2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071016

NOMAD: Enabling Non-blocking OS-managed DRAM Cache via Tag-Data Decoupling

Youngin Kim, Hyeonjin Kim, and William J. Song
School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea
yiwkd2@yonsei.ac.kr, hyeonjin_kim@yonsei.ac.kr, and wjhsong@yonsei.ac.kr

***Abstract*—This paper introduces a DRAM cache architecture that provides near-ideal access time and non-blocking miss handling. Previous DRAM cache (DC) designs are classified into two categories, *HW-based* and *OS-managed* schemes. Hardware-based designs implement non-blocking caches that can handle multiple DC misses using MSHRs, but they have drawbacks in metadata management since storing tags in on-package DRAM significantly increases the effective cycle time of DC accesses. In contrast, OS-managed schemes utilize PTEs for storing tags and caching them in TLBs, which can achieve ideal DC access time. However, they implement blocking caches that stall application threads on misses until cache fills are completed. To overcome the limitations of both HW-based and OS-managed schemes, this paper introduces a DRAM cache architecture named *Non-blocking OS-managed DRAM cache (NOMAD)*. Unlike conventional caches that guarantee the presence of data on tag hits, NOMAD decouples tag and data management to enable non-blocking miss handling in an OS-managed DRAM cache. The *front-end* OS routines of NOMAD manage DC tags using PTEs and TLBs, and its *back-end* hardware handles data management in the DRAM cache. On a DC miss, the OS updates a tag, offloads a cache-fill command to the back-end, and immediately resumes an application thread without waiting for the cache fill to complete. Instead, the back-end hardware handles the cache fill without blocking the application thread. By decoupling tag and data management in NOMAD, a tag hit does not necessarily guarantee the presence of data in the DRAM cache. The back-end traces which DC lines are still in transfers and checks if the demanded part of a cache line has been transferred yet for every DC access. Notably, this back-end procedure does not require an OS intervention, thereby implementing a non-blocking DRAM cache. Experiment results show that NOMAD reduces application stall cycles by 76.1% and improves IPC by 16.7% over a state-of-the-art OS-managed scheme.**

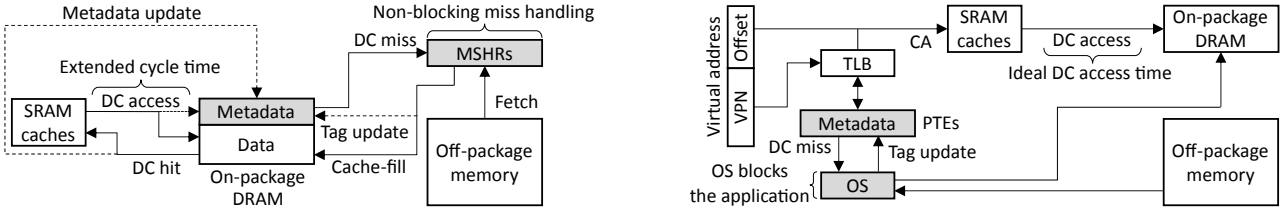
I. Introduction

As emerging applications demand increasingly larger memory bandwidth and capacity, heterogeneous memory systems pairing up high-bandwidth on-package DRAM with large-capacity off-package memory [5], [8], [9], [14], [42], [46] have been developed as promising solutions to overcome memory wall problems. To effectively utilize the heterogeneous memory systems, previous studies proposed to architect on-package DRAM as a cache [18], [24]–[26], [31], [39], [49], [50]. The prior DRAM cache (DC) designs are classified into two categories, *HW-based* and *OS-managed* schemes.

Hardware-based DRAM caches are implemented based on the design principle of traditional data caches [13], [28]. They operate as non-blocking caches that can handle multiple outstanding misses using miss status/information holding registers (MSHRs), which enable the caches to service subsequent memory requests while the miss handling operations are in progress. However, HW-based designs have fundamental drawbacks in managing metadata (e.g., tags). Since the gross size of DC metadata is too large to be stored in an SRAM array, tags are stored in on-package DRAM for a scalable implementation [24], [31], [39]. Such a design causes a DC controller to spend a significant portion of memory bandwidth on transferring the metadata to read tags, update control bits, etc. The extra bandwidth consumption of on-package DRAM increases the effective cycle time of DC accesses and thus adversely affects the performance [10], [18], [19].

OS-managed DRAM caches [22], [29], [37] were proposed to overcome the limitations of HW-based designs in the metadata management. OS-managed schemes expose on-package DRAM to operating systems and store data at the page granularity via changes to address translation mechanisms. They utilize a page table entry (PTE) to store a DC tag and retrieve the information from a translation lookaside buffer (TLB) to eliminate the overhead of metadata transfers from/to the on-package DRAM, thereby providing ideal DC access time. However, prior OS-managed designs have a critical limitation in that they implement blocking caches, where application threads are stalled while OS routines are handling DC misses. On a DC miss, a miss handler allocates a cache frame and also controls loading page data to the on-package DRAM. During the miss handling procedure, an application thread is suspended for thousands of cycles. As a result, OS-managed DRAM caches suffer from severe miss penalties.

To resolve the problems of previous DRAM cache designs, this paper proposes *Non-blocking, OS-managed DRAM cache (NOMAD)*. NOMAD as an OS-managed DRAM cache provides near-ideal DC access time by storing DC tags in PTEs in a similar way as prior OS-managed schemes [22], [29]. But, it allows non-blocking miss handling via a hardware support, which can handle cache-fill operations without suspending application threads. The primary limitation of prior DC designs lies in that they have coupled tag and data management such that a tag hit must guarantee the presence of data in the cache, referred to as a data hit. In such a tag/data-coupled environment, an OS-managed scheme is bound to be a blocking cache because a DC handler cannot resume an application thread until a cache fill is completed to ensure a data hit on a tag hit. In contrast, NOMAD effectively *decouples* tag and data management by leveraging both OS and hardware features.



(a) HW-based DRAM cache

(b) OS-managed DRAM cache

Fig. 1: (a) A HW-based scheme supports non-blocking miss handling using MSHRs. However, it stores DC tags in on-package DRAM, which consumes memory bandwidth for reading and updating metadata. (b) An OS-managed scheme utilizes PTEs to store DC tags, which can provide ideal DC access time. However, it suffers from a significant miss penalty since an application thread is stalled while the OS handles the DC miss.

The proposed NOMAD scheme is comprised of two parts, *front-end* OS routines and *back-end* hardware. On a DC miss, the front-end only updates a PTE and TLB without waiting for the cache fill to complete, and it resumes an application thread immediately. Instead, a cache-fill command is offloaded to the back-end hardware, where the outstanding miss is traced by a page copy status/information holding register (PCSHR). By allowing a swift restart on a DC miss and tracing multiple misses using PCSHRs, the DRAM cache can service subsequent memory requests in a non-blocking manner. The back-end concurrently executes cache-fill commands in PCSHRs similar to MSHRs in traditional hardware caches. Since the NOMAD front-end updates only a PTE and TLB, a TLB hit does not guarantee the presence of page data in the DRAM cache. To confirm the validity of page data, it requires looking up PCSHRs on a DC access to check if the demanded page is still in a transfer. No matched tags in the PCSHRs indicate that the whole page data have been already fetched, or otherwise it waits for the data block to arrive. Importantly, this procedure does not involve an OS intervention. Referencing PCSHRs adds a delay to the critical path of DC access time, but the overhead is substantially smaller than consuming on-package DRAM bandwidth to read and update the metadata of cached pages in HW-based schemes. Also, NOMAD has an advantage of minimizing miss handling latency via the back-end hardware support by rapidly restarting thread executions and tracing multiple outstanding misses in a non-blocking fashion. Experiment results show that NOMAD reduces application stall cycles by 76.1% and improves IPC by 16.7% on average over a state-of-the-art OS-managed DRAM cache.

II. Background and Motivation

To overcome the bandwidth and capacity challenges of memory systems, heterogeneous memory systems utilizing high-bandwidth on-package DRAM and large-capacity off-package memory have been widely explored. To effectively utilize the heterogeneous memory systems, prior studies proposed to architect on-package DRAM as a cache [18], [22], [24]–[26], [29], [31], [37], [39], [49], [50]. The previous DRAM cache designs are classified into two categories, HW-based and OS-managed schemes.

A. Hardware-based DRAM Caches

The key advantage of HW-based schemes is that they implement non-blocking DRAM caches. As depicted in Fig. 1a, a HW-based scheme employs several MSHRs to handle multiple misses at a time without impeding subsequent memory requests. The non-blocking miss handling capability affects several aspects of the DRAM cache performance. In particular, it enables the DRAM cache to exploit memory-level parallelism by concurrently tracing multiple outstanding misses. By offloading miss handling to MSHRs, the DRAM cache can proceed to service subsequent memory requests without needing to wait for the preceding misses to complete beforehand. Also, the HW-based scheme minimizes the effective miss latency since a DC controller can send a response to a last-level cache (LLC) as soon as a demanded data block arrives from the off-package memory instead of waiting for the entire cache line.

However, HW-based schemes have fundamental drawbacks in managing metadata (e.g., tags, valid and dirty bits). Since these bits are stored in on-package DRAM for scalable implementations [24], [31], [39], accessing the metadata incurs extra bandwidth consumption of the on-package DRAM as illustrated in Fig. 1a with dashed lines. For every DC access, a DC controller should retrieve a tag from the on-package DRAM to determine whether the access is hit or miss. In addition, all metadata updates (e.g., dirty, LRU bits) also require the use of memory bandwidth [10]. Although the tag serialization latency can be hidden by transferring a pair of tag and data in a single burst [24], [39], the extra bandwidth expenses increase the effective cycle time of DC accesses and thus result in performance degradation.

B. OS-managed DRAM Caches

OS-managed DRAM caches [22], [29], [37] were proposed to overcome the limitations of HW-based designs. An OS-managed scheme exposes on-package DRAM to an operating system and stores data at the page granularity as shown in Fig. 1b. The caching mechanism of the OS-managed scheme is enabled without modifications to a memory controller by storing a DC tag (i.e., cache frame number) in a PTE and retrieving it from TLBs. Since the OS-managed scheme can directly obtain a cache address (CA) on a TLB hit, the memory

TABLE I: Workload Characteristics

Class	Abbr.	Workloads	RMHB (GB/s)	LLC MPMS	Memory footprint (GB)
Excess	cact	cactusADM [3]	43.8	486.6	11.9
	sssp	sssp [3]	38.8	511.1	2.3
	bwav	bwaves [16]	31.7	588.1	4.5
Tight	les	leslie3d [16]	26.5	532.8	7.5
	libq	libquantum [16]	25.1	210.6	4.0
	gems	gemsFDTD [16]	24.8	269.2	6.3
	bfs	bfs [3]	23.1	298.5	2.4
Loose	cc	cc [3]	13.5	183.1	2.3
	lbm	lbm [16]	12.4	270.5	3.2
	mcf	mcf [16]	12.2	472.0	2.8
	bc	bc [3]	10.8	533.7	1.3
Few	ast	astar [16]	6.9	72.1	1.0
	pr	pr [3]	3.4	691.9	4.8
	sop	soplex [16]	1.7	310.2	1.2
	tc	tc [3]	1.66	226.3	2.3

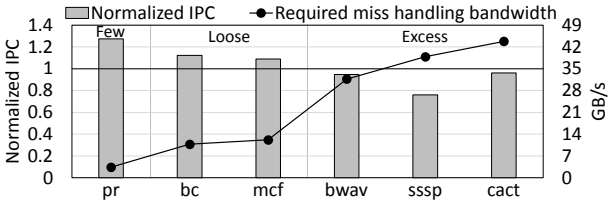


Fig. 2: IPC of OS-managed DRAM cache [29] relative to that of the HW-based design [24] for benchmarks exhibiting high LLC MPMS in different workload classes.

bandwidth of on-package DRAM is not wasted on transferring metadata, thereby providing ideal DC access time.

However, the OS-managed scheme implements a blocking cache in that an application thread is stalled while the OS handles a DC miss. On a DC miss, a miss handler allocates a new cache frame and performs two tasks, tag and data management. For the tag management, it records the mapping information between a pair of physical and cache frames in a kernel data structure, and a page frame number in the PTE that was previously a physical frame number (PFN) is replaced with a cache frame number (CFN), which serves as the tag of the new cache frame. For the data management, the miss handler executes a cache fill by copying page data from the off-package memory to the DRAM cache. Since these two tasks are processed while the application thread is suspended, the DC miss is penalized by thousands of cycles mainly due to the cache-fill execution.

C. Motivation

To demonstrate the effect of miss handling mechanisms in different DRAM cache schemes, we chose tagless DRAM cache (TDC) and the tag management mechanism of Unison cache [24], which we call tags-in-DRAM (TiD), as representative OS-managed and HW-based designs, respectively. The miss handling mechanisms of these schemes were modeled using a cycle-level simulator [4], [30], [32] and tested with various workloads from SPEC2006 [16] and GAP Benchmark Suite (GAPBS) [3]. To evaluate inherent workload characteristics, we measured the required miss handling bandwidth (RMHB) of the off-package memory and last-level cache misses per microsecond (LLC MPMS) for each workload under an ideal OS-managed configuration. The stall time of the blocking OS-managed scheme (i.e., TDC) should be proportional to the RMHB metric, and the LLC MPMS manifests the bandwidth consumption of on-package DRAM for metadata accesses in the HW-based scheme (i.e., TiD). The benchmarks are categorized based on their RMHB results into four classes as shown in Table I. The Excess-class workloads have greater RMHB than the available bandwidth, and thus they put strong pressure on the off-package memory. Benchmarks of the Tight class consume nearly all of the off-package memory bandwidth for miss handling, and those of the Loose class need about a half of the memory bandwidth. Lastly, Few-class benchmarks use a negligible portion of the bandwidth for miss handling.

Fig. 2 plots the instructions per cycle (IPC) of TDC relative to that of TiD for six benchmarks exhibiting high LLC MPMS in Table I except for the les benchmark whose anomaly behavior is described in Section IV-B. The graph shows that the OS-managed scheme (i.e., TDC) outperforms the HW-based scheme (i.e., TiD) for workloads with low RMHB (i.e., pr, bc, mcf). The HW-based design suffers from extended DC access time due to substantial on-package DRAM bandwidth usage, whereas the OS-managed scheme provides the benchmarks in Few and Loose classes with the ideal DC access time. On the contrary, the OS-managed scheme becomes inferior to the HW-based design when it comes to high-RMHB benchmarks in the Excess class since the severe miss handling latency of the OS-managed scheme leads to performance degradation, while the HW-based scheme with non-blocking miss handling effectively tolerates a large number of DC misses.

III. NOMAD Cache Design

A. Decoupled Tag-Data Management Scheme

The primary limitation of prior OS-managed DRAM caches is that application stalls are inevitable since the OS performs cache fills on DC misses to ensure data hits on tag hits. We refer to such a method as *coupled* tag-data management. To enable a non-blocking OS-managed DRAM cache, this paper proposes a *decoupled* tag-data management scheme. In the decoupled scheme, tags and data are controlled in different procedures, which implies that a tag hit does not necessarily guarantee a data hit in the DRAM cache. On a DC miss, the front-end OS routines of NOMAD update only a tag and immediately resume a stalled application thread. Instead, a cache-fill command is offloaded to the back-end hardware. Although this requires verifying the presence of data on all DC accesses through the back-end hardware, the decoupled tag-data management allows non-blocking miss handling and substantially reduces the miss handling latency.

B. Overall Structure

The proposed NOMAD scheme consists of two parts, *front-end* OS routines and *back-end* hardware. The front-end manages DC tags using PTEs and TLBs, which is implemented in a similar way as previous OS-manged schemes [22], [29],

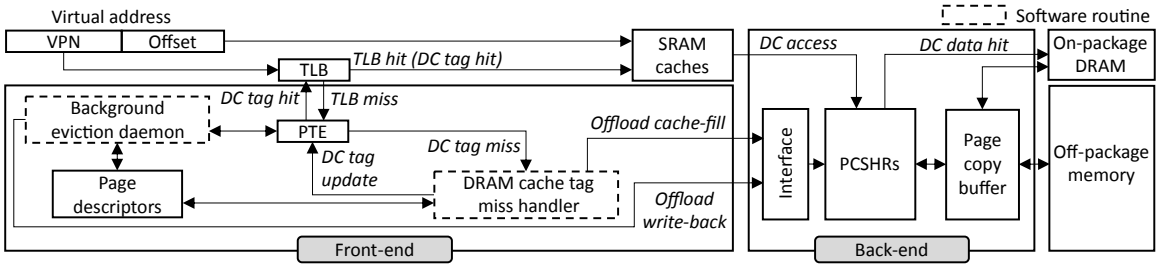


Fig. 3: The overall structure and memory access path of NOMAD, which consists of *front-end* OS routines and *back-end* hardware. The front-end manages DC tags using PTEs and TLBs, and the back-end controls data management in the DRAM cache (i.e., cache fill, writebacks, verifying data hits).

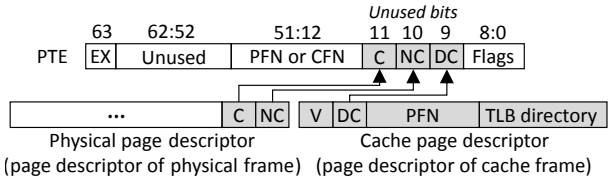


Fig. 4: Page descriptor and PTE extension [1] in NOMAD.

[37]. The back-end governs data management (i.e., cache fills, writebacks, verifying data hits), and its miss handling architecture is implemented based on the design principle of traditional non-blocking hardware caches [13], [28].

Fig. 3 illustrates the overall structure and memory access path of NOMAD. As an OS-managed DC scheme, NOMAD exposes the address space of on-package DRAM to the OS and stores data at the page granularity. On a TLB miss, the front-end examines a PTE to find if the page is cached. If not, a new cache frame is allocated, and its cache-fill task is offloaded to the back-end hardware. A DC tag miss handler replaces a PFN in the PTE with a CFN, which serves as the tag of the new cache frame. The mapping information between the PFN and CFN is stored in a kernel data structure called page descriptor. Then, the execution of an application thread is immediately resumed. The front-end also includes a background eviction daemon, which proactively evicts cache frames from the DRAM cache. When a cache frame is evicted, its PTE is restored with the original PFN from the page descriptor. If the evicted page is dirty, the eviction daemon offloads a writeback command to the back-end hardware. More details of the cache frame management are presented in Section III-C.

The NOMAD back-end allocates a PCSHR on receiving a command from the front-end. The back-end concurrently executes the cache-fill and writeback commands of PCSHRs, and their page copy states are progressively updated. A DC controller does not need to perform extra tasks to obtain a cache address (CA) because a memory request is initiated with a valid CA by referencing a TLB. However, a TLB hit does not guarantee the presence of page data on a DC access because of the decoupled tag-data management. Therefore, the back-end looks up PCSHRs to check if a target data block is available. If the DC access has no matched tags in any PCSHRs, it implies that the whole page data have been already fetched to the on-package DRAM, which is referred to as a *data hit*. Then, the access can safely proceed to retrieve the data block from the DRAM cache. If a PCSHR has a matched tag, the access is regarded as a *data miss*, and handling this access depends on the page copy status recorded in the PCSHR. If the demanded part of the page has been fetched at least, the PCSHR can service the request. Otherwise, it has to wait for the data to arrive from the off-package memory.

C. Front-end OS Routines

Conventional operating systems manage page frames mainly using two data structures, page tables and page descriptors (a.k.a. PFN database in Windows [47]). Page tables are used for virtual-to-physical address translations, and page descriptors contain page frame information such as recently accessed, shared states, etc. A part of the page frame information (e.g., present, permission) is also needed for address translations, so a few of those bits are included in PTEs. Page descriptors are used for memory management tasks in the OS (e.g., allocation, reclamation). The descriptors are organized as an array and indexed by page frame numbers in that the array is virtually and physically contiguous [7]. A heterogeneous memory system such as NOMAD has to maintain two kinds of page frames, physical and cache frames [29]. Similar to traditional page frame management methods, the NOMAD front-end manages page frames using page tables, physical page descriptors (PPDs), and cache page descriptors (CPDs).

1) Page Descriptor and PTE Extension: Fig. 4 shows kernel data structures for managing page frames. A PPD is extended to append two additional bits, cached (C) and non-cacheable (NC). The C bit indicates whether the page is cached in the DRAM cache, and NC tells if the page is cacheable or not. These two bits are also added to the unused field of a PTE [1] since they are used for determining a DC tag miss (i.e., cacheable but not cached) during an address translation.

A CPD contains the information for handling a cache frame as shown in Fig. 4. A valid bit (V) indicates the validity of a cache frame mapping, and a dirty-in-cache (DC) bit tells if a writeback to the off-package memory is required for the cache frame on eviction. Similar to dirty bits in conventional PPDs

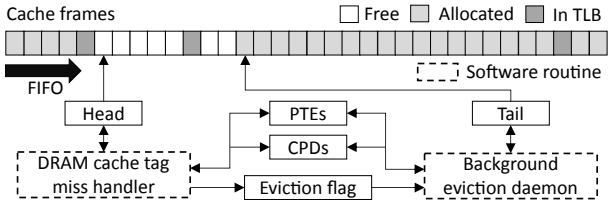


Fig. 5: The front-end manages cache frames in a FIFO manner using a circular free queue. A DC tag miss handler allocates cache frames from the head on demand, a background eviction daemon proactively evicts victim frames from the tail.

Algorithm 1. DRAM Cache Tag Miss Handling Routine

```
Input: pte (page table entry), va (virtual address)
 1: mutex_lock(&cache_frame_management_mutex)
    /* Find free cache frame */
 2: cpd = cache_page_descriptors[head]
 3: while unlikely cpd.valid do
 4:    cpd = cache_page_descriptors[++head]
 5: end while
    /* Offload data management task to back-end */
 6: send_cachefill(head, pte.page_frame_num, offset(va))
    /* Tag management */
 7: cpd.valid = true
 8: cpd.pfn = pte.page_frame_num
 9: pte.cached = true
10: pte.page_frame_num = head
    /* Set eviction flag */
11: num_free_cache_frames--
12: if num_free_cache_frame < eviction_threshold then
13:    eviction_flag = true
14: end if
15: mutex_unlock(&cache_frame_management_mutex)
```

and PTEs, the DC bit is also stored in a CPD and PTE. The DC bit can be set on write accesses without extra overhead, as conventional systems already do so for the dirty bits of PPDs and PTEs [7]. The CPD also includes the PFN of a physical frame mapped to the cache frame for reclamation. Lastly, a TLB directory is used for TLB shootdown avoidance [29], which traces whether the cache frame information resides in TLBs. When the PTE of a cache frame is allocated to or evicted from TLBs, corresponding bits in the TLB directory are set or cleared, respectively. The OS utilizes the TLB directory of CPDs when it searches for cache frames to evict such that page frames whose address translation information remains in TLBs are skipped to avoid the invocation of a TLB shootdown protocol [6], [37].

2) Cache Frame Management: NOMAD employs a simple first-in, first-out (FIFO) policy for replacing cache frames in the OS-managed DRAM cache similar to TDC [29] because other replacement policies such as least recently used (LRU) require frequent access profiling, which incurs performance degradation [7]. Despite the simplicity, the fully-associative nature of the OS-managed design combined with the FIFO replacement policy exhibits about 23% less DC misses on average than a 16-way set-associative HW-based DRAM cache using an LRU policy. Since HW-based designs in the prior work [24], [49] could have only 4-way set-associative structures because of scalability issues, engaging a FIFO replacement policy in a fully-associated OS-managed DRAM

Algorithm 2. Background Eviction Routine

```
 1: mutex_lock(&cache_frame_management_mutex)
    /* Reset flag */
 2: eviction_flag = false
    /* Flush cache for data consistency */
 3: flush_cache_range(tail,n) /* (start,victims) */
    /* Evict pages */
 4: for i = 0 to n − 1 do
 5:    cpd = cache_page_descriptors[tail]
       /* Skip this page if it is in any of TLB. */
 6:    if unlikely cpd.tlb_directory != 0 then
 7:       tail++; continue
 8:    end if
       /* Perform writeback for dirty victims */
 9:    if cpd.dirty-in-cache then
10:       send_writeback(tail, cpd.pfn)
11:    end if
       /* Recover PTEs */
12:    rmap = get_reverse_mapping(cpd.pfn)
13:    for all pte in rmap do
14:       pte.page_frame_num = cpd.pfn
15:    end for
16:    cpd.valid = false
17:    num_free_cache_frames++
18: end for
19: mutex_unlock(&cache_frame_management_mutex)
```

cache is not necessarily a compromise. Thus, NOMAD utilizes a FIFO policy and manages cache frames using a circular free queue, which can be implemented as Fig. 5 with head and tail pointers. In particular, cache frames are allocated from the head on demand by a *DRAM cache tag miss handler* and evicted from the tail proactively by a *background eviction daemon*, which takes the eviction overhead out of the critical path of the miss handler in a similar way as what conventional operating systems do for managing physical frames [7].

3) DRAM Cache Tag Miss Handler: A DC tag miss is detected during a page table walk if a page is cacheable but not present in the DRAM cache. It invokes a DC tag miss handler whose procedure is outlined in Algorithm 1. The miss handler first examines the CPD of a cache frame pointed by the head of the free queue to check its validity (line 2). The cache frame can possibly be unfree because of past TLB shootdown avoidance [29] as shown in Fig. 5, although it is very unlikely to happen since the TLB coverage is much smaller than the DRAM cache capacity. The head advances until a free cache frame is found (lines 3-5). Once a free frame is found, a cache-fill command is sent to the back-end hardware (line 6). The original PFN is saved in the CPD, and the PTE is updated with the new CFN and cached (C) bit set (lines 7-10). The miss handler also tracks the number of free cache frames to set an eviction flag, which will be handled by the background eviction daemon (lines 11-14). Lastly, the NOMAD front-end resolves the tag miss and resumes an application, even though the data of the new cache frame may still be unavailable.

4) Background Eviction Daemon: The background eviction daemon of NOMAD performs cache frame reclamation in a FIFO manner as shown in Algorithm 2. The eviction procedure is invoked when the DC tag miss handler sets the eviction flag. To avoid frequent invocation of the background daemon, it proactively evicts a series of cache frames at once; `n` in line 3 is a power of two for alignment. To evict cache frames, SRAM cache lines containing the data of victim frames are invalidated

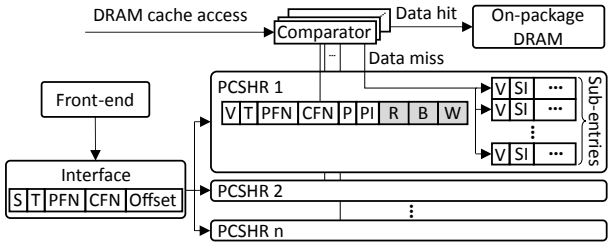


Fig. 6: The back-end hardware is comprised of an interface and PCSHRs. The interface is a memory-mapped device register than can be controlled by the OS. Each PCSHR contains the page copy status/information of a DC-missed page, and sub-entries are used for tracing the status of sub-blocks in a page.

for data consistency, and this can be done in one shot because aligned cache frames share the same CFN prefix (line 3). Then, the frames are evicted in a loop (lines 4-18). The eviction daemon skips cache frames whose address translation information remains in TLBs by referencing the TLB directory field of CPDs to avoid a TLB shootdown (lines 6-8). It also checks the dirty-in-cache (DC) bit of CPDs and sends write-back commands to the back-end hardware if necessary (lines 9-11). To replace CFNs stored in PTEs with original PFNs, the daemon accesses PTEs via existing reverse mappings that are indexed by PFNs [7], [33], [47] (lines 12-15). Lastly, the victim cache frames are invalidated (line 16).

D. Back-end Hardware

The back-end hardware of NOMAD is devised based on the design principle of traditional non-blocking hardware caches [13], [28]. However, it differs in that the back-end is controlled by software via an *interface* and performs cache fills and writebacks using *PCSHRs* and *page copy buffers*.

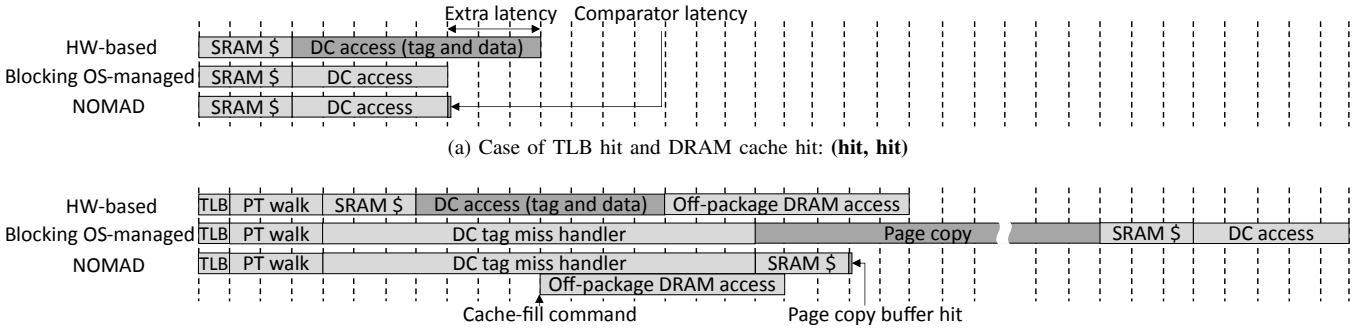
1) Interface: The back-end interface is a memory-mapped device register, which can be accessed by the OS and is comprised of the following fields as shown in Fig. 6. A state (S) bit indicates whether the interface is busy or idle, and the front-end can send a command only if the interface is in an idle state. A type (T) bit specifies a command type, which is either a cache-fill or writeback execution. The register also contains address information including a PFN, CFN, and offset (i.e., 76 bits total). On receiving a command from the front-end, the back-end allocates a PCSHR, and then the interface turns into an idle state. If there are no free PCSHRs remaining, the interface continues to stay in the busy mode.

2) PCSHRs and Page Copy Buffers: The back-end hardware employs several PCSHRs to concurrently handle multiple page copy commands as depicted in Fig. 6. Since a DRAM channel has a limited burst size [21], a back-end memory controller has to issue many DRAM read and write accesses to copy a page (e.g., 64 accesses for the burst size of 64 bytes). We refer to a data block retrieved from a single burst as a *sub-block*, and a PCSHR traces page copy status at the sub-block granularity. A valid (V) bit in the PCSHR indicates if it has an active command. Command type (T), PFN, and CFN (65 bits in total) are the command information obtained from the interface register on allocation. A priority (P) bit and prioritized sub-block index (PI, 6 bits) are used for critical-data-first miss handling [15]. The P bit indicates if there is a prioritized sub-block, and the PI identifies the index of the prioritized sub-block. On the allocation of a cache-fill command, the P bit is set, and the PI is deduced from the offset of the interface register. Read-issued (R), in-buffer (B), and partial-write (W) bit vectors (64 bits per vector) are used for tracing the status of sub-blocks. The R and W vectors show whether read and write transfers of sub-blocks have been issued, respectively. When a read request is sent out for a sub-block, the corresponding bit in the R vector is set to avoid generating redundant memory accesses. Sub-blocks of a page are fetched sequentially by default, unless the P bit of a PCSHR is set to prioritize a certain sub-block. The B vector indicates which sub-block data are currently available in a page copy buffer (see Fig. 3) while the page is in a transfer. The W vector is used for tracing sub-block states when writing them back to the off-package memory or filling them in the DRAM cache depending on the command type. A PCSHR also contains a set of sub-entries, each comprised of a valid (V) bit and sub-block index (SI, 6 bits) along with access-specific information such as an access ID. For every memory access that hits in a PCSHR (i.e., data miss), the SI of a memory address is saved in a sub-entry as a pending request, which will be serviced when the corresponding sub-block becomes available in the page copy buffer.

3) Handling DRAM Cache Accesses: Since NOMAD implements a decoupled tag-data management scheme, the back-end should be able to resolve data misses. For every DC access on a TLB hit, the CFN tags of PCSHRs are compared with the DC access address as shown in Fig. 6. No matched tags indicate that the page is not in a transfer, and thus the whole page data are available in the DRAM cache (i.e., data hit). Otherwise, a matched tag implies a data miss, and the access request is scheduled in a sub-entry of the matched PCSHR. Pending accesses in sub-entries are handled differently based on access types (i.e., read or write) and the status of target sub-blocks. The data miss of a write access can immediately write data in a page copy buffer and set the corresponding bit in the B vector. For a read data miss, the target sub-block may be already available in the page copy buffer or still in a transfer from the off-package memory. In case of a page copy buffer hit (i.e., corresponding bit set in the B vector), the access can be immediately serviced by reading the data from the buffer instead of accessing the on-package DRAM, which saves latency and bandwidth. Otherwise, the pending access must wait until the missed sub-block arrives.

E. Analysis of Effective Cache Access Latency

Fig. 7 visualizes the access latency of different DRAM cache schemes for two cases, i) TLB and DC tag hits and ii) TLB and DC tag misses, which are denoted as (hit, hit) and (miss, miss), respectively. A HW-based scheme exhibits longer access latency in the (hit, hit) case than OS-managed



(a) Case of TLB hit and DRAM cache hit: **(hit, hit)**

(b) Case of TLB miss and DRAM cache miss: **(miss, miss)**

Fig. 7: Comparison of effective access latency between different DRAM cache schemes. (a) In a (hit, hit) case, OS-managed schemes provide near-ideal access latency, whereas a HW-based design exhibits longer access latency to read a DC tag from the on-package DRAM. (b) In a (miss, miss) case, non-blocking miss handling of HW-based scheme and NOMAD effectively hides the latency via critical-data-first scheduling, while the blocking OS-managed scheme has a substantial latency penalty.

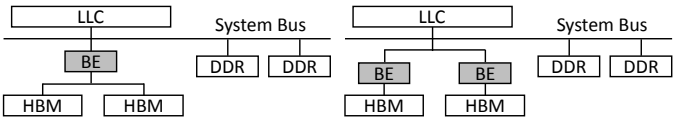
schemes because it has to access the on-package DRAM to read a DC tag. Instead, it can effectively hide the miss handling latency using MSHRs with critical-data-first scheduling [15] for the (miss, miss) situation. On the contrary, a blocking OS-managed scheme delivers ideal access latency in the (hit, hit) case since it reads a DC tag during a TLB access. However, it suffers from significantly long miss handling latency in the (miss, miss) case due to the blocking mechanism.

In contrast, the OS-managed DC design of NOMAD provides near-ideal DC access time in the (hit, hit) case, and its non-blocking miss handling mechanism via decoupled tag-data management substantially subdues the latency overhead of the blocking OS-managed scheme in the (miss, miss) case as well. The decoupled tag-data management in fact makes a synergistic effect to hide the miss handling latency. A memory request that had caused a DC tag miss is very likely to hit in a page copy buffer even though an application thread resumes immediately after the NOMAD front-end resolves only the DC tag miss. Since the cache fill is offloaded to the back-end hardware in advance by the DC tag miss handler as shown in Fig. 7b, critical-data-first scheduling in the back-end can promptly fetch the data. In our experiments, we observe that 91.6% data misses hit in page copy buffers.

The following describes other cases that are not illustrated in Fig. 7. In a (miss, hit) situation, a TLB miss penalty including a page table walk is added to the access latency of the (hit, hit) case for all DC schemes, which is unavoidable. For a (hit, miss) case, the DC access path of the HW-based scheme along the SRAM cache hierarchy is not affected. Thus, its effective access latency is the same as the (miss, miss) case without the TLB miss penalty. When it comes to OS-managed DRAM caches including both blocking and non-blocking schemes, the (hit, miss) case is equivalent to accessing a non-cacheable page, which behaves like a conventional memory system without the on-package DRAM.

F. Centralized vs. Distributed Back-end Designs

Contemporary processors utilize multiple memory channels, where a memory controller is assigned to each channel (or two) [5], [42]. In such a multi-channel design, a centralized back-end such as Fig. 8a can raise routing and bandwidth concerns since it must handle all the DC traffic. Instead, NOMAD can implement distributed back-ends as shown in Fig. 8b, where DC accesses head for different memory channels based on CFNs. Tuck et al. [44] observed that a distributed miss handling architecture is vulnerable to access imbalance, which causes frequent lockups and thus requires more resources. However, NOMAD with distributed back-ends does not have this problem because page copy commands are uniformly distributed to the back-ends of different on-package DRAM channels due to the FIFO allocation policy.



(a) Centralized back-end (b) Distributed back-ends

Fig. 8: Centralized and distributed back-end (BE) structures that keep the generality of HBMs (on-package DRAM) and DDRs (off-package memory).

G. Support for Shared Pages and Superpages

NOMAD is distinguished from prior OS-managed schemes in that it supports caching shared pages [22], [29]. When updating a PTE on a cache frame allocation, the DC tag miss handler can check if the page is shared by referencing its PPD, which can be accessed by using the PFN as an index of the PPD array. If the page is shared with multiple PTEs, the handler can retrieve them via reverse mappings and update them all [7], [33], [47]; this is omitted in Algorithm 1 for simplicity. Evicting the cache frame of a shared page can also be performed in a similar way, and it incurs no additional overhead because the eviction daemon has to read its PPD and reverse mapping anyway (lines 12-15 in Algorithm 2).

Recent operating systems utilize superpages that are a few megabytes or even gigabytes in size to extend the TLB

TABLE II: System and DRAM Configurations

System parameters	
Processor	8 out-of-order cores, 4GHz, 192 ROB entries, 8 issue width
L1 TLB	128 entries, 8-way
L2 TLB	1536 entries, 6-way
L1 I/D cache	32KB, 2-way, 2 cycles, private
L2 cache	128KB, 4-way, 4 cycles, private
L3 cache	8MB, 8-way, 17 cycles, shared
DRAM cache	1GB, shared
On-package DRAM [45]	HBM2, 2GHz, 128-bit, 64-byte burst size, 16 banks/rank, 1 rank/channel, 8 channels, 4KB row page, RoBaRaChCo, open page, tRCD-tCAS-tRP-tRAS 7-7-7-17 (ns)
Off-package memory [35]	DDR4-3200, 64-bit, 64-byte burst size, 16 banks/rank, 2 rank/channel, 2 channels, 4KB row page, RoBaRaChCo, open page, tRCD-tCAS-tRP-tRAS 14-14-14-34 (ns)
Tags-in-DRAM [24]	
Organization	1KB cache line, 4-way, LRU, perfect way prediction on hit, write-no-allocate, 32 MSHRs, 4 sub-entries/MSHR, 32 write buffers, 64-byte-granular dirty tracking
NOMAD (this work)	
Front-end	400-cycle tag management latency
Back-end	16 PCSHRs, 4 sub-entries/PSCHR, no write buffer

reach. Prior work on DRAM caches showed that there could be significant performance degradation due to over-fetching issues even with a few kilobytes of cache line sizes [22], [24], [25]. However, restricting a supportable page size only to 4KB may become a problem for some workloads whose working sets are allocated with superpages [17]. To resolve the problem, NOMAD can utilize a fragmented memory allocation technique to support superpages [38], which only brings 4KB-granular sub-pages on demand while preserving the benefits of superpage allocations. Since this technique itself is orthogonal to the key of this paper, we do not dive deep into supporting superpages in a DRAM cache in this paper.

IV. Evaluation

A. Experiment Environment

We modeled NOMAD based on the gem5 simulator [4], [32], and a chip multiprocessor (CMP) system is configured with private L1, L2 caches and a shared L3 cache. We used CACTI-7.0 [36] to determine the access time of SRAM caches, and detailed system parameters are summarized in Table II. For memory modeling, we implemented a DRAM cache controller in gem5 and integrated DRAMsim3 [30] to configure a heterogeneous memory system that consists of HBM [23] and DDR4 [35]. We selected nine memory-intensive benchmarks from SPEC2006 [16] and six graph processing benchmarks from GAPBS [3]. As described in Section II-C, the benchmarks are categorized into four different classes (i.e., Excess, Tight, Loose, and Few) based on their RMHB characteristics. Table I summarizes the workloads used in our experiments. We assigned a single-threaded program to each CPU and fast-forwarded the applications via an atomic CPU model to reach the region of interest (ROI) of the workloads and also to warm up DRAM caches. On reaching the ROI, the timing simulation of out-of-order cores initiated and ran for 500 million instructions per core (i.e., total 4 billion instructions).

To analyze OS-managed DRAM caches, we implemented multi-level TLBs and modified the OS emulation functionalities of SE-mode gem5 including address translation, memory management, etc. Shared resources used for the cache frame management (e.g., CPDs, head and tail of the free queue) are frequently accessed, so we assumed that they are kept in the on-package DRAM, which take up several pages. Caching the shared resources requires total 2MB, assuming that each CPD is 8 bytes with alignment although the actual size 42 bits.

We conservatively modeled the latency overhead of OS routines in NOMAD. The critical path of a DC tag miss handler is primarily attributed to two sequential read accesses to the on-package DRAM, one for probing the head of the free cache frame queue and another for reading its CPD. Other accesses such as updating a PTE and CPD or reading a PPD can be overlapped, so their latency impacts can be effectively hidden. Considering the average on-package DRAM access time of OS-managed schemes (i.e., 93 cycles), we have conservatively set the tag management latency to 400 cycles including a synchronization overhead. In our experiments, we observed between 400 and 3200 cycles of DC tag management delays in NOMAD because the miss handling routine is treated as a critical section that allows only one CPU to enter at a time. CPUs executing OS routines are stalled during timing simulations as if the OS occupies the CPUs. In contrast, the TDC scheme is allowed to perform multiple page copy executions in parallel by locking only the critical PTEs without an extra penalty. NOMAD is compared with four other memory schemes, and the following highlights the key of each implementation.

- **Baseline:** It models a traditional system only with the off-package memory. This design serves as the lower bound of the DRAM cache performance.
- **TiD:** This HW-based scheme adopts the tag management method of Unison Cache [24]. The cache line size of TiD is set to 1KB by idealizing a DRAM row architecture, and the DRAM cache is configured as a four-way set-associative structure with an ideal way predictor.
- **TDC:** It represents the state-of-the-art OS-managed DC design [29]. We implemented this scheme similar to the NOMAD front-end except for the blocking miss handling mechanism in order to disregard the effects of other efficiencies in the TDC scheme (e.g., lacking dirty-in-cache bits for efficient data management).
- **Ideal:** The ideal OS-managed DRAM cache has no latency penalties for tag miss handling, page copy, etc. It serves as the upper bound of the OS-managed DRAM cache performance.

B. Evaluation Results

Fig. 9, 10, and 11 show evaluation results in a nutshell. Fig. 9 plots the IPC of various memory schemes relative to the baseline. It also shows the average DC access time in CPU cycles, which is measured at DC controllers. Fig. 10 displays

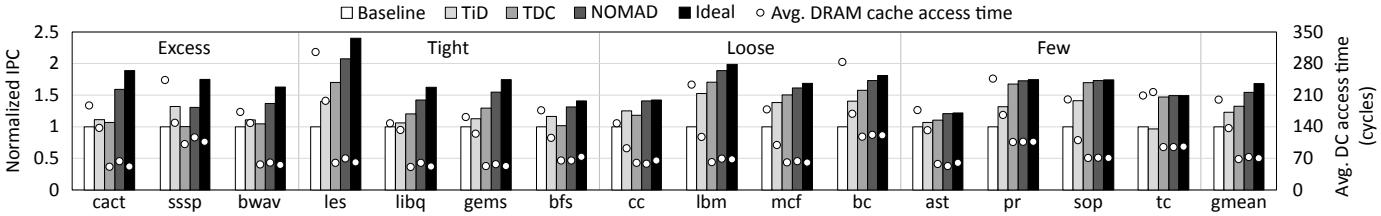


Fig. 9: IPC normalized to the baseline and average DRAM cache access time in CPU cycles.

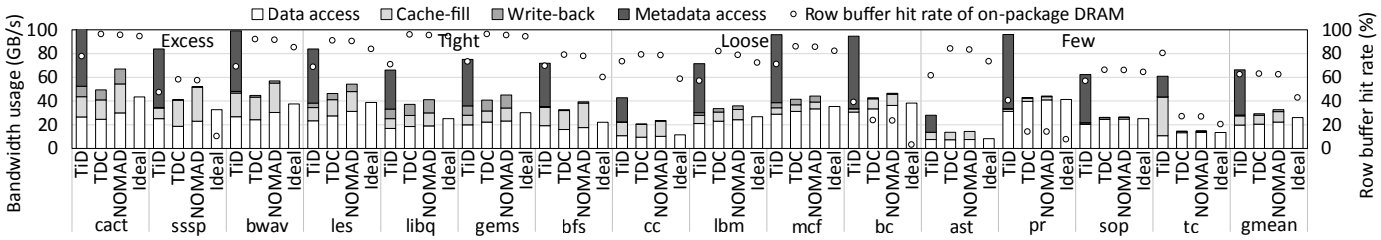


Fig. 10: Breakdown of on-package DRAM bandwidth usage and row buffer hit rates of the on-package DRAM.

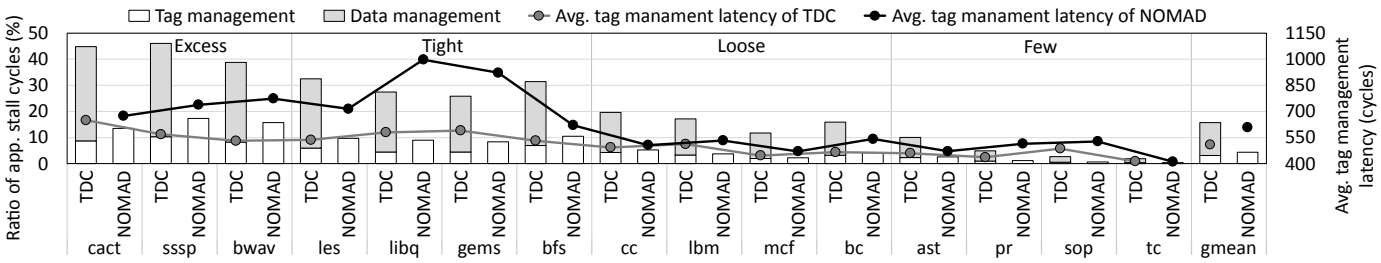


Fig. 11: Breakdown of application stall cycle ratios and the average tag management latency of OS-managed schemes.

the breakdown of the on-package DRAM bandwidth usage and the row buffer hit rates. Lastly, Fig. 11 compares application stall cycle ratios and the tag management latency of two OS-managed schemes, TDC and NOMAD.

1) Class Excess: Workloads in the Excess class have high RMHB and LLC MPMS. TDC struggles with these benchmarks in that the workloads are stalled around 43% of the runtime (Fig. 11). As a result, TDC shows almost no performance gains over the baseline although it has ideal DC access time (Fig. 9). On the other hand, TiD consumes a substantial portion of the on-package DRAM bandwidth for accessing DC metadata and miss handling (i.e., cache fills and writebacks) (Fig. 10), resulting in longer average DC access time (Fig. 9). In contrast, NOMAD achieves considerable performance speedup over the baseline by i) significantly reducing application stall cycles via non-blocking miss handling compared to TDC (Fig. 11) and ii) providing near-ideal DC access time (Fig. 9). Even for sssp that has low spatial locality, NOMAD achieves a similar performance enhancement as TiD (Fig. 9) although it uses greater miss handling bandwidth than TiD because of the larger cache line size (Fig. 10).

2) Class Tight: For les, libq, and gems in the Tight class with abundant spatial locality (i.e., high row buffer hit rates in Fig. 10), TDC shows better performance than TiD thanks to the ideal DC access time (Fig. 9) even though it has large application stall ratios around 29% (Fig. 10). TDC is also resilient to bursty LLC miss traffic such as les compared to TiD since the HW-based scheme amplifies the use of on-package DRAM bandwidth for accessing DC metadata, which leads to increases in the average DC access time (Fig. 9). However, for the workload with less spatial locality (i.e., bfs), TDC achieves almost no performance improvements over the baseline, while TiD does. This is because bfs has spatial locality far less than the 4KB page size but close to the 1KB cache line size of the HW-based scheme. For all workloads in the Tight class, NOMAD shows the greatest performance enhancements even for bfs since it can tolerate DC tag misses and has near-ideal DC access time.

3) Class Loose: For workloads in the Loose class, TDC achieves greater performance improvements than TiD (Fig. 9) since application stall ratios around 15% are at the affordable level (Fig. 11). As an exception, TiD shows better performance than TDC for cc whose LLC MPMS is low, because TiD does not require much on-package DRAM bandwidth to access DC metadata. Other workloads in this class have high LLC MPMS, so TDC performs better than TiD even for the workload with low spatial locality (i.e., bc). NOMAD shows less than 5% application stall ratios for the Loose-class benchmarks not only because the workloads have few DC tag misses but also because there are less contentions for shared

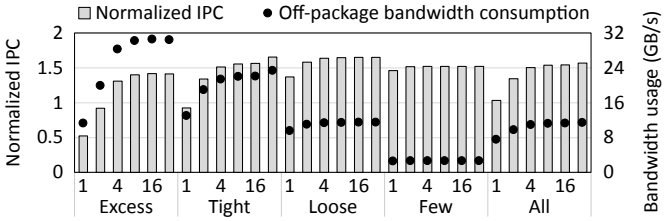


Fig. 12: Per-class average IPC relative to the baseline and the average off-package memory bandwidth consumption of NOMAD with respect to the number of PCSHRs.

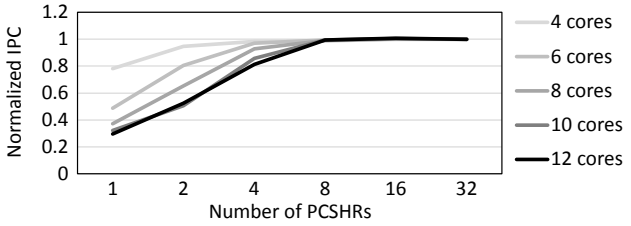


Fig. 13: Average IPC of Excess-class benchmarks with different number of PCSHRs for increasing CPU core count.

resources (e.g., critical sections in front-end routines). Thus, the average tag management latency of NOMAD approaches the minimum 400 cycles (Fig. 11), and it achieves near-ideal performance speedup (Fig. 9).

4) Class Few: At this level of RMHB, application stall ratios become almost negligible (about 4%) even for TDC (Fig. 11). Thus, TDC shows similar performance as NOMAD and the ideal scheme (Fig. 9) except for ast, which has relatively higher RMHB than other Few-class benchmarks. On the contrary, TiD suffers from long DC access time (Fig. 9) especially for workloads with high LLC MPMS (e.g., pr) because of significant on-package DRAM bandwidth consumption for metadata accesses (Fig. 10). TiD performs even worse than the baseline for tc because it consumes a substantial portion of the on-package DRAM bandwidth for cache fills due to conflict misses of the set-associative cache organization, whereas TDC and NOMAD feature fully-associative caches.

5) Summary: TDC excels at processing Loose and Few-class benchmarks by providing ideal DC access time, but its blocking miss handling mechanism cannot efficiently deal with workloads in the Excess class. On the other hand, TiD as a non-blocking HW-based cache is resilient to DC misses. However, Table I shows that workloads exhibiting high RMHB also have large LLC MPMS, which implies that TiD is likely to suffer from increased DC access time due to excessive on-package DRAM bandwidth usage for DC metadata accesses.

The decoupled tag-data management of NOMAD effectively overcomes the drawbacks of both OS-managed and HW-based DRAM cache designs by enabling non-blocking miss handling and achieving near-ideal DC access time. As a result, NOMAD improves IPC by 16.7% and 25.5% over TDC and TiD, respectively. Although NOMAD shows greater tag management latency than TDC (Fig. 11) because of contentions on shared

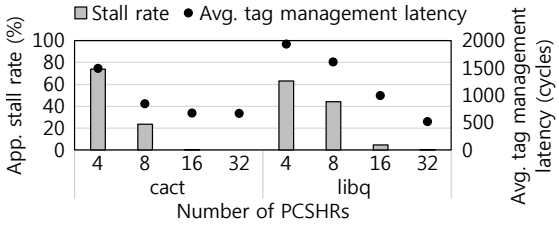


Fig. 14: Application stall rates and the average tag management latency of cact (highest RMHB) and libq (bursty RMHB) with respect to the number of PCSHRs.

resources (e.g., critical sections in front-end routines), the penalty is subdued by efficient data management offloaded to the back-end hardware. Consequently, NOMAD reduces application stall cycles by 76.1% on average compared to TDC. In addition, the latency overhead of data verification on DC accesses is insignificant in the NOMAD back-end because reading and comparing the CFN tags of PCSHRs account for only 0.21 CPU cycle based on a CACTI analysis [36]. Even if the data verification procedure takes up one full CPU cycle, it results in only 0.1% performance decrease on average.

6) Sensitivity to the Number of PCSHRs: Fig. 12 plots per-class average IPC normalized to the baseline and the average off-package memory bandwidth usage with varying PCSHRs in NOMAD. In overall, attainable miss handling bandwidth is a performance bottleneck when NOMAD is provisioned with a small number of PCSHRs. As the number of PCSHRs increases, the performance escalates until the PCSHRs provide sufficient miss handling bandwidth. For Excess-class benchmarks that have higher RMHB than the off-package memory bandwidth, eight PCSHRs offer near-max performance, and further increasing the count gives no noticeable improvements because the off-package memory bandwidth becomes the bottleneck rather than PCSHRs. Fig. 13 shows the average IPC of Excess-class benchmarks with increasing PCSHRs relative to 32-PCSHR setups for different number of CPUs. Since the performance is bounded by the off-package memory beyond eight PCSHRs, increasing the number of CPU cores in the CMP does not require a greater number of PCSHRs.

Fig. 14 compares application stall rates and the average tag management latency between cact (highest RMHB) and libq (bursty RMHB) with respect to the number of PCSHRs. The graph reveals that the bursty-RMHB benchmark (i.e., libq) experiences more severe contentions on PCSHRs than the highest-RMHB benchmark (i.e., cact). In this case, increasing PCSHRs from 16 to 32 helps reduce the average tag management latency by 48% and improves performance by 11.3%. However, other than two bursty-RMHB benchmarks (i.e., libq, gems), we observe that four PCSHRs provide Tight-class benchmarks with sufficient miss handling bandwidth. For Loose and Few-class workloads, NOMAD even with only one or two PCSHR(s) has enough missing handling bandwidth.

7) Area-optimized Design: The area overhead of NOMAD back-end hardware is mostly attributed to page copy buffers, which in total account for 4KB page size $\times$ the number of

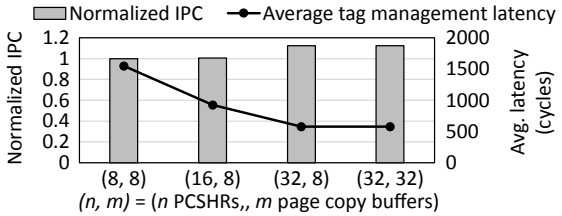


Fig. 15: Normalized IPC and the tag management latency of burst-RMHB workloads (i.e., libq, gems) with different configurations of (n PCSHRs, m page copy buffers).

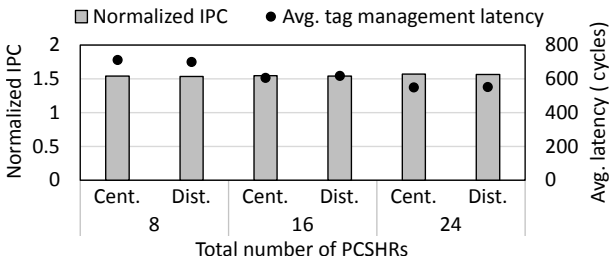


Fig. 16: Average IPC normalized to the baseline and the average tag management latency of centralized and distributed back-end designs with different numbers of PCSHRs.

PCSHRs. Other fields are only 45 bytes in size per PCSHR (i.e., 37 bytes of control bits and four sets of 2-byte sub-entries). Instead, the back-end hardware can be designed to have a smaller number of page copy buffers to reduce the area overhead but a larger number of PCSHRs to alleviate resource contentions on them particularly for bursty-RMHB workloads. In this design, the back-end has to explicitly allocate free page copy buffers to PCSHRs. Fig. 15 plots the average IPC and the tag management latency of two bursty-RMHB benchmarks (i.e., libq, gems) with different numbers of PCSHRs and page copy buffers, where (n, m) denotes n PCSHRs and m buffers. The graph shows that increasing the PCSHR count helps the bursty-RMHB workloads, but the number of page copy buffers does not have to scale up proportionally.

8) Centralized vs. Distributed Back-ends: Similar to multi-ported cache designs [43], NOMAD can implement distributed back-ends to support a multi-channel system. This requires only a minor modification to the OS to distribute page copy commands based on a few bits of CFNs. Tuck et al. [44] observed that a distributed miss handling architecture could suffer from frequent lockups due to imbalanced accesses and thus would require greater hardware resources. However, this is not the case for NOMAD because page copy commands are uniformly distributed to DC channels thanks to sequential cache frame allocations in a FIFO manner. Fig. 16 compares the average IPC and tag management latency between centralized and distributed back-ends with different numbers of PCSHRs. The graph shows that the centralized and distributed back-ends achieve similar performance. Thus, NOMAD can be implemented in both ways with comparable hardware costs.

V. Related Work

1) Scalable Designs: Prior studies [31], [39] proposed scalable TiD designs by using small cache lines (i.e., 64 bytes) and storing tags and data in the same DRAM row to optimize the tag serialization latency. To mitigate the bandwidth problems of TiD schemes, other related work [10], [18] suggested using SRAM caches to filter metadata accesses. If on-package DRAM supports an error correction code (ECC), direct-mapped DRAM caches may save the bandwidth consumption by storing tags in spare ECCs and accessing them along with the ECC metadata [42], [48].

2) Exploiting Spatial Locality: DRAM caches with small cache lines cannot exploit abundant spatial locality of memory-bound applications, so prior work proposed DC designs with large cache lines [24]–[26], [40]. To mitigate the waste of bandwidth consumption caused by large cache lines, they used selective caching mechanisms at the page or block levels.

3) Flexible and Low-cost Designs: Using large cache lines in TiD schemes imposes several implementation challenges. It incurs non-trivial design and verification efforts because cache functions (e.g., tag matching, allocation) have to be tightly coupled with conventional memory controller tasks (e.g., row activation, refresh). Also, set-associative cache organizations and selective caching mechanisms are not flexibly adaptable to different characteristics of workloads [34], [41]. Thus, OS-managed DC solutions have been proposed to overcome these limitations [2], [12], [17], [27], [29], [34], [37], [41].

4) Hybrid Approaches: The performance of an OS-managed DRAM cache is bounded by page migration bandwidth such that TDC struggles with Excess-class workloads. To blend HW-based and OS-managed solutions, related work [49] proposed a hybrid method. However, it is basically a HW-managed design with passive OS supports, which is inflexible and aggravates the hardware design complexity. In contrast, NOMAD is an OS-managed DRAM cache that can flexibly utilize various selective caching mechanisms [12], [27], [41]. It also mitigates the limitations of prior OS-only solutions by introducing a low-cost hardware support in the back-end.

5) Multi-socket Support: On-package DRAMs in a multi-socket system can be configured as partitioned or coherent caches. In a partitioned scheme, a physical frame can only be cached in a local on-package DRAM. NOMAD can be easily extended to support this organization by binding local memory nodes to the OS and applying a NUMA allocation algorithm. In a coherent-cache scheme, DRAM caches require a coherence protocol [11], [20] because a physical frame can be duplicately cached in any on-package DRAMs. Although NOMAD is an OS-managed DRAM cache, existing DC coherence protocols are compatibly applicable since the front-end sends all the mapping information to the hardware.

VI. Conclusion

The proposed NOMAD scheme enables a non-blocking OS-managed DRAM cache via tag-data decoupling. It overcomes the disadvantages of prior OS-managed and HW-based designs

by providing near-ideal DC access time and enabling non-blocking miss handling via a low-cost hardware support. NOMAD implements a flexible hybrid DRAM cache design without excessive hardware design complexity and resource overhead unlike previous hybrid DRAM caches.

VII. Acknowledgment

This work was supported by the National Research Foundation (NRF) of Korea under Grant #2021R1A2C1095162 and the Ministry of Science and ICT of Korea through the ITRC-supported program supervised by the Institute of Information and Communications Technology Planning and Evaluation under Grant #IITP-2020-0-01847.

References

[1] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual Volumne 2: System Programming," Publication No. 24593, pp. 1–660, Nov. 2020, [Online], Available: https://www.amd.com/system/files/TechDocs/24593.pdf.

[2] N. Agarwal and T. F. Wenisch, "Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory," *International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2017, p. 631–644.

[3] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," pp. 1–16, Aug. 2015, [Online], Available: https://arxiv.org/abs/1508.03619.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, May 2011, pp. 1–7.

[5] A. Biswas, "Sapphire Rapids," *IEEE Hot Chips Symposium*, Aug. 2021, pp. 1–22.

[6] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation Lookaside Buffer Consistency: A Software Approach," *International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989, p. 113–122.

[7] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*, 3rd ed. O'Reilly Media, 2005, pp. 1–1229.

[8] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, Apr. 2018.

[9] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "3.2 The A100 Datacenter GPU and Ampere Architecture," *IEEE International Solid-State Circuits Conference*, Feb. 2021, pp. 48–50.

[10] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches," *ACM/IEEE International Symposium on Computer Architecture*, June 2015, pp. 198–210.

[11] C. Chou, A. Jaleel, and M. K. Qureshi, "CANDY: Enabling coherent DRAM caches for multi-node systems," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016, pp. 1–13.

[12] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: a Hybrid Memory Page Scheduler with Machine Intelligence," *ACM International Symposium on High-Performance Parallel and Distributed Computing*, June 2019, pp. 37–48.

[13] K. I. Farkas and N. P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," *ACM/IEEE International Symposium on Computer Architecture*, June 1994, pp. 212–222.

[14] D. Foley and J. Danskin, "Ultra-Performance Pascal GPU and NVLink Interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, Mar.-Apr. 2017.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*, 6th ed. Elsevier, 2017, pp. 1–936.

[16] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, Sep. 2006, pp. 1–17.

[17] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, Jan. 2022.

[18] C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," *International Conference on Parallel Architectures and Compilation Techniques*, Aug. 2014, pp. 51–60.

[19] C. Huang, V. Nagarajan, and A. Joshi, "DCA: a DRAM-cache-aware DRAM contoller," *ACM International Conference on Supercomputing*, Nov. 2016, pp. 887–897.

[20] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3d: Mitigating the numa bottleneck via coherent dram caches," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016, pp. 1–12.

[21] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*, 1st ed. Morgan Kaufmann Publication, 2007, pp. 1–900.

[22] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient Footprint Caching for Tagless DRAM Caches," *IEEE International Symposium on High Performance Computer Architecture*, Mar. 2016, pp. 234–248.

[23] JEDEC, "High Bandwidth Memory (HBM) DRAM," JESD235, Mar. 2013, [Online], Available: https://www.jedec.org/standards-documents/docs/jesd235a.

[24] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 25–37.

[25] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," *ACM/IEEE International Symposium on Computer Architecture*, Jun. 2013, pp. 404–415.

[26] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," *IEEE International Symposium on High Performance Computer Architecture*, Jan. 2010, pp. 1–12.

[27] S. Kannan, Y. Ren, and A. Bhattacharjee, "KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2021, pp. 65–78.

[28] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *ACM/IEEE International Symposium on Computer Architecture*, May 1981, pp. 81–87.

[29] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A Fully Associative, Tagless DRAM Cache," *ACM/IEEE International Symposium on Computer Architecture*, Jun. 2015, pp. 211–222.

[30] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, Feb. 2020, pp. 106–109.

[31] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011, p. 454–464.

[32] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and E. F. Zulian, "The gem5 Simulator: Version 20.0+," pp. 1–21, July 2020, [Online], Available: https://arxiv.org/abs/2007.03152.

[33] D. McCracken, "Object-based Reverse Mapping," *Ottawa Linux Symposium*, July 2004, pp. 357–360.

[34] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2015, pp. 126–136.

[35] Micron, "DDR4 SDRAM datasheet MT40A1G16KD-062E," pp. 1–372, 2018. [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_ddr4_sdram.pdf?rev=bbe79ab4b3f547e08817f6ede1850c50

[36] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Labo-

ratories, HPL-2009-85, pp. 1–20, 2009, [Online], Available: https://www.hpl.hp.com/techreports/2009/HPL-2009-85.html.
[37] M. Oskin and G. H. Loh, "A Software-Managed Approach to Die-Stacked DRAM," *International Conference on Parallel Architecture and Compilation*, Oct. 2015, pp. 188–200.
[38] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated page: Supporting fragmented memory allocation for large pages," *ACM/IEEE International Symposium on Computer Architecture*, May 2020, pp. 913–925.
[39] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 235–246.
[40] L. E. Ramos, E. Gorbatov, and R. bianchini, "Page Placement in Hybrid Memory Systems," *ACM International Conference on Supercomputing*, May 2011, pp. 85–95.
[41] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning," *IEEE International Symposium on High Performance Computer Architecture*, Mar. 2021, pp. 598–611.
[42] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel(R) Xeon Phi(TM) Processor," *IEEE Hot Chips Symposium*, 2015, pp. 1–24.
[43] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *ACM/IEEE International Symposium on Computer Architecture*, May 1991, pp. 53–62.
[44] J. Tuck, L. Ceze, and J. Torrellas, "Scalable Cache Miss Handling for High Memory-Level Parallelism," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006, pp. 409–422.
[45] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining Caching and Migration in Hybrid Memory Systems," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2020, pp. 649–662.
[46] XILINX, "Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance," XILINX White Paper, WP485_v1.1, pp. 1–11, Jul. 2019.
[47] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th ed. Microsoft, 2017.
[48] V. Young, Z. A. Chishti, and M. K. Qureshi, "TicToc: Enabling Bandwidth-Efficient DRAM Caching for Both Hits and Misses in Hybrid Memory Systems," *IEEE International Conference on Computer Design*, Oct. 2019, pp. 341–349.
[49] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017, pp. 1–14.
[50] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM Cache Architecture for CMP Server Platforms," *IEEE International Conference on Computer Design*, Oct. 2007, pp. 55–62.