BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches

Chiachen Chou† Aamer Jaleel‡ Moinuddin K. Qureshi†

†School of Electrical and Computer Engineering
Georgia Institute of Technology
{cc.chou, moin}@ece.gatech.edu

‡ NVIDIA
ajaleel@nvidia.com

Abstract

Die stacking memory technology can enable gigascale DRAM caches that can operate at 4x-8x higher bandwidth than commodity DRAM. Such caches can improve system performance by servicing data at a faster rate when the requested data is found in the cache, potentially increasing the memory bandwidth of the system by 4x-8x. Unfortunately, a DRAM cache uses the available memory bandwidth not only for data transfer on cache hits, but also for other secondary operations such as cache miss detection, fill on cache miss, and writeback lookup and content update on dirty evictions from the last-level on-chip cache. Ideally, we want the bandwidth consumed for such secondary operations to be negligible, and have almost all the bandwidth be available for transfer of useful data from the DRAM cache to the processor.

We evaluate a 1GB DRAM cache, architected as Alloy Cache, and show that even the most bandwidth-efficient proposal for DRAM cache consumes 3.8x bandwidth compared to an idealized DRAM cache that does not consume any bandwidth for secondary operations. We also show that redesigning the DRAM cache to minimize the bandwidth consumed by secondary operations can potentially improve system performance by 22%. To that end, this paper proposes Bandwidth Efficient ARchitecture (BEAR) *for DRAM caches. BEAR integrates three components, one each for reducing the bandwidth consumed by miss detection, miss fill, and writeback probes. BEAR reduces the bandwidth consumption of DRAM cache by 32%, which reduces cache hit latency by 24% and increases overall system performance by 10%. BEAR, with negligible overhead, outperforms an idealized SRAM Tag-Store design that incurs an unacceptable overhead of 64 megabytes, as well as Sector Cache designs that incur an SRAM storage overhead of 6 megabytes.*

‡Aamer Jaleel contributed to this work while at Intel.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06$15.00
DOI:http://dx.doi.org/10.1145/2749469.2750387

1. Introduction

Recent advancements in on-chip packaging and 3D interconnect technology have enabled interconnecting several DRAM modules to offer significantly higher bandwidth than conventional DIMM-based DDR memories. Examples of such memory technology includes Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM), Wide I/O (WIO), and Graphic Double Data Rate (GDDR) memory [1, 2, 3, 4, 5]. These memory technologies offer high memory bandwidth, even though their access latency may be similar to existing DDR memories. Furthermore, modules from these technology may not have enough capacity to fully replace conventional DIMMs, so future memory systems are likely to consist of heterogeneous memory technologies. One attractive design to use these high bandwidth DRAM technologies is to architect them as a DRAM cache [6, 7, 8, 9, 10, 11], as an intermediate level between on-die caches and the DDR-based main memory.

Architecting gigascale DRAM as a hardware managed cache faces several challenges, including designing a tag storage of several tens of megabytes. For example, a 1GB cache contains 16 million lines (of 64 bytes each), which means that if each tag store entry requires four bytes, we would need 64MB of storage for the tag store. Provisioning such a large tag store on-chip is impractical. Architecting the DRAM cache as a Sector Cache [12, 10] reduce the SRAM overheads to 6MB, which is still quite large. Thus, SRAM based tag store suffers from both high storage overhead as well as tag access latency overheads, as shown in Figure 1(a).

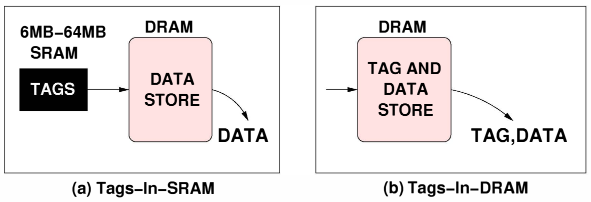

Figure 1: Organizations for 1GB DRAM cache (a) Storing tags in SRAM incurs overhead of 64MB (6MB with sector cache) (b) Tags-in-DRAM approach is more scalable and avoids the SRAM overheads.

The more practical approach for architecting large DRAM cache is to place tags in DRAM and uses intelligent placement of tags and data to reduce the lookup latency, as shown in Figure 1(b). For example, Loh and Hill propose (*Loh-Hill Cache*) [6] a design that organizes 29-way in a single DRAM row buffer to leverage row buffer hit. For each request, Loh-

Hill Cache transfers 4 cache lines (3 lines for tags and one for data). Another design, *Alloy Cache* [9] organizes tag and data together to form a Tag And Data (TAD) entry in a direct-mapped cache, and transfers only one TAD per request. Thus, Alloy Cache optimizes for hit latency and bandwidth instead of hit rate. As Alloy Cache is more bandwidth efficient compared to the Loh-Hill design, we use the Alloy Cache as our baseline to study the problem of bandwidth inefficiency in gigascale DRAM caches.

Stacked memory has 4x-8x higher bandwidth than DRAM memory. If the requested cache line is found in the DRAM cache, it can be serviced at a faster rate. For an idealized scenario, we can expect all of the DRAM cache bandwidth to get consumed only by transfers of data lines from the DRAM cache to the processor chip. However, a cache design also needs to perform several secondary operations that consume cache bandwidth. Unlike SRAM cache that is well customized in terms of accessing points (e.g., tag/data ports, read/write ports), any operations to DRAM cache shares the same DRAM interface. These secondary operations includes *Miss Probe* (to detect a miss, we need to look up the tag store in the DRAM cache), *Miss Fill* (on a cache miss the missed line is obtained from memory and filled in the cache), *Writeback Probe* (on a dirty eviction from the on-chip LLC identifying if that line is present in the DRAM cache), *Writeback Update* (if writeback probe gives a hit, updating the content of the line in DRAM cache), and *Writeback Fill* (filling the writeback data in the cache, if a writeback probe gives a miss). We would like the bandwidth consumed by such secondary operations to be negligibly small, and have almost all the bandwidth available dedicated to critical data transfers.

To measure the consumption of bandwidth due to secondary operations we define a metric called *Bloat Factor*, which is the ratio of the total bandwidth consumed by the DRAM cache to the bandwidth required for transferring only the data lines to the processor chip. For an idealized cache that does not spend any bandwidth for secondary operations, Bloat Factor equals 1. Our study with a 1GB DRAM cache shows that Loh-Hill Cache and Alloy Cache have a Bloat Factor of 7.3X and 3.8X, respectively. Thus, recent designs for DRAM cache do not use bandwidth efficiently.

For a DRAM cache, the cache hit latency comprises two parts: the DRAM array access latency and the queuing delay. When bandwidth is sufficient, the queuing delay is negligible compared to DRAM array latency. However, when bandwidth is a scarce resource, queuing delay becomes significant in the cache hit latency. We show that 2.8X bandwidth bloat more than doubles the DRAM cache hit latency, due to an increase in the queuing delay. Thus, if we can redesign the cache to have a Bloat Factor close to 1, we can reduce cache hit latency and potentially improve performance by 22%. To this end, this paper makes following contributions.

1. To the best of our knowledge, this is the first paper that identifies and quantifies the bandwidth bloat in DRAM caches due to secondary operations. We show that secondary operations cause even the most bandwidth-efficient DRAM-cache design to consume 3.8x the bandwidth, thereby reducing the bandwidth available for the critical operations.
2. We propose *Bandwidth Efficient ARchitecture (BEAR)* for DRAM caches. BEAR consists of three component techniques, each aimed at making one of the secondary operations bandwidth efficient, namely:
   (a) *Bandwidth Efficient Cache Fills.* Inserting missed lines in the cache consumes cache bandwidth, but many of the inserted lines do not get reused while resident in the cache. Bypassing some of the Miss Fills can reduce bandwidth bloat, but naive bypassing can reduce the hit rate of the DRAM cache significantly for some workloads and degrade performance. We propose *Bandwidth Aware Bypass (BAB)* to reduce the bandwidth consumed by fill operations while limiting the loss in cache hit rate to a desired level.
   (b) *Bandwidth Efficient Writeback Probe* The second component scheme, *DRAM Cache Presence (DCP)*, reduces Writeback Probe by introducing state information in the on-chip Last Level Cache (LLC) to track if the line exists in the DRAM cache.[1] DCP associates each cache line in the LLC with one bit, which keeps track of the line's presence state information in the DRAM cache; when a dirty line is evicted from the LLC, this bit guides to issue a Writeback Probe.
   (c) *Bandwidth Efficient Miss Probe.* We reduce the bandwidth consumed by Miss Probe by leveraging the property of DRAM caches to streams multiple tags on each access. We buffer the tags of recently accessed adjacent cache line's tags in the *Neighboring Tag Cache (NTC).* On a LLC miss, the request first looks up the NTC. If the tag for the requested cache location is found in the NTC, it avoids the Miss Probe.

Overall, the three component schemes of BEAR can be implemented with a storage overhead of only 4KB (and one bit per line in the L3 cache). BEAR can be implemented without any changes to the architecture of the DRAM array. BEAR reduces the bandwidth consumption of DRAM cache by 32%, which reduces the cache hit latency by 24%. Even though BEAR degrades the cache hit rate by 2% (due to BAB), it increases overall system performance by 10.1%.

While we focus on tags-in-DRAM designs in our studies, we show that BEAR outperforms an idealized design that stores the tags on-chip using 64MB SRAM, as well as the sector cache design that has an 6MB SRAM overhead.

[1]If the DRAM cache is designed to be inclusive of Last Level Cache (LLC) then a Writeback Probe is not necessary. Unfortunately, inclusive DRAM caches do not support bypassing, as all the lines in the LLC must be present in the DRAM cache and bypassing breaks this requirement. Ideally, we want to avoid the bandwidth of Writeback Probe, while still being able to do bypassing. Our solution can obtain both benefits, while inclusive DRAM caches cannot. We compare our proposal to inclusive cache in Section 7.

2. Background and Motivation

Processor architects face a key design decision on how to utilize high bandwidth memory in a heterogeneous memory system. Using high bandwidth memory as a cache, which we refer to as *DRAM cache*, is attractive as it is transparent to the software, and allows hardware vendors to use stacked memory without relying on support from the software companies. In this paper, we focus on efficient management of such hardware-managed gigascale DRAM caches.

2.1. Recent Designs for DRAM Cache

Architecting high performance DRAM cache have received significant attention over the past few years. These proposals have mostly focused on managing tag storage overhead, cache hit rate, and/or cache access latency. We show two recent designs for DRAM cache in Figure 2.

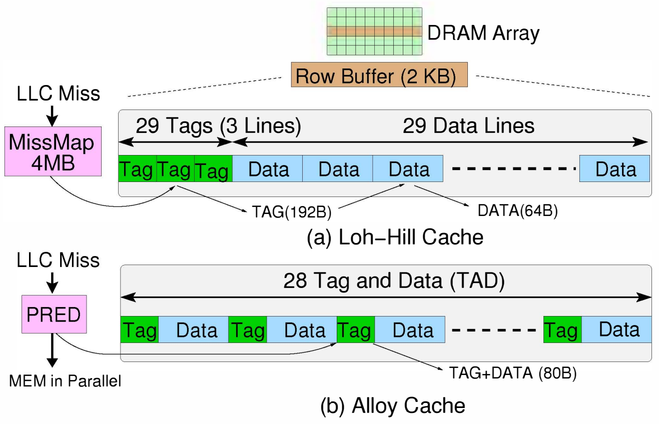

Figure 2: Recent Designs for DRAM Cache: (a) Loh-Hill Cache transfers 3 lines for tag and 1 line for data on each hit (256 bytes), and (b) Alloy Cache transfers 1.25 cache line for each hit.

The tag storage overhead problem arises because gigascale DRAM caches hold millions of cache lines. For example, a 1GB DRAM cache holds 16 million 64B cache lines which requires 64MB of tag storage overhead (assuming four bytes per tag). Since it is impractical to accommodate such a large tag structure on chip, most proposals suggest storing the tag and data in the high bandwidth memory itself.

For example, Loh and Hill propose a set-associative DRAM cache by implementing a 29-way cache stored in a single DRAM row buffer, as shown in Figure 2(a) [6]. This proposal stores the tags for the 29 cache ways in the first three cache lines of the 2KB row buffer. Servicing a cache hit requires first reading (and checking) the tags from the DRAM row buffer and then reading the data from the DRAM row buffer (on a tag match). This proposal targets improving cache hit rate using higher associativity at the expense of increased access latency and higher read bandwidth.

More recently, Qureshi and Loh tackled the latency and bandwidth problem with the direct-mapped *Alloy Cache* proposal, shown in Figure 2(b). Alloy Cache targets both cache access latency and bandwidth, organizing the tag and data together to form a Tag And Data (TAD) entry [9]. Servicing a cache hit requires a single read of the TAD entry from the DRAM row buffer and then checking for a tag match. If the tag matches, the associated data entry within the TAD is forwarded to the requesting core. Doing so, the Alloy Cache improves cache access latency and bandwidth at the expense of a slight reduction in hit-rate.

2.2. Bandwidth Bloat in DRAM Caches

The conventional approach of using DRAM to architect main memory follows a simple request response protocol. When an address misses in the on-chip Last-Level Cache (LLC), the memory controller fetches the data from the DRAM devices. With DRAM configured as main memory, the effective raw DRAM bandwidth is the total number of bytes transferred on the data bus (i.e., #LLC_misses * LLC_Line_Size).

Architecting DRAM as a cache, on the other hand, requires additional bandwidth to implement cache functionality (e.g., cache fills, cache probes). We propose a metric termed *Bloat Factor*, which is defined as the total bytes transferred on the DRAM cache data bus divided by the total bytes required to satisfy all LLC misses, as shown in Equation 1.

$$BloatFactor = \frac{\sum \text{Total Bytes Transferred}}{\sum \text{Useful Bytes Transferred}} \tag{1}$$

Ideally, the Bloat Factor value should be 1, meaning that the entire DRAM cache bandwidth contributes to servicing LLC misses. However, as Figure 3(a) illustrates, Bloat Factors are 7.3X and 3.8X for Loh-Hill and Alloy caches, respectively. DRAM cache hit latency comprises two parts: DRAM array access latency and queuing delay. Bandwidth bloat increases DRAM service time due to increasing queuing delay. Shown in Figure 3(b), DRAM cache hit latency is 409 cycles and 239 cycles with respect to Loh-Hill and Alloy cache, while an ideal case (termed *Bandwidth-Optimized cache (BW-Opt)*) that all secondary operations are free has DRAM cache hit latency of only 97 cycles. BW-Opt reduces L4 hit latency significantly, and thus BW-Opt outperforms both Loh-Hill and Alloy cache, as shown in Figure 3(c). Detailed experimental methodology is in Section 3.

In the rest of the paper, we use Alloy cache as our baseline DRAM cache model (comparison to LH-cache in Section 7). Although Alloy cache is more efficient, it still has room to improve (3.8X in Bloat Factor, and 22% in performance).

2.3. Breakdown: Where Does the Bandwidth Go?

Bandwidth bloat in DRAM caches corresponds to the steps in implementing cache functionality. Unlike memory which only holds data, DRAM caches hold both tag and data. Typically, on read requests, a tag is used to determine if an address exists in the cache. Thus, every cache lookup requires both tag and data to be fetched from the DRAM cache. If the cache lookup

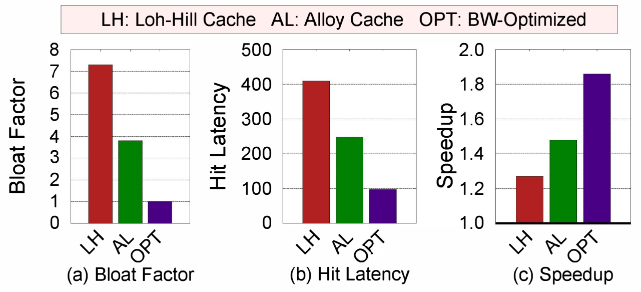

Figure 3: Comparison of Loh-Hill (LH), Alloy (AL), and Bandwidth-Optimized (OPT) cache: (a) Bloat Factor, (b) Hit Latency, and (c) Speedup with respect to no DRAM cache.

results in a hit, the first source of bandwidth bloat (referred to as *Hit Probe*) can be attributed to tag fetch (the data is critical and hence not a bandwidth bloat). If the cache lookup results in a miss, the second source of bandwidth bloat (referred to as *Miss Probe*) can be attributed to the fetching of both tag and data.[2] Typically, a cache miss requires inserting a line into the cache; thus, the third source of bandwidth bloat (referred to as *Miss Fill*) is to fill the new tag and data into the cache.[3]

In addition to read requests, the processor can return dirty data from the on-chip LLC by issuing writeback requests. On a writeback request, the DRAM cache must be consulted to determine whether the corresponding line already exists in the DRAM cache. Should the line exist in the DRAM cache, the DRAM cache contents must be updated for correctness. Thus, the fourth source of bandwidth bloat (referred to as *Writeback Probe*) can be attributed to fetching the tag to detect whether or not to update the DRAM cache contents. If the Writeback Probe results in a cache hit the new data and existing tag are written back to the DRAM cache.[4] Thus, the fifth source of bandwidth bloat (referred to as *Writeback Update*) can be attributed to re-writing the tag (not data). On the other hand, if the Writeback Probe results in a cache miss there are two possibilities. If a writeback no-allocate policy is used, the data is sent to main memory. However, if a writeback allocate policy is used, the new data and new tag are written to the DRAM cache replacing the existing data. Thus, the sixth source of bandwidth bloat (referred to as *Writeback Fill*) can be attributed to updating tag and data on writeback requests.

Figure 4 shows the bandwidth breakdown for the Alloy cache. In a BW-Opt cache, the Bloat Factor is 1, and all the bandwidth is dedicated to Hit: The cache performs all the secondary cache operations logically, without using any of the physical resources. On the other hand, Alloy Cache

[2]Note that bandwidth bloat is only due to Miss Probes that fetch clean lines. If a dirty line is fetched, the Miss Probe is necessary for correctness to write the dirty data to main memory. Most Miss Probes cause bandwidth bloat since the majority of DRAM cache lines tend to be clean.

[3]As Alloy Cache is direct-mapped, it does not require replacement updates on cache hits. For Loh-Hill cache, replacement update on cache hit is another source of bandwidth bloat (if LRU/DIP replacement is used).

[4]In the Alloy Cache, if a writeback allocate policy is used, Writeback Fills must be preceded with a Writeback Probe to determine if a dirty line is being evicted and a writeback to memory is necessary.

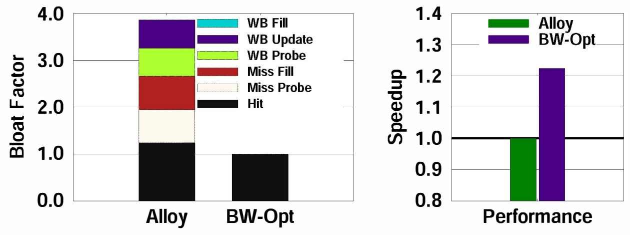

Figure 4: Comparison of Bloat Factor and Potential Performance.

requires five 128-bit bus transfers (80 bytes) to transfer the tag and data (72 bytes). This is a Bloat Factor of 1.25X for Hit compared to BW-Opt cache. Miss Probe and Miss Fill each take about 0.67X. Writeback Probe and Writeback Update each take about 0.57X. Note that we use write-allocate policy for DRAM cache, and hence do not have Writeback Fill in the baseline. Overall, the Bloat Factor for Alloy cache is 3.8X.

Note that the cache operations corresponding to bandwidth bloat are common to both SRAM and DRAM cache designs. However, these cache operations do not degrade performance in SRAM caches primarily because of SRAM cache implementation. Unlike DRAM caches that share a single narrow data bus for all read and write operations, SRAM caches typically consist of separate read and write ports that match the width of the corresponding tag and data. Furthermore, SRAM caches have much higher read/write bandwidth because of separate banked tag and data arrays each with their own read and write port. Therefore, the bandwidth utilized by these secondary operations has not been a critical concern for the on-chip SRAM caches. Unfortunately, for DRAM caches bandwidth is a scarce resources, so the performance overhead of these secondary operations becomes significant, and there is an opportunity to improve performance by reducing the number of cache operations that result in bandwidth bloat.

2.4. Goal: Bandwidth Efficient ARchitecture (BEAR)

DRAM cache bandwidth bloat is attributed to six different cache operations: *Hit Probe, Miss Probe, Miss Fill, Writeback Probe, Writeback Update, and Writeback Fill.* Among these operations, only the Hit Probe contributes towards useful bandwidth to service the LLC miss request. All other cache operations are either targeted for improving performance (Miss Fill, and Writeback Fill), or for ensuring correctness (Miss Probe and Writeback Probe).

Since bandwidth bloat increases DRAM cache access latency, we investigate opportunities to reduce bandwidth bloat. In this paper, we target three sources of following bandwidth bloat:

1. Bandwidth-Efficient Miss Fill. Miss Fill consumes significant DRAM cache bandwidth. A typical cache design inserts all cache lines on a miss, with the assumption that such lines will later provide cache hits. However, a significant percentage of lines are not referenced again [13, 14]. Consequently, we can use cache bypassing to reduce the

bandwidth consumed by Miss Fills, even if it degrades cache hit rate by a marginal amount.

2. Bandwidth-Efficient Writeback Probe. Typically, a Writeback Probe is issued before a Writeback Update to determine whether the line already exists in the DRAM cache. If the architecture provides guarantees on whether or not a line already exists in the DRAM cache, the majority of Writeback Probes can be eliminated. We propose enhancements to the on-chip LLC to avoid Writeback Probes.
3. Bandwidth-Efficient Miss Probe. Miss Probes waste bandwidth when the requesting line misses in the cache. We leverage DRAM cache design to buffer recently accessed neighboring tags to reduce the bandwidth of Miss Probes.

We discuss our experimental methodology before describing each of our solutions in detail.

3. Experimental Methodology

3.1. System Configuration

We use a x86 simulator with a detailed memory system model. Table 1 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 the off-chip DRAM cache). All cache hierarchy uses 64B line size. We use Alloy Cache as the baseline L4 cache, and the results are normalized to Alloy Cache unless stated otherwise. Cache misses fill all levels of the hierarchy. We equip the Alloy Cache with a the MAP-I miss predictor [9] to overcome the tag lookup latency for cache misses.

Our baseline assumes that L4 is non-inclusive of L3 cache. (L3 cache can be either inclusive or non-inclusive of L1/L2 caches, although we model non-inclusive L3 cache for simplicity). Also, we assume the DRAM cache is a memory-side cache, and hence we do not consider coherence traffic. Writeback misses do not allocate and instead send data to the next cache level. We model a virtual memory system to perform virtual to physical address translations.

We assume a heterogeneous memory system with the DRAM cache using HBM technology [2] and main memory using conventional DIMM technology [4]. In accordance with the specification for stacked memory, we assume the same access latency in both DRAM technologies. However, the bandwidth of DRAM cache is much higher than main memory. In our baseline system, DRAM cache has 8x bandwidth of main memory (2X channel, 2X bus width, 2X bus frequency). A DRAM cache bandwidth sensitivity study is presented in Section 8. We model DRAM timing based on USIMM[15]. For both the stacked DRAM and off-chip DRAM, we equip each memory channel with separate read queue and write queue, and the scheduler prioritizes read requests over write requests, and writes are issued in batches.

Table 1: Baseline System Configuration

Processors	
Number of Cores	8
Frequency	3.2GHz
Core Width	2 wide out-of-order
Last Level Cache	
Shared L3 Cache	8MB, 16-way, 24 cycles
DRAM Cache	
Capacity	1GB
Bus Frequency	1.6GHz (DDR 3.2GHz)
Channels	4
Banks	16 Banks per rank
Bus Width	128 bits per channel
tCAS-tRCD-tRP-tRAS	36-36-36-144 CPU cycles
Main Memory (Conventional DRAM)	
Capacity	16GB
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	2
Banks	8 Banks per rank
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	36-36-36-144 CPU cycles

3.2. Workloads

We use a representative region of 1-billion instructions from the SPEC CPU2006 benchmark suite, captured by SimPoints [16]. We present our study using workloads that have Miss Per Thousand Instruction (MPKI) greater than 1, as shown in Table 2. We group the workloads into two categories: High Intensive (MPKI greater than 12) and Medium Intensive (MPKI between 2 and 12). We evaluate our study by executing benchmarks in rate mode, where all eight cores execute the same benchmark, as shown below.

Table 2: Workload Characteristics for Rate Mode.

Category	Name	L3 MPKI	Footprint
High Intensive	mcf	74.6	10.2 GB
	lbm	32.7	3.1 GB
	soplex	27.1	1.9 GB
	milc	26.1	4.5 GB
	libquantum	25.5	256 MB
	omnetpp	21.1	1.1 GB
	bwaves	18.7	1.5 GB
	gcc	18.6	680 MB
Medium Intensive	sphinx3	12.4	136 MB
	GemsFDTD	9.9	5.3 GB
	leslie3d	7.6	616 MB
	wrf	6.8	488 MB
	cactusADM	5.5	1.2 GB
	zeusmp	4.8	1.5 GB
	bzip2	3.7	2.4 GB
	xalancbmk	2.3	1.3 GB

We also evaluate 38 mixed workloads that are selected from the above 16 benchmarks. Table 3 shows the workloads for which we will show detailed results. The virtual-to-physical page mapping ensures that two benchmarks do not map to the same address.

Table 3: Workload Characteristics for Mixed Workloads.

Name	Workloads	Class
MIX1	libq-mcf-soplex-milc-bwaves-lbm-omnetp-gcc	8H
MIX2	libq-mcf-soplex-milc-lbm-omnetpp-Gems-sphinx	6H+2M
MIX3	mcf-soplex-milc-bwave-gcc-lbm-leslie-cactus	6H+2M
MIX4	libq-mcf-soplex-milc-Gems-leslie-wrf-zeusmp	4H+4M
MIX5	bwave-lbm-omnetp-gcc-cactus-xalanc-bzip-sphinx	4H+4M
MIX6	libq-gcc-Gems-leslie-wrf-zeusmp-cactus-xalanc	2H+6M
MIX7	mcf-omnetp-Gems-leslie-wrf-xalanc-bzip-sphinx	2H+6M
MIX8	Gems-leslie-wrf-zeusmp-cactus-xalanc-bzip-sphinx	8M

3.3. Figure of Merit: Performance

For rate mode workloads, we use the total execution time as the performance metric. The reported normalized speedup is the normalized execution time with respect to the baseline system. For mixed workloads, we use weighted speedup as the performance metric, and the reported normalized speedup is the normalized weighted speedup with respect to the baseline system. The weighted speedup is given by Equation 2.

$$WeightedSpeedup = \frac{\sum_i IPC_i^{shared}}{\sum_i IPC_i^{single}} \qquad (2)$$

To report performance, we use geometric mean to report the average speedup for the 16 rate-mode runs (RATE), 8 mix-mode runs (MIX), and all 54 workloads (ALL).

3.4. Measuring Bandwidth Efficiency

Another important aspect of our study is the bandwidth consumption of the DRAM cache. We define *Bloat Factor* as the amount of total bytes transferred divided by the useful bytes transferred on the bus, as shown in Equation 1. The denominator also means the total cache lines transferred to the processor multiplied by cache line size (i.e., 64 bytes). Note that bandwidth efficiency is the inverse of the Bloat Factor.

4. Bandwidth-Efficient Miss Fill

Among secondary operations, our first target is Miss Fill. Miss Fill takes 17% DRAM cache bandwidth (Bloat Factor 0.67 of 3.8). If all inserted cache lines are accessed again, the future access will be served by DRAM cache, and have lower latency. However, not all inserted cache lines will be re-referenced again [13, 14], which gives us an opportunity to bypass some Miss Fills without impacting hit rate significantly. In this section, we first examine a naive approach which bypasses a fixed fraction of the cache fills randomly. We show that while such a scheme can improve hit latency of the cache, in some cases it can cause severe degradation in both the hit rate and overall system performance. We propose a *Bandwidth Aware Bypass (BAB)* scheme that tries to free up the bandwidth consumed by Miss Fills while limiting the degradation in hit rate to a predetermined amount.

4.1. Probabilistic Bypass: A Simple and Naive Scheme

A fairly simple and straight forward way to reduce the bandwidth consumed by Miss Fill is to not perform Miss Fill for a given percentage of cache misses. Let the *Bypass Probability* (*P*) denote the fraction of total cache misses for which we decide to skip the Miss Fill and instead bypass the cache. On a cache miss, we could make the decision of install of bypass by consulting a random number generator. If the value of the random number generator is less than P, perform bypass, otherwise fill the line in the cache. We call this scheme *Probabilistic Bypass (PB)*. The parameter P regulates the effectiveness of PB at reducing the bandwidth consumed by Miss Fills. At high value of P, we would expect a larger number of lines to be bypassed, which reduces the bandwidth consumption of Miss Fills, and therefore improves the cache hit latency. Unfortunately, bypassing a larger number of cache lines can have adverse impact on hit rate too, and thus harm overall system performance. To analyze this phenomenon, we study two values of bypass probability: P=50% and P=90%. Note, PB with P=0% is the same as the baseline design which does not perform bypass.

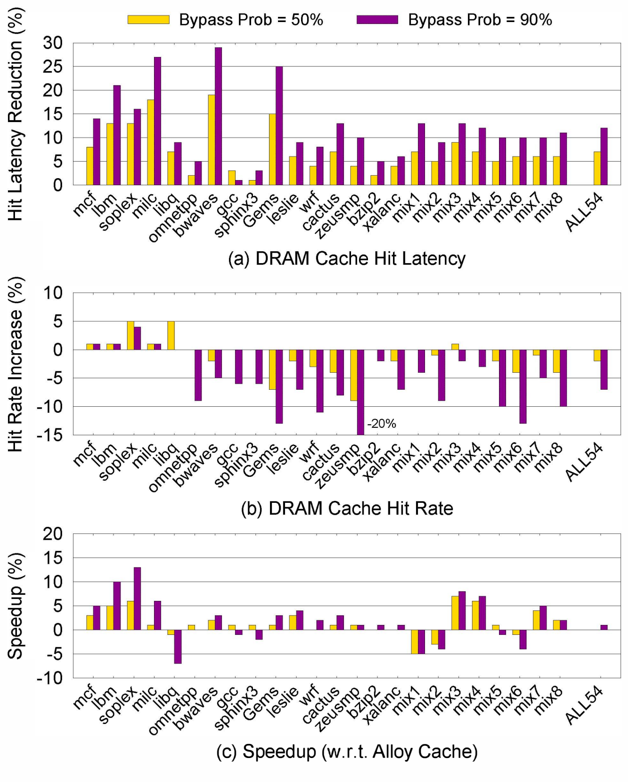

Figure 5: Comparison of Probabilistic Bypass with P=50% and P=90% in terms of impact on (a) Cache Hit Latency (b) Cache Hit Rate (c) Speedup. All numbers are with respect to the baseline.

Figure 5 shows the reduction in cache hit latency (higher is better), increase in cache hit rate (higher is better) and speedup (higher is better) with PB for P=50% and P=90%. As expected, aggressive bypassing can reduce hit latency significantly, on average by 12% for P=90%. Unfortunately, probabilistic bypassing can also degrade hit rate significantly for several workloads (such as Gems and Zeusmp), which can reduce performance. Overall, the speedup from probabilistic bypass is negligible, and we may deem PB to be ineffective at improving performance.

4.2. Bandwidth Aware Bypass: Limiting Hit-Rate Loss

Ideally, we desire the benefits from cache hit latency reduction using PB, without significantly impacting the DRAM cache hit rate. For the DRAM cache to be high performing, PB should not degrade cache hit rate significantly with respect to the baseline. This suggests a dynamic mechanism that measures the differential in hit rate (or miss rate) between the baseline and PB. If the differential is lower than some threshold then it should use PB, otherwise the baseline. We call this mechanism as *Bandwidth Aware Bypass (BAB)* as it tries to continue to bypass (in order to free up the bandwidth) even if such bypassing causes a minor degradation in cache hit rate. This is unlike prior schemes on cache replacement that aim to do bypassing solely with the aim of maximizing cache hit rate, and would try to disable the bypassing mechanism if there is any loss of hit rate.

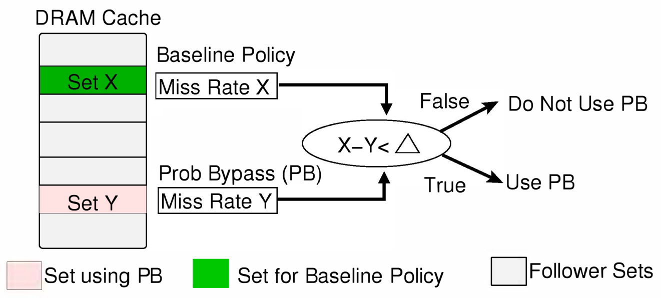

Figure 6: Design of Bandwidth Aware Bypass.

We use Set Dueling [13] to dynamically select between PB and the baseline. Of the 16M sets in the DRAM cache, we create two sampling monitors of 512K sets each for PB and baseline policy, and the remaining 15M sets are the follower sets. We use two 16-bit counters for each sampling monitor: one counts misses, and the other counts accesses. Misses and accesses to the sampled sets increment the corresponding 16-bit counters. When any of the access counters saturates, all the counters are shifted right by 1 bit. We compute the miss rates of the baseline and PB sampled sets and then compare the difference in the two miss rates to a threshold, Δ. If the difference is smaller than the threshold, then PB and baseline have similar miss rates Therefore, we can be bandwidth efficient and set the mode-bit to enable the follower sets to use PB. Whereas, if the difference is greater than or equal to the threshold, then baseline has better miss rate and we unset the mode-bit to enable the follower sets to use the baseline policy. Note that there is a single mode-bit for the entire cache and it changes only when one of the access-bit counter saturates.

We conduct a sensitivity study using 90% probability to determine the best threshold for the differential in miss rate for the mechanism to select between PB and the baseline, and found that using $\Delta = \frac{1}{16}$ gave the best overall performance. Which means that PB must provide a hit rate of at least 15/16th of the baseline hit rate for the bypassing to continue, otherwise PB get disabled.

4.3. Effectiveness of BAB

Figure 7 shows the speedup of BAB, in which the component PB policy uses a bypass probability of 90%. On average, BAB improves performance by 5.1% (and as much as 15%) over the baseline, without causing degradation in any of the workloads. The cache hit rate with and without BAB are 61% and 63%, respectively. Thus, BAB sacrifices a small amount of cache hit rate to free the Miss Fill bandwidth, which reduces hit latency and improves performance.

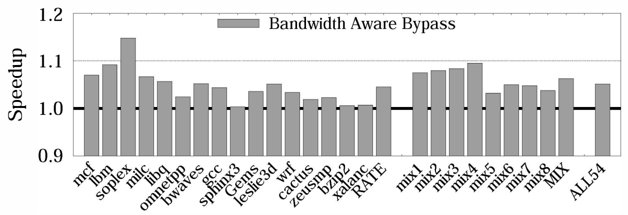

Figure 7: Speedup from Bandwidth Aware Bypass.

5. Bandwidth-Efficient Writeback Probe

Cache writebacks update main memory with new data modified by the processor core. On a writeback request, the DRAM cache must be consulted using a WriteBack Probe to determine whether the line already exists in the DRAM cache. The probe is necessary for correctness to update the DRAM cache data. Updates enable the DRAM cache to service future requests with the most recent data value.

In general, a Writeback Probe is wasteful if the line evicted from the on-chip LLC (i.e., dirty line) already exists in the DRAM cache. Since DRAM caches are generally much larger than on-chip LLCs, the probability that a writeback request misses in the DRAM cache tends to be very low ($<$ 1% in our studies). This suggests that the majority of Writeback Probes are useless and cause unnecesary bandwidth bloat. Hence, it is highly desirable if the cache architecture can provide some guarantees on whether (or not) a dirty line evicted from the on-chip LLC exists in the DRAM cache.

5.1. Limitation of Inclusive Caches

One approach is to enforce the inclusion property [17] for the DRAM cache. Enforcing inclusion mandates that all lines resident in the small on-chip caches must also be resident in the DRAM cache. When evicting lines from the DRAM cache, inclusion is enforced by sending a back-invalidate request to also evict the line from all on-chip caches (should the line be present). Consequently, enforcing inclusion property for

DRAM caches eliminates the need for Writeback Probes since writebacks are guaranteed to hit in the DRAM cache.

While an inclusive DRAM cache eliminates the bandwidth bloat due to Writeback Probes, inclusion prevents bypassing of cache lines on misses. Consequently, inclusion eliminates the 5-15% performance benefits from our bandwidth conscious Adaptive Fill policy. Ideally, we want to eliminate both Miss Fill and Writeback Probe, but inclusive cache can avoid only Writeback Probe. Therefore, we desire a mechanism that not only reduces Writeback Probes, but also is able to bypass Miss Fills.[5] We show that our proposed design outperforms inclusive cache in Section 7.

5.2. Tracking Residency of Line in DRAM Cache

Writeback Probes can be avoided if there exists some state information in the memory hierarchy that specifies which cache lines are resident in the DRAM cache. Note that this state information need not be for every line in the DRAM cache, but only those lines that are dirty in the on-chip caches. Thus, the state information can be reduced from tracking millions of lines in the DRAM cache to only a few thousand lines present in the on-chip caches.

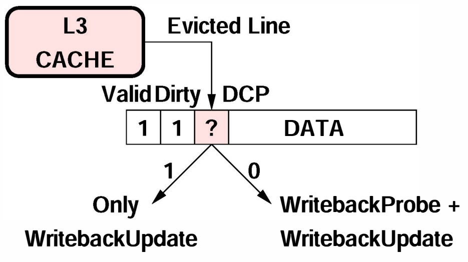

Figure 8: Design of DRAM Cache Presence Bit.

The state information is a one-bit field that tracks whether or not a line is present in the DRAM cache. We refer to this one-bit field as *DRAM Cache Presence (DCP)* and propose adding DCP to each line in the LLC, as in Figure 8. DCP is modified on LLC fills and DRAM cache evictions. On LLC fills, DCP is set to one if the line was serviced from the DRAM cache, zero otherwise. DCP is kept up-to-date on DRAM cache evictions. When a line is evicted from the DRAM cache, the LLC is conveyed this information (similar to the flow of an inclusive DRAM cache). If the line is present in the LLC, DCP is updated to zero (instead of invalidating the line as in inclusive cache). Consequently, the LLC knows that the line is no longer present in the DRAM cache.

DCP enables writeback requests to have full knowledge on whether or not the line is present in the DRAM cache. On writeback requests from the LLC, if the DCP value is one, bandwidth bloat due to Writeback Probes can be avoided altogethor and only a Writeback Update is necessary to update the line in the DRAM cache.

A DCP value of zero implies a writeback miss since the dirty line is no longer present in the DRAM cache. With the writeback-miss allocate policy, to ensure correctness, a Writeback Probe is required before performing a Writeback Fill. This is to determine whether the writeback allocate is replacing a dirty line from the DRAM cache.

[5]Inclusion is commonly used to simplify cache coherence. Since DRAM caches are usually implemented as memory side caches, they do not participate in coherence. Hence, relaxing/enforcing inclusion guarantees for DRAM caches have no impact on cache coherence.

5.3. Effectiveness of DRAM Cache Presence

Figure 9 illustrates the performance improvement of DCP in the presence of BAB. We observe that DCP improves performance by an additional 4%, with maximum of 12.8% in *omnetpp*, and 11.3% in *gcc*, both of which have very high hit rate for writebacks.

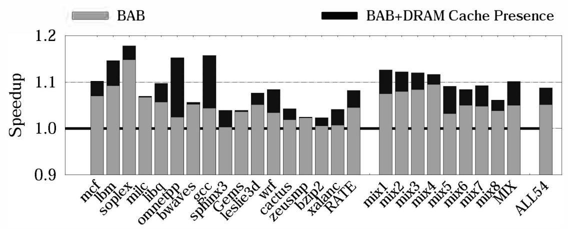

Figure 9: Performance for DRAM Cache Presence (DCP) over the baseline system that implements Bandwidth-Aware Bypass.

6. Bandwidth-Efficient Miss Probe

DRAM cache lookups can either result in a cache hit or a cache miss. Cache hits result in useful bandwidth whereas cache misses unnecessarily waste bandwidth by needless fetching clean data that is not utilized. We refer to the bandwidth bloat due to cache misses as Miss Probe. If the DRAM cache architecture can provide some guarantees on whether (or not) a line is present in the DRAM cache, Miss Probe bandwidth bloat can be minimized. We observe that current DRAM cache designs, including both Loh-Hill, and Alloy cache, enables us to develop simple mechanisms to provide such guarantees; these designs locate tag and data together in the same DRAM row buffer, and hence accessing one cache line also reads tags of other adjacent lines, making additional information available. We use Alloy cache as an example, but the idea can also be easily extended to Loh-Hill cache.

6.1. Neighboring Tag Cache

The Alloy Cache organizes the tag and data together to form a single Tag and Data (TAD) entry. Each TAD entry is 72 bytes long (8 bytes for tag and 64 bytes for the data). Alloy Cache organizes consecutive cache sets into the same row buffer as illustrated in Figure 10. With a 128-bit (16-byte) DRAM data bus, a cache lookup transfers a TAD entry in five bursts (total of 80 bytes are transferred). In doing so, any cache lookup also transfers the neighboring tag of the line present in the next cache set. This spatial locality can be exploited by storing the neighboring tag in a small fully associative structure called the *Neighboring Tag Cache (NTC)*. Each NTC entry contains two fields: tag and DRAM cache set index.

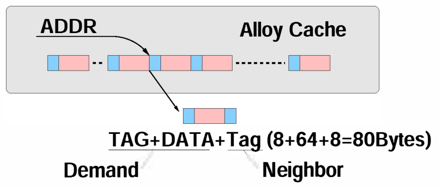

Figure 10: Alloy Cache brings in two tag entries with each access by default (due to bus being 16 bytes and tag being 8 bytes).

A miss in the LLC first consults the NTC by performing a set index match and tag match. If there is no set index match, the NTC can not provide any guarantees on the existence of the line in the DRAM cache. Therefore, a Miss Probe must be issued to to determine DRAM cache hit or miss. If there is a set index match and a tag match, the NTC guarantees that the request is present in the DRAM cache. Finally, if there is a set index match but a tag mismatch, the NTC gurantees that the request is not present in the DRAM cache. In this situation, the NTC can reduce the bandwith bloat due to Miss Probes. However, note that if the tag suggests that the DRAM cache entry is dirty, a Miss Probe is still necessary for correctness to read the dirty line and write it back to main memory.

6.2. Effectiveness of NTC

We assume an 8-entry NTC for every DRAM bank. For a DRAM cache with four channels and 16 banks per channel, the overall NTC size is 512 entries. However, note that only the eight NTC entries that correspond to the DRAM cache bank are accessed on an LLC miss. Finally, we assume single-cycle acces to the NTC and also ensure that the NTC is kept up-to-date on DRAM cache evictions.

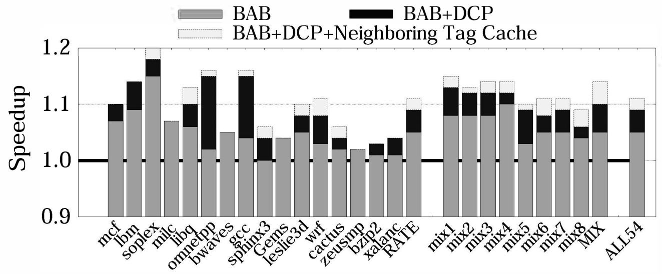

Figure 11: Performance for Neighboring Tag Cache for a baseline with BAB and DCP.

Figure 11 shows that NTC improves performance of the Alloy Cache by an additional 2%. Detailed analysis reveals that the NTC provides performance benefits by reducing two sources of wasteful bandwidth. First, by design NTC reduces bloat bandwidth due to Miss Probes.[6] Second, as a side benefit, the DRAM cache miss predictor takes advantage of the NTC to verify parallel memory access predictions. If a given entry is present in the NTC, the DRAM cache miss predictor squashes the wasteful parallel access to main memory.

7. Tying-it-All-Together: BEAR

BEAR consists of three schemes to reduce bandwidth bloat in the DRAM cache. This section compares the performance of BEAR to an idealized cache, shows the impact of BEAR on Bloat Factor, and asseses the storage overheads of BEAR.

7.1. Overall Performance Results

Figure 12 shows the performance of between the baseline Alloy Cache, BEAR DRAM, and an ideal Bandwith Optimized (BW-Optimized) DRAM cache. Note that RATE and MIX are referred to as the geometric mean of rate and mix workloads, while ALL54 is the geometric mean of all of our 54 workloads. On average, BEAR improves performance over the Alloy Cache by 10.1%. BEAR outperforms the BW-Optimized DRAM cache in some workloads: *soplex*, *milc*, and *libq*. This is because Adaptive Fill increases the hit rate for these benchmarks, which reduces overall memory latency and hence provides better performance than BW-Optimized cache. This is not the typical case, however, as Adaptive Fill causes a hit rate degradation of 2%, on average.

Table 4: Comparison of DRAM Cache Hit-Rate and Latency.

Design	Hit Rate	Latency (cycles)		
		Hit	Miss	AVG
Alloy	63.2%	239	391	326
BEAR	**61.0%**	**182**	356	282

Table 4 shows the hit rate and latency of DRAM cache. On average, BEAR is able to reduce DRAM cache hit latency from 239 to 182 cycles (24% improvement), while only sacrificing 2% hit rate. Also, the miss latency reduces, because of Neighboring Tag Cache's side effect that reduces unnecessary parallel access to off-chip memory.

7.2. Impact on Bloat Factor

Figure 13 illustrates the effectiveness of our proposals in reducing bandwidth bloat. We illustrate a bandwidth breakdown for every bandwidth factor, including Hit, Miss Probe, Miss Fill, Writeback Probe, Writeback Update, and Writeback Fill normalized to the Bloat Factor of a BW-Optimized DRAM cache. The BW-Optimized case only consumes Hit bandwidth, and transfers 64 bytes for every request. For other configurations, the basic unit of data transfer is 80 bytes. In the baseline Alloy cache, the Bloat Factor on average is 3.8, in which only 1.25 is critical to service the LLC miss requests.

[6]NTC does not consume any extra bandwidth for prefetching the neighboring tag. The extra tag is anyways fetched even in the baseline design because the width of the DRAM bus (16 bytes) is greater than the tag (8B).

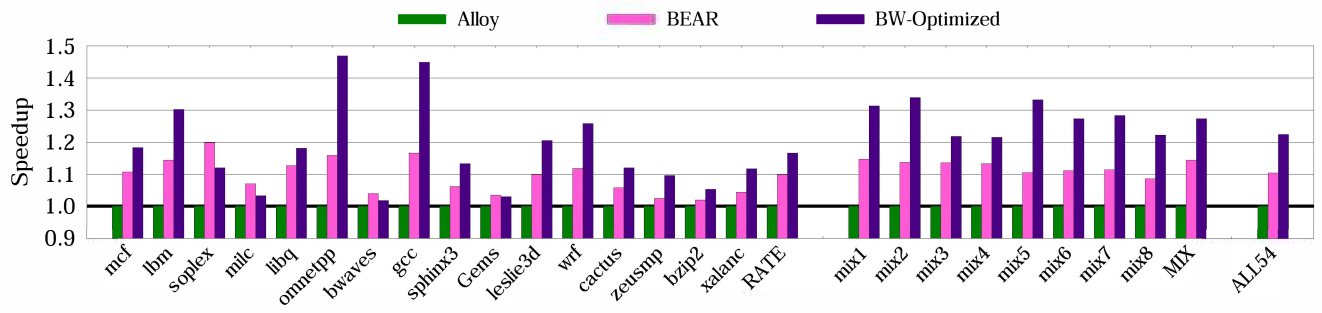

Figure 12: **Performance Improvement for Alloy, BEAR, and ideal case. Note that RATE and MIX are for 16 rate mode workloads, and 8 mixed workloads, respectively; ALL54 means the geometric mean across all 54 workloads.**

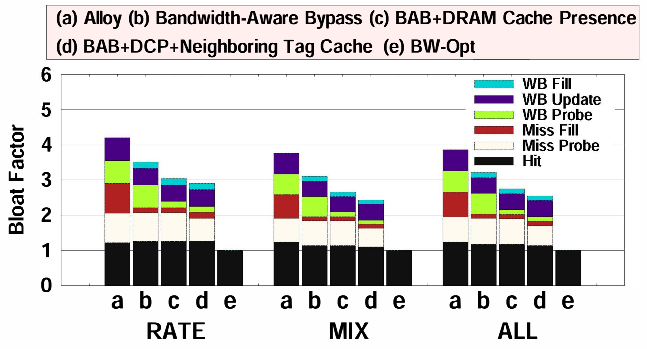

Figure 13: **Bloat Factor for Different Schemes.**

To review, Adaptive Fill targets reduction in Miss Fills, DRAM Cache Presence (DCP) targets reduction in Write Probes, and Neighboring Tag Cache (NTC) targets reduction in Miss Probe. Overall, BEAR is able to reduce the Bloat Factor by 32%.

7.3. Sensitivity to DRAM Cache Bandwidth and Capacity

We have assumed that the bandwidth of DRAM cache is 8X of the off-chip DRAM. We conduct a sensitivity study by studying DRAM caches with 4X, 8X and 16X bandiwdth (by varying the number of channels) while keeping the cache size constant. Figure 14(a) shows the performance improvement when the DRAM cache bandwidth varies. BEAR continues to provide performance improvements of more than 10% for all the bandwidth configurations.

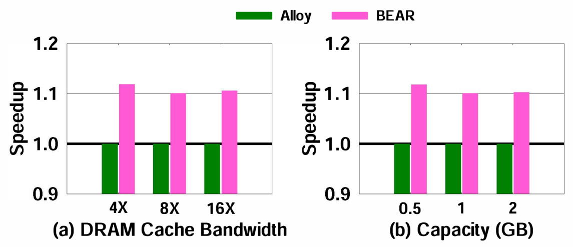

Figure 14: **Sensitivity to DRAM Cache: (a) Bandwidth (b) Capacity. Note that all numbers are normalized to Alloy cache with respect to each configuration.**

We also conduct a sensitivity study by varying the size of the DRAM cache while keeping the bandwidth constant. Figure 14(b) shows the performance improvement when the DRAM cache size changes from 512 MB to 2 GB. BEAR consistently improves performance by more than 10% across all DRAM cache capacity.

7.4. Sensitivity to DRAM Banks

We also conduct a study varying the number of banks to understand the delay contributed by bank conflict, and bus contention. Figure 15 shows the performance of BEAR, when the number of banks increases from 64 to 2048. Note the speedup is normalize to the baseline, with respect to each configuration.

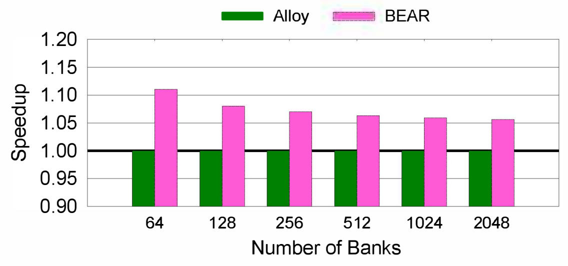

Figure 15: **Sensitivity to DRAM Banks. Note that all numbers are normalized to Alloy cache with respect to each configuration.**

BEAR consistently outperforms Alloy from 11% at 64 banks, to 6% at 2048 banks. The speedup remains constant at 6%, when the number of banks is 512, or more. As the number of banks increases, the row buffer conflict reduces. This suggests that the performance improvement provided by BEAR comes from two parts. The first is contributed by the reduction of bank conflicts, which is the speedup difference between 64 banks, and 2048 banks, or 5%. Second, the speedup saturation indicates that the remaining 6% performance improvement results from the reduction of bus contention.

7.5. Comparison to Alternative DRAM Cache Designs

We also compare our proposal to various implementation of DRAM cache, including Loh-Hill cache (LH-cache) [6], Mostly-Clean cache (MC-Cache) [8], and Inclusive Alloy cache (Incl-Alloy) [9], using default parameters (8X bandwidth, 1GB capacity). For LH-cache, we assume the MissMap has the same latency of LLC, which is 24 cycles in our study. MC-cache is an extension of LH-cache, which tries to reduce the Miss Probe bandwidth, and deploys memory requests to

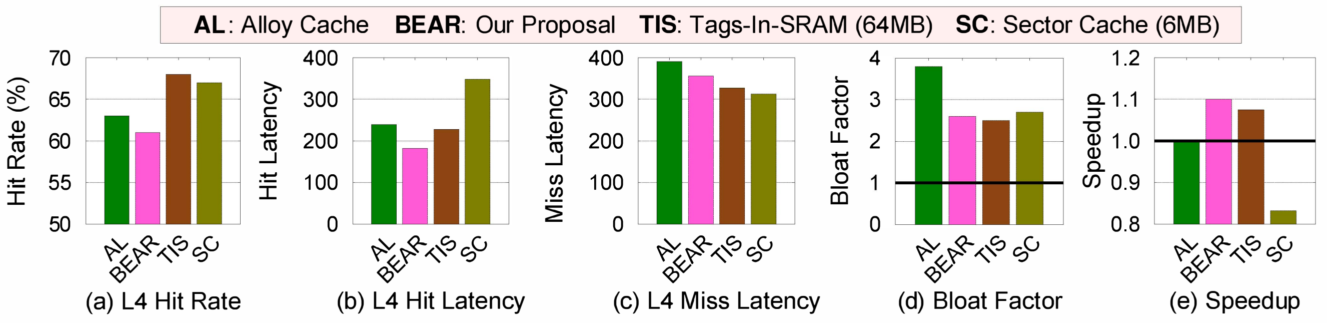

Figure 16: Comparison to Tags-In-SRAM (TIS) Cache and Sector Cache (SC): (a) L4 Hit Rate, (b) L4 Hit Latency, (c) L4 Miss Latency, (d) Bloat Factor, and (e) Speedup (w.r.t. Alloy). Note that TIS requires 64MB SRAM storage and SC requires 6MB SRAM storage.

off-chipe memory. For MC-cache, we assume a perfect predictor for hits and misses, and if the outcome of the predictor is a miss, the request will be serviced by the off-chip memory. For Incl-Alloy cache, we apply the inclusive property to DRAM cache with respect to the on-chip LLC.

LH-cache has 27% performance improvement across all the workloads, while MC-cache has 30%. LH-cache is a implementation that uses a MissMap structure to avoid Miss Probe bandwidth to DRAM cache as a trade-off an additional latency of 24 cycles for all the requests. MC-cache uses a predictor and removes the additional latency. However, both of them do not reduce Miss Fills or Writeback Probes.

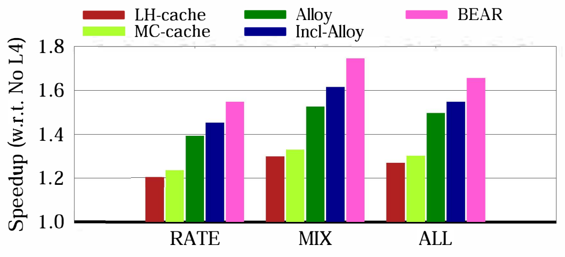

Figure 17: Speedup from different implementations of DRAM cache, normalized to a system without DRAM caches.

Inclusive Alloy cache is also a valid design. Incl-Alloy cache improves performance by 55% on average, which is 9% more than baseline non-inclusive Alloy Cache, while our proposal provides 66% performance improvement. Inclusiveness can avoid the Writeback Probe bandwidth, because it enforces every cache line in the LLC must be in the DRAM cache. However, inclusive DRAM cache loses the opportunity for other bandwidth optimization, such as Miss Fill, whereas, our proposal reduces both bandwidth bloat.

7.6. Storage Overhead of BEAR

Table 5 show the hardware overhead of each proposal. Overall, BEAR incurs negligible hardware overhead of 19.2 KB, the majority of which is due to the DCP-bit in the LLC.

8. Analysis of Tags-In-SRAM Designs

Thus far, we have only analyzed the Tags-in-DRAM designs, as such designs are scalable to large cache sizes. We now make a comparison of alternative designs that store Tags in SRAM. An unconstrained version of such a design, which we call *Tags In SRAM (TIS)* stores all the tags on-chip in an SRAM structure, and would incur a prohibitive storage of 64MB (at four byte of tag storage per line). The SRAM storage can be reduced to 6MB by architecting the cache as a Sector Cache (SC), as has been considered in recent designs such as the *Footprint Cache* [12, 10]. The advantage of these SRAM based designs is that they can support high set associativity and avoid some probe operations (e.g. Miss Probe and Writeback Probe). Unfortunately, these advantages come at a high storage overhead and also high latency overheads of accessing the tag store, before accessing the data store.

Table 5: Storage Overhead of BEAR

Design	Cost
Bandwidth-Aware Bypass	8 bytes per thread, total 64 bytes
DRAM Cache Presence	One bit per line in LLC, total 16K bytes
Neighboring Tag Cache	44 bytes per bank, total 3.2K bytes
Total	19.2K bytes.

We compare TIS and SC with BEAR. For both TIS and SC we provision the appropriate SRAM structure for tag store without penalizing either design for the added storage or the extra latency of tag access. Both caches are architected 32-way set associative. SC uses 4KB as an sector, which has 64 blocks (64B). We show statistics related to the DRAM cache (L4), including L4 hit rate, L4 hit latency, L4 miss latency, and Bloat Factor as well as the speedup in Figure 16.

BEAR outperforms tags-in-SRAM cache designs for the following reasons. (1) **Hit Rate:** For a gigascale DRAM cache, set-associativity contributes only to a limited improvement in hit rate (from 63% to 68%, consistent with prior studies [9].) (2) **Bloat Factor:** Both TIS and SC still incur bandwidth bloat from Miss Fill, Writeback Update, and Dirty Evictions. BEAR has very similar Bloat Factor as TIS and SC, because the amount BEAR saves is close to the amount of Miss Probe TIS and SC save. One can use the principles of BEAR to reduce the bandwidth Bloat of TIS and SC also. (c) **Latency:** We found latency is the decisive reason for the performance difference. Although SRAM caches do not need to look up tags in DRAM to detect cache misses, they do incur the penalty of dirty replacement, which gets exacerbated in SC as an evicted page can have a large number of dirty lines.

Overall, BEAR has 10.1% performance improvement, which exceeds the 7.5% speedup with TIS and 18% slowdown with SC. BEAR requires an SRAM overhead of only 20KB, whereas TIS and SC incur respective 64MB and 6MB SRAM storage overhead, which may be prohibitive.

9. Other Related Work

9.1. DRAM Cache

An extension of Loh-Hill DRAM cache is Mostly-Clean DRAM cache, which uses a miss predictor to save the MissMap storage overhead, and use parallel access to avoid serialization penalty. However, their proposal did not consider other bandwidth bloat factors [8].

Footprint cache is also another DRAM cache design, which is a sector cache design with prefetcher [10]. It uses SRAM storage to store the tag arrays; however, this is not a scalable solution, since the tag storage of 1GB DRAM cache can be as large as 6 megabytes. Also, enabling prefetch requests might exacerbate the bandwidth bloat problem in DRAM cache due to the extra bandwidth consumed by inaccurate prefetches.

9.2. Adaptive Fill

There are several studies that have focused on cache line install or bypass policy, which is related to Adaptive Fill. Most of the work are trying to identify cache blocks that are never reused after being installed in the cache[18, 19, 20, 21]. These cache lines are also referred to as *Dead Blocks.* One example is to sample dead block prediction, which was proposed to identify dead blocks in the last-level cache, and the fill process is skipped, if predicted dead[20]. However, most of the work requires a status update, leading to an additional access in the case of DRAM cache. Also, the goal to identify dead blocks is to improve hit rate of the cache, not improve DRAM cache bandwidth efficiency.

9.3. Cache Optimization

Cache inclusiveness is referred to as the property of the bigger cache with respect to smaller cache. Inclusive or exclusive cache are valid design choices that could be adopted by different chip vendors [22, 23, 17]. Recent study shows non-inclusive cache has better performance improvement than inclusive cache, because it avoids the *Back-Invalidate* operation, and provide higher hit rate than the inclusive cache. However, in the case of DRAM cache, the capacity is huge, but inclusive cache limits the scope of further optimization.

Other cache optimization, including replacement policy [13, 14, 24, 25, 26, 27], and write-allocation policy [28] , has been an active research area for the past decade. These optimization improves cache hit rate and therefore improve off-chip DRAM bandwidth. In contrast, our proposal aims at reducing DRAM cache bandwidth to improve performance.

9.4. Tag Cache

Separate structure to keep recently used tag has been proposed to avoid the tag look-up latency[29], when the last level cache is off-chip, and the latency for off-chip access is very high. Neighboring Tag Cache is storing the tag that has not been referenced (i.e., adjacent cache lines), but is highly likely to be referenced in the future. Unlike prior Tag Cache proposals which exploit temporal locality, Neighboring Tag Cache only exploits the spatial locality of the neighboring lines. However, these two schemes are orthogonal, and can be adopted simultaneously.

10. Summary

This paper aims at making DRAM caches bandwidth-efficient. We identify and classify DRAM cache operations into six categories: Hit Probe, Miss Probe, Miss Fill, Writeback Probe, and Writeback update, and Writeback Fill. Among those operations, only Hit Probe contributes to satisfy the miss request from L3 cache. Secondary bandwidth factor are either for performance or for correctness.

We define a metric termed *Bloat Factor* to understand how these secondary operations use DRAM cache bandwidth. We found only 33% of the DRAM cache bandwidth is used to satisfy L3 miss requests. Other secondary operations increase the queuing delay and thus DRAM cache hit latency. If bandwidth bloats are eliminated, hits can be serviced quickly.

We propose Bandwidth-Efficient ARchitecture (BEAR) DRAM cache to mitigate the bandwidth bottleneck in DRAM cache. BEAR has three different schemes, each of which targets its own bandwidth component: Bandwidth Aware Bypass for Miss Fill, DRAM Cache Presence bit for Writeback Probe, and Neighboring Tag Cache for Miss Probe.

Overall, the three component schemes of BEAR can be implemented with a storage overhead of only 4KB (and one bit per line in the L3 cache). BEAR can be implemented without any changes to the architecture of the DRAM array. Our evaluations show that BEAR reduces the bandwidth consumption of DRAM cache by 32% and improves system performance by 10.1%. BEAR achieves half the performance possible from an idealized bandwidth-optimized design that consumes no bandwidth for any of the secondary operations.

While we focus on tags-in-DRAM designs in our studies, we show that Bandwidth Bloat is a problem for Tags-in-SRAM designs too. BEAR outperforms an idealized design that stores the tags on-chip using 64MB SRAM, as well as the sector cache design that incurs an SRAM overhead of 6MB.

Acknowledgements

We thank Brian Railing, Sriseshan Srikanth, and anonymous reviewers for their comments and feedbacks. This work was supported in part by NSF grant 1319587 and the Center for Future Architecture Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] *HMC Specification 1.0*, 2013. [Online]. Available: http://www.hybridmemorycube.org

[2] JEDEC, *High Bandwidth Memory (HBM) DRAM (JESD235)*, JEDEC, 2013.

[3] Micron, *HMC Gen2*, Micron, 2013.

[4] *1Gb_DDR3_SDRAM.pdf - Rev. I 02/10 EN*, Micron, 2010.

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

[6] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual International Symposium on Microarchitecture*, 2011.

[7] G. H. Loh, N. Jayasena, J. Chung, S. K. Reinhardt, J. 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*, 2012.

[8] 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 *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*, 2012.

[9] 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 *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*, 2012.

[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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[11] C.-C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[12] J. B. Rothman and A. J. Smith, "Sector cache design and performance," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000.

[13] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[14] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[15] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, and M. Awasthi, *USIMM*, University of Utah, 2012.

[16] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[17] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies," in *Proceedings of the 2010 43rd Annual International Symposium on Microarchitecture*, 2010.

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

[19] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction &amp; dead-block correlating prefetchers," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[20] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual International Symposium on Microarchitecture*, 2010.

[21] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[22] *AMD Phenom II.* [Online]. Available: http://www.amd.com/us/products/desktop/processors/phenom-ii

[23] "Intel core i7-3940xm processor specification." [Online]. Available: http://ark.intel.com/products/71096/

[24] 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 *Proceedings of the 44th Annual International Symposium on Microarchitecture*, 2011.

[25] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual International Symposium on Microarchitecture*, 2013.

[26] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual International Symposium on Microarchitecture*, 2006.

[27] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[28] S. M. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving cache performance using read-write partitioning," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20thInternational Symposium on.*, 2014.

[29] Z. Zhang, Z. Zhu, and X. Zhang, "Design and optimization of large size and low overhead off-chip caches," *IEEE Trans. Comput.*, Jul. 2004.