TicToc: Enabling Bandwidth-Efficient DRAM Caching for both Hits and Misses in Hybrid Memory Systems

Vinson Young†, Zeshan A. Chishti‡, and Moinuddin K. Qureshi†

†*Georgia Institute of Technology* ‡*Intel*

{vyoung,moin}@gatech.edu, zeshan.a.chishti@intel.com

***Abstract*—This paper investigates bandwidth-efficient DRAM caching for hybrid DRAM + 3D-XPoint memories. 3D-XPoint is becoming a viable alternative to DRAM as it enables high-capacity and non-volatile main memory systems. However, 3D-XPoint has several characteristics that limit it from outright replacing DRAM: 4-8x slower read, and even worse writes. As such, effective DRAM caching in front of 3D-XPoint is important to enable a high-capacity, low-latency, and high-write-bandwidth memory. There are currently two major approaches for DRAM cache design: (1) a Tag-Inside-Cacheline (TIC) organization that optimizes for hits, by storing tag next to each line such that one access gets both tag and data, and (2) a Tag-Outside-Cacheline (TOC) organization that optimizes for misses, by storing tags from multiple data lines together in a tag-line such that one access to a tag-line gets information on several data-lines. Ideally, we would like to have the low hit-latency of TIC designs, and the low miss-bandwidth of TOC designs. To this end, we propose a *TicToc* organization that provisions both TIC and TOC to get the hit and miss benefits of both.**

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

I. Introduction

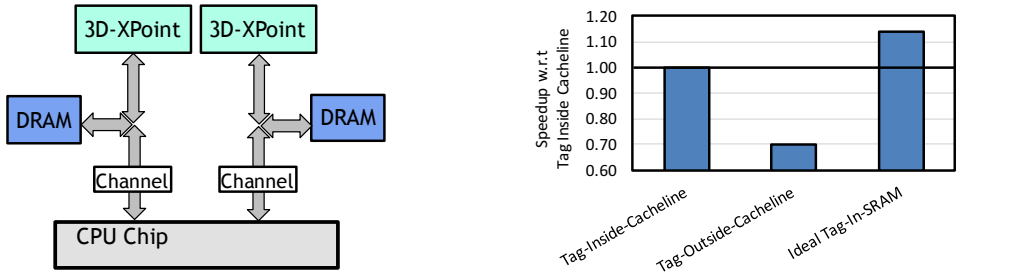
As memory systems scale, non-volatile memories or NVMs (such as, 3D-XPoint [1]) are emerging as viable alternatives to DRAM. NVMs offer the advantages of higher bit density and the ability to retain data after power outages. However, NVMs also have significant limitations that prevent them from outright replacing DRAM in the memory hierarchy. For example, 3D-XPoint is reported to have 4-8x slower read, and even slower writes compared to DRAM [2]. As such, future systems are likely to utilize hybrid memory systems [3], [4], [5], [6] consisting of both DRAM and 3D-XPoint. We focus on the setup where DRAM is operated as a hardware-managed cache for 3D-XPoint based main memory, since such a setup enables applications to benefit from the lower latency and higher write-bandwidth of DRAM and the higher capacity of 3D-XPoint without relying on any software or OS support.

Recently, there have been many works [7], [8], [9], [10], [11], [12], [13] on architecting High Bandwidth Memory (HBM) [14] caches in front of traditional DRAM main memory [15]. These works target *improving memory bandwidth* by migrating data between DRAM and HBM, and servicing most data at the higher internal/bus bandwidth of HBM. These works are effective due to HBM having dedicated higher-bandwidth channels/interfaces compared to commodity DDRx DRAM. We would like to utilize the insights learned from these works to design effective DRAM caches in front of NVMs, such as 3D-XPoint. However, we note that there are significant differences in setup and goals for a DRAM+3D-XPoint hybrid memory as compared to a HBM+DDRx hybrid memory.

First, in a 3D-XPoint based hybrid memory, 3D-XPoint and DRAM shareg the same DDRx channel interfaces [16]. Second, DRAM caches in front of 3D-XPoint target *reducing read latency* and *improving write bandwidth and endurance* of 3D-XPoint, by servicing most data at the lower latency and higher write bandwidth of DRAM. An added complexity is that the DRAM cache and 3D-XPoint are likely to sit behind the same channel [17], as depicted in Figure 1(a). Such a set-up enables a balanced configuration where every channel has DRAM backing it. However, in such a channel-sharing set-up, bandwidth needed for maintaining DRAM cache metadata now comes directly at a cost to bus bandwidth available for memory. As such, there is a renewed need for bandwidth-efficient DRAM caches. We analyze prior DRAM caching approaches, highlight cases of bandwidth-inefficiency, and rigorously target remaining bandwidth overheads to develop a bandwidth-efficient DRAM cache suitable for DRAM + 3D-XPoint systems.

We start with a baseline hit-optimized *Tag-Inside-Cacheline (TIC)* DRAM cache design [7], [11], [18]. A TIC design organizes its DRAM cache as a direct-mapped cache with tags stored inside each cacheline, such that one access can retrieve both tag and data. *TIC has good hit-latency*, as it can service cache hits in one DRAM access. However, TIC incurs bandwidth overhead on cache misses as it needs to probe the tag in DRAM in order to determine a miss. This approach of trading miss-bandwidth for hit-latency has been proven effective in situations where the cache has its own dedicated access channel, such as the HBM+DRAM hybrid memory in Intel's Knights Landing [11]. However, in a channel-sharing setup, the miss probe bandwidth directly consumes available main memory bandwidth, resulting in bandwidth inefficiency. As we show in Figure 1(b), there is a 14% performance gap between TIC and an idealized Tag-In-SRAM approach.

An alternative approach to DRAM cache design is a miss-optimized *Tag-Outside-Cacheline (TOC)* design [8], [12], [13]. A TOC design stores tags of multiple cachelines together in



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

Fig. 1. (a) Channel-Sharing Hybrid Memory, and (b) Performance of hit-optimized Tag-Inside-Cacheline (TIC) [7], miss-optimized Tag-Oustide-Cacheline (TOC) [8], and idealized Tag-In-SRAM, normalized to TIC.

a tag-only-line, such that one access to a tag-line can obtain information for multiple cachelines at once. We can bring in these bundles of tags as needed, and cache them in a small tag/metadata cache (e.g., 32KB SRAM) [8]. If the metadata cache has high hit-rate, TOC can service most hits with one DRAM access, and misses to clean lines without a DRAM access. However, if the metadata cache has low hit-rate, TOC may need two accesses to service a hit, and one access to service a miss. As such, *TOC consumes lower bandwidth on misses* than TIC; however, it consumes higher bandwidth on hits due to separate tag and data read. Overall, as shown in Figure 1(b), TOC approach performs worse than TIC due to bandwidth overheads.

We notice that TIC is good for hits, while TOC is good for misses – one can perhaps combine both approaches to get both good hit and miss bandwidth. Fortunately, it is cheap to provision both metadata at once: TIC uses spare ECC bits [11], and TOC needs to dedicate only 1.5% of DRAM cache capacity to store metadata and a 32KB SRAM for a tag/metadata cache [8]. To decide when to use TOC or TIC, one can employ a hit/miss predictor [7] that uses TIC for likely hits and TOC for likely misses. We call this proposal that provisions both TIC and TOC metadata as *TicToc*. Unfortunately, we find that naively combining TIC and TOC in fact leads to performance worse than TIC by itself. This is because maintaining and updating TOC metadata bits consumes significant DRAM bandwidth. In order for *TicToc* to be effective, we need mechanisms to reduce TOC maintenance bandwidth.

TOC incurs bandwidth overheads for the following three cases: (i) tag-check on hits, (ii) tag-update on installs, and (iii) dirty-bit-update on writebacks. Hit overhead is easily mitigated by additionally storing TIC metadata in *TicToc*. Tag updates are generally inexpensive because they occur at miss time, and miss traffic usually has good spatial locality and therefore a high metadata-cache hit-rate. Dirty-bit updates, however, remain costly because they are carried out when dirty lines are evicted from an earlier level of cache. Such evictions have poor access locality and therefore low metadata-cache hit-rates. Hence, we identify dirty bit updates as the most significant bandwidth overhead for *TicToc*.

To reduce dirty data tracking costs for TOC, we target the following two cases: initial write to a cache line, and repeated writes to the same cache line. For repeated writes, we propose to store a *DRAM Cache Dirtiness* bit alongside the line in an earlier level of cache, to track the current dirty status of the line in the DRAM cache. On a writeback to DRAM cache, we need to update the TOC metadata only if the line in the DRAM cache has changed from clean to dirty. However, many workloads write to lines only once. For such workloads, we propose *Preemptive Dirty Marking* that predicts likely-to-be-written cache lines and proactively marks those lines as dirty in the TOC at install time. This avoids needing to update dirty information at eviction time, thereby avoiding metadata-cache misses. We develop a PC-based *Write Predictor* that is 92% accurate for our Preemptive Dirty Marking.

Even after solving for hit and miss bandwidth, when data has poor reuse, installing lines and updating TOC tag can become a major source of bandwidth overhead. To mitigate that problem, we develop a *Write-Aware Bypassing* technique that reduces install and tag-update bandwidth, without increasing writes to write-constrained 3D-XPoint.

Overall our paper makes the following contributions:

Contribution-1: This paper evaluates and rigorously targets the bandwidth overheads of prior DRAM-cache organizations. We find that we can combine two tag-storage methods with a *TicToc* organization to obtain both good hit and good miss paths. However, such an approach suffers significant bandwidth cost to maintain TOC dirty information on writes.

Contribution-2: We develop two techniques to reduce the cost of tracking dirty information. *DRAM Cache Dirtiness Bit* targets reducing cost of dirty-bit updates for repeated writes to the same location, via maintaining DRAM cache dirty information alongside the line in an earlier level of cache. And, *Preemptive Dirty Marking* targets reducing cost of the initial dirty-bit update to a location, via predicting which lines are likely to be written to (with our *Signature-based Write Predictor*) and preemptively setting the dirty-bit.

Contribution-3: To reduce install bandwidth while not increasing 3D-XPoint write traffic, we develop a Write-Aware Bypass technique. This technique bypasses most clean lines by default to save install bandwidth. And, it installs most dirty and predicted write-likely lines to buffer writes to write-constrained 3D-XPoint.

Overall, our proposed TicToc organization, enables 10% speedup over TIC baseline, nearing the 14% speedup of an idealized Tag-In-SRAM approach, while needing significantly less SRAM storage (34 KB vs. 64 MB).

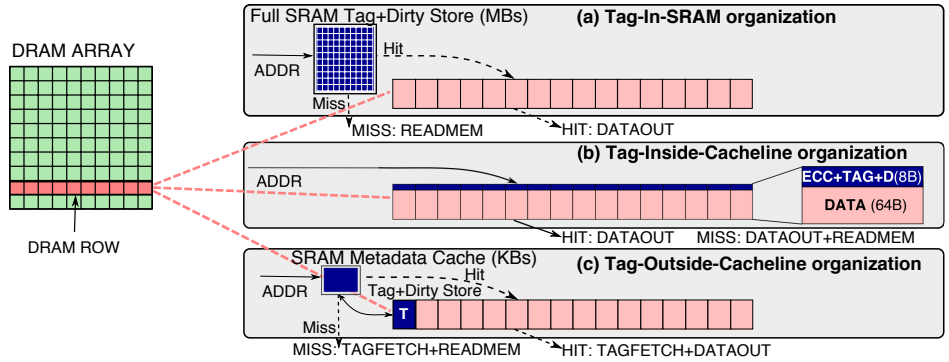


Fig. 2. DRAM cache org and flow: (a) idealized Tag-In-SRAM, (b) hit-optimized Tag-Inside-Cacheline [7], and (c) miss-optimized Tag-Outside-Cacheline [8].

II. Background and Motivation

DRAM caches are important for enabling heterogeneous memory systems to have the effective latency and bandwidth of one memory technology, and the capacity of another. It is desirable to organize DRAM caches at the granularity of a cache line to efficiently utilize cache capacity, and to minimize the consumption of main memory bandwidth [10]. A key challenge in designing such large line-granularity caches is deciding where to store the tag and dirty-bit metadata. For a moderately-sized 4GB DRAM cache with 64B lines, there would be 64 million lines. Even if each metadata required 8 bits (6 tag, 1 dirty, 1 valid bit), this would result in 64MB storage for metadata. Next, we discuss different options for DRAM cache metadata management, and their impact on SRAM storage cost and bandwidth consumption.

TABLE I
Bandwidth of DRAM Caches – ρ is Metadata-Cache Miss Ratio

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

A. Tag In SRAM

A costly method to design high performance DRAM caches is to maintain all of the tag and dirty bits in on-chip SRAM, and query the on-chip SRAM metadata to determine hit or miss, in a *Tag-In-SRAM* approach, shown in Figure 2(a). Such an approach would require 64MB of SRAM for a 4GB cache (>20MB with sectoring [10], [19]). Table I shows the DRAM bandwidth consumption for such an approach. SRAM metadata is queried first to determine hit or miss. A hit can be serviced with one DRAM access to data. A miss can be serviced without a DRAM access to data, unless installing the new line evicts a dirty line. A write needs one DRAM access. Such a design represents the minimum DRAM bandwidth needed for DRAM cache maintenance, and an upper-bound for performance. We aim to achieve Tag-In-SRAM performance at low SRAM cost.

B. Tag Inside Cacheline

To reduce SRAM storage costs, one could store tags inside each line in DRAM [7], [11], [18] in a *Tag-Inside-Cacheline (TIC)* approach, shown in Figure 2(b). TIC optimizes for hit-latency by using a direct-mapped design and storing tag inside each data-line such that one access can retrieve both tag and data. Direct-mapped organization enables the controller to know which location to access, without waiting for tags.

Table I shows the bandwidth of such an approach. Hits are serviced with one DRAM access that retrieves both tag and data. However, misses also need to access tag in DRAM. As such, TIC is effective for hit-latency, but consumes extra bandwidth on misses. This approach of trading miss-bandwidth for hit-latency has been proven effective in commercial products [11], and, as such, we use the TIC organization [7] as our baseline.

Setup: We store metadata alongside data in unused ECC bits similar to Intel's Knights Landing [11]. TIC additionally employs a <1KB SRAM hit-miss predictor to guide when to access cache+memory either in a parallel or serial manner. We include bandwidth-reducing enhancements from Chou et al. [18], such as DCP to reduce writeback probe.

C. Tag Outside Cacheline

Another option with reduced SRAM storage costs, is to store metadata lines in a separate area of DRAM and bring them in as needed in a *Tag-Outside-Cacheline (TOC)* [8], [12], [13] approach, shown in Figure 2(c). To determine hit or miss, TOC first accesses a metadata line to get tag+dirty information for the requested data line, then routes the request appropriately to DRAM cache or to memory. Of note, each of these metadata lines actually stores metadata of several adjacent data lines. An enhanced design [8] caches these metadata lines in a small metadata cache, to amortize metadata lookup. Table I shows the bandwidth consumption of such an approach. In case of a metadata-cache hit, TOC performs similar to idealized Tag-In-SRAM. But, in case of a metadata-cache miss, TOC needs additional bandwidth to access the metadata. Overall, TOC has the potential for reducing miss bandwidth, but can suffer from significant bandwidth overhead when the metadata-cache has poor hit rate (due to poor spatial locality).

Setup: We assume 1-byte metadata (6 tag, 1 dirty, 1 valid bits), and 64 tags stored in each metadata entry. The metadata are stored in a separate part of DRAM, consisting of 64MB out of the 4GB DRAM capacity. Recently accessed metadata are stored in a 512-entry metadata cache, which requires 32KB of SRAM. We employ a direct-mapped organization and a hit-miss predictor [7] for latency and bandwidth considerations.

D. Insight: Combine Metadata Approaches

The TIC approach has good *hit-latency*, but suffers from extra miss bandwidth. Whereas, the TOC approach has good

miss bandwidth but incurs extra hit bandwidth. Our insight is that if one could use TIC for hits and TOC for misses, then one could potentially achieve both good hit and miss bandwidth.

We note that provisioning metadata for both TIC and TOC is relatively inexpensive: TIC simply uses spare ECC bits [11], and TOC needs to dedicate only ~1.5% of DRAM cache capacity to store metadata lines and employs 32KB SRAM for its metadata cache [8]. However, we need an effective design that can use TIC for hits and TOC for misses. In addition, we need to solve for TOC bandwidth costs as such a combined proposal still needs significant bandwidth to maintain TOC tag and dirty metadata. We discuss methodology before design.

III. Methodology

A. Framework and Configuration

We use USIMM [20], an x86 simulator with detailed memory system model. We extend USIMM to include a DRAM cache. Table II shows the configuration used in our study. We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache). All caches use 64B line size. The baseline L4 is a 4GB DRAM-cache [11], which is direct-mapped and places tags with data in unused ECC bits. DRAM cache parameters are based on DDR4 DRAM [15]. The main memory is based on 3D-XPoint [1], [2], [21]: the read latency is ~6X, the write latency is ~24X that of DRAM, and there are 64 rowbuffers each 256B in size.

TABLE II
System Configuration

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

B. Workloads

We use a representative slice of 2-billion instructions selected by PinPoints [22], from benchmark suites that include SPEC 2006 [23] and GAP [24]. For SPEC, we pick a subset of high memory intensity workloads that have at least 2 L3 misses per thousand instructions (MPKI). The evaluations execute benchmarks in rate mode, where all eight cores execute the same benchmark. In addition to rate-mode workloads, we also evaluate 4 mixed workloads, which are created by randomly choosing 8 of the 17 SPEC workloads. Table III shows L3 miss rates, and memory footprints for the 8-core rate-mode workloads in our study. We perform timing simulation until each benchmark in a workload executes at least 2 billion instructions. We use weighted speedup to measure aggregate performance of the workload normalized to the baseline and report geometric mean for the average speedup across all the 17 workloads (11 SPEC, 2 GAP, 4 MIX).

TABLE III
Workload Characteristics

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

IV. TicToc Design

We want the hit-path of TIC, the miss-path of TOC, all without the cost to maintain TOC metadata. This section is organized as follows: we describe how to provision and effectively utilize both TIC and TOC, describe how to reduce TOC maintenance cost, and show effectiveness of design.

A. TicToc Metadata Organization

We propose TicToc, a metadata organization that combines the benefits of TIC and TOC DRAM cache designs. Figure 3 shows the metadata organization of our *TicToc* design. TicToc provisions TIC metadata – tag-bits and dirty-bit are stored inside the cacheline in unused ECC bits, similar to commercial designs [11]. TicToc also provisions TOC metadata – metadata is stored in dedicated metadata lines, taking up 1.5% of DRAM capacity, and cached in a 32KB on-chip metadata cache. While provisioning both TIC and TOC metadata is cheap, the complexity lies in using TIC and TOC metadata effectively to save on bandwidth for hits, misses, and writes.

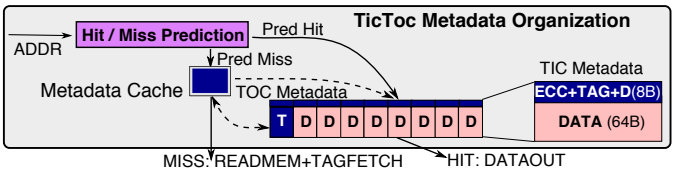


Fig. 3. TicToc Organization queries hit/miss predictor to use TIC for hits and TOC for misses. TicToc enables good hit latency, and good hit/miss bandwidth.

1) TicToc Operation: Figure 3 shows the operation of TicToc. We want to use TIC metadata for hits and TOC metadata for misses. Our key insight is that one can use hit/miss prediction [7], [25] to help guide when to use which metadata. Hit/miss predictors have previously been used to hide the serialization latency that can occur from waiting on last-level cache miss response before accessing main memory. A hit/miss predictor works by predicting which cache accesses are likely to miss, and sending both cache and memory requests in parallel to avoid serialization. We leverage a hit/miss predictor [7] to guide TicToc to use TIC metadata on likely-hit and TOC metadata on likely-miss. The common result: a hit is serviced in one cache access (TIC path), a miss with clean eviction directly goes to memory (TOC path), and a miss with dirty eviction goes to cache and memory (TIC path).

2) TicToc Effectiveness: To analyze effectiveness of TicToc, Figure 4 and Figure 5 show proportion of channel bandwidth used for useful operations, install operations, and assorted maintenance operations, for baseline TIC and proposed TicToc. Useful operations include 3D-XPoint Read and Write, and DRAM Cache Hit and Writeback. Install operations refer to cache installs, which are important for improving hit-rate but incur bandwidth to write the line to DRAM. Lastly, Maintenance operations refer to bandwidth-wasting operations used to confirm whether a line is cache resident or not: miss probes for TIC, and accessing/updating TOC metadata for TOC.

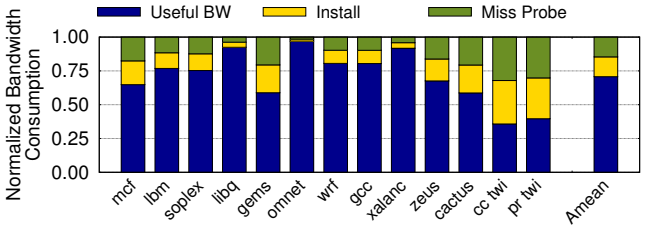


Fig. 4. Breakdown of bus bandwidth consumption for TIC organization [7]. Workloads with low hit-rate waste significant bandwidth to confirm misses.

As expected, Figure 4 shows that TIC wastes bandwidth when probing the DRAM cache to confirm misses. The proposed TicToc can utilize TOC to reduce such miss probes. However, Figure 5 shows that TicToc actually fares worse due to needing bandwidth to maintain TOC tag and TOC dirty-bit.

TOC tag-updates happen when a new line is installed in the cache on a miss. Note that a large fraction of misses happen when a workload starts accessing a new page. Therefore, misses generally have high spatial locality. In such cases, our small metadata cache does a good job in amortizing metadata updates.

TOC dirty-bit-updates, on the other hand, occur upon eviction of a dirty line from an earlier level of cache. Evictions generally have poor spatial and temporal locality; therefore, eviction-related TOC dirty-bit updates often miss in the metadata cache and consume additional bandwidth.

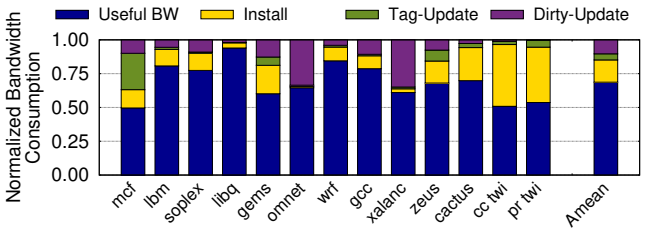


Fig. 5. Breakdown of bus bandwidth consumption for proposed TicToc organization. Write-heavy workloads waste bandwidth updating TOC dirty-bit.

B. Reducing Dirty-Bit Tracking Costs

The main source of bandwidth overhead of TicToc is maintaining the dirty-bit information in TOC metadata. To mitigate this overhead, we need effective methods to cut down dirty bit updates. We explain the difficulty in doing that before describing our solution.

1) Understanding Dirty-bit Updates: The dirty-bit update procedure starts upon an eviction of a dirty line from L3. First, we need to probe the tag&dirty-bit of the destination set of

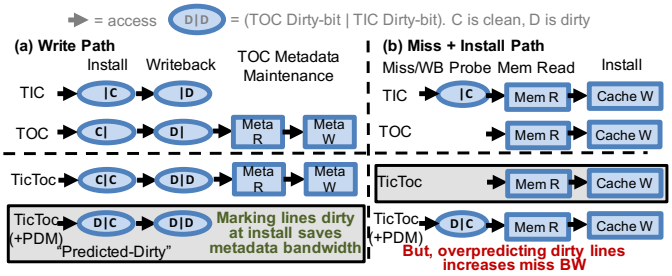


Fig. 6. Bandwidth for a typical (a) write path and (b) miss+install path. TicToc+PDM adds "Predicted-Dirty" state, where TOC dirty-bit is installed as dirty but TIC dirty-bit is installed as clean. Installing lines in Pred-Dirty can (a) save TOC dirty-bit update, but (b) increase miss cost. Using Pred-Dirty only for write-likely lines can save bandwidth

the L4 cache to see if we can directly overwrite the L4 cache. In case of a tag mismatch, one would first need to evict (and potentially write back) the conflicting L4 line. However, the common case is that the line evicted from L3 is resident in L4 cache. Chou et al. [18] eliminates the tag-check for this case by maintaining a *DRAM Cache Presence bit (DCP)* along side every L3 line. If the DCP bit is set, the line is present in L4 and can be safely overwritten. Second, the data evicted from L3 is written to the L4. Third, we need to update any pertinent tag and dirty-bit metadata. The tag-update for TIC and TOC is uncommon, as typically L3 writebacks hit in L4. The dirty-bit-update for TIC is sent along with L4 install, therefore it does not incur bandwidth overhead. However, Figure 6(a)[TOC,TicToc] shows that the dirty-bit update for TOC often needs to be separately queried and potentially updated, which results in bandwidth overhead.

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

2) Reducing Repeated TOC Dirty-bit Checks: Our key insight is as follows: if one knew that a line is already marked dirty in the L4 cache, then there is no need to access/update the L4 dirty status on L3 writebacks. In presence of such information, the dirty bit check/update cost would be incurred only when the L4 line status changes from clean to dirty.

DRAM Cache Dirtiness: To enable this optimization, we propose to additionally store a *DRAM Cache Dirtiness bit (DCD)* alongside the DCP [18] next to each line in the L3 cache. While DCP tracks whether the L3 line is also resident in L4, DCD stores the dirty status of that line in L4. We set the DCD, when a dirty line is read from L4. On an L3 writeback, we check both the DCP and DCD. If both DCD and DCP are set, we know the line is resident in L4 and already marked dirty in the TOC metadata – tag and dirty-bit will be unchanged and we do not need to fetch TOC. Hence, DCP reduces tag checks when tag will not be modified, and DCD reduces dirty-bit checks when dirty-bit will not be modified.

Figure 7 shows that DCD reduces dirty-bit updates for many workloads that repeatedly write to same lines (e.g., *omnet*, *soplex*). However, there are other write-intensive workloads (e.g., *zeusmp*) where most L4 lines are written only once – we want to reduce dirty-bit updates for those workloads as well.

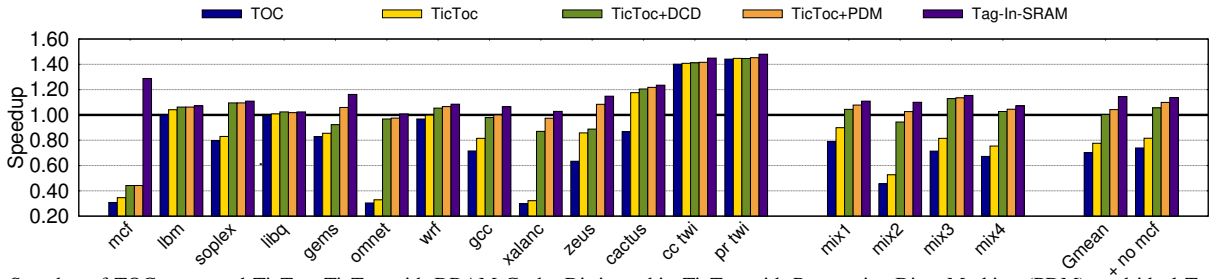


Fig. 7. Speedup of TOC, proposed TicToc, TicToc with DRAM Cache Dirtiness bit, TicToc with Preemptive Dirty Marking (PDM), and ideal Tag-In-SRAM, normalized to TIC. TicToc+PDM performs near ideal for most workloads.

3) Reducing Initial TOC Dirty-bit Update: For workloads that write-once to lines, we make the following key observation: if the dirty bit in the TOC tag is pre-emptively marked at line install time, then one can avoid the TOC clean-to-dirty update that would have been incurred at L3 eviction time. We call this approach *Preemptive Dirty Marking (PDM)*.

Preemptive Dirty Marking: Figure 6 shows the typical write and miss+install bandwidth for TicToc and the approach that preemptively marks TOC dirty-bit. Figure 6(a)[TicToc] shows that a typical write path needs 4 accesses: a normal line would incur clean install, a write, TOC dirty-bit read and TOC dirty-bit write. Figure 6(a)[TicToc+PDM] shows that PDM can limit writes to 2 accesses. We add a new dirty state of "Predicted-Dirty," where TOC dirty-bit is marked as dirty but TIC dirty-bit is marked as clean. If we install lines in "Predicted-Dirty" with PDM, the TOC dirty-bit is set at install time, and the TOC clean-to-dirty update can be avoided.

However, while early marking can save bandwidth on writes, PDM incurs a different problem on miss path. Figure 6(b)[TicToc] shows that a typical miss+install path needs 2 accesses: TOC metadata informs residence and dirtiness so that miss+install can be accomplished with a memory read and a DRAM cache install. However, Figure 6(b)[TicToc+PDM] shows that PDM can increase miss+install to 3 accesses. For instance, if a clean line has been preemptively marked as dirty in the TOC dirty-bit, we would need to read the DRAM cache line in preparation for eviction of a dirty line, thereby adding an extra DRAM read. Thus, being aggressive in marking lines as "Predicted-Dirty" with PDM will save write bandwidth, but it can come at the cost of increasing miss cost for clean lines.

Thus, if we install a write-likely line as clean, it will pay additional miss cost [TicToc] – if we install a write-unlikely line as "Predicted-Dirty," it will pay additional cost to update TOC dirty bit [TicToc+PDM]. To address these costs, we make the following key insight: if we can accurately predict write behavior at install time and use PDM only for write-likely lines, we can reduce both TOC dirty bit update and TIC miss probe bandwidth.

Write Predictor: For accurate write-classification for PDM, we develop a *Signature-based Write Predictor (SWP)* to predict likelihood that an incoming line will be written. SWP employs a sampling PC-based prediction, inspired by SHiP [26], [27]. Figure 8 shows structures and operation of SWP. SWP consists of write-behavior observation, learning, and prediction.

Observation is done by maintaining signature (installing-PC in this case) and a written-to bit inside the metadata of each

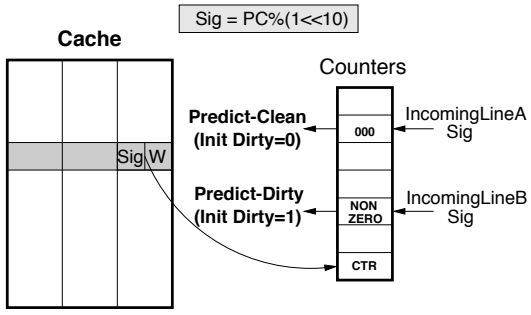


Fig. 8. Signature-based Write Predictor learns which sigs correspond to eventual write, to aid PDM technique.

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

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

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

Accuracy of Write Predictor: Effectiveness of PDM is contingent on good prediction of write-likely (dirty) lines to reduce dirty-bit update cost, and write-unlikely (clean) lines to reduce miss-probe cost. Figure 9 shows fraction of lines that are predicted clean or dirty, and actually end up clean or dirty. SWP has a high prediction accuracy (92% on average), and enables PDM to save most dirty-update and miss-probe costs.

4) Effectiveness of Dirty-Tracking Optimizations:

Performance: Figure 7 shows the speedup of TOC, our TicToc, TicToc with DCD, TicToc with PDM, and idealized Tag-In-SRAM, normalized to TIC approach. TOC performs poorly due to poor metadata-cache hit-rate, for 30% slowdown. TicToc reduces hit bandwidth, for 22% slowdown. Adding DCD to TicToc avoids dirty-bit tracking for repeated writes, bringing performance on par with TIC. Adding PDM further

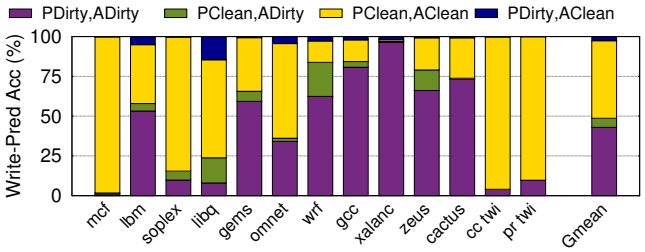


Fig. 9. Accuracy of Write Prediction (P=predicted, A=actual). Low PClean/ADirty and PDirty/AClean reflects accurate write-behavior prediction

reduces bandwidth overhead by avoiding the initial dirty-bit updates. Notably, TicToc+PDM achieves near ideal Tag-In-SRAM performance for most workloads, resulting in a 10% average speedup (excluding the worst-case *mcf*). Some workloads, however, exhibit significant gap to ideal. We analyze bandwidth consumption to gain insight into this problem.

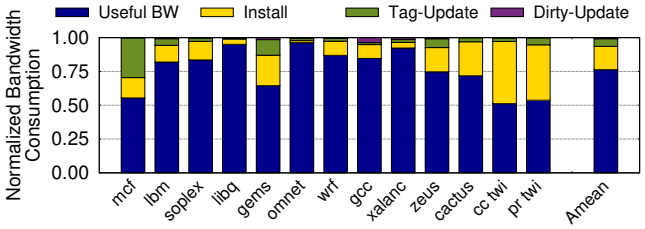


Fig. 10. Breakdown of bus bandwidth for dirty-optimized TicToc. Dirty-bit updates are greatly reduced.

Bandwidth: Figure 10 shows the bandwidth breakdown of TicToc + dirty-bit optimizations. Overall, our approach reduces nearly all of the TOC dirty-bit update bandwidth (decreased fraction from 10% to 0.8%) and frees up bandwidth for useful reads and writes. However, we note that installing lines and updating the TOC-tag now becomes the main source of DRAM cache bandwidth overhead. We target this overhead next.

V. Reducing Install Cost With Write-Aware Bypass

When data has poor reuse, installing lines and updating TOC metadata waste bandwidth. In such cases, employing a DRAM cache could actually hurt performance, as the line install and tag maintenance operations needlessly steal bus bandwidth from memory accesses. Therefore, we need effective mechanisms to reduce the cost of unnecessary installs.

Insight – Write-Aware Bypassing: Prior work has proposed cache bypassing [18], [28], [29] to avoid unnecessary installs. On an L3 miss, one can bypass the DRAM cache and install the line only in L1/L2/L3 caches, thereby saving the DRAM cache install bandwidth. However, bypassing must be done selectively and carefully, otherwise it may increase writes to 3DXPoint and degrade performance, endurance, and power.

A. Design of Write-Aware Bypassing

Figure 11 shows our Write-Aware Bypassing policy. We start with the default 90%-bypass policy proposed in [18], which bypasses 90% of all installs. While such aggressive bypassing was shown to work well for an HBM+DDR hybrid memory [18], we note that it can increase write traffic to the write-constrained 3D-XPoint memory. To address this problem, we add write awareness to the bypass policy. We augment the

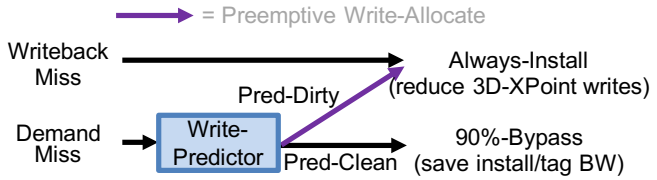


Fig. 11. Write-Aware Bypass. Reduce install bandwidth by bypassing most write-unlikely lines. Reduce 3D-XPoint writes by installing write-likely lines.

default bypass policy with a write-allocate condition, which requires that dirty L3 evictions would *always* install DRAM cache lines. Thus, the DRAM cache would act as a write buffer for 3D-XPoint memory. Unfortunately, the drawback of such an approach is that installing DRAM cache lines at the time of L3 evictions may result in significant tag-update costs. L3 evictions often have poor spatial locality, causing TOC tag updates carried out at L3 eviction to exhibit poor metadata cache hit rates and incur extra DRAM accesses.

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

B. Effectiveness of Write-Aware Bypassing

Bandwidth: To understand the effectiveness of our install and metadata-update reducing optimizations, we show the bandwidth breakdown of our approach in Figure 12. Overall, we find that install-reducing optimizations can eliminate nearly all of the install bandwidth overheads and leave much more bandwidth for useful reads and writes. In total, the combination of our cache bandwidth reducing optimizations improves fraction of bandwidth going to useful operations (servicing reads / writes) from 70% to 90% on average.

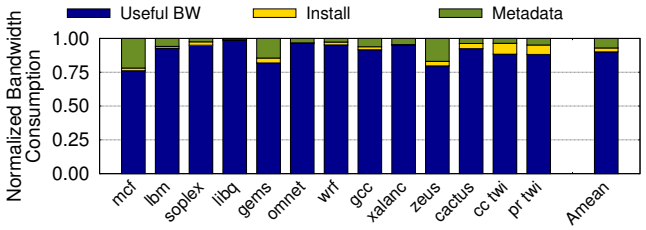


Fig. 12. Breakdown of bus bandwidth for dirty-optimized TicToc w/ Write-Aware Bypassing. Installs are mitigated.

Performance: Figure 13 shows the performance of TicToc with dirty-optimizations, TicToc with 90%-bypass, TicToc with 90%-bypass and write-allocate, and TicToc with 90%-bypass and preemptive write-allocate, relative to TIC.

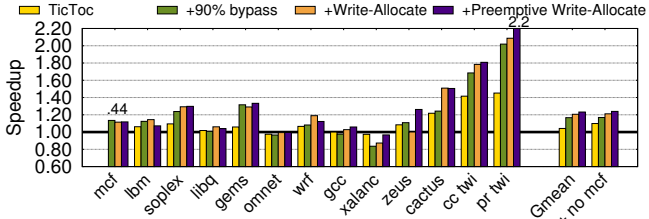


Fig. 13. Speedup of TicToc organization, adding 90%-bypass, adding Write-Allocate, and adding Preemptive Write-Allocate, relative to TIC approach.

TicToc with dirty-bit optimizations does well for most workloads, with an average speedup of 4.2%, but can suffer for workloads with poor spatial locality and low hit-rate (e.g., *mcf*). TicToc with 90%-bypass reduces install and TOC tag-update cost to improve speedup to 16.7%. Notably, the slowdown for *mcf* is mitigated. TicToc with 90%-bypass and write-allocate enables effective write-buffering to improve speedup to 20.6%. Finally, TicToc with 90%-bypass and preemptive write-allocate further amortizes TOC metadata-update (e.g., *zeusmp* and *pr twi*) to improve speedup to 23.2%.

C. Putting it all together

This work targets all DRAM cache maintenance bandwidth operations to achieve a bandwidth-efficient (>90% of channel bandwidth to useful operations) and low SRAM storage overhead (34KB) DRAM cache organization: *TicToc* improves hit and miss bandwidth, *DRAM Cache Dirtiness bit* and *Preemptive Dirty Marking* reduces dirty-bit-tracking bandwidth, and *Write-Aware Bypass* reduces install and tag-tracking bandwidth. TicToc enables 23.2% speedup with ~34KB SRAM.

VI. Results and Discussion

A. Storage Requirements

We analyze the SRAM storage needs of TicToc organization. TicToc requires structures from its component TIC and TOC organizations. From TIC, we need ~1KB for PC-based hit/miss prediction [7], and 1 bit alongside each L3 line for DRAM Cache Presence bit to avoid tag-check for writes to resident lines [18]. From TOC, we need 32KB for a metadata cache [8].

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

TABLE IV
Storage Requirements of TicToc

TicToc Component	SRAM Storage
Hit-Miss Predictor [7]	1 KB
DRAM Cache Presence [18]	1-bit / L3-line
Metadata Cache [8]	32 KB
DRAM Cache Dirtiness	1-bit / L3-line
Signature-based Write Predictor	1 KB
TicToc	34KB + 2-bits/L3-line

VII. Related Work

A. Line-based DRAM Caches

In our work, we utilize and combine the two major types of line-granularity DRAM cache designs: *Tag-Inside-Cacheline (TIC)* and *Tag-Outside-Cacheline (TOC)* approaches.

TIC designs [7], [11], [18], [30], [31], [32] organize their cache as direct-mapped and store tag inside the cacheline, such that one access can retrieve both tag and data. Such approaches are optimized for hits, but pay bandwidth to confirm misses [7]. BEAR [18] proposes several enhancements to reduce bandwidth cost of cache maintenance: we include its DRAM Cache Presence bit that targets reducing write probe in our baseline TIC design, we compare with Bandwidth-Aware Bypass with 90%-bypass in Figure 13, but we do not include Neighboring Tag Cache since current implementations cannot obtain neighboring tag for free [11]. We use BEAR as our TIC component of TicToc, and improve upon TIC miss-bandwidth cost to enable a scalable bandwidth-efficient DRAM cache. Associativity [31] enhancements can be easily incorporated.

TOC designs [8], [9], [12], [33], [34] store tags in a separate area of the DRAM cache and fetch them as needed. Early designs were highly associative and would need a serial tag-data lookup [9]. Some enhancements used tag-prefetching [33] or way-prediction [34] to avoid this serialized tag lookup. Others used direct-mapped structure [8], [12] to avoid serialized tag lookup, with one employing a tag cache [8] to reduce bandwidth of tag lookup as well. We use Timber [8] as our TOC component of TicToc, and improve upon TOC bandwidth overhead to enable a scalable bandwidth-efficient DRAM cache.

B. Page-based DRAM Caches

Alternatively, one can use large-granularity DRAM caches to amortize tag and metadata overhead, in hardware [10], [13] or software [35], [36], [37], [38]. Hardware-based approaches have bandwidth overheads similar to TOC designs. Software-based approaches need OS support and are out of scope for our investigation. Swap-based memory management techniques [39], [40], [41], [42], [43] also exist, but have increased write cost due to needing to write back clean pages (inefficient for 3D-XPoint).

C. On Reducing Dirty-bit Tracking

Tracking dirty-status of cachelines efficiently with low SRAM storage is a known difficult problem. Many works limit the amount of lines that can be kept dirty [25], [44], [45], [46], to reduce SRAM storage needed to track dirty lines. However, for our work, we target a DRAM + 3D-XPoint system, which is often constrained by 3D-XPoint write bandwidth. Such mostly-clean caching techniques hamper the ability for the DRAM cache to act as an effective write buffer for 3D-XPoint – and can cause slowdown. Our approach, on the other hand, does not impose any write limitation and instead uses architectural techniques (DCD+PDM) to reduce over 90% of the bandwidth cost to track dirty information, with only 34KB of SRAM.

VIII. Conclusion

This paper investigates bandwidth-efficient DRAM caching for hybrid DRAM + 3D-XPoint memories. Our proposal combines two major approaches for DRAM cache design: (1) a Tag-Inside-Cacheline (TIC) organization that optimizes for hits by storing tag alongside data, and (2) a Tag-Outside-Cacheline (TOC) organization that optimizes for misses by storing tags from multiple data lines together in a tag-line. We call this proposal TicToc. To enhance TicToc, we propose three new optimizations in order to reduce the bandwidth cost of updating and maintaining cache metadata. Together, our evaluations show that our TicToc organization enables 10% speedup over the baseline TIC, nearing the 14% speedup possible with an idealized DRAM cache design with 64MB of SRAM tags, while needing only 34KB SRAM.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our colleagues from the Memory Systems Lab for their critique and suggestions. This work was supported, in part, by a grant from the Semiconductor Research Center (SRC) and a gift from the Intel corporation.

REFERENCES

[1] Intel and Micron, "A revolutionary breakthrough in memory technology," 2015.

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

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

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09*, (New York, NY, USA), pp. 24–33, ACM, 2009.

[5] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 664–669, July 2009.

[6] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao, "Architectural design for next generation heterogeneous memory systems," in *International Memory Workshop (IMW)*, 2010.

[7] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *MICRO '12*, pp. 235–246, Dec 2012.

[8] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE CAL*, vol. 11, pp. 61–64, July 2012.

[9] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *MICRO '11*, (New York, NY, USA), pp. 454–464, ACM, 2011.

[10] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA '13*, (New York, NY, USA), pp. 404–415, ACM, 2013.

[11] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, pp. 34–46, Mar 2016.

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

[13] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *MICRO '14*, pp. 25–37, IEEE, 2014.

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

[15] JEDEC, *DDR4 SPEC (JESD79-4)*, 2013.

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

[17] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora, "Cascade lake: Next generation intel xeon scalable processor," *IEEE Micro*, vol. 39, pp. 29–36, March 2019.

[18] C. Chou, A. Jaleel, and M. K. Qureshi, "Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches," in *ISCA '15*, (New York, NY, USA), pp. 198–210, ACM, 2015.

[19] J. B. Rothman and A. J. Smith, "Sector cache design and performance," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, pp. 124–133, Aug 2000.

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

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

[22] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *MICRO '14*, pp. 81–92, Dec 2004.

[23] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[24] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[25] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO '12*, pp. 247–257, IEEE, 2012.

[26] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *MICRO '11*, (New York, NY, USA), pp. 430–441, ACM, 2011.

[27] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA '17)*, 2017.

[28] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Trans. Comput.*, vol. 57, pp. 433–447, Apr. 2008.

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

[30] C. Chou, A. Jaleel, and M. K. Qureshi, "Candy: Enabling coherent dram caches for multi-node systems," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[31] V. Young, C. Chou, A. Jaleel, and M. K. Qureshi, "Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction," in *ISCA '18*, pp. 328–339, June 2018.

[32] V. Young, P. J. Nair, and M. K. Qureshi, "Dice: Compressing dram caches for bandwidth and capacity," in *ISCA '17*, (New York, NY, USA), pp. 627–638, ACM, 2017.

[33] C.-C. Huang and V. Nagarajan, "Atcache: reducing dram cache latency via a small sram tag cache," in *PACT '14*, pp. 51–60, ACM, 2014.

[34] Z. Wang, D. A. Jimnez, T. Zhang, G. H. Loh, and Y. Xie, "Building a low latency, highly associative dram cache with the buffered way predictor," in *SBAC-PAD '16*, pp. 109–117, Oct 2016.

[35] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless dram cache," in *ISCA '15*, (New York, NY, USA), pp. 211–222, ACM, 2015.

[36] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for tagless dram caches," in *HPCA '16*, pp. 237–248, IEEE, 2016.

[37] G. H Loh, N. Jayasena, J. Chung, S. K Reinhardt, M. O'Connor, and K. McGrath, "Challenges in heterogeneous die-stacked and off-chip memory systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, 02 2012.

[38] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *MICRO '17*, (New York, NY, USA), pp. 1–14, ACM, 2017.

[39] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *MICRO '14*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2014.

[40] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *MICRO '14*, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2014.

[41] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "Silc-fm: Subblocked interleaved cache-like flat memory organization," in *HPCA '17*, pp. 349–360, Feb 2017.

[42] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *HPCA '17*, pp. 433–444, Feb 2017.

[43] A. Kokolis, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," *HPCA '19*, pp. 596–608, 2019.

[44] C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3d: Mitigating the numa bottleneck via coherent dram caches," in *MICRO '16*, pp. 1–12, Oct 2016.

[45] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for GPU architectures," in *HPCA '13*, 2013.

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