will need to update any pertinent tag and dirty-bit metadata. The tag-update for TIC and TOC is uncommon, as typically L3 to L4 writebacks will hit. The dirty-bit-update for TIC is sent along with L4 install, so it does not incur bandwidth overhead. However, Figure 6.6(a)[TOC,TicToc] shows that the dirty-bit update for TOC often needs to be separately queried and potentially updated. This TOC dirty-bit update is TicToc's main source of bandwidth overhead.

The overhead of dirty-bit updates is comprised of two parts: repeated TOC dirty-bit checks for already-dirty lines, and the initial TOC dirty-bit update to mark clean-to-dirty transition. We target these two scenarios with two techniques.

*Reducing Repeated TOC Dirty-bit Checks with DRAM Cache Dirtiness bit (DCD)*

We have an insight that if we also knew the dirty state of the corresponding line in the DRAM cache, we can avoid the need to check the TOC dirty-bit. Instead, we can check (and update) the TOC dirty-bit only if the dirty status changes.

To enable this optimization, we propose to additionally store a *DRAM Cache Dirtiness bit (DCD)* alongside the DCP [17] next to each line in the L3 cache. The DCP stores information that the current L3 line is also resident in L4. Meanwhile, the DCD will additionally store the dirty status of that L4 line. We set the DCD on read of a dirty line from L4. On a DRAM cache write, we check both the DCP and DCD. If both DCD and DCP are set, we know the line is resident and already dirty in the TOC metadata – tag and dirty-bit will be unchanged and we do not need to fetch TOC. Hence, DCP reduces tag checks when tag will not be modified, and DCD reduces dirty-bit checks when dirty-bit will not be modified.

Figure 3.11 shows that DCD reduces dirty-bit check of many workloads that repeatedly write to same lines (e.g., *omnet*, *soplex*). However, there are several workloads (e.g., *zeusmp*) that are write-heavy and write to most lines only once – we want to reduce dirty-bit updates for those workloads as well.

*Reducing Initial TOC Dirty-bit Update with Preemptive Dirty Marking (PDM)*

For workloads that write-once to lines, we have an insight that if we can preemptively mark the dirty bit in the TOC at install time, we can avoid even the initial TOC clean-to-dirty update that would have occurred at L3 eviction time. We call this approach *Preemptive Dirty Marking (PDM)*.

Figure 6.6 shows the typical write and miss+install bandwidth for TicToc and one that preemptively marks TOC dirty-bit. Figure 6.6(a)[TicToc] shows a typical write path needs 4 accesses: a normal line would incur clean install, a write, and TOC dirty-bit read and write. Figure 6.6(a)[TicToc+PDM] shows that PDM can limit writes to 2 accesses. We add a new dirty state of "Predicted-Dirty," where TOC dirty-bit is marked as dirty but TIC

dirty-bit is marked as clean. If we install lines in "Predicted-Dirty," the TOC dirty-bit is set at install time, and even the initial TOC clean-to-dirty update can be avoided.

However, while early marking can save bandwidth on writes, PDM incurs a different problem on the miss path. Figure 6.6(b)[TicToc] shows a typical miss+install path needs 2 accesses: TOC metadata informs residence and dirtiness so miss+install can be accomplished with a memory read and a DRAM cache install. However, Figure 6.6(b)[TicToc+PDM] shows that PDM can increase miss+install to 3 accesses. For instance, if an otherwise clean line has been preemptively marked as dirty in the TOC dirty-bit, we would read the DRAM cache line in preparation for an eviction of a dirty line, thereby adding an extra DRAM read. Note that the Predicted-Dirty state does not cause extra memory writebacks as the miss/wb probe will find the TIC dirty-bit, and write back only if the data is dirty. Thus, being aggressive in marking lines as "Predicted-Dirty" will save write bandwidth, but it can come at the cost of increasing miss bandwidth.

Ideally, we want to avoid write costs by installing write-likely lines as "Predicted-Dirty", and avoid increased miss costs by installing write-unlikely lines as clean. However, if we install a write-likely line as clean, it will pay increased miss cost. Conversely, if we install a write-unlikely line as "Predicted-Dirty," it will pay cost to update TOC dirty bit. Hence, performance of Preemptive Dirty Marking is contingent on good classification of write-likely and write-unlikely lines at install-time to avoid both TOC dirty bit update and TIC miss probe bandwidth.

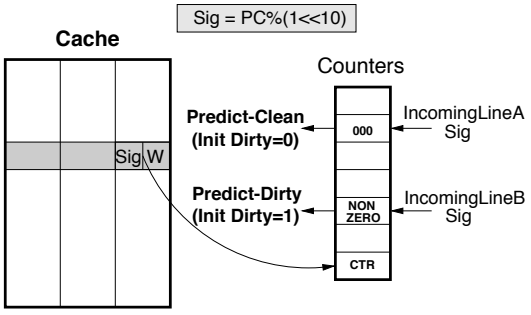



Figure 6.7: Signature-based Write Predictor learns which sigs correspond to eventual write, to aid PDM technique.

*Write Predictor for Preemptive Dirty Marking*

For accurate write-classification for PDM, we develop a *Signature-based Write Predictor (SWP)* to predict likeliness an incoming line will be written. SWP employs a sampling PC-based prediction, inspired by SHiP [81, 82]. Figure 6.7 shows structures and operation of SWP. SWP consists of write-behavior observation, learning, and prediction.

Observation is accomplished by maintaining signature (installing-PC in this case) and a written-to bit inside the metadata of each line (10 bits additional metadata for the 1% sampled lines, stored in TOC-metadata). Signature is set at install-time, and written-to bit is updated on first write to line. On eviction of such a sampled line, we get the information that this PC installed a line that was either written-to or never written-to in its cache lifetime.

Learning is then accomplished by storing observed write-behavior into a PC-indexed table of saturating 3-bit counters. On eviction of a line that has the written-to bit set, the counter corresponding to installing-PC is incremented. On eviction of a line that does not have written-to bit set, the counter corresponding to installing-PC is decremented. This counter table becomes a PC-indexed table of write-behavior.

Prediction is then simple – on install, the installing-PC is used to index into the counter-table to provide a write-likely or write-unlikely prediction. If the counter is non-zero, this PC has seen write behavior and the incoming line should be installed in "Predicted-Dirty" state to avoid initial TOC clean-to-dirty update. If the counter is zero, then this PC has not seen much write behavior and the incoming line should be installed as clean to avoid miss/wb probes.

*Accuracy of Write Predictor:* Effectiveness of PDM is contingent on good classification of write-likely (dirty) lines to reduce dirty-bit update cost, and write-unlikely (clean) lines to reduce miss-probe cost. Figure 6.8 shows the fraction of lines that are predicted clean or dirty, and actually clean or dirty. On average, SWP predicts clean and dirty with 92% accuracy, and enables PDM to save most dirty-update and miss-probe bandwidth.

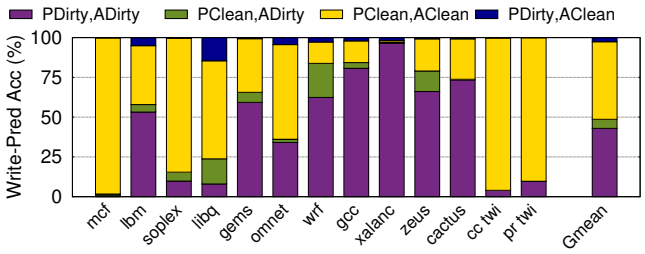



Figure 6.8: Accuracy of Write Prediction (P=predicted, A=actual). Low PClean/ADirty and PDirty/AClean rate reflects accurate write-behavior prediction.

## 6.3 Methodology

### 6.3.1 Framework and Configuration

We use USIMM [86], an x86 simulator with detailed memory system model. We extend USIMM to include a DRAM cache. Table 6.2 shows the configuration used in our study. We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache). All caches use 64B line size. We model a virtual memory system to perform virtual to physical address translations. The baseline L4 is a 4GB DRAM-cache[1], which is direct-mapped and places tags with data in unused ECC bits. The parameters of our DRAM cache are based on DDR4 DRAM technology [4]. The main memory is based on 3D-XPoint[9, 10, 8]: the read latency is ~6X, the write latency is ~24X that of DRAM, and there are 64 rowbuffers each 256B in size.

Table 6.2: System Configuration

| | |
|---|---|
| Processors | 8 cores; 3.0GHz, 4-wide OoO |
| Last-Level Cache | 8MB, 16-way |
| **DRAM Cache** | |
| Capacity | 4GB |
| Bus Frequency | 1000MHz (DDR 2GHz) |
| Configuration | 1 channel, 64-bit bus, shared |
| Aggregate Bandwidth | 16 GB/s, shared with Memory |
| tCAS-tRCD-tRP-tRAS | 13-13-13-30 ns |
| **Main Memory (3D XPoint)** | |
| Capacity | 64GB |
| Bus Frequency | 1000MHz (DDR 2GHz) |
| Configuration | 1 channel, 64-bit bus, shared |
| Aggregate Bandwidth | 16 GB/s, shared with DRAM |
| tCAS-tRCD-tRP | 4-80-0 ns |
| tRAS-tWR | 96-320 ns |

### 6.3.2 Workloads

We run 2-billion instruction slices selected by PinPoints [88], from benchmark suites of SPEC 2006 [92] and GAP [89]. For SPEC, we pick a subset of high memory intensity workloads that have at least 2 L3 misses per thousand instructions (MPKI). The evaluations execute benchmarks in rate mode, where all eight cores execute the same benchmark. In addition to rate-mode workloads, we also evaluate 21 mixed workloads, which are created by randomly choosing 8 of the 17 SPEC workloads. Table 6.3 shows L3 miss rates and memory footprints for the 8-core rate-mode workloads in our study.

We perform timing simulation until each benchmark in a workload executes at least 2 billion instructions. We use weighted speedup to measure aggregate performance of the workload normalized to the baseline and report geometric mean for the average speedup across all the 17 workloads (11 SPEC, 2 GAP, 4 MIX). We provide key performance results for additional 17 SPEC-mixed workloads in Section 6.4.6.

Table 6.3: Workload Characteristics

| Suite | Workload | L3 MPKI | Footprint |
|---|---|---|---|
| SPEC | mcf | 101.14 | 13.4 GB |
| | lbm | 49.3 | 3.2 GB |
| | soplex | 35.3 | 1.8 GB |
| | libq | 30.1 | 256 MB |
| | gems | 29.1 | 6.4 GB |
| | omnet | 29.0 | 1.2 GB |
| | wrf | 10.4 | 1.1 GB |
| | gcc | 7.6 | 1.5 GB |
| | xalanc | 7.4 | 1.5 GB |
| | zeus | 7.0 | 1.6 GB |
| | cactus | 6.5 | 2.6 GB |
| GAP | cc twitter | 116.8 | 9.3 GB |
| | pr twitter | 126.6 | 15.3 GB |

## 6.4 Results

### 6.4.1 Bandwidth of TicToc and Dirty-Tracking Optimizations

To analyze the effectiveness of our TicToc organization on reduccing DRAM cache maintenance, we analyze bus bandwidth consumption. Figure 6.9 shows the bandwidth breakdown of TicToc + dirty-bit optimizations. Overall, our approach eliminates nearly all of the TOC dirty-bit update bandwidth (decreased fraction from 10% to 0.8%) and frees up bandwidth for useful reads and writes. This bandwidth saving can provide corresponding benefits to performance.

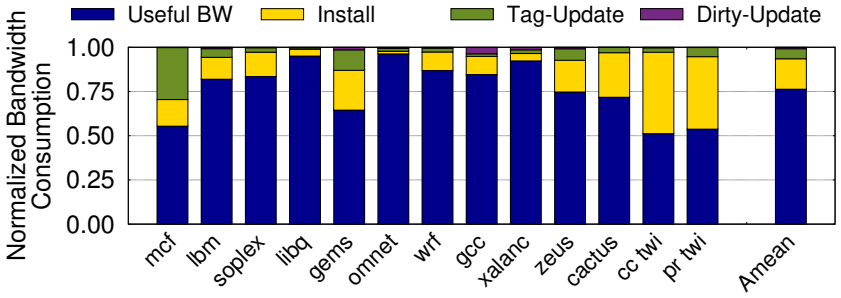



Figure 6.9: Breakdown of bus bandwidth for dirty-optimized TicToc. Dirty-bit updates are greatly reduced.

### 6.4.2 Performance of TicToc and Dirty-Tracking Optimizations

Figure 6.10 shows the speedup of TOC, our TicToc, TicToc with DCD, TicToc with PDM, and idealized Tag-In-SRAM, normalized to TIC approach. TOC performs poorly due to poor metadata-cache hit-rate, for 30% slowdown. Our TicToc reduces hit bandwidth, for 22% slowdown. Adding DRAM Cache Dirtiness bit reduces dirty-bit tracking for repeated writes to same lines, for 0% speedup. Adding Preemptive Dirty Marking reduces the initial dirty-bit update without incurring extra miss bandwidth due to accurate Write Predictor. Notably, TicToc+PDM achieves near idealized Tag-In-SRAM performance for most workloads for 10% speedup, without including worst-case *mcf* in the average.

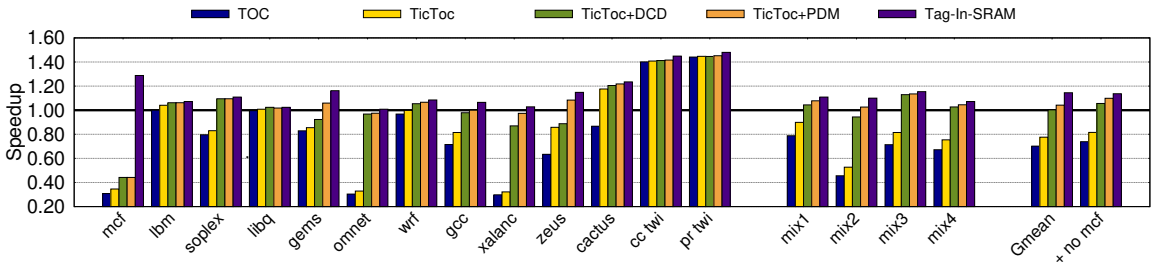



Figure 6.10: Speedup of TOC, proposed TicToc, TicToc with DRAM Cache Dirtiness bit, TicToc with Preemptive Dirty Marking (PDM), and ideal Tag-In-SRAM, normalized to TIC. TicToc+PDM performs near ideal for most workloads.

### 6.4.3 Storage Requirements

We analyze the SRAM storage requirements of our TicToc organization in Table 6.4. TicToc requires structures from its component TIC and TOC organizations. Inheriting from TIC, we need ˜1KB for PC-based hit/miss prediction [6], and 1 bit alongside each L3 line for DRAM Cache Presence bit to avoid tag-check for writes to resident lines [17]. Inheriting from TOC, we need 32KB for a metadata cache [24].

Specific to TicToc, to implement our dirty-bit optimizations, we need a 1-bit bit alongside each L3 line for DRAM Cache Dirtiness, and ˜1KB for our Signature-based Write-Predictor (512 entries of 3-bit counters with 9-bit PC tag). Our bypassing optimizations do not require additional space. In total, TicToc needs 34KB SRAM storage in the memory controller, with 2 bits alongside each L3 line.

Table 6.4: Storage Requirements of TicToc

| TicToc Component | SRAM Storage |
|---|---|
| Hit-Miss Predictor [6] | 1 KB |
| DRAM Cache Presence [17] | 1-bit / L3-line |
| Metadata Cache [24] | 32 KB |
| DRAM Cache Dirtiness | 1-bit / L3-line |
| Signature-based Write Predictor | 1 KB |
| TicToc | 34KB + 2-bits/L3-line |

### 6.4.4 Sensitivity to Metadata-Cache Size

The largest SRAM component of our TicToc proposal is the TOC metadata cache. Table 6.5 shows performance sensitivity of our TicToc proposal to metadata-cache sizing. We show average speedup of TicToc with dirty-bit optimizations, when employing metadata-caches with sizes ranging from 8KB to 64KB. The dirty-bit tracking optimizations enable TOC approaches to be much more effective with small metadata-cache sizes, as the metadata caches do not need to sized to handle writeback traffic that has poor spatial locality.

Table 6.5: Sensitivity to Metadata Cache Sizing

| Num. Entries | TicToc | TicToc (no mcf) |
|---|---|---|
| 128 (8KB) | -3.0% | +1.9% |
| 256 (16KB) | +1.5% | +7.0% |
| 512 (32KB) | +4.2% | **+10.0%** |
| 1024 (64KB) | +4.8% | +10.7% |

### 6.4.5 Impact of Channel-Sharing

DRAM and 3D-XPoint are likely to be behind the same channel to maximize the bandwidth out of each physical pin, as shown in Figure 6.1. Figure 6.11 shows the system performance of a *channel-shared* system (two channels of TIC DRAM cache + 3D-XPoint), normalized over previously assumed *dedicated-channel* systems (one channel of 2x TIC DRAM cache, and one channel of 2x 3D-XPoint). We find that channel-shared systems enable more balanced channel bandwidth usage due to each channel having a DRAM cache. For example, under high DRAM cache hit-rate, a channel-shared system would be able to utilize all channels, whereas a dedicated-channel system would only be able to use the half of channels employing DRAM caches. Such channel-shared approaches enable up to 40% speedup compared to the traditional dedicated-channel setups.

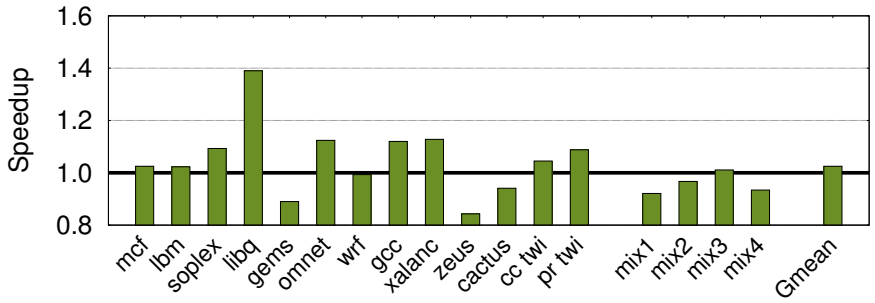



Figure 6.11: Speedup of Channel-Shared Hybrid Memory, over Dedicated-Channel Hybrid Memory. Channel-sharing enables up to 40% speedup.

### 6.4.6 Multi-program Workloads

To show robustness of our proposal to multi-programmed workloads, we conduct evaluations over a larger set of 17 mix-application workloads. Figure 6.12 shows that our dirty-optimized TicToc organization provides 11% speedup across 17 mixes, with no workloads experiencing slowdown.

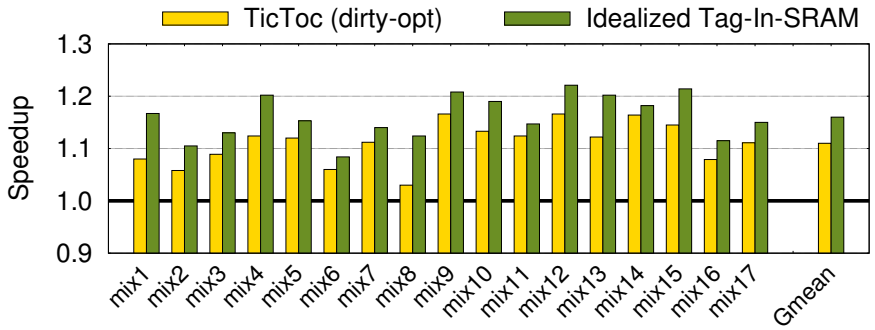



Figure 6.12: Speedup of TicToc with dirty-bit optimizations, and idealized Tag-In-SRAM, on mixed workloads.

## 6.5 Enhancement: Reducing Install Bandwidth With Write-Aware Bypass

When data has poor reuse, installing lines and updating TOC metadata wastes bandwidth. In fact, in such cases, employing a DRAM cache could actually hurt performance, as the line install and tag maintenance operations needlessly steal bus bandwidth from memory accesses. Figure 6.15 shows the performance of a setup without a DRAM cache, normalized to a setup with a TIC DRAM cache. We note that there are multiple workloads (e.g., *pr twi* and *cc twi*), for which "no DRAM cache" performs better than TIC. While one can

avoid this degradation by disabling the DRAM cache at boot-time, doing so would then hurt the cache-friendly workloads. Therefore, we need effective mechanisms to reduce the cost of unnecessary installs.

*Insight – Write-Aware Bypassing:* Prior work has proposed cache bypassing [17, 80, 79] to avoid unnecessary installs. On an L3 miss, one can bypass the DRAM cache and install the line only in L1/L2/L3 caches, thereby saving the DRAM cache install bandwidth. However, such bypassing must be done selectively and carefully, otherwise it may increase writes to 3D-XPoint memory, and degrade performance, endurance, and power.

### 6.5.1 Design of Write-Aware Bypassing

Figure 6.13 shows our Write-Aware Bypassing policy. We start with the default 90%-bypass policy proposed in [17], which bypasses 90% of all installs. While such aggressive bypassing was shown to work well for an HBM+DDR hybrid memory [17], we note that it can increase write traffic to the write-constrained 3D-XPoint memory. To address this problem, we add write awareness to the bypass policy. We augment the default bypass policy with a write-allocate condition, which requires that dirty L3 evictions would *always* install DRAM cache lines. Thus, the DRAM cache would act as a write buffer for 3D-XPoint memory. Unfortunately, the drawback of such an approach is that installing DRAM cache lines at the time of L3 evictions may result in significant tag-update costs. L3 evictions often have poor spatial locality, so TOC tag updates carried out at L3 eviction time exhibit poor metadata cache hit rates and incur extra DRAM accesses.

To amortize the TOC tag-update cost of our write-allocate policy, we propose *Preemptive Write-Allocate*, whereby we also *always-install* write-likely lines (predicted with SWP). Preemptive Write-Allocate enables our write-allocate installs to happen at L3 miss time. Such installs have higher spatial locality, resulting in more metadata cache hits and more effective amortization of TOC metadata updates.

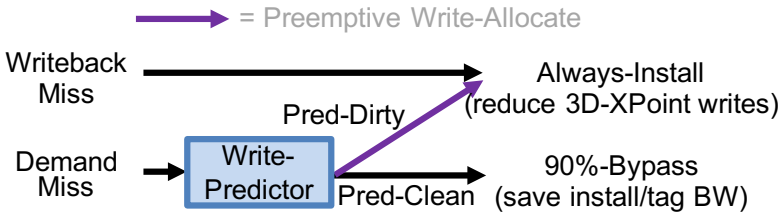



Figure 6.13: Write-Aware Bypass. Reduce install bandwidth by bypassing most write-unlikely lines. Reduce 3D-XPoint writes by installing write-likely lines.

### 6.5.2 Bandwidth of Write-Aware Bypassing

To understand the effectiveness of our install and metadata-update reducing optimizations, we show the bandwidth breakdown of our approach in Figure 6.14. Overall, we find that install-reducing optimizations can eliminate nearly all of the install bandwidth overheads and leave much more bandwidth for useful reads and writes. In total, the combination of our cache bandwidth reducing optimizations improves fraction of bandwidth going to useful operations (servicing reads / writes) from 70% to 90% on average.

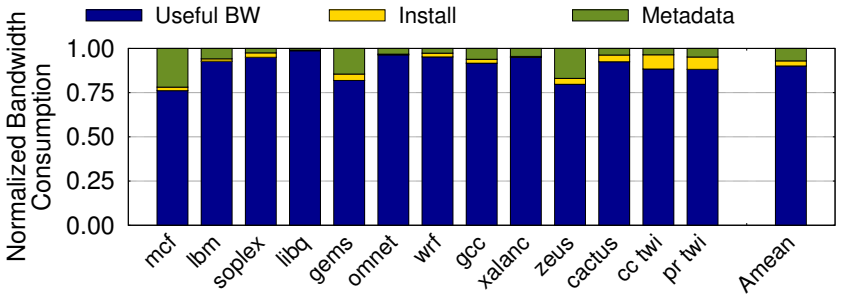



Figure 6.14: Breakdown of bus bandwidth for dirty-optimized TicToc w/ Write-Aware Bypassing. Installs are mitigated.

### 6.5.3 Performance of Write-Aware Bypassing

Figure 6.15 shows the performance of TicToc with dirty-optimizations, TicToc with 90%-bypass, TicToc with 90%-bypass and write-allocate, and TicToc with 90%-bypass and pre-emptive write-allocate, relative to TIC.

TicToc with dirty-bit optimizations does well for most workloads for an average 4.2% speedup, but can suffer for workloads with poor spatial locality and low hit-rate (e.g., *mcf*).

TicToc with 90%-bypass reduces install and TOC tag-update cost to improve speedup to 16.7%. Notably, the performance degradation for *mcf* has been mostly mitigated. TicToc with 90%-bypass and write-allocate enables effective write-buffering to improve speedup to 20.6%. Finally, TicToc with 90%-bypass and preemptive write-allocate further amortizes TOC metadata-update (e.g., useful for *zeusmp* and *pr twi*) to improve speedup to 23.2%.

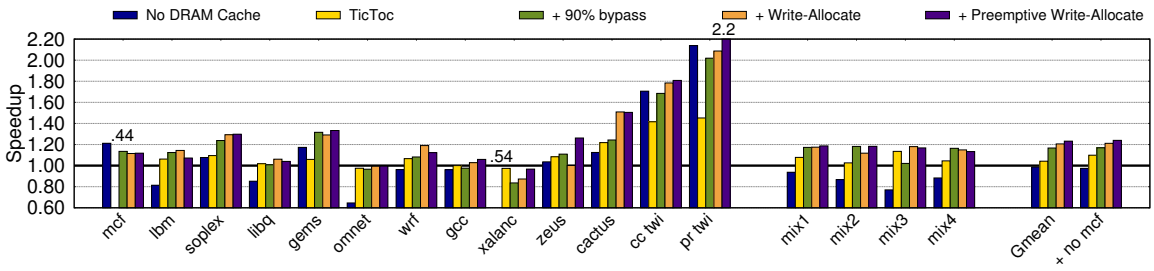



Figure 6.15: Speedup of a no-DRAM-cache configuration, proposed TicToc organization, adding 90%-bypass, adding Write-Allocate, and adding Preemptive Write-Allocate, relative to TIC approach.

### 6.5.4 Putting it all together

Overall, our proposed techniques target all forms of DRAM cache maintenance bandwidth to achieve a bandwidth-efficient (>90% of channel bandwidth to useful operations) and low SRAM storage overhead (34KB) DRAM cache organization: *TicToc* improves hit and miss bandwidth, *DRAM Cache Dirtiness bit* and *Preemptive Dirty Marking* reduces dirty-bit-tracking bandwidth, and *Write-Aware Bypass* reduces install and tag-tracking bandwidth. Our TicToc with dirty-bit and install bandwidth reducing optimizations enables 23.2% speedup at the cost of only 34KB of SRAM.

## 6.6 Summary

This work investigates bandwidth-efficient DRAM caching for hybrid DRAM + 3D-XPoint memories. Effective DRAM caching in front of 3D-XPoint is critical to enabling a memory system that has the apparent high-capacity of 3D-XPoint, and the low-latency and high-

write-bandwidth of DRAM. There are two currently major approaches for DRAM cache design: (1) a Tag-Inside-Cacheline (TIC) organization that optimizes for hits, by storing tag next to each line such that one access gets both tag and data, and (2) a Tag-Outside-Cacheline (TOC) organization that optimizes for misses, by storing tags from multiple data lines together in a tag-line such that one access to a tag-line gets information on several data-lines. Ideally, we would like to have the low hit-latency of TIC designs, and the low miss-bandwidth of TOC designs. To this end, we propose a *TicToc* organization that provisions both TIC and TOC to get the hit and miss benefits of both.

However, we find that naively combining both techniques actually performs worse than TIC individually, because one has to pay the bandwidth cost of maintaining both metadata. We find the majority of update bandwidth is due to maintaining the TOC dirty information. We propose *DRAM Cache Dirtiness Bit* that helps prune repeated dirty-bit updates for known dirty lines. We propose *Preemptive Dirty Marking* technique that predicts which lines will be written and proactively marks the dirty bit at install time, to help avoid even the initial dirty-bit update for dirty lines. To support PDM, we develop a novel PC-based *Write-Predictor* to aid in marking only write-likely lines. Our evaluations on a 4GB DRAM cache in front of 3D-XPoint show that our TicToc organization enables 10% speedup over the baseline TIC, nearing the 14% speedup possible with an idealized DRAM cache design with 64MB of SRAM tags, while needing only 34KB SRAM.

# CHAPTER 7

# CONCLUSION

DRAM caches are important for enabling effective heterogeneous memory systems that can transparently provide the bandwidth of high-bandwidth memories and the capacity of high-capacity memories. This dissertation investigates enabling intelligent cache management for tag-inside-cacheline DRAM cache designs. Such a DRAM cache uses a direct-mapped design, co-locates the tag and data within the DRAM array, and streams out the tag and the data concurrently on an access. This direct-mapped design has been shown to be effective for enabling low latency and bandwidth-efficient tag access; however, such a direct-mapped design can have lower hit-rate and high bandwidth cost to confirm misses. DRAM caches suffer from conflict misses and need high bandwidth cost to confirm misses; simple architectural innovation that addresses the challenges of enabling associativity, replacement policy, and reduced miss probes at low bandwidth cost can improve the performance of heterogeneous memory systems.

Chapter 3 investigates cache compression as a means to enable DRAM cache associativity and improve bandwidth. Cache compression allows fitting more lines in a set, provided the lines are compressible. This dissertation investigates enabling cache compression for DRAM caches via simple modifications local to the memory controller. The techniques enable improving cache associativity and capacity to obtain 7% speedup on average. However, this dissertation finds that additional performance can be gained by tuning cache compression to enable bandwidth benefits as well. To enable bandwidth benefits, this dissertation proposes compressing adjacent lines together and using location prediction – this enables retrieving multiple useful lines per DRAM cache access while avoiding the bandwidth to probe multiple locations. The bandwidth benefits in addition to capacity benefits enable 19% speedup on average.

Chapter 4 investigates way-prediction as a means to enable associative DRAM caches with the hit-latency of direct-mapped caches. Way-prediction can be used to enable associativity for DRAM caches in a bandwidth-efficient manner; however, traditional way-prediction techniques typically require significant per-line SRAM storage. This dissertation shows that coordinating way-install and way-prediction can enable a storage-efficient ($<$1KB SRAM) and accurate way-prediction. This dissertation proposes three coordinated way-prediction methods: *Probabilistic Way-Steering*, *Ganged Way-Steering*, and *Skewed Way-Steering*. The three techniques enable associativity at low bandwidth cost to provide 11% speedup.

Chapter 5 investigates intelligent bypass policies as a means to enable replacement policies for direct-mapped DRAM caches. Cache replacement policy can improve hit-rate by predicting which lines will have reuse and keeping in the cache longer. This dissertation discusses how to enable replacement policies for direct-mapped caches with two techniques: formulating replacement policies as bypass policies with *Age-On-Bypass*, and exploiting spatial locality to reduce cost of maintaining per-line replacement state with *Efficient Tracking of Reuse*. The combination of the two techniques on RRIP reduce miss-rate by 10% and reduce 70% of the bandwidth cost of maintaining replacement state to achieve 18% speedup on average.

Chapter 6 investigates hybrid DRAM cache tag organizations as a means to reduce miss bandwidth cost of DRAM caches. The baseline *Tag-Inside-Cacheline (TIC)* organization is good for hits, but can suffer from needing bandwidth to check DRAM array to confirm misses. Alternatively, a *Tag-Outside-Cacheline (TOC)* organization that stores tags of multiple lines together can save on miss cost. This dissertation discusses how to intelligently combine both organizations in a *TicToc* organization that can provide both good hit and good miss path. Additional enhancements for dirty-bit tracking enables 10% speedup over baseline TIC organization.

## REFERENCES


[1] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, 2016.

[2] AMD, *Radeon's next-generation vega architecture*, Accessed: 2017-09-19, 2017.

[3] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[4] *Ddr4 spec (jesd79-4)*, JEDEC, 2013.

[5] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *MICRO '11*, ACM, 2011.

[6] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *MICRO '12*, IEEE Computer Society, 2012.

[7] Micron, "Hmc gen2," *Micron*, 2013.

[8] Intel and Micron, *A revolutionary breakthrough in memory technology*, 2015.

[9] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, 2019. arXiv: 1903.05714.

[10] Intel, *Fact sheet: New intel architectures and technologies target expanded market opportunities*, Accessed: 2019-03-20, 2018.

[11] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, 2011.

[12] Y. C. et al., "A 20nm 1.8v 8gb pram with 40mb/s program bandwidth," in *ISSCC '12*, 2012.

[13] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *IEEE*, 2010.

[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*, ACM, 2009.

[15] ArsTechnica, *Intel's crazy-fast 3d xpoint optane memory heads for ddr slots (but with a catch)*, Accessed: 2019-01-23, 2018.

[16] A. Ilkbahar, *Intel© optane™ dc persistent memory operating modes explained*, Accessed: 2019-03-20, 2018.

[17] C. Chou, A. Jaleel, and M. K. Qureshi, "Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches," in *ISCA '15*, ACM, 2015.

[18] C. Chou, A. Jaleel, and M. K. Qureshi, "Candy: Enabling coherent dram caches for multi-node systems," in *MICRO '16*, 2016.

[19] V. Young, C. Chou, A. Jaleel, and M. K. Qureshi, "Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction," in *ISCA '18*, 2018.

[20] V. Young, P. J. Nair, and M. K. Qureshi, "Dice: Compressing dram caches for bandwidth and capacity," in *ISCA '17*, ACM, 2017.

[21] C.-C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *PACT '14*, ACM, 2014.

[22] Z. Wang, D. A. Jimnez, T. Zhang, G. H. Loh, and Y. Xie, "Building a low latency, highly associative dram cache with the buffered way predictor," in *SBAC-PAD '16*, 2016.

[23] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked dram caches," in *ISCA '13*, ACM, 2013.

[24] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Computer Architecture Letters*, 2012.

[25] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA '13*, ACM, 2013.

[26] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *MICRO '14*, IEEE, 2014.

[27] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless dram cache," in *ISCA '15*, ACM, 2015.

[28] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for tagless dram caches," in *HPCA '16*, IEEE, 2016.

[29] G. H Loh, N. Jayasena, J. Chung, S. K Reinhardt, M. O'Connor, and K. McGrath, "Challenges in heterogeneous die-stacked and off-chip memory systems," in *SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, Feb. 2012.

[30] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *MICRO '17*, ACM, 2017.

[31] J. B. Rothman and A. J. Smith, "Sector cache design and performance," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, 2000.

[32] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *MICRO '14*, IEEE Computer Society, 2014.

[33] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *MICRO '14*, IEEE Computer Society, 2014.

[34] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "Silc-fm: Subblocked interleaved cache-like flat memory organization," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[35] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[36] A. Kokolis, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[37] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO '12*, IEEE, 2012.

[38] C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3d: Mitigating the numa bottleneck via coherent dram caches," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[39] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for GPU architectures," in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, 2013.

[40] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," in *MICRO '18*, 2018.

[41] A. R. Alameldeen, D. Wood, *et al.*, "Adaptive cache compression for high-performance processors," in *ISCA '04*, IEEE, 2004.

[42] S. Sardashti, A. Seznec, D. Wood, *et al.*, "Skewed compressed caches," in *MICRO '14*, IEEE, 2014.

[43] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *MICRO '13*, ACM, 2013.

[44] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *MICRO '16*, 2016.

[45] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, 2004.

[46] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT '12*, ACM, 2012.

[47] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2010.

[48] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *PICS '09*, ACM, 2009.

[49] S. Baek, H. G. Lee, C. Nicopoulos, and J. Kim, "Designing hybrid dram/pcm main memory systems utilizing dual-phase compression," *DAC '14*, Nov. 2014.

[50] Y. Du, M. Zhou, B. Childers, R. Melhem, and D. Mossé, "Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory," *TACO '13*, 2013.

[51] B. Abali, H. Franke, X. Shen, D. Poff, and T. Smith, "Performance of hardware compressed main memory," in *HPCA '01*, 2001.

[52] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *MICRO '13*, ACM, 2013.

[53] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *HPCA '19*, 2019.

[54] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *HPCA '96*, IEEE Computer Society, 1996.

[55] H.-C. Chen and J.-S. Chiang, "Low-power way-predicting cache using valid-bit pre-decision for parallel architectures," in *(AINA'05)*, 2005.

[56] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A way-halting cache for low-energy high-performance systems," in *ISLPED '04*, 2004.

[57] F. M. Sleiman, R. G. Dreslinski, and T. F. Wenisch, "Embedded way prediction for last-level caches," in *ICCSD '12*, 2012.

[58] J. J. Valls, J. Sahuquillo, A. Ros, and M. E. Gmez, "The tag filter cache: An energy-efficient approach," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[59] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO '99*, IEEE, 1999.

[60] J. J. Valls, A. Ros, J. Sahuquillo, and M. E. Gomez, "Ps-cache: An energy-efficient cache design for chip multiprocessors," *J. Supercomput.*, Jan. 2015.

[61] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee, "Way guard: A segmented counting bloom filter approach to reducing energy for set-associative caches," in *ISLPED '09*, ACM, 2009.

[62] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Comput. Syst.*, Nov. 1988.

[63] A. Agarwal and S. D. Pudar, *Column-associative caches: A technique for reducing the miss rate of direct-mapped caches*. ACM, 1993.

[64] A. Seznec, "A case for two-way skewed-associative caches," in *ISCA '93*, ACM, 1993.

[65] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *MICRO '10*, IEEE, 2010.

[66] W. A. Wong and J.-L. Baer, "Modified lru policies for improving second-level cache behavior," in *HPCA '00*, IEEE, 2000.

[67] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *MICRO '13*, ACM, 2013.

[68] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru: Simple and effective adaptive page replacement," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, 1999.

[69] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07*, ACM, 2007.

[70] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *PACT '08*, ACM, 2008.

[71] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *SIGMETRICS '90*, ACM, 1990.

[72] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand-based associativity via global replacement," in *ISCA'05*, IEEE, 2005.

[73] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA '00*, ACM, 2000.

[74] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *SIGMOD '93*, ACM, 1993.

[75] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Comput.*, Dec. 2001.

[76] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ISCA '10*, ACM, 2010.

[77] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *MICRO '12*, IEEE, 2012.

[78] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *ICCD '07*, IEEE, 2007.

[79] H. Gao and C. Wilkerson, "A dueling segmented lru replacement algorithm with adaptive bypassing," in *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, 2010.

[80] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Trans. Comput.*, Apr. 2008.

[81] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *MICRO '11*, ACM, 2011.

[82] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

[83] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *MICRO '43*, IEEE Computer Society, 2010.

[84] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *ISCA '16*, 2016.

[85] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *ISCA '18*, 2018.

[86] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S Pugsley, A Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: The utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[87] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, 2016.

[88] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, 2004.

[89] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, 2015.

[90] J. Guar, A. R. Alameldeen, and S. Subramoney, "Base-victim compression: An opportunistic cache compression architecture," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, 2016.

[91] S. Ohya, "Skewed compressed dram cache ni yori," 2016.

[92] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, Sep. 2006.

[93] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, Dec. 2011.

[94] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA '06*, IEEE Computer Society, 2006.

[95] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora, "Cascade lake: Next generation intel xeon scalable processor," *IEEE Micro*, 2019.