RedCache: Reduced DRAM Caching

Payman Behnam *and* Mahdi Nazm Bojnordi
School of Computing, University of Utah, Salt Lake City, USA
{behnam, bojnordi}@cs.utah.edu

***Abstract*—Adapting in-package caching to run-time characteristics of user applications seems a promising approach to improve bandwidth efficiency and performance. However, fine-grained cache block monitoring and adaptation are often impractical due to the significant bandwidth and energy overheads. This paper proposes RedCache that enables fine-grained adaptation at run-time via reduced DRAM caching. Two adaptive parameters are proposed to start and stop caching for individual blocks. Architectural techniques and DRAM specific control mechanisms are proposed to alleviate overheads. Our simulation results indicate averages of 31% and 24% performance improvements over the state-of-the-art Alloy and Bear cache architectures. Respective energy savings over the same baselines are 29% and 18% on average.**

I. Introduction

3D die-stacked DRAM has been proposed to enable gigascale in-package memory systems providing bandwidths in the excess of Tbps [1]–[6]. One promising design approach to exploiting such bandwidth potentials is to build a cache for accelerating data intensive applications. Existing DRAM cache proposals have examined two different approaches for fine- and coarse-granularity cache architectures. While coarse-grained DRAM caches [4], [6]–[9] reduce the tag management overhead by increasing the size of cache blocks from tens of bytes to kilo-bytes, a fine-grained cache [1]–[3] provides a better data management within the cache space. This paper centers on improving the efficiency of DRAM caching for a class of data-intensive parallel applications that do not benefit from coarse-grained caching.

One of the key challenges in fine-grained cache architecture 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 [3], [6], [10]. In particular, stochastic mechanisms have used sampling counters for page placement [10], replacement policy in coarse-grained architectures [6], and bypassing the DRAM cache during a miss fill [3]. While these solutions are effective in reducing the implementation costs, they may lead to making costly inaccurate decisions and become suboptimal solutions. Therefore, more recent work [11] suggests tracking the reuse of cache lines to guide the replacement decisions for sequential applications.

We introduce a fine-grained DRAM cache management architecture, called RedCache, that relies on reducing the load of DRAM cache adaptively with respect to the application characteristics at run-time. RedCache provides a more deterministic approach to run-time monitoring of individual cache blocks through a low cost framework. Using a pair of upper (γ) and lower (α) bounds, all of the data accesses are monitored to identify bandwidth-hungry data blocks. RedCache tunes the bounds at run-time to better capture the characteristics of each application and aims at caching only the bandwidth-hungry blocks in the DRAM cache. RedCache employs DRAM to store the information of individual blocks and exploits a novel WideIO scheduling mechanism to access the block information when the bandwidth overhead becomes minimum. The update mechanism effectively alleviates the diverse impacts of updates on the performance such that RedCache achieves virtually the same performance as an ideal RedCache implementation with in-situ processing capabilities.

Our simulation results on a set of eleven 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.

This work was supported in part by the National Science Foundation (NSF) under Grant CCF-1755874.

II. Design Principles

Caching is not free and may not be useful for all data blocks due to low bandwidth efficiency and frequent block updates.

A. Bandwidth Efficiency

Here, 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. 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.

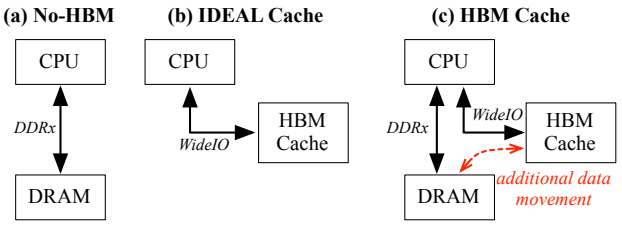

Fig. 1. Example system topologies for No-HBM (a), IDEAL (b), and HBM cache (c).

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× 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× 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.

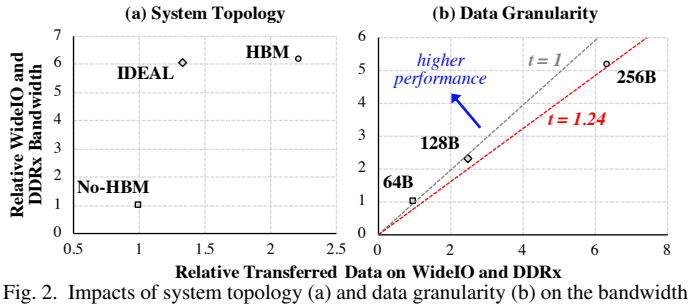

Fig. 2. Impacts of system topology (a) and data granularity (b) on the bandwidth efficiency.

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.) 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%.

B. Bandwidth Requirements

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.

C. 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 for evaluated benchmarks, more than 82% of the last 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. The recent studies prove that avoiding frequent changing of bus direction improves the performance and energy efficiency of a system [12], [13]. One difficulty is to accurately recognize the last writes among all the requests generated for each application. RedCache employs a monitoring mechanisms on individual cache blocks to identify the last writes and route them to DRAM directly.

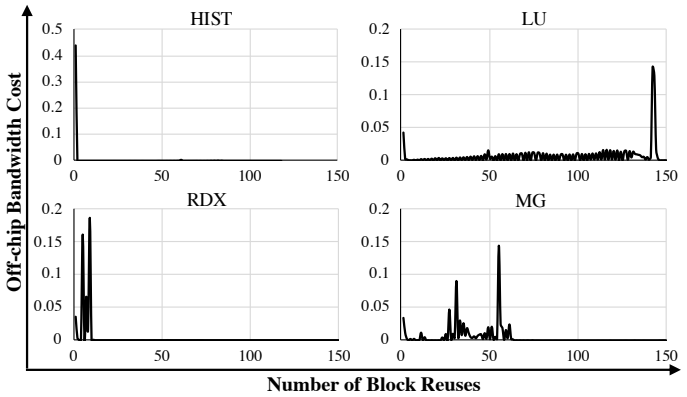

Fig. 3. Bandwidth requirements for four example parallel applications.

III. Adaptively Reduced Caching

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[1]. 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
...	...

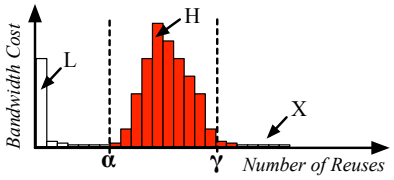

Fig. 4. Demonstration of how α and γ can help to define *costly cache blocks* for insertion into HBM.

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

[1]This paper identifies multiple data block as homo-reuse if they exhibit the same number of reuses at runtime.

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 [14]–[18]. 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 largely unable to improve performance and bandwidth efficiency.

A. 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 feasible. 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.

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. For evaluated benchmarks, we observed that the majority of cache blocks within each 4KB OS page exhibit the same reuse counts. We compute he 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 [6], [19] , the count values are added to the page table in the main memory. [2] On every update to the CPU TLBs (i.e., miss rate in TLBs is low [6], [20]), 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 [21]. 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.

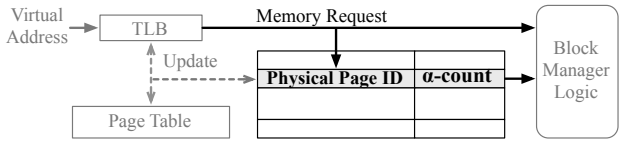

Fig. 5. Illustrative example of the proposed alpha counting mechanism.

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.[3] A cache block becomes a candidate for 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.

[2]We can store them in the main memory independent of page tables. The overhead would be 8MB.

[3]In practice, RedCache employs saturating counters for tracking block reuses.

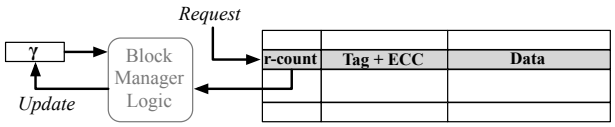

Fig. 6. Illustrative example of the proposed gamma counting mechanism.

B. System Overview

The block manager at the RedCache controller follows the operations shown in Figure 7 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.

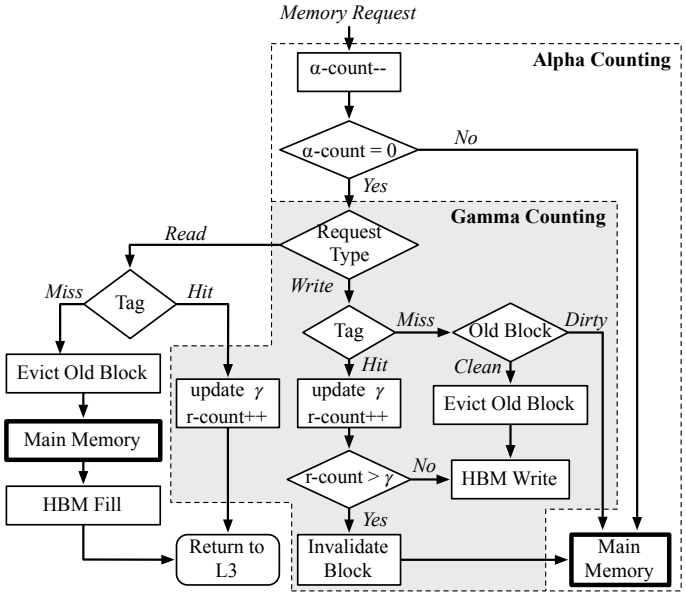

Fig. 7. Example flow of the necessary operations for applying alpha and gamma counting in RedCache.

C. Counter-based Implementation

Recent studies shows 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 [12], [13]. 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 8 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 parallel applications indicate that in more than 97% of the times, 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 which is a costly operation.

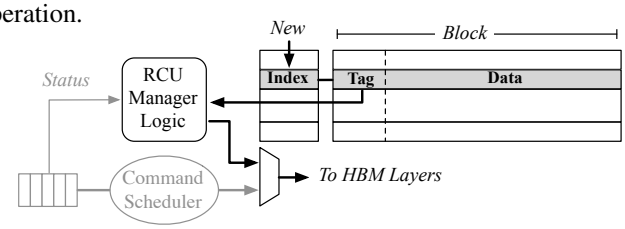

Fig. 8. The proposed RCU manager.

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.

IV. Evaluation

A. Methodology

We use the ESESC [22] simulator for modeling a multicore processor system with three on-die cache levels. Similar to prior work [23], [24], the simulator is heavily modified to model an integrated cycle-accurate module for the in-package DRAM under the WideIO interface. Also, we integrate a cycle-accurate model for off-chip DRAM under the DDR4 memory interface. On top of the WideIO controller, we implement the cache controllers for the Alloy [2] and Bear [25] as baseline models.

To fully assess the performance and energy benefits of RedCache, we implement six variants that include all or some of the proposed optimizations. RedCache is the main architecture that includes alpha and gamma counting, Bypass due to refresh 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 the RCU management from the RedCache.

For better understanding of the potentials, we also model a more futuristic architecture with in-DRAM processing capabilities. This baseline system is called Red-InSitu that enables updating the r-counts inside DRAM layers with no need for transferring r-count values over the WideIO bus. Red-InSitu enriches the global row buffer to implement tag checking, r-count updating, and gamma comparing.

We compute the area and delay overheads of in-situ r-count management using the DRAM Power Model [26] tool with a 55nm technology. We then scale the results down to the DRAM 22nm technology. We also model Red-Gamma that represents

an in-DRAM version of gamma counting applied to the Alloy caches. Red-Alpha is developed to represent a direct mapped cache with alpha counting only.

To compute the per-access energy overhead of the proposed controller, including table accesses for alpha counting and RCU management, we use CACTI 7.0 [27]. The system energy and power computation is done using the ESESC simulator [22] in coordination with McPAT [28] tool for the processor die, Micron power calculator [29] for the main memory, and prior work on HBM memories [30] for the in-package DRAM cache architecture.

For all of the evaluations, we consider a sixteen-core out-of-order CPU with three levels of on-die cache. The L1 and L2 are private per core; while the 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 [31]. The bus width is 128 bits and HBM cache puts tags with data in the unused ECC bits [32]. We model a 32GB main memory using a two-channel DDR4 DRAM [33] that comprises two ranks per channel and eight banks per rank. The access latency of the main memory and HBM are considered the same. Table I shows the simulation parameters considered for the baselines and RedCache architectures.

TABLE I
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

B. Workloads

We assess the energy and performance potentials of the proposed and baseline systems by executing 11 parallel applications. The selected parallel applications represent a mix of data-intensive programs from the NAS [34], SPLASH-2 [35], and Phoenix [36] benchmark suites. 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 applicationuntil it completes.Table II shows the workload characteristics and their corresponding input sets.

TABLE II
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

C. Execution Time

Figure 9 indicates the relative system execution time of different DRAM cache architectures normalized to Alloy Cache for executing 11 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 plays a role in the case of write requests to invalidate a blocks from the HBM-cache while α parameter contributes in both read and write requests to decide when we should put data blocks in the DRAM cache or when we should bypass it. Besides, Figure 9 manifests by putting counter values wisely in the RCU queue, RedCache can reach almost similar execution time of Red-InSitu (i.e., about 98% of Red-InSitu). RedCache outperforms Alloy Cache and Bear Cache baselines by 31% and 24% respectively while Red-InSitu outperforms them by 33% and 26%.

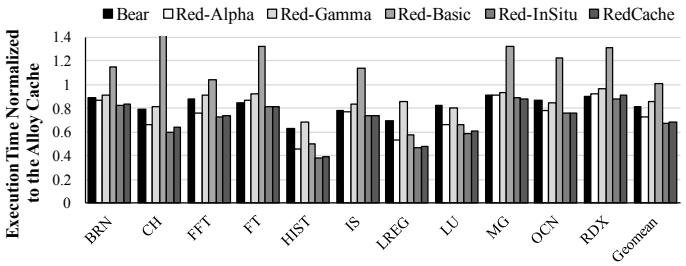

Fig. 9. Relative execution time

D. Energy

RedCache reduces DRAM cache energy for several reasons. First, several accesses to the HBM cache will be omitted since α parameter may decide to bypasses HBM cache (i.e., depending upon the counter values) and guide the request to the main memory directly. Second, due to exiting α parameter , if there is miss and the existing block in HBM cache is dirty, the request directly is sent to the main memory and hence writing back the old dirty block into main memory and installing new cache block into DRAM cache is removed. In addition, RedCache decrease execution time compared to exiting baselines. Figure 10 shows DRAM cache energy over the all possible configurations. RedCache improves HBM cache energy by 37% and 42% over the Bear and Alloy baselines. RedCache also outperforms Red-InSitu because it does not perform any computation inside HBM cache.

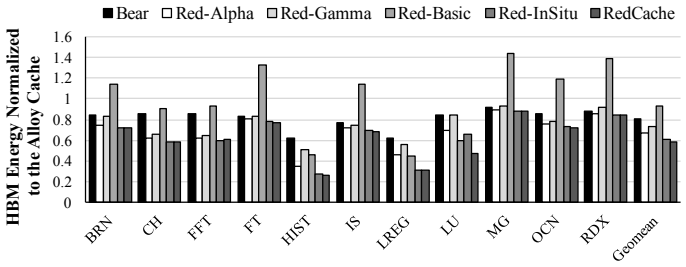

Fig. 10. Relative HBM cache Energy

Due to decreasing HBM cache energy, keeping bandwidth hungry blocks in HBM cache and reducing the overall execution time, the RedCahe decreases system energy as well. RedCache ameliorate system energy by 29% and 18% compared to Alloy and Bear Caches. Red-InSitu outperforms other architecture since it doesn't need to transfer the counter values over HBM channels and reaches the best performance compared to other

baselines (33% over the Alloy Cache). Figure 11 indicates the system energy consumption for all architectures.

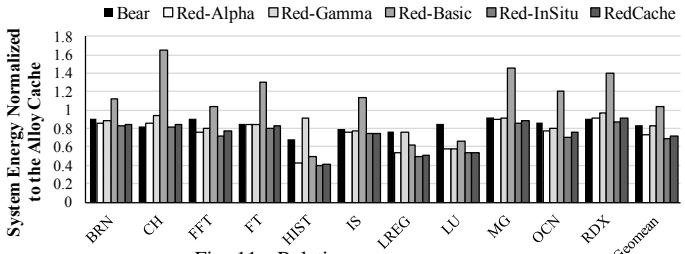

Fig. 11. Relative system energy.

V. 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] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, pp. 454–464, 2011.

[2] 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 *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*, pp. 235–246, 2012.

[3] C. Chou, A. Jaleel, and M. K. Qureshi, "Bear: techniques for mitigating bandwidth bloat in gigascale dram caches," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, pp. 198–210, 2015.

[4] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for tagless dram caches," in *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, pp. 237–248, 2016.

[5] P. Behnam, A. P. Chowdhury, and M. N. Bojnordi, "R-cache: A highly set-associative in-package cache using memristive arrays," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pp. 423–430, IEEE, 2018.

[6] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'50)*, pp. 1–14, 2017.

[7] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *ACM SIGARCH Computer Architecture News*, vol. 29, pp. 144–154, ACM, 2001.

[8] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *47th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'14)*, pp. 25–37, 2014.

[9] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless dram cache," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 211–222, ACM, 2015.

[10] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *International Symposium on High Performance Computer Architecture (HPCA'10)*, pp. 1–12, 2010.

[11] V. Young and M. K. Qureshi, "To update or not to update?: Bandwidth-efficient intelligent replacement policies for dram caches," *International Conference on Conputer Design (ICCD)*, pp. 119–128, 2019.

[12] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged reads: Mitigating the impact of dram writes on dram reads," in *IEEE International Symposium on High-Performance Comp Architecture (HPCA'12)*, pp. 1–12, 2012.

[13] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: Coordinating dram and last-level cache policies," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 72–82, ACM, 2010.

[14] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, ACM, 2007.

[15] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[16] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 144–154, IEEE, 2001.

[17] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'10)*, pp. 175–186, 2010.

[18] 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*, pp. 355–366, 2012.

[19] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 74–85, IEEE Computer Society, 2005.

[20] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pp. 29–40, 2009.

[21] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[22] E. K. Ardestani and J. Renau, "ESESC: A Fast Multicore Simulator Using Time-Based Sampling," in *International Symposium on High Performance Computer Architecture(HPCA'13)*, 2013.

[23] M. N. Bojnordi and F. Nasrullah, "Retagger: An efficient controller for dram cache architectures," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[24] M. N. Bojnordi and E. Ipek, "Pardis: A programmable memory controller for the ddrx interfacing standards," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 13–24, 2012.

[25] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro'46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[26] 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)*.

[27] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, p. 14, 2017.

[28] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, pp. 469–480, 2009.

[29] M. Technology, "Micron ddr4 power calculator," 2018.

[30] M. OConnor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*, pp. 41–54, 2017.

[31] S. JEDEC, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[32] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[33] J. S. S. T. Association *et al.*, "Jedec standard: Ddr4 sdram," *JESD79-4, Sep*, 2012.

[34] D. H. Bailey *et al.*, "NAS parallel benchmarks," tech. rep., NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.

[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] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *International Symposium on Workload Characterization(IISWC'09)*, pp. 198–207, 2009.