Hybrid Main Memory for High Bandwidth Multi-Core System

Dongki Kim, Sungjoo Yoo, *Member, IEEE*, and Sunggu Lee, *Member, IEEE*

Abstract—In the hybrid main memory sub-system which consists of DRAM and phase change RAM (PRAM), DRAM is often used as a cache. Such a configuration of DRAM cache and PRAM main memory has two problems. First, it requires high bandwidth of data movement between DRAM and PRAM due to the large granularity (i.e., DRAM row) of caching. Second, the memory bandwidth of PRAM is not fully utilized since it can be accessed only by the DRAM cache thereby limiting the memory bandwidth of hybrid main memory to that of DRAM cache. In this paper, we propose a novel architecture where both DRAM and PRAM can be accessed in parallel for higher memory performance. We analyze that the performance and fairness of hybrid memory sub-system is limited by the stall time at the data queue of memory controller. In order to resolve this problem, we propose caching data selectively in a way to reduce stall time thereby reducing memory access latency and improving fairness. Our experimental results show significant improvements in performance (by an average of 21 percent), fairness (by 2.1 times), and energy consumption (by 10 percent) compared with the best of the existing methods in a multi-core system which consists of multiple CPUs and GPU.

Index Terms—Cache memory, memory management, multicore processing, phase change random access memory

1 Introduction

DRAM has been used as main memory for a few decades due to its benefits of fast response time, low cost, etc. However, DRAM-based main memory is expected to face significant problems in terms of energy consumption and cost. Large DRAM-based main memory suffers from significant energy consumption due to refresh operations. Especially, DRAM is facing its scaling limit [1], which prevents us from benefiting from continued reduction in cost per bit.

In order to overcome these problems, there are many studies on applying emerging memory to main memory sub-system. Phase change RAM (PRAM) is considered as one of promising candidates for main memory [2], [3], [4], [5], [6], [7]. Due to its non-volatility, it does not consume refresh power. In addition, it gives better scaling than DRAM. For instance, the cell size of 3 nm [2] is already demonstrated for PRAM while it is expected to be very difficult for DRAM to enter 1x nm technology. Furthermore, PRAM can store multiple bits in one cell (called multi-level cell) like NAND flash memory. For those reasons, PRAM is more likely to be used as the low-cost large main memory. In this paper, we utilize MLC PRAM as the main memory due to its benefit of low cost.

Compared with DRAM, PRAM gives long latency, poor endurance (e.g., 10^6-10^8 writes [3]), and high write power consumption. Thus, the hybrid structure of DRAM and PRAM has been considered [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. In this perspective, Intel introduced near/far memory architecture [16], [17]. Near memory is defined as fast and small main memory which is located near the processor, while far memory is defined as slow and large main memory located far from the processor. In such a near/far memory, DRAM and PRAM are good fit for near and far memory, respectively.

Hybrid main memory structure discussed above can be classified into hierarchical [8] and flat structures [10], [11], [12]. In the hierarchical structure, PRAM is considered as background main memory and DRAM as a cache in front of PRAM. This structure gives short access latency (due to DRAM). In addition, since DRAM can absorb writes, it mitigates the problems related to PRAM write (high write power and poor write endurance). This structure, however, lacks in exploiting total memory bandwidth available in DRAM and PRAM. To be specific, the bandwidth of PRAM is not fully exploited since PRAM can be accessed only via the DRAM cache, without direct accesses from the processor.

In the flat structure, DRAM and PRAM cover disjoint regions in the address space of main memory. Thus, in terms of bandwidth utilization, the total memory bandwidth of DRAM and PRAM can be fully utilized since both can be directly accessed from the processor. However, it can suffer from the problems of long latency, high write power consumption, and short lifetime due to direct accesses to PRAM. Note that in the hybrid DRAM/PRAM main memory, the smaller lifetime of the two determines the lifetime of entire memory sub-system.

In this paper, we present a novel memory sub-system which aims at taking the best of the two structures, i.e., low latency, small write power consumption and long lifetime from the hierarchical structure and high bandwidth from the flat structure. Our proposed architecture integrates both hierarchical and flat structures. Our study shows that the performance and fairness of hybrid memory sub-system are

• *D. Kim and S. Lee are with the Pohang University of Science and Technology, Pohang, Korea. E-mail: {dongki.kim, slee}@postech.ac.kr.*
• *S. Yoo is with the Seoul National University, Seoul, Korea. E-mail: sungjoo.yoo@gmail.com.*

Manuscript received 7 Aug. 2015; revised 26 Oct. 2015; accepted 29 Oct. 2015. Date of publication 6 Nov. 2015; date of current version 18 Dec. 2015. Recommended for acceptance by Y. Chen, K. Choi, and W. Zhao. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TMSCS.2015.2498549

2332-7766 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

often significantly limited by high stall time at the data queue of memory controller which is utilized for internal data movement between DRAM and PRAM. In order to resolve this problem, we propose caching data selectively in a way to reduce stall time thereby improving both performance and fairness.

This paper is an extension of our previously published work [10] where a flat structure of hybrid main memory is proposed for CPU/GPU system and a write buffer in the DRAM absorbs write traffics bound for PRAM thereby improving performance, power efficiency and lifetime. In this paper, we extend this work as follows.

- We extend the utility of write buffer to allow for caching read data as well as write data. It is based on our analysis that read traffics often go to dirty data (in the write buffer) with high spatial locality (i.e., to the neighbor of dirty data). Thus, keeping the large granularity data in the DRAM cache is beneficial for performance.
- We identify that stall time at the data queue of the memory controller often dominates total memory access latency and incurs starvations in memory access, i.e., a fairness problem. In order to resolve this problem, we propose selectively caching the data which incur the stall time problem, which reduces pressure on the data queue and leads to the improvement of memory access latency.

The rest of paper is organized as follows. Section 2 reviews related work. Section 3 explains background and our motivation. Section 4 describes the proposed architecture. Section 5 reports experimental results. Section 6 concludes this paper.

2 Related Work

Recently, there have been active studies on hybrid main memory sub-system. Qureshi et al. [8] propose to use DRAM as a cache for large PRAM-based main memory which gives performance improvements by reducing page faults. Phadke et al. [9] analyze the potential performance of hybrid main memory sub-system and classify main memory into three types: latency optimized memory, bandwidth optimized memory and power optimized memory. They propose mapping programs to the best suitable type of memory based on profiling. Kim et al. [10] propose to use DRAM as a write buffer of PRAM on a flat structure. Chou et al. [11] propose a method to access near memory and far memory concurrently to maximize bandwidth utilization. They allocate memory pages to one of two memories considering the access rate of near memory. Sim et al. [12] also propose a flat structure which performs remapping and swap operations managed by hardware.

Yoon et al. [13] propose row buffer locality aware data caching for hybrid main memory. Based on the fact that PRAM can give the same level of bandwidth as DRAM when consecutively assessing the row buffer, they propose to cache, on the DRAM, random data incurring frequent row activations in the PRAM. In terms of selective caching in the DRAM, this is similar to our method. However, we utilize stall time as a trigger condition of data caching instead of row buffer locality.

Jevdjic et al. [14] propose to reduce bandwidth overhead of data caching by predicting the footprint of programs and caching only predicted blocks instead of entire page. Wang et al. [15] propose hybrid main memory for high bandwidth processors such as GPU. Their method monitors bandwidth required by processors and migrates data to DRAM only when there is available bandwidth. Ours is different from this work in that caching is performed based on the stall time.

The DRAM cache in the hybrid main memory sub-system shares several important issues with the 3D-stacked DRAM cache. Especially, the issue of SRAM or in-DRAM tag [18], [19], [20] and that of caching granularity [21], [22] are important and closely related with each other in both cases. Zhao et al. [18] explore the DRAM cache architecture by changing the location of tag array. Franey and Lipasti [19] propose low overhead tag array for large set associative caches with find-grained block sizes. Qureshi and Loh [20] and Loh and Hill [21] propose tag-in-DRAM (TID) to enable block sized cache line in the DRAM cache. Jevdjic et al. [22] propose the TID structure with page sized cache line as well. In our experiments, we adopt the large caching granularity, i.e., 2 KB and SRAM tag due to the small DRAM cache. In larger memory sub-systems, in-DRAM tag methods [19], [20], [21], [22] could be applied together with ours to the hybrid main memory sub-system.

There are several studies in performance improvement of PRAM-based main memory [3], [4], [5], [6], [7] which are complementary to our proposed architecture. Qureshi et al. [3] propose write cancellation (pausing) which cancels (pauses) long write operation in order to serve read requests which tend to be more latency-critical than write ones. Write pausing is also implemented in real PRAM chips [23], [24]. Jiang et al. [4] propose write truncation method for MLC PRAM. The write latency of MLC PRAM is often determined by the PRAM cells finishing program operations last. Write truncation method abandons write operations on such cells if error correction code (ECC) can recover the data. Jiang et al. [5] propose a write scheduling method considering the power budget of PRAM chip. Yue and Zhu [6] propose utilizing the asymmetry of write latency in SET and RESET. Li et al. [7] propose a method of reducing the latency of SET operation by dividing SET operation into two steps. In order to address the problem of poor endurance in PRAM, there are studies on wear leveling [25], [26], [27], [28], [29] and error correction [30], [31], [32]. These can be applied to our proposed architecture to address endurance problems. Meanwhile, there are studies on using PRAM to improve the performance and endurance of storage system [33], [34], [35]. Liu et al. [33] present a method to apply to the swap area non-volatile memory (NVM) such as PRAM. Kim et al. [34] and Liu et al. [35] propose utilizing the byte addressability and high endurance of PRAM in the mapping table of flash translation layer (FTL) in the NAND Flash memory-based storage system.

3 Background and Motivation

Fig. 1 shows a block diagram of main memory sub-system which consists of DRAM and PRAM. In the figure, the shaded area indicates the memory controller. There are two queues, request and data queues in the memory controller.

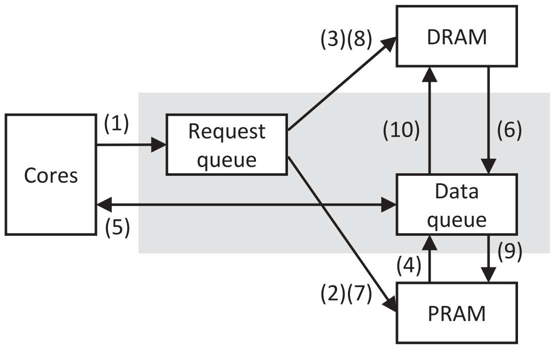


Fig. 1. Simple block diagram of hybrid memory sub-system. The arrows indicate request and data movement (shaded area: memory controller).

The data queue is utilized to temporarily store data from/to DRAM and PRAM. The data queue is also utilized for internal data movement (during fetch and dirty eviction operations) between DRAM and PRAM. Fig. 1 shows how the data queue is utilized in the hybrid memory sub-system.

In Fig. 1, assume that the DRAM is utilized as a cache. In the figure, arrows indicate request and data movements. Assume that a read miss occurs in the DRAM cache and a dirty victim is evicted from the DRAM cache. (1) When the core sends a request to the memory controller, the request is stored in the request queue. In order to process a DRAM cache miss, the memory controller sends internal requests to (2) PRAM (to read the required data) and (3) DRAM (to read the victim data), respectively. After receiving the data for caching from PRAM, (4) the data are stored on the data queue and (5) forwarded to the core. After receiving the data for eviction from the DRAM, (6) the data are stored on data queue as well. Then, in order to write the data from the data queue to each memory, the controller sends internal write requests to (7) PRAM (to store the dirty victim) and (8) DRAM (to store the required data fetched from the PRAM), respectively. After that, (9) the victim data are written to the PRAM and (10) the data for caching are written to the DRAM.

If one of the two queues of the memory controller becomes full, the core cannot help being stalled because the memory controller cannot receive requests any more. High bandwidth cores, especially, GPUs require large queues to avoid the situation that the queues are full. Especially, considering the large granularity of DRAM cache block (e.g., 2 KB DRAM page in our case), a large data queue is required.

Fig. 2 shows three cases where the data queue is (almost) full and the execution of (some of) the cores is stalled by the congested data queue. In the figure, two cores access the hybrid main memory sub-system having a DRAM cache. Fig. 2a shows the case where the data queue is completely full and no requests can be served. Figs. 2b and 2c show the cases that the data queue has a small amount of free space (white area in the data queue).

In Fig. 2b, both requests from the two cores give DRAM cache misses thereby requiring data caching (and, if needed, dirty eviction). In this figure, we assume that the size of free space in the data queue is smaller than that for handling a cache miss (total size of fetched block from PRAM and evicted block from DRAM). Especially, in the case of DRAM cache, the size of cache block (e.g., that of DRAM page or row, 2 KB) is typically larger than that of last-level cache (LLC) block (64 or 128 B). Thus, large free space would be required to serve both miss requests. In Fig. 2b, both cores will be stalled until more space becomes available.

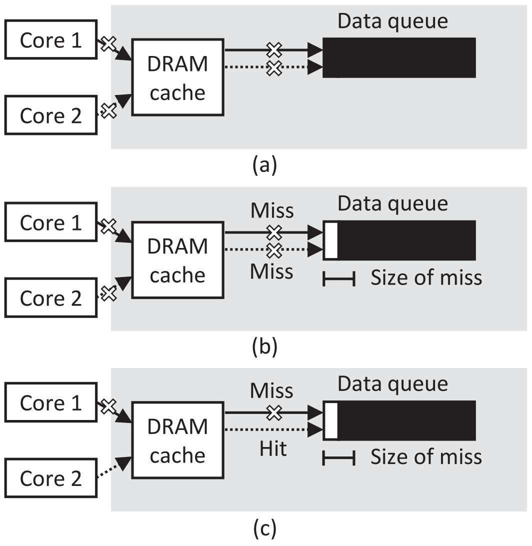


Fig. 2. Three types of data queue stall in the memory controller (shaded area).

In Fig. 2c, two cores give a hit (for core 2) and a miss (core 1) in the DRAM cache, respectively. We assume that the free space is not large enough for miss handling, but can be used to serve the cache hit (due to the small size of last-level cache block). In such a case, the hit request is served while the service of the miss one is being stalled until more free space becomes available. In this case, if the service of miss request is significantly delayed due to consecutive hit requests, core 1 can suffer from starvation. Conventional fairness schemes in memory access scheduling, e.g., PAR-BS [36] can be applied to address this problem. However, the fair memory scheduling policy alone may not be able to completely resolve the problem since the policies of fair memory scheduling, though they can enforce the ordering of request services in a fair manner, would indirectly address this problem. To be specific, the memory scheduling policy selects one of the requests ready to be served. However, according to our study, the readiness of request is often determined by the availability of data queue space. Our proposed method addresses the problem by reducing congestion on the data queue. We think a combination of our proposed method (to enable as many ready requests as possible) and existing schemes of fair memory scheduling (to determine the order of request service in a fair manner) will be an interesting topic for the future work.

Fig. 3 shows the breakdown of average memory access time. We obtained the data from 50 program sets running on CPU/GPU cores with the DRAM/PRAM main memory. The details of our experiments will be given in Section 5. As the figure shows, a significant amount (average 58 percent) of memory access time is due to stall at the data queue. Note that the cases of Figs. 2b and 2c are dominant in the case of data queue stall.

In order to reduce stall time, a large data queue can be utilized at a high area cost. However, according to our analysis in Section 5, the large data queue is not effective in reducing stall time. It is because the congestion of data queue occurs when the total memory bandwidth to serve

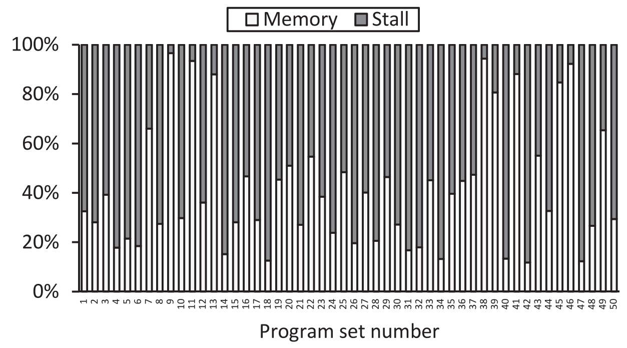


Fig. 3. The breakdown of average memory access time (normalized).

incoming requests (for reads, writes, evictions, and fetches from/to both DRAM and PRAM) becomes larger than the peak bandwidth of main memory sub-system (i.e., the total bandwidth of DRAM and PRAM). In other words, a large data queue can only delay the beginning of congestion, but cannot avoid it. Thus, we need new methods to reduce the total memory bandwidth to serve incoming requests. The total memory bandwidth required by incoming requests can be decomposed into four parts, i.e., reads, writes, fetches and evictions. We tackle the memory bandwidth required by internal movements (fetches and evictions) since the memory bandwidth required to serve reads and writes cannot be reduced by the main memory sub-system.

4 Proposed Architecture

4.1 Selective Caching

We aim at making best use of the benefits of the hierarchical and flat structures, i.e., short access latency of hierarchical structure [8] and large available memory bandwidth of flat structure [10], [11]. The proposed architecture extends the one in [10]. It is basically a flat structure. Thus, each of DRAM and PRAM is allocated disjoint regions of physical address space. In addition, as explained in [10], in order to hide the long write latency of PRAM, a dedicated portion of DRAM is used as a write buffer for the PRAM region.

In this paper, we extend the write buffer in [10] to additionally support *selective caching* in the DRAM. The dedicated portion of the DRAM previously used only for the write buffer is now used for selective caching as well as write buffering. Thus, in this paper, we call it *in-DRAM selective cache* (in short, in-DRAM cache). In the proposed architecture, we allow the following data to be cached in the in-DRAM cache.

- The data in the same page of dirty data written to the in-DRAM cache during write operation (to be explained in detail later in this section)
- The read data which are required by a core the policy of which is set to caching, typically, to reduce data queue stall (Section 4.2)

Compared with the write buffer of the baseline in [10], the in-DRAM cache has a larger granularity of data. An entry of the write buffer could contain the data of 64 bytes while an entry of in-DRAM cache can contain a DRAM page, i.e., the data of 2 KB. We adopt the larger granularity to exploit the memory access behavior as shown in Fig. 4. The figure shows the page-level re-reference behavior of CPU

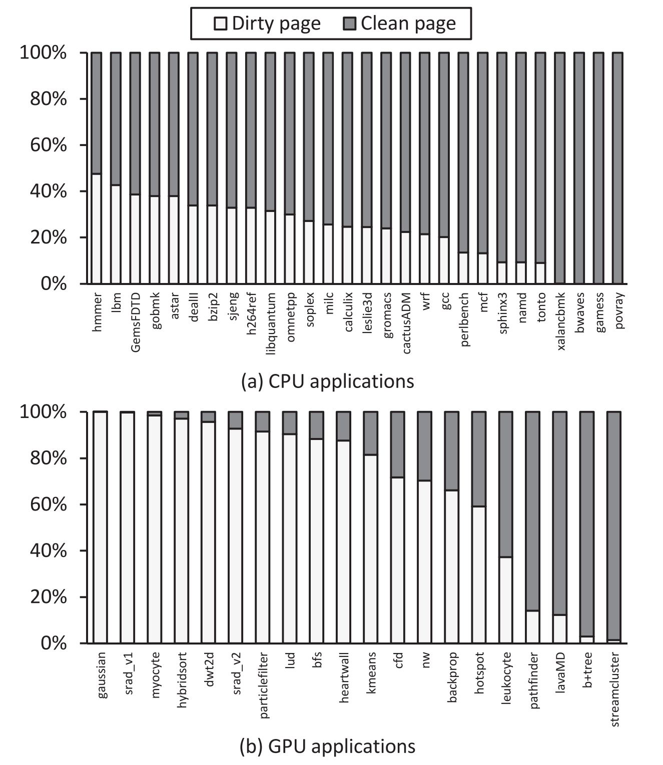


Fig. 4. Re-reference ratios of dirty page (including at least one dirty LLC block) and clean page.

and GPU programs used in our experiments. As shown in the figure, both CPU and GPU programs frequently access dirty pages. Especially, GPU programs exhibit stronger re-reference behavior on dirty pages than CPU ones. Thus, it would be beneficial to keep the entire dirty page in the in-DRAM cache in order to exploit re-reference behavior. In order to support the function of original write buffer and selective caching, we adopt sub-blocking in the in-DRAM cache. Thus, a tag entry has multiple valid bits, one for the data of 64 bytes, i.e., the size of last-level cache block.

Fig. 5 shows the proposed architecture. In the figure, shaded area indicates the memory controller. There are two queues and tag array in the memory controller. The function of two queues is the same as in the conventional DRAM cache as explained in Section 3. The tag array is used for the in-DRAM cache.

The proposed architecture serves memory requests as follows. When the memory controller receives a request, it first checks to see if the requested data belong to the DRAM region. In this case, the request is served by accessing the DRAM ((1) in Figs. 5a and 5b). If the requested data belong to the PRAM region ((2) and (5) in Fig. 5), first, the tag of in-DRAM cache is searched for a hit. Note that the tag match is performed at the data granularity of last-level cache block, i.e., 64 bytes. In case of hit (for read or write), the in-DRAM cache is accessed ((3) for write and (6) for read in Fig. 5). If there is a miss, depending on the request type (read or write), the proposed architecture performs different operations.

In case of read miss, we check to see if the required data belong to the DRAM page in the in-DRAM cache. If it is the case (which we call *page hit and LLC block miss* in (7)), we fetch the entire data of corresponding page from the PRAM

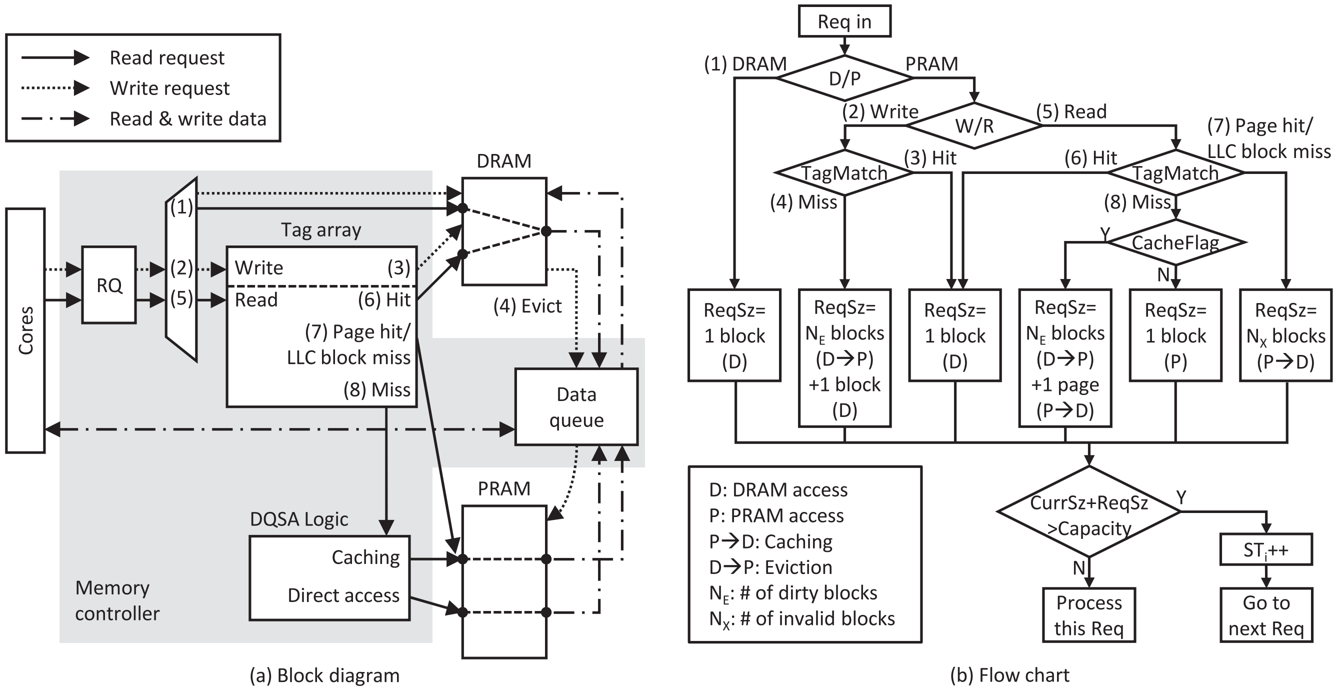


Fig. 5. Proposed hardware architecture of hybrid memory controller (RQ: request queue, Req: request, Sz: size, and Curr: current).

to the in-DRAM cache. It exploits the spatial locality shown in Fig. 4. If the required data of read request does not belong to any DRAM page in the in-DRAM cache ((8) in Fig. 5), then we apply data queue stall-aware (DQSA) selective caching. Thus, the DQSA logic is invoked to make a choice of caching the read data (after fetching the associated page from the PRAM) or direct access to the PRAM. Note that *CacheFlag* in Fig. 5b is the output register of DQSA logic. The details of DQSA logic will be explained in Section 4.2. In case of write miss, the write data of LLC cache block size are inserted in the in-DRAM cache as in the baseline [10]. The insertion can incur the write-back of dirty victim from the in-DRAM cache to the PRAM ((4) in Fig. 5). Since the in-DRAM cache can contain the data of PRAM region, we need to make sure that there is no consistency problem that stale data are accessed from the PRAM. Our proposed architecture addresses this problem as the conventional cache system. Thus, as explained in this section, it first checks the existence of data in the in-DRAM cache before accessing the PRAM.

4.2 Data Queue Stall-Aware Selective Caching

DQSA selective caching determines the per-core policy, caching or direct access for PRAM region data. In the beginning, the policies of all cores are initialized to caching. Periodically (every 100,000 cycles in our experiments), the DQSA logic is invoked to evaluate the effect of the current per-core policy, and, if necessary, change the policy in order to reduce stall time. Fig. 6 shows how the DQSA logic works. As shown in the figure, the total stall time is calculated by summing the accumulated stall time of each core ($\mathrm{N}+1$ cores in the figure). For each core (e.g., core i at the center of the figure), the difference ($Delta_i$) between the total stall time (ST_{tot}) and its own stall time (ST_i) is calculated. Note that Fig. 5b shows how the stall time of core i, ST_i is calculated. If there is any request of core i stalled due to the lack of space on the data queue, ST_i increments. Since the flow of Fig. 5b is executed every clock cycle, the stall time increments as long as there is any stalled request. If the difference of the current interval ($Delta_i$) is larger than that of the previous interval ($preDelta_i$), then the current policy does not contribute to reducing total stall time. Thus, we change the current policy. Since there are only two policies, caching and direct access, we take the other policy than the current one ($CacheFlag_i = !CacheFlag_i$). Then, both $preDelta_i$ and ST_i are updated and reset, respectively as shown in the figure. Note that the policy is independently determined at each core.

Toggling the policy has the effect of balancing the usage of data queue thereby improving fairness and, finally, overall performance. Recall Fig. 2b. In this case, both cores 1 and

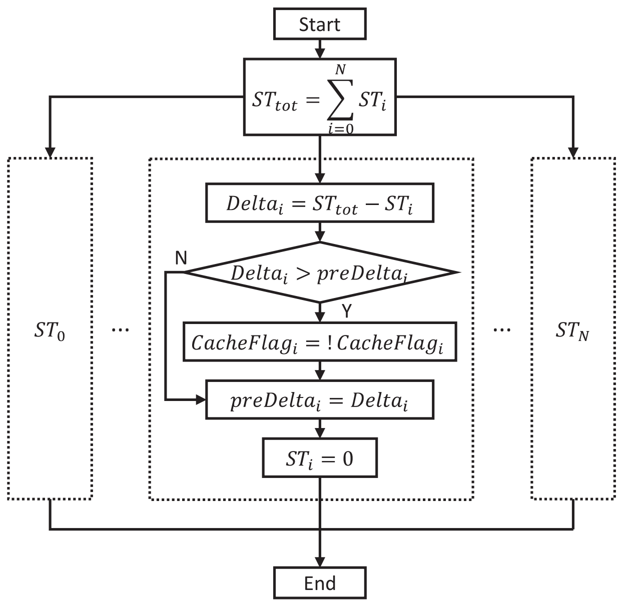

Start
$ST_{tot} = \sum_{i=0}^{N} ST_i$
ST_0
...
$Delta_i = ST_{tot} - ST_i$
N
$Delta_i > preDelta_i$
Y
$CacheFlag_i = !CacheFlag_i$
$preDelta_i = Delta_i$
$ST_i = 0$
...
ST_N
End

Fig. 6. Toggling the policy between caching and direct access.

TABLE 1
Cache Tag Array Overhead

	1,024 MB	512 MB	256 MB	128 MB
Timing	4.09 ns	3.37 ns	3.19 ns	2.62 ns
Area	5.42 mm^2	2.77 mm^2	1.48 mm^2	0.73 mm^2

2 make misses in the DRAM cache. Core 1 is assumed to take the policy of caching. Thus, it requires a large portion of data queue for a page-level data fetch from the PRAM. Core 2 is assumed to take the policy of direct access and requires a small amount of data queue resource to fetch a LLC block from the PRAM. In such a case, if core 2 continues to issue direct accesses with small-size data blocks and occupies more resource in the data queue, then core 1 can be starved, which hurts fairness and overall speedup. In this case, keeping the policy of direct access for core 2 will increase *Delta* in Fig. 6 for core 2 since the total stall time will increase (due to the increase in the stall time of core 1) and that of core 2 will not increase much.

In this case, our proposed method in Fig. 6 toggles the policy of core 2 from direct access to caching. The toggling has double effects in this example. First, the new policy of caching will make both cores 1 and 2 have a fair usage of data queue since both requires large amount of data queue resource, in case of DRAM cache miss, and the currently available free space is not enough for either of the two cores. Thus, fairness is achieved. Second, core 2, which is likely to be memory-intensive, will ultimately have more data in the in-DRAM cache due to the policy of caching, which will improve the performance of core 2 due to the low latency of DRAM. In our experiments, the data queue stall time-aware method in Fig. 6 proves effectiveness in terms of both fairness and performance.

4.3 Hardware Overhead

Compared with the conventional DRAM/PRAM hybrid memory [8], the proposed architecture has additional area cost for DQSA logic and additional tag information (i.e., valid bits for LLC block-level validity). The overhead of DQSA logic was estimated by synthesizing our RTL code (with Synopsys Design Compiler) at a proprietary 65 nm technology and is shown to give a negligible overhead of 0.0027 mm^2.

The proposed architecture is based on SRAM tag for the DRAM cache. The area overhead of SRAM tag is proportional to the amount of DRAM resource used for the in-DRAM cache. Table 1 shows the area overhead varying the size of in-DRAM cache. We obtained this with CACTI 6.5 at 65 nm technology. Note that the SRAM tag overhead includes the valid bits required for sub-blocking at the granularity of LLC block size which is the same as that of two of our baselines, row buffer locality aware data caching [13] and footprint cache [14]. As the table shows, the SRAM tag occupies up to 5.42 mm^2 at 65 nm technology. We expect such an overhead could be allowed for better performance due to DRAM/PRAM hybrid main memory compared with the iso-cost DRAM only main memory [8] and it is expected to become much smaller at advanced technologies.

TABLE 2
Architectural Parameters

CPU	8 Out-of-Order cores; 3.2 GHz; ROB size = 160; width = 4; depth = 10; 16 KB/16 KB L1$ per core; 1 MB L2$ per core
GPU	8 shaders (8 sclar processors per shader); 800 MHz; L1I/D/T/C$ = 2 KB/16 KB/12 KB/8 KB per shader; L2$ = 768 KB (shared)
DRAM	DDR 800 MHz; 1 GB; 2 channels; 1 rank; 8 banks; 1 Gb chip (x16); 4 chips per channel (x64); row buffer size = 8 KB; tRCD-CL-tRP = 16-16-16
PRAM	DDR 800 MHz; 16 GB; 2 channels; 1 rank; 8 banks; 16 Gb chip (x16); 4 chips per channel (x64); row buffer size = 8 KB; MLC read: 250 ns; MLC write: 1 fixed iteration for ‘00’, 2 fixed iterations for ‘11’, variable iterations based on probability [4] for ‘01’ and ‘10’ (averagely 8 and 6 iterations, respectively); write iteration: 250 ns; write pausing [3]; write truncation [4]; differential write [39]
Memory Controller	tag array for in-DRAM cache: line size = 2 KB; associativity = 32 ways; # sets = 16 K/8 K/4 K/2 K for 1 GB/512 MB/ 256 MB/128 MB in DRAM cache; data queue size = 128 KB

5 Experiments

5.1 Experimental Setup

Table 2 shows the architectural parameters used in our experiments. As shown in the table, we use MLC PRAM due to its benefit of low cost for large capacity main memory. We use McSimA+ [37] for CPU and GPGPU-sim for GPU as a frontend simulator to obtain the trace of memory requests. As a cycle-accurate simulator of hybrid main memory subsystem, USIMM [38] is used. We added an in-house memory trace player for GPU and an additional memory channel model for PRAM into USIMM. Our PRAM model was developed by modifying the DRAM model. It also supports write pausing [3], write truncation [4] and differential write [39]. The energy consumption of PRAM is calculated using Micron Power Calculator [40], PRAM parameters in [23][1] and the runtime results obtained by the cycle-accurate simulation.

For benchmark programs, as Table 3 shows, we used SPEC CPU 2006 benchmark suite for CPU and Rodinia benchmark suite for GPU. We made 50 program sets by randomly selecting programs, i.e., eight programs from SPEC CPU 2006 benchmark and one program from Rodinia benchmark since our architecture consists of 8 CPU cores and 1 GPU core as shown in Table 2. We used the reference input set [41] of SPEC CPU 2006 benchmark suite and also used Simpoint to simulate representative intervals of the programs. We ran 200 million instructions per CPU core and 1,600 million instructions per GPU core since our GPU consists of eight shaders. Note that Table 3 gives the information of row buffer hit rate (RBHR) and misses per kilo instructions (MPKI) to be used in our analysis.

1. Note that we used the detailed specification of PRAM chip in [23] which is confidential. For the modeling of power consumption in MLC PRAM, we utilized the same Idd values as in [23]. Thus, the write energy consumption of MLC PRAM is modeled with longer program time than in SLC PRAM which is proportional to the number of program & verify steps.

TABLE 3
Benchmark Programs

		MPKI	RBHR	Footprint [MB]		MPKI	RBHR	Footprint [MB]		MPKI	RBHR	Footprint [MB]
CPU	mcf	72.3	3%	5269	wrf	8.75	56%	934	sjeng	0.74	7%	611
	lbm	51.2	32%	2999	cactusADM	6.62	0%	544	gobmk	0.65	21%	126
	soplex	33.2	39%	469	gcc	3.73	6%	139	dealII	0.35	80%	35
	libquantum	33.0	89%	544	hmmer	2.64	58%	32	tonto	0.23	56%	37
	omnetpp	29.4	28%	1197	bzip2	2.41	48%	76	namd	0.13	59%	30
	milc	29.2	56%	3603	calculix	1.83	68%	187	gamess	0.05	34%	11
	leslie3d	26.9	58%	1243	perlbench	1.53	39%	188	povray	0.03	49%	9
	xalancbmk	15.6	83%	40	h264ref	1.32	55%	38	bwaves	0.02	75%	4
	sphinx3	14.1	61%	101	gromacs	1.13	57%	100				
	GemsFDTD	9.00	14%	1186	astar	1.05	22%	33				
GPU	gaussian	247	41%	8	srad_v2	31.0	21%	96	particlefilter	0.62	83%	9
	bfs	207	31%	144	srad_v1	26.2	63%	7	heartwall	0.34	56%	4
	streamcluster	83.3	51%	73	lud	9.86	37%	16	lavaMD	0.14	92%	2
	kmeans	41.3	54%	214	pathfinder	7.90	86%	95	leukocyte	0.05	83%	1
	cfd	41.1	49%	29	hotspot	5.04	69%	44				
	dwt2d	32.2	69%	213	myocyte	1.44	59%	3				

MPKI: misses per kilo instructions.
RBHR: row buffer hit rate.

As shown in Table 4, we evaluate five methods in our experiments including two state-of-the-art designs, footprint cache (FOOT) and row buffer locality aware data caching (RBLA) as well as the conventional DRAM cache (CONV) and our previous work (WRBF). It is because FOOT and RBLA adopt selective caching in the main memory sub-system. FOOT [14] predicts the footprint of data and caches only the data which are expected to be accessed. In our experiments, we assume an ideal footprint cache which can achieve the footprint prediction accuracy of 100 percent. Thus, it gives the maximum performance of footprint cache. RBLA [13] selectively caches in the DRAM only the data which have low row buffer locality. Both FOOT and RBLA adopt sub-blocking as our proposed one as mentioned in Section 4.3. In both cases of WRBF and DQSA, we utilize the entire DRAM for the write buffer in WRBF and the in-DRAM cache in DQSA. Thus, in WRBF, we do not apply the runtime adjustment of write buffer size in [10]. We will give a sensitivity analysis of smaller in-DRAM caches than the entire DRAM in Section 5.3.

To evaluate multi-core performance, we use the weighted speedup [42] metric which is the sum of each program's speedups when executed together compared to when executed alone. For the fairness of multi-core programs, we use the maximum slowdown [43]. This is determined by the largest slowdown of all programs in a program set.

5.2 Experimental Results

Fig. 7 compares performance, fairness and endurance of PRAM of the five methods. Fig. 7a shows weighted speedup. Since we run 50 program sets for each of the five methods, the figure shows the distributions of weighted speedup. In terms of average weighted speedup, as the figure shows, DQSA gives by 68, 50, 32 and 21 percent better performance than CONV, WRBF, FOOT and RBLA, respectively. More detailed analysis on the performance improvement of DQSA will be given shortly in this sub-section.

TABLE 4
Compared Methods

CONV	Conventional DRAM cache [8]
WRBF	DRAM as a write buffer [10]
FOOT	Footprint cache (with 100 percent accuracy) [14]
RBLA	Row buffer locality aware data caching [13]
DQSA	Data queue stall time aware data caching

Fig. 7b compares fairness in terms of maximum slowdown. Thus, the lower, the better. As the figure shows, DQSA gives the best fairness. It gives by 5.5, 3.6, 3.4 and 2.1 times better fairness than CONV, WRBF, FOOT and RBLA, respectively. It is because DQSA improves performance by avoiding the starvation in memory accesses. As the figure shows, DQSA improves both the mean and the largest values of maximum slowdown.

Fig. 7c compares the PRAM writes in terms of total bit updates. As the figure shows, there is no noticeable difference in total bit updates between DQSA and the existing methods. Even though we did not utilize wear leveling method in our experiments, we expect similar lifetimes across existing methods and ours when adopting existing wear leveling methods, e.g., security refresh [26].

Fig. 8 gives a more detailed comparison of weighted speedup between CONV and DQSA. In the figure, each point corresponds to one of the 50 program sets. We also draw a dashed line, y = x where y and x are the weighted speedup of DQSA and CONV, respectively. As the figure shows, all the points are located above the line, which means DQSA always gives better results than CONV in the 50 program sets. We found similar behavior that the weighted speedup is highly correlated between DQSA and the other methods.

Figs. 7 and 8 show the overall comparisons. Fig. 9 shows detailed comparisons for two representative cases shown as best and worst cases (in terms of the ratio of weighted speedup between DQSA and CONV) in Fig. 8. Note that even in the case of worst case in Fig. 8, the weighted speedup of DQSA is by 36 percent better than that of CONV. Fig. 9a shows a detailed comparison between

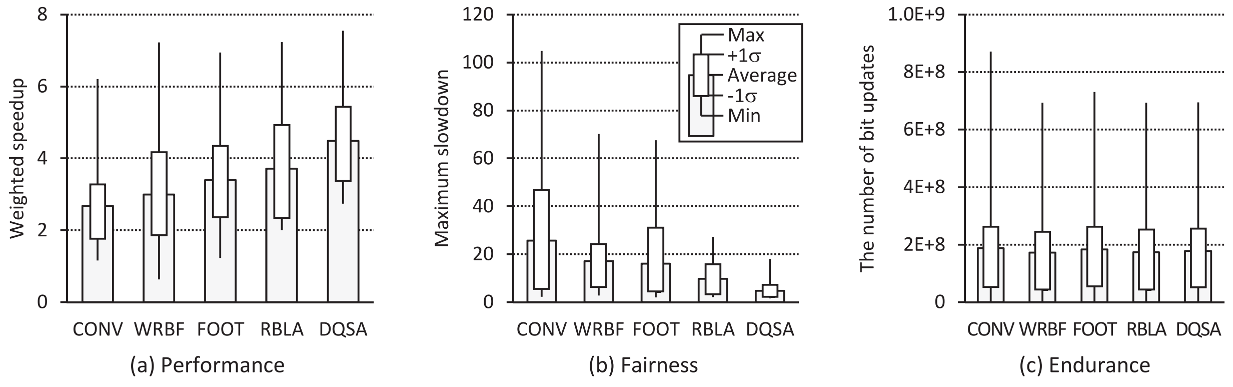


Fig. 7. Experimental results of five methods: (a) weighted speedup, (b) maximum slowdown, and (c) total number of bit updates of PRAM.

CONV and DQSA for the best case. In the figure, there are nine programs (eight CPU programs and one GPU program) along the *x*-axis. The left-axis (average memory access latency) is for bar graph and the right-axis (per-program speedup) is for circle and triangle. For each program, the left (right) bar corresponds to CONV (DQSA).

As Fig. 9a shows, DQSA gives performance improvement by reducing the stall time. Especially, low MPKI programs (e.g., *gromacs* and *povray*) benefit most from DQSA. It is because, in CONV, high MPKI programs incur stall time and low MPKI programs suffer from long stall time. To be specific, high MPKI program tend to give high row buffer hit rates and high DRAM cache hit rates, which allows them to monopolize the data queue thereby increasing stall time for the other (low MPKI) programs. In Fig. 9a, two high MPKI programs (*milc* and *sphinx3*) have relatively high row buffer hit ratio (56 and 61 percent, respectively, as shown in Table 3) and incur stall time for the other programs. Especially, the two low MPKI programs, *povray* and *lavaMD* are almost starved in memory accesses. As shown in Fig. 9a, for those low MPKI programs, DQSA significantly reduces stall time thereby improving their speedup. In this case, DQSA gives both speedup and fairness since the speedup is realized by improving fairness.

Fig. 9b gives a detailed comparison of the worst case. DQSA reduces the stall time of five programs (*gcc*, 2x *lbm*, *soplex*, and *streamcluster*) while increasing that of four programs (*sphinx3*, 2x *mcf* and *gamess*). In this case, the program incurring a significant amount of DRAM cache misses, namely, *mcf* affects the overall performance. *mcf* has a high MPKI (72.3) and low row buffer hit rate (3 percent) which translates into high DRAM cache misses. A DRAM cache miss puts more pressure on the data queue than a DRAM cache hit since the large data of DRAM page size needs to be fetched from the PRAM to the data queue and, possibly, incurring the write-back of dirty page from the DRAM to the PRAM via the data queue. In this case, DQSA changes the policy of *mcf* from caching to direct access, which reduces fetches and evictions incurred by *mcf* thereby reducing pressure on the data queue. The other programs benefit from the reduced pressure on the data queue. However, there is one exception, *gamess*, which gives low MPKI (0.05) and relatively low row buffer hit rate (34 percent).

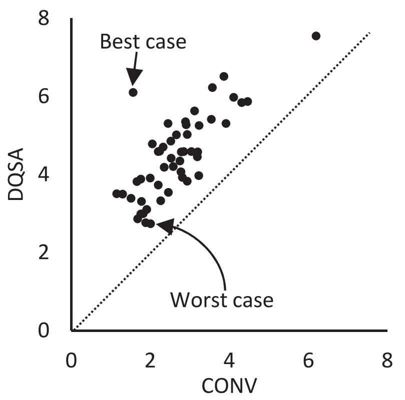


Fig. 8. Correlation of weighted speedup: DQSA versus CONV.

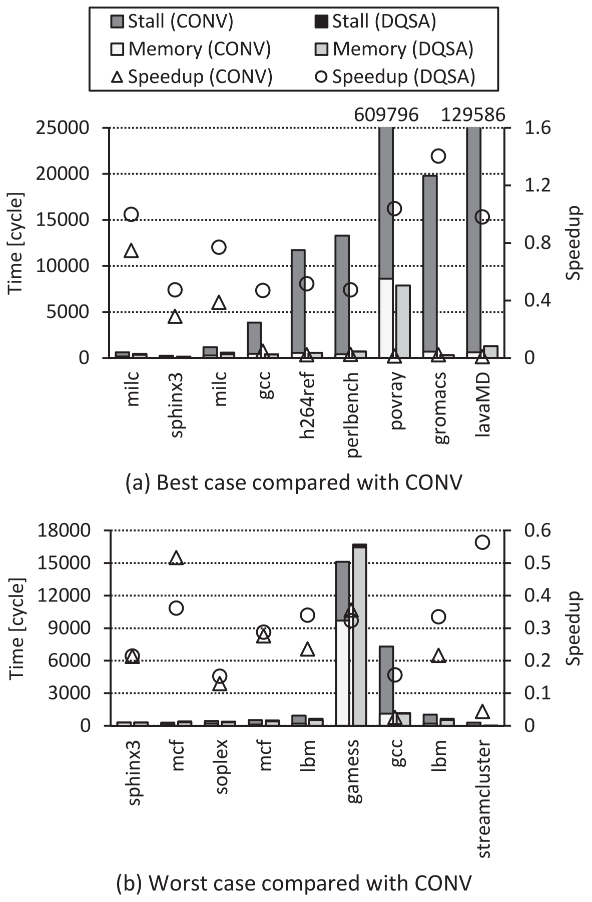


Fig. 9. Comparison of DQSA and CONV in terms of average memory access latency (left) and per-program speedup (right).

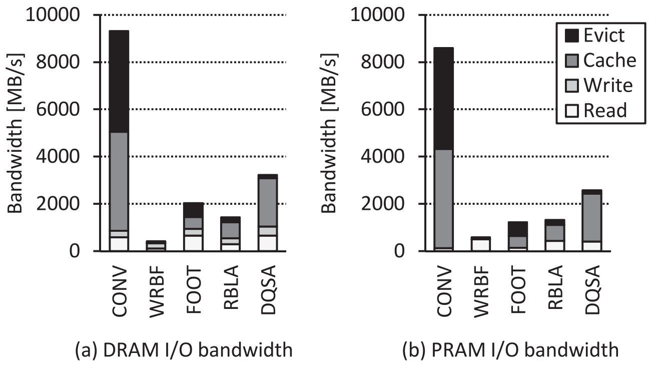


Fig. 10. Bandwidth consumption of (a) DRAM and (b) PRAM of five methods.

Our analysis is that the data of *gamess* are frequently evicted (by the other DRAM cache-friendly programs like *lbm*, *soplex*, and *sphinx3*) from the in-DRAM cache thereby incurring frequent PRAM accesses due to DRAM cache misses. As Fig. 9b shows, DQSA increases, for *gamess*, the portion of memory access (due to frequent accesses to the PRAM with longer read latency than the DRAM). Though the detailed comparisons with the other two methods, FOOT and RBLA are not given for a succinct analysis of experimental results, they also show that DQSA outperforms the other two methods by the reduction of stall time (especially for FOOT) and memory time (especially for RBLA).

Fig. 10 compares bandwidth consumption. In the figure, *read* and *write* corresponds to the bandwidth consumption to serve read and write requests while *cache* and *evict* represent the bandwidth consumption utilized for internal data movements, i.e., fetching data from PRAM to DRAM (*cache*) and evicting dirty data from DRAM to PRAM (*evict*). As shown in the figure, CONV consumes much more bandwidth than the other methods due to frequent internal data movements. Especially, the bandwidth consumption is highly skewed, i.e., the DRAM is heavily accessed while the PRAM is not. In case of FOOT, compared with CONV, bandwidth consumption for data caching is reduced since only the (to-be-)required data are fetched from the PRAM. The figure shows that RBLA has reduced DRAM traffics by selectively caching data. WRBF consumes the smallest amount of bandwidth for both DRAM and PRAM since the DRAM is utilized only as a write buffer. Thus, the DRAM is typically utilized for write requests and PRAM for read requests, which significantly reduces internal data movements. The figure shows that DQSA consumes relatively high memory bandwidth for data caching. However, caching more data contributes to improving data queue stall time as explained above.

Fig. 11 compares energy consumption. As Fig. 11a shows, DQSA gives the lowest energy consumption. It consumes by 31, 13, 7 and 10 percent less energy than CONV, WRBF, FOOT and RBLA, respectively. Fig. 11b shows that, in CONV, reads and writes (including activation), which are mostly incurred by fetching and eviction as shown in Fig. 10a, consume more than a third of total energy consumption in the DRAM. Caching and eviction occur at the granularity of DRAM page, i.e., 2 KB, which reduces the relative portion of activation energy consumption in CONV compared with in the other methods. On the contrary, in WRBF, activation occupies a significant portion of total

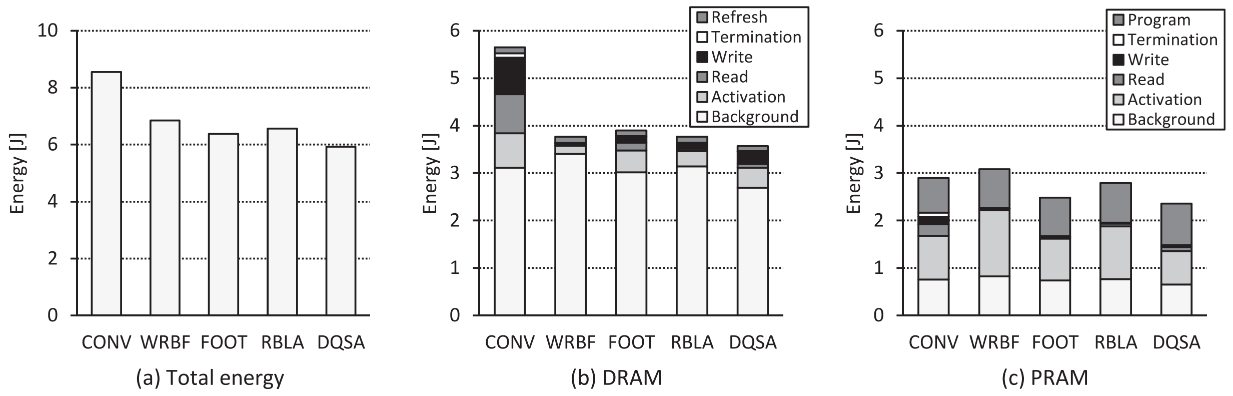


Fig. 11. Energy consumption of (a) total memory, and breakdown of energy consumption of (b) DRAM and (c) PRAM.

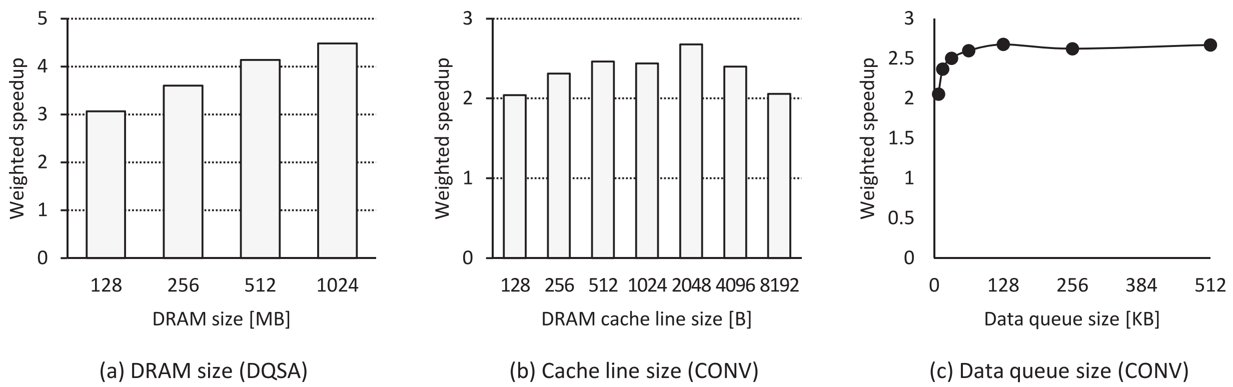


Fig. 12. Sensitivity analysis of (a) DRAM size, (b) DRAM cache line size, and (c) data queue size of memory controller.

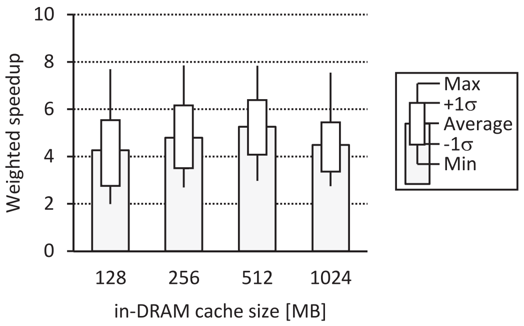


Fig. 13. Feasibility study of DRAM allocation to cache and main memory.

energy consumption while read/write energy consumption is small. It is because caching effects is small in WRBF while PRAM receives a significant amount of direct accesses of small data size, i.e., LLC block size. The other three methods (FOOT, RBLA and ours) show a similar behavior in energy consumption. DQSA gives the lowest energy consumption due to the smallest runtime as shown in Fig. 7a.

5.3 Sensitivity Analyses

In this subsection, we explain how we obtained DRAM size, DRAM cache line size, and data queue size for our experiments. In addition, we give a feasibility study on optimizing the size of in-DRAM cache. Note that different configurations, e.g., single level cell (SLC) PRAM instead of MLC PRAM can require new runs of sensitivity analysis to give a different set of best architectural parameters from the one used in our experiments.

Fig. 12a shows how the performance of DQSA increases as the DRAM size increases. As shown in the figure, the larger DRAM, the better performance is obtained. We set the DRAM size to 1 GB in order to limit the area overhead of DRAM cache tag shown in Table 1. As mentioned in Section 2, it will be interesting to explore the design space of in-DRAM tag in the hybrid main memory sub-system.

The DRAM cache line size can affect the performance of entire system [22], [23]. If the cache line size becomes large, high hit rate can be obtained at a cost of low utilization of DRAM cache. Fig. 12b shows the impact of DRAM cache line size on performance. It shows that the DRAM cache line of 2 KB gives the best performance in our system.

Fig. 12c shows the impact of data queue size on performance. In the figure, as the data queue size increases, the performance improves. However, after it reaches 128 KB, the improvement saturates. Thus, we used the size of 128 KB in our experiments. It allows us to fully exploit the bank parallelism in our memory sub-system consisting of four channels and eight banks per channel.[2]

In our experiments, we set the size of in-DRAM cache to that of the entire DRAM, 1 GB. According to our investigation to be explained below, there are possibilities of further improvements in performance by adjusting the size of in-DRAM cache and allocating the non-cached region of DRAM, i.e., a portion of memory address space to latency-critical programs. Fig. 13 shows the result of feasibility study. As shown in the figure, we vary the size of in-DRAM cache from 128 MB to 1 GB. In the cases that the size of in-DRAM cache is smaller than that of DRAM, 1 GB, we allocate the non-cached region of DRAM to latency critical programs, i.e., low MPKI programs in Table 3. To be specific, in this study, we select two programs which have the lowest MPKI values among eight CPU programs and the footprint of which does not exceed the size of non-cached region of DRAM. The other six CPU programs run on the memory address space of PRAM while utilizing the in-DRAM cache. As the figure shows, in the case that the size of in-DRAM cache is 256 MB (512 MB), we can obtain a similar (better) performance to (than) the case that the entire DRAM is utilized as the in-DRAM cache, 1 GB case in the figure. We think better partitioning (between in-DRAM cache and non-cached DRAM region) and program allocation (to determine which programs to access non-cached DRAM region) can give further improvements in performance, which is left for the future work.

2. Note that, in the worst case, the maximum amount of data transfer between DRAM and PRAM will reach 128 KB (= 4 channels $\times$ 8 banks $\times$ 4 KB for both eviction and insertion in the DRAM cache).

6 Conclusion

In this paper, we proposed a novel architecture of hybrid main memory sub-system where both DRAM and PRAM can be accessed in parallel for higher memory performance and an in-DRAM cache is utilized to exploit the data locality of dirty data and mitigate the long read/write latency of PRAM. In order to resolve the problem of excessive stall time at the data queue of memory controller, we proposed selectively caching data which incur the stall thereby reducing memory access latency and improving fairness. Our experimental results show that our proposed method, DQSA gives by 21 percent higher program performance and 2.1 times improvement in fairness compared with the best of existing methods in a multi-core system which consists of multiple CPUs and GPU.

Acknowledgments

This work was supported by Research Resettlement Fund for the new faculty of Seoul National University and the IT R&D program of MKE/KEIT (No. 10041608, Embedded System Software for New Memory based Smart Devices).

References

[1] U. Kang, H. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *Proc. Memory Forum: A Workshop 41st Int. Symp. Comput. Archit.*, 2014, pp. 1–4.
[2] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, no. 4, pp. 465–479, Jul. 2008.
[3] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proc. IEEE 16th Int. Symp. High Perform. Comput. Archit.*, 2010, pp. 1–11.
[4] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in MLC phase change memory," in *Proc. IEEE 18th Int. Symp. High Perform. Comput. Archit.*, 2012, pp. 1–10.
[5] L. Jiang, Y. Zhang, B. R. Childers, and J. Yang, "FPB: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 1–12.

[6] J. Yue and Y. Zhu, "Accelerating write by exploiting PCM asymmetries," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 282–293.

[7] B. Li, S. Shan, Y. Hu, and X. Li, "Partial-SET: Write speedup of PCM main memory," in *Proc. Des. Autom. Test Eur. Conf.*, article 53, 2014.

[8] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.

[9] S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2011, pp. 1–6.

[10] D. Kim, S. Lee, J. Chung, D. H. Kim, D. H. Woo, S. Yoo, and S. Lee, "Hybrid DRAM/PRAM-based main memory for single-chip CPU/GPU," in *Proc. 49th ACM/EDAC/IEEE Des. Autom. Test Conf.*, 2012, pp. 888–896.

[11] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Maximizing bandwidth utilization of hybrid memory systems," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. TR-CARET-2015-01, 2015.

[12] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked DRAM as part of memory," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 13–24.

[13] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proc. IEEE 30th Int. Conf. Comput. Des.*, 2012, pp. 337–344.

[14] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proc. 40th Int. Symp. Comput. Archit.*, 2013, pp. 404–415.

[15] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetterz, "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 93–102.

[16] B. Nale, R. Ramanujan, M. Swaminathan, and T. Thomas, "Memory channel that supports near memory and far memory access," PCT/US2011/ 054421, Intel Corporation, 2013.

[17] R. Ramanujan, R. Agarwal, and G. Hinton, "Apparatus and method for implementing a multi- level memory hierarchy having different operating modes," US 20130268728 A1, Intel Corporation, 2013.

[18] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Proc. 25th Int. Conf. Comput. Des.*, 2007, pp. 55–62.

[19] S. Franey and M. Lipasti, "Tag tables," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2014, pp. 514–525.

[20] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-offs in architecting DRAM caches outperforming impractical SRAM-Tags with a simple and practical design," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 235–246.

[21] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 454–464.

[22] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 25–37.

[23] H. Chung, B. H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J.-M. Park, Q. Wang, Y.-J. Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K.-W. Lim, W.-R. Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S.-W. Chang, W.-Y. Cho, J.-H. Yoo, and Y.-H. Jun, "A 58 nm 1.8V 1 Gb PRAM with 6.4 MB/s program BW," in *Proc. Int. Solid-State Circuits Conf.*, 2011, pp. 500–502.

[24] T. Lee, H. Park, D. Kim, S. Yoo, and S. Lee, "FPGA-based prototyping systems for emerging memory technologies," in *Proc. IEEE 25th Int. Symp. Rapid Syst. Prototyping*, 2014, pp. 115–120.

[25] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 14–23.

[26] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 383–394.

[27] J. Yun, S. Lee, and S. Yoo, "Dynamic wear leveling for phase-change memories with endurance variations," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, no. 9, pp. 1604–1615, Sep. 2015.

[28] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li, "Wear rate leveling: Lifetime enhancement of PRAM with endurance variation," in *Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf.*, 2011, pp. 972–977.

[29] M. Zhao, L. Jiang, Y. Zhang, and C. J. Xue, "SLC-enabled wear leveling for MLC PCM considering process variation," in *Proc. 51st Annu. Des. Autom. Conf.*, 2014, pp. 1–6.

[30] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 141–152.

[31] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H. H. S. Lee, "SAFER: Stuck-at-fault error recovery for memories," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 115–124.

[32] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *Proc. IEEE 17th Int. Symp. High-Perform. Comput. Archit.*, 2011, pp. 466–477.

[33] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu, W. Liu, Z. Shao, and E. H.-M. Sha, "Building high-performance smartphones via non-volatile memory: The swap approach," in *Proc. 14th Int. Conf. Embedded Softw.*, 2014, article 30.

[34] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng, "A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems," in *Proc. 8th ACM Int. Conf. Embedded Softw.*, 2008, pp. 31–40.

[35] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao, "PCM-FTL: A write-activity-aware NAND flash memory management scheme for PCM-based embedded systems," in *Proc. IEEE 32nd Real-Time Syst. Symp.*, 2011, pp. 357–366.

[36] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 63–74.

[37] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 74–85.

[38] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: The Utah simulated memory module," Univ. Utah, Salt Lake City, UT, USA, Tech. Rep. UUCS-12-002, 2012.

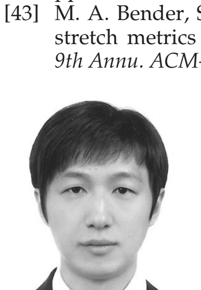
[39] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2007, pp. 3014–3017.

[40] Micron Technology Inc. (2001, May) SDRAM Power Calculator manual [Online]. Available: http://www.micron.com/~/media/Documents/Products/Technical%20Note/DRAM/TN4603.pdf

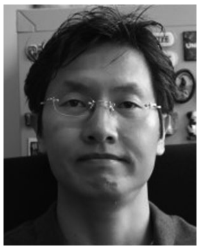
[41] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 412–423.

[42] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proc. 9th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2000, pp. 234–244.

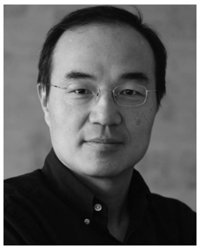
[43] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *Proc. 9th Annu. ACM-SIAM Symp. Discr. Algorithms*, 1998, pp. 270–279.



Dongki Kim received the BS degree in electrical engineering from Kyungpook National University, Daegu, Korea, in 2009 and the MS degree in electrical engineering from Pohang University of Science and Technology (POSTECH), Pohang, Korea, in 2011. He is currently working toward the PhD degree. His current research interests include memory access scheduling in hybrid main memory consists of DRAM and emerging nonvolatile memories and high-performance NAND flash memory-based solid state disks.



Sungjoo Yoo (M'00) received the PhD degree from Seoul National University, Seoul, Korea, in 2000. He is currently an associate professor in the Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea. He was a researcher with TIMA Laboratory, Grenoble, France, from 2000 to 2004. He was also with Samsung System LSI from 2004 to 2008. From 2008 to 2015, he was an associated professor at Pohang University of Science and Technology (POSTECH). In 2015, he joined the Department of Computer Science and Engineering, Seoul National University. His research interests include new memory-based computer architecture and deep learning for mobile devices. He is a member of the IEEE.



Sunggu Lee (M'88) received the BSEE degree, with Highest Distinction, from the University of Kansas, Lawrence, KS, in 1985, and the MSE and PhD degrees from the University of Michigan, Ann Arbor, MI, in 1987 and 1990, respectively. He was an assistant professor with the Department of Electrical Engineering, University of Delaware, Newark, DE. From 1997 to 1998, he was a visiting scientist with the IBM T.J. Watson Research Center, Yorktown Heights, NY, and a visiting researcher with the DREAM Laboratory, University of California at Irvine, Irvine, CA, from 2005 to 2006. He is currently a professor with the Department of Electrical Engineering, Pohang University of Science and Technology, Pohang, Korea. His current research interests include wireless sensor networks, cloud computing, real-time computing, parallel computing, and fault-tolerant computing. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.