Adaptively Reduced DRAM Caching for Energy-Efficient High Bandwidth Memory

Payman Behnam,*Student Member IEEE*, and Mahdi Nazm Bojnordi, *Member IEEE*

Abstract—In-package DRAM cache provides a higher bandwidth than conventional memory systems. Adapting the cache management to the run-time characteristics of each application seems a promising approach improving bandwidth efficiency and performance. Regrettably, fine-grained cache block monitoring and adaptation often becomes impractical due to its significant bandwidth, performance and hardware overheads. This paper proposes a novel mechanism for monitoring cache blocks using two parameters that are adjustable at run time. We propose two low-cost counter-based mechanisms to realize the block monitors in DRAM. Moreover, we propose a novel scheduling mechanism that opportunistically transfers the counter information to the DRAM stack when the data movement overhead reaches its minimum. Our simulation results on a set of data intensive parallel applications indicate that the proposed mechanisms achieve averages of 31%, 24% performance improvements over the state-of-the-art DRAM cache architectures. System energy savings over the same baselines are 29%, 18% on average.

Index Terms—Computer Architecture, High Bandwidth Memory, DRAM Caching.

1 Introduction

High bandwidth memory (HBM) has been proposed to enable large scale in-package memory systems that provide high bandwidths in excess of Tbps [3], [9]. One promising design approach to HBM systems is to build a general-purpose cache for accelerating data-intensive applications. However, remarkable challenges must be addressed properly before one can fully exploit all the performance and energy potentials of HBM caching.

One key challenge for designing an efficient HBM cache is to select an appropriate caching granularity. Existing proposals have examined two different approaches for fine- and coarse-granularity cache architectures. A fine-grained cache [12], [25], [29], [37] provides a better data management within the cache space; however, significant bandwidth and memory storage may be required for tag management. In contrast, a coarse-grained DRAM cache [13], [16], [17], [18], [23], [26], [39] reduces the tag management overhead by increasing the size of cache blocks from tens of bytes to kilobytes. A coarser granularity may result in a more significant bandwidth consumption and more data transfer over the in-package and off-chip interfaces. The simultaneous increase in bandwidth and data transfer may degrade performance of most applications that have limited spatial locality.

One of the key challenges in fine-grained cache architectures is the high cost of monitoring individual cache blocks at run-time. This has been the main motivation behind numerous stochastic solutions for DRAM caching in the literature [12], [18], [39]. In particular, stochastic mechanisms have used sampling counters for page placement [18], replacement policy in coarse-grained architectures [39], and bypassing the DRAM cache during a miss fill [12]. While these solutions can reduce the implementation costs, they may lead to making costly inaccurate decisions and becoming suboptimal.

We introduce *RedCache*, a fine-grained HBM cache architecture based on adaptively reducing the load of DRAM cache according to the run-time application characteristics. RedCache provides a more deterministic approach to run-time monitoring of individual cache blocks using a pair of access indicators: upper (γ) and lower (α) bounds. RedCache tunes the bounds at run-time to better capture only bandwidth-hungry blocks in HBM. As DRAM read and write have different requirements, RedCache provides different mechanisms for the read and write accesses. Alpha is used to eliminate all unnecessary cache accesses (including the first tag checks) for cache blocks that have not been identified as bandwidth hungry. RedCache employs HBM to store the information of individual blocks and exploits a novel DRAM scheduling mechanism to access the block information when bandwidth overhead is minimal. For a wide range of data-intensive applications, RedCache achieves virtually the same performance as an ideal RedCache implementation with in-situ tag processing capabilities. Overall, this article makes the following contributions.

Contribution 1: We study the run-time behavior of various data-intensive applications and count their number of reuses for individual cache blocks. We then estimate the bandwidth requirements of each block at run-time. We observe that (1) DRAM cache blocks have different bandwidth requirements, and (2) most last accesses to a DRAM cache block are writes. These two observations become the foundation of various optimizations in this work.

Contribution 2: To facilitate adaptive caching of DRAM blocks, we define two run-time parameters: (1) a local parameter (α) for each DRAM cache page that exploits the similarities of behaviors among all its cache blocks. This parameter determines when a block should be installed in HBM. (2) a global parameter (γ) that captures the temporal similarities of all the accessed blocks during an execution.

Payman Behnam and Mahdi Nazm Bojnordi are with the School of Computing, University of Utah, Salt Lake City, Utah. E-mail: {behnam, bojnordi}@cs.utah.edu

Contribution 3: To alleviate the high cost of updates to a fine-grained HBM cache, we propose a novel scheduling mechanism that opportunistically transfers cache blocks to HBM when the bandwidth overhead is minimal.

Our simulation results on a set of data-intensive parallel applications indicate that RedCache achieves averages of 31% and 24% performance improvements over the state-of-the-art Alloy and Bear cache architectures, respectively. Respective energy savings over the same baselines are 29% and 18% on average. When applied to a two-way set associative in-package cache, RedCache achieves averages of 32% and 30% improvements in performance and energy.

2 Design Principles

Caching is not free and may not be useful for all data blocks. Inserting a cache block into an HBM cache consumes additional memory bandwidth for data placement and tag management, occupies in-package DRAM, and necessitates tag checking on every future access to the block. On the other hand, HBM provides a higher bandwidth than main memory, which makes it more suitable for storing bandwidth-hungry blocks. Therefore, a significant challenge in designing efficient HBM caches becomes how to strike a balance between bandwidth consumption and caching overheads for individual data blocks. To address this design challenge, we make the following observations across a set of parallel applications[1]. First, increasing the bandwidth utilization does not necessarily results in a higher system efficiency; second, HBM cache blocks exhibit diverse bandwidth requirements over the execution time of different applications; and third, the majority of last accesses to the cache blocks are write accesses. We exploit these findings to monitor the individual cache blocks at run-time and adaptively tune HBM caching at fine granularity.

2.1 Bandwidth Efficiency

The overall performance of a system with HBM caching is significantly influenced by how efficiently one can utilize the aggregate bandwidth of HBM and main memory. We consider three system topologies to analyze the bandwidth efficiency of HBM-based caches (Figure 1). We model a No-HBM system comprising a multicore CPU and off-chip DRAM without an HBM cache. We also consider an IDEAL HBM system that employs a perfect HBM cache with 100% hit rate. IDEAL never misses a requested cache block; however, it consumes additional bandwidth and storage for tag checks. (Please notice that IDEAL is not necessarily the best system configuration due to not exploiting the off-chip bandwidth at all.) The No-HBM and IDEAL systems represent two extreme cases in comparison with a third system using a normal HBM cache between the CPU and off-chip DRAM. HBM implements the Alloy Cache architecture [29] for understanding the bandwidth requirements of various applications.

Assume that CPU sends a read request for a 64B data block. No-HBM transfers the requested data on the DRAM interface with no overhead bits, thereby requiring the least amount of transferred data per block. The IDEAL system

1. Details of the experimental setup are provided in Section 5.

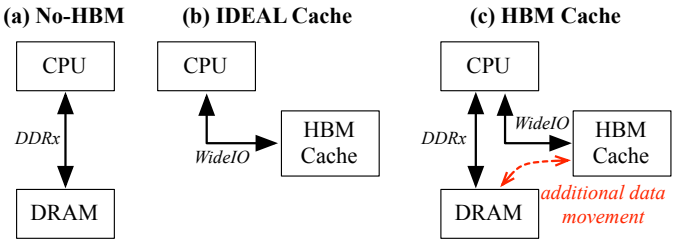

Fig. 1. Example system topologies for No-HBM (a), IDEAL (b), and HBM cache (c).

provides a higher memory bandwidth and achieves a better performance than No-HBM; however, it needs to transfer 72B cache lines (including data and metadata) on the WideIO interface. In the HBM cache system, the bandwidth consumption and the transferred data both increase because of the additional traffic between main memory and the HBM cache for block placement and replacement on every cache miss. We study the efficiency of all three systems through measuring the aggregate bandwidth consumption and the transferred data over the WideIO and DDRx interfaces. Figure 2 shows how bandwidth efficiency is impacted by system topology (a) and data granularity (b). Each design point represents relative amounts of aggregated bandwidth and data transfer averaged across all of the evaluated applications. In the system topology plot, all of the design points are normalized to No-HBM. IDEAL with more channels consumes about $6\times$ of the No-HBM bandwidth and requires 33% more data to be transferred on the WideIO and DDRx interfaces. This significant increase in the bandwidth utilization results in a $4.5\times$ superior performance over No-HBM. The HBM cache system benefits from both WideIO and DDRx interfaces to utilize a slightly higher bandwidth than IDEAL. However, a considerable portion of the WideIO and DDRx bandwidths is consumed for transferring blocks between main memory and HBM, which results in 40% performance degradation over IDEAL. Based on these observations, we set our design objectives towards balancing the bandwidth utilization and the amount of data movement over the interfaces.

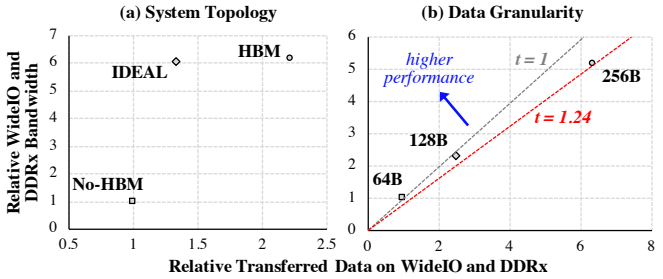

Fig. 2. Bandwidth efficiency of various system topologies (a) and block granularities (b).

We further study how bandwidth efficiency is impacted by the data movement between main memory and HBM. Figure 2(b) shows three HBM cache systems using various data granularities (i.e., 64, 128, and 256 bytes) for transferring cache blocks between main memory and HBM. (All the numbers are normalized to 64B HBM.) Prior work has

also studied coarse-grained caching for large in-package memories to reduce the overhead of tag management [16], [17], [18], [23], [39]. Installing a larger data block in the HBM cache may help to reduce the miss rate and gain speedup for the applications with high spatial locality. However, transferring large data blocks requires more bandwidth that may become counterproductive for the cache blocks with low spatial locality and high address conflicts such as our evaluated parallel applications. For the evaluated parallel benchmarks, we observe respective averages of 12% and 21% hit-rate improvements when increasing the granularity from 64B to 128B and 256B. However, using coarse grained blocks results in a significantly larger bandwidth consumption and more transferred data, thereby degrading the average performance by 8-24%. We use the bandwidth-efficiency plots to visualize the direction of performance optimization in HBM cache systems. Given that x and y axes are respectively the transferred data and bandwidth, the slope of the plot is the inverse of the execution time (i.e., $bandwidth = \frac{TransferredData}{Time}$). Every design point is located on a line ($y = x/t$) that passes through the origin with the gradient $1/t$. (Although not shown in the figure, each performance line follows a roofline model that is bounded at the peak WideIO plus DDRx bandwidth.) To achieve a higher performance, the goal is to select from design points that are located on lines with larger gradients. Among the points of each performance line, we observe that the closer point to the origin results in less interface utilization and better energy-efficiency. Accordingly, for the evaluated parallel applications, fine-grained block management (e.g., 64B blocks) is the best configuration.

2.2 Bandwidth Requirements

Every application exhibits a unique bandwidth requirement for each cache line that depends on the mapping of data to the cache address space at run-time and the inherent characteristics of the applications. RedCache is designed to monitor the run-time requirements of individual cache lines and to identify the cache blocks with high bandwidth costs for insertion to the HBM cache. Figure 3 shows the relationship between bandwidth costs and the number of block reuses for different applications in the No-HBM system. On the y-axis, each plot represents the total amount of off-chip bandwidth consumed over the course of execution for various blocks. Every point on the x-axis indicates a set of all data blocks with the same number of reuses, called a *homo-reuse* group. To accurately capture the applications' characteristics, we compute the bandwidth cost based on the exact number of DDRx cycles required for serving each DRAM request. For the evaluated benchmarks, we observe that a considerable amount of bandwidth cost is due to accessing only a subset of cache blocks that exhibit a narrow range of reuses. This observation motivates us to design a low cost mechanism that identifies these *costly blocks* and transfers them to the HBM cache. Details on the proposed techniques are provided in the next sections.

2.3 Last Block Updates

Most applications exhibit a common pattern for updating the HBM blocks, which can be used for improving the bandwidth efficiency. We observe that more than 82% of the last

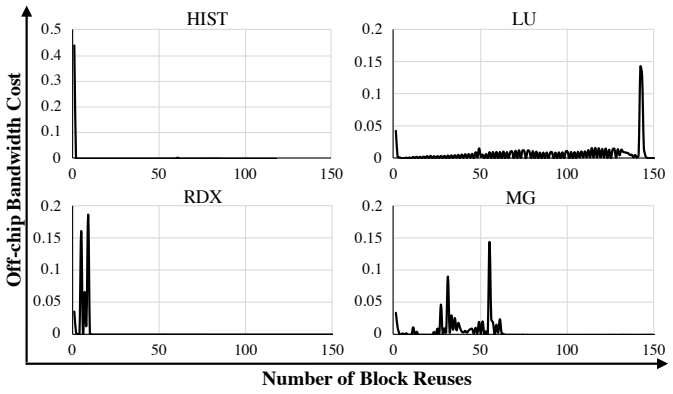

Fig. 3. Bandwidth requirements for four example parallel applications.

accesses to cache blocks in HBM cache are writebacks from the CPU to update a cache line. These last write accesses are counterproductive mainly because they (1) introduce an unnecessary bandwidth and energy overhead for updating cache lines before being moved to the main memory and (2) impose an additional bandwidth overhead for changing the HBM bus direction from a read for tag checking to a write for updating data. Recent studies [11], [33] prove that avoiding frequent changes of bus direction improves the system performance and energy efficiency. One difficulty is to accurately recognize the last writes among all the requests generated for each application. RedCache employs a monitoring mechanism on individual cache blocks to identify the last writes and route them to DRAM directly.

3 Adaptively Reduced HBM Caching

3.1 RedCache Block Management

RedCache proposes a novel control mechanism for managing bandwidth-hungry data in the HBM cache. One way to identify *costly cache blocks* for insertion into HBM is to compute a reuse count for each individual block and compare the resultant value against a threshold to determine if the block contributes in bandwidth consumption significantly. Beside reuse counts, the population of blocks with the same number of reuses determines the significance of bandwidth consumption by each group of *homo-reuse* blocks[2]. Figure 4 illustrates the computed reuse counts for an example application and a histogram plot of the bandwidth costs required by homo-reuse blocks. (Real examples of such histograms are provided in Figure 3.)

Block	Count
X	20
...	...
L	0
...	...
H	10
...	...

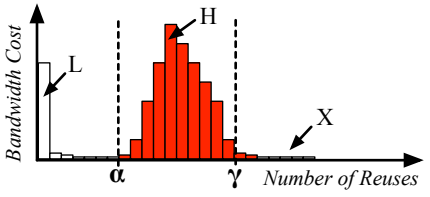

Fig. 4. Demonstration that shows how α and γ can help to define *costly cache blocks* for insertion into HBM

2. RedCache identifies multiple data blocks as a homo-reuse group if they exhibit the same number of reuses at runtime.

L, H, and X are three cache blocks that exhibit the key attributes of three possible classes of data. The entire dataset is classified using two reuse count thresholds, α and γ. The α parameter determines *the minimum number of reuses* for a block to be identified as highly reused data. Cache blocks with reuse counts less than α, such as L, are not able to amortize the bandwidth and storage costs of caching in HBM due to their relatively low reuse counts. As a result, RedCache prefers to keep such L-type blocks in the off-chip DRAM even if they require high memory bandwidths. The γ parameter threshold is defined to categorize *the highly reused blocks based on the significance of bandwidth consumption* made by homo-reuse groups. H-type blocks are highly reused and contribute to the majority of bandwidth consumption. In contrast, despite their high reuse counts, the X-type blocks require a relatively lower bandwidth. RedCache transfers the H- and X-type data to the HBM cache; however, the X-type blocks are considered as the first candidates for eviction or invalidation from HBM if further capacity for H-type blocks is necessary. Please note that α and γ are determined at run-time based on application demeanor. Caching the frequently access blocks and on-chip dead block prediction mechanism have been explored by recent work in the literature [20], [22], [28], [30]. RedCache is different from dead block prediction solutions in the following ways. First, almost all of dead block methods try to increase hit rate or decrease power consumption using prediction and stochastic solutions. Second, in dead block prediction, some status bits need to be kept that leads to an extra accesses to DRAM cache located in a different die. This accesses leads to performance and energy-efficiency degradation. Third, dead block prediction evict some blocks with zero reuse that may waste the bandwidth and performance. In RedCache, the goal is to improve system performance and decrease system energy by increasing the efficiency of bandwidth utilization in HBM caches by identifying bandwidth-hungry blocks rather than evicting dead blocks. Moreover, instead of evicting zero reuse blocks, we only invalidate some low bandwidth-hungry blocks during only write operations without a need to an additional access to the DRAM cache. We believe the existing dead block prediction solutions in the case of DRAM cache are not helpful to improve performance and bandwidth efficiency Although previous work such as SMS [32] and STeMS [31] show high spatial locality in streaming applications, this is not the case in evaluated parallel applications. Especially in the case of graph applications, there are a lot of irregular access patterns that make spatial locality less effective. we should explain independent relation to physical address as well.

3.2 Runtime Block Classification

A perfect implementation of the RedCache block management requires a global knowledge of the ultimate number of block reuses and the aggregate bandwidth consumptions for homo-reuse groups per every user application, which is not practical. Instead, RedCache proposes an adaptive block management mechanism that employs runtime counters to estimate the bandwidth costs and the number of block reuses. The proposed mechanism employs the counters to constantly tune the α and γ thresholds based on the runtime characteristics of applications. Theoretically, every cache block requires a pair of α- and γ-counters that compute the number of reuses for adjusting the thresholds and maintaining data in HBM. The α-counter computes the number of accesses to every cache block stored in the main memory before placement in the HBM cache. Whereas, the γ-counters track the total number of reuses for individual cache blocks in the HBM cache before eviction.

3.2.1 Alpha Counting

As alpha counting is necessary for the entire memory space, every cache block requires a counter. However, tracking each cache block with a counter may result in a significant memory overhead. For example, a 32GB main memory requires a 512MB additional space for storing 8-bit α-counts per 64B data blocks. In addition to the significant area and capacity overheads, accessing a large table of α-counts may result in large energy and delay overheads per memory access, thereby degrading performance and energy-efficiency. RedCache reduces the costs of α-counts through (1) sharing counters among cache blocks, (2) storing the counts in the main memory, and (3) buffering only a subset of the α-counts on the processor die for fast and energy-efficient block management. We observe that the majority of cache blocks within each 4KB OS page exhibit the same reuse counts. We compute the average standard deviation bins of number of reused blocks within a page across all the evaluated applications. Our results show that in average, 90% of blocks inside a page falls into [0,1), 6% of the blocks falls into [1,2) and the rest belong to other intervals. Based on this observation, RedCache provides a single α-count to compute the average number of accesses to all the 64B blocks within each 4KB page. Therefore, the memory requirement for maintaining α-counts is decreased by 64×. Similar to existing work [39], the count values are added to the page table in the main memory. [3] On every update to the CPU TLBs (i.e., miss rate in TLBs is low [39]), the α-counts are fetched from the main memory (as the part of the page table) and stored in a buffer at the block manager of the RedCache controller. RedCache enjoys a virtually free ride by the existing mechanism for accessing α-counts stored in the main memory [8]. An on-chip buffer with the same number of entries as in the CPU TLBs is used to store the α-counts for physical page numbers (Figure 5). For every incoming memory request, the contents of a corresponding α-count is updated and its new value is sent to the block manager logic. Then, the block manager determines if the block is yet placed in the HBM cache.

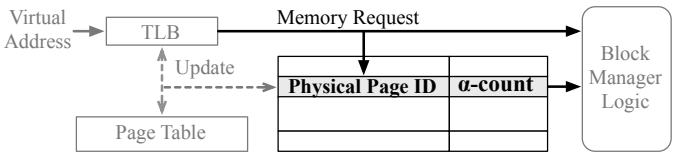

Fig. 5. Illustrative example of the proposed alpha counting mechanism.

3. Also, they can be stored in main memory independent of the page tables requiring only 8MB of memory overhead.

3.2.2 Gamma Counting

Gamma counting is only necessary for the existing cache blocks in HBM. The γ-counts may be stored as parts of the tag bits. For example, every 64B data block with 8B tags and ECC is now augmented with an additional byte that represents the reuse count. (A 1.3% memory overhead is required for storing the reuse counts.) Each reuse count is set to zero once its corresponding block is placed in HBM and is incremented on every following reads and writes.[4] A cache block becomes a candidate for eviction or invalidation from HBM if its reuse count is greater than or equal to the adaptive γ value. In other words, γ represents an expected lifetime for the HBM cache blocks at any time.

Not only do multiple applications exhibit different lifetimes but also the expected lifetime varies during the execution of a single application. The γ value is updated on a regular basis to capture the temporal characteristics of each execution phase. On every cache hit, RedCache uses the count value of the recently accessed block to compute the new γ (Figure 6). To average out the abrupt deferences among the counts, we adopt a linearly ascending/descending approach to update γ. The count and γ values are compared by the block manager. If they are different, γ will be incremented or decremented to reduce the gap. Please note that we thoroughly study both sub- and super-linear functions as well as different approaching speeds. However, the evaluated applications exhibit a better performance and energy-efficiency when a linear function is employed.

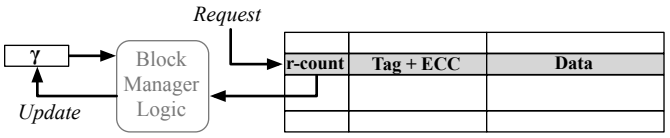

Fig. 6. Illustrative example of the proposed gamma counting mechanism.

4 RedCache Architecture

This section provides the RedCache system overview and explains two architectural mechanisms for alpha and gamma counting in RedCache. As the proposed mechanisms are supplementary to each other, the designers may include one or both of the proposed mechanisms in an in-package cache systems based on the application needs.

4.1 System Overview

Figure 7 shows an overview of the RedCache architecture. An off-chip DRAM system is employed as the main memory under a DDR4 interface. The RedCache controller exploits the HBM dice as an in-package cache for storing bandwidth-hungry data blocks in the processor package, thereby eliminating the needs for accessing the off-chip memory frequently. Along the lines of prior work on fine-grained cache [12], [29], HBM rows are partitioned into multiple blocks of tag and data, each of which is accessed using one read or write command. For example, the Bear cache stores 28 blocks per every 2KB row, where each block comprises 64B data and 8B metadata [12]. Similarly, RedCache stores 28 blocks per row while further utilizing the remainder bytes per row for augmenting every metadata with an eight-bit reuse count. Therefore, RedCache provides virtually the same capacity as the Bear cache. To implement the gamma counting mechanism, the controller maintains a register for the γ value on die. Moreover, a lookup table is used by the RedCache controller to store the α-counts for managing the cache blocks.

4. In practice, RedCache employs saturating counters for tracking block reuses.

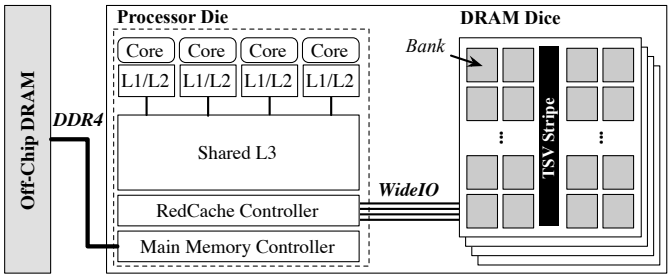

Fig. 7. Illustrative top view of the proposed RedCache architecture in a multicore system.

The block manager at the RedCache controller follows the operations shown in Figure 8 to optimize HBM caching based on the proposed alpha and gamma counting mechanisms. All the caching operations are optimized based on the assumption that a single tag and data may be accessed per every transfer on the HBM interface. All memory requests need an initial read access for tag checking, where it also fetches the data from HBM to the controller. On a read hit, no follow up accesses are necessary; however, a second HBM access is required if the request is a write hit. The three main components of the flow are (1) alpha counting and forwarding the least frequently accessed blocks to the main memory, (2) gamma counting and evicting the last writes from HBM, and (3) normal HBM caching for the bandwidth hungry blocks.

4.2 Alpha Counting Unit

The RedCache controller employs a set of 6-bit down counters to realize alpha counting for individual memory pages. The α-counts start from a non-zero positive number and are decremented on every memory access until they reach 0. (No further decrement is performed if the counter is zero.) The initial value of an α-count must represent the expected number of page accesses before any data from the page is reused. For example, the 65th access to a page with 64 cache blocks will be a block reuse if all the previously accessed data are touched once. We count reuse per page; Since it is a direct mapped all access may refer to one block. So, no need to touch all of them once to be considered as reuse On the first access to a page, α-count is initialized with $\frac{n}{\zeta}$; where, n is the total number of blocks per page and ζ represents the expected block reuses at the time of initialization RedCache is able to adapt α to the application characteristics at run-time. For example, consider initializing α for a new page in two different applications HIST and LU when ζ is respectively set to 1 and 32. RedCache considers a larger α

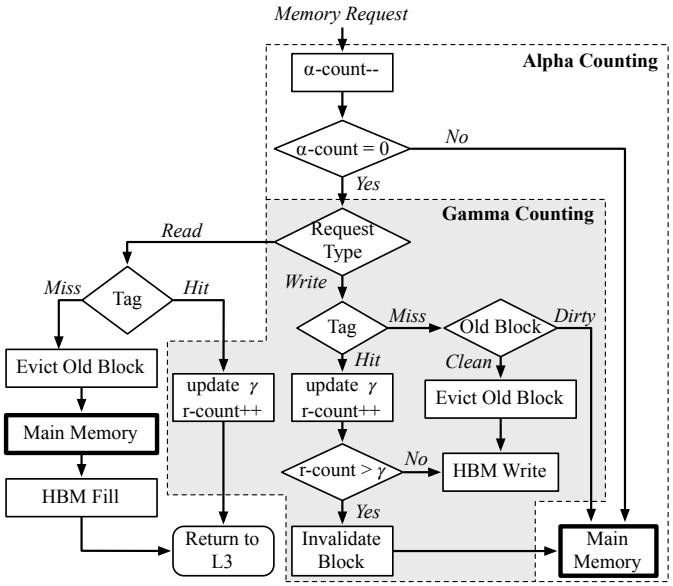

Fig. 8. Example flow of the necessary operations for applying alpha and gamma counting in RedCache.

few important WideIO timing parameters [34] required for two tag reads with and without a write in between. In the absence of the write, the second read is delayed by a column-to-column delay (*tCCD*). The write, however, increases the delay significantly due to three additional timing parameters enforced between the write and the second read. The additional timing parameters (i.e., *tBURST*, *tCWD*, and *tWTR*) are necessary to ensure (1) the block transfer on the bus is not interrupted, (2) write operation in DRAM layers is complete, and (3) the bus direction is ready for reading data.

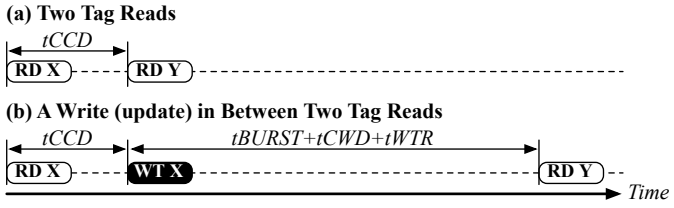

Fig. 9. Diverse impact of block updates on the WideIO bandwidth.

(=64) for HIST to skip more blocks from HBM caching as the application exhibits a low block reuse. In contrast, a smaller α (=2) is computed for LU that results in installing cache blocks in HBM after the first page; therefore, bandwidth-hungry data blocks are more likely to be cached by HBM.

4.3 Gamma Counting Unit

RedCache employs γ to alleviate the overheads of last block updates in the HBM cache (recall from Section 3). As shown in Figure 8, an incoming request may undergo gamma counting if its corresponding α-count is zero. A tag checking is necessary for either of reads and writes to determine if the requested block exists in the HBM cache. On both read and write hits, the block's r-count is incremented to track the number of block reuses. Moreover, γ is updated with respect to the r-count value on every tag hit (Section 3.2.2). For every write hit, if the newly computed r-count equals γ, the HBM cache block is invalidated and the write request is forwarded to the main memory. Otherwise, the RedCache controller updates the block contents with an HBM write operation and r-count is incremented. On a write miss, pushing the block into HBM may require an additional writeback to the main memory if an eviction is required and the existing old block is dirty. To avoid updating both HBM and main memory, RedCache writes the block into HBM only if the old block is clean; otherwise, the block is directly written to the main memory.

Unlike alpha counting, the counters for tracking gamma (i.e., r-counts) are stored in HBM; therefore, any update to an r-count must be performed using a WideIO *write* command. As mentioned in Figure 8, r-counts doesn't need to be updated on cache misses. For any write hit, the block's r-count can be updated as part of the block write operation; therefore, there is no need for any additional writes. However, every read hit necessitates an extra WideIO write to update the block. These additional writes may significantly degrade bandwidth efficiency and performance. Figure 9 shows a

Changing bus direction from a read to a write for updating data is expensive in terms of latency and energy. For instance, This bus turnaround delay *tWTR* is about 7.5-9.5 ns for multiple DDR generations [11]. To alleviate the high cost of block updates in RedCache, we propose an r-count update (RCU) manager that supplements the WideIO command scheduler. On every read hit, the RCU manager receives a copy of the block with updated r-count. The block is stored in an internal buffer that accommodates up to 32 entries. Figure 10 shows an illustrative example of the RCU architecture including the RCU manager logic, a 32-entry content-addressable memory (CAM) for block indices, and a 32-entry random addressable memory (RAM) for storing the cache blocks. The RCU manager relies on a set of status signals from the transaction queue to decide when an update can be performed with a minimal impact on bandwidth-efficiency and performance. To accomplish this goal, the RCU manager postpones each r-count update until at least one of the following events occurs. (1) The command scheduler serves a block write to the same index—i.e., channel, rank, bank, and row—as that of the queued RCU request. Therefore, the additional delay by the RCU request can be lowered to *tCCD*. This condition is evaluated by the CAM component on every write issued by the command scheduler. (2) The transaction queue becomes empty. Thus, all of the queued RCU request are served without delaying any cache requests. (3) The RCU queue is full. Our simulation results on the evaluated parallel applications indicate that in more than 97% of the total cases, none of the condition becomes true. This means that the additional latency will be reduced by a factor $\frac{tCCD}{tBurst+tCWD+tWTR}$ = 6.375. Moreover, it prevents changing bus direction from read to write and vice versa.

The RCU queue can be viewed as a 2.5KB memory that stores 32 recently read data. We observe that a number of requested blocks by future accesses may be found in the RCU queue. As a result, RedCache further employs the RCU buffer as a block cache for eliminating some of the HBM accesses.

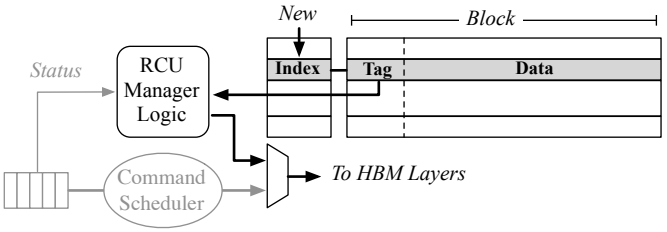

Fig. 10. The proposed RCU manager.

5 Experimental Setup

5.1 Architecture

For all of the evaluations, we consider a sixteen-core out-of-order CPU with three levels of on-die cache: L1 and L2 are private per core and L3 is shared by all of the cores. For the HBM cache system, we consider multiple in-package DDR4 DRAM layers connected to the processor die through an eight-channel WideIO interface [14], [34].Data bus is 128-bit wide and cache tags are placed together data in unused ECC bits [15], [37]. We model a 32GB main memory with two DDR4 channels, two ranks per channel, and eight banks per rank [5]. The main memory and HBM have the same access latency. Table 1 shows the simulation parameters considered for RedCache and the baselines.

TABLE 1
The evaluated system configurations.

Processor	
Core	16 4-issue OoO cores, 256 ROB entries, 3.2 GHz
IL1/DL1 cache	64KB/64KB, 2-way/4 way, LRU, 64B block
L2 cache	128KB, 8-way, LRU, 64B block
L3 cache	8MB, 8-way, LRU, 64B block
DRAM cache	
Specifications	2GB, 4 channels, 8 rank/channel, 16 banks/channel, 1600MHz DDR4, 128 bits per channel
Timing (CPU cycles)	tRCD:44, tCAS:44, tCCD:16, tWTR:31, tWR:4, tRTP:46, tBL:10 tCWD:61, tRP:44, tRRD:16, tRAS:112, tRC:271, tFAW:181
Off-Chip Main Memory	
Specifications	32GB, 2 channel, 2 ranks/channel, 8 banks/rank,1600 MHz DDR4, 64 bits per channel
Timing (CPU cycles)	tRCD:44, tCAS:44, tCCD:61, tWTR:31, tWR:4, tRTP:46, tBL:10 tCWD:44, tRP:44, tRRD:16, tRAS:112, tRC:271, tFAW:181

5.2 Workloads

We assess power and performance of RedCache and the baseline systems by executing a mix of 17 applications: 15 parallel and six serial applications. The parallel programs are selected from the NAS Parallel Benchmarks [6], SPLASH-2 [35], and Phoenix [36] benchmark suites. The serial programs are selected from integer and floating-point SPEC2017 benchmark suites [4]. All the serial applications are executed in the rate mode. We use `GCC` to compile all of the applications with `-O3` flag. For all the benchmarks, we consider warming up the cache until the cache is full; then, we simulate the next ten billion instructions or until the application completes, whichever happens first. Table 2 shows the workload characteristics and their corresponding input sets.

5.3 Methodology

We use the ESESC [19] simulator for modeling a multi-core system with three on-die cache levels. The simulator is heavily modified to model an integrated cycle-accurate module for the in-package DRAM using a detailed WideIO interface. Similarly, we integrate a cycle-accurate model for the off-chip DRAM via a DDR4 interface. On top of the WideIO controller, we implement the cache controllers for the Alloy [29] and Bear Cache [12].

TABLE 2
Workloads and data sets.

Label	Benchmarks	Suite	Input
FT	Fourier Transform	NAS	Class A
IS	Integer Sort	NAS	Class A
MG	Multi-Grid	NAS	Class A
CH	Cholesky	SPLASH-2	tk29.0
RDX	Radix	SPLASH-2	2M integer
OCN	Ocean	SPLASH-2	514x514 ocean
FFT	FFT	SPLASH-2	1048576 data points
LU	Lower/Upper Triangular	SPLASH-2	isiz02=64
BRN	Barnes	SPLASH-2	16K particles
HIST	Histogram	PHOENIX	100MB file
LREG	Linear Regression	PHOENIX	50MB key file
GCC	C Compiler	SPEC2017	gcc-smaller.c
X264	Video Compression	SPEC 2017	BUCKBunny.264 1280x720
XZ	Data Compression	SPEC 2017	CPU2006docs.tar
MCF	Route Planning	SPEC2017	15000 nodes, 162202 Active Arcs
NAB	Molecular Dynamics	SPEC 2017	1am0 11222214447 122
NAMD	Molecular Dynamics	SPEC 2017	apoa1.input

We implement six variants of RedCache to fully assess its performance and energy benefits, which include all or some of the proposed optimizations. RedCache is the main architecture that includes alpha and gamma counting, as well as the RCU management to alleviate the cost of r-count updates. We also model a basic version of the RedCache, called Red-Basic, that exclude RCU. We model a more futuristic architecture with in-DRAM processing capabilities, called Red-InSitu. This variant of RedCache enables updating r-counts inside DRAM layers with no need for transferring r-count values over the WideIO bus. As Figure 11 shows, Red-InSitu enables the global row buffer (ROB) to perform tag checking, r-count updating, and gamma comparing. Red-InSitu employs XOR and NOR gates for tag checking and comparing gamma against block r-counts. To reduce the complexity of the increment logic, we use a linear feedback shift register (LFSR) that generates a pseudo-random sequence instead of ordered numbers. The LFSR's polynomial is set to $X^8 + X^6 + X^5 + X^4 + 1$ to produce $2^8 - 1 = 255$ distinct pseudo random numbers [27]. For each increment, a previously computed count is first loaded from DRAM to the LFSR; then, the result of increment is written back to DRAM. This, however, makes the count values incompatible with γ. To address the issue, the cache controller converts γ to a corresponding γ' before sending it to the in-situ logic. This structure requires only three additional gates rather than sixteen gates for an 8-bit block counting with ripple carry adders.

We compute the area and delay overheads of in-situ r-count management using the DRAM Power Model [10] with a 55nm technology. We then scale the results to a 22nm technology. We also model Red-Gamma an an in-DRAM version of gamma counting with Alloy caches and Red-Alpha as a direct mapped cache with alpha counting only. We accurately model the energy and performance of the HBM and DRAM controllers. We use CACTI 7.0 [7] to compute the additional delay and per-access energy of table accesses for alpha counting and RCU management at the memory controller. The system energy and power

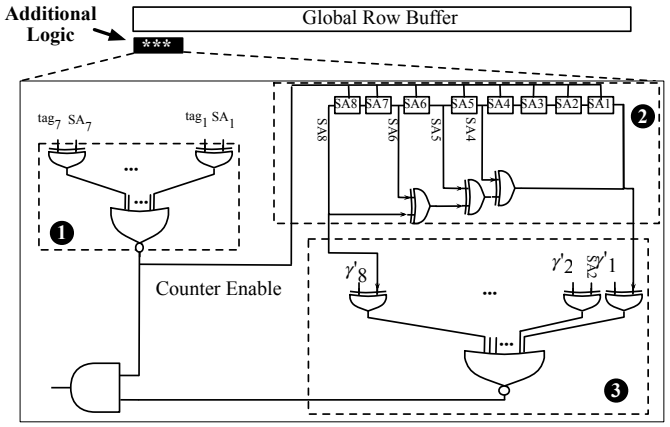

Fig. 11. The proposed in-memory logic for gamma counting in RedCache.

computation is done using the ESESC simulator [19] in coordination with McPAT [24] for the processor die, Micron power calculator [1], [2] for the main memory, and prior work on HBM memories for the in-package DRAM cache architecture.

6 Evaluation

We evaluate the area overhead, system performance, the in-package cache and main memory traffics, and the energy consumption of the baseline systems as well as RedCache. We further study the impact of each optimization solutions and different parameters such as DRAM cache sizes, and number of banks. Also, we examine a set associative RedCache.

6.1 Hardware Overhead

RCU requires a 2.5KB buffer with 32 entries, each of which is 73B wide. We model the RCU buffer as a fully associative cache, where the CAM component stores 16-bit cache indices. The results indicate a 300ps access time for the buffer, which is modeled as one additional CPU cycle to every HBM access. For every search and read operations we consider 16pJ and 17pJ respectively for our energy evaluations. Similarly, we compute the energy and delay overheads of the 768B alpha counting table, which is modeled as a 4-way set-associative cache. The energy and delay of the alpha counting table are 1ps and 25ps, respectively. Our results for the Red-InSitu model indicate less than 1% area overhead and 3% more power consumption due to the additional logic for r-count management and tag checks.

6.2 Execution Time

Figure 12 shows the execution time of different DRAM cache architectures normalized to that of the Alloy Cache for 14 parallel workloads. For all of the applications both α and γ contribute in reducing the execution time; however, the impact of α is greater than γ (27% versus 14%). The reason is that γ influences the write requests by invalidating a block from the HBM cache. In contrast, α affects both read and write requests by decoding if a data block should be placed in the DRAM cache. By putting counter values in the RCU queue, RedCache can reach almost an execution time close to that of Red-InSitu (i.e., about 98% of Red-InSitu). RedCache outperforms Alloy Cache and Bear Cache by 31% and 24%, respectively. Red-InSitu outperforms them by 33% and 26%. Unlike Bear Cache, RedCache is not a stochastic solution. RedCache can eliminate all the unnecessary cache accesses before tag checking, even for the blocks that have not been identified as bandwidth hungry. Moreover, Bear Cache does not provide solutions for installing and evicting blocks in the HBM cache.

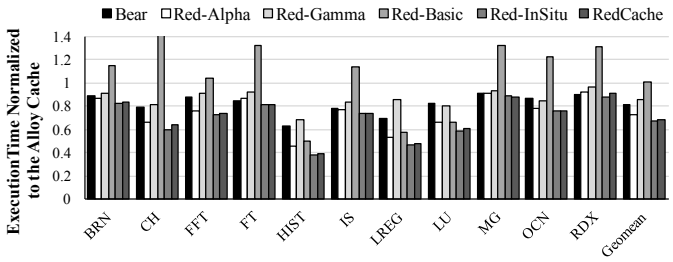

Fig. 12. Relative execution time.

6.3 Energy

RedCache reduces the HBM energy for several reasons. First, many HBM accesses are omitted because α can decide to bypass HBM and send the request to the main memory directly. Second, further HBM accesses are eliminated on a cache miss if the existing block in HBM is dirty. The request is sent to the main memory directly without evicting the existing dirty block from HBM. Third, RedCache decreases the execution time hence the static energy. Figure 13 shows the DRAM cache energy for all the evaluated systems. RedCache improves the HBM energy by 37% and 42% over the Bear and Alloy baselines. RedCache also outperforms Red-InSitu in terms of HBM energy because it does not perform any computation inside HBM.

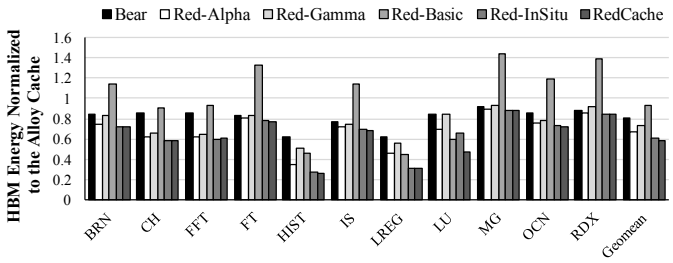

Fig. 13. Relative HBM cache Energy

RedCache reduces the system energy by 29% and 18% compared to Alloy and Bear Caches. Red-InSitu doesn't need to transfer the counter values over the HBM channels and reaches the best system energy compared to the other baselines (33% over the Alloy Cache). Figure 14 shows the system energy for all the evaluated systems.

We also compute the energy-delay product for the evaluated architectures. RedCache reduces the energy-delay product by 51% and 40% compared to the Alloy and Bear baselines, respectively.

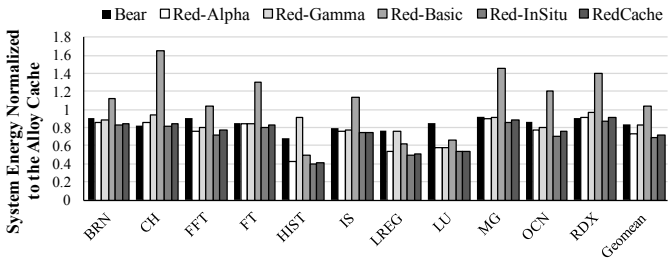

Fig. 14. Relative system energy.

6.4 HBM and Main Memory Traffic

The aggregated transferred bits over the HBM and main memory channels for the evaluated workloads normalized to the Alloy Cache baseline is demonstrated in Figure 15. In general, the amount of transferred data is reduced due to eliminating HBM accesses. However, sending metadata (i.e., counter values) over the HBM interface increases the aggregated traffic in Red-Basic and RedCache. Hence, the aggregated traffic will not be reduced considerably in comparison with Alloy Cache (i.e., about 2-3% less than Alloy Cache).

Red-InSitu overcomes this problem by updating the counters inside the HBM cache. It outperforms Bear Cache by 11%. In RedCache, we exploit the HBM bandwidth to achieve the same gains of Red-InSitu by putting the counter values in RCU. Note that unlike Bear [12] and Banshee [39], RedCache doesn't take advantage of bypassing policy in the case of a miss fill [12]. Nevertheless, this technique is orthogonal to the proposed RedCache and can further reduce the aggregated traffic for RedCache. However, both α and this method has to be considered simultaneously to efficiently put the cache blocks inside HBM. Further studies of applying this technique to the RedCache is left for future work.

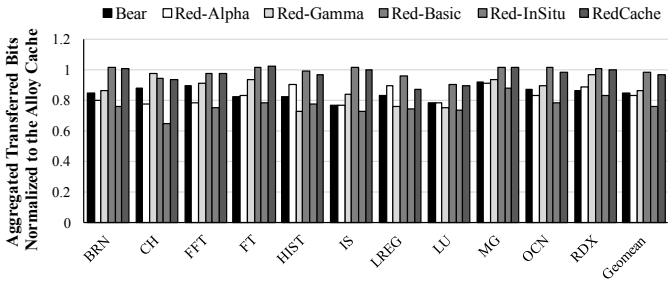

Fig. 15. Relative aggregated transferred bits over HBM and main memory channels.

6.5 Discussion

This section examines several sensitivity analyses to study the impacts of HBM sizes and banks. For the following experiments, we use the same configuration mentioned in Table 1 and only change the parameters that are specifically mentioned. All the results are normalized to Alloy Cache with the same configuration.

6.5.1 Varying the HBM Size

Tables 3 and 4 report the normalized execution time and HBM energy to Alloy cache for various HBM sizes. Table 3 shows that increasing the HBM size for RedCache results in a reduced execution time benefit by 4.85% (form 38.89% to 34.04%). In the case of Bear Cache, this reduction is larger (12.81%).

TABLE 3
Geomean of execution time normalized to Alloy Cache for various HBM sizes.

Sizes	RedCache	Bear Cache
0.5GB	61.11%	77.71%
1GB	62.04%	80.19%
4GB	65.96%	89.92%

As shown in Table 4, the RedCache energy benefit is reduced by 5.13% while increasing the HBM size from 0.5GB to 4GB. This reduction in energy benefits is 9.33% in the case of Bear Cache.

TABLE 4
Geomean of the HBM cache energy normalized to Alloy Cache for various HBM sizes.

Sizes	RedCache	Bear Cache
0.5GB	56.48%	75.70%
1GB	57.29%	78.07%
4GB	61.61%	85.03%

6.5.2 Varying the Number of HBM Banks

A similar trend is observed when the number of banks varies. By increasing the number of banks per channel from 4 to 32, the performance benefits are decreased by 3.98% for RedCache, which is less than 10.36% for the case of Bear Cache.

TABLE 5
Geomean of the execution time normalized to Alloy Cache for various number of HBM banks.

Sizes	RedCache	Bear Cache
4banks	61.85%	76.25%
8banks	62.02%	80.20%
32banks	65.83%	86.61%

Table 6 shows that the DRAM cache energy for both RedCache and Bear Cache changes equally (i.e., about 4%) while varying the number of HBM banks.

TABLE 6
Geomean of HBM cache energy normalized to Alloy Cache by varying number of banks.

Sizes	RedCache	Bear Cache
4banks	56.54%	77.95%
8banks	56.88%	79.10%
32banks	60.32%	81.87%

Overall, these results demonstrate that the Bear baseline is more sensitive than RedCache to the in-package DRAM size and the number of HBM banks.

6.6 Evaluating for the SPEC Workloads

We study the potentials of RedCache for multiple serial programs selected from SPEC 2017. In general, serial programs show less conflict misses in HBM. Therefore, we also consider mixed workloads that is a combination of multiple programs. Due to the heavy load of simulations in our cycle-accurate simulator, we only present the results for Bear, Red-InSitu, and RedCache. Figures 16 and 17 illustrate the execution time and system energy normalized to those of Alloy Cache. RedCache and Red-InSitu improve the execution time by respectively 7% and 9% over Bear Cache. In terms of system energy, RedCache and Red-InSitu are better than Bear Cache by 10% and 12%, respectively.

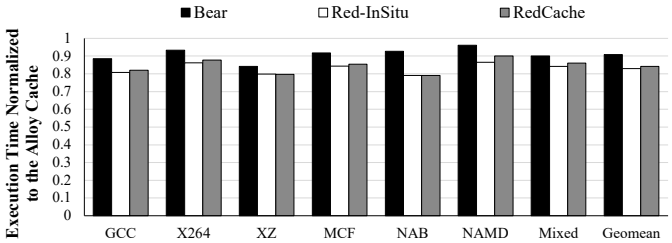

Fig. 16. Relative execution time of SPEC benchmarks.

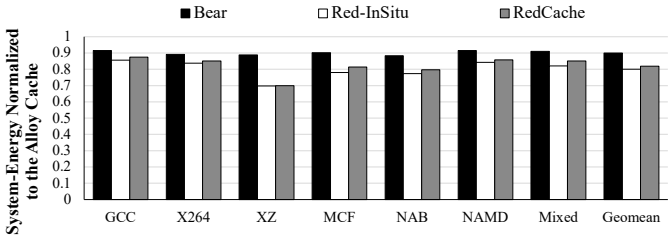

Fig. 17. Relative system energy of SPEC benchmarks.

6.7 Set Associative RedCache

To improve hit rate and execution time, we build a 2-way set associative RedCache architecture. Similar to Accord Cache [37], the set associative RedCache is based on way-install and way-prediction. For each cache block, if α decides to put the cache block in HBM, the incoming cache block is routed to a preferred way. To do so, we employ the Probabilistic Way-Steering (PWS) and Ganged Way-Steering (GWS). In PWS, cache blocks are driven to a preferred way based on a probabilistic approach. In GWS, cache blocks of a spatially contiguous region are sent to a way where an earlier cache block from that region was installed. This preferred way is used as the default way prediction. Figures 18 and 19 demonstrate the relative execution time and system energy of a set associative Alloy Cache (Assoc-Alloy), Red-Cache, and a 2-way set associative RedCache (Assoc-Red). Associative RedCache improves the execution time and system energy by respectively 32% and 30% over Assoc-Alloy.

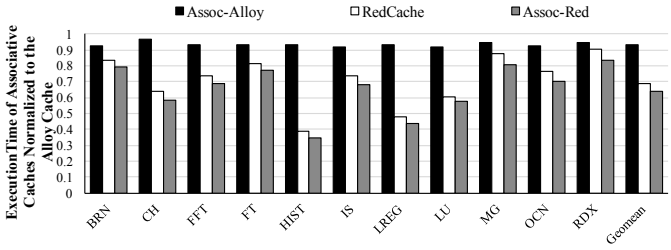

Fig. 18. Relative execution time for associative cache.

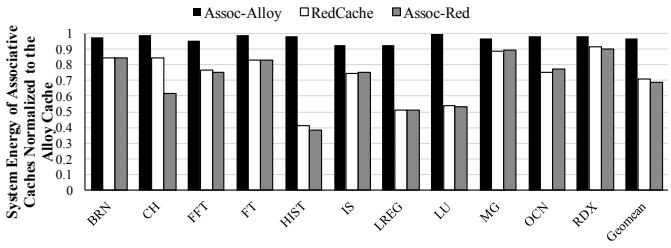

Fig. 19. Relative system energy for associative cache.

6.8 Where RedCache Stands

Figure 20 shows the bandwidth efficiency of all the evaluated cache architectures for the parallel workloads. RedCache and Red-InSitu are almost located on the same line, which indicates that RedCache achieves virtually the same performance as Red-InSitu. RedCache achieves the highest bandwidth utilization. However, due to transferring metadata, the amount of transferred data is worse than Bear Cache.

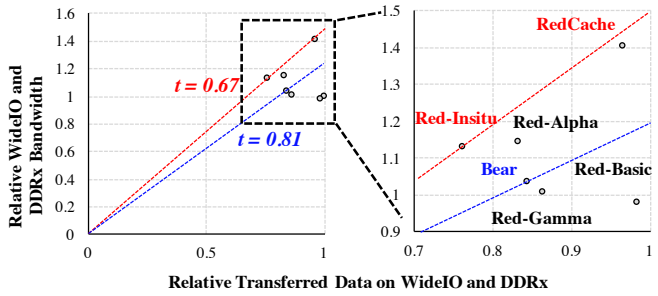

Fig. 20. Efficiency of bandwidth utilization.

7 Related Work

A large body of work has been done to improve the DRAM cache performance. Loh-Hill Cache [25] proposes a 29-way set-associative architecture that increases the hit ratio. It also increases the latency and bandwidth waste per request as it requires transferring three cache lines for a tag check. Instead, Alloy Cache [29] forms a tag and data entry in a direct mapped architecture to reduce latency.

Bear Cache [12] (1) employs a bit for each cache block (DCP) indicating if it's in LLC to eliminate probing on a dirty block eviction; and (2) alleviates bandwidth waste due to misses via buffering tags of lately accessed neighbor blocks (NTC). Unlike Bear Cache, RedCache is not a stochastic solution. RedCache can eliminate all unnecessary cache

accesses before tag checking, even for the blocks that have not been identified as bandwidth hungry blocks. Also, Bear Cache does not provide solutions for installing and evicting blocks in/from HBM cache. The proposed optimizations by Bear Cache (e.g., bandwidth aware bypass) can complement RedCache.

Footprint Cache [17] is a page-based set associative DRAM cache that fetches only those blocks within a page that is accessed during the page's residency in cache. Although it eliminates extra off-chip bandwidth associated with the page-based architectures, it is not scalable since it stores tags on-chip. Moreover, it does not provide low hit latency. Unison cache [16] tries to achieve high hit rates and low DRAM cache access latency by taking the best of Alloy Cache [29] and Footprint Cache [17]. In the Unison Cache, each DRAM row includes two sets and four pages. It reads all tags in DRAM row, predicts the way and then read the data block from that way. However, in the case of miss prediction, it needs to read all the way serially. Both Footprint and Unison Caches, wastes the DRAM cache capacity due to unused blocks within a page.

TDC [23] proposes a tagless cache architecture for in-package DRAM by introducing a cache-map Translation Lookaside Buffer that stores virtual to cache address mappings. FTDC [13] solves the problem of over-fetching blocks within a page that never get used by fetching only the blocks that are likely to be used during the page's lifetime. Both TDC and FTDC impose performance overhead and complexity to keep coherent address mappings in TLBs among all the cores [39]. Banshee [39] is another page-granularity DRAM cache that takes advantage of software/hardware techniques to optimize bandwidth without compromising the latency for applications with high spatial locality. Like TDC, Banshee also uses TLB to eliminate the tag lookup overhead. However, it updates the page table and TLB entries when a *tag buffer* is full. RedCache does not impose any TLB, page table entries, tag buffer, and operating system overheads for tag checking. Instead, it has a tag management mechanism and solutions for data block placement, eviction, and bypassing the DRAM cache based on precise monitoring of the application behavior.

Young et al. [38] suggest a replacement policy to improve the hit rate of direct-mapped DRAM caches. They propose a replacement that monitors and protects the highly reused cache block via bypassing others. Similar to RedCache, the authors observed that coarse-grained approaches like Bear Cache are not efficient when the HBM size or associativity increases. However, RedCache is different from this work in several ways. This prior work contemplates replacement policies for improving the hit rate; while, RedCache goes further and even investigates placement in the main memory before tag checking. The prior work mainly suites applications with high spatial locality; while, RedCache targets parallel multi-threaded applications with low spatial locality and high address conflicts. Moreover, Redcache identifies block with a high number of reuses with two deterministic parameters at run-time. While the local parameter (α) is to some extent similar to *Efficient Track Reuse* proposed by the prior work, the global parameter (γ) is different and identifies when to evict a block from HBM. To mitigate the overhead of such precise monitoring, RedCache takes into account the physical restrictions on the bus for switching from a tag check to a data write and performs the counter updates at the right time to minimize the overheads.

Unlike the precise monitoring of RedCache, there exist several methods to detect cache blocks that are not reused after placement in caches [20], [21], [22], which are based on stochastic sampling and prediction.

8 Conclusions

RedCache brings new insight for designing control mechanisms for DRAM caches, especially for the parallel applications that show considerable amounts of conflict misses in DRAM cache. The insight is based on the new observations of bandwidth requirement of a set of parallel applications. RedCache proposes a unified architecture for block installation and eviction and also bypassing HBM cache based on the exact monitoring of dynamic behavior of applications. RedCache creates a balance between bandwidth utilization, bandwidth efficiency and caching overhead to improve performance and system energy significantly.

References

[1] Micron ddr4 power calculator. https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm.

[2] Micron lpddr3 power calculator. http://www.micron.com/.

[3] Amd. high bandwidth memory. *URL http://www.amd.com/ en-us/innovations/software- technologies/hbm.*, 2016.

[4] Spec cpu20017 benchmark suites. *https://www.spec.org/cpu2017/.*, 2017.

[5] JEDEC Solid State Technology Association et al. Jedec standard: Ddr4 sdram. *JESD79-4, Sep*, 2012.

[6] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.

[7] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):14, 2017.

[8] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[9] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, Don McCaule, Pat Morrow, Donald W Nelson, Daniel Pantuso, et al. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–479. IEEE Computer Society, 2006.

[10] K Chandrasekar, C Weis, Y Li, B Akesson, N Wehn, and K Goossens. Drampower: Open-source dram power & energy estimation tool. 2014. *URL: http://www. drampower. info (visited on 11/14/2017)*.

[11] Niladrish Chatterjee, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P Jouppi. Staged reads: Mitigating the impact of dram writes on dram reads. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[12] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. Bear: techniques for mitigating bandwidth bloat in gigascale dram caches. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 198–210. IEEE, 2015.

[13] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. Efficient footprint caching for tagless dram caches. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 237–248. IEEE, 2016.

[14] Spec JEDEC. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[15] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[16] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE Computer Society, 2014.

[17] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 404–415. ACM, 2013.

[18] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[19] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture*, HPCA'19, 2013.

[20] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186. IEEE Computer Society, 2010.

[21] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[22] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 144–154. IEEE, 2001.

[23] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless dram cache. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 211–222. ACM, 2015.

[24] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[25] Gabriel H Loh and Mark D Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 454–464. ACM, 2011.

[26] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136. IEEE, 2015.

[27] Amit Kumar Panda, Praveena Rajput, and Bhawna Shukla. Fpga implementation of 8, 16 and 32 bit lfsr with maximum length feedback polynomial using vhdl. In *Communication Systems and Network Technologies (CSNT), 2012 international conference on*, pages 769–773. IEEE, 2012.

[28] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[29] Moinuddin K Qureshi and Gabe 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 IEEE/ACM International Symposium on Microarchitecture*, pages 235–246. IEEE Computer Society, 2012.

[30] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd 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*, pages 355–366. ACM, 2012.

[31] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 69–80. ACM, 2009.

[32] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 252–263. IEEE, 2006.

[33] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C Hunter, and Lizy K John. The virtual write queue: Coordinating dram and last-level cache policies. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 72–82. ACM, 2010.

[34] Wide i/o 2 (wideio2). http://www.jedec.org/standards-documents/results/jesd229-2.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, 1995.

[36] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *International Symposium on Workload Characterization*, 2009.

[37] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 328–339. IEEE, 2018.

[38] Vinson Young and Moinuddin K Qureshi. To update or not to update?: Bandwidth-efficient intelligent replacement policies for dram caches. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 119–128. IEEE, 2019.

[39] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–14. ACM, 2017.

Payman Behnam received his B.S. and the first M.S. degrees with distinction in computer engineering from Shiraz University, Iran and the University of Tehran, Iran. He received his second master degree from the School of Computing at the University of Utah, UT, USA. His research centers on high-performance/energy-efficient designs at the intersection of machine learning and hardware systems.

Mahdi Nazm Bojnordi received the Ph.D. degree from the University of Rochester, Rochester, NY, USA, in 2016 in electrical and computer engineering. He is currently an Assistant Professor of School of Computing with the University of Utah, Salt Lake City, UT, USA, where he leads the Energy-Efficient Computer Architecture Laboratory (ECAL). His current research interests include energy-efficient architectures, low-power memory systems, and the application of emerging memory technologies to computer systems. Professor Bojnordi's research has been recognized by two IEEE Micro Top Picks Awards, an HPCA 2016 Distinguished Paper Award, and a Samsung Best Paper Award.