ARCHITECTING HETEROGENEOUS MEMORY SYSTEMS WITH 3D DIE-STACKED MEMORY

A Dissertation
Presented to
The Academic Faculty

by

Jaewoong Sim

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

School of Electrical and Computer Engineering
Georgia Institute of Technology
August 2015

ARCHITECTING HETEROGENEOUS MEMORY SYSTEMS WITH 3D DIE-STACKED MEMORY

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my advisor, Hyesoon Kim, for her guidance, support, and encouragement. She is the definition of a great advisor. She has provided me with every opportunity to conduct the highest quality of research while allowing me the freedom to pursue research topics that I was interested in. Her technical creativity, insightful feedback, and high standards for research have tremendously helped me focus on important problems, refine ideas, and write strong papers with a high level of clarity. I am extremely fortunate to have her as my advisor, and I can only hope that someday I will be as good as her.

I am also very fortunate to have the opportunity to work with Gabriel Loh at AMD Research. He has been a fantastic mentor, and I had great fun working with him. His sharp intellect and superb mentoring skills had a profound influence on my graduate career, and I have great respect for him as a mentor, a researcher, and a friend.

I would like to thank other members of my dissertation committee, Moinuddin Qureshi, Sudhakar Yalamanchili and Milos Prvulovic, for invaluable comments and feedback to improve this dissertation. In particular, I am grateful to Moinuddin Qureshi for motivating me with interesting research ideas and providing me with valuable advice and help during my graduate studies.

At Georgia Tech, I had great moments with my friends and colleagues. In particular, I owe many thanks to the HPArch members: Jaekyu Lee, Nagesh Lakshminarayana, Sunpyo Hong, Joo Hwan Lee, Minjang Kim, Pranith Kumar, Hyojong Kim, Jen-Cheng Huang, Dilan Manatunga, Prasun Gera, and Lifeng Nai. I also thank Sungkap Yeo, Seungjoon Paik, Daehyun Kim, Davide Pluda, Aaron Ciaghi, Seunghwa Kang, Nimit Nigania, Puyan Lotfi, Andrei Bersatti, Manoj Athreya, Jungju Oh, Ching-Kai Liang, and Sunjae Park.

I am fortunate to work with wonderful researchers outside of Georgia Tech. I would like to thank Chris Wilkerson, Alaa Alameldeen, Zeshan Chishti, and Shih-Lien Lu for their mentorship and friendship at Intel Labs. I also thank Vilas Sridharan, Mike O'Connor, and Mithuna Thottethodi for their help on my research.

Above all, I am deeply thankful to my parents, Wonsub Sim and Jungsoon Park. They have given me unconditional love and always supported my decisions throughout my life. Thank you and I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The main objective of this research is to efficiently enable 3D die-stacked memory and heterogeneous memory systems. 3D die-stacking is an emerging technology that allows for large amounts of in-package high-bandwidth memory storage. Die-stacked memory has the potential to provide extraordinary performance and energy benefits for computing environments, from data-intensive to mobile computing. However, incorporating die-stacked memory into computing environments requires innovations across the system stack from hardware and software. This dissertation presents several architectural innovations to practically deploy die-stacked memory into a variety of computing systems.

First, this dissertation proposes using die-stacked DRAM as a hardware-managed cache in a practical and efficient way. The proposed DRAM cache architecture employs two novel techniques: hit-miss speculation and self-balancing dispatch. The proposed techniques virtually eliminate the hardware overhead of maintaining a multi-megabytes SRAM structure, when scaling to gigabytes of stacked DRAM caches, and improve overall memory bandwidth utilization.

Second, this dissertation proposes a DRAM cache organization that provides a high level of reliability for die-stacked DRAM caches in a cost-effective manner. The proposed DRAM cache uses error-correcting code (ECCs), strong checksums (CRCs), and dirty data duplication to detect and correct a wide range of stacked DRAM failures—from traditional bit errors to large-scale row, column, bank, and channel failures—within the constraints of commodity, non-ECC DRAM stacks. With only a modest performance degradation compared to a DRAM cache with no ECC support, the proposed organization can correct all single-bit failures, and 99.9993% of all row, column, and bank failures.

Third, this dissertation proposes architectural mechanisms to use large, fast, on-chip memory structures as part of memory (PoM) seamlessly through the hardware. The proposed design achieves the performance benefit of on-chip memory caches without sacrificing a large fraction of total memory capacity to serve as a cache. To achieve this, PoM implements the ability to dynamically remap regions of memory based on their access patterns and expected performance benefits.

Lastly, this dissertation explores a new usage model for die-stacked DRAM involving a hybrid of caching and virtual memory support. In the common case where system's physical memory is not over-committed, die-stacked DRAM operates as a cache to provide performance and energy benefits to the system. However, when the workload's active memory demands exceed the capacity of the physical memory, the proposed scheme dynamically converts the stacked DRAM cache into a fast swap device to avoid the otherwise grievous performance penalty of swapping to disk.

CHAPTER I

INTRODUCTION

The improvement of microprocessors allows for processing an ever increasing amount of data these days. Unfortunately, memory performance has not been improved at the same rate as microprocessor performance over the decades. Consequently, we have been facing the *Memory Wall* [85] and *Memory Bandwidth Wall* [8] problems, and system performance has been increasingly limited by memory performance in modern computing systems.

1.1 The Problem: Unpromising Conventional Memory Systems

To mitigate the *processor-memory performance gap*, both industry and academia have put enormous efforts into conventional memory systems. From industry, we have been improving DRAM—which has been used as main memory for the past four decades—to deliver large amounts of data faster to the processor. From academia, a significant amount of research has been focused on memory management including caching [84], prefetching [21], and memory scheduling [66]. Despite all these efforts, however, the processor-memory performance gap has become more widened, and today's memory system *truly* has become the fundamental bottleneck in computing systems. To make the problem worse, the recent explosion in data makes the memory system to have more stringent requirements on performance, power, and energy than ever before.

We now reached the point where it is not viable to meet such requirements with conventional memory systems anymore, *even if they are intelligently managed.* Given such a situation, we project that future memory systems will require employing emerging memory technologies and architectures (e.g., die-stacked DRAM) to streamline memory systems. These emerging memory architectures create a new problem space; they cannot simply be put in a memory hierarchy as a drop-in replacement for conventional memory. To deploy them in commercial products, it is crucial to architecting them in an effective way that deals with the peculiarities of such emerging memory architectures (e.g., in die-stacked DRAM, all bits coming from the same DRAM chip, relatively long latency compared to SRAM). Furthermore, management techniques on conventional memory systems will also likely be ineffective for such emerging memory architectures; therefore, memory management techniques need to be redesigned for new memory architectures as well.

1.2 The Contributions: Efficiently Architecting Die-Stacked Memory and Heterogeneous Memory Systems

Among emerging memory architectures, this dissertation investigates using *3D die-stacked memory* and focuses on the question of how to manage resulting *heterogeneous memory systems*.[1] 3D die-stacking is an emerging technology that allows for large amounts of high-bandwidth and energy-efficient memory storage within a processor package. This in-package die-stacked memory has the great potential to provide extraordinary performance and energy benefits for computing environments, from data-intensive to mobile computing. Consequently, it is gaining significant industrial traction, with activities in the memory standard bodies (e.g., JEDEC) and multiple companies (e.g., Intel, AMD, NVIDIA, Qualcomm) seriously considering the integration of die-stacked DRAM into their future products.

With the die-stacked memory within a processor package, future memory systems are expected to be *heterogeneous*—gigabytes of memory structures on-chip that provide higher bandwidth and faster access time than off-chip memory—and it is important to integrate the die-stacked memory *seamlessly* into current and future computing systems and enable heterogeneous memory systems in a practical manner. So, the question is *how to make efficient use of gigabytes of die-stacked memory in the heterogeneous memory system.*

In general, there will be a number of options of managing the stacked memory, and the

[1]We define a *heterogeneous memory system* as a memory subsystem in which some portions of memory provide different performance and power characteristics than other portions of memory in the same node.

answers will likely to be varying depending on its deployment scenarios. For instance, the stacked memory could be better used as a large cache for the systems employing hundreds of gigabytes of off-chip memory (e.g., servers). However, for desktop or laptop systems where off-chip memory capacity is only a few gigabytes, the stacked memory may need to be managed as part of main memory (rather than as a cache) to fully utilize the memory capacity available in the system. Alternatively, for other scenarios where system's memory demands only occasionally exceed off-chip physical memory, we may want a hybrid design option of a cache and part of memory for the best use of the stacked memory.

To provide the benefits of die-stacked memory *across computing environments*, this dissertation proposes novel organizations and usage models while addressing the challenges of enabling die-stacked memory—scalability, resiliency, hardware management, and software interaction—in different deployment scenarios. The proposed organizations below would help system/chip vendors efficiently exploit die-stacked memory for their own needs.

Practical Die-Stacked DRAM Cache Architectures

To avoid dependencies on operating system vendors, maintain software transparency, and provide benefits to legacy software, using die-stacked DRAM as a large cache is one of the directions that both academia and industry are interested in. The key challenge in this direction is how to manage metadata when scaling to *gigabytes* of stacked DRAM caches. The prior state-of-the-art design requires a multi-megabyte SRAM structure (e.g., 8MB for a 2GB DRAM cache) in order to maintain *precise* information about the DRAM cache's contents, which is prohibitively costly to allow it to be deployed in commercial products. Furthermore, the design is also not scalable.

To make the cache approach more practical, this dissertation research proposes a DRAM cache architecture that employs two innovations [75], each of which exploits the bursty nature of memory requests. The first is a low-cost cache Hit-Miss Predictor (HMP) that *virtually eliminates* the hardware overhead of maintaining a multi-megabyte SRAM structure for precise metadata. We demonstrate that it is actually possible to *speculate* on whether a request can be served by the DRAM cache or main memory, with 97% prediction accuracy and a hardware cost of less than 1KB, by exploiting spatial correlation and the bursty nature of cache traffic. The second is a Self-Balancing Dispatch (SBD) mechanism that dynamically sends some requests to off-chip memory even though the request may have hit in the die-stacked DRAM cache. This makes effective use of otherwise idle off-chip bandwidth when the DRAM cache is servicing a burst of cache hits.

These techniques, however, are hampered by dirty data in the DRAM cache. We propose a hybrid write policy for the cache that simultaneously supports write-through and write-back policies for different pages. Only a limited number of pages are permitted to operate in a write-back mode at one time, thereby bounding the amount of dirty data in the DRAM cache. By keeping the majority of the DRAM cache clean, most HMP predictions do not need to be verified, and self-balancing dispatch has more opportunities to redistribute requests.

Resilience Support for Die-Stacked DRAM Caches

Many high-end markets (e.g., server, datacenter, high-performance computing) require superior reliability, availability and serviceability (RAS). Die-stacked memories will likely need to be more reliable than external memories because *DRAM stacks are not serviceable*. Compared to conventional dual-inline memory modules (DIMMs) that can be easily replaced, a die-stacked memory failure may require discarding the entire package including the perfectly functioning processor.

To address the reliability concerns, this dissertation proposes a series of modifications to a stacked-DRAM cache to provide both fine-grain protection (e.g., for single-bit faults) and coarse-grain protection (e.g., for row-, bank-, and channel-faults) while only utilizing commodity non-ECC stacked DRAM chips. Furthermore, these RAS capabilities can be selectively enabled to tailor the level of reliability for different market needs [76, 77].

Die-Stacked DRAM as Part of Main Memory

For some computing environments, depending on its deployment scenarios, making die-stacked DRAM invisible to overall system memory (i.e., used as a cache) could lead to a non-negligible loss of a performance opportunity. For stacked DRAM to be deployed in such computing environments, this dissertation conducts a study of how to effectively combine fast stacked DRAM and slow off-chip memory to create a single physical address space [72].

Such *heterogeneous main memory* is typically managed by the operating system (OS). To maximize the performance benefits of fast memory, the OS could allocate heavily used pages to the portion of the physical address space mapped to the fast memory. Alternatively, the OS could dynamically migrate memory pages between fast and slow memory runtime. However, this dissertation shows that an OS-managed heterogeneous system is often unable to capture pages that are highly utilized for short periods of time, thereby leading to a non-negligible loss of a performance opportunity.

Consequently, this dissertation proposes a practical, low-cost architectural solution to efficiently enable using large stacked DRAM as Part-of-Memory (PoM) *without the involvement of the OS*. The hardware-managed PoM architecture dynamically remaps regions of memory, based on their access patterns and expected performance benefits, and performs even better than an ideal OS-based migration design that assumes zero-cost migration overheads.

Stacked DRAM Interacting with Operating Systems

This dissertation presents a new usage model for die-stacked DRAM involving a hybrid of caching and virtual memory support. In the common case where system's off-chip physical memory is not over-committed, die-stacked DRAM operates as a cache to provide performance and energy benefits to the system. However, when the workload's demands exceed the capacity of the physical memory, the proposed scheme can dynamically convert (part of) the stacked DRAM cache into a fast swap device, with minimum OS and hardware changes, in order to avoid the otherwise grievous performance penalty of swapping to disk.

This work increases the value proposition of stacked DRAM. In the common case, any of the previously proposed DRAM cache organizations can be used to increase system performance. However, the proposed scheme allows for tolerating a moderate level of memory over-commitment and thus opens up new opportunities for datacenter operators to more aggressively consolidate multiple workloads/virtual machines on to the same physical servers; or alternatively, the operators can maintain the current levels of consolidation but reduce the amount of overprovisioning of main memory. Such optimizations can have a direct impact on both operating expenses (e.g., more consolidation leads to more powered-down machines which saves on power costs) and/or capital expenses (e.g., purchasing less memory if less over-provisioning is needed).

1.3 Thesis Statement

Die-stacked memory can provide performance benefits across computing environments once it is effectively exploited with low-cost architectural innovations to address its challenges: scalability, resiliency, hardware management, and software interaction.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter II provides background and summarizes related work. Chapter III discusses a practical die-stacked DRAM cache architecture. Chapter IV presents a design of providing resilience support for die-stacked DRAM caches. Chapter V proposes using large, on-chip memory as part of addressable physical memory space. Chapter VI proposes a hybrid scheme where the hardware and operating system cooperatively manage stacked DRAM. Lastly, Chapter VII concludes this dissertation and discusses future work.

CHAPTER II

BACKGROUND AND RELATED WORK

In this chapter, we provide an overview of conventional off-chip memory and die-stacked DRAM (Sections 2.1, 2.2, 2.3, and 2.4) and describe the prior state-of-the-art using die-stacked DRAM as a large cache (Section 2.5). We then discuss prior work related to this dissertation (Section 2.6).

2.1 DRAM Architecture

DRAM consists of arrays of bit-cells, where each bit-cell is comprised of a capacitor to store charge and an access transistor to enable reading or writing of the cell. Accessing bit-cells in a DRAM requires storing the bit-cell values in a row buffer, and all read and write operations effectively operate directly on the row buffer (rather than the actual bit-cells). When the row is no longer needed (or often when a different row is requested), the contents of the row buffer are written back into the original row of bit-cells and then a new row may be accessed.

The DRAM access mechanism is quite different from SRAM arrays. In the case of DRAM, an entire bank is occupied while the row is open, and therefore any requests to other rows in this bank will be delayed until the current operations complete (although operations in independent banks may proceed concurrently subject to DRAM timing and bus constraints). In an SRAM, the access paths are more easily pipelined, and so even if a request has been sent to a particular bank, subsequent requests need only wait a few cycles before they can proceed.

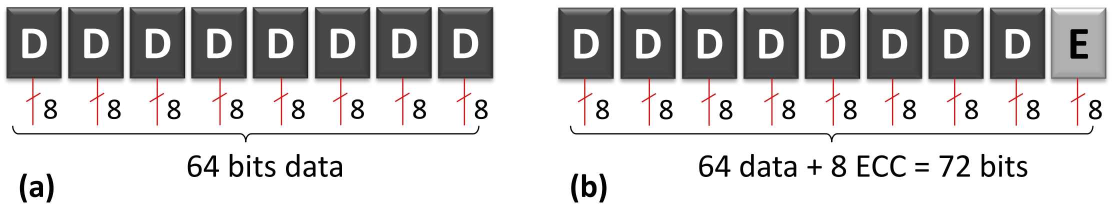


Figure 1: Organization of DRAM chips on a DIMM (one side, x8 chips) for (a) non-ECC DRAM, and (b) ECC DRAM.

2.2 Error Correcting Code for Conventional DRAM

Conventional off-chip memory is organized on dual-inline memory modules (DIMMs), with each side consisting of multiple DRAM chips. For the sake of simplicity, this discussion focuses on an organization in which each chip provides eight data bits at a time (“x8”), so the eight chips ganged together implement a 64-bit interface as shown in Figure 1(a). Typically a Hamming code [26] with seven check bits (or a similar code) is used to provide single-bit error correction (SEC) for 64 bits of data. In addition to the SEC code, an additional parity bit is provided to enable double-bit error detection (DED). This introduces eight bits of overhead per 64 bits of data, which is implemented by adding a ninth chip to the DIMM, as shown in Figure 1(b). All 72 bits are read in parallel, and the 8 bits of SECDED coding are used to check and possibly correct one of the 64 bits. Chipkill protection can be achieved in the same area overhead (although typically using x4 chips) by using a Reed-Solomon symbol-correction code and laying out the memory system so each DRAM chip contributes bits to exactly one error-correcting code (ECC) “symbol” [13].

A key advantage of the conventional ECC DIMM approach is that the silicon for each of the individual chips is identical, which allows the memory manufacturers to incur only the engineering expenses of designing a single memory chip. The difference comes from the design of the DIMM modules: the non-ECC version supports (for example) eight memory chips and the ECC version supports nine, but the engineering cost of designing and manufacturing multiple printed circuit boards is far cheaper than doing the same for multiple chip designs. Maintaining a single chip design also helps optimize a memory vendor's silicon inventory management.

2.3 Die-Stacked DRAM Organization

Die-stacked DRAM consists of one or more layers of DRAM with a very-wide data interface—using through-silicon vias (TSVs)—connecting the DRAM stack to whatever it is stacked with (e.g., a processor die). Whereas a conventional memory chip may provide only a four- or eight-bit data interface (the reason multiple chips are ganged together on a DIMM), a single layer of die-stacked memory can provide a much larger interface, such as 128 bits [17, 43]. Given this wider interface, all bits for a read request can be transferred across a single interface, and therefore all bits are sourced from a single chip in the stack, as shown in Figure 2(a).

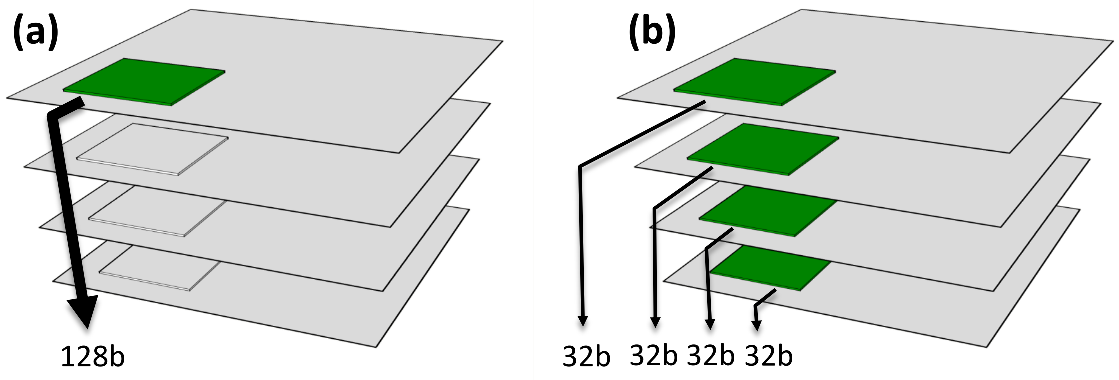


Figure 2: Reading data from a die-stacked DRAM with (a) all data delivered from a single bank from a single layer (similar to JEDEC Wide I/O [34]), and (b) data distributed across banks from four layers.

In theory, the stacked DRAM could be organized to be more like a DIMM, in that each of the N chips in a stack provides $\frac{1}{N}^{\text{th}}$ of the bits, as shown in Figure 2(b). This approach is undesirable for a variety of reasons. Requiring the parallel access of N chips means activating banks on all chips. This reduces peak bank-level parallelism by a factor of N, which reduces performance [46]. In addition to the N bank activations, accessing all chips in parallel requires switching N row and column decoders and associated muxes on each access, increasing both the power as well as the number of points of possible failure. Timing skew between different bits coming from different layers for the same request may also make the die-stacked I/O design more challenging. Distributing data across layers also limits flexibility in the design of the stacks. If, for example, data are spread across four layers, then DRAM stack designs will likely be constrained to have a multiple of four DRAMs per stack. In summary, a “DIMM-like” distribution of data across the layers of the DRAM stack is problematic for many reasons.

2.4 DRAM Failure Modes

Conventional DRAM exhibits a variety of failure modes including single-bit faults, column faults, row faults, bank faults, and full-chip faults [30, 80]. These faults can affect one or more DRAM sub-arrays and either be permanent or transient. Recent field studies on DDR-2 DRAM indicate that over 50% of DRAM faults can be large, multi-bit (row, column, bank, etc.) faults, and that DRAM device failure rates can be between 10-100 Failures in Time (FIT) per device, where 1 FIT is one failure per billion hours of operation [80]. Neutron beam testing also shows significant inter-device and inter-vendor variation in failure modes and rates [61].

The internal organization of a die-stacked DRAM bank is similar to an external DRAM, thus failure modes that occur in external DRAM devices are likely to occur in die-stacked DRAM. Die-stacked DRAM may also experience other failure modes, such as broken through-silicon vias (TSVs), and accelerated failure rates from causes such as negative-bias temperature instability (NBTI) and electromigration due to elevated temperatures from being in a stack. Some of these new failure modes (e.g., broken TSVs) will manifest as a single failing bit per row, while others (e.g., electromigration) may cause multiple bits to fail simultaneously.

The cost of an uncorrectable/unrepairable DRAM failure in a die-stacked context may be significantly more expensive than for conventional DIMMs. In a conventional system, a failed DIMM may result in costly down time, but the hardware replacement cost is limited to a single DIMM. For die-stacked memory, the failed memory cannot be easily removed from the package as the package would be destroyed by opening it, and the process of detaching the stacked memory would likely irreparably damage the processor as well. Therefore, the entire package (including the expensive processor silicon) would have to be replaced.

To summarize, die-stacked DRAM RAS must provide robust detection and correction for all existing DRAM failure modes, should be robust enough to handle potential new failure modes as well, and will likely need to endure higher failure rates due to the reduced serviceability of 3D-integrated packaging.

2.5 Die-Stacked DRAM Caches

Caches store two types of information: tags and data. In conventional SRAM-based caches, these are stored in two physically distinct structures (the tag and data arrays, respectively). For a DRAM cache, one could consider an SRAM-based tag array, as shown in Figure 3(a), but previous estimates have shown that such a structure would require tens of megabytes of SRAM, and therefore this approach is not practical considering that current L3 cache sizes are typically only around 8 MB [11, 12].

Instead, recent research has considered organizations where the tags and data are directly co-located within the die-stacked DRAM, as shown in Figure 3(b) [15, 51]. While this eliminates the unwieldy SRAM tag array, it introduces two more problems. First, naively accessing the cache could double the DRAM cache latency: one access to read the tags, followed by another access to read the data (on a hit). Second, for cache misses, the cost of the DRAM cache tag access is added to the overall load-to-use latency.

Loh and Hill observed that the tags and data reside in the same DRAM row, and so the actual latency of a cache hit can be less than two full accesses by exploiting row buffer locality [51]. That is, the DRAM row is opened or activated into the row buffer only

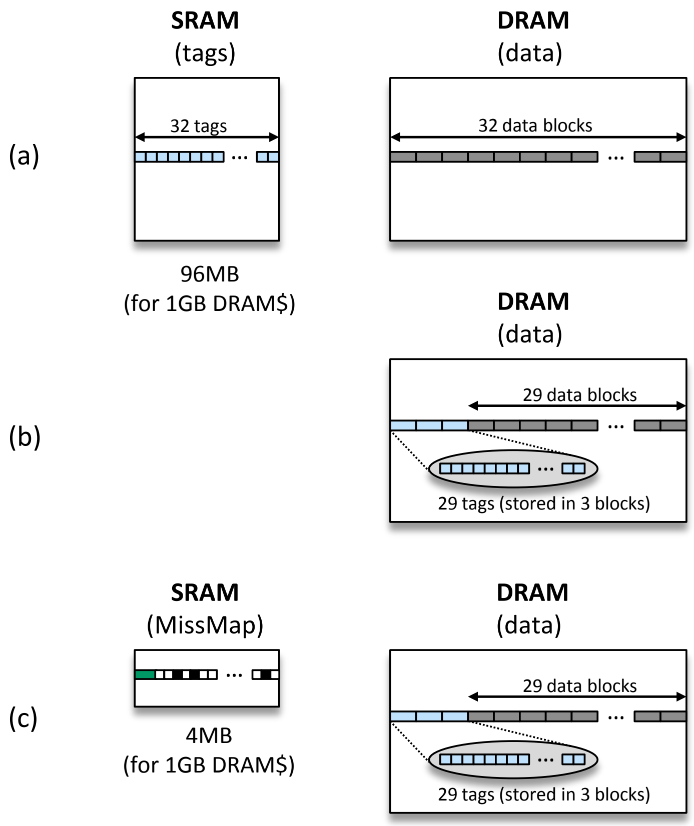


Figure 3: DRAM cache organizations using (a) an SRAM tag array, (b) tags embedded in the DRAM, and (c) tags in DRAM with a MissMap.

once, and then tag and data requests can be served directly out of the row buffer at a lower latency compared to two back-to-back accesses to different rows. They also proposed a hardware data structure called a MissMap that precisely tracks the contents of the DRAM cache, as shown in Figure 3(c). Before accessing the DRAM cache, the MissMap is first consulted to determine whether the requested cache block is even resident in the cache. If the block is not in the cache (i.e., miss), the request can be sent directly to main memory without incurring the tag-check cost of the DRAM cache. For a 512MB DRAM cache, the MissMap needs to be about 2MB in size (which provides tracking of up to 640MB of data), and a 1GB cache would need a 4MB MissMap. While Loh and Hill argue that part of the L3 cache could be carved out to implement the MissMap, using the AMD Opteron^{TM} processor that consumes 1MB of its L3 to implement a “Probe Filter” as an example [11], it seems unlikely that designers would be willing to sacrifice *half* of their L3 to implement the MissMap.[1]

2.6 Related Work

2.6.1 Related Work on Heterogeneous Memory Systems

NUMA Systems: Non-uniform memory architectures have been used in most large-scale symmetric multiprocessors. In the CC-NUMA architecture, each processing node maintains a cache to reduce traffic to remote nodes [45]. Cache-only memory architectures (COMA) [24] use memory as a hardware-managed cache, like the ALLCACHE design in the KSR1 system [20]. An S-COMA system allocates part of the local nodes main memory to act as a large cache for remote pages [67]. Falsafi and Wood propose Reactive NUMA (R-NUMA) that reacts to program behavior and enables each node to use the best of CC-NUMA or S-COMA for a particular page [19]. The Sun WildFire prototype shows that an R-NUMA-based design can significantly outperform a NUMA system [23]. Although the heterogeneous memory system analyzed in Chapter V has properties similar to a NUMA system (with variable memory latencies), our work differs from traditional NUMA research since we treat both the fast and slow memory as local to a node.

DRAM Caches: Recent papers on heterogeneous memory systems have proposed the use of fast stacked memory as a cache for the slow memory [36, 51]. However, the previous designs had shortcomings when being deployed in commercial products due to hardware overhead. In contrast, the proposed approach in Chapter III greatly reduces hardware overhead and enables using gigabytes of die-stacked DRAM as a cache in a practical way.

In Chapter V, we present a Part-of-Memory (PoM) architecture. The key difference between PoM and all previous work on DRAM caches (including the work in Chapter III) is that PoM provides higher total memory capacity as compared to DRAM caches. The

[1]Assuming a 4MB MissMap to support a 1GB DRAM cache and a baseline L3 cache size of 8MB. If such a system employed a Probe Filter as well, then only 3MB out of the original 8MB L3 would actually be available as a cache.

caching approach benefits from simple, fast allocation to fast memory (DRAM cache) but suffers from wasted capacity due to data duplication. Enabling PoM to maximize memory capacity requires quite different design approaches from previously described DRAM cache architectures, including support for complex swapping operations (Section 5.3.3) and memory permutation (Section 5.3.4 for fast-swap) that results when different memory locations are swapped. The benefits of the additional memory capacity provided by PoM extend beyond the performance benefits harvested through reduced disk swapping, directly impacting one of the attributes consumers use when making a purchase decision.

Software-Managed Heterogeneous Memory: Prior work has explored managing heterogeneous memory systems using software, as opposed to our hardware-based proposal. Loh et al. [52] studied the benefits and challenges of managing a die-stacked, heterogeneous memory system under software control. The authors discussed that even OS-managed heterogeneous memory systems require non-negligible hardware/software overheads for page activity monitoring and migration. Ekman and Stenstrom [16] also discussed a two-level main memory organization in the server systems under OS control. As we discuss in Chapter V, all these techniques suffer from OS-related overheads and are less responsive to program phase changes.

Hardware-Managed Heterogeneous Memory: Some recent papers have explored hardware implementations for supporting page migrations in heterogeneous memories [15, 63]. The work that is most closely related to the work in Chapter V is the one from Dong et al. [15]. Their hardware-only implementation maintains a translation table in the memory controller, which keeps track of all the page remappings. To keep the table size small, their implementation uses large 4MB pages, which incurs both high migration latencies and increased bandwidth pressure on the slow memory. In comparison, our approach supports small page sizes by keeping the remapping table in the fast memory and caching the recent remapping table accesses in a small remapping cache. Ramos et al. [63] propose hardware support for OS-managed heterogeneous memories. In their work, the page table keeps a master copy of all the address translations, and a small remapping table in the memory controller is used to buffer only the recent remappings. Once the remapping table becomes full, the buffered remappings need to be propagated to the page table, requiring the OS to update the page table and flush all the TLBs. Thus, their approach requires costly OS interventions, which our technique avoids by maintaining page remappings in a dedicated hardware-managed remapping table in the fast memory. In Chapter V, we detail how to practically combine and manage fast die-stacked and slow off-chip memory through hardware.

2.6.2 Related Work on RAS Support

Chapter IV is the first work to target increased RAS capabilities in die-stacked DRAM designs. However, several recent works target more power-efficient external DRAM implementations, and bear some similarities to the techniques described in this work. Single-subarray access (SSA) proposes fetching an entire cache line from a single DRAM device, using a RAID scheme to support chipkill [83]. LOT-ECC and virtualized ECC propose supporting chipkill with x8 DRAM devices by separating the error-detection and error-correction steps and storing some ECC information in data memory [82, 89]. The tiered coverage of memory-mapped ECC [88] and LOT-ECC share some similarities with the work in Section 4.4, but the structures and implementations are quite different.

Mini-rank proposes storing ECC bits along with the associated data bits to efficiently support RAS for their mini-rank design [92]. The embedded ECC complicates memory address translation (this technique is proposed for main memory), but the approach taken in our work does not affect the address translation as we store ECC information in the tag storage and thus do not change the data block size. Abella et al. propose adding hard-wired ROM entries to each row of an SRAM to provide a hard-wired row index when reading out data to detect row-decoder errors [1], but such an approach does not extend easily to commodity DRAMs. All of these techniques would require significant modifications to support die-stacked DRAM architectures.

The idea of replicating cache blocks has been studied before, although in a different context. In-cache replication (ICR) improves the resiliency of level-one data (DL1) caches [90]. ICR differs from our proposed Duplicate-on-Write (DOW) scheme (Section 4.4.2) in which the structures, constraints, and performance implications of the DL1 force a very different design from DOW; ICR makes use of dead-line predictors, more complex indexing of duplicate copies (possibly with multiple locations), and in some variants ICR duplicates both clean and dirty data.

There has also been a significant amount of recent research in using die-stacked DRAM as large caches [5, 15, 36, 51, 62, 75], but none of these have addressed the problems of providing robust error correction and detection capabilities for such large, in-package storage arrays. Micron's Hybrid Memory Cube provides ECC support for its 3D DRAM stacks [60], but this requires custom DRAM chips, whereas the proposed approaches in Chapter V are compatible with commodity non-ECC DRAM.

CHAPTER III

AN EFFICIENT DIE-STACKED DRAM CACHE

As described in Section 2.5, while the MissMap provides a more practical approach than using a massive SRAM tag array, its implementation is prohibitively costly to allow it to be deployed in commercial products. Furthermore, the access latency of the MissMap is not trivial (the original work used a latency of 24 cycles, which is added to *all* DRAM cache hits *and* misses). In this chapter, we propose a practical DRAM cache organization while eliminating the inefficiencies of the previous DRAM cache design.

3.1 The Problems

In this section, we identify inefficiencies with the previously proposed DRAM cache organizations (Section 2.5). First, we explain why the MissMap is overly conservative, which ultimately leads us to consider more speculative techniques with significantly lower overheads (both in terms of hardware cost and latency). Second, we describe scenarios where a conventional cache organization under-utilizes the available aggregate system bandwidth, which motivates our proposal for a *Self-Balancing Dispatch* mechanism. Third, we discuss how the presence of dirty/modified data in the DRAM cache can potentially limit how aggressively we can speculate on or rebalance DRAM cache requests.

3.1.1 The Overkill of the MissMap

The MissMap precisely maintains a bit-vector that records which cache blocks are currently resident in the DRAM cache. However, it is possible to allow the MissMap to have false positives. That is, if the MissMap says that a block is present in the DRAM cache when in fact it is not, then there is only a performance impact as the system needlessly pays for the latency of the DRAM cache before going to main memory. However, if the MissMap reports that a line is *not* present when in fact it is (false negative), the request would be sent to main memory and returned to the processor. If the DRAM cache contains this block in a dirty state, then this can lead to incorrect program execution.

On a DRAM cache miss (whether the MissMap said so or not), the system sends the request to main memory. When the response returns, the data are sent back to the L3 and the processor, and the data are also installed into the DRAM cache.[1] Prior to the installation of a new cache block, a victim must be selected. Furthermore, if the victim has been modified, then it must also be written back to main memory.

Note that when selecting a victim, the DRAM tags are checked. Therefore, if the system issued a request to main memory even though a modified copy of the block is in the DRAM cache, this can still be detected at the time of victim selection. Given this observation, the constraint that the MissMap *must* not allow false negatives is overly conservative. False negatives are tolerable so long as responses from memory are not sent back to the processor before having verified that a dirty copy does not exist in the DRAM cache.

Based on these observations, we propose a DRAM cache organization that can speculatively issue requests directly to main memory regardless of whether the decisions are "correct" or not. Section 3.2 describes a predictor design that exploits spatial correlation and the bursty nature of cache traffic to provide a light-weight yet highly accurate DRAM cache hit-miss predictor.

3.1.2 Under-Utilization of Aggregate System Bandwidth

Die-stacked DRAM can potentially provide a substantial increase in memory bandwidth. Previous studies have assumed improvements in latency of 2×, 3× and as much as 4× compared to conventional off-chip DRAM [36, 51, 91]. At the same time, the clock speed can be faster, bus widths wider, and independent channels more numerous [43, 51]. Even with a rough estimate of half the latency, twice the channels, and double-width buses

[1]For this study, we assume that all misses are installed into the DRAM cache. Other policies are possible (e.g., write-no-allocate, victim-caching organizations), but these are not considered here.

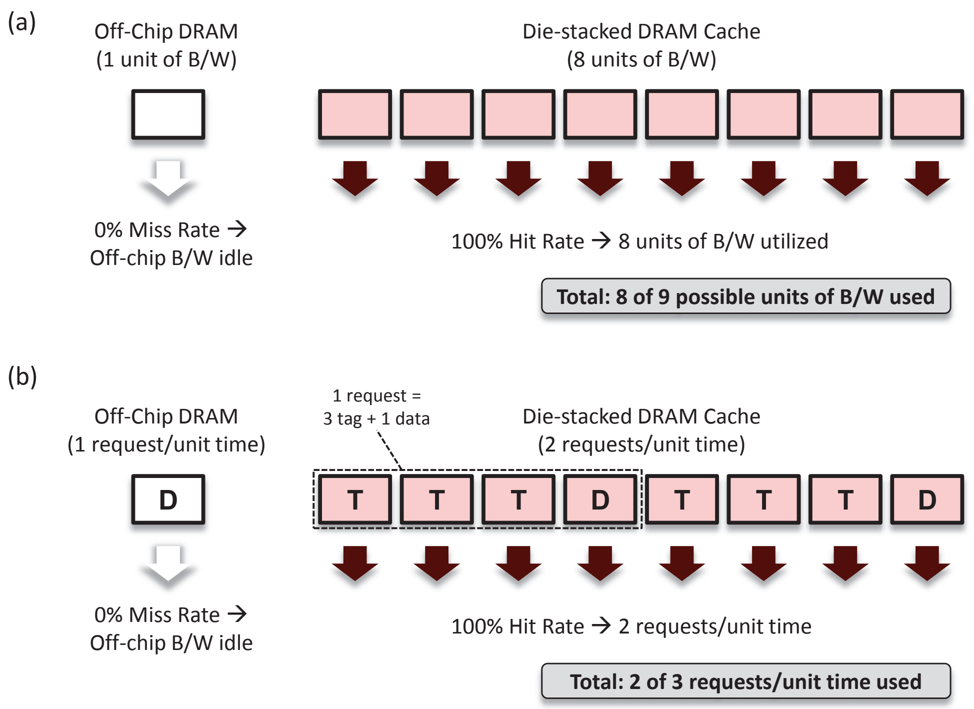


Figure 4: Example scenario illustrating under-utilized off-chip memory bandwidth in the presence of very high DRAM cache hit rates when considering (a) raw bandwidth in Gbps, and (b) in terms of request service bandwidth.

(compared to conventional off-chip memory interfaces), the stacked DRAM would provide an 8× improvement in bandwidth. In an "ideal" case of a DRAM cache with a 100% hit rate, the memory system could provide an eight-fold increase in delivered bandwidth, as shown in Figure 4(a). However, the off-chip memory is completely idle in this scenario, and that represents 11% ($\frac{1}{1+8}$) of the overall system bandwidth that is being wasted.

Figure 4(b) shows the same scenario again, but instead of raw bandwidth (in terms of Gbps), we show the *effective bandwidth* in terms of requests serviced per unit time. Note that a request to main memory only requires transferring a single 64B cache block, whereas a request to a tags-in-DRAM cache requires transferring three tag blocks (64B each) and finally the data block. Therefore, the sustainable effective bandwidth of the DRAM cache is only twice that of the off-chip memory (8× the raw bandwidth, but 4× the bandwidth-consumption per request). In this case, a 100%-hit rate DRAM cache would leave 33%

of the overall effective bandwidth unused ($\frac{1}{1+2}$). While the DRAM cache typically does not provide a 100% hit rate, hits often come in bursts that can lead to substantial queuing delays from bank and bus contention.

Apart from the available bandwidth, bank and bus conflicts at the DRAM cache can lead to increased queuing delays, some of which could potentially be mitigated if some of these requests could be diverted to the off-chip memory. In practice, other timing constraints, resource conflicts, and specific access patterns and arrival rates would affect the exact amount of bandwidth available for both the DRAM cache and the off-chip memory. However, this simple example highlights that there will be times where the system will have some idle resources, and we propose a Self-Balancing Dispatch technique to capitalize on these resources.

3.1.3 Obstacles Imposed by Dirty Data

Dirty data in the DRAM cache can severely restrict the aggressiveness of speculatively sending requests to main memory, as the copy in main memory is stale and its usage can result in incorrect executions. Likewise, dirty data prevents the system from exploiting idle main-memory bandwidth because accesses to dirty data must be sent to the DRAM cache regardless of how busy the DRAM cache or how idle the off-chip memory is. This also raises the question as to how the system can know ahead of time that a request targets a dirty cache line without having first looked up in the cache to see if the line is present and dirty. A key contribution of this work is a new way to operate the DRAM cache (which could be applied to other types of caches) such that most of the cache will be clean, and for the majority of the cache, we can guarantee its cleanliness without having to check the cache's tags. This removes major limitations for both cache hit speculation and Self-Balancing Dispatch.

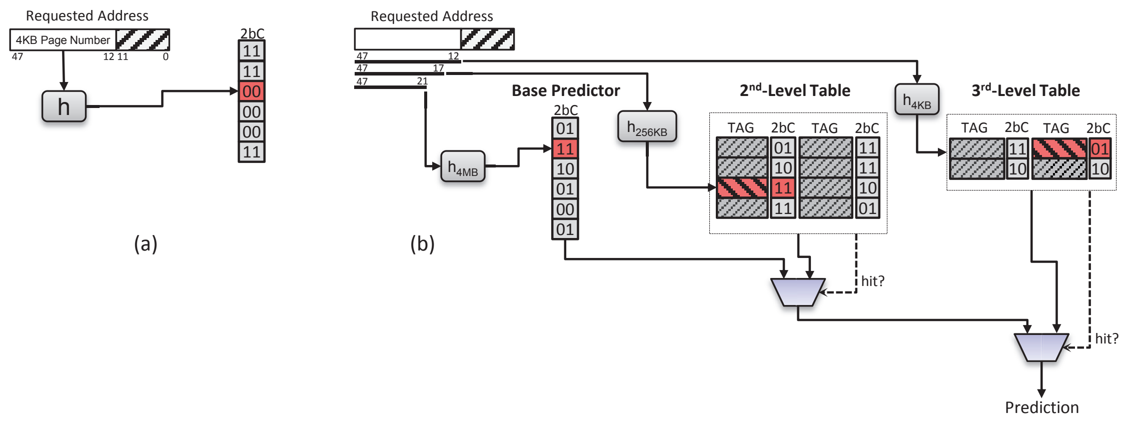


Figure 5: Hit-Miss Predictor designs: (a) one-level $\text{HMP}_{\text{region}}$, (b) multi-granular HMP_{MG}.

3.2 DRAM Cache Hit Speculation

The previously proposed MissMap provides precise tracking of DRAM cache contents, but as a result, the size (2-4MB) and latency (tens of cycles) of the structure introduce significant overheads. Section 3.1 explained how the DRAM cache can check for the existence of a dirty block at the time of a cache fill, and how this allows the DRAM cache to speculatively send requests to main memory so long as we ensure that the data are not returned to the processor until it has been verified that a modified copy is not also in the DRAM cache. This section presents the designs for lightweight and accurate region-based predictors that exploit the bursty nature of cache hits and misses [49].

3.2.1 Region-Based Hit-Miss Prediction

Our region-based Hit-Miss Predictor ($\text{HMP}_{\text{region}}$) is structurally similar to a classic bimodal branch predictor [78]. The predictor itself consists of a table of two-bit saturating counters. For a DRAM cache with millions of cache blocks, it is not practical to directly index into the $\text{HMP}_{\text{region}}$ table with a hash of the raw physical address; the aliasing and interference would render the predictor table nearly useless, or a gigantic table would be needed. Instead, we break up the memory space into coarser-grained regions (e.g., 4KB), and index into the $\text{HMP}_{\text{region}}$ with a hash of the region's base address as shown in Figure 5(a). This allows the $\text{HMP}_{\text{region}}$ table to be much smaller, but it also means that *all* accesses within a region will

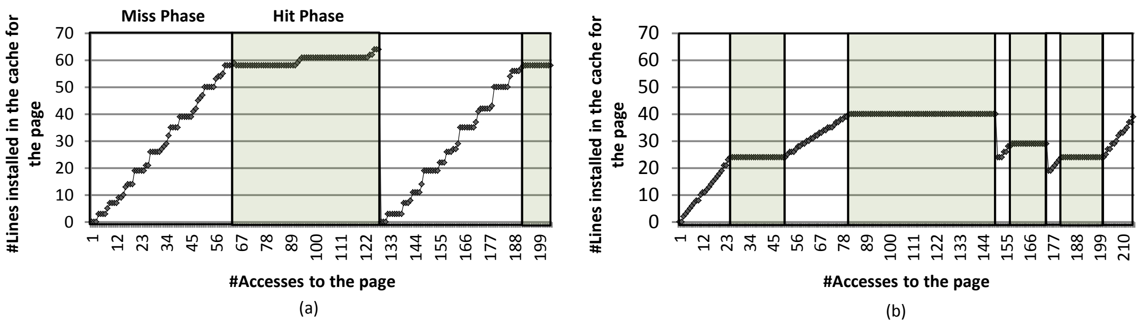


Figure 6: Hit and miss phases for two example pages from leslie3d (when run as part of the multi-programmed workload WL-6).

follow the same prediction. The operation of the HMP$_{\text{region}}$ is otherwise analogous to the bimodal predictor: DRAM cache hits increment the predictor, and misses decrement the predictor (saturating at 3 or 0, respectively).

The coarse-grained predictor organization of HMP$_{\text{region}}$ is actually a benefit rather than a shortcoming. Accesses tend to exhibit significant spatial locality, especially at lower-level caches such as a large DRAM cache. Figure 6(a) shows the number of cache blocks present in the DRAM cache for one particular 4KB page of leslie3d (from the WL-6 workload) with respect to the number of accesses to this page (our methodology is explained in Section 3.5). Initially, nothing from this page is in the DRAM cache, but as the page is used, more and more lines are installed. During this installation phase, most accesses result in cache misses, and a simple 2-bit counter corresponding to this region would mostly predict that these requests result in misses. After this "warm up" phase, the footprint from this region is stable, and all subsequent accesses to this region result in hits. Again, a simple 2-bit counter would quickly switch over to predicting "cache hit" for this region and achieve high accuracy. When the application is finished with using this region, the contents will gradually get evicted from the cache, as shown by the drop back down to zero. At some future point, the page becomes hot again and the process repeats. Note that the figure's x-axis is based on accesses to the page. The time that elapses from the last access in the hit phase until the first access in the miss phase could easily span many thousands or even millions of cycles, but this all gets compressed here.

Figure 6(b) shows another 4KB region taken from leslie3d (from the same workload WL-6). This is just to illustrate that different regions and different applications may show different types of patterns, but so long as there exist sustained intervals where the curve is consistently increasing (mostly misses) or is consistently flat (mostly hits), then the simple 2-bit counter approach will be effective for making hit-miss predictions for the region.

The $\text{HMP}_{\text{region}}$ approach is different from other previously proposed history-based hit-miss predictors. Past work has considered hit-miss predictors for L1 caches based on PC-indexed predictor organizations [87]; such an approach may not be as easy as to implement for a DRAM cache because PC information is not typically passed down the cache hierarchy, may not exist for some types of requests (e.g., those originating from a hardware prefetcher), or may not be well defined (e.g., a dirty cache block being written back to the DRAM cache that was modified by two or more store instructions may have multiple PCs associated with it).

3.2.2 Multi-Granular Hit-Miss Predictor

The $\text{HMP}_{\text{region}}$ predictor requires approximately one two-bit counter per region. For example, assuming a system with 8GB of physical memory and a region size of 4KB, the $\text{HMP}_{\text{region}}$ would still need 2^{21} two-bit counters for a total cost of 512KB of storage. While this is already less than a 2-4MB MissMap, there is still room to further optimize.

We observed that even across large contiguous regions of memory spanning multiple physical pages, the hit-miss patterns generally remained fairly stable; that is, sub-regions often have the same hit-miss bias as other nearby sub-regions. While, in theory, different nearby physical pages may have nothing to do with each other (e.g., they may be allocated to completely independent applications), in practice memory allocation techniques such as page-coloring [81] tend to increase the spatial correlation across nearby physical pages. In our experiments, we found that memory would often contain large regions with *mostly* homogeneous hit/miss behavior, but smaller pockets within the larger regions would behave differently.

We propose a *Multi-Granular Hit-Miss Predictor* (HMP_{MG}) that is structurally inspired by the TAGE branch predictor [70], but operates on the base addresses of different memory regions (as opposed to branch addresses) and the different tables capture hit-miss patterns corresponding to different region sizes (as opposed to branch history lengths). Figure 5(b) shows the hardware organization of HMP_{MG}. The first-level predictor is similar to $\text{HMP}_{\text{region}}$, except that it makes predictions over very large 4MB regions. The second and third-level tables consist of set-associative tagged-structures that make predictions on finer-grained 256KB and 4KB region sizes, respectively. Each entry in the tagged tables consists of a (partial) tag and a two-bit counter for prediction. Tag hits in the tagged HMP_{MG} tables will override predictions from larger-granularity predictor tables.

The overall structure of the HMP_{MG} provides a more efficient and compact predictor organization. A single two-bit counter in the first-level table covers a memory range of 4MB. In the single-level $\text{HMP}_{\text{region}}$ predictor, this would require 1024 counters to cover the same amount of memory.

3.2.3 Predictor Operation

The entries in the HMP_{MG}'s base predictor are initially set to "weakly miss" or 1. To make a hit/miss prediction for a request, the base and tagged components are all looked up simultaneously. The base component makes a default prediction, while tagged components provide an overriding prediction on a tag hit.

The HMP_{MG} is updated when it has been determined whether there was a DRAM cache hit or not. The 2-bit counter of the "provider" component is always updated.[2] On a misprediction, an entry from the "next" table is allocated (a victim is chosen based on LRU). For example, if the prediction came from the first-level table, then a new entry will

[2]The "provider" is the terminology used for the TAGE predictor to indicate the table from which the final prediction came from.

be allocated in the second-level table. Mispredictions provided by the third table simply result in the corresponding counter being updated without any other allocations. The newly allocated entry's 2-bit counter is initialized to the weak state corresponding to the actual outcome (e.g., if there was a DRAM cache hit, then the counter is set to "weakly hit" or 2).

3.2.4 Implementation Cost

Table 1 shows the storage overhead of the HMP_{MG} configuration used in this work. Compared to a MissMap that requires 2-4MB of storage, the HMP_{MG} only requires 624 bytes of total storage. A single predictor is shared among all cores. At this size, the entire L3 cache can once again be used for caching (as opposed to implementing a MissMap). Also importantly, the small size of the HMP_{MG} allows it to be accessed in a single cycle as it is smaller than even many branch predictors. Compared to the 24-cycle latency assumed for the MissMap [51], this provides significant benefits both for performance and implementability.

Table 1: Hardware cost of the Multi-Granular Hit-Miss Predictor.

Hardware	**Size**
Base Predictor (4MB region)	1024 entries × 2-bit counter = 256B
2nd-level Table (256KB region)	32 sets × 4-way × (2-bit LRU + 9-bit tag + 2-bit counter) = 208B
3rd-level Table (4KB region)	16 sets × 4-way × (2-bit LRU + 16-bit tag + 2-bit counter) = 160B
Total	**624B**

3.3 Exploiting Unused Bandwidth

As described in Section 3.1, there are scenarios where a burst of DRAM cache hits (or predicted hits for that matter) can induce significant DRAM cache bank contention while the off-chip memory remains largely idle. In this section, we describe a *Self-Balancing Dispatch* (SBD) mechanism that allows the system to dynamically choose whether (some) requests should be serviced by the DRAM cache or by the off-chip memory.

In an ideal case, every request could be routed to either the DRAM cache or to off-chip memory. If both memories had the same latency per access, then the system could simply look at the number of requests already enqueued for each and send the request to the one with fewer requests. However, the different memories have different latencies, and so the request should be routed to the source that has the lowest *expected* latency or queuing time. The expected latency for each memory can be estimated by taking the number of requests already "in line" and then multiplying by the average or typical access latency for a single request at that memory. Overall, if one memory source is being under-utilized, then it will tend to have a lower expected latency and the SBD mechanism will start directing requests to this resource. In the steady-state, the bandwidth from both sources will be effectively put to use.

Complications arise due to the fact that not every request can or should be freely routed to whichever memory has the lowest expected latency. If a request is for a dirty block in the DRAM cache, then routing the request to the off-chip memory is of no use (in fact, it just wastes bandwidth) because the data must ultimately come from the DRAM cache. If the HMP predicts that a request will miss in the DRAM cache, then there is likely little benefit in routing it to the DRAM cache (even if it has a lower expected latency), because if the prediction is correct, there will be a cache miss which in the end simply adds more latency to the request's overall service time.

The above constraints mean that SBD can only be gainfully employed for requests that would have hit in the DRAM cache where the corresponding cache block is not dirty. To determine whether a request will (likely) hit in the DRAM cache, we simply rely on the HMP. While the HMP is not perfectly accurate, mispredictions simply result in lost opportunities for SBD to make better use of the available bandwidth. To deal with dirty data, we will first simply assume that the DRAM cache makes use of a write-through policy to ensure that all blocks are always clean. Algorithm 1 below describes the basic SBD algorithm assuming a write-through cache. In the next section, we will show how to remove the strict write-through requirement to avoid the unnecessary write traffic to main memory.

Algorithm 1 Self-Balancing Dispatch

```
0) Self-balancing dispatch operates only on (predicted) hit requests.
1) $N_{\text{Off-Chip}}$ := Number of requests already waiting for the same bank in the off-chip memory.
2) $L_{\text{Off-Chip}}$ := Typical latency of one off-chip memory request, excluding queuing delays.
3) $E_{\text{Off-Chip}} := N_{\text{Off-Chip}} \times L_{\text{Off-Chip}}$. (Total expected queuing delay if this request went off-chip.)
4-6) $N_{\text{DRAM\_Cache}}$, $L_{\text{DRAM\_Cache}}$, $E_{\text{DRAM\_Cache}}$ are similarly defined, but for the die-stacked DRAM cache.
7) If $E_{\text{Off-Chip}} < E_{\text{DRAM\_Cache}}$, then send the request to off-chip memory; else send to DRAM cache.
```

Note in Algorithm 1, we do not count *all* of the requests that are waiting to access off-chip memory, but we limit the count to those waiting on the same bank as the current request that is under SBD consideration (similar for the number of requests to the target off-chip DRAM cache bank). The above description uses the "typical" latency (e.g., for main memory we assume the latency for a row activation, a read delay (tCAS), the data transfer, and off-chip interconnect overheads; for the DRAM cache we assume a row activation, a read delay, three tag transfers, another read delay, and then the final data transfer). Other values could be used, such as dynamically monitoring the actual average latency of requests, but we found that simple constant weights worked well enough. Note also that these latency estimates only need to be close enough relative to each other; slight differences in the estimated expected latency and the actual observed latency do not matter if they do not lead to different SBD outcomes (i.e., an error of a few cycles will in most cases not cause the SBD mechanism to change its decision).

3.4 Maintaining a Mostly-Clean Cache

When a request is for a cached dirty block, the SBD mechanism has no choice but to send the request to the DRAM cache (it is possible that the HMP mispredicted it as a miss, but this would ultimately be detected and requires reading the data from the DRAM cache

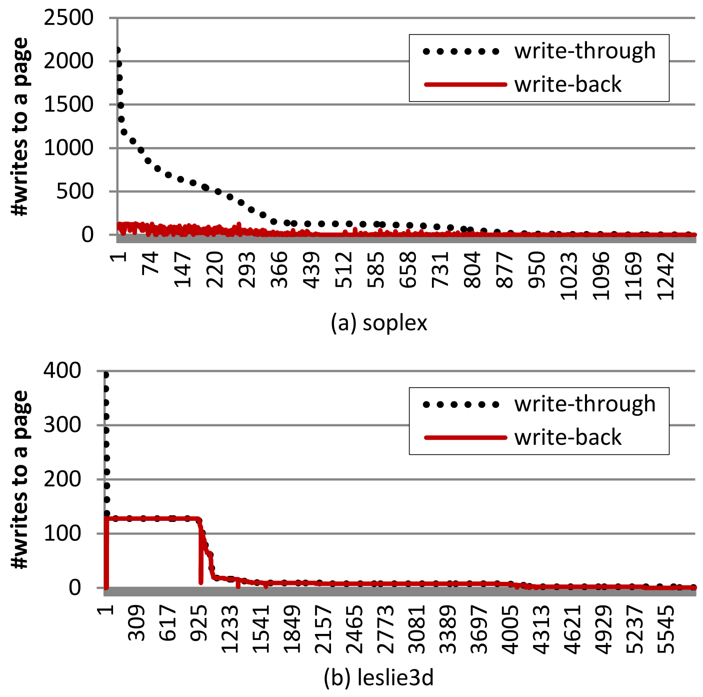


Figure 7: Number of writes for each page with write-through and write-back policy. The x-axis is sorted by top most-written-to pages.

anyway). If the system could *guarantee* that a requested block is not cached *and* dirty, then SBD could more freely make bandwidth-balancing decisions with its effectiveness only constrained by the accuracy of the HMP.

3.4.1 Write-Through vs. Write-Back

We earlier discussed how employing a write-through policy for the DRAM cache can in fact ensure that all requests that hit in the cache are *not* for dirty blocks, but applying a write-through policy wholesale to the entire DRAM cache can result in significant increases in write-through traffic to main memory. Figure 7(a) shows the top most-written-to pages in the DRAM cache for the SPEC2006 benchmark soplex. The upper curve (dotted) shows the write traffic for a write-through policy, and the lower curve (solid) shows the write traffic for a write-back policy. The large differences between the curves indicate that the write-back policy achieves significant write-combining, and therefore employing a write-through policy could significantly increase write traffic to main memory. There are other scenarios, such as that shown in 7(b), where, even in a write-back cache, dirty lines are usually only written to once before they are subsequently evicted. However, on average across all of our workloads, we observed that a write-through DRAM cache results in $\sim$3.7$\times$ greater write traffic to main memory than a write-back policy (although the amount varies significantly based on the exact workloads).

Another important statistic is that, on average for our experiments, only about 5% of an application's pages ever get written to. This indicates that in typical scenarios, the vast majority of the DRAM cache's blocks are in fact clean. A write-through cache ensures cleanliness, but costs significantly more main-memory write traffic. A write-back cache minimizes off-chip write traffic, but then cannot provide any guarantees of cleanliness despite the fact that most blocks will in fact be clean.

3.4.2 The Dirty Region Tracker

We propose a hybrid write-policy for the DRAM cache where, by default, pages employ a write-through policy (to handle the common case of clean pages), but a limited subset of the most write-intensive pages are operated in write-back mode to keep the main-memory write traffic under control. To support this hybrid policy, we introduce the *Dirty Region Tracker* (DiRT). The DiRT consists of two primary components as shown in Figure 8. The first structure is a counting Bloom filter (CBF) that is used to approximately track the number of writes to different pages. On each write, the page address is hashed differently for each of the CBF tables, and the corresponding counters are incremented. We use three CBFs with different hash functions, which increases the efficacy of identifying the most write-intensive pages due to the reduction in aliasing. When a page's counters in *all three* CBFs exceed a threshold, then it is determined to be a write-intensive page (and each indexed CBF counter is reduced by half).

At this point, we introduce the second structure that is a *Dirty List* of all pages that are currently operated with a write-back policy. The Dirty List is a set-associative tagged

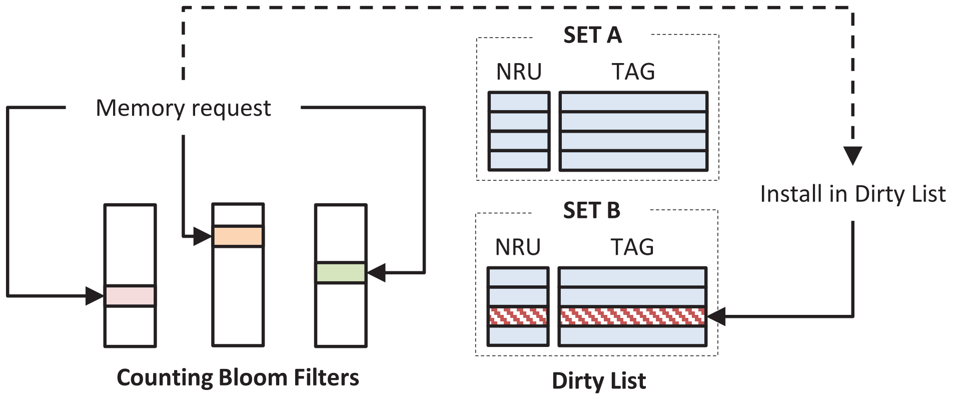


Figure 8: Dirty Region Tracker (DiRT).

structure where each entry consists of a tag to store a physical page number and 1 bit of storage to implement a not-recently-used (NRU) replacement policy. A page not currently in the Dirty List, but whose counters have exceeded the threshold, gets inserted into the Dirty List (and the NRU entry from the Dirty List is evicted). Note that when a page is evicted from the Dirty List, its write policy is switched back to write-through; at this point, the system must ensure that any remaining dirty blocks from this page are written back to main memory. At first blush, this may seem like a high overhead, but a 4KB page only contains 64 cache blocks. Current die-stacked DRAMs already support 32 banks (e.g., 4 channels at 8 banks each [43]), and so the latency overhead is only two activations per bank (and the activations across banks can be parallelized) plus the time to stream out the data back to main memory. Note that all of these cache blocks will have very high spatial locality because they are all from the same page, so practically all of the writeback traffic will experience row buffer hits in main memory. Also, any clean blocks of course need not be written back. The detailed algorithm for DiRT management is listed in Algorithm 2.

3.4.3 Putting the DiRT to Work

3.4.3.1 Streamlining HMP

The DiRT works synergistically with Hit-Miss Prediction. In parallel with the HMP lookup, a request can also check the DiRT to see if it accesses a guaranteed clean page. If the page

Algorithm 2 DiRT Management

```
**1)** Check Dirty List for the written-to page; if it's there, update NRU replacement meta-
data.
**2)** If not, increment each indexed counter in all three CBFs.
**3)** If all indexed counters are greater than threshold:
  **a)** Evict the NRU entry from the Dirty List;
      writeback any associated dirty blocks.
  **b)** Allocate the new page to the Dirty List.
  **c)** Reduce each indexed CBF counter by half.
```

is clean (i.e., not currently in the Dirty List), then requests that are predicted misses can be issued directly to main memory. When the value is returned, this data can be forwarded directly back to the processor without having to verify whether there was actually a dirty copy of the block in the DRAM cache because the DiRT has already guaranteed the block to be clean. Without the DiRT, all returned predicted-miss requests must stall at the DRAM cache controller until the fill-time speculation has been verified. During times of high bank contention, this prediction-verification latency can be quite substantial.

3.4.3.2 Streamlining SBD

When combining the DiRT with the SBD mechanism, the DiRT can guarantee that accesses to certain (most) pages will be clean, and so SBD can freely choose the best memory source to route the request to. When the HMP predicts a hit, the system first consults the DiRT's Dirty List. If the requested page is found in the Dirty List, then we do not know if the requested *block* is dirty or not (e.g., it could be one of the few clean blocks in a mostly-dirty page). In this case, SBD always routes the request to the DRAM cache. However, if the requested page is not in the Dirty List, then the page (and therefore the requested block) is *guaranteed* to be clean, and therefore SBD can do as it wishes. Note that clean pages are the overwhelming common case (expect for a few benchmarks), and so using the DiRT provides SBD with many more opportunities to make use of otherwise under-utilized off-chip bandwidth.

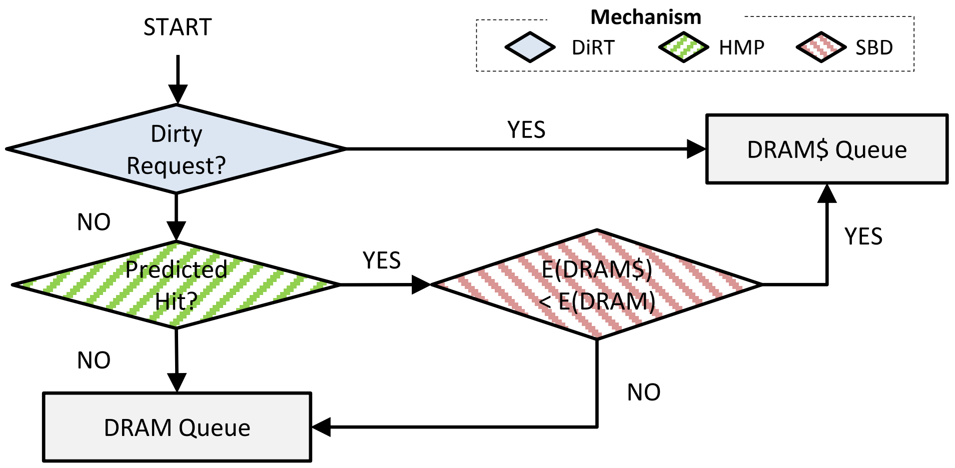


Figure 9: Decision flow chart for memory requests.

3.4.4 Putting It All Together

Figure 9 shows the decision flow chart for memory requests with all of the proposed mechanisms. One should note that Hit-Miss Prediction, SBD, and the DiRT can all be accessed in parallel (SBD can speculatively make a decision assuming an access to a clean, predicted-hit block). Furthermore, HMP and DiRT lookups could even be initiated early before the L2 hit/miss status is known as these components only require the requested physical address.

3.4.5 DiRT Implementation Cost

The DiRT is a slightly larger structure compared to the simple hit-miss predictors, but the overall hardware cost is still quite manageable (6.5KB, just 0.16% of our L2 data array size). Each of the three CBF tables has 1024 entries, and each entry consists of a five-bit saturating counter. We use a threshold of 16 writes to consider a page as write-intensive. For Dirty List, we use a 4-way set associative structure with 256 sets, so it supports up to 1024 pages operating in write-back mode at a time. Each entry of the Dirty List consists of 1-bit reference information for NRU replacement policy and a tag for the page. Other approximations (e.g., pseudo-LRU, SRRIP [33]) or even true LRU (this only requires 2-bit for a 4-way set associative structure) could also be used for the replacement policy, but a simple NRU policy worked well enough for our evaluations. In Section 3.6.5.4, we provide additional results while comparing our implementation with different DiRT organizations and management policies. For these estimates, we also conservatively assumed a 48-bit physical address (12 bits used for 4KB page offset), which increases our tag size. The total overheads are summarized in Table 2.

Table 2: Hardware cost of the Dirty-Region Tracker.

Hardware	**Size**
Counting Bloom Filters	3 $\times$ 1024 entries $\times$ 5-bit counter = 1920B
Dirty List	256 sets $\times$ 4-way $\times$ (1-bit NRU + 36-bit tag) = 4736B
Total	**6656B = 6.5KB**

3.5 Experimental Methodology

Simulation Infrastructure: We use MacSim [42], a cycle-level x86 simulator, for performance evaluations. We model a quad-core processor with two-level SRAM caches (private L1 and shared L2) and an L3 DRAM cache. The stacked DRAM is expected to support more channels, banks, and wider buses per channel [43]. In this study, the DRAM cache has four channels with 128-bit buses, and each channel has eight banks, while the conventional off-chip DRAM has 2 channels, each with 8 banks and a 64-bit bus. Also, key DDR3 timing parameters with bank conflicts and data bus contention are modeled in our DRAM timing module. Table 3 shows the system configurations used in this study.

Workloads: We use the SPEC CPU2006 benchmarks and sample 200M instructions using SimPoint [71]. Then we categorize the applications into two different groups based on the misses per kilo instructions (MPKI) in the L2 cache. We restrict the study to workloads with high memory traffic; applications with low memory demands have very little performance sensitivity to memory-system optimizations and therefore expose very little insight (we did

Table 3: System parameters used in this study.

CPU	
Core	4 cores, 3.2GHz out-of-order, 4 issue width, 256 ROB
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	16-way, shared 4MB (4 tiles, 24-cycle)
Stacked DRAM cache	
Cache size	128MB
Bus frequency	1.0GHz (DDR 2.0GHz), 128 bits per channel
Channels/Ranks/Banks	4/1/8, 2KB row buffer
tCAS-tRCD-tRP	8-8-15
tRAS-tRC	26-41
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11
tRAS-tRC	28-39

verify that our techniques do not negatively impact these benchmarks). Out of the memory-intensive benchmarks, those with average MPKI rates greater than 25 are in Group H (for **H**igh intensity), and of the remaining, those with 15 MPKI or more are in Group M (for **M**edium). Table 4 shows MPKI values of the benchmarks and their group.

Table 4: L2 misses per kilo instructions (L2 MPKI).

Group M	**MPKI**	**Group H**	**MPKI**
GemsFDTD	19.11	leslie3d	25.85
astar	19.85	libquantum	29.30
soplex	20.12	milc	33.17
wrf	20.29	lbm	36.22
bwaves	23.41	mcf	53.37

We select benchmarks to form rate-mode (all cores running separate instances of the same application) and multi-programmed workloads. Table 5 shows the primary workloads evaluated for this study. Section 3.6.1 also includes additional results covering a much larger number of workloads.

For each workload, we simulate 500 million cycles of execution. We verified that the DRAM cache is sufficiently warmed up: the DRAM cache access statistics at the end of

Table 5: Multi-programmed workloads.

Mix	Workloads	Group
WL-1	4 × mcf	4×H
WL-2	4 × lbm	4×H
WL-3	4 × leslie3d	4×H
WL-4	mcf-lbm-milc-libquantum	4×H
WL-5	mcf-lbm-libquantum-leslie3d	4×H
WL-6	libquantum-mcf-milc-leslie3d	4×H
WL-7	mcf-milc-wrf-soplex	2×H + 2×M
WL-8	milc-leslie3d-GemsFDTD-astar	2×H + 2×M
WL-9	libquantum-bwaves-wrf-astar	1×H + 3×M
WL-10	bwaves-wrf-soplex-GemsFDTD	4×M

the simulation show that the number of valid cache lines is equal to the total capacity of the cache, and the total number of evictions is 5-6× greater than the total cache capacity.

Performance Metric: We report performance using weighted speedup [18, 79], which is computed using Equation (1).

$$\text{Weighted Speedup} = \sum_{\mathrm{i}} \frac{\mathrm{IPC}_{\mathrm{i}}^{\mathrm{shared}}}{\mathrm{IPC}_{\mathrm{i}}^{\mathrm{single}}} \tag{1}$$

The geometric mean is also used to report average values.

3.6 Results and Analysis

3.6.1 Performance

Figure 10 shows the performance of the proposed hit-miss predictor (HMP), self-balancing dispatch (SBD), and dirty region tracker (DiRT) mechanisms for multi-programmed workloads. For comparison, we use a baseline where the DRAM cache is not employed. We also compare our mechanisms with the previously proposed MissMap structure (denoted as MM in the figure). We model a MissMap with zero storage overhead; i.e., no L2 cache capacity is sacrificed for the MissMap, but the L2 latency is still incurred for the lookup.

We first evaluate the impact of hit-miss prediction (HMP) without DiRT. In this usage scenario, every predicted miss request serviced from off-chip memory must wait to be

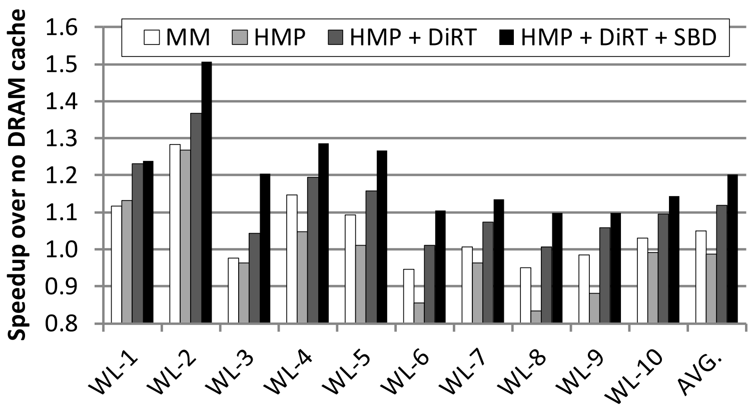


Figure 10: Performance normalized to no DRAM cache for MissMap (MM), and combinations of HMP, SBD, and DiRT.

verified as *safe* (i.e., no dirty data in the DRAM cache). As a result, for most benchmarks, HMP without DiRT performs worse than MissMap. This is not necessarily a negative result when one considers that the HMP approach sacrifices the multi-megabyte MissMap for a much smaller sub-kilobyte predictor. Achieving even close to similar performance while removing such a large hardware overhead is still a desirable result. However, with DiRT support, HMP+DiRT performs even better than MissMap due to the elimination of fill-time prediction verifications for clean blocks (which are the common case). At this point, the performance benefit *over* MissMap is primarily due to the replacement of the 24-cycle MissMap latency with a 1-cycle HMP lookup.

Next, we apply the SBD mechanism on top of HMP+DiRT. As shown in the results, SBD further improves performance (often significant, depending on the workload). Compared to HMP+DiRT, SBD provides an additional 8.3% performance benefit on average. In summary, the proposed mechanisms (HMP+DiRT+SBD) provide a 20.3% performance improvement over the baseline. Also, compared to MissMap, they deliver an additional 15.4% performance over the baseline. On a last remark, one should note that the evaluated MissMap does not sacrifice the L2 cache (i.e., ideal), so our mechanisms would perform even better when compared to a non-ideal MissMap that reduces the effective SRAM cache size.

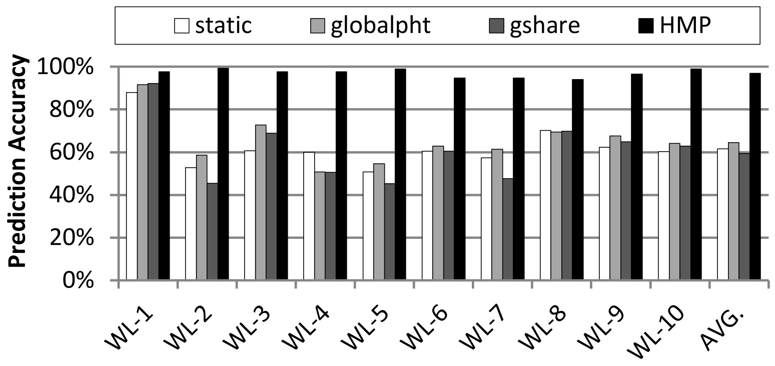


Figure 11: Prediction accuracy of HMP and its comparison with other types of predictors.

3.6.2 HMP: Prediction Accuracy

Figure 11 shows the prediction accuracy of the proposed predictor with comparison to some other types of predictors. `static` indicates the best of either static-hit or static-miss predictors, so the value is always great than 0.5. A reasonable predictor at least should be better than `static`. `globalpht` is the implementation of only one 2-bit counter for all memory requests, where it is incremented/decremented on a hit/miss. `gshare` is a gshare-like cache predictor (i.e., using the XOR of a requested 64B block address with a global history of recent hit/miss outcomes to index into a pattern history table).

First, the results show that our predictor provides more than 95% prediction accuracy on the evaluated workloads (average of 97%), which implies that the spatial locality-based hit/miss prediction is highly effective. Next, compared to `static`, we can see that the other predictors actually do not improve prediction accuracy much. For WL-1, all of the predictors perform well because the workload has a high hit rate and is simply easy to predict. But, if the hit ratio is around 50% as in other workloads, the other predictors perform poorly. For `globalpht`, one core may be consistently hitting while the other is consistently missing, and as a result the simple counter could ping-pong back and forth generating mispredictions. For `gshare`, the hit/miss history register provides poor information, and its inclusion often introduces more noise than useful correlations, resulting in overall lower prediction rates. In summary, HMP outperforms

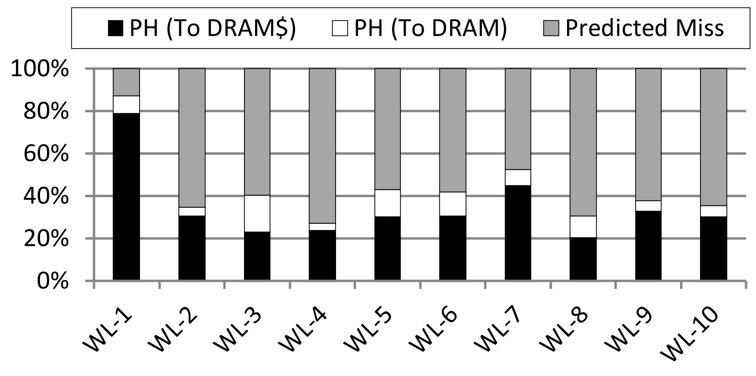


Figure 12: Issue direction breakdown. PH indicates predicted hit requests.

other predictors that use the individual 64B request address and/or history information of the actual outcomes.

3.6.3 SBD: Percentage of Balanced Hit Requests

Figure 12 shows the distribution of SBD's issue decisions (i.e., DRAM or DRAM cache). The black bar (PH: To DRAM$) represents the percentage of requests that are predicted hits and are actually issued to the DRAM cache, while the white bar (PH: To DRAM) represents the predicted-hit requests that were diverted to off-chip memory. Note that SBD does not work on the predicted-miss requests; thus, the requests in the (Predicted Miss) portion are always issued to off-chip memory.

At first thought, one might think that SBD does not operate on the benchmarks whose hit ratios are low (e.g., below 50%) because the amount of traffic to off-chip DRAM would be greater than that to the DRAM cache. Due to the bursty nature of memory requests, however, the instantaneous hit ratio and/or bandwidth requirements vary from the average values; thus, the balancing mechanism provided by SBD can still be beneficial even for the low hit-ratio workloads. In fact, as shown in the results, SBD was able to redistribute some of the hit requests for all of the benchmarks.

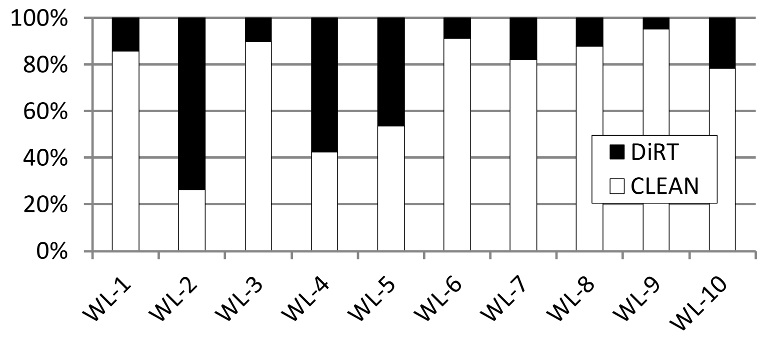


Figure 13: Percentage distribution of memory requests captured in DiRT.

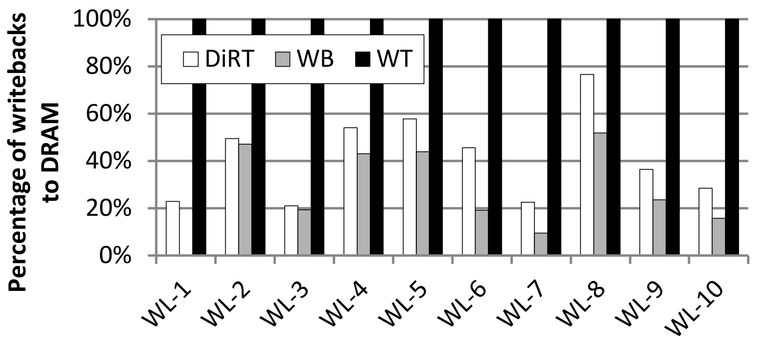


Figure 14: Write-back traffic to off-chip DRAM between write-through, write-back, and DiRT (WL-1 does not generate WB traffic), all normalized to the write-through case.

3.6.4 DiRT: Benefit and Traffic

Figure 13 shows the percentage distribution of write-through mode (CLEAN) and write-back mode (DiRT) memory requests. The "CLEAN" portion indicates the number of requests that are not found in the DiRT; thus, they are free to be predicted or self-balanced. The results show that DiRT allows a significant amount of memory requests to be handled without fear of returning a stale value. Note that without DiRT, every request that is a predicted miss (or a predicted hit but diverted to DRAM) needs to wait until it has been verified that the DRAM cache does not contain a dirty copy.

Figure 14 illustrates the amount of write-back traffic to off-chip DRAM for write-through, write-back, and the DiRT-enabled hybrid policy, all normalized to the write-through case. As shown in the results, a write-back policy performs a significant amount of

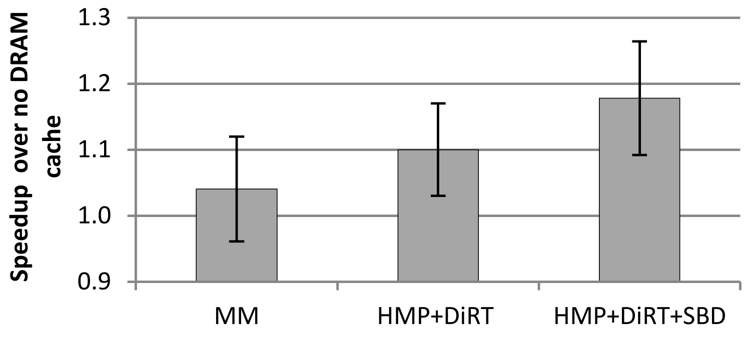


Figure 15: Average performance of MissMap and our proposed mechanisms over no DRAM cache baseline with +/-1 std. deviation for 210 workloads.

write-combining and thereby greatly reduces the amount of write traffic to main memory as compared to a write-through policy. DiRT is not perfect and it does increase write traffic slightly compared to a true write-back policy, but the total write traffic from a DiRT-enabled DRAM cache is much closer to that of the write-back case than it is to write-through. The relatively small increase in write traffic due to the DiRT is more than compensated by the streamlined HMP speculation and increased opportunities for SBD.

3.6.5 Sensitivity Results

3.6.5.1 Sensitivity to Different Workloads

To ensure that our mechanisms work for a broader set of scenarios beyond the ten primary workloads used thus far, we simulated all 210 combinations (${}_{10}C_4$) of the ten Group H and Group M benchmarks. Figure 15 shows the performance results averaged over all of the 210 workloads, along with error-bars to mark one standard deviation. As shown in the figure, our mechanisms combine to deliver strong performance over the previously proposed MissMap-based DRAM cache approach.

3.6.5.2 Sensitivity to DRAM Cache Sizes

Figure 16 shows the performance of the proposed mechanisms with different sizes of DRAM caches. The results show that the benefit of MissMap, HMP+DiRT, and HMP+DiRT+SBD increases as the cache size grows. For all cache sizes, HMP+DiRT+SBD

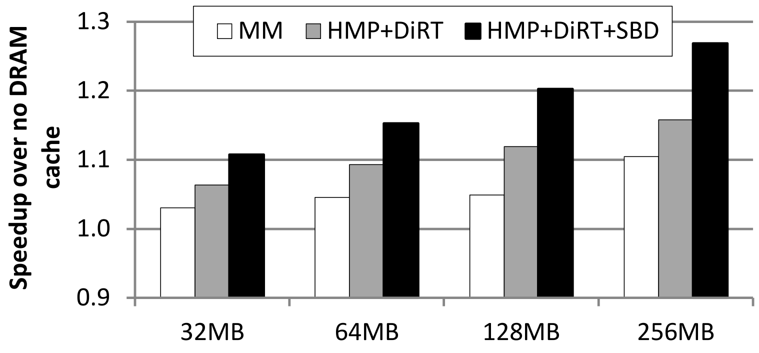


Figure 16: Performance sensitivity of the proposed mechanisms to different DRAM cache sizes.

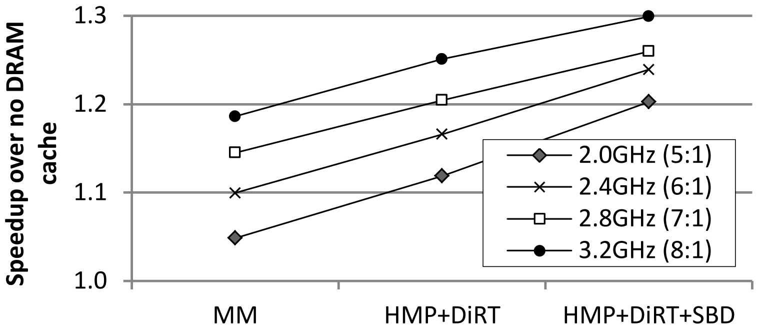


Figure 17: Performance sensitivity to different ratios of DRAM cache bandwidth to off-chip memory.

still performs best. In addition, the benefit of SBD increases as the DRAM cache size increases because the higher hit rate provides more opportunities for SBD to dispatch requests to main memory.

3.6.5.3 Sensitivity to DRAM Cache Bandwidth

In our evaluation, the ratio of peak DRAM cache bandwidth to main memory is 5:1 (2GHz vs. 1.6GHz, 4 channels vs. 2 channels, and 128-bit bus per channel vs. 64-bit bus per channel). While we believe that this is reasonable for plausible near-term systems,[3] it is also interesting to see how the effectiveness of HMP and SBD scales under different bandwidth assumptions. Figure 17 shows the performance sensitivity when we increase

[3]For example, current x86 processors tend to have two DDR3 memory channels (some have three or four, which would provide even more opportunities for SBD). The JEDEC Wide-IO standard provides four channels at 128 bits each, which is the same as our stacked DRAM configuration.

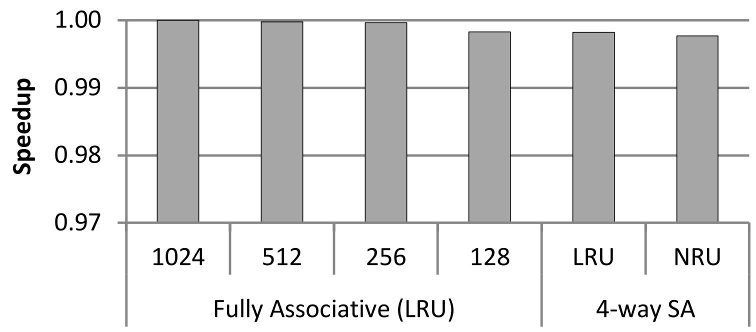


Figure 18: Performance sensitivity to different DiRT structures and management policies.

the DRAM cache frequency from 2.0GHz (what was used so far in this work) up to 3.2GHz. First, as shown in the results, the benefit of HMP does not decrease if future die-stacked DRAMs provide more bandwidth (relative to off-chip). As the DRAM-cache frequency increases, the cost of the 24-cycle MissMap increases relative to the DRAM-cache latency, and therefore HMP provides a small but increasing relative performance benefit as well. On the other hand, increasing the DRAM cache frequency reduces the *relative* additional bandwidth provided by the off-chip DRAM, thereby potentially decreasing the effectiveness of SBD. In our experiments, we do observe that the relative benefit of SBD over HMP reduces as the DRAM cache bandwidth increases, but overall, SBD still provides non-negligible benefits even with higher-frequency DRAM caches. Note that the die-stacked DRAM bandwidth may not grow too rapidly (such as 32:1), as adding more TSVs requires die area on the memory chips (which directly impacts cost), and increasing bandwidth via higher-frequency interfaces has power implications.

3.6.5.4 Sensitivity to DiRT Structures

Figure 18 shows the performance results as we vary the number of Dirty List entries (first four bars), assuming a fully-associative structure with LRU replacement. Note that such a structure would be difficult to implement for these sizes (e.g., true LRU on 1K entries). Overall, there is very little performance degradation even when reducing the size of the DiRT to only 128 entries, but we still chose to employ a 1K-entry table to reduce the performance variance across workloads.

The right side of Figure 18 also shows the results for 4-way set-associative implementations, each with 1K entries. The right-most bar (1K entry, 4-way, NRU) is the configuration used thus far in this work and has the lowest implementation complexity and cost. Overall, the results show that even with our simplified DiRT organization, we lose very little performance compared to an impractical fully-associative, true-LRU solution.

3.7 Summary

In this chapter, we have shown that there exist inefficiencies in the prior DRAM cache solution. In particular, the assumption that precise cache-content tracking was needed led to a MissMap structure that was over-designed for the DRAM cache. By taking advantage of the simple observation that on a miss, tag reads for victim selection need to occur anyway, false-negative mispredictions can be verified to prevent returning stale data from main memory back to the processors. The ability to freely speculate enables our DRAM cache organization that avoids the hardware overheads of the MissMap.

We also observed that while the die-stacked DRAM may provide significant bandwidth, the off-chip memory bandwidth is still a valuable resource that should not be disregarded. Our Self-Balancing Dispatch approach allows our DRAM cache design to make better use of the system's *aggregate* bandwidth. For both the HMP and SBD approaches, we found that life is significantly easier when we do not need to worry about dirty data. Completely abolishing dirty data from the DRAM cache with a write-through policy causes write traffic to increase tremendously. However, by bounding (and tracking) a limited number of pages in write-back mode, we could greatly amplify the effectiveness of both HMP and SBD techniques. Overall, we proposed a significantly streamlined DRAM cache organization that improves performance over the prior DRAM cache design. More importantly, this work addresses the scalability concerns of the die-stacked DRAM caches and thus makes a step forward towards a practical design for gigabytes of stacked DRAM caches.

CHAPTER IV

A RESILIENT DIE-STACKED DRAM CACHE

Traditionally, RAS for memory has been provided by using error-correcting code (ECC)-enabled DIMMs, where each rank's memory chips are augmented with one or more additional chips to store the ECC/parity information needed to protect the data. Such ECC DIMMs can provide basic single-error correction/double-error detection (SECDED) capabilities, or more complex ECCs (e.g., Reed-Solomon codes [65]) can provide chipkill protection that allows an entire memory chip to fail without compromising any data [13].

For die-stacked DRAM, it may not be practical to add extra chips to provide the additional capacity for storing ECC information. For example, if a DRAM stack has only four chips to begin with, it may not be physically practical to add another "half" chip to provide the extra storage (assuming +12.5% for SECDED). There are other complications with trying to extend a conventional ECC organization to stacked DRAM such as storage/power efficiency and economic feasibility, which we will cover in Section 4.1. In this chapter, we discuss how to provide resilience support for die-stacked DRAM caches in a cost-effective way.

4.1 Applying Conventional ECC to Stacked DRAM

Conventional external DRAM handles the ECC overhead by increasing overall storage capacity by 12.5%. There are two straightforward approaches for this in die-stacked memories. The first is to add additional chips to the stack to provide the capacity, just like what is done with DIMMs. For instance, in a chip stack with eight layers of memory, adding a ninth would provide the additional capacity. In stacked DRAM, however, when the additional chip is used to store ECC information (as in the conventional SECDED ECC), extra contention occurs on the ECC chip as the stacked DRAM does not require sourcing data bits from multiple chips unlike conventional DIMMs (Section 2.3); this makes an ECC check a significant bottleneck under heavy memory traffic, thereby increasing the load-to-use latency of memory requests.

Note that when the nine-chip stack is used as DIMM-like organizations where accesses are distributed across all of the layers, this approach suffers from all of the described shortcomings (e.g., performance/power issues). In addition, if the number of chips in the stack is not equal to eight (or some integral multiple thereof), then adding another chip is not cost effective. For example, in a four-layer stack, adding a fifth layer provides +25% capacity, which may be an overkill when, for example, SECDED only requires +12.5%.

The second straightforward approach is to increase the storage capacity of each individual chip by 12.5%. The width of each row could be increased from, for example, 2KB to 2.25KB, and the data bus width increased correspondingly (e.g., from 128 bits to 144 bits). There are no significant technical problems with this approach, but instead the problem is an economic one. A key to conventional ECC DIMMs is that the same silicon design can be deployed for both ECC and non-ECC DIMMs. Forcing memory vendors to support two distinct silicon designs (one ECC, one non-ECC) greatly increases their engineering efforts and costs, complicates their inventory management, and therefore makes this approach financially undesirable.

4.2 Objective and Requirements

The primary objective of this chapter is to provide a high level of reliability for die-stacked DRAM caches in a practical and cost-effective manner. From a performance perspective, memory requests usually require data only from a single chip/channel (i.e., bits are not spread across multiple layers of the stack). From a cost perspective, regular-width (non-ECC) DRAM chips must be used. From a reliability perspective, we must account for single-bit faults as well as large multi-bit (row, column, and bank) faults.

In Sections 4.3 and 4.4, we detail how to provide RAS support for die-stacked DRAM cache architectures while satisfying the above constraints. Furthermore, the proposed approach can provide varying levels of protection, from fine-grain single-bit upsets (SEC coverage), to coarser-grained faults (failure of entire rows or banks), and the protection level can be optionally adjusted at runtime by the system software or hardware.

4.3 Isolated Fault Types in DRAM Caches

In this section, we describe an approach to supporting single-bit error correction and multi-bit error detection for a DRAM cache to significantly diminish the probability of silent data corruptions (SDC). Correction of coarser-grained failures (e.g., row, bank or even channel faults) is covered later in Section 4.4.

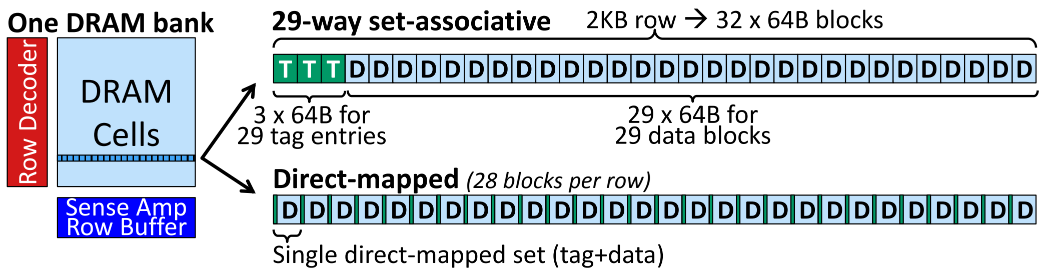

One DRAM bank
Row Decoder
DRAM Cells
Sense Amp Row Buffer
29-way set-associative
2KB row → 32 x 64B blocks
T T T D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
3 x 64B for 29 tag entries
29 x 64B for 29 data blocks
Direct-mapped (28 blocks per row)
D D D D D D D D D D D D D D D D D D D D D D D D D D D D
Single direct-mapped set (tag+data)

Figure 19: A DRAM bank with 2KB row size. When used as a cache, the row can be organized (top) as a 29-way set-associative set, or (bottom) as 28 individual direct-mapped sets.

4.3.1 Supporting Single-Bit Error Correction

In DRAM cache organizations, tags and data are placed together in the same row to take advantage of row buffer hits so that it is relatively easy to reallocate storage between different organizations. For example, Figure 19 provides one example in which the same physical 2KB row buffer can be re-organized to provide either a set-associative or a direct-mapped cache. This only requires the change of the control logic that accesses the cache; the DRAM itself is oblivious to the specific cache organization.

This same observation that data and tags are “fungible” leads us to a simple way to provide error correction for a DRAM cache. Figure 20(a) shows the components of a basic

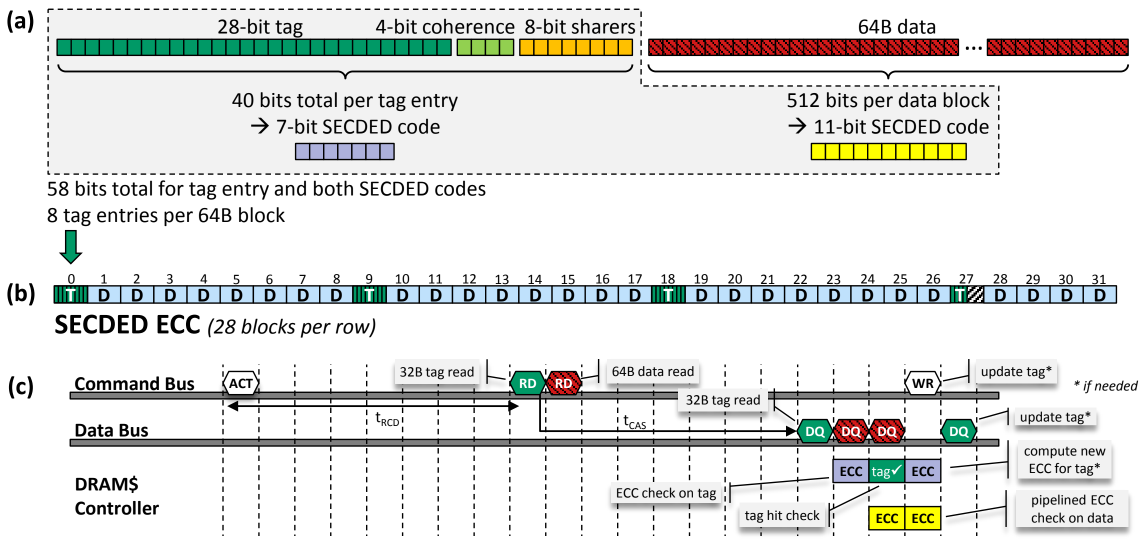


Figure 20: (a) Contents of one tag entry and one 64B data block, along with SECDED ECC codes for each, respectively. (b) Contents of a 2KB DRAM row, with eight tag entries packed into a 64B block and the corresponding eight data blocks following. (c) Timing diagram for reading a 64B cache block.

tag entry and a 64B cache block. This example tag entry consists of a 28-bit tag, a 4-bit coherence state, and an 8-bit sharer vector (used to track inclusion in eight cores)[1]; this example does not include replacement information because we assume a direct-mapped cache. We provide one SECDED ECC code to cover each tag entry, and then a separate SECDED ECC code to cover the corresponding 64B data block. In general, a block of n bits requires a SEC code that is about $\lceil \log_2 n \rceil$ bits wide to support single-bit correction, and then a parity bit to provide double-bit error detection [26]. The tag entry consists of 40 bits in total, thereby requiring a 7-bit SECDED code; the 512 data bits use an 11-bit code.

Placement of tags and data in a single 2KB row requires a little bit of organization to keep blocks aligned. We pack the original tag entry, the ECC code for the tag entry, and the ECC code for the data block into a single combined tag entry. These elements are indicated by the dashed outline in Figure 20(a), which collectively add up to 58 bits. We store eight of these tag entries in a 64B block, shown by the tag blocks ('T') in Figure 20(b). Following

[1]The actual sizes of each field will of course depend on the exact cache size, associativity, coherence protocol, etc.

each tag block are the eight corresponding data blocks. Overall, a single 2KB row can store 28 64B blocks plus the tags.

Inclusion of ECC codes requires slightly different timing for DRAM cache accesses. Figure 20(c) shows the timing/command sequence to read a 64B block of data from the cache. After the initial row activation, back-to-back read commands are sent to read both the tags and then data [62]. We assume a DRAM interface that can support 32B or 64B reads.[2] The ECC check of the tag entry occurs first to correct any single-bit errors; the corrected tag is then checked for a tag hit. The 64B data block is read in parallel with the tag operations (speculatively assuming a hit), and the data's ECC check is pipelined out over the two 32B chunks. At the end of the ECC check (assuming a cache hit and no errors), the data can be returned to the processor. If the tag must be updated (e.g., transitioning to a new coherence state), the tag entry and the corresponding ECC needs to be updated and then written back to the cache (marked with asterisks in the figure).

The case for a write is actually slightly simpler because the ECC for the data can be pre-computed based on the data value in hand. The updated tag cannot necessarily be precomputed depending on the meta-data stored in the tag. For example, the current coherence state is not known without first reading out the tag entry, therefore the tag will have to be read, the new tag contents updated, and then the new ECC code bits computed based on these new tag contents.

4.3.2 Supporting Multi-Bit Error Detection

In certain mission-critical environments, correction of errors is critical to maintain system uptime. However, many such environments also require robust *detection* of errors (even without hardware correction). It is bad to have a system crash after months of simulation time, but it is even worse for that system to suffer an undetected error and silently produce

[2]Current DDR3 supports both sizes, but it does not support switching between them on a per-request basis. We assume that with die-stacked DRAMs, the TSVs provide enough command bandwidth that adding a one-bit signal to choose between single-cycle (32B) or two-cycle (64B) bursts is not an issue.

erroneous results. Therefore, we are likely to want more robust detection than provided by the SECDED ECC scheme in Section 4.3.1.

For such mission-critical systems, we replace the DED parity bit in each ECC with a very-strong cyclic redundancy check (CRC). While CRCs are traditionally used for bursts of errors in data communications, they can detect much broader classes of errors beyond those that occur in bursts. For example, a 16-bit CRC is capable of detecting all errors of up to 5 bits in 46-bit data (40bit tags+6bit SEC), and all errors of up to 3 bits in 265-bit data (256bit data+9bit SEC), regardless of whether these errors are clustered (i.e., in a burst) or not. Furthermore, these CRCs can detect all burst errors up to 16 bits in length, all errors with an odd number of erroneous bits, and most errors with an even number of erroneous bits [44]. While these CRCs do not increase the DRAM cache's error-correction capabilities, they greatly increase its error-*detection* capability, thereby drastically reducing SDC rates.

Figure 21(a) shows the layout of the tag blocks. Here, we only use SEC codes (not SECDED); the CRCs provide multi-bit error detection and so the parity bit for double-error detection is not needed. We divide the 64B data into two 32B protection regions, each covered by its own SEC and CRC codes. This allows up to two errors to be corrected if each error occurs in a separate 32B region.

The storage requirement for the original tag plus the SEC and CRC codes is 112 bits. Therefore, tag information for four cache blocks can be placed in a 64B block. The overall layout for a 2KB DRAM row is shown in Figure 21(b), with a 64B tag block containing four tags (including SEC/CRC), followed by the four respective data blocks, and then repeated. The increased overhead reduces the total number of data blocks per 2KB row to 25.

The hardware complexity of a CRC is exactly equivalent to that of an equivalent-size ECC. Both CRCs and ECCs are expressed as an H-matrix, and require a logic tree of XOR operations for encode and decode. We assume four DRAM cache cycles to complete the CRC operation on data, which is conservative when considering the logic depth (about 8

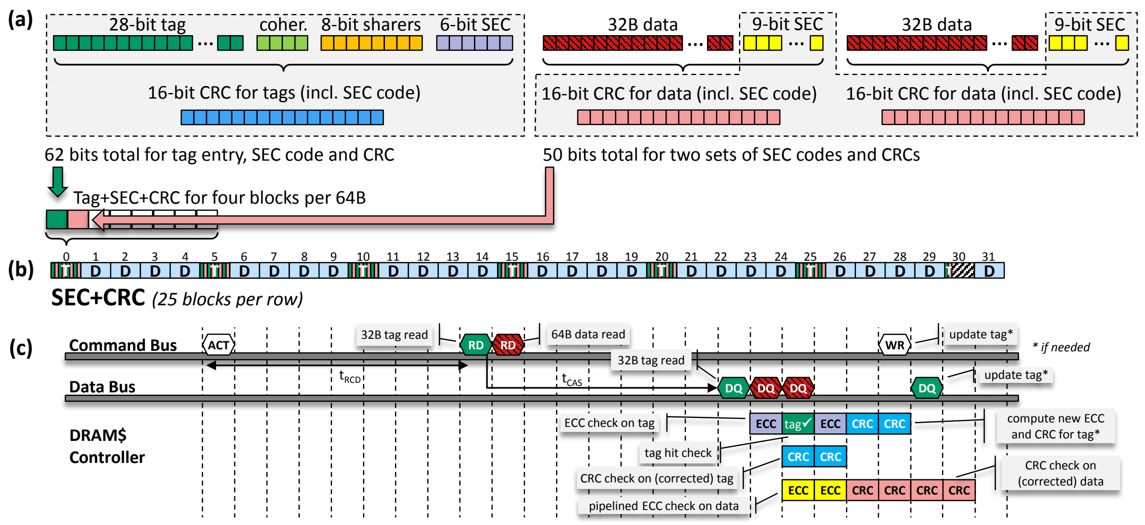

(a)
28-bit tag
coher.
8-bit sharers
6-bit SEC
16-bit CRC for tags (incl. SEC code)
62 bits total for tag entry, SEC code and CRC
32B data
9-bit SEC
32B data
9-bit SEC
16-bit CRC for data (incl. SEC code)
16-bit CRC for data (incl. SEC code)
50 bits total for two sets of SEC codes and CRCs
Tag+SEC+CRC for four blocks per 64B
(b)
SEC+CRC (25 blocks per row)
(c)
Command Bus
Data Bus
DRAM$ Controller
ACT
RD
RD
WR
DQ
tRCD
tCAS
32B tag read
64B data read
32B tag read
update tag*
update tag*
* if needed
ECC check on tag
tag hit check
CRC check on (corrected) tag
pipelined ECC check on data
compute new ECC and CRC for tag*
CRC check on (corrected) data
ECC
tag✓
CRC

Figure 21: (a) Contents of one tag entry and one 64B data block (treated as two 32B chunks for protection purposes), along with SEC ECC and CRC codes. (b) Contents of a 2KB DRAM row, with four tag entries (tag+SEC+CRC) packed into a 64B block and the corresponding four data blocks following. (c) Timing diagram for reading a 64B cache block.

XORs deep), additional cycles for wire delay, and so on. For the CRC operation on tags, we use two DRAM cache cycles, which is also conservative.

Figure 21(c) shows the timing for reads, which is very similar to the SECDED-only case from Figure 20, apart from a few minor differences. First, the tag block now contains both ECC and CRC information, so when the tag block must be updated, the final tag writeback is delayed by two extra cycles for the additional latency to compute the new CRC. Second, both the tag and data SEC ECC checks are followed by the corresponding CRC checks. We can return data to the CPU as soon as the ECC check finishes; that is, data can be sent back before the CRC check completes (or even starts). Even if a multi-bit error were detected, there is nothing the hardware can do directly to correct the situation. We assume the hardware simply raises an exception and relies on higher-level software resiliency support (e.g., checkpoint restore) to handle recovery.

]

4.3.3 Discussions

Storage Overhead: We use a direct-mapped design in which a single DRAM row contains 28 data blocks. The baseline has enough left-over tag bits to fit the ECC codes; so, SECDED-only can be supported *without* sacrificing further capacity efficiency compared to the non-ECC case. For SEC+CRC, the effective capacity has been reduced from 28 to 25 ways, or an overhead of 3/28 = 10.7%. Compare this to conventional ECC DIMMs in which eight out of nine (or 16 of 18) DRAM chips are used for data, and therefore the corresponding ECC overhead is 1/9 = 11.0%; i.e., the storage overhead of our SEC+CRC approach for DRAM caches is comparable to the effective overhead of conventional off-chip ECC DIMMs.

Controller Support: Our schemes do not require any changes to the stacked DRAM itself; they only require appropriate stacked-DRAM controller support depending on the exact schemes supported. For example, previous work described how non-power-of-two indexing for a direct-mapped DRAM cache can be easily implemented because modulo-by-constant operations are significantly simpler than general-case remainder computations [62]. For a stacked-DRAM controller that supports both SECDED (28 sets per row) and SEC+CRC (25 sets per row), the controller would require two separate modulo-by-constant circuits (i.e., mod 28 and mod 25) and use the appropriate circuit depending on the current RAS mode.

Comparison to Stronger ECC: An alternative approach to our SEC+CRC method is to provide a stronger ECC, such as a Double-Error-Correct Triple-Error-Detect (DECTED) ECC, instead of a CRC. However, note that ECC codes trade detection capability for correction, so they will always detect fewer errors than an equivalent-length CRC. For example, we evaluated using a DECTED ECC and found that it has substantially higher SDC rates than our SEC+CRC (7.9% versus 0.0007%).

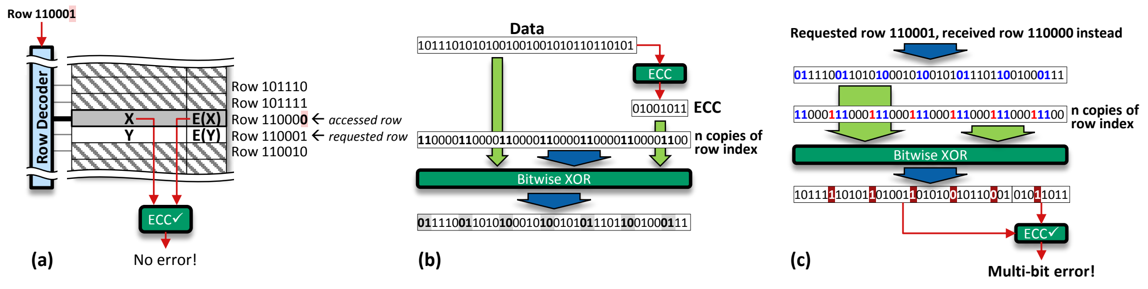


Figure 22: (a) Row decoder error that selects the incorrect row, which is undetectable using within-row ECC. (b) Process for folding in the row index (row 110000_2), and (c) usage of the folded row index to detect a row-decoder error.

4.4 Coarse-Grain Failures in DRAM Caches

DRAM failure modes are not limited to single-bit/few-bit soft errors from corrupted bitcells. Coarse-grain failures also occur with a non-trivial frequency in real systems [80], affecting entire columns, rows, banks, etc. This section details an approach to deal with coarse-grain DRAM cache failures.

4.4.1 Identifying Coarse-Grain Failures

Before handling failures, the failure must be detected. Here we cover how failures are detected for different scenarios.

Row Decoder Failures: The failure of an entire row can occur due to a fault in the row decoder logic [1]. If the row decoder has a fault in which the wrong wordline is asserted, then the data from the wrong row will be sensed, as shown in Figure 22(a). The DRAM should have returned row 110001's contents: the data Y and the ECC codes for Y, namely E(Y). In this case, however, the adjacent row's contents, X and E(X), are returned instead, but the data and ECC codes are self-consistent, and so the system would not be able to detect that the wrong row had been accessed.

To detect this scenario, we fold in the row index into the data. Figure 22(b) shows example data and ECC fields. We first compute the ECC on the data. Then, instead of storing the raw data, we store the exclusive-OR of the data and several copies of the row index. When reading the data, we first XOR out the row index, which returns the original data; from here, we perform the ECC check.

If the row decoder has a fault, then the wrong row will be read. For example, Figure 22(c) shows a case when reading row 110001_2 results in the previous row 110000_2 instead. We XOR out the row index *for the row that we requested* (e.g., 110001_2), but this leaves the data in a state with multiple "wrong" bits with respect to the stored ECC code. The multi-bit error is detected, and the system can then attempt to do something about it (e.g., raise a fault). A similar approach was proposed in the Argus microarchitecture [54], although our usage covers more fault scenarios due to our CRCs.

Column Failures: An entire column may fail due to problems in the peripheral circuitry. For example, the bitline may not be precharged precisely to the necessary reference voltage, or variations in transistor sizing may make the sense amplifier for a particular column unable to operate reliably. These faults may be permanent failures (e.g., manufacturing defects, wearout) or intermittent (e.g., temperature-dependent). For a bank with a single column failure, reading any row from this bank will result in a corresponding single-bit error. This can be caught and corrected by the baseline ECC. If multiple columns have failed, the baseline CRC checksum can detect the failure in most cases.

Column failures may also occur if the column decoder selects the wrong column (e.g., if the column index was incorrectly latched due to noise). Similar to hashing in the row index already described, we can easily extend the scheme to also XOR in the column index. Prior to reading bits from the DRAM caches, both the row and column indexes are XOR'ed out. An error in either case will cause a CRC mismatch with high probability. The system may record some amount of fault history, and from this it would be very easy to detect that errors consistently occur in a particular column or set of columns. When this has been detected, the system could map out the column using spare resources [41], but such schemes are beyond the scope of this work.

Bank/Channel Failures: In the case of an entire bank failing, reading *any* row from that bank likely will result in garbage/random bit values, all zeros, or all ones. For random bit values, the probability of the CRC fields being consistent with the data portion will be very low, and so this would manifest itself as an uncorrectable multi-bit error. For all zeros or all ones, instead of just XORing in multiple copies of the row index, some copies (e.g., every other one) are bitwise inverted. Similar to the row-decoder failure, it is possible that the bank decoder fails and sends a request to the wrong bank. The row-index XOR scheme can be extended to include the bank index. The failure of an entire channel and/or channel decoder faults can be treated in a similar manner.

4.4.2 DOW: Duplicate-on-Write

To tolerate multi-bit errors, row failures and even bank failures, we use a *Duplicate-on-Write* (DOW) approach that has some similarities to RAID-1 used for disk systems, but does not incur the same amount of storage overhead. RAID-1 duplicates *every* disk block (so the filesystem can tolerate the failure of a complete disk), and therefore a storage system must sacrifice 50% of its capacity to provide this level of protection (and 50% of its bandwidth for writes).

The key observation for the DRAM cache is that for *unmodified* data, it is sufficient to detect that an error occurred; the correct data can always be refetched from main memory. For dirty data, however, the modified copy in the cache is the *only* valid copy, and so there is no where else to turn to if this sole copy gets corrupted beyond repair. This observation has been leveraged to optimize the protection levels of physically distinct caches (e.g., parity in the IL1 and SECDED ECC in the DL1), we extend this concept to vary protection within the shared, unified same cache structure.

DOW stores a duplicate copy of data *only when the data are modified*. This way, the duplication overhead (capacity reduction) is limited to only dirty cache blocks. Figure 23 shows a few example cache blocks; blocks A, B, C, and D are all clean, and so the cache stores only a single copy of each. If any are corrupted beyond repair (e.g., C), then the

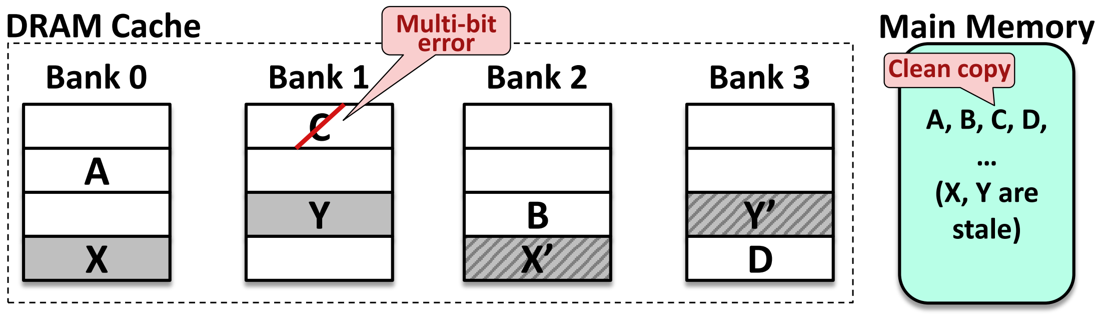


Figure 23: Example DRAM cache contents in which clean data are backed-up by main memory, but dirty data are duplicated into other banks.

clean copy in memory can provide the correct value.[3] Blocks X and Y are dirty, and so we create duplicate copies X' and Y' in other banks. If the rows (or entire banks) for X or Y fail, we can still recover their values from X' or Y'.

In this example, we use a simple mapping function for placing the duplicate copy. For N banks, a cache line mapped to bank i has its duplicate placed in bank $i + \frac{N}{2} \mod N$. To support channel-kill, the duplicate from channel j is instead mapped to channel $j + \frac{M}{2} \mod M$, assuming M total DRAM cache channels. More sophisticated mapping could reduce pathological conflict cases, but we restrict our evaluations to this simple approach.

Operational Details Here, we briefly explain the mechanics of the different possible scenarios for reading, writing, evictions, etc., and how DOW operates for each of these. The guiding invariants for DOW operation are: (1) if a cache line is clean, then there exists one and only one copy in the DRAM cache, and (2) if a line is dirty, then there exists exactly two copies of the modified data in the DRAM cache.

Read, Clean Data: The cache line is clean; so, only one copy exists in the cache in its original location. This line is used.

Read, Dirty Data: While two copies exist, we simply use the copy located in the original location.

[3]We assume some adequate level of protection of main memory; protection of main memory has been well researched and is outside the scope of this work.

Write, Currently Clean Data: The cache line is now being modified, and so a duplicate must be created to satisfy the invariant that two copies of a dirty line must exist in the cache. A line from the duplicate's target bank must first be evicted, and then the (modified) duplicate is installed. The original copy is modified as well. These operations may be overlapped.

Write, Already Dirty Data: The original copy is modified. When checking the location of the duplicate, there will be a hit and so no additional lines need to be evicted. The duplicate is modified to match the newly updated original copy.

Read/Write, Cache Miss: The bank where the original copy should reside is checked and a miss is discovered. The request is then forwarded directly to main memory. The location to where a duplicate would map is not checked because the duplicate exists only if the line was dirty (by invariant #2). Given that the original location resulted in a miss, the cache line necessarily cannot be dirty and therefore the duplicate cannot exist.

Eviction of Clean Data: This is the only copy in the cache, and it is consistent with main memory, so the cache line may simply be dropped. Updates to the home node may still be required if using a directory-based cache-coherence protocol.

Eviction of Duplicate: The line is dirty and so it must be written back to main memory on an eviction. The original copy either may be invalidated or downgraded to a clean state (our invariants do not permit a single copy of dirty data existing by itself, although one clean copy or zero copies are allowed).

Eviction of Dirty Original: Like the previous case, the line is dirty and so it is written back to main memory. In this case, the duplicate is invalidated; it is not useful to keep the duplicate around even if we downgrade it to a clean state, because on a cache miss to the original cache line's bank, we would proceed directly to main memory and *not* check the duplicate's bank.

Read, Corrupted Data: For a single-bit error, the baseline ECC will correct it and then the read proceeds as normal according to the relevant scenario described previously. In the case of an uncorrectable multi-bit error, if the data is clean, then a request is sent to main memory. The value from memory is returned to the requesting core. If the data is dirty, then the copy from the duplicate is retrieved and returned to the user (assuming that the duplicate has no uncorrectable errors). If both copies have uncorrectable errors, then a fault is raised (i.e., there is nothing more the hardware can do about it). Whether the correct data is provided by main memory or the duplicate copy, the correct data are rewritten into the original location, which effectively corrects the multi-bit errors. Optionally, the data can immediately be read out again and compared to the known-good value. If the data come out corrupted again, then this strongly suggests that the problems are not the result of random soft errors (e.g., high-energy particle strikes), but are in fact due to an intermittent or hard fault of some sort.

Read, Corrupted Tag: If a tag entry has an uncorrectable multi-bit error, then we cannot know for certain whether we would have had a cache hit or miss, nor whether the data would have been dirty or clean. In this case, we must conservatively assume that there was a dirty hit, so we access the duplicate location. If we find the requested block in the duplicate location, then that is the sole surviving copy of the dirty data, which we return to the requesting core and reinstall into the original cache line location. If we do not find the line in the duplicate location, then either the line was not present in the cache to begin with, or it (the original copy) was present but in a clean state when it was corrupted. For either case, it is safe (in terms of correctness) to read the value from main memory.

4.4.3 Summary of Coverage

The ECC+CRC scheme provides single-error correction and strong multi-error detection for one or more individual bitcell failures as well as column failures. Further layering DOW provides row-failure coverage and complete bank-kill and channel-kill protection. The idea of providing two different levels of protection is similar in spirit with the memory-mapped ECC proposal [88], although the details for our DRAM cache implementation are completely different. In the best case for our approach, each of the N banks is paired with one other bank (in another channel), and so one bank may fail from each of the $\frac{N}{2}$ pairs. If a system implements multiple DRAM stacks with channels interleaved across stacks, then DOW automatically provides *stack-kill* protection as well. Note that while this work focuses on a specific configuration and corresponding set of parameters, the proposed *approach* is general and can be tailored to specific stacked-DRAM performance and capacity attributes as well as the overall RAS requirements for different systems.

4.4.4 Optimizations for DOW

While DOW limits the duplication overhead to only those cache lines that have been modified, in the worst case (i.e., when *every* cache line is dirty) the overhead still degenerates back to what a "RAID-1" solution would cost. We now discuss several variants on DOW that can reduce duplication overheads.

Selective DOW: Not all applications and/or memory regions are critical, and therefore not all need to have the high-level of reliability that DOW provides. IBM's Power7 provides a feature called *selective memory mirroring* where only specific portions of memory regions are replicated into split channel pairs [39]. Similarly, dirty-data duplication (DOW) can be selectively applied to specific applications and/or memory regions. For example, critical operating-system data structures may require strong protection to prevent system-wide crashes, but low-priority user applications need not be protected beyond basic ECC. Even within an application, not all pages need to be protected, although to support this level of fine-grain coverage, the application must provide some hints about what should be protected.

Background Scrubbing: On-demand scrubbing can occur based on the amount of dirty data in the cache. Each time a dirty line is added to the cache, a counter is incremented (and likewise decremented for each dirty eviction). If the counter exceeds a threshold, then scrubbing is initiated. The scrubbing would proceed until the number of dirty lines drops below a “low-water mark” (some value less than the threshold) to prevent the scrubbing from constantly kicking in. The writeback traffic can be scheduled during bus idle times and/or opportunistically when a DRAM row is already open. This can also be used in concert with eager-writeback techniques [48].

Duplication De-prioritization: When initially installing duplicates, or when updating existing duplicates, these writes are not typically on the program’s critical path (these are the result of dirty line evictions from the upper-level caches, not the result of a demand request). The DRAM cache’s controller can buffer the duplicate write requests until a later time when the DRAM cache is less busy, thereby reducing the bank-contention impact of the duplicate-update traffic.

4.5 Experimental Results

4.5.1 Methodology

Simulation Infrastructure: We model a quad-core processor with two-level SRAM caches and an L3 DRAM cache. The stacked DRAM is expected to support more channels, banks, and wider buses per channel [17, 43, 50]. In this study, the DRAM cache has eight channels each with 128-bit buses, and each channel has eight banks [37], while the off-chip DRAM has two channels, each with eight banks and a 64-bit bus. Also, key DRAM timing parameters with bank conflicts and data bus contention are modeled for both the DRAM cache and main memory. Virtual-to-physical mapping is also modeled to ensure that the same benchmarks in different cores do not map into the same physical addresses. Table 6 shows the system configurations used in the study.

Workloads: We use the SPEC CPU2006 benchmarks and sample one-half billion instructions using SimPoint [71]. We then categorize the applications into two different groups based on the misses per thousand instructions (MPKI) in the L2 cache. We restrict the study to workloads with high memory traffic; applications with low memory demands have very little performance sensitivity to memory-system optimizations and therefore

Table 6: System parameters used in the study.

Processors	
Core	4 cores, 3.2 GHz out-of-order, 4 issue width
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	16-way, shared 4MB (24-cycle)
Stacked DRAM caches	
Cache size	128 MB
Bus frequency	1.6 GHz (DDR 3.2GHz), 128 bits per channel
Channels/Ranks/Banks	8/1/8, 2048 bytes row buffer
tCAS-tRCD-tRP	8-8-8
tRAS-tRC	26-34
Off-chip DRAM	
Bus frequency	800 MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11

expose very little additional insight (we did verify that our techniques do not negatively impact these benchmarks). From the memory-intensive benchmarks, those with average MPKI rates greater than 27 are in Group H (for **H**igh intensity), and of the remaining, those with 15 MPKI or more are in Group M (for **M**edium). Table 7 shows MPKI values of the benchmarks and their groups.

Table 7: L2 misses per thousand instructions (L2 MPKI).

Group M	**MPKI**	**Group H**	**MPKI**
milc	18.59	leslie3d	27.69
wrf	19.04	libquantum	28.39
soplex	23.73	astar	29.26
bwaves	24.29	lbm	36.62
GemsFDTD	26.44	mcf	52.65

We select benchmarks to form rate-mode (all cores running separate instances of the same application) and multi-programmed workloads. Table 8 shows the primary workloads evaluated for this study. Section 4.6 also includes additional results covering a much larger number of workloads. We simulate 500 million cycles of execution for each workload. We verified that this simulation length is sufficiently long to cause the contents of the 128MB DRAM cache to turn over many times (i.e., the DRAM cache is more than sufficiently warmed up).

Table 8: Multi-programmed workloads.

Mix	Workloads	Group
WL-1	4 × mcf	4×H (rate)
WL-2	4 × leslie3d	4×H (rate)
WL-3	mcf-lbm-milc-libquantum	4×H
WL-4	mcf-lbm-libquantum-leslie3d	4×H
WL-5	libquantum-mcf-milc-leslie3d	4×H
WL-6	mcf-lbm-libquantum-soplex	3×H + 1×M
WL-7	mcf-milc-wrf-soplex	2×H + 2×M
WL-8	lbm-leslie3d-wrf-soplex	2×H + 2×M
WL-9	milc-leslie3d-GemsFDTD-astar	2×H + 2×M
WL-10	libquantum-bwaves-wrf-astar	1×H + 3×M
WL-11	bwaves-wrf-soplex-astar	1×H + 3×M
WL-12	bwaves-wrf-soplex-GemsFDTD	4×M

Performance Metric: We report performance of our quad-core system using weighted speed-up [18, 79] and use the geometric mean to report average values.

Failure Modes and Rates: We assume DRAM failure modes and rates similar to those observed from real field measurements (FIT in Table 9 [80]). We report results using both the observed failure rates (Table 9) as well as failure rates of 10× the observed rates to account for potential increases in failures due to die stacking and for inter-device variations in failure rates [61]. At this point, we are not aware of any published studies reporting failure types and rates for die-stacked DRAM. The assumption that failure rates will be 10× may be somewhat pessimistic, but by providing our results across the range of 1-10× the currently known FIT rates, the actual stacked DRAM results should fall somewhere in between. Furthermore, the *relative* mean time to failure (MTTF) rates for our proposed schemes compared to the baseline are independent of the underlying device FIT rates.

ECC and CRC simulation: We evaluate the performance of our ECC and ECC+CRC schemes using Monte Carlo simulation assuming a bit error rate (BER) of 0.5 in a faulty DRAM location (e.g., within a faulty row). This bit error rate corresponds to a 50%

Table 9: Failure rates measured on external DRAM [80].

Failure Mode	Failure Rate (FIT)
Single Bit	33
Complete Column	7
Complete Row	8.4
Complete Bank	10

probability that a given bit is in error, and is chosen because the erroneous value returned by the DRAM will sometimes match the expected data value. For example, a DRAM row fault will not produce errors on every bit within that row (which would correspond to a BER of 1), but rather only on those bits where the returned value does not match the expected value.

We separately simulate row, column, and bank faults, and run 100 million simulations for each type of fault. Each simulation randomly flips bits within a single row, column, or bank (with a probability of 50%), and then applies the ECC and CRC logic to determine whether the error will be corrected or detected. We do not model row decoder or column decoder faults; we assume these are caught 100% of the time by the row and column address XOR.

To calculate failure rates, we apply the detection and correction coverage to previously reported FIT rates [80]. For the purposes of this work, we assume all undetected errors result in silent data corruption (SDC). This is likely to be somewhat conservative due to fault masking effects [58], but note that our relative improvements will remain accurate.

In evaluating DOW, we assume a single-fault model, i.e., only one fault at a time will exist in the DRAM cache. Similar to traditional chipkill, DOW does not guarantee correction when there are multiple independent faults in the DRAM cache (i.e., DOW provides "bank-kill" or "channel-kill" capability, depending on placement of the duplicate line). Unlike traditional chipkill, there is a very small likelihood that DOW will not detect all two-fault cases. This can occur if both the original and duplicate lines have a fault and the duplicate's CRC mismatches. Given the failure rates and detection coverage, the failure rate of this case is less than 1e-13 FIT, or once per 80 quadrillion years for a four-chip DRAM cache.

4.5.2 Fine-Grain Protection

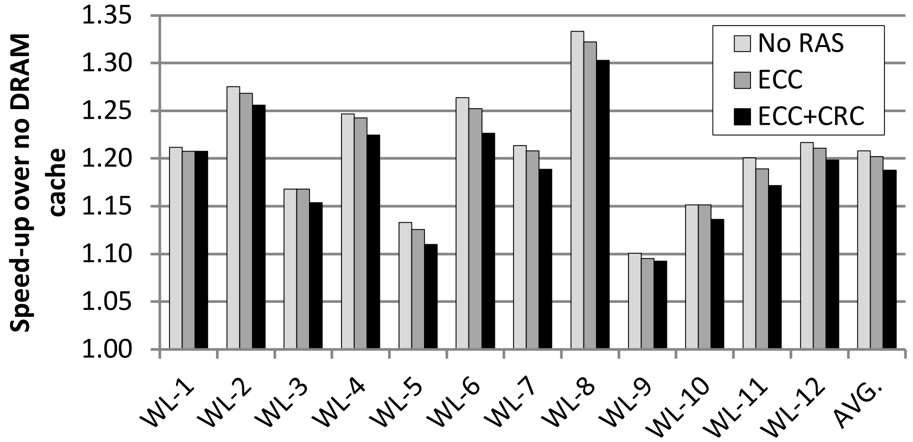


Figure 24: Performance comparison among no RAS, ECC, and ECC+CRC (normalized to no DRAM cache).

We first evaluate the performance impact of the proposed fine-grain protection schemes on DRAM caches. Figure 24 shows the speed-up over no DRAM cache between no RAS, ECC and ECC+CRC configurations. The results show that the performance impact of our proposed schemes is small. On average, the ECC and ECC+CRC schemes only degrade an average of 0.50% and 1.68% compared to a DRAM cache with no RAS support, respectively. ECC+CRC reduces the cache capacity compared to no RAS, and it also slightly increases bandwidth consumption; we further analyze the capacity and bandwidth impact of ECC+CRC in Section 4.6.2.

4.5.3 Coarse-Grain Protection

Figure 25 shows the performance of our proposed coarse-grain protection scheme (DOW) when applied on top of ECC+CRC. We compare the results with ECC+CRC and ECC+CRC+RAID-1 to show the effectiveness of DOW. In the case of the RAID-1-style approach (i.e., full duplication), not only is cache capacity reduced by half, but effective write bandwidth is reduced by half as well, thereby leading to a non-negligible performance degradation of as much as 13.1% compared to ECC+CRC (6.5% on average). However, the DOW scheme retains much of the overall performance benefit of having a DRAM cache (on average, only -2.5% and -0.8% performance loss compared to no RAS and ECC+CRC, respectively) while providing substantial RAS improvements.

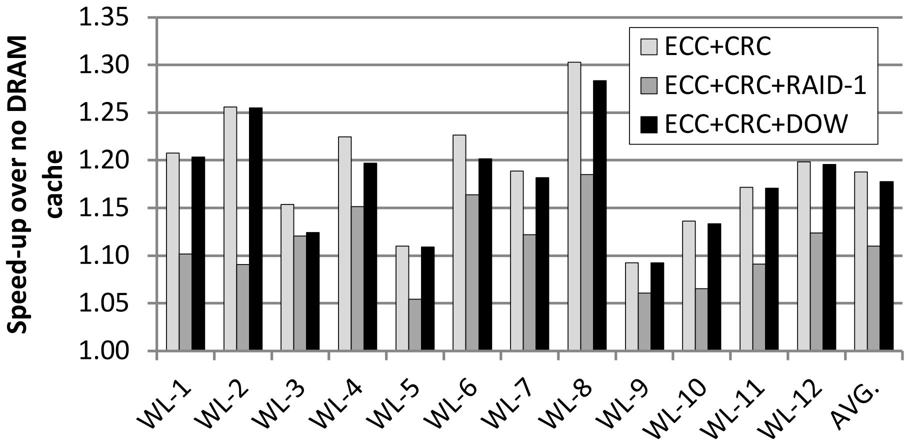


Figure 25: Performance comparison between fine-grain (ECC+CRC) and coarse-grain (ECC+CRC+RAID-1 and ECC+CRC+DOW) schemes (normalized to the performance without the DRAM cache).

Table 10: Detection coverage for each technique.

	Detection			
Failure Mode	**No RAS**	**ECC Only**	**ECC+CRC**	**DOW**
Single Bit	0%	100%	100%	100%
Column	0%	85%	99.9993%	99.9993%
Row	0%	50%	99.9993%	99.9993%
Bank	0%	50%	99.9993%	99.9993%

4.5.4 Fault Coverage and Failure Rates

Table 10 shows the percentage of faults detected in each failure mode by each of our schemes, assuming a single four-layer stack of DRAM. ECC-only detects all single-bit

Table 11: Correction coverage for each technique. Cases where correction coverage differs from detection coverage (Table 10) are marked with $\triangleright\triangleleft$.

	Correction			
Failure Mode	**No RAS**	**ECC Only**	**ECC+CRC**	**DOW**
Single Bit	0%	100%	100%	100%
Column	0%	85%	$\triangleright$ 85% $\triangleleft$	99.9993%
Row	0%	$\triangleright$ 0% $\triangleleft$	$\triangleright$ 0% $\triangleleft$	99.9993%
Bank	0%	$\triangleright$ 0% $\triangleleft$	$\triangleright$ 0% $\triangleleft$	99.9993%

Table 12: Results using observed and 10× DRAM failure rates.

Failure Rates	**No RAS**	**ECC Only**	**ECC+CRC**	**DOW**
SDC FIT	234-2335	41-410	0.0008-0.0075	0.0008-0.0075
DUE FIT	0	37-368	52-518	0

faults and most column faults, because most of these faults affect only one bit per row [80]. ECC-only also detects 50% of all row and bank faults, which look to the ECC like double-bit errors. ECC+CRC substantially improves the detection coverage of column, row, and bank faults, detecting 99.9993% of all faults. The detection percentage of the CRC depends on the fault model. We assume that every bit in the row or bank has a 50% chance of being incorrect; lower error rates (as might be observed with different failure modes) substantially increase the detection percentage of the CRC. We also conservatively assume that row and bank failures are due to all bits in the row or bank being bad, rather than to row or bank decoder failures which would be caught by XORing the row and bank address.

Table 11 shows the fraction of faults, by failure mode, that our schemes can correct. ECC-only and ECC+CRC correct all single-bit and 85% of column faults, but cannot correct any row or bank faults.[4] DOW, on the other hand, corrects all detected faults.

Table 12 shows the overall silent data corruption (SDC) and detectable unrecoverable error (DUE) FIT rates for our techniques, using both observed and 10× DRAM failure

[4]Row and bank faults with only a single bit in error can be corrected by these schemes, but we assume that this does not occur.

rates. We assume that all undetected failures will cause a silent data corruption. No RAS leads to a SDC FIT of 234-2335, or an SDC MTTF of 49-488 years. ECC-only reduces the SDC FIT by 5.7×, but increases the DUE FIT to 37-368 FIT. ECC+CRC reduces the SDC FIT to just 0.0008-0.0075 FIT, but this comes at the expense of an increase in DUE FIT to 52-518 (220-2200 years MTTF). Finally, DOW adds the ability to correct all detected errors while maintaining the same SDC FIT as ECC+CRC. Overall, DOW provides a more than 54,000× improvement in SDC MTTF compared to the ECC-only configuration. While the reported MTTF rates may appear to be adequately long, these can still result in very frequent failures in large datacenter or HPC installations. The impact on such systems will be further quantified in Section 4.6.

4.6 Analysis and Discussions

4.6.1 Reliability Impact on Large System Sizes

The bandwidth requirements of future HPC systems will likely compel the use of significant amounts of die-stacked DRAM. The impact of DRAM failures in these systems is significantly worse than for single-socket systems because FIT rates are *additive* across nodes. For example, using the baseline (1×) FIT rates from Table 9, a 100,000-node HPC system with four DRAM stacks per node would have an SDC MTTF of only 10 hours from the die-stacked DRAM alone with no RAS support. Our ECC-only technique would increase the SDC MTTF to only 60 hours. By contrast, ECC+CRC and DOW have an SDC MTTF of 350 years for the entire 100,000-node system. Inclusion of DOW is likely necessary, because ECC+CRC (without DOW) has a 48-hour DUE MTTF on such a system. While DUEs might be handled by software techniques, the performance overheads of restoring the system to a checkpoint every two days may not be acceptable. This analysis optimistically assumes the baseline DRAM FIT rates, which are likely lower than what will be observed with die-stacked DRAM (e.g., if we assume a 10× FIT rate, then without DOW, the system would have to rollback to a checkpoint about every five hours).

4.6.2 Capacity and Bandwidth Impact

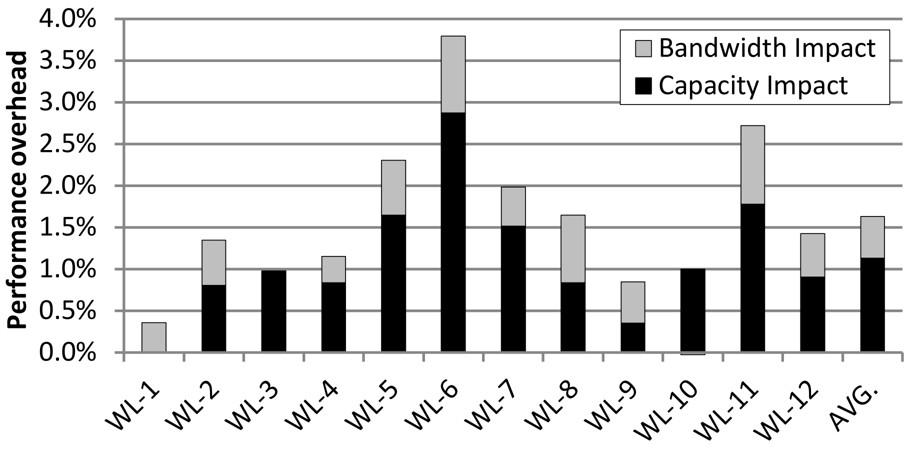


Figure 26: Capacity and bandwidth impact of the ECC+CRC scheme.

With RAS support, the performance of DRAM caches is lower due to reduced cache capacity as well as higher latency/bandwidth. To quantify the capacity impact, we evaluate a DRAM cache configuration with the same capacity as ECC+CRC, but without the additional traffic. The bottom portion of the bars in Figure 26 shows the performance *reduction* due to the cache capacity reduction alone. The remaining performance loss comes from the additional bandwidth and latency effects of handling the ECC and CRC codes.

4.6.3 Impact of Early Data Return

As described in Section 4.3.2, our ECC+CRC scheme (no DOW) returns data before the completion of data CRC checks, which is based on the observation that hardware cannot correct the multi-bit errors anyway; thus, we do not need to wait for the check to be finished for every cache hit. We also simulated a version of ECC+CRC where we wait for the full CRC check to complete before returning data, which may be of use to support data poisoning. On average, this degrades performance by only 0.5%; this is because the CRC check adds only four more DRAM cache cycles to the load-to-use latency, which is a small relative increase compared to the latency of activating and reading the DRAM cache row in the first place.

4.6.4 Duplication Overheads of DOW

While DOW provides much better performance than a naive RAID-1 approach, DOW still causes extra data duplication and writeback traffic. Figure 27(a) shows the percentage of cachelines in the DRAM cache that are dirty for ECC+CRC and also when DOW is added. One would expect that the amount of dirty data should *increase* due to duplication, but in fact the amount of dirty data decreases because of the maintenance of the invariant that all dirty cache lines must have two copies. Thus, if *either* copy is evicted, then the amount of dirty data in the cache goes down. The impact of this is that there is an increase in the DRAM cache's writeback traffic. Figure 27(b) shows the number of writebacks from the DRAM cache per thousand cycles. For most workloads, the increase in writeback traffic is not significant. The standouts are workloads WL-3, WL-4, WL-6 and WL-8 that start with relatively higher amounts of writeback traffic, and then DOW causes that traffic to increase by another approximately 20%. Not surprisingly, these are also the workloads that exhibited the largest performance losses when applying DOW (see Figure 25). While some applications show moderate increases in writeback traffic, the absolute traffic is still low, which explains why DOW does not have a significant negative performance impact in most cases.

4.6.5 Sensitivity to Cache Size

Figure 28 shows the average speed-up of no RAS, ECC, ECC+CRC, and ECC+CRC+DOW over no DRAM cache with different cache sizes. For the fine-grain protection schemes, the performance degradation is relatively small across the cache sizes. For DOW, the relative performance loss increases with larger caches, which is due to an increase in the amount of duplicated data. However, DRAM caches with DOW still deliver the majority of the performance benefit of the no RAS approach while providing far superior RAS capabilities.

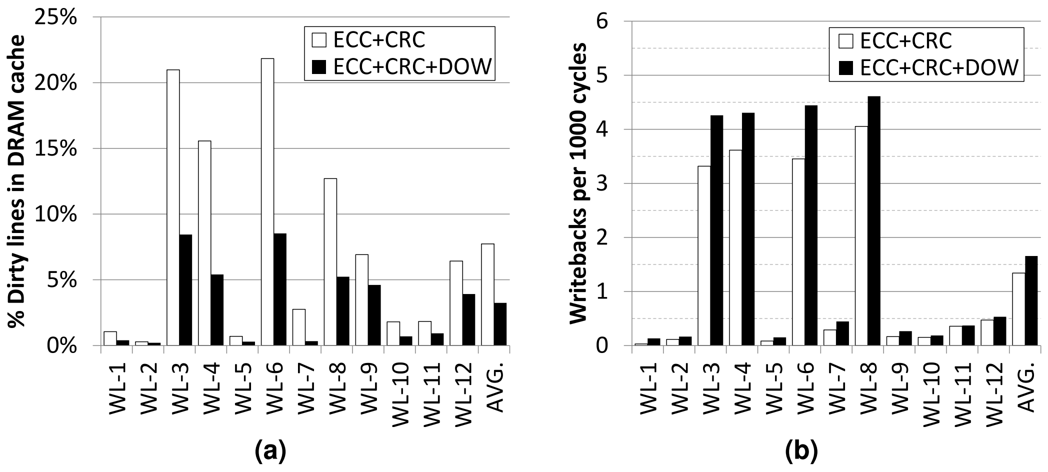


Figure 27: Impact of DOW on (a) the amount of dirty lines in the DRAM cache and (b) the writeback traffic from the DRAM cache.

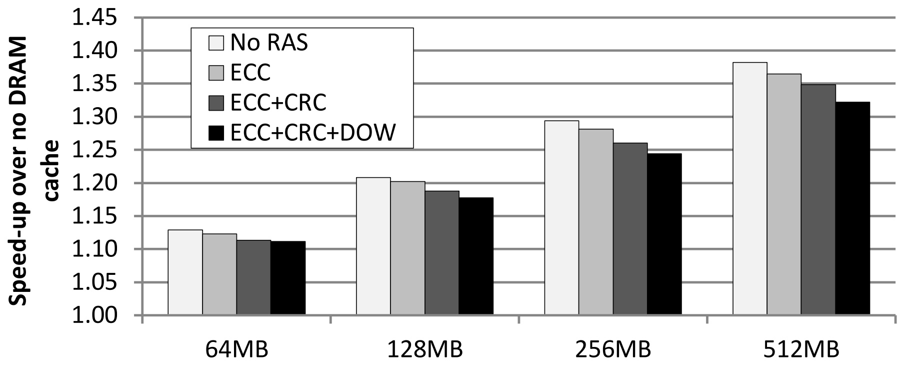


Figure 28: Sensitivity to different cache sizes (workloads in Table 8).

4.6.6 Sensitivity to Different Workloads

We apply our protection schemes to all $\binom{10}{4} = 210$ combinations of the applications in Table 7 to ensure that the performance impact is also minimal for a broader set of workloads beyond the primary workloads used in the work. Figure 29 presents the average speed-up (with $\pm$ one standard deviation) over the 210 workloads. The proposed RAS capabilities have a relatively small impact on performance. ECC, ECC+CRC and ECC+CRC+DOW degrade performance by only 0.50%, 1.65% and 2.63% on average over no RAS, respectively.

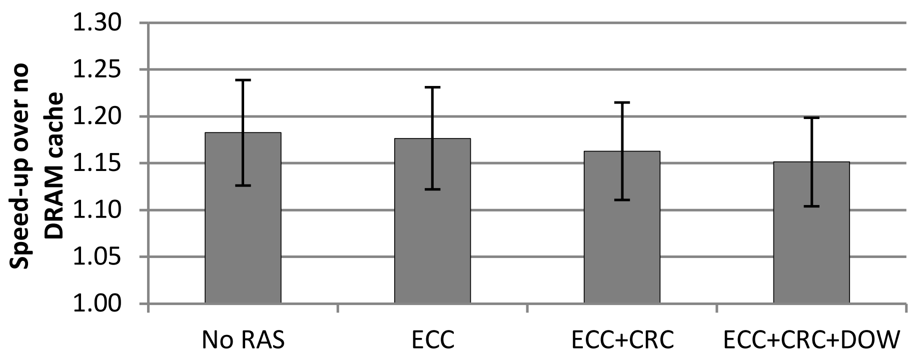


Figure 29: Average speed-up of no RAS, ECC+CRC and ECC+CRC+DOW over the 210 workloads.

4.6.7 Value of Configurable RAS Levels

The proposed protection schemes are just a few possible implementations of RAS support for DRAM caches. Other approaches are possible, from providing only light-weight parity-based error-detection, to very robust multi-bit error correction (e.g., BCH or Reed-Solomon codes). The more sophisticated schemes may require higher overheads, which reduce the overall data storage capacity or available bandwidth, but the high-level approach described here (e.g., embedding tags, ECC, and data in the same DRAM row) allows the designer to make the appropriate performance-versus-resiliency trade-offs.

At the implementation level, the DRAM cache is just a commodity memory component with no knowledge of how the data storage is being used. This allows system designers (OEMs) to take a single processor (with stacked DRAM) design but configure different levels of protection for different deployments of the design. In a commodity consumer system, one may choose to turn off ECC entirely and make use of the full DRAM cache capacity. In servers and certain embedded systems, basic SECDED ECC may be sufficient; this is comparable to the level of protection used today (e.g., SECDED ECC DIMMs). In mission-critical enterprise servers and HPC supercomputers, the full ECC+CRC and DOW protections could be used. The selection of which level of protection to use could be configured simply by a BIOS setting read during system boot (e.g., NVIDIA's Fermi GPU can enable/disable ECC in the GDDR5 memories with a reboot [59]).

Protection levels conceivably could be configured dynamically. Critical memory resources (e.g., operating system data structures) may receive a high level of protection, while other low-priority user applications may receive no or minimal protection. The level of protection could be reactive to the observed system error rates; e.g., a system may by default use only ECC+CRC fine-grain protection, but as the number of corrected errors increases beyond a certain threshold, DOW is enabled to prevent data loss. Likewise, this selective protection could be applied to only specific banks or even rows of the DRAM cache if elevated error rates are localized. The enabled protection level can be increased gradually as a product slowly suffers hardware failures from long-term wear-out [14, 69].

4.7 Summary

Stacked DRAM caches may play a significant role in attacking the Memory Wall [85] going forward, but the technology will be economically viable only if it can be deployed in sufficient volume. Only a minority segment of the market demands RAS support, so developing ECC-specific stacked DRAM chips is likely undesirable for memory manufacturers. This chapter presented a general approach for enabling high levels (well beyond current ECC schemes) as well as configurable levels of RAS that works within the constraints of commodity, non-ECC DRAM stacks. Beyond the benefit to memory vendors from having to support only a single (non-ECC) DRAM chip design, this approach also benefits system vendors (i.e., OEMs, ODMs) who can stock a single processor type and then deploy it in consumer systems with the stacked-DRAM ECC support disabled while using the same part in workstations and servers with RAS support turned on.

CHAPTER V

MANAGEMENT OF STACKED DRAM AS PART OF MEMORY

There are a number of ways to manage heterogeneous memory that is composed of fast stacked DRAM and slow off-chip memory. The most common approach is to manage the fast stacked DRAM as a hardware-managed cache, as discussed in the previous chapters. However, the cache approach may not be the right choice for the systems where the capacity of stacked DRAM is comparable to off-chip memory. In this chapter, we discuss another design choice that is to use large, fast, on-chip DRAM structures as part of memory (PoM) seamlessly through the hardware.

5.1 Stacked DRAM as Part of Main Memory

Generally, in the cache approach, allocating a block in the fast memory entails the duplication of the slow memory block into the fast memory. Although such duplication results in capacity loss, it makes block allocations simple and fast. With conventional cache size of a few megabytes per core (or tens of MBs per core as in today's eDRAMs), the opportunity costs of losing overall memory capacity to cache copies are insignificant. However, the integrated memory structures driven by die-stacking technology could provide hundreds of megabytes of memory capacity *per core*. Micron already offers 2GB Hybrid Memory Cube (HMC) samples [55], and by integrating multiple stacks on a 2.5D interposer, it is also plausible to integrate even tens of gigabytes of memory on package. For some computing environments, such as mobile or client systems, making the integrated memory invisible to overall system memory (i.e., used as a cache) could lead to a non-negligible loss of a performance opportunity. In these cases, both fast and slow memory may be combined into a single flat address space. We refer to this as a *PoM (Part-of-Memory)* architecture. In the simplest PoM architecture, a portion of the address space can be statically mapped to the fast memory while the remainder is mapped to the slow memory. To maximize the performance benefits of fast memory, the operating system (OS) could allocate heavily used pages to the portion of the physical address space mapped to the fast memory.

5.1.1 Dynamic PoM Management

The key advantage of the PoM architecture is the ability to minimize duplication; however, its performance may suffer relative to a simple cache. The PoM architecture is at a disadvantage for two reasons. First, the performance benefits of the fast memory will depend on the operating system's ability to identify frequently used portions of memory. Second, even if the most frequently used portions of memory can be successfully identified, a replacement policy that relies on frequency of use may underperform the typical cache recency based replacement algorithm [47].

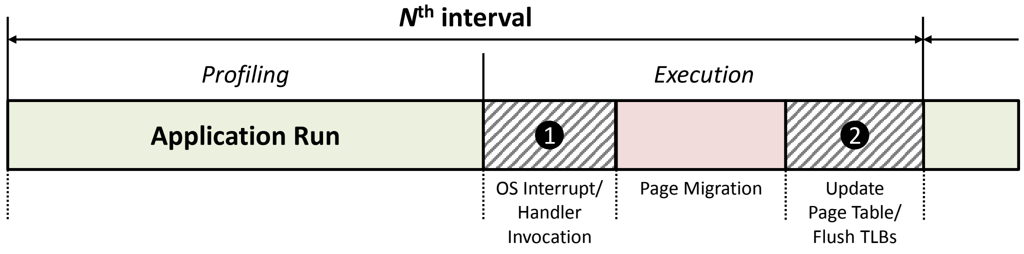


Figure 30: A high-level view of an OS-based PoM management.

Figure 30 shows an overview of an OS-based PoM management. At a high-level, such dynamic management consists of two phases of *profiling* and *execution*. At every interval, we first need to collect information that helps determine the pages to be mapped into fast memory during run-time (Application Run). The operating system generally has a limited ability to obtain such information; a reference bit in page tables is mostly the only available information, which provides a low resolution of a page activity. Thus, richer hardware support for profiling may be desirable even for an OS-based PoM management. A typical way of profiling such as used in the work of Loh et al. [52] has hardware counters associated with every active page and increments the counter for the corresponding page on a last-level cache (LLC) miss.[1] The profiled data is then used to perform page allocations for the next interval during the *execution* phase. In an OS-based management, the *execution* is costly since it involves an OS interrupt/handler invocation, counter sorting, page table modification, and TLB flushing in addition to the *actual* page allocation cost. As such, the OS-involved execution must be *infrequent* and thus often fails to exploit the full benefits of fast memory.

5.1.2 Potential of Hardware-Managed PoM Architecture

By managing the PoM architecture without an involvement of the operating system, we can eliminate the overhead of an OS interrupt/handler invocation (❶ in Figure 30) in the execution phase. More importantly, we do not need to wait for an OS quantum in order to execute page allocations, so the *execution* can happen at any rate. Therefore, the conventional approach of (long) interval-based profiling and execution is likely to be a less effective solution in the hardware-managed PoM architecture. To see how much benefits we can approximately expect by exploiting recency, we vary the interval size in a frequency-based dynamic mechanism similar to described in Section 5.1.1.

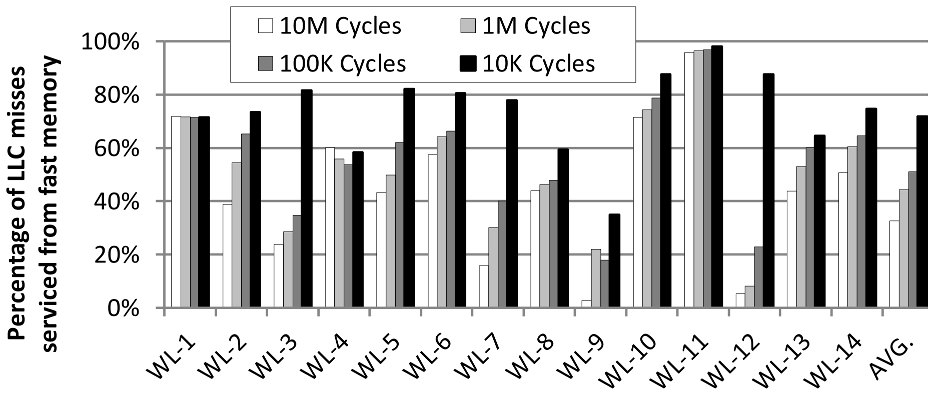


Figure 31: Percentage of LLC misses serviced from fast memory across different intervals.

[1]Note that only LLC misses are of interest when it comes to page allocations between fast and slow memory.

Figure 31 presents the percentage of LLC misses serviced from fast memory while varying the interval size from 10M cycles to 10K cycles (see Section 5.4 for our methodology). As the interval size decreases, the service rate from fast memory significantly increases for many workloads. This implies that we could miss many opportunities for a performance improvement if a tracking/replacement mechanism in the PoM architecture fails to capture pages that are highly utilized for short periods of time. The potential of the hardware-managed PoM architecture could be exploited only when we effectively deal with the cost of hardware management, which we describe in the next section.

5.2 Challenges of Hardware-Managed PoM

The high-level approach of *profiling* and *execution* remains the same in the hardware-managed PoM architecture. However, the hardware-managed PoM architecture introduces the following new challenges.

5.2.1 Hardware-Managed Indirection

Dynamic PoM management performs relocating pages into the memory space that is different from what OS originally allocated to, thus the hardware-managed PoM must take responsibility for maintaining the integrity of the operating system's view of memory. There are two ways this could be achieved. First, PoM could migrate memory regions at the OS page granularity, update the page tables, and flush TLBs to reflect the new locations of the migrated pages. Unfortunately, this method is likely infeasible in many architectures since it would require the availability of all the virtual addresses that map to the migrating physical page in order to look up and modify the corresponding page table entries. In addition, the OS page granularity could be too coarse-grained for migration, and updating page tables and flushing TLBs (❷ in Figure 30) still need to be infrequent since they are expensive to perform, thereby leading to lack of adaptation to program phase changes. Therefore, using this method in the hardware-managed PoM is unattractive.

The other approach is to maintain an indirection table that stores such new mapping information and to *remap* memory requests targeting the pages that have been relocated into the non-original memory space. The remapping table, however, could be too large to fit in on-die SRAM storage. For example, 2GB of fast memory managed as 2KB segments[2] will require a remapping table consisting of 1M entries *at least* to support the cases where all the segments in the fast memory have been brought in from slow memory. Note, in this approach, that every memory request that missed in the LLC must access the remapping table to determine where to fetch the requested data (i.e., whether to fetch from the original OS-allocated address or from the hardware-remapped address). Thus, in addition to the concern of providing such large storage on-chip, the additional latency of the indirection layer would be unmanageable with a single, large remapping table, which is another critical problem.

To overcome the problem of a single, large remapping table, we propose a PoM architecture with *two-level* indirection in which the large remapping table is embedded into fast memory, while only a small number of remapping entries are cached into an on-die SRAM structure. Although the two-level indirection may make the PoM architecture more feasible in practice, naively designing the remapping table makes the caching idea less effective. Section 5.3 describes the remapping table design that is suitable for such caching yet highly area-efficient.

5.2.2 Swapping Overhead

A key distinction between PoM and cache architectures is the need to *swap* a segment to bring it to fast memory, rather than just *copy* a memory block when allocating it to the cache. PoM is also different from an exclusive cache hierarchy since caches are backed up by memory (i.e., a clean block in any level of an exclusive cache hierarchy is also available in memory). Conversely, only one instance of each segment exists in PoM, either in slow

[2]We use the term *segment* to refer to the management granularity in our PoM architecture.

or in fast memory.

Swapping a segment is different from allocating a cache block because the segment allocated to fast memory replaces another segment that occupied its new location, and the swapped-out segment needs to be written back to slow memory. Therefore, every allocation to fast memory requires a write-back of the evicted data to slow memory. This swapping overhead could be significant depending on the segment size and the width of the channel between fast and slow memory. A small segment size reduces the swapping cost of moving large blocks between large and slow memory, and provides more flexibility in the replacement policy. However, a small segment size such as 64B or 128B (typical in caches) reduces spatial locality, incurs a much higher storage overhead for the remapping table, and suffers from a higher access latency due to the large remapping table size. In Section 5.3, we explore a different design that provides a balance between minimizing swap overhead and remapping table size.

5.2.3 Memory Activity Tracking and Replacement

Providing efficient memory utilization tracking and swapping mechanisms specifically tailored to the hardware-managed PoM architecture is another major challenge. If we simply use the mechanism similar to that in Section 5.1.1, we need to maintain a counter per active page that could be as many as the number of page table entries in the worst case. In addition, due to the required large interval, each counter needs to have a large number of bits to correctly provide the access information at the end of each interval. For example, with a 4GB total memory with 2KB segments, we need to track as many as 2M entries; then, assuming that the size of each entry is 16 bits (in order not to be saturated during a long interval), the tracking structure itself requires 4MB storage. Having shorter intervals could help mitigate the storage overhead a bit by reducing the number of bits for each counter, but comparing all the counters for shorter intervals would greatly increase the latency and power overhead. Furthermore, the storage overhead would still be bounded to the number of page table entries, which may be undesirable for scalability.

To make a responsive allocation/deallocation decision with a low-cost tracking structure, we propose competing counter-based tracking and swapping for our PoM architecture in which a single counter is associated with multiple segments in fast and slow memory. Section 5.3.6 discusses how we reduce the number of counters as well as the size of each counter while providing *responsiveness*.

5.2.4 Objective and Requirements

The primary objective of this work is to efficiently enable the PoM architecture in heterogeneous memory systems. For this purpose, we need to address the previous challenges. The hardware-managed indirection needs to be fast and area-efficient. The swapping cost needs to be minimized. The memory utilization tracking structure also needs to be small in area while being designed to provide *responsive* swapping decisions. In the next section, we describe our PoM architecture and how it addresses these main challenges.

5.3 A Practical PoM Architecture

5.3.1 Design Overview

In a conventional system, a virtual address is translated to a physical address, which is then used to access DRAM. In contrast, our system *must* provide the ability to remap physical addresses in order to support the transparent swapping of memory blocks between fast and slow memory. Starting with the physical address retrieved from the page tables (Page Table Physical Address, *PTPA*), we must look up in a remapping table to determine the actual address of the data in memory (DRAM Physical Address, *DPA*). Unfortunately, as discussed in Section 5.2.1, such single-level indirection with a large remapping table not only has a non-negligible storage overhead but incurs a long latency penalty on every access.

Figure 32 presents the overview of our PoM architecture. One of the key design points

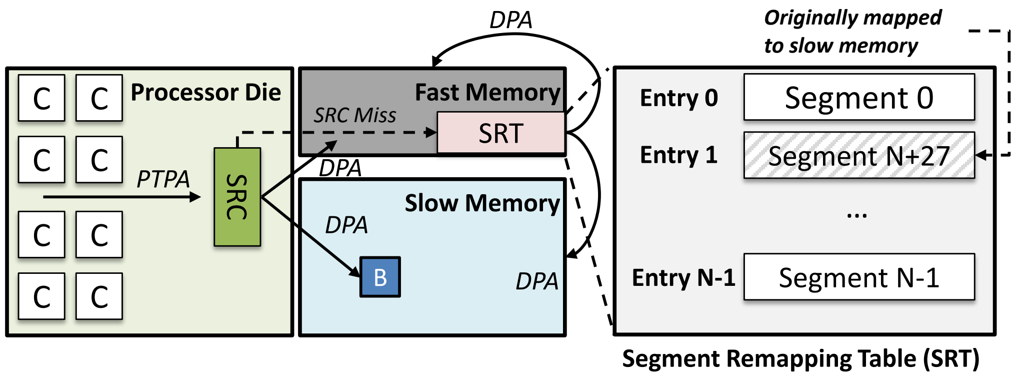


Figure 32: Overview of the PoM architecture.

in our PoM architecture is *two-level* indirection with a remapping cache. Each request to either slow or fast memory begins by looking for its remapping information in the segment remapping cache (SRC). If the segment remapping cache does not contain an appropriate remapping entry (*SRC Miss*), then the memory controller retrieves the remapping entry from the segment remapping table (SRT) located in fast memory and allocates it in the SRC. Once the remapping entry has been fetched, the location of the request (i.e., DPA) is known, and the data can be fetched.

At a minimum, a remapping entry needs to indicate which segment is currently located in *fast* memory. For example, with 4GB fast/16GB slow memory and 2KB segments, the maximum number of segments that fast memory can accommodate is 2048K (out of total 10M segments). In this configuration, the *minimum* number of remapping entries required for the SRT would be 2048K. With the minimal remapping table design, when a segment originally allocated to slow memory by the operating system is brought into one of the locations in fast memory, the corresponding remapping entry is modified to have new mapping information, such as Entry 1's ”Segment N+27” in Figure 32; then, ”Segment 1” is stored in the original OS-allocated location of ”Segment N+27”. Note that, even with the simplest design, the size of the remapping table is bounded to the number of segments in the fast memory, so the storage and latency overheads would still be high to use an on-chip SRAM structure for the remapping table. We discuss the implementation of the remapping table in more detail in Section 5.3.4.

5.3.2 Segment-Restricted Remapping

At first blush, it seems that we can simply cache some of the SRT entries. However, our caching idea may not be easily realized with the SRT described in the previous section due to a huge penalty on an SRC miss. On an SRC miss, we need to access fast memory to retrieve the remapping information. In the above SRT design, however, since the remapping information can be located anywhere (or nowhere) for a miss-invoked memory request, we may need to search all the SRT entries in the worst case, which could require thousands of memory requests to fast memory just for a single SRC miss.

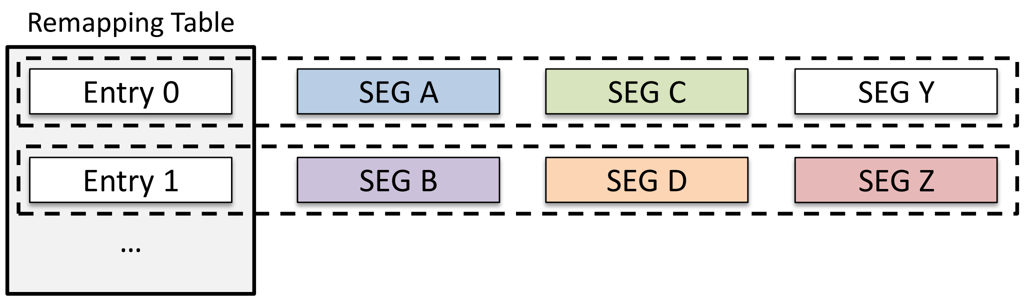


Figure 33: Segment-restricted remapping.

Our solution to restricting the SRC miss penalty within a single fast memory access is *segment-restricted* remapping, as shown in Figure 33. Each entry in the remapping table *owns* some number of segments where the number can be determined by the total number of segments over the number of SRT entries in the simplest case. A segment is restricted to be swapped only for the segments that are owned by the same entry. This is a similar concept to direct-mapping in cache designs (but is easily extensible to set-associative segment-restricted designs). For example, segments A, C and Y are only allowed to be swapped for the segments owned by Entry 0, whereas segments B, D and Z are swapped only for the segments owned by Entry 1. In this segment-restricted remapping, even if the remapping information for segment A is not found in the SRC, we can retrieve the remapping information with a single access to fast memory since it is only kept in Entry 0. To determine the SRT entry to which a segment is mapped, we simply use a few bits from the page table physical address (PTPA), which is good enough for our evaluated workloads.

5.3.3 Segment Allocation/Deallocation: Cache vs. PoM

In this section, we compare a cache allocation and the swap operation required by our PoM architecture. Throughout the examples, segments X, Y and Z are all mapped to the same location in fast memory (as in the segment-restricted remapping), and non-solid segments in slow memory represent the segments displaced from their original OS-allocated address.

First, Figure 34 illustrates a cache block allocation under two different conditions. In the example on the left (Clean), segment Z is brought into a local buffer on the CPU (❶) and simply overwrites segment Y (❷). Before Z is allocated, both fast memory and slow memory have identical copies of segment Y. As a result, allocating Z requires nothing more than overwriting Y with the contents of the newly allocated segment Z. The right example (Dirty) shows a case in which Y is modified and thus the copy of Y held in fast memory is unique. Allocating Z, in this case, requires reading Y from fast memory (❶) and Z from slow memory (❷) simultaneously. After storing them in buffers in the memory controller, Z is written back to fast memory (❸), and Y is written back to slow memory (❹).

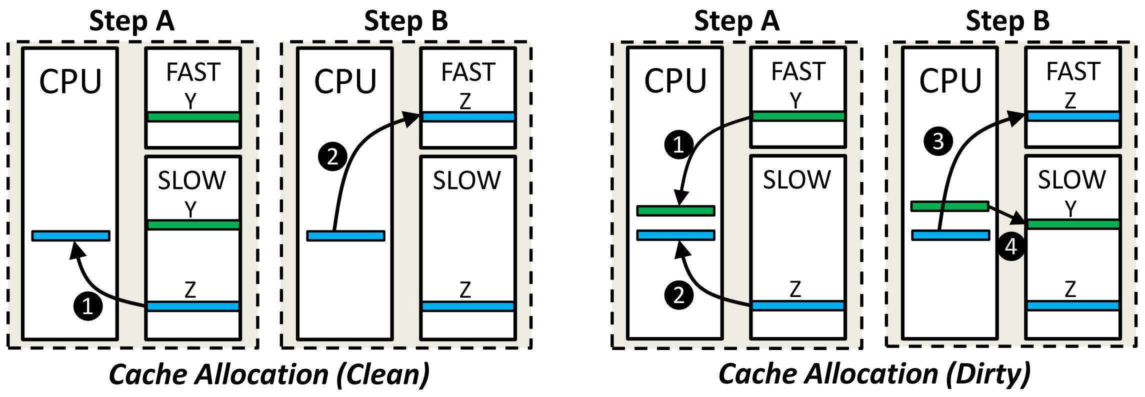


Figure 34: Cache allocation.

In contrast to the cache allocation, the PoM architecture makes all of both fast and slow memory available to the running software. To prevent duplicates and to ensure that all data is preserved as data is moved back and forth between fast and slow memory, PoM replaces the traditional cache allocation with a swap operation. The PoM swap operation, illustrated in Figure 35, differs depending on the contents of the segment displaced from fast memory. The PoM swap operation on the left (*PoM Fast Swap1*) occurs when the displaced segment X was originally allocated by the operating system to fast memory. In this case, a request to segment Z in slow memory requires segments X and Z to be read simultaneously (❶,❷) from fast and slow memory into on-chip buffers. The swap completes after copying Z from the on-chip buffer to fast memory (❸) and copying X from the buffer to slow memory (❹).

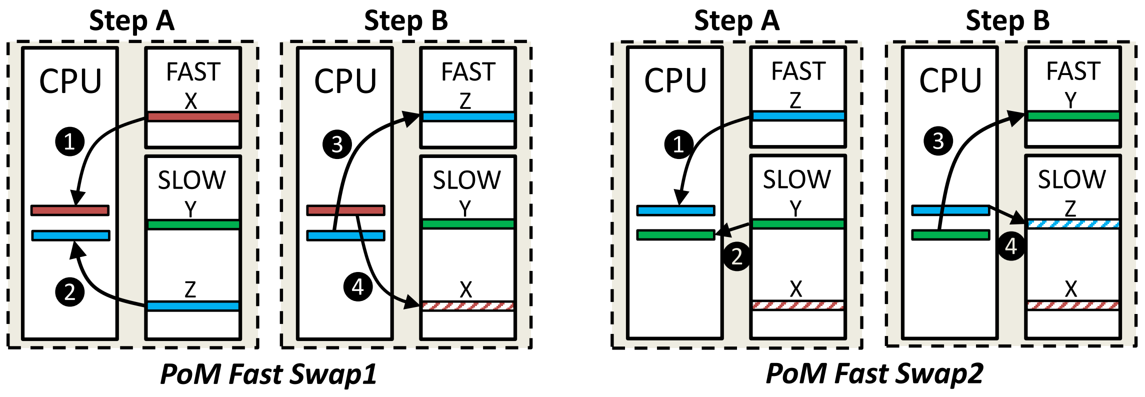


Figure 35: Fast swap operation in the PoM architecture.

As different segments from slow memory are swapped into fast memory, the straightforward swap operation previously described will result in segments migrating to different locations in slow memory, as illustrated in Figure 35 (*PoM Fast Swap2*). After swapping segments X and Z in swap1, a request to segment Y causes a second swap. The second swap (swap2) simply swaps segment Y with segment Z, resulting in segment Z assuming the position in slow memory that was originally allocated to segment Y. With more swaps, all slow memory segments could end up in locations different than their original location. This implies that the PoM remapping table must not only identify the current contents of fast memory, but must also track the current location of *all* segments in slow memory. Note that recording only the segment number brought into fast memory, as the remapping entry shown in Figure 32, would not allow this fast swap in most cases. The ability to support segment motion throughout slow memory adds complexity to the remapping table, but simplifies the swapping operation.

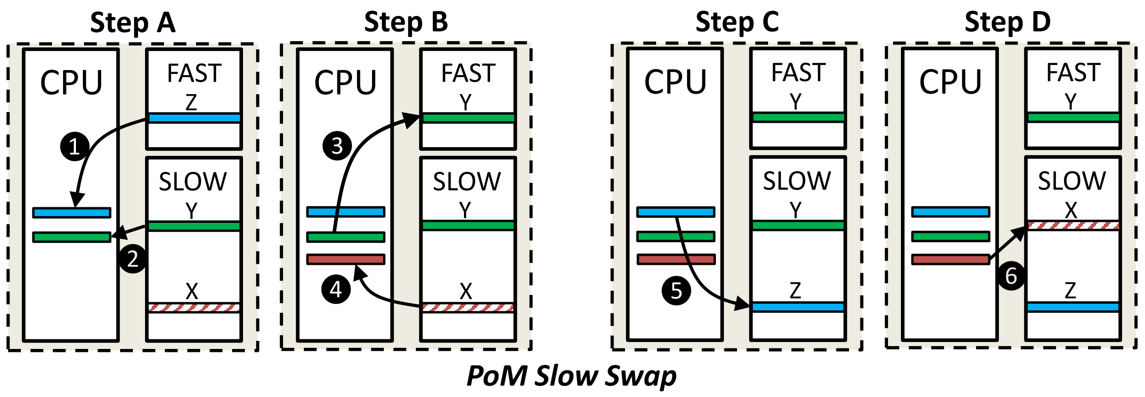


Figure 36: Slow swap operation in the PoM architecture.

An alternative approach to remapping segments requires segments to always return to their original position in slow memory. In this approach, the positions of all segments in slow memory can be inferred from their page table physical address, with the exception of segments currently mapped to fast memory. To ensure this, we can employ a second swapping algorithm depicted in Figure 36 (*PoM Slow Swap*). As in *PoM Fast Swap2*, a request to segment Y causes a swap with segment Z, currently in fast memory. In this case, however, rather than perform a simple swap between Z and Y, we restore Z to its original position in slow memory, currently occupied by X. We accomplish this in four steps: (A) Fetching Z and Y simultaneously (❶,❷); (B) Writing Y to fast memory (❸) and simultaneously fetching X from slow memory (❹); (C) Freeing X's location then writing Z back to its original location (❺); (D) Writing X to Y's previous location in slow memory (❻). The slow PoM swap generally requires twice as much time as the fast PoM swap with each of the four steps requiring the transfer of a segment either to or from slow memory.

5.3.4 Segment Remapping Table (SRT)

The segment remapping table (SRT) size depends on the swapping type we support. The PoM slow swap ensures that all data in slow memory is stored at its original location as indicated by its page table physical address. As a result, the SRT can include remapping information only for the segments in fast memory. Conversely, PoM fast swap allows data to migrate throughout slow memory; thus, the remapping table *must* indicate the location of each segment in slow memory. For instance, consider a system consisting of 1GB of fast memory and 4GB of slow memory divided up into 2KB segments. This system would require a remapping table with the ability to remap 512K (i.e., 1GB/2KB) segments if it implemented slow swaps, and 2048K (4GB/2KB) segments if it implemented fast swaps. The discussion in the remainder of this section will focus on a remapping table designed to support fast swaps.

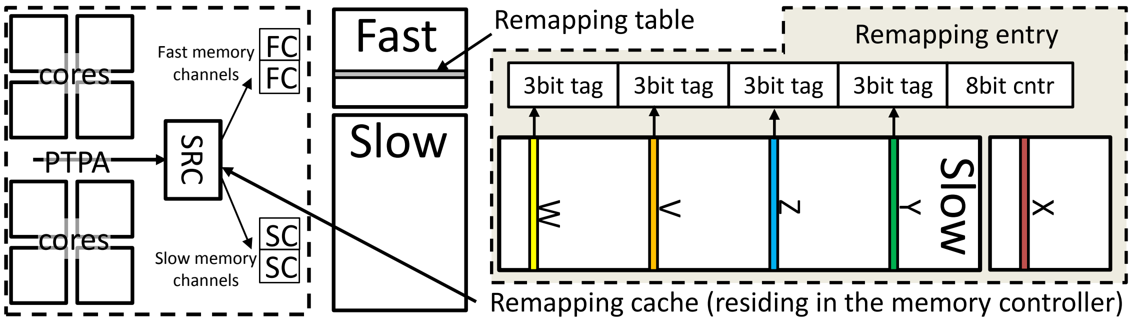


Figure 37: Remapping table organization.

In the case of fast swaps with the previous system configuration, a total of five possible segments compete for a single location in fast memory, and the other segments will reside in one of the four locations available in slow memory. The SRT needs to record which of the five possible segments currently resides in each of five possible locations. Figure 37 illustrates the organization of the remapping table with the five segments, V, W, X, Y and Z competing for a single location in fast memory. Each remapping entry in the SRT contains tags for four of the five segments; the contents of the fifth segment can be inferred from that of other segments. In addition to the four 3-bit tags, the remapping table contains a shared 8-bit counter used to determine when swapping should occur (Section 5.3.6). Co-locating the tags for conflicting segments has two advantages. First, since all swaps are performed between existing segments, this organization facilitates updates associated with the swap operation. Second, it facilitates the usage of the shared counter that measures the relative usage characteristics of the different segments competing for allocation in fast memory.

5.3.5 Segment Remapping Cache (SRC)

The remapping cache must be designed with two conflict objectives in mind. On the one hand, a desire to minimize misses provides an incentive to increase capacity and associativity. On the other hand, increasing capacity can increase access latency, which negatively impacts the performance of both hits and misses. To strike this balance, we choose a 32KB remapping cache with limited (4-way) associativity. On an SRC miss, we capture limited spatial locality by fetching both of the requested remapping entry and the second remapping entry that cover an aligned 4KB region. It is worth noting that with a protocol similar to DDR3, our fast memory will deliver 64B blocks. Although a single 64B block would contain tens of remapping entries, we found that SRC pollution introduced by allocating a large number of remapping entries outweighed the spatial locality we could harvest. Since we modeled 4KB OS pages, any spatial locality that existed beyond a 4KB region in the virtual address space could potentially have been destroyed after translation to the physical address space. It is also noted that an SRC hit for a given memory request does not guarantee that the requested data is found in fast memory.

5.3.6 Segment Activity Tracking

We previously discussed in Section 5.2.3 that a conventional segment tracking/replacement mechanism is not suitable for a hardware-managed PoM architecture, and a tracking mechanism for PoM needs to respond quickly with a low storage overhead (e.g., a small number of counters, fewer bits per counter). In this section, we discuss the tracking/replacement mechanism for our PoM architecture.

5.3.6.1 Competing Counter

To make a segment swapping decision, we need to compare the counter values of all involved segments at a decision point (e.g., sorting). Here, the information of interest is in fact the one relative to each other rather than the absolute access counts to each segment. For example, assume that one slot exists in fast memory with both segments A and Y competing for the slot, which is currently taken by segment Y. To decide which segment should reside in fast memory, we allocate a counter associated with a segment in fast memory (segment Y) and another segment in slow memory (segment A). During an application run, we decrement the associated counter on an access to segment Y, and increment it on an access to segment A. By having this *competing counter* (CC), we can assess which of the two segments has been accessed more during a certain period, which would be useful for swap decisions.

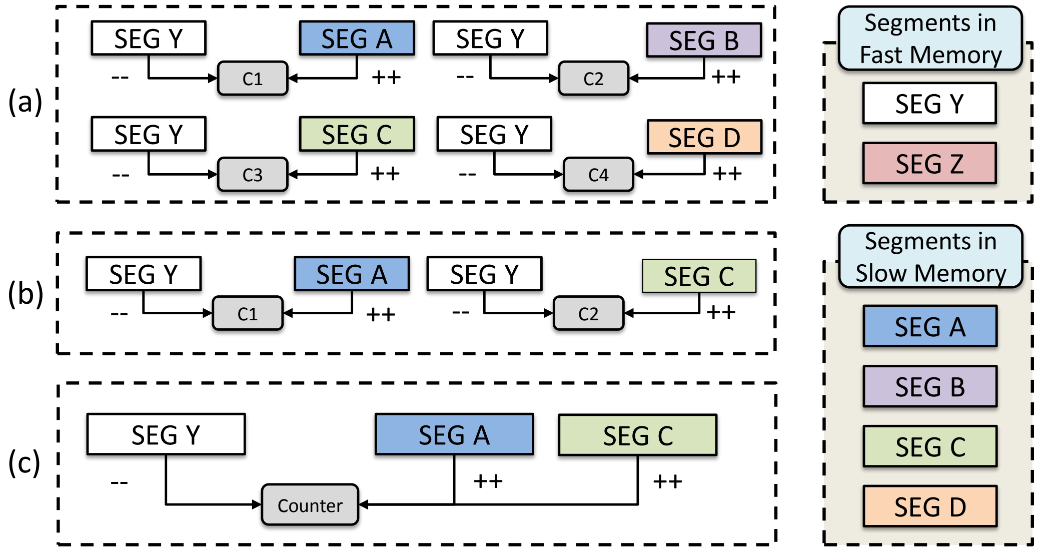


Figure 38: Competing counters.

Figure 38 illustrates a general case in which multiple slots exist in fast memory, while also a number of segments are competing for the slots. At first blush, the CC-based approach seems to incur high overhead since the competing counters need to be allocated to all combinations of the segments in fast and slow memory, as shown in Figure 38(a) (counters shown only for segment Y due to space constraints), if segments are allowed to be mapped into *any* location in fast memory. However, thanks to the segment-restricted remapping described in Section 5.3.2, the number of competing counters required is in fact bounded to the number of segments in slow memory, as shown in Figure 38(b). Although this already reduces the storage overhead compared to the tracking structure in Section 5.2.3 while providing more responsiveness, we can further reduce the storage overhead by *sharing* a single counter between competing segments, as shown in Figure 38(c). This reduces the number of counters to a single counter for each segment in fast memory. In this *shared* counter case, the segment that has just triggered swapping is chosen for allocation in fast memory.

Sharing the competing counters between competing segments provides us with two benefits. First, it reduces the overall memory capacity required by the segment remapping table (Section 5.3.4). Second, and more importantly, it reduces the size of each SRC entry by a little more than 50%, allowing us to effectively double its capacity. Furthermore, sharing counters between competing segments seems to have little to no effect on the performance of our replacement algorithm. Theoretically, references to segment A could have incremented the shared counter just below a threshold, and segment C could cause the counter to reach the allocation threshold and is chosen for allocation. In practice, however, we found this to be rare since the usage of different segments among competing segments tends to be highly asymmetric. Even though this rare case could happen, it is likely to have a temporary effect, and the highly-referenced segment would end up residing in fast memory soon afterwards.

5.3.6.2 Swapping Operation

Swapping occurs when the counter value is greater than a threshold, which implies that the segment currently residing in fast memory may not be the best one. For example in Figure 38(c), when an LLC miss request to segment A increments its associated counter, if the resulting counter value is greater than a threshold, segments A and Y will be swapped and their associated counter will reset.

An optimal threshold value would be different depending on the application due to the different nature of memory access patterns. To determine a suitable swapping rate for different applications, the PoM architecture samples memory regions. The locations in fast memory are grouped into 32 distinct regions in an *interleaving* fashion, and four regions are dedicated to sampling, while other 28 regions follow the threshold decision from sampling. The segments in the sampling regions modify the remapping table/cache when their counter values are greater than the assigned thresholds, but the actual swapping is *not* performed for the segments restricted to the sampling region. For the memory requests that target sampling regions, we simply get the data with static mapping without looking up in the remapping table (i.e., DRAM PA = Page Table PA). In order to drive the suitable swapping rate, we collect the following information for each sampling region:

- N_{static}: # of memory requests serviced from fast memory with static mapping
- N_{dynamic}: # of memory requests *expected* to be serviced from fast memory when swapping with a given threshold
- N_{swap}: # of *expected* swaps for a given threshold.

For each of four sampling regions, we then compute the expected benefit (B_{expected}) using Equation (2) and choose the threshold used in the sampling region that provides the highest *non-negative* B_{expected} value at every 10K LLC misses. In the case where such B_{expected} does not exist (i.e., all negative), the following regions do not perform any swapping operations.

$$B_{\text{expected}} = (N_{\text{dynamic}} - N_{\text{static}}) - K \times N_{\text{swap}} \tag{2}$$

N_{dynamic} is counted just by checking the remapping table for the requests to the segments in fast/slow memory dedicated to the sampling regions. N_{static} is counted when the requests target the segments originally assigned to the fast memory. K is the number of extra hits required for a swapped-in segment (over a swapped-out segment) to compensate the cost of a single swap. K differs depending on the fast and slow memory latency in heterogeneous memory systems. In our configuration (see Table 13), the cost of a single fast swap is about 1.2K cycles, and the difference in access latency between fast and slow memory is 72 cycles.[3] Thus, in general, the swapped-in segment needs to get at least 17 more (future) hits than the swapped-out segment for swapping to be valuable. K is computed in hardware at boot time. Note that the memory controller knows all the timing parameters in both fast and slow memory. In our evaluation, we use 1, 6, 18, and 48 for the thresholds in four sampling regions. We also use K as 20.

5.4 Experimental Methodology

Simulation Infrastructure: We use a Pin-based cycle-level x86 simulator [42] for our evaluations. We model die-stacked DRAM as on-chip fast memory, and we use the terms of fast memory and stacked memory interchangeably in our evaluations. The simulator is extended to provide detailed timing models for both slow and fast memory as well as to support virtual-to-physical mapping. We use a 128MB stacked DRAM and determine its timing parameters to provide the ratio of fast to slow memory latency similar to that in other stacked DRAM studies [15, 35, 36, 51, 62, 75, 76]. Table 13 shows the configurations used in this study.

Workloads: We use the SPEC CPU2006 benchmarks and sample one-half billion

[3](11 ACT + 11 CAS + 32$\times$4 bursts) $\times$ 4 (clock ratio of CPU to DRAM) = 600 cycles. Fast swapping requires two of these (Section 5.3.3).

Table 13: Baseline configuration used in this study.

CPU	
Core	4 cores, 3.2GHz out-of-order, 4 issue width, 256 ROB
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	8-way, private 256KB (8-cycle)
L3 cache	16-way, shared 4MB (4 tiles, 24-cycle)
SRC	4-way, 32KB (2-cycle), LRU replacement
Die-stacked DRAM	
Bus frequency	1.6GHz (DDR 3.2GHz), 128 bits per channel
Channels/Ranks/Banks	4/1/8, 2KB row buffer
tCAS-tRCD-tRP	8-8-8
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11

instructions using SimPoint [71]. Out of the benchmarks, we selected memory-intensive applications with high L3 misses per kilo instructions (MPKI) since other applications with low memory demands have very little sensitivity to different heterogeneous memory management policies. To ensure that our mechanism is not harmful for less memory-intensive applications, we also include two applications that show intermediate memory intensity. We select benchmarks to form rate-mode, where all cores run separate instances of the same applications, and multi-programmed workloads. Table 14 shows the 14 workloads evaluated for this study along with L3 MPKI of a single instance in each workload as well as the speedup of the all-stacked DRAM configuration where all the L3 miss requests are serviced from stacked DRAM instead of from off-chip DRAM. For each workload, we simulate 500 million cycles of execution and use weighted speedup [18, 79] as a performance metric.

5.5 *Experimental Evaluations*

5.5.1 Performance Results

Figure 39 shows the performance of our proposed scheme and a few other static/OS-based dynamic mappings for comparisons. We use a baseline where no stacked DRAM is employed, and all performance results are normalized to the baseline.

Table 14: Evaluated multi-programmed workloads.

Mix	Workloads	L3 MPKI (single)	All-stacked
WL-1	4 × mcf	71.48	1.88x
WL-2	4 × gcc	12.13	1.27x
WL-3	4 × libquantum	35.56	2.12x
WL-4	4 × omnetpp	7.30	1.47x
WL-5	4 × leslie3d	16.83	1.68x
WL-6	4 × soplex	31.56	1.81x
WL-7	4 × GemsFDTD	12.15	1.33x
WL-8	4 × lbm	32.83	3.37x
WL-9	4 × milc	18.01	1.75x
WL-10	4 × wrf	6.28	1.49x
WL-11	4 × sphinx3	11.89	1.52x
WL-12	4 × bwaves	19.07	2.00x
WL-13	mcf-lbm-libquantum-leslie3d	N/A	1.87x
WL-14	wrf-soplex-lbm-leslie3d	N/A	1.77x

First, `static(1:8)` and `static(1:4)` show the speedups when we assume that the capacity ratios of fast to slow memories are approximately 1:8 and 1:4, respectively. We assume that the OS would allocate memory pages such that one ninth or one fifth of the total pages are placed in fast memory for each workload. Static mapping results show the performance improvement due to having a part of memory that is accessed quickly without making any changes to hardware or OS page allocation policies. On average, we achieve a 7.5% (11.2%) speedup over the baseline with this 1:8 (1:4) static mapping.

Our proposed scheme achieves a 31.7% performance improvement over the baseline on average, and also shows substantial performance improvements over static mappings. Compared to `static(1:4)`, our scheme improves performance by 18.4% on average, and many of the evaluated workloads show huge speedups due to serving more requests from fast memory (see Section 5.5.2). On the other hand, a few other workloads show performance similar to that of `static(1:4)`. This happens because of one of two reasons. First, our dynamic scheme could determine that the cost of page swapping would outweigh its benefit for some workloads (e.g., WL-4 and WL-9), so the original page allocation by the OS remains the same in both fast and slow memories, and none (or only

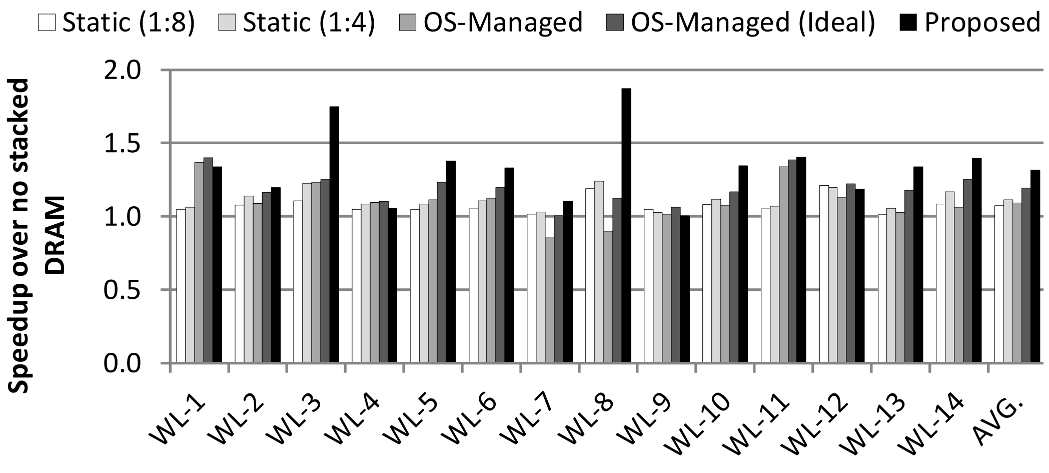


Figure 39: Speedup with our proposed mechanism compared to other schemes (normalized to no stacked DRAM).

a small number) of the pages are swapped in-between. Second, some workloads are not as sensitive to memory access latency as others, so their performance improvement due to our mechanism is limited by nature (e.g., WL-2 and WL-7).

Next, `OS-Managed` and `OS-Managed(Ideal)` are OS-based migrations *with* and *without* remapping overhead, respectively. We use a mechanism similar to that used in prior work [52]. In the mechanism, the OS first collects the number of accesses (LLC misses) to each 4KB OS page during an interval. Then, at the end of the interval, the most frequently accessed pages are mapped into the fast memory for the next interval. To mitigate remapping overheads, the mechanism uses a 100M cycle interval (31.25 ms on a 3.2GHz processor) and also does not select the pages with fewer than 64 LLC misses.[4] In addition, if selected pages for the next interval were already present in fast memory, the pages are not moved. On average, the OS-based migration achieve a 9.3% (19.1% without overhead) speedup over the baseline.

Compared to `OS-Managed`, our PoM achieves 20.5% performance improvement. In our proposed scheme, we start out with the same static page allocation policy as

[4]We have also performed experiments with shorter intervals and different thresholds, but they showed significantly lower performances since the remapping overheads were too large to amortize.

`static(1:4)`. However, we continuously adapt and remap data into fast memory during the workload runtime. Although `OS-Managed` does the same things, it only adapts to the working set changes at a coarse granularity, so the service rate from fast memory would be quite lower than our mechanism. As a result, even assuming zero-cost overheads, such as `OS-Managed(Ideal)`, the OS-based mechanism performs worse than ours.

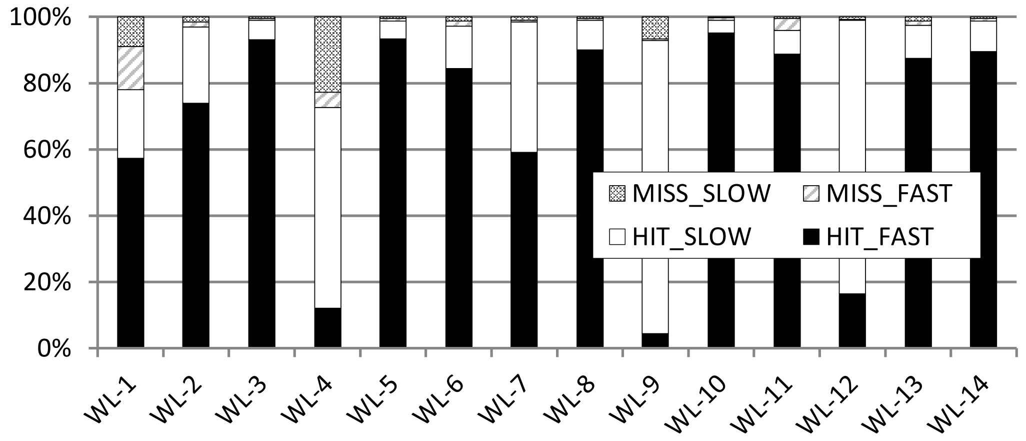


Figure 40: Physical address translation breakdown.

5.5.2 Effectiveness of Remapping Cache

The effectiveness of the remapping cache is very crucial to our two-level indirection scheme. Figure 40 shows the percentage of memory requests serviced from fast memory along with the source of the translation (i.e., whether translations are obtained from the remapping cache or the remapping table).

The HIT_FAST bar represents the percentage of requests whose translations hit in the remapping cache and the corresponding data are serviced from fast memory. The HIT_SLOW bar represents the percentage of requests whose translations hit in the remapping cache but are serviced from slow memory. MISS_FAST and MISS_SLOW represent remapping cache misses that are serviced from fast and slow memory, respectively. Note that the hit rate of the remapping cache is independent of swapping schemes since it is a function of an LLC-filtered memory access stream.

Our remapping cache is shown to be quite effective with more than 95% hit rate for most workloads. The high hit rate is due to the spatial locality in the lower levels of the memory hierarchy. WL-1 and WL-4 are the only benchmarks that show slightly lower hit rates due to the low spatial locality.

5.5.3 Sensitivity to Remapping Cache Size

One of the main performance drawbacks of remapping is that address translation latency is now added to the overall memory latency. For an effective design, most of translations need to hit in the remapping cache to minimize the cost of access serialization. In this subsection, we analyze the effectiveness of different sizes of the remapping cache. Figure 41 shows

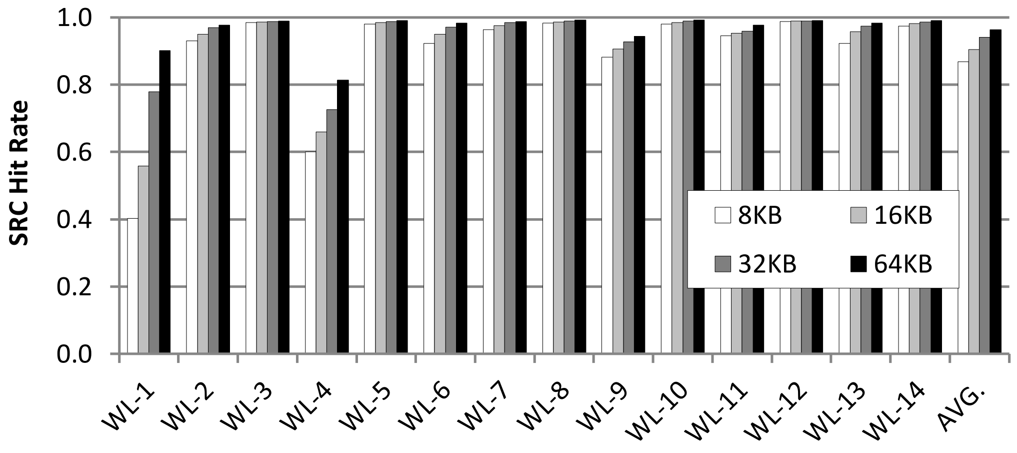


Figure 41: SRC hit rate across different cache size.

the hit rate of the remapping caches when we change the size of the remapping cache from 8KB to 64KB. Our workloads show high hit rates even with the 8KB remapping cache, but some workloads (e.g., WL-1) experience a non-negligible number of remapping cache misses with small size caches since their accesses are spread out across large memory regions.

5.5.4 Swapping Overhead

Figure 42 shows the swapping overhead of our proposed mechanism compared to the unrealistic *ideal* case in which swapping has no overhead; i.e., swapping does not generate any memory requests, and updates the swapping table automatically without using up any latency or bandwidth.

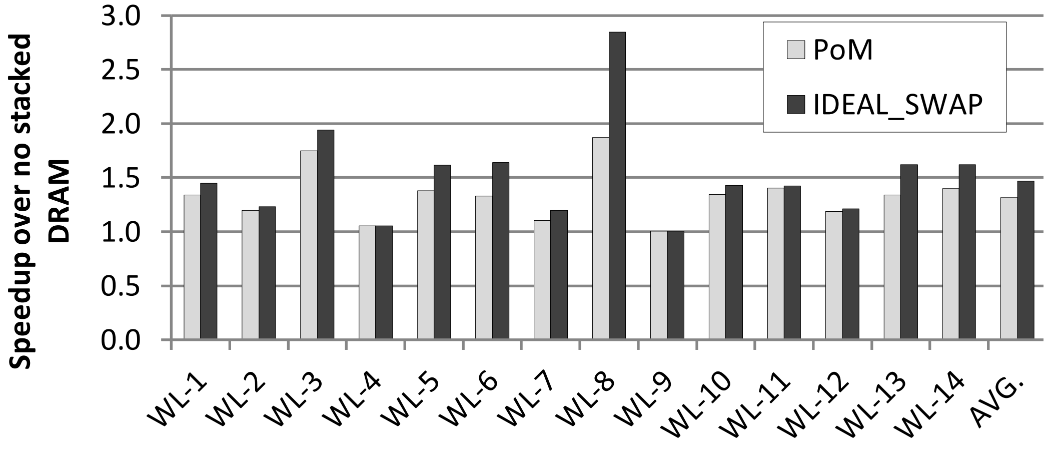


Figure 42: Comparison with ideal swapping.

Most of our evaluated workloads show reasonable swapping overhead since our mechanism enables swapping for small data segments (i.e., 2KB segments) and also attempts to avoid unnecessary swaps that are not predicted to improve performance. Some workloads such as WL-4 and WL-9 show no swapping overhead. This is because our dynamic scheme determines that the swapping would not be beneficial compared to static mapping as discussed in Section 5.5.1.

Intuitively, the performance cost of page swapping diminishes as the number of hits per swap increases. This can be achieved by swapping in more *guaranteed-to-be-hit* pages into fast memory. However, such a conservative decision is likely to reduce the amount of the memory requests serviced from fast memory. Thus, we first need to make a careful decision on when to swap in order to optimize the swapping cost without sacrificing the fast memory service rate.

The swapping overhead may also be partially hidden by performing swap operations in a more sophisticated fashion. In our design, we already defer the write-backs of *swapped-out* pages since they are likely to be non-critical. More aggressively, we may delay the entire execution of swapping operations until the memory bus is expected to be idle (long enough for swapping). We may also employ a mechanism similar to the critical block first technique; only the requested 64B block out of a 2KB segment is first swapped into fast memory, and other blocks are *gradually* swapped in when the bus is idle. These are good candidates to further alleviate the swapping cost although they need more sophisticated hardware structures and careful considerations on its operations.

5.5.5 Sensitivity to Swap Granularity

In our mechanism, we manage remapping at a granularity of 2KB segments. This is the same as the row size in the stacked memory. This section discusses the sensitivity of our proposed mechanism as the granularity varies from 128B to 4KB.

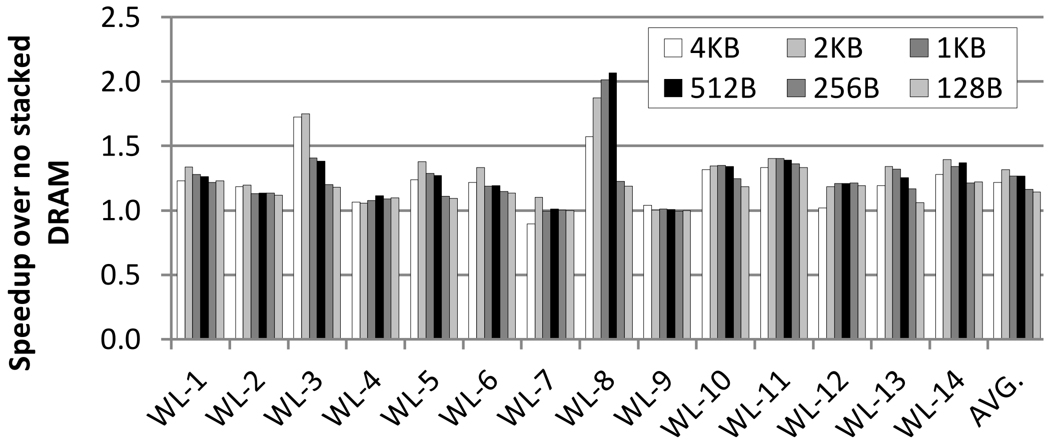


Figure 43: Speedup across different segment granularity.

Figure 43 shows the speedup across different segment granularity. Using a smaller segment size has the advantage of reducing the overhead of any individual swap. This allows for more frequent swaps and increases the rate of serving requests from fast memory. On the other hand, using a smaller segment size loses the benefits of prefetching that a larger granularity can provide. For the applications that have spatial locality, employing a larger swapping granularity may allow fast memory to service more requests, and the overall swapping overhead can also be smaller due to high row buffer hit rates. In addition, as the segment size decreases, the overhead of the remapping table increases since it needs to have more entries. This, in turn, reduces the effective coverage of the remapping cache. When deciding on the segment size, all these factors need to be considered. We chose a 2KB segment size since it achieved a decent speedup, while its overhead is still manageable.

5.5.6 Energy Comparison

Figure 44 shows the energy consumption of OS-managed and our proposed heterogeneous memory systems compared to no stacked DRAM. We compute the off-chip memory power based on the Micron Power Calculator [57] and the Micron 1Gb DDR3 DRAM data sheet (-125 speed grade) [56]. Since no stacked DRAM data sheet is publicly available, we compute the stacked memory power based on the access energy and standby power numbers reported in [22].[5] The results show that PoM reduces energy per memory access by an average of 9.1% over the evaluated OS-managed heterogeneous memory system. In general, PoM migrates more data blocks between fast and slow memories than OS-based migrations, which increases the amount of energy used for data migration. However, the increased energy could be amortized by the increased number of hits in fast memory. More importantly, the performance improvement by our PoM reduces static energy in heterogeneous memory systems, which leads to a significant amount of savings in energy.

5.5.7 Comparison to Hardware-Managed Cache

Figure 45 compares two different DRAM cache implementations with two different implementations of PoM including a naive version of PoM (`Unmanaged`) and our proposal. `LH-Style` uses 64B lines similar to [51] but includes improvements found in subsequent work such as a direct-mapped organization [62] and the hit-miss predictor discussed in Chapter III. Note that we do not use the self-balancing dispatch (SBD)

[5]Although our stacked memory is not identical to the one used in [22], we have verified that the conclusion remains the same across reasonably different power/energy numbers expected for die-stacked DRAM.

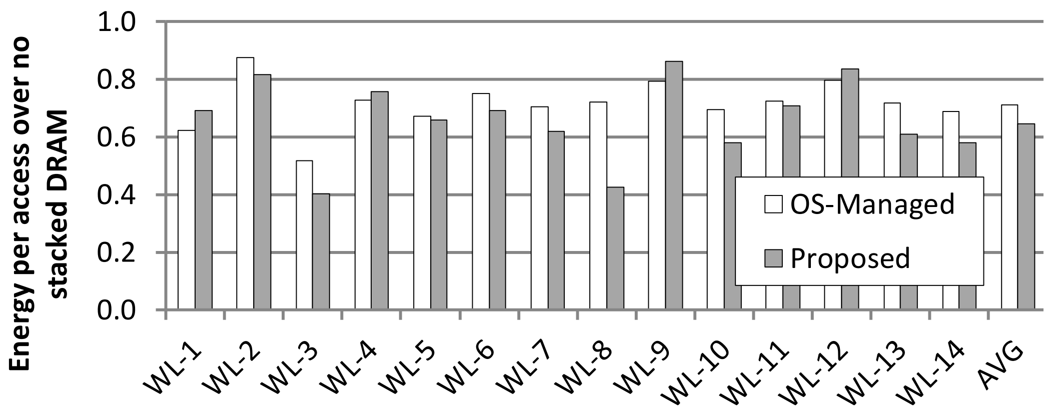


Figure 44: Energy per memory access of OS-managed management and our PoM.

mechanism presented in Chapter III for `LH-Style` to compare the impacts only due to the difference in organization between caches and PoM while excluding the benefits of improving bandwidth utilization from SBD. The PoM-Style cache uses 2KB cache lines and an improved replacement policy similar to our PoM proposal (Section 5.3.6). The performance benefits we observe for the `PoM-Style` cache result from the additional prefetching due to the large 2KB lines.[6]

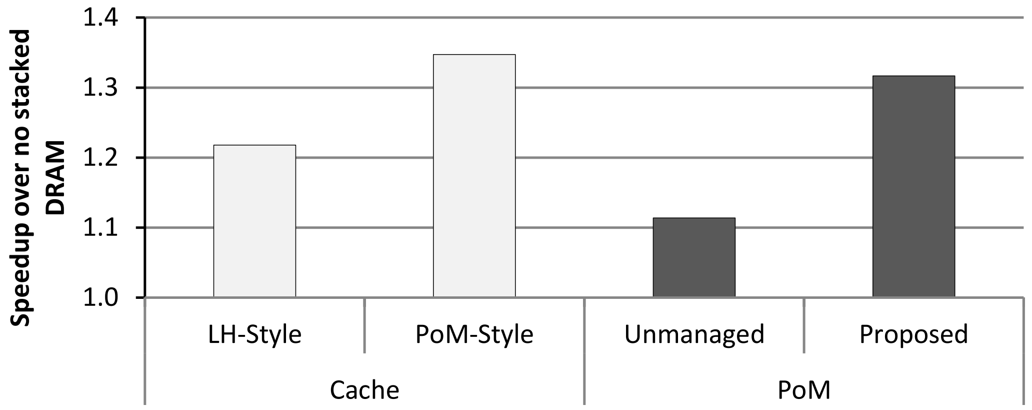


Figure 45: Comparison to hardware-managed caches.

The last two bars depict alternative implementations of PoM, both delivering the same capacity benefits. `Unmanaged` depicts a naive implementation of PoM without the benefits

[6]The LH-style cache can be managed similar to PoM using prefetching and bypass techniques, therefore providing better performance.

provided by the remapping cache and modified allocation algorithm (competing counters). The performance of our proposal is depicted on the far right. Since our experiments do not account for the performance impact of page faults, the best we can expect from our proposal is to match the performance benefits of the PoM style cache. In fact, our proposal delivers a speedup close to what is achieved with `PoM-Style`, and it does this while avoiding data duplication and allowing running software to use all available fast and slow memory.

5.5.8 Sensitivity to Fast to Slow Memory Capacity Ratio

Figure 46 shows the average speedup of static mapping and our proposed scheme over no fast memory across different ratios of fast to slow memory capacities. As the ratio becomes larger, the percentage of memory requests serviced from fast memory is likely to increase, and we observe a large performance improvement even with static mapping. Although, the potential of our dynamic scheme would decrease as the ratio of fast to slow memory capacity increases, our results show that the proposed scheme still leads to non-negligible performance improvements over static mapping (a 13.8% improvement for a 1:1 ratio) due to the ability to adapt workload phases as the working set changes. For smaller ratios, we achieve much higher speedups compared to static mapping (e.g., we achieve a speedup 20.0% over static mapping when the ratio is 1:8).

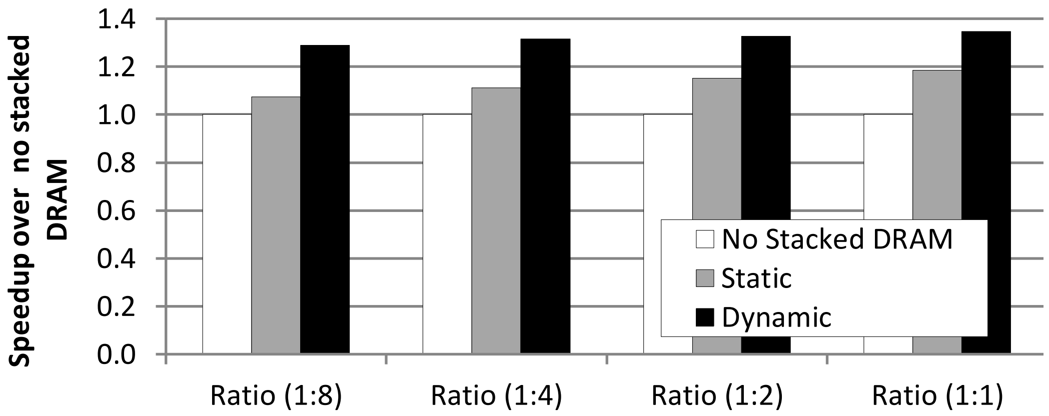


Figure 46: Speedups of static mapping and our scheme across different ratios of fast to slow memory.

5.5.9 Transparency to Virtual Memory Subsystem

PoM is transparent to current virtual memory/TLB subsystems and also ensures that a functionally correct memory image is maintained in the context of OS-invoked page migrations, copy-on-write, TLB shoot-downs, I/O requests, and so on. This is because in today's systems all memory requests (even from other devices such as disk and network) must go through the system-on-chip memory controller to access memory. In PoM, these requests must all lookup the SRC/SRT as they pass through the memory controller and before they access DRAM physical memory. For example, OS-invoked physical page migrations may result in page table updates and TLB shoot-downs, but since these involve the OS-maintained virtual-to-physical mappings and not the PoM-maintained physical page to segment mappings, these would be handled in a PoM system just like they would in a system without PoM. These events might make the physical segments allocated in stacked memory no longer hot (even dead); then, PoM would have no knowledge of the migration. However, since PoM uses a dynamic replacement algorithm and the competing counters are constantly comparing eviction and allocation candidates, the cold segments would quickly be replaced by hotter segments.

5.6 Summary

This chapter presented a Part-of-Memory (PoM) architecture that effectively combines slow and fast memory to create a single physical address space in an OS-transparent fashion. By employing two-level indirection with a remapping cache as well as competing counter-based tracking and swapping, the proposed PoM architecture provides substantial speedups over static mapping and alternative OS-based dynamic remapping. We expect that our PoM design can be best utilized for the computing environments where system's off-chip memory capacity is comparable to on-chip memory capacity as well as memory demands frequently exceed the off-chip memory capacity.

CHAPTER VI

STACKED DRAM INTERACTING WITH OPEARTING SYSTEMS

In the previous chapters, we have presented practical designs of using large, in-package memory structures for two primary use cases in today's computing systems: *caches* and *part of system memory*. Using die-stacked memory as a cache is a simple and efficient option, especially for the systems where off-chip memory capacity is much larger than die-stacked memory capacity. Alternatively, PoM has the key benefit to allow us to enjoy the full system memory capacity—with slightly more overheads in its implementation and operation than the cache approach—while keeping hot pages in fast stacked DRAM, which is useful when off-chip memory capacity is comparable to in-package memory.

However, for the systems where memory demands are mostly within off-chip DRAM capacity but occasionally exceeds it, neither PoM nor a stacked DRAM cache would be the best option. When system's off-chip memory is still available, PoM incurs unnecessary memory traffic due to swap operations and consumes off-chip memory bandwidth compared to the cache approach (similar to inclusion/exclusion in caches [74]). On the other hand, the cache approach makes gigabytes of stacked DRAM largely useless when system is under memory pressure after consuming off-chip memory. Ideally, we want to dynamically configure stacked DRAM depending on memory demands, with non-intrusive hardware or software changes, and the key idea behind this chapter is to make use of well-implemented today's virtual memory subsystems.

In this chapter, we propose a new use case for die-stacked DRAM. In the common case where system's off-chip physical memory is *not* over-committed, die-stacked DRAM operates as a cache to provide performance and energy benefits to the system. However, when the workload's active memory demands exceed the capacity of the off-chip physical memory, our scheme can dynamically convert the stacked DRAM cache into a fast swap device to avoid the We define the hardware-software interfaces needed to support the dynamic conversion of cache to swap (and back), provide algorithms to control the transitions to respond only to disk-thrashing scenarios (as opposed to non-performance critical paging out of cold data), and detail the DRAM cache indexing schemes that enable the graceful resizing of the cache/swap regions.

6.1 Stacked DRAM as Temporary Swap Space

6.1.1 Virtual Memory and Swap

The virtual memory (VM)[1] system offers each process the illusion of sole access to a large address space, even though the size of the address space may be larger than the total physical memory actually available on a machine. More so, the VM enables this for *every* process, where the aggregate addressable space certainly would exceed the physical memory capacity. To achieve this illusion, VM systems make use of a combination of physical memory and swap (typically implemented in the storage/file system). As physical memory is consumed, pages are swapped out to disk, and pages are brought back to memory when needed.

In modern operating systems (OS), the swap component can be implemented in a variety of ways. Typical instantiations include dedicated disk partitions for swap, as well as swap files that reside in a conventional file system. In modern Linux-based OSes, the VM system may use multiple swap devices, they may simultaneously be a combination of swap partitions and swap files, swap devices can be added and removed on-the-fly without a system reboot, and different priorities may be assigned to the different swap devices (so the OS preferentially pages virtual memory out to some devices before others) [7]. This all occurs in a way that is functionally transparent to applications (i.e., programmers need not do anything to their code to make use of the VM system).

[1]Throughout this work, we use the acronym "VM" (in capital letters) to refer to *only* virtual memory, and *not* to virtual machines.

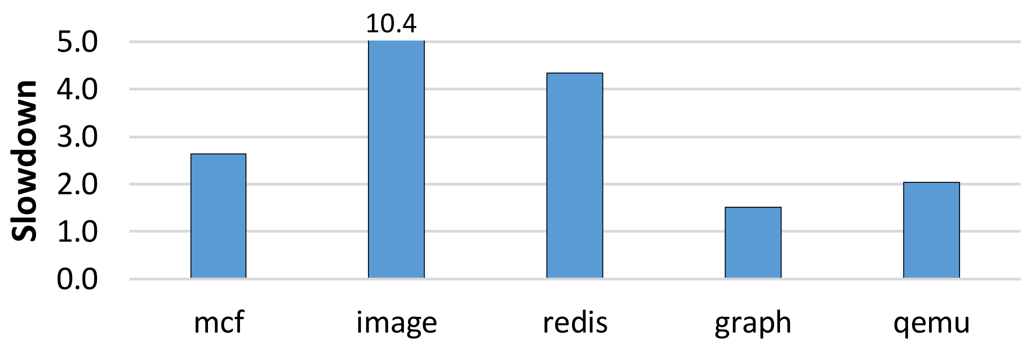


Figure 47: Slowdowns of applications running on an over-committed system.

The VM system enables processes to operate on virtual memory sizes exceeding a system's physical memory capacity, but even a low swapping frequency can lead to significant performance degradations due to very high disk latencies. Figure 47 shows the performance impact (measured on a real system) for several workloads running on a system with 12GB of DRAM, but where each workload's memory footprint exceeds 12GB by varying amounts (methodology and workload details are in Section 6.4). Each bar shows the *slowdown* in execution time compared to a system with 16GB of physical memory where there is effectively no swap-related disk activity.

As the swap subsystem is the primary performance limiter in an over-committed VM system (i.e., one where the collective memory capacity requirements of the active processes exceed the available physical memory, thereby causing significant swap activity), there have been enormous efforts to alleviate swapping overheads from both software and hardware perspectives. For example, modern OSes employ techniques such as pre-cleaning (evicting dirty pages earlier than new page allocations), clustering (avoiding random writes in the swap device), and page prefetching (reading in consecutive pages from swap space for a single page fault) [7]. Researchers have also proposed new VM systems to address swapping problems for garbage-collected languages [28, 86] and to use Flash/SSD as swap devices [3, 68]. We believe that this is the first work that examines opportunities to leverage die-stacked DRAM to help address VM swapping overheads.

6.1.2 Interaction of Stacked DRAM and Swapping

In the context of a system that may suffer from over-commitment of memory, we now examine how die-stacked DRAM may impact (or not) overall performance.

6.1.2.1 Stacked-DRAM Caches

While there have been many different DRAM cache organizations proposed in recent literature [35, 36, 51, 62, 75, 91], the basic function remains the same, which is to provide a large, very high bandwidth last-level cache to relieve the pressure from the off-package main memory in a software-transparent manner. When a system's physical memory is not over-committed, a DRAM cache can provide significant levels of speedup depending on the application characteristics and the exact DRAM cache organization. However, in an over-committed memory scenario, the DRAM cache is largely useless, as its benefits are completely dwarfed by the penalties of swapping to disk.

6.1.2.2 Stacked DRAM as NUMA

Another alternative usage model for stacked DRAM is to expose it as part of the system's physical memory in a non-uniform memory access (NUMA) style. There are many challenges and open research questions regarding how this would even be accomplished [52] in terms of the system and application software being able to efficiently manage two regions of memory with very different performance characteristics. However, assuming such an organization, one benefit is that it increases the total physical memory capacity of the system, which could prevent (or greatly reduce the frequency of) over-committing the system's memory. However, in the *common* case where the system is not over-committed, such a NUMA-like organization is unlikely to provide significant performance improvement unless additional research efforts on developing the necessary OS/runtime/compiler/profiling/programming language/application-level support have been made to take advantage of NUMA behavior.[2]

Table 15: Performance impacts of different stacked DRAM employment approaches under over-committed/under-committed memory scenarios.

System memory	**Stacked DRAM configuration**		
scenario (off-chip)	**Cache-only**	**NUMA**	**CacheSwap**
Under-committed	Best	Low-to-okay	Best
Over-committed (modest)	Poor	Fine	Fine
Over-committed (severe)	Very Poor	Poor	Poor
Application Transparent?	Yes	Unlikely	Yes

6.1.2.3 Stacked-DRAM Caches with Reconfiguration

Table 15 qualitatively compares the performance impacts of the above two stacked DRAM approaches and our dynamic reconfiguration approach (CacheSwap) under different scenarios. In the common case where system is under-committed, a traditional cache-only approach and CacheSwap provide the same good performance, while a NUMA (memory-only) approach would perform low-to-okay, depending on how well the software or the OS is able to manage the NUMA space. For a modestly over-committed scenario, cache-only would suffer tremendously, but CacheSwap and NUMA would perform fine by avoiding all or some of swap traffic. For severely over-committed situations (where the active memory usage *much* exceeds the sum of both stacked and off-package DRAM capacities), all three cases would perform poorly as there is simply not enough memory capacity to avoid excessive swapping. As none of the approaches can effectively address the last scenario, we restrict our focus to the first two scenarios (under-committed, and modestly over-committed) in the rest of this work.

Table 15 also summarizes the application-level impact of the different approaches. As mentioned earlier, NUMA-approaches will likely require application-level modifications

[2]It should be noted that conventional NUMA memory management algorithms (e.g., first-touch allocation) do not obviously extend to a die-stacked NUMA scenario. For example, first-touch allocation depends on the accessing thread having strong locality with the data it touches and the policy places that data in closer NUMA domains; however stacked DRAM is equally “close” to all cores, and the off-package memory is just as far from all cores, so first-touch would be ineffective for NUMA die-stacked DRAM.

to effectively manage data access locality between the different memories, which may limit the usefulness of the stacked DRAM, especially for legacy workloads. Caches are desirable as they require no application-level modifications. Virtual memory swap is also transparent to application programmers, and so CacheSwap leverages this to dynamically provide qualities of both cache and swap in a transparent manner.

6.1.3 Design Objectives

The high-level objective of our work is to devise a stacked DRAM mechanism that allows it to operate as a large last-level cache in the typical case where the system's memory is not over-committed, but also to allow the stacked DRAM's capacity to be leveraged to alleviate swap overhead in emergency (over-committed) scenarios. Along with the overall goal, we also want to keep hardware changes to a minimum, keep OS changes to a minimum, and require no modifications on the user-level applications. Given that future stacked-DRAM systems may support multiple GB of memory capacity, an additional objective is that any proposed scheme need not convert the *entire* stacked DRAM into swap (i.e., different regions of die-stacked DRAM can simultaneously operate in different modes), which introduces additional challenges.

6.2 CacheSwap: Swappable DRAM Cache

In this section, we explain the high-level operation of our proposed CacheSwap scheme, and then we delve into the implementation details of the OS and hardware components. Different indexing schemes that support stacked-DRAM reconfiguration will be discussed in Section 6.3.

6.2.1 Overview

The CacheSwap use of stacked DRAM is intended to operate primarily as a large LLC. Figure 48(a) shows an example system where the stacked DRAM is operated as a cache, and the OS uses disk to swap out pages when under memory pressure. For illustrative

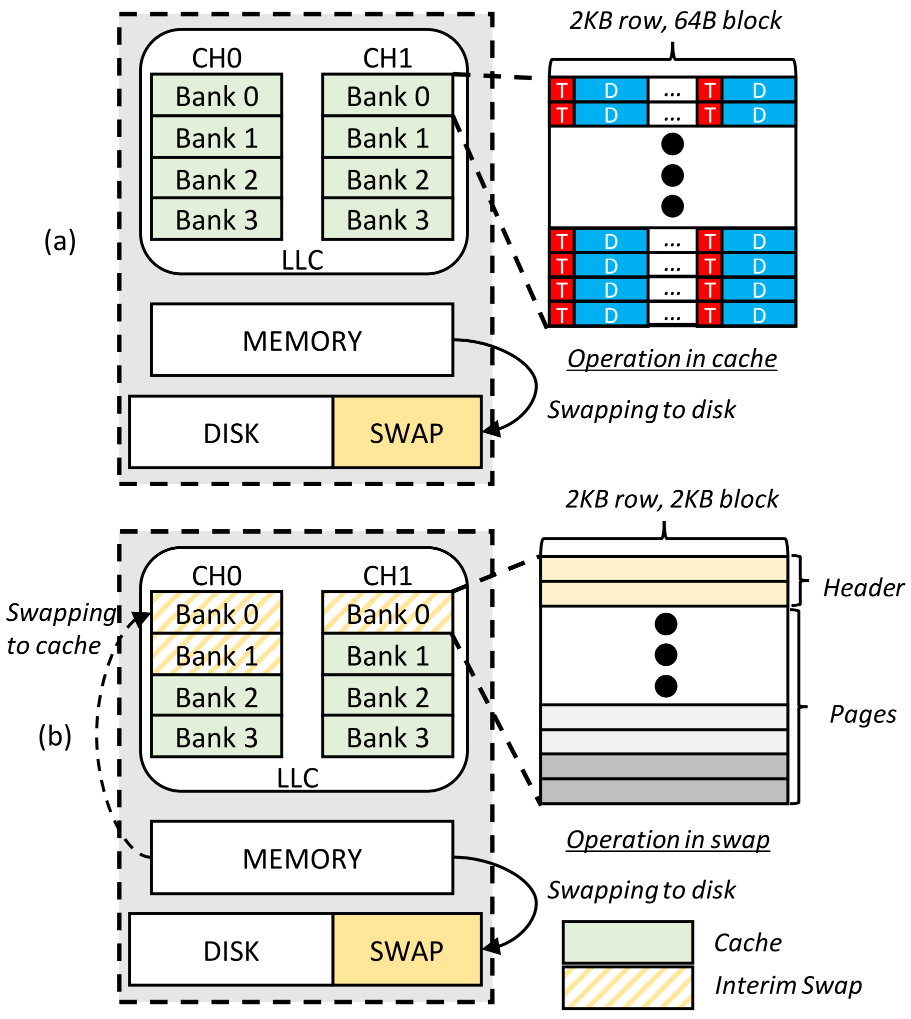


Figure 48: High-level overview of (a) a conventional memory system with a large DRAM last-level cache, and (b) a CacheSwap memory system that cannibalizes LLC capacity to provide interim swap devices.

purposes, the figure depicts the DRAM LLC organized as two channels with four banks each, where the cache's contents are laid out in a "tags-in-DRAM" style as proposed by several previous works [51, 62] (although the general framework of CacheSwap can be applied to any of the other previously proposed DRAM cache organizations, or to other types of large on-chip/off-chip caches in general, e.g., eDRAM). The right portion of Figure 48(a) shows an example DRAM bank's contents, where each 2KB row contains both cache tags (T) and data (D). That is, the tag and corresponding data blocks are self-contained in a row.

When the system's physical memory is over-committed, the large LLC does nothing to address the long disk I/O latencies associated with swap operations. CacheSwap dynamically cannibalizes the DRAM cache's capacity and transforms it into very fast swap, which we refer to as *interim swap*, thereby converting (at least a part of) the costly disk I/O time into significantly faster swap accesses to the stacked DRAM. Figure 48(b) shows the CacheSwap memory system operating in this mode where several banks of the LLC no longer provide cache capacity, but they are instead exposed to the OS as additional swap devices. Note that swapping to disk may still occur after using up all of the stacked DRAM capacity, or if the swapping is not an *emergency* case (i.e., reducing non-I/O time is still beneficial). Eventually when the memory capacity demands subside, the interim swap reverts back to cache space. We next describe the operational details.

6.2.2 Converting Cache Capacity to Interim Swap

Converting portions of the cache into interim swap involves three primary steps. The OS first interrupts the affected process(es), and issues a special command (HW/SW interface described below) to force the writeback of any modified cache lines to main memory for any LLC bank or banks that are to be converted to interim swap. The DRAM cache's controller must walk through each row in the affected bank(s), retrieve the tags, check the dirty bits, and issue a writeback request for any modified blocks. The latency to perform this operation depends the DRAM cache organization, the size of each bank, and the number of dirty blocks involved[3], but in general the latency is quickly amortized by the disk latencies that CacheSwap ends up avoiding. At the end of this operation, the LLC controller reconfigures its indexing to reflect the reduced number of banks (see Section 6.3), and then the OS is notified that this step has been completed.

Next, the OS initializes the cache bank(s) to act as swap. In Linux, this is done by creating a 4KB page-size header for a swap area as performed by `mkswap`. For the DRAM

[3]The dirty-data writeback latency can be reduced with previously proposed techniques (e.g., the Dirty Region Tracker [75]).

cache bank example, the 4KB swap header will be placed into the first two rows as shown in Figure 48(b).

Finally, the OS activates the interim swap device via a `swapon` system call and assigns the highest priority to it. In this way, any new pages that need to be swapped out of main memory will first use the interim swap from the stacked DRAM (by virtue of the interim swap having the highest priority). How the hardware exposes the stacked DRAM interim swap to the OS is discussed later in Section 6.2.5.

6.2.3 Reverting Interim Swap Back to Cache

When the OS detects that the interim swap is no longer needed (detection criteria described later), the OS disables the interim swap and allows the corresponding stacked DRAM banks to resume operating as part of the LLC. The OS first calls the `swapoff` system call to stop swapping to the affected interim swap devices. Because swapped-out pages may still exist in these interim swap devices (e.g., non-referenced pages), `swapoff` migrates any remaining content back into main memory to ensure that no data are lost. Then, the corresponding page table entries (PTEs) are corrected to point to relocated physical addresses in main memory. When pages are migrated back to conventional DRAM, TLB shootdown operations can be skipped as any existing PTEs from the affected pages would be invalid (i.e., they previously pointed to the interim swap) and would cause a page table walk to retrieve the new updated mapping.

After migrating all pages from the interim swap back into memory, the OS issues a special command (similar to the one originally used to convert the cache banks to interim swap) indicating to the hardware which bank(s) are now to be returned back to the LLC. The DRAM cache controller must walk through every row of the affected bank(s), and reset all of the cache tags' valid bits to indicate invalid cache lines. As an additional security precaution, it may also be desirable to zero-out the data fields of all cache lines. At the end of this operation, the DRAM controller reconfigures its indexing to reflect the increased number of banks (again, see Section 6.3), and then the OS is notified that this operation is complete.

6.2.4 When to Convert Cache to Interim Swap (and Back)

In our CacheSwap scheme, the OS bears the responsibility for deciding when DRAM cache resources should be converted to interim swap (and back). The OS monitors physical memory usage and swap activities for the existing swap devices (HDD and/or SSD). A simplistic approach would be to start converting DRAM cache to interim swap when the OS detects that the physical memory is full (e.g., observing page-out daemon activities or memory reclamation during page allocations in Linux) and enable the stacked-DRAM interim swap. However, it is not always effective to immediately cannibalize the DRAM cache as soon as the physical memory is full, because the DRAM cache capacity could quickly be entirely converted into swap without contributing much to the overall system performance. In addition, swapped-out pages could in fact be *inactive* and are not likely to be swapped-in for the time being (i.e., they belong to inactive applications). In such cases, we still probably want the victim pages to age out to conventional disk because moving them to the stacked DRAM would reduce the effective size of the DRAM cache without significantly reducing paging.

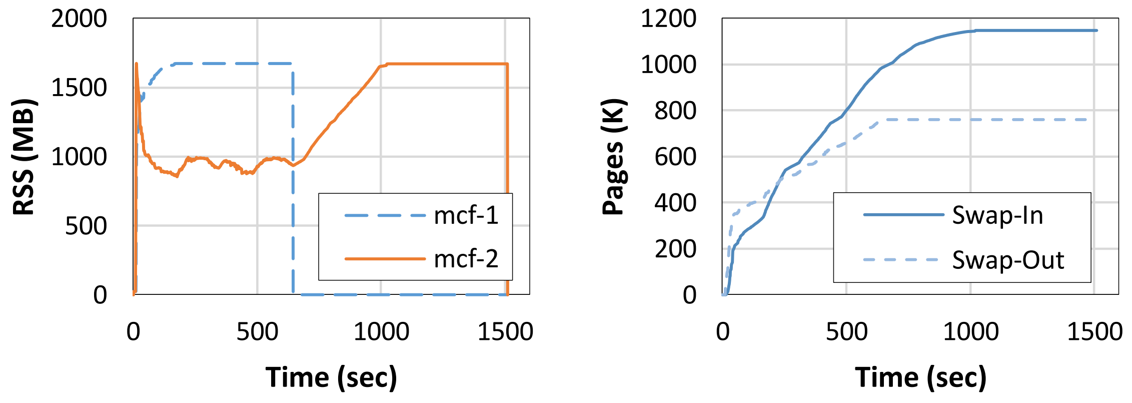


Figure 49: Resident set size of each process and (system-wide) swap-in and swap-out pages for two active mcf processes.

When to Enable Interim Swap: In this work, we convert cache to interim swap only when the memory system appears to be *thrashing* to disk. The OS determines whether the memory system is in an emergency by monitoring *both* swap-in and swap-out activities. For example, the Linux VM provides information on system-wide swap-in and swap-out counts since system boot. In our CacheSwap scheme, the OS simply starts conversion when the deltas of both statistics exceed pre-determined thresholds.

For example, Figure 49 illustrates the resident set size (RSS), which is the physical memory used by each process, and the number of swap-in/swap-out pages when two `mcf` processes run on a 3GB memory system. A single instance of `mcf` constantly consumes ∼1.7GB of physical memory; thus, as shown in Figure 49, "Swap-Out" begins to increase as the two processes contend for physical memory. In this case, `mcf-1` forces `mcf-2`'s pages to be swapped out while retaining its working set in physical memory.[4] Because `mcf-2` is also an active process, however, it references the swapped-out pages and brings them back into memory again. So, we can observe that "Swap-In" also increases in this high-contention situation. In contrast, a scenario where inactive pages are being aged out to disk by the OS will observe some amount of Swap-Out activity, but a much lower count for Swap-In events.

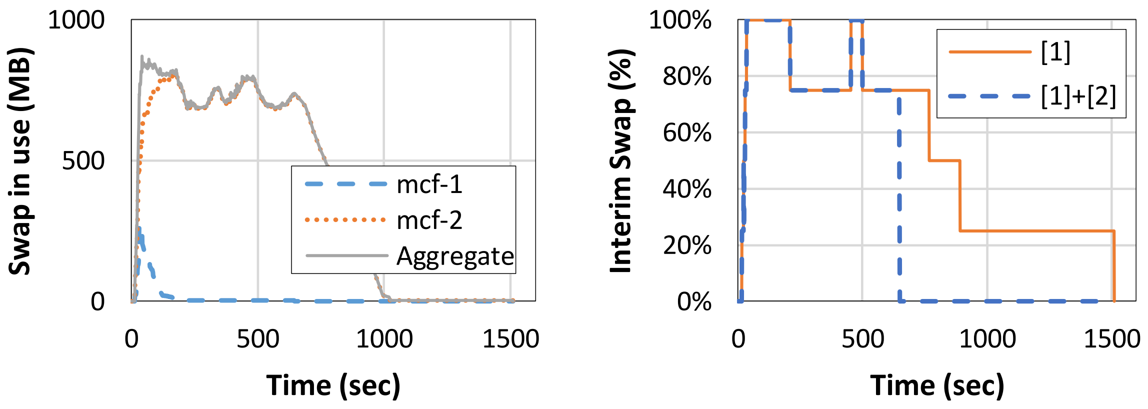


Figure 50: Swap space in use and the percentage of interim swap to total stacked DRAM size.

When to Convert Back: To determine when to return from the interim swap back to the

[4]Linux employs a global page replacement algorithm.

DRAM cache, the OS monitors the amount of swap space in use for each interim swap device (ISD). Then, interim swap devices are converted back one by one when one of the following conditions is satisfied.

$$\sum_{i \in \text{ISDs}} \text{SWAP}_{\text{in\_use}}(i) < (\text{\# of ISDs} - 1) \times \text{ISD}_{\text{capacity}} \tag{3}$$

$$\text{RSS}_{\text{sum}} + \sum_{i \in \text{ISDs}} \text{SWAP}_{\text{in\_use}}(i) < \text{DRAM}_{\text{capacity}} \tag{4}$$

Figure 50 shows the aggregate swap space in use for interim swap devices (left) and the percentage of interim swap to stacked DRAM (right) by Equation (3) and by Equations (3,4) for the dual-`mcf` workload. Equation (3) attempts to reduce the number of ISDs if interim swap devices are currently being allocated more than needed, which is determined by comparing the current swap usage in ISDs (LHS) and the total amount of ISD capacity when converting back an interim swap device (RHS). This performs well under memory pressure, but it could be too conservative when the system gets out of memory pressure. For example, although `mcf-1` finishes around 650 seconds and thus there is enough physical memory to accommodate all the pages from interim swap, Equation (3) loses some caching benefits due to the slow conversion. The figure also shows that the last interim swap device is not converted to cache until the end of `mcf-2` execution because it contains swapped-out (but not actively used) pages from `mcf-2`. Equation (4) helps expedite the conversion process by addressing such situations.

6.2.5 HW-SW Interfaces

Exposing Interim Swap to the OS: Exposing the interim swap banks to the system software can be achieved in multiple ways. One approach is to reserve a region of the physical address space that the OS knows about, that corresponds to activated interim swap banks. The OS can then directly issue loads/stores to this region to move data from/to the interim swap. If needed, existing mechanisms can be used to “wrap” the low-level stacked DRAM storage to present a more conventional interface to the OS’s swap system. For example, a “RAM disk” approach can convert a memory-mapped region of the stacked DRAM into a disk-like object that the OS can mount as a seemingly regular swap device.

Another approach is to extend the hardware so that it presents the interim swap to the software as block-based storage devices. In one possible implementation, the system IO controller (e.g., the IOMMU [2]) would have access to the stacked DRAM banks, and swap-related IO requests issued by the OS would then be delivered (with any necessary command conversion) to the stacked DRAM controller. This would require the direct integration of at least some part of the IO controller directly on the same chip as the processor, but this is increasingly common with integrated Northbridge [10] and integrated Southbridge [6] technologies in modern processors.

Command Interface for Configuring Interim Swap: The ISA interface between the OS and the processor to support the CacheSwap use of the stacked DRAM can be implemented in several ways. Below, we briefly explain a few, but do not specifically argue for one over another as the final choice would depend on the exact engineering costs to the processor and OS vendors.

At its most basic, the OS must communicate with the processor how much of the stacked DRAM should be converted to (or from) interim swap. This can be accomplished by providing a new kernel-mode instruction or by writing to a model-specific register (MSR). Depending on the granularity at which the stacked DRAM can be converted to interim swap (see Section 6.3), the interface could use an incremental approach (i.e., repeatedly add or remove one unit of interim swap at a time through multiple calls to the interface) or a more general approach that allows the OS to specify the exact allocation. The schemes that we present in this work only have a finite number of allowed stacked DRAM allocations between cache and interim swap capacity, and so the more general approach could be easily supported. While not strictly required, it would likely be useful for the OS if the hardware also provided a command (or MSR) that allows the OS to query the current cache vs. interim swap allocation of the stacked DRAM.

The processor must also be able to communicate back to the OS to indicate when the reconfiguration has been completed. The most straightforward mechanism is to make the reconfiguration instruction (or MSR modification) be a synchronous, blocking event. However, reconfiguration from cache to interim swap may require writing back all of the cache's contents in the worst case, which could take several milliseconds (e.g., $\sim$39ms for 1GB of stacked DRAM assuming it is written back across a 25.6GB/s DDR3 interface). Alternatively, the reconfiguration could be carried out asynchronously with the processor raising an interrupt upon completion. However, it is expected that reconfiguration occurs infrequently, so the latency of the reconfiguration operation does not have a substantial impact on overall performance, although it could still be of concern if the system has response time or quality of service requirements.

6.3 Cache vs. Interim Swap Sizing Options

When converting the die-stacked DRAM cache into interim swap, certain portions (e.g., banks) of the stacked DRAM are removed from being used as a cache. If not done properly, this could severely interfere with the cache's indexing functions that normally map a physical address to a stacked DRAM's row, bank, channel (and stack, if multiple stacks are employed). For example, with a direct-mapped DRAM cache organization [62], removing a DRAM cache bank to use as interim swap would leave a "hole" in the physical address-to-bank/row mapping. Either some addresses would no longer be cacheable (because their corresponding sets have been cannibalized for interim swap), or all of the sets of the cache would need to be reindexed using a new mapping function that avoided the missing bank(s). In this section, we cover different approaches to size the cache versus interim swap while minimizing overheads related to indexing (or reindexing) the cache portion.

6.3.1 All-or-Nothing

The simplest approach is to use an "all-or-nothing" allocation. Normally, the stacked DRAM is operated as a cache, but when the system enters an over-committed state, the *entire* stacked DRAM is converted to interim swap. This has the advantage that the cache's indexing is trivial (because there is *no* cache anymore), however, this is a very heavy-handed solution. First, it may be the case that only a fraction of the stacked DRAM's capacity is required to avoid catastrophic paging to disk, and so any remaining capacity should in theory still be useable as a cache to provide some performance benefit. This could be particularly problematic in future systems where the stacked DRAM capacity is on the order of several to over ten gigabytes. Second, all modified content in the entire DRAM cache must be written back to main memory. While only a fraction of the DRAM cache may be modified, the processor still must walk the entire cache to check all of the tags to find the modified data, although as discussed in Section 6.2, this may be a secondary issue if configuration of the stacked DRAM is infrequent.

6.3.2 Halving

A second approach incrementally halves the amount of stacked DRAM used for cache. When an over-committed scenario arises, the OS causes the DRAM cache's capacity to be halved. This could either be from removing half of the stacked DRAM channels from the cache's usage, or removing half of the individual stacked DRAM banks from the cache. The number of channels and the number of banks are typically a power of two (even if a direct-mapped DRAM cache uses a non-power-of-two indexing for the rows [62]), so the indexing modification to support halving amounts to simply ignoring one bit from the bank index.[5] This requires that the DRAM cache tags be wider than strictly needed if the DRAM cache were to never be resized, which should not be a problem as the tags-in-DRAM cache proposals typically have several left-over bits or even bytes in the tag blocks. To convert even more DRAM cache capacity to interim swap, the system can perform halving again which leaves the DRAM cache with only one quarter of the total stacked DRAM capacity (by omitting two bank index bits), and then an eighth, and so on.

[5]For the remainder of this section, we will discuss converting stacked DRAM resources at the bank granularity, but all of the techniques apply to stacked DRAM channels, and even a mix of channels and banks.

When halving, the DRAM cache contents in the half that is being converted to interim swap still needs to be scanned for modified content that needs to be written back. The overhead is approximately halved compared to the all-or-nothing approach. Modified lines actually have a choice to be inserted into their new sets in the still active half of the cache, or they could be written back to main memory (either choice maintains correctness). However, migrating modified lines to the remaining active DRAM cache may be faster; write back to main memory is potentially limited by the lower bandwidth of the off-chip DDR interface, whereas laterally migrating across stacked DRAM banks can still exploit the higher bandwidth of stacked DRAM. For the clean lines, the lines could simply be dropped/invalidated (a subsequent demand miss in the DRAM cache will cause the line to be installed into its new set in the remaining DRAM cache portion).

When converting the interim swap back into cache, cache lines that should originally have mapped to the cannibalized banks, but were displaced, need to be "repatriated" back to their corresponding banks in the enlarged DRAM cache. Modified lines *must* either be repatriated or written back to main memory to maintain correctness. Clean lines can either be repatriated back (increases interim swap-to-cache reconfiguration latency) or simply dropped/invalidated and refetched into their updated banks on the next demand misses (increases post-reconfiguration "cold" DRAM cache misses).

Halving provides more options compared to the all-or-nothing approach, which can maintain some of the DRAM cache performance benefits while simultaneously avoiding VM over-commit paging issues. However, due to the simple indexing used (i.e., ignoring one bit of bank index at a time), the smallest interim swap allocation is still *half* of the total stacked DRAM capacity, which could be many gigabytes for a future system with significant stacked DRAM resources.

6.3.3 Consistent Hashing

We propose a technique to selectively convert stacked DRAM cache banks to interim swap that allows for much finer-grained partitionings. The objectives are that (1) we can incrementally convert banks at a fine granularity, and (2) when a bank is converted from cache to interim swap, the bank's contents are redistributed to the remaining banks in a load-balanced manner. To achieve this, we take inspiration from the *consistent hashing* algorithm, which was originally proposed to uniformly and dynamically distribute load across a large number of web servers [40]. Cloud providers may have servers go down at any point in time (e.g., server crashes, scheduled maintenance); similarly, new servers may come online (e.g., machines rebooted, new machines deployed). The availability requirements of typical server deployments make it impractical to perform a complete re-indexing (e.g., re-sharding of an entire index of the World Wide Web) each time the pool of available servers changes.

6.3.3.1 Overview of the Original Algorithm

Consistent hashing maps addresses to banks indirectly by first hashing both the banks and addresses on to the same unit circle. Each address maps to the first available bank encountered in a clockwise walk of the unit circle starting from its own hash-value on the unit circle. As shown in Figure 51(a), address X maps to bank 2 because it is the first bank encountered in a clockwise walk (see solid-arrow). If bank 2 were to be removed from the DRAM cache to be used as interim swap, address X maps to the next bank in clockwise order, which is bank 3 (see dashed arrow). We define bank 3 to be the *failover* bank for bank 2 for address X. Note that in this example, all addresses that map into the *region* of the unit circle between banks 1 and 2 (e.g., X) map to bank 2, and subsequently fail over to bank 3 if bank 2 is disabled. Such a mapping achieves our first goal (only bank 2 is converted to interim swap), but not the second goal (bank 3 receives the entire failover load from bank 2).

To provide load balancing, consistent hashing creates multiple *virtual banks* for each physical bank, such that each bank is actually associated with multiple regions distributed around the unit circle. For example, Figure 51(b) shows three copies of each bank. An address maps to a bank if it maps to any of the virtual copies of the same bank. Thus, both X and Y map to bank 2. Because the virtual bank copies are permuted by the hashing function, there is not one single failover bank for all addresses in bank 2. For example, the failover bank for address Y is bank 1 whereas the failover bank for address X is bank 3. Using this concept, we next describe a consistent-hashing based indexing scheme for the DRAM cache to enable fine-grained conversion of DRAM banks to interim swap.

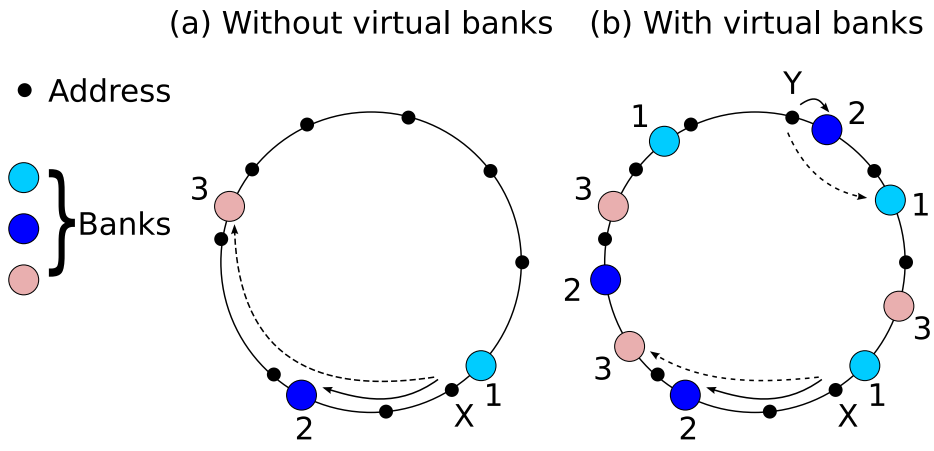


Figure 51: Overview of consistent hashing.

6.3.3.2 *Adapting Consistent Hashing for DRAM Cache*

The basic idea for our hardware implementation of consistent hashing is to use static permutations to remap banks. For example, consider a permutation {5,4,0,6,2,7,1,3}, which denotes that an address that mapped to bank 0 gets diverted to bank 5 (the first element of the sequence), addresses mapped to bank 1 get diverted to bank 4, bank 2 to bank 0, and so on. Now, if a bank is removed from the cache to serve as interim swap, then we simply fail over to the next active bank in the permutation list. So if bank 4 is used for interim swap, any accesses to bank 4 would fail over to bank 0 (i.e., the next bank listed in the permutation).

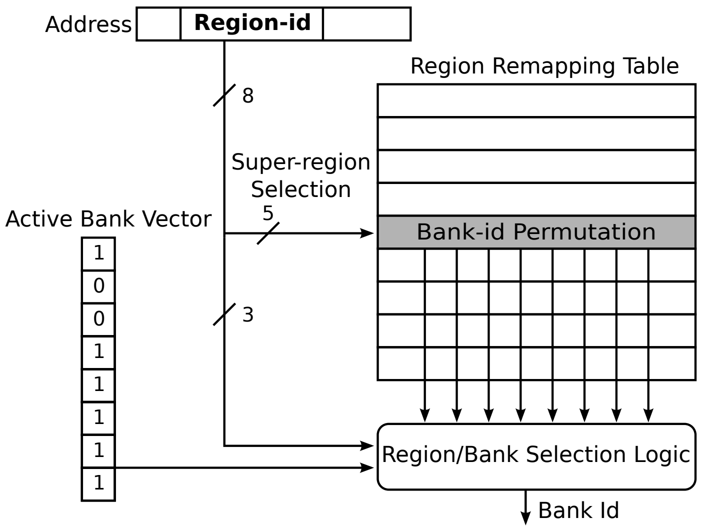


Figure 52: Consistent hashing-based bank selection for DRAM caches.

Now this has the same load-imbalance problem as the first example from Figure 51(a). To provide the same type of functionality as the virtual banks, we use a *region remapping table (RRT)*, with an example shown in Figure 52. The idea is that different portions of the address space select different RRT entries, where each entry records a different permutation. For example, if bank 1 is converted to interim swap, one address may map to one RRT entry where the permutation indicates a failover to bank 2, whereas a different address will use a different RRT entry with a different permutation resulting in a failover to bank 5. Given enough RRT entries with a unique (or sufficiently unique) set of permutations, load balancing will be achieved.

Each RRT corresponds to a slice of the physical address space that we call a "super region". Within each super region, all addresses use the same permutation/RRT entry. We then take the selected RRT entry, and also combine it with an *active bank vector* (ABV) that indicates which banks are currently available as cache (i.e., banks with a ABV bit of zero have been cannibalized for interim swap). The final bank mapping requires simple bit-mask logic and priority encoding to select the first RRT element corresponding to a non-zero ABV bit. Because RRT lookup is at hardware speeds, DRAM cache access latency remains comparable to traditional bank-selection latency, which typically uses simpler forms of hashing ranging from trivial bit-selection to bit "swizzling" techniques.

When converting a stacked DRAM bank to interim swap, we still must check all of the tags for modified data to be written back or migrated to their new destination (failover) banks. With our consistent hashing-based scheme, this will be significantly faster than the all-or-nothing and halving schemes. When converting a bank from interim swap back to cache, we similarly must write-back or repatriate blocks that have been displaced. One may need to to scan *all* of the remaining DRAM cache banks as the prior contents of the bank being converted have now been distributed across the active DRAM cache banks. Techniques to both bound the amount of dirty data as well as minimize the number of locations that need to be checked for dirty data would likely be needed [75].

In the following sections, we evaluate CacheSwap schemes using all three of the cache-interim swap re-indexing schemes.

6.4 Experimental Methodology

6.4.1 Evaluation Framework

We model an 8-core processor with a 1GB stacked DRAM cache and 12GB off-chip DRAM for our evaluations. Because we target full-program execution and disk-level accesses, we cannot directly use cycle-level simulation for our performance evaluations. Instead, we use an analytical model for overall program execution time (similar to the methodology in [29, 73], which is fed with inputs from real-system measurements.

Experimental System: We perform experiments in the Linux OS (kernel 3.2.0) running on a 2.26GHz x86 machine (2P quad-core system, 8 total cores) with 16GB DDR3 memory. For each workload, we first run the workload with the entire 16GB memory enabled, such that the system does not incur any major page faults related to swapping, and we measure the total execution time along with other statistics. We then reduce the amount of physical memory available to the OS to 12GB via Linux's GRUB boot loader and re-run the workload. When collecting data from the real hardware, we run with *thread affinity*.

The measured data are then provided to our analytical model which can compute execution times for different scenarios (e.g., no stacked DRAM, DRAM cache-only, CacheSwap, etc.). Linux occupies some physical memory by default, which makes about 11.6GB of physical memory available to the workload. We use this value as physical memory capacity in our model.

Workloads: We select five workloads that cover a range of multi-tenant shared-system scenarios, each of which makes the system's memory over-committed in the course of the execution. These workloads have a mixture of different over-committed samples (%) and over-committed memory sizes, which are the two most critical workload characterization parameters in these evaluations. It is noted that CacheSwap provides the same performance as the DRAM cache for the under-committed case; so, we do not choose any workloads with a footprint lower than physical memory capacity. Table 16 shows the evaluated workloads. The over-committed samples (%) is the percentage of samples whose memory demands are more than physical memory capacity. The maximum over-commit figure is the largest observed over-commit amount in our measured data. Below, we describe each of our scenarios.

- S1 (`mcf`) represents the scenario where each job has *static, constant* memory demands, and so users (or systems) know how many job instances should be run on the system but more jobs are accidently launched and make the system over-committed. `mcf` comprises eight instances of mcf from SPEC2006 [27]. We launch all instances at the same time and collect statistics until the last process finishes.

- S2 (`image`) is the scenario when users resize multiple images in the machine. Each job experiences *dynamic* memory demands, which could accidently over-commit system memory depending on the processing images. Similar to [68], `image` runs eight processes of an image converter (ImageMagicks), and each of which resizes an image by 500%. We configure each process to be single-threaded.

- S3 (`redis`) is the scenario when executing a mix of latency-sensitive and batch applications to increase the utilization of servers. In `redis`, two batch jobs are (occasionally) scheduled and cause dynamic memory pressure, while an in-memory, key-value store, called *redis* [64], is serving a total number of 5 million GET requests on a 10GB database.

- S4 (`graph`) is the scenario that could happen when a server performs graph analytics algorithms, in which case memory demands depend on graph inputs; some graphs could have a large number of nodes/vertices, and their memory demands could exceed available system memory. `graph` runs the PageRank algorithm using GraphLab [53]. We run eight instances, each of which works on 3M vertices using one core each.

- S5 (`qemu`) is the scenario where the memory system is *deliberately* over-committed in the virtual machine environment. `qemu` runs two QEMU [4] instances (PageRank/mcf processes are running inside them), each of which has 8GB of guest physical memory.

Table 16: Evaluated workloads.

Scenario	**Workload**	**Over-committed**	
		Samples (%)	Max Over-committed (MB)
S1	`mcf`	21.2%	1529.6
S2	`image`	51.6%	2494.6
S3	`redis`	29.0%	351.6
S4	`graph`	41.3%	258.2
S5	`qemu`	75.2%	708.6

6.4.2 Analytical Performance Model

To evaluate our proposal, we run each workload on the experimental system and collect *per-process* statistics including user time, kernel time, resident set size, the number of major page faults, and swap space usage by reading out `/proc/(pid)/stat` and `/proc/(pid)/status` every second (wall clock time).[6] We then use our analytical model to compute the execution time for *each* sample, and then we determine the overall execution time for the process by summing across all samples. Each workload consists of multiple processes, and so to compare performance between different schemes, we use weighted speedup [18, 79].

We focus our model on the first-order impact of the disk-related fault cost and cache/swap transition overhead, which will be the primary and secondary performance factors in our case. For each sample, we divide the execution time (T_{sample}) into three components: (1) T_{noswap} is the execution time when the system is not under memory pressure, (2) T_{swap} is the cost of swapping operations, and (3) O_{conv} is the overhead due to cache-to-swap or swap-to-cache conversion for the sample period, as shown in Equation (5).

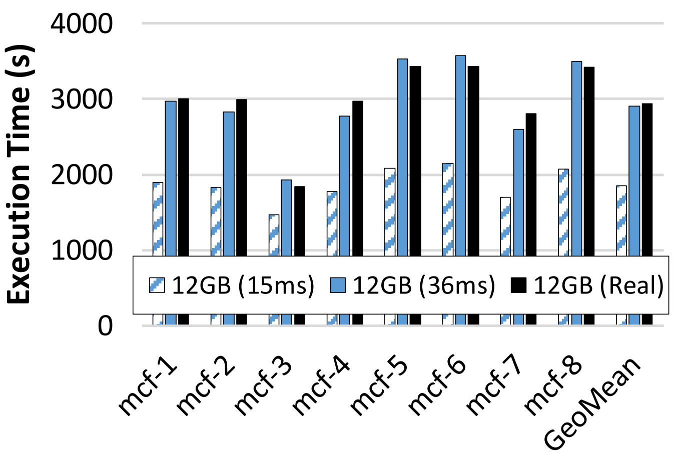
$$T_{\text{sample}} = \underbrace{\frac{T_{\text{base\_noswap}}}{\text{DRC}_{\text{speedup}}}}_{T_{\text{noswap}}} + \underbrace{N_{\text{faults}} \times \text{Latency}_{\text{avg}}}_{T_{\text{swap}}} + O_{\text{conv}} \tag{5}$$

$$\text{Latency}_{\text{avg}} = L_{\text{disk}} \times (1-p) + L_{\text{drc}} \times p \tag{6}$$

T_{noswap} is the fraction execution time when the process runs when the memory system is *under-committed*. When modeling a system employing a DRAM cache, this portion of execution time would experience a speedup by a factor of $\text{DRC}_{\text{speedup}}$. For our initial results, we use $\text{DRC}_{\text{speedup}}$=1.2 (i.e., a 20% performance improvement), and we later also conduct a sensitivity study on this parameter.[7] As the size of the DRAM cache is reduced due to interim swap cannibalization, the $\text{DRC}_{\text{speedup}}$ performance benefit is reduced by a square-root scaling (e.g., when the cache is reduced to half capacity, the performance benefit is $\sqrt{1/2} \cdot \text{DRC}_{\text{speedup}}$) [9]

[6] We use C++11's `sleep_for` for periodic sampling. It is noted that some sample intervals could be longer than a second (up to a few seconds) because disk I/O can delay the `sleep_for` call.

[7] A DRAM cache speedup of 1.2 is reasonably conservative compared to recent DRAM cache studies [35, 62]. We also simulated representative samples of one half billion instructions from each of our workloads on a cycle-level simulator and found DRAM cache speedups to be similar.



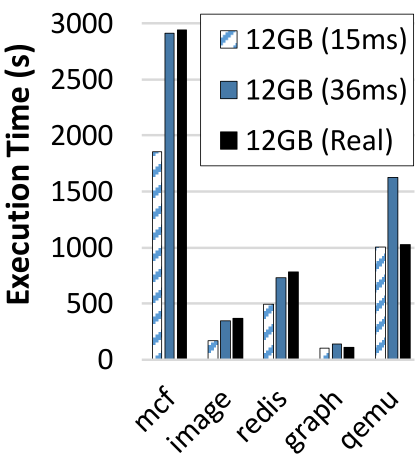


Figure 53: Comparison of the execution time on the real machine and the computed time for each workload.

T_{swap} is computed by multiplying the number of major faults related to swapping during the sample (N_{faults}) and the average page fault latency (Latency$_{\text{avg}}$). The factor p represents the portion of swap events serviced from interim swap devices (likewise, 1-p are the one serviced from conventional disk-based swap). We assume that if the interim swap capacity is larger than the over-commit amount, then p=1; in the baseline case where interim swap is not employed, then p=0; otherwise, we assume faults are serviced from interim swap and disk in proportion to their respective sizes. Latency$_{\text{avg}}$ is calculated using Equation (6), where L_{disk} and L_{drc} are the average page fault costs for swapping to disk and for swapping to the DRAM cache, respectively.

As mentioned in Section 6.1.1, the Linux VM system fetches multiple pages (default is 8) for a major page fault to amortize overheads. For L_{disk}, regardless of the number of pages swapped in, "seek time + rotational delay" and "queuing delay (which varies depending on the workload)" dominate the page fault cost. Figure 53 shows the execution time estimates produced by our analytical model when L_{disk} is set to 15ms (low-side), 36ms (high-side), and the actual measured execution time when running on real hardware (all with 12GB of physical memory enabled). For `mcf` experiencing high thrashing activity, using L_{disk}=36ms is a good match to the execution times across processes. At the same time, for some workloads, 36ms results in an over-estimation of the swap-related overheads.

As these workloads have different average page fault costs, we *conservatively* choose a constant L_{disk}=15ms for all workloads. This is a conservative assumption because the swap overheads will be higher for some workloads in which case CacheSwap would provide a greater performance improvement than what we report. We also evaluate the sensitivity of our results to the choice of L_{disk} in Section 6.5.3.

For L_{drc}, we assume that eight pages are always read in from the DRAM cache and are written back to memory in a serialized manner, which results in a $\sim$4.8μs latency in our system configuration.[8] We also consider the software cost associated with a major fault as it is not negligible for L_{drc}. We use the value reported in [38] (58.6μs), which results in L_{drc}=63.4μs.

It is noted that Figure 53 provides a good validation of our analytical model, showing that the estimated execution times (with L_{disk}=15ms) either match well or are conservative compared to the measured execution times from real hardware.

For conversion overhead, O_{conv} is computed at every sample by checking RSS and swap space usage across the processes. Depending on the condition (Section 6.2.4), either reconfiguration does not occur (i.e., O_{conv} is zero), or the DRAM cache is converted back from/to interim swap at a conversion granularity. For cache-to-swap conversion, we conservatively assume the worst case where all cache blocks in the banks are dirty and thus are written back into memory (i.e., O_{conv}= 39ms $\times$ Ratio$_{\text{conversion}}$).

6.5 Results and Analysis

6.5.1 Performance Under Memory Pressure

Figure 54 shows the performance of the CacheSwap system. We use a baseline where no stacked DRAM cache is employed and also compare against a configuration where the stacked DRAM is always configured as an LLC (DRC). Then we evaluate three versions

[8]Reading 4KB in from the stacked DRAM takes (8+8+32*2)*2 / 1.0GHz = 160ns, and writing 4KB out to the off-chip DRAM takes (9+9+32*4)*2 / 0.666GHz = 438ns.

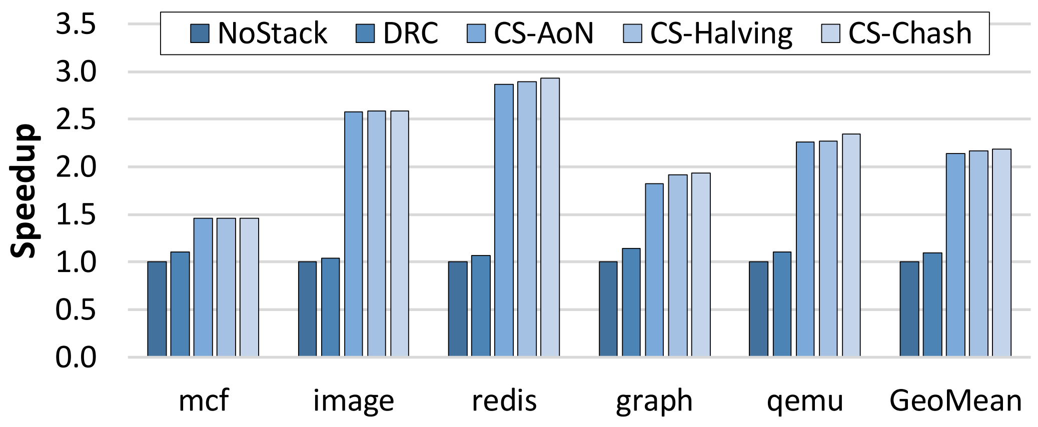


Figure 54: Performance of CacheSwap and other systems for comparison over no stacked DRAM cache.

of CacheSwap using an all-or-nothing cache-to-interim swap conversion scheme (CS-AoN), the halving scheme (CS-Halving), and the consistent-hashing-based scheme (CS-CHash). For these results, CS-CHash can convert the stacked DRAM into interim swap in increments of 1/8 of the stacked DRAM size (i.e., each interim swap device would be 128MB). All configurations assume 11.6GB of physical memory, and, except for the baseline, 1GB of stacked DRAM.

When the system exclusively uses the stacked DRAM as cache (DRC), the performance benefit is generally not very significant. Recall that under typical conditions (when memory is not over-committed), a DRAM cache offers significant performance benefits. However when over committed, it is not surprising that DRC offers no significant benefits because it does nothing to speed up swap activity. The DRAM cache does nothing to speed up the swap activity, so the caching benefits are only felt during the non-swap portions of workload's execution. Given the fraction of the over-committed samples in these over-committed scenarios (Table 16), this is expected.

The remaining three bars show the benefits of the different CacheSwap variants where the DRAM cache can be cannibalized for interim swap during periods of memory over-commitment. For workloads that over-commit the memory system by an amount that exceeds the stacked DRAM size (`mcf`,`image`), the interim swap conversion granularity is irrelevant because in all cases, the entire stacked DRAM would be converted to interim swap. These all show near-identical speedups. The speedups are still significant; for example, `mcf` over-commits memory by 1.5GB, but 1GB of that can be served with very low latency from the stacked DRAM, leaving only about 33% of the swapped-out pages to be served from the slower disk. Note that each instance of `mcf` has a working set size of 1.67GB, and so as soon as one instance completes execution, the remaining processes now fit within the physical memory and all 1GB worth of interim swap can be converted back to cache.

In contrast, other workloads are over-committed by smaller amounts amounts and the frequency of over-commitment is more dynamic in time as well (each instance of `mcf` uses about 1.67GB of memory throughout its entire execution). As a result, being able to convert only a fraction of the stacked DRAM to interim swap allows the remaining capacity to continue to provide benefit as a cache. While the performance difference between, for example CS-Halving and CS-CHash is not huge, we expect the ability to perform incremental interim swap conversion to be much more important as stacked DRAM capacities increase [32].

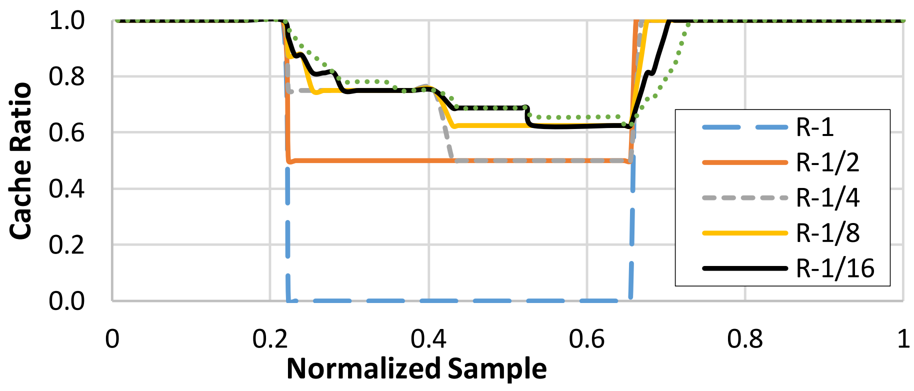


Figure 55: Cache portions of `graph` during the execution with different conversion granularities.

6.5.2 Sensitivity to Conversion Granularity

For the CS-CHash in the previous section, we allowed the stacked DRAM to be converted to interim swap in increments of 128MB (1/8 of the total capacity). However, other increments/granularities could be considered. We denote R-x to mean CS-CHash where the interim swap conversion granularity is x times the total stacked DRAM size. For example, R-1/4 can only use all, 3/4, 1/2, 1/4, or none of the stacked DRAM as cache. The R-1 case is equivalent to CS-AoN.

Figure 55 shows the fraction of the stacked DRAM that is used as cache throughout the execution of the `graph` workload. In the beginning of the program, the workload does not exceed the physical memory limit, and therefore the entirety of the stacked DRAM operates as cache (Cache Ratio = 1.0). About 20% of the way through, the workload starts to over-commit the physical memory; we can see that CacheSwap effectively responds to this by converting DRAM cache to interim swap (decreasing Cache Ratio). We can see that if the conversion granularity is set too coarsely, then much more of the stacked DRAM is converted to interim swap than is necessary. On the other extreme, setting the conversion size too small does not provide much additional benefit, and it also causes CacheSwap to adapt more slowly (shallower slopes in the figure) as DRAM cache capacity is converted to interim swap one increment at a time. Furthermore, as the interim swap increment decreases, the hardware overhead to implement the RRT increases (e.g., with R-1/32, each RRT entry would have to provide a 32-element permutation). R-1/8 appeared to provide a reasonable tradeoff that adapted well to the interim swap capacity demands while only requiring a very small 96-byte RRT (32 entries, each with eight 3-bit elements to encode an 8-element permutation).

6.5.3 Sensitivity to Fault Latency

We have so far used 15ms for the average page fault cost for our evaluations, which we showed was conservative for our results (Section 6.4). With higher disk latencies, the

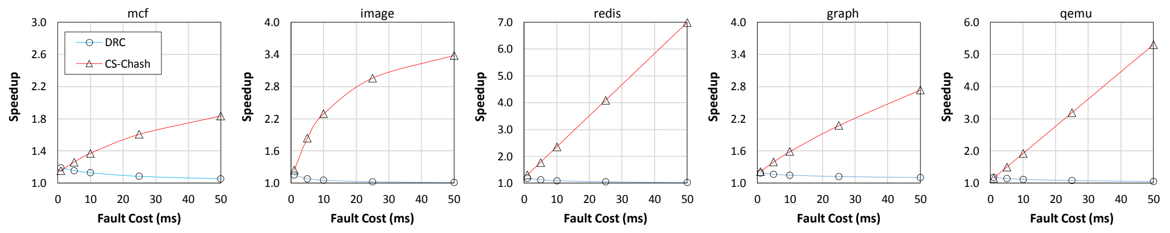


Figure 56: Sensitivity to page fault latency.

benefit of CacheSwap would be greater. We now analyze the sensitivity of CacheSwap's performance benefits with respect to different page fault costs. Figure 56 shows the speedups of DRC and CS-CHash over the baseline with no stacked DRAM when the fault costs are 1ms, 5ms, 10ms, 25ms, and 50ms. The results show that the benefit of CacheSwap could be significant for the systems that use hard disks as swap devices since its page fault cost would typically be more than 15ms under memory pressure. For example, the average page fault cost for the `mcf` is around 36ms, as discussed in Section 6.4.2, which makes the benefit of CS-CHash over DRC increase from 1.3x to 1.7x. If our model used a more accurate model of the disk bandwidth while taking into account the reduction in disk queuing latency, the speedup of CacheSwap would be even greater. If the servers move to using solid-state disks for storage/swap, then that would improve disk latency and naturally the benefit of reducing swap activity decreases proportionately.

As shown in the results, the sensitivity to fault costs varies across the workloads, which the OS could exploit to increase overall throughput for over-committed systems. For example, the OS could place swapped-out pages into fast or slow swap devices, depending on the sensitivity of each application; e.g., memory pages from more sensitive applications are swapped out to faster devices. The OS might also schedule I/O requests from the page-fault sensitive applications over the requests from less sensitive ones for the purpose. Such additional optimizations are left for future research.

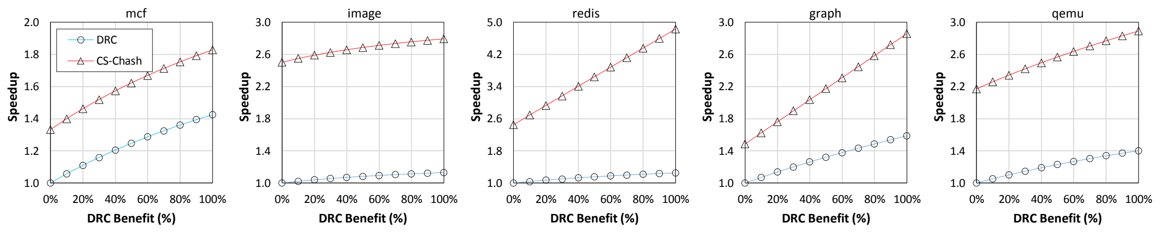


Figure 57: Sensitivity to cache performance.

6.5.4 Sensitivity to DRAM Cache Performance

To keep our analytical model simpler, we assumed a constant performance benefit from operating the stacked DRAM as a cache (20% in all experiments thus far). However, the actual benefit will vary by applications, the number of cores, the off-chip memory bandwidth, and other factors in the target system. So we now examine CacheSwap's sensitivity to the $\text{DRC}_{\text{speedup}}$ model parameter.

Figure 57 shows the speedups of CacheSwap and DRC over no DRAM cache across different values for $\text{DRC}_{\text{speedup}}$. The results show that larger $\text{DRC}_{\text{speedup}}$ values do not much contribute to the performance for over-committed systems when CacheSwap is not used (note that the slope of DRC would be one if the system was not over-committed). For over-committed systems, these results indicate that the top priority must be to reduce swap-related I/O overheads. Only after alleviating I/O overheads the system can have a chance to benefit from the DRAM cache. For example in `redis`, CacheSwap converts part of cache portions as swap and eliminates most of disk swapping operations, so the remaining cache portion could provide a visible speedup for the system. For some other workloads, the speedup is less sensitive to the $\text{DRC}_{\text{speedup}}$ value than `redis`. There are two primary scenarios where this occurs. For some workloads (e.g., `qemu`), the reduction in effective cache size remains long (e.g., 75.2% of sample periods for `qemu`) due to continuous memory pressure. For other workloads (e.g., `mcf`, `image`) CacheSwap is indeed effective at eliminating much of the swap overhead, but to do so, the entirety of the stacked DRAM capacity is used as interim swap. This leaves no room for the DRAM cache, which in turn renders the exact value of $\text{DRC}_{\text{speedup}}$ irrelevant during over-committed periods.

6.5.5 Impact of Cache Capacities on CacheSwap

The key idea of our work stems from the observation of the trend of having very large DRAM caches in future computer systems. In this section, we explore the possibilities of extending our proposal for smaller cache sizes (e.g., eDRAM or traditional SRAM).[9] Figure 58 shows the speedup of CacheSwap over DRC for the cache capacities from

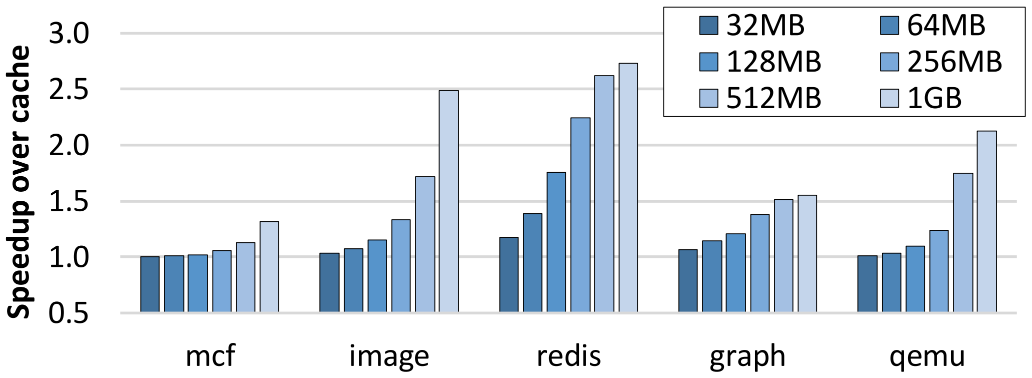


Figure 58: Speedup of CacheSwap over the cache-only use.

32MB to 1GB. We have shown that 1GB of stacked DRAM using the CacheSwap approach provides substantial benefits under memory pressure. With 32MB LLC, which is a typical cache capacity for today's high-end systems, CacheSwap is not particularly effective. The reason is that for our workloads that over-commit the physical memory by 258MB(`graph`) to 1.5GB (`mcf`), 32MB of additional fast swap only absorbs a small fraction of the overall swap activity. As the cache size increases to 128MB-256MB, the performance improvements become more substantial. This is within the realm of current eDRAM caches (e.g., the IBM Power8 processor [31], the Intel Haswell processor [25]). We therefore conclude that CacheSwap is definitely worthwhile for stacked DRAM, likely of benefit for large eDRAM caches, and not worth the effort for conventional SRAM-based LLCs.

[9] CacheSwap's benefits would only increase with *larger* cache sizes, so we do not show those results in the interest of space.

6.6 Summary

In this chapter, we explored a new usage model, CacheSwap, for die-stacked DRAM. Compare to other ways of managing stacked memory, CacheSwap provides performance benefits for both under-committed and over-committed scenarios, which makes the design attractive to various computing environments. Overall, CacheSwap provides a unique design that blends hardware and OS components to make stacked DRAM even more compelling than it already is.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Advances in die-stacking technologies have made it possible to integrate gigabytes of DRAM within processor packages. This in-package die-stacked DRAM has the potential to significantly improve memory systems and thus alleviate the processor-memory performance gap. This research starts from a key question: how can we effectively bring the benefit of the die-stacked memory to a variety of computing environments?

In this dissertation, we have shown that conventional designs of using on-chip memory as caches or part of memory are not practical nor efficient when integrating gigabytes of die-stacked DRAM into computing systems. Consequently, we have proposed novel designs that employ several architectural innovations to exploit the stacked memory and heterogeneous memory systems, which are summarized as follows.

- Chapter III proposed a streamlined DRAM cache architecture that employs three innovations. First, we introduced a light-weight, low-latency *Hit-Miss Predictor* (HMP) that provides 97% accuracy on average, with a hardware cost of less than 1KB. Second, we proposed a self-balancing dispatch (SBD) mechanism that dynamically steers memory requests to either the die-stacked DRAM cache or to the off-chip main memory depending on the instantaneous queuing delays at the two memories. Lastly, to ensure correctness in the presence of dirty data in the cache, we introduced a hybrid write policy that forces the majority of the DRAM cache to operate in a write-through mode, and only enables write-back for a limited set of pages that have high write traffic.

- Chapter IV presented a DRAM cache organization that uses error-correcting codes (ECCs), strong checksums (CRCs), and dirty data duplication to detect *and* correct a wide range of stacked DRAM failures, from traditional bit errors to large-scale row, column, bank, and channel failures. Furthermore, these RAS capabilities can be selectively enabled to tailor the level of reliability for different market needs. With only a modest performance degradation compared to a DRAM cache with no ECC support, the proposed organization can correct all single-bit failures, and 99.9993% of all row, column, and bank failures, providing more than a 54,000$\times$ improvement in the FIT rate of silent-data corruptions compared to basic SECDED ECC protection.

- Chapter V proposed a low-cost architectural solution to efficiently enable using large fast memory as Part-of-Memory (PoM) seamlessly, without the involvement of the OS. The proposed PoM architecture effectively manages two different types of memory (slow and fast) combined to create a single physical address space. To achieve this, PoM implements the ability to dynamically remap regions of memory based on their access patterns and expected performance benefits.

- Chapter VI proposed a hybrid scheme where the hardware and operating system cooperatively manage the stacked DRAM such that it normally operates as a cache (for performance benefit) but can also be dynamically converted to a fast swap space (to avoid costly paging to disk) depending on the run-time active memory load.

Recommended Use Cases for Die-Stacked Memory

As discussed in Chapter I, depending on its deployment scenarios, die-stacked memory can be best exploited in a different way. Among the cache, PoM, and hybrid options presented in this dissertation, the cache usage is recommended compared to others when the off-chip to stacked memory capacity ratio is high. In the other extreme where the ratio is low and on-chip memory frequently needs to be served as main memory, PoM is recommended to enjoy the full memory capacity available in the system. For the moderate case where system's memory demands only occasionally exceed the off-chip memory capacity, a hybrid scheme will likely be a good choice rather than PoM to avoid unnecessary memory traffic caused by swapping operations. Table 17 summarizes the recommended use cases for the die-stacked memory.

Table 17: Recommended use cases for die-stacked memory.

Ratio (off-chip to stacked memory)	**High**	**Moderate**	**Low**
Recommendation	Cache (Chapter III)	Hybrid (Chapter VI)	PoM (Chapter V)

Future Research Directions

In this dissertation, we showed that incorporating die-stacked memory into the computing system—even while maintaining *software transparency*—could greatly improve system performance. However, devising techniques across the layers of the system stack would unleash the full potential of the stacked memory and enable new levels of performance and energy efficiency. In this context, it is interesting to investigate how the software stack can efficiently support die-stacked DRAM and heterogeneous main memory. For example, the OS may need to determine whether memory objects should be allocated on fast stacked memory or slow off-chip memory to begin with. Compiler support for die-stacked memory would also be required to unlock its full potential, so studies on how to generate stacked memory-aware code and perform feedback-directed optimizations could be interesting research directions.

It is also interesting to study software-hardware cooperative mechanisms to improve the resiliency of the systems with die-stacked memory. One could dynamically configure protection levels in order to strike a balance between reliability and performance. For example, critical memory resources (e.g., operating system data structures) may receive a high level of protection, while other low-priority user applications may receive no or minimal protection.

Implementing the processing-in-memory (PIM) functionality in die-stacked memory is another interesting research direction. Die-stacking technology allows multiple dies with different process technologies to be stacked together (e.g., logic dies + DRAM dies), thus presenting the opportunities to implement PIM in the foreseeable future. A variety of operations can be implemented in the logic layer of the PIM stack, from fixed-function operations (e.g., reductions) to more complex, programmable operations (e.g., garbage collection). We first need to investigate the computations, operations, functions, and algorithms that would benefit from offloading to the PIM stack. We then need to integrate PIM into the layers of the system stack. At higher levels, programming models, libraries, compilers, and runtimes must be adapted or newly developed. At the hardware level, data synchronization and communication interfaces between the processor, caches, and memory must be effectively designed. We leave all these interesting topics for future work.

REFERENCES

[1] ABELLA, J., CHAPARRO, P., VERA, X., CARRETERO, J., and GONZÁLEZ, A., "On-Line Failure Detection and Confinement in Caches," in *14th IEEE International On-Line Testing Symposium (IOLTS)*, pp. 3–9, July 2008.

[2] AMD, "AMD I/O Virtualization Technology (IOMMU) Specification." http://support.amd.com/TechDocs/48882.pdf.

[3] BADAM, A. and PAI, V. S., "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, pp. 16–29, 2011.

[4] BELLARD, F., "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, pp. 41–46, 2005.

[5] BLACK, B., ANNAVARAM, M., BREKELBAUM, N., DEVALE, J., JIANG, L., LOH, G. H., MCCAULE, D., MORROW, P., NELSON, D. W., PANTUSO, D., REED, P., RUPLEY, J., SHANKAR, S., SHEN, J., and WEBB, C., "Die Stacking (3D) Microarchitecture," in *Proceedings of the 39th International Symposium on Microarchitecture (MICRO-39)*, pp. 469–479, 2006.

[6] BOUVIER, D., COHEN, B., FRY, W., GODEY, S., and MANTOR, M., "Kabini: An AMD Accelerated Processing Unit System on A Chip," *Micro, IEEE*, vol. 34, pp. 22–33, Mar 2014.

[7] BOVET, D. and CESATI, M., *Understanding the Linux Kernel*. O'Reilly Series, O'Reilly, 2002.

[8] BURGER, D., GOODMAN, J. R., and KÄGI, A., "Memory Bandwidth Limitations of Future Microprocessors," in *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA-23)*, pp. 78–89, 1996.

[9] CHOW, C. K., "On Optimization of Storage Hierarchies," *IBM Journal of Research and Development*, vol. 18, no. 3, pp. 194–203, 1974.

[10] CONWAY, P. and HUGHES, B., "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, pp. 10–21, Mar. 2007.

[11] CONWAY, P., KALYANASUNDHARAM, N., DONLEY, G., LEPAK, K., and HUGHES, B., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro Magazine*, pp. 16–29, March–April 2010.

[12] DAMARAJU, S., GEORGE, V., JAHAGIRDAR, S., KHONDKER, T., R., M., SARKAR, S., SCOTT, S., I., S., and SUBBIAH, A., "A 22nm IA Multi-CPU and GPU System-on-Chip," in *Proceedings of the 2012 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 56–57, 2012.

[13] DELL, T. J., "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," 1997.

[14] D'HEURLE, F. M., "Electromigration and Failure in Electronics: An Introduction," *Proceedings of the IEEE*, vol. 59, no. 10, 1971.

[15] DONG, X., XIE, Y., MURALIMANOHAR, N., and JOUPPI, N. P., "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pp. 1–11, 2010.

[16] EKMAN, M. and STENSTROM, P., "A Cost-Effective Main Memory Organization for Future Servers," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS-19)*, pp. 45.1–, 2005.

[17] ELPIDA CORPORATION, "Elpida Completes Development of TSV (Through Silicon Via) Multi-Layer 8-Gigabit DRAM." Press Release, August 27, 2009. http://http://www.elpida.com/en/news/2009/index.html.

[18] EYERMAN, S. and EECKHOUT, L., "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro Magazine*, vol. 28, May–June 2008.

[19] FALSAFI, B. and WOOD, D. A., "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pp. 229–240, 1997.

[20] FRANK, S., BURKHARDT, H., and ROTHNIE, J., "The KSR 1: Bridging the Gap between Shared Memory and MPPs," in *Compcon Spring '93, Digest of Papers.*, 1993.

[21] GINDELE, J. D., "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin*, vol. 20, pp. 696–697, July 1977.

[22] GIRIDHAR, B., CIESLAK, M., DUGGAL, D., DRESLINSKI, R., CHEN, H. M., PATTI, R., HOLD, B., CHAKRABARTI, C., MUDGE, T., and BLAAUW, D., "Exploring DRAM Organizations for Energy-Efficient and Resilient Exascale Memories," in *Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, pp. 23:1–23:12, 2013.

[23] HAGERSTEN, E. and KOSTER, M., "WildFire: A Scalable Path for SMPs," in *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 172–181, 1999.

[24] HAGERSTEN, E., LANDIN, A., and HARIDI, S., "DDM - A Cache-Only Memory Architecture," *IEEE Computer*, vol. 25, pp. 44–54, Sep 1992.

[25] HAMMARLUND, P., MARTINEZ, A., BAJWA, A., HILL, D., HALLNOR, E., JIANG, H., DIXON, M., DERR, M., HUNSAKER, M., KUMAR, R., OSBORNE, R., RAJWAR, R., SINGHAL, R., D'SA, R., CHAPPELL, R., KAUSHIK, S., CHENNUPATY, S., JOURDAN, S., GUNTHER, S., PIAZZA, T., and BURTON, T., "Haswell: The Fourth-Generation Intel Core Processor," *Micro, IEEE*, vol. 34, pp. 6–20, Mar 2014.

[26] HAMMING, R. W., "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, 1950.

[27] HENNING, J. L., "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[28] HERTZ, M., FENG, Y., and BERGER, E. D., "Garbage Collection Without Paging," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 143–153, 2005.

[29] HONG, S. and KIM, H., "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, pp. 152–163, 2009.

[30] HWANG, A. A., STEFANOVICI, I. A., and SCHROEDER, B., "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-17)*, pp. 111–122, 2012.

[31] IBM, "IBM Power8." http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.210-POWER-Studecheli-IBM.pdf.

[32] INTEL, "KnightsLanding." http://www.realworldtech.com/knights-landing-details/.

[33] JALEEL, A., THEOBALD, K., STEELY, S. C., and EMER, J., "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA-37)*, pp. 60–71, 2010.

[34] JEDEC, "Wide I/O Single Data Rate (Wide I/O SDR)." http://www.jedec.org/standards-documents/docs/jesd229.

[35] JEVDJIC, D., VOLOS, S., and FALSAFI, B., "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA-40)*, pp. 404–415, 2013.

[36] JIANG, X., MADAN, N., ZHAO, L., UPTON, M., IYER, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., and BALASUBRAMONIAN, R., "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA-16)*, pp. 1–12, 2010.

[37] JOINT ELECTRON DEVICES ENGINEERING COUNCIL, "JEDEC: 3D-ICs." http://www.jedec.org/category/technology-focus-area/3d-ics-0.

[38] JUNG, J.-Y. and CHO, S., "Memorage: Emerging Persistent RAM Based Malleable Main Memory and Storage Architecture," in *Proceedings of the 27th International Conference on Supercomputing (ICS'13)*, pp. 115–126, 2013.

[39] KALLA, R., SINHAROY, B., STARKE, W. J., and FLOYD, M., "Power7: IBM's Next-Generation Server Processor," *IEEE Micro Magazine*, vol. 30, March–April 2010.

[40] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., and LEWIN, D., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the 29th Symposium on Theory of Computing (STOC'97)*, pp. 654–663, 1997.

[41] KIKUDA, S., MIYAMOTO, H., MORI, S., NIIRO, M., and YAMADA, M., "Optimized Redundancy Selection Based on Failure-Related Yield Model for 64-Mb DRAM and Beyond," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, 1991.

[42] KIM, H., LEE, J., LAKSHMINARAYANA, N. B., SIM, J., LIM, J., and PHO, T., *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*. Georgia Institute of Technology, 2012.

[43] KIM, J.-S., OH, C., LEE, H., LEE, D., HWANG, H.-R., HWANG, S., NA, B., MOON, J., KIM, J.-G., PARK, H., RYU, J.-W., PARK, K., KANG, S.-K., KIM, S.-Y., KIM, H., BANG, J.-M., CHO, H., JANG, M., HAN, C., LEE, J.-B., KYUNG, K., CHOI, J.-S., and JUN, Y.-H., "A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking," in *Proceedings of the 2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 496–498, 2011.

[44] KOOPMAN, P. and CHAKRAVARTY, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN-2004)*, pp. 145–154, 2004.

[45] LAUDON, J. and LENOSKI, D., "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pp. 241–251, 1997.

[46] LEE, C. J., NARASIMAN, V., MUTLU, O., and PATT, Y. N., "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, pp. 327–336, 2009.

[47] Lee, D. and others, "LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, pp. 1352–1361, Dec. 2001.

[48] Lee, H.-H. S., Tyson, G. S., and Farrens, M. K., "Eager Writeback - a Technique for Improving Bandwidth Utilization," in *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, pp. 11–21, 2000.

[49] Liu, H., Ferdman, M., Huh, J., and Burger, D., "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," in *Proceedings of the 41st International Symposium on Microarchitecture (MICRO-41)*, pp. 222–233, 2008.

[50] Loh, G. H., "3D-Stacked Memory Architectures for Multi-Core Processors," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA-35)*, pp. 453–464, 2008.

[51] Loh, G. H. and Hill, M. D., "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *Proceedings of the 44th International Symposium on Microarchitecture (MICRO-44)*, pp. 454–464, 2011.

[52] Loh, G. H., Jayasena, N., McGrath, K., O'Connor, M., Reinhardt, S., and Chung, J., "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, 2012.

[53] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M., "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, pp. 716–727, Apr. 2012.

[54] Meixner, A., Bauer, M. E., and Sorin, D., "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-40)*, pp. 210–222, 2007.

[55] Micron Technology.

[56] Micron Technology, "1Gb: x4, x8, x16 DDR3 SDRAM," 2006.

[57] Micron Technology, "Calculating Memory System Power for DDR3," 2007.

[58] Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, pp. 29–41, 2003.

[59] NVidia Corp., "Tuning CUDA Applications for Fermi," DA 05612-001_v1.5, 2011.

[60] PAWLOWSKI, J. T., "Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem," in *Hot Chips 23*, 2011.

[61] QUINN, H., GRAHAM, P., and FAIRBANKS, T., "SEEs Induced by High-Energy Protons and Neutrons in SDRAM," in *Proceedings of the 2011 Radiation Effects Data Workshop (REDW)*, pp. 1–5, 2011.

[62] QURESHI, M. K. and LOH, G. H., "Fundamental Latency Trade-Offs in Architecting DRAM Caches," in *Proceedings of the 45th International Symposium on Microarchitecture (MICRO-45)*, pp. 235–246, 2012.

[63] RAMOS, L. E., GORBATOV, E., and BIANCHINI, R., "Page Placement in Hybrid Memory Systems," in *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*, pp. 85–95, 2011.

[64] REDIS, "http://redis.io."

[65] REED, I. S. and SOLOMON, G., "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, 1960.

[66] RIXNER, S., DALLY, W., KAPASI, U., MATTSON, P., and OWENS, J., "Memory Access Scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, pp. 128–138, 2000.

[67] SAULSBURY, A., WILKINSON, T., CARTER, J., and LANDIN, A., "An Argument for Simple COMA," in *Proceedings of the 1st Symposium on High Performance Computer Architecture (HPCA-1)*, pp. 276–285, 1995.

[68] SAXENA, M. and SWIFT, M. M., "FlashVM: Virtual Memory Management on Flash," in *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, pp. 187–200, 2010.

[69] SCHRODER, D. K. and BABCOCK, J. A., "Negative Bias Temperature Instability: Road to Cross in Deep Submicron Silicon Semiconductor Manufacturing," *Journal of Applied Physics*, vol. 94, no. 1, pp. 1–18, 2003.

[70] SEZNEC, A. and MICHAUD, P., "A Case for (Partially) TAgged GEometric History Length Branch Prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

[71] SHERWOOD, T., PERELMAN, E., HAMERLY, G., and CALDER, B., "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-10)*, pp. 45–57, 2002.

[72] SIM, J., ALAMELDEEN, A. R., CHISHTI, Z., WILKERSON, C., and KIM, H., "Transparent hardware management of stacked dram as part of memory," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, pp. 13–24, 2014.

[73] SIM, J., DASGUPTA, A., KIM, H., and VUDUC, R., "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pp. 11–22, 2012.

[74] SIM, J., LEE, J., QURESHI, M. K., and KIM, H., "FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA-39)*, pp. 321–332, 2012.

[75] SIM, J., LOH, G. H., KIM, H., O'CONNOR, M., and THOTTETHODI, M., "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *Proceedings of the 45th International Symposium on Microarchitecture (MICRO-45)*, pp. 247–257, 2012.

[76] SIM, J., LOH, G. H., SRIDHARAN, V., and O'CONNOR, M., "Resilient Die-stacked DRAM Caches," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA-40)*, pp. 416–427, 2013.

[77] SIM, J., LOH, G. H., SRIDHARAN, V., and O'CONNOR, M., "A Configurable and Strong RAS Solution for Die-Stacked DRAM Caches," *Micro, IEEE*, vol. 34, pp. 80–90, May 2014.

[78] SMITH, J. E., "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture (ISCA-8)*, pp. 135–148, 1981.

[79] SNAVELY, A. and TULLSEN, D. M., "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, pp. 234–244, 2000.

[80] SRIDHARAN, V. and LIBERTY, D., "A Study of DRAM Failures in the Field," in *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pp. 76:1–76:11, 2012.

[81] TAYLOR, G., DAVIES, P., and FARMWALD, M., "The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism," in *Proceedings of the 17th International Symposium on Computer Architecture (ISCA-17)*, pp. 355–363, 1990.

[82] UDIPI, A., MURALIMANOHAR, N., BALASUBRAMONIAN, R., DAVIS, A., and JOUPPI, N. P., "LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA-39)*, pp. 285–296, 2012.

[83] UDIPI, A., MURALIMANOHAR, N., CHATTERJEE, N., BALASUBRAMONIAN, R., DAVIS, A., and JOUPPI, N. P., "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA-37)*, pp. 175–186, 2010.

[84] WILKES, M., "Slave Memories and Dynamic Storage Allocation," *IEEE Transactions on Electronic Computers*, vol. EC-14, pp. 270–271, April 1965.

[85] WULF, W. A. and MCKEE, S. A., "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, pp. 20–24, March 1995.

[86] YANG, T., BERGER, E. D., KAPLAN, S. F., and MOSS, J. E. B., "CRAMM: Virtual Memory Support for Garbage-Collected Applications," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, pp. 103–116, 2006.

[87] YOAZ, A., EREZ, M., RONEN, R., and JOURDAN, S., "Speculation Techniques for Improving Load Related Instruction Scheduling," in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26)*, pp. 42–53, 1999.

[88] YOON, D. H. and EREZ, M., "Memory Mapped ECC: Low-cost Error Protection for Last Level Caches," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, pp. 116–127, 2009.

[89] YOON, D. H. and EREZ, M., "Virtualized and Flexible ECC for Main Memory," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-15)*, pp. 397–408, 2010.

[90] ZHANG, W., GURUMURTHI, S., KANDEMIR, M., and SIVASUBRAMANIAM, A., "ICR: In-Cache Replication for Enhancing Data Cache Reliability," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN-2003)*, pp. 291–300, 2003.

[91] ZHAO, L., IYER, R., ILLIKKAL, R., and NEWELL, D., "Exploring DRAM Cache Architectures for CMP Server Platforms," in *Proceedings of the 25th International Conference on Computer Design (ICCD-25)*, pp. 55–62, 2007.

[92] ZHENG, H., LIN, J., ZHANG, Z., GORBATOV, E., DAVID, H., and ZHU, Z., "Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *Proceedings of the 41st International Symposium on Microarchitecture (MICRO-41)*, pp. 210–221, 2008.