Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures

Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He[§], Long Zheng, Rentong Guo
Services Computing Technology and System Lab/Big Data Technology and System Lab/Cluster and Grid Computing Lab
School of Computing Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[§]School of Computing, National University of Singapore, Singapore, 117418
{hkliu,yujiechen,xfliao,hjin,longzh,rtguo}@hust.edu.cn,hebs@comp.nus.edu.sg

ABSTRACT

Non-Volatile Memory (*NVM*) has recently emerged for its non-volatility, high density and energy efficiency. Hybrid memory systems composed of DRAM and NVM have the best of both worlds, because NVM can offer larger capacity and have near-zero standby power consumption while DRAM provides higher performance. Many studies have advocated to use DRAM as a cache to NVM. However, it is still an open problem on how to manage the DRAM cache effectively and efficiently. In this paper, we propose a novel *Hardware/Software Cooperative Caching* (*HSCC*) mechanism that organizes NVM and DRAM in a flat address space while logically supporting a cache/memory hierarchy. HSCC maintains the NVM-to-DRAM address mapping and tracks the access counts of NVM pages through a moderate extension to page tables and TLBs. It significantly simplifies the hardware design and offers several optimization opportunities for cache management in software layers. We thus propose utility-based cache filtering policies to improve the efficiency of DRAM cache. Experimental results show that HSCC improves system performance by up to 9.6X (77.2% on average) and reduces energy consumption by 34.3% on average, compared to a hardware-assisted DRAM/NVM memory system. HSCC also presents 15.4% and 14.5% performance improvement against a flat-addressable memory architecture and a *Row Buffer Locality Aware* (*RBLA*) caching policy for hybrid memories, respectively.

CCS CONCEPTS

•**Computer systems organization** →*Architectures;*

KEYWORDS

Non-Volatile Memory (NVM), Hybird memory, Caching

ACM Reference format:
Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He[§], Long Zheng, Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079089

ICS '17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5020-4/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079089

1 INTRODUCTION

In-memory computing is becoming increasingly popular for data-intensive applications in the big data era. Traditional big memory systems [1, 2] using Dynamic Random Access Memory (DRAM) are facing severe scalability challenges in terms of power and capacity [3]. Emerging Non-Volatile Memory (NVM) technologies, such as PCM, STT-RAM and ReRAM generally offer higher memory density, much lower cost-per-bit and standby power consumption than DRAM. NVM is considered as a competitive alternative to DRAM. Despite the advantages, NVM exhibits higher access latency and write power consumption compared to DRAM. Moreover, NVM has limited write endurance. These disadvantages make it hard to be a direct substitute for DRAM. Thus, hybrid memory architectures composed of both DRAM and NVM are more promising for building high-performance, large-scale, and energy-efficient main memory systems [4, 5].

An immediate question in the design of a DRAM/NVM hybrid memory system is how to fully exploit the advantages of DRAM and NVM technologies while avoiding their disadvantages as many as possible. There have been many proposals that organize DRAM and NVM in a flat address space and uniformly use them as main memory [4, 6, 7]. Those memory architectures need sophisticated page migration schemes to overcome the disadvantages of NVM. Some other studies architect DRAM as a large hardware-managed cache to NVM [5, 8–10]. The DRAM is usually organized as N-way set-associative cache which is managed by hardware (the memory controller) and is invisible to software [5]. This hierarchical memory architecture can benefit from the high performance and endurance of DRAM, while enlarging the capacity of main memory with NVM.

Although the hierarchical DRAM/NVM memory architecture is promising for high performance [5, 11], the implementation complexity of hardware-managed caches is quite significant for the following reasons: (1) Managing metadata (i.e., tags) that are used to track the data cached in DRAM can incur prohibitively high overhead. Fine-grained data tracking (e.g., a 64 Byte cacheline) results in high storage overhead, for example, a typical cache design requires 96 MB of tag storage to track 1 GB DRAM cache in 64 bytes data blocks [12]. Storing those tags with on-chip SRAM is impractical if DRAM capacity becomes very large. However, co-locating the tags and data in DRAM needs additional tags lookup in DRAM (similar to on-chip cache lookup) and thus increases memory access latency. Although coarse-grained data tracking (e.g., 4 KB page granularity) mitigates tag storage overhead, it wastes the capacity and bandwidth of DRAM cache if most data in the large blocks are not touched, causing inefficient DRAM cache evictions.

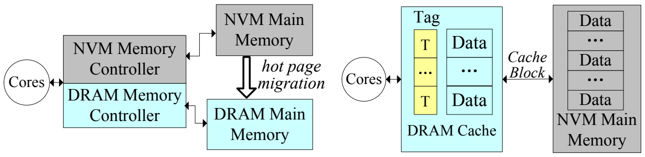
(2) Hardware-managed caches usually utilize on-demand fetching or locality-based prefetching polices for simplicity. However, they are not always efficient in such hybrid memory systems because the costly page migrations between DRAM and NVM can offset the benefit of data caching. (3) The CPU design also needs to be aware of the DRAM cache implementation. For example, the memory management unit (MMU) of general CPUs is mainly designed for addressing DRAM as main memory, not off-chip cache in hybrid memory architectures. To this end, a hardware-managed DRAM cache makes the cache organization and management inflexible.

In this paper, we explore a different direction to manage DRAM cache in a DRAM/NVM hybrid memory system. We propose a *Hardware/Software Cooperative Caching* (HSCC) mechanism for hybrid memory architectures to resolve the constraints of hardware-based approaches. HSCC organizes DRAM and NVM in a single (flat) physical address space while logically supporting a hierarchical memory architecture. HSCC simplifies the hardware design by pushing the burden of DRAM cache management to the software layers. This approach improves the flexibility of DRAM cache management and can leverage several existing memory management optimizations offered by Operating Systems (OSes). HSCC maintains the physical address remapping from NVM to DRAM through a moderate extension of page table and translation lookaside buffer (TLB). This design also enables lightweight memory access monitoring and a DRAM cache bypassing mechanism.

To utilize the DRAM cache effectively and efficiently, we propose utility-based DRAM cache filtering policies to mitigate potential cache thrashing. The basic idea is to place frequently accessed (hot) pages in DRAM cache while keeping infrequently accessed (cold) pages in NVM. We track hotness of NVM pages through the extended TLB and fetch those NVM pages whose access counts become larger than a given threshold into DRAM cache. To adapt to diversifying and dynamic memory access patterns, we further develop a dynamic threshold adjustment algorithm based on hotness of NVM pages and DRAM cache utilization.

In summary, we make the following contributions.

- We propose a *Hardware/Software Cooperative Caching* (HSCC) mechanism for DRAM/NVM hybrid memory architectures, which organizes all memories in a flat address space while logically utilizing DRAM as a cache to NVM. We extend page tables and TLBs to maintain the NVM-to-DRAM address mappings and to track the access counts of NVM pages. Our tagless design significantly simplifies the hardware and enables flexible DRAM cache management in the software layer.
- We propose utility-based cache filtering policies to selectively cache hot pages in DRAM and keep cold pages in NVM. These schemes can avoid DRAM cache thrashing and significantly reduce the cost of page migrations.
- We model HSCC with zsim [13] and NVMain [14] simulators and compare it with several state-of-the-art policies using a wide range of workloads. Experimental results show that HSCC can improve application performance by up to 9.6X (77.2% on average) and reduce about 34.3% energy consumption compared to the hardware-managed DRAM caching scheme [5]. HSCC also demonstrates 15.4% and 14.5% higher performance than a flat-addressable memory architecture and the row buffer locality aware caching policy [10], respectively.



(a) Flat-addressable hybrid memory architecture (b) Hierarchical hybrid memory architecture

Figure 1: Typical hybrid memory system architectures

The remainder of this paper is organized as follows. Section 2 introduces the background and motivates a hardware/software cooperative caching for DRAM/NVM hybrid memory architectures. Section 3 gives the detailed design of HSCC. Section 4 presents utility-based DRAM cache filtering schemes. Experimental results are presented in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2 BACKGROUND AND MOTIVATION

We first review two typical kinds of DRAM/NVM hybrid memory architectures. Next, we experimentally investigate memory access patterns of several representative applications and demonstrate that current on-demand cache fetching policies are ineffective for a DRAM/NVM hierarchical memory architecture.

2.1 Hybrid Memory Architectures

Flat-addressable Hybrid Memory Systems. A number of hybrid memory systems [4, 6, 7, 15] organize both DRAM and NVM as main memory in a flat address space. Such hybrid memory architectures need to overcome the disadvantages of NVM by migrating frequently accessed (hot) NVM pages to DRAM, as shown in Figure 1(a). One page migration can induce multiple costly page read/write operations, especially when the write latency on NVM is much higher than that of DRAM. On the other hand, flat-addressable hybrid memory systems require additional hardware in the memory controller to detect hot pages.

Hierarchical Hybrid Memory Systems. A large body of hybrid DRAM/NVM memory systems use DRAM as a cache to NVM [5, 8–11]. Figure 1(b) shows a DRAM/NVM hybrid memory architecture with a cache/memory hierarchy. Since DRAM is excluded from system memory address space, it is transparent to OSes and is completely managed by hardware. A moderate storage space is required to store the metadata (i.e., tags) of data blocks in DRAM cache. Moreover, a hardware lookup circuit is required to determine whether requested data hits in DRAM cache. Hence, one last level cache (LLC) miss leads to two memory accesses (one for accessing metadata, and another for accessing the requested data). In addition, hardware-managed hierarchical memories usually adopt on-demand cache fetching policies [5, 8], and thus every data block missed in on-chip cache should be fetched into DRAM cache from NVM and then is loaded to CPU.

Table 1 compares the aforementioned hybrid memory architectures with HSCC. HSCC addresses the limitation of these two architectures while enabling new desirable properties. To manage DRAM cache more flexibly and efficiently, HSCC pushes DRAM cache management to the OS layer and manages the cache in a page granularity. Furthermore, HSCC supports DRAM cache bypass, and thus cold (infrequently accessed) NVM pages can be

Table 1: The Comparison between HSCC and Typical Hybrid Memory Architectures

Hybrid memory architectures	DRAM organization	DRAM bypass	Performance optimization	DRAM visibility
Hierarchical [5, 8, 10]	N-ways set assoc. cache	No	On-demand fetching	Hardware
Flat [4, 6]	Page-sized main memory	/	Hot page migration	OS
HSCC	Page-sized cache	Support	Utility based fetching	OS

directly accessed without involving in the DRAM cache. HSCC exploits utility-based data fetching policy to improve DRAM cache efficiency.

2.2 Memory Access Pattern of Applications

To characterize memory access pattern of applications, we have conducted experiments to profile applications' working sets and hot (frequently accessed) page statistics in an interval of 10^8 cycles, as shown in Table 2. The representative applications are selected from three typical benchmark suites SPEC CPU 2006 [16], Parsec [17] and Problem Based Benchmark Suite (PBBS) [18]. These workloads cover a wide range of memory access patterns. All experiments are conducted in a simulated platform, as presented in Section 5.1.

Similar to a previous study (CHOP [19]), we define an application's *hot pages* as the most frequently accessed pages that contribute to 70% of its total page access number in our experiments. For each application, Table 2 shows the minimum number of accesses to its hot pages, the average number of accesses to all pages, and the memory footprint in every 10^8 cycles. The hot page percentage is defined as the ratio of hot page counts to the total number of accessed pages. We have the following observations.

Observation 1: the distribution of hot pages is very sparse for many applications. We find that many applications show a relatively small percentage of hot pages, such as mcf, astar, and Canneal. This implies only a small portion of pages are accessed frequently for these applications, hence it is more beneficial to fetch the hot pages rather than every requested pages from NVM to DRAM in a DRAM/NVM hierarchical memory system.

Observation 2: the page hotness (i.e., access frequency) varies widely between different applications. Applications with larger working sets usually show relatively lower page access frequency, such as astar (14) and Canneal (8). In contrast, applications with smaller working sets generally exhibit higher page access frequency and a larger percentage of hot pages, such as Blackscholes, MSF, KNN, and ISORT. This implies the criteria for quantifying the "hotness" should dynamically adapt to different applications.

Observation 3: applications' working sets vary in different phases of execution. We find that most applications exhibit a relatively large memory footprint in their whole lifetime, while the working set in a short time slot is rather small. For example, KNN only touches less than 10 MB memory in each sampled time slot while its total memory footprint can reach 1.5 GB. This implies that the working set of an application is changing over time, and thus hot pages may become cold during the execution of application.

The above observations motivate us to develop a dynamic cache filtering policy for hierarchical DRAM/NVM memory systems, as described in Section 4.

Table 2: Hot Page Access Statistics for Typical Applications

Application	Page access statistics (every 10^8 cycles)				Total memory footprint
	Hot page min# access	Ave.# access	Working set	Hot page percent	
mcf	70	32	43.1 MB	22.6%	1.6 GB
bzip2	132	130	4.7 MB	47.5%	581 MB
astar	14	11	96.0 MB	22.2%	382 MB
omnetpp	15	10	60.1 MB	28.4%	174 MB
sphinx3	34	26	9.7 MB	25.1%	48 MB
soplex	56	50	21.6 MB	50.9%	1.49 GB
milc	16	15	57.5 MB	44%	575 MB
Canneal	8	6	147.5 MB	21.5%	940 MB
Blackscholes	64	63	8.4 MB	61.1%	613 MB
MSF	68	79	8.1 MB	60.0%	2.5 GB
SetCover	37	44	30.5 MB	52%	2.4 GB
KNN	64	83	9.2 MB	59%	1.5 GB
DICT	65	67	29.6 MB	47.5%	623 MB
ISORT	67	71	5.7 MB	60%	526 MB
NBODY	34	35	23.7 MB	34.0%	311 MB
BFS	39	42	83.7 MB	38.8%	3.2 GB

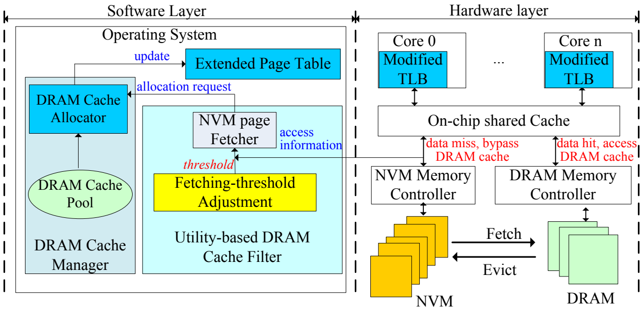


Figure 2: Architecture of HSCC

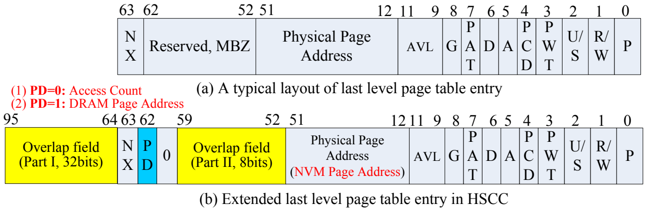
3 DESIGN

In this section, we first give an overview of HSCC from the hardware and software layers. We then introduce our lightweight hardware/software co-design schemes.

3.1 Architecture Overview

Figure 2 depicts the architecture of HSCC. In the software layer, HSCC extends the last level page table entries to maintain address mappings between DRAM pages and NVM pages, and adds a counter to record the number of page accesses. Moreover, an utility-based DRAM cache filter is implemented to improve the effectiveness and efficiency of page caching. In the hardware layer, we modify TLBs to distinguish virtual-to-DRAM and virtual-to-NVM address mappings, and add hardware counters to track access counts of NVM pages. With these extensions, HSCC supports fast address translations of virtual-to-NVM as well as NVM-to-DRAM, and logically supports a cache/memory hierarchy for DRAM/NVM memories. Because DRAM is inherently managed by most OSes, we push DRAM cache management to the software layer to fully exploit the existing memory management optimizations offered by OSes.

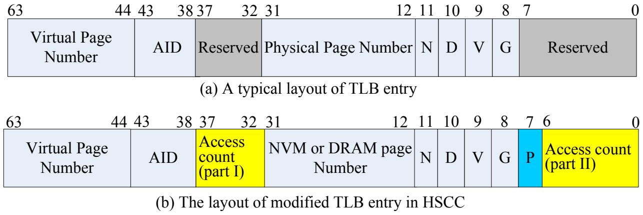
As described in Section 2.2, on-demand data fetching from NVM to DRAM is not beneficial because most applications' hot pages are sparse due to multiple layers of caching on CPU. Hence, we propose a DRAM cache bypassing scheme based on the TLB and page table extensions. HSCC can directly determine whether the requested data is stored in DRAM or NVM with a flag in modified TLB or page table. To filter cold pages in the NVM, HSCC detects hot pages according to the page access counters maintained in the TLBs. NVM pages whose access counts exceed a certain threshold



(a) A typical layout of last level page table entry

(b) Extended last level page table entry in HSCC

Figure 3: Extended last level page table entry in HSCC



(a) A typical layout of TLB entry

(b) The layout of modified TLB entry in HSCC

Figure 4: Modified TLB entry in HSCC

(dynamically adjusted at runtime) are fetched from NVM to DRAM cache. As the *threshold* has a significant impact on page hotness, we propose dynamic *fetching_threshold* adjustment algorithms to improve the benefit of data fetching from NVM (Section 4.2).

3.2 Page Table and TLB Extension

HSCC organizes DRAM and NVM in a single flat address space while managing the DRAM as a cache to the NVM. One immediate question is how to maintain the NVM-to-DRAM address mapping to enable a logical caching architecture with minimal hardware support. In addition, HSCC requires tracking the frequency of page access to support selective page caching. In this section, we present our hardware/software co-design to address these issues, including lightweight page table and TLB extensions, which are widely exploited in the design of memory architectures [20, 21].

Our caching scheme needs to manage the physical address mappings between NVM pages and DRAM pages. A typical approach is to use a dedicated table to record the NVM-to-DRAM address mappings. However, this approach increases memory addressing latency due to the additional memory access to the mapping table and lookup cost. Considering that the last level page table records the mappings from virtual pages to physical NVM pages, we extend the page table by adding a new field to record the corresponding DRAM page number of the cached NVM page, as shown in Figure 3. In this way, HSCC can efficiently obtain address mappings of virtual-to-NVM and NVM-to-DRAM through only one memory access. This design scatters the NVM-to-DRAM address mappings into processes' Page Table Entries (PTEs) to avoid costly lookup overhead in a dedicated mapping table.

In addition, HSCC requires tracking the access frequency of NVM pages to support cache filtering. As TLB is in the critical path of memory access, we add a counter in each TLB entry to track the number of page accesses, along with corresponding extension of the PTEs at the software layer.

Figure 3 shows the data structure of extended last level PTEs in HSCC. A *PD* flag and a *overlap field* are added by using some reserved bits and extended bits. The *PD* flag reflects whether a NVM page has been fetched into DRAM cache. To take full advantage of the limited space of PTEs, the *overlap field* plays different roles depending on the value of *PD*. When *PD* is equal to 0 (meaning the page is still in NVM), the *overlap field* serves as a counter to record the number of page accesses. In contrast, when *PD* is equal to 1 (meaning the NVM page has been loaded into DRAM cache), the *overlap field* is used to store the physical address of the corresponding DRAM page.

Figure 4 depicts the logical data structure of modified TLB. We assume the CPU implements a single L1 TLB. Similar to the extended PTE, a *P* flag is added in each TLB entry to indicate whether a NVM page has been fetched into DRAM cache. If *P* equals to 1, the TLB entry maintains virtual-to-DRAM page mapping. Otherwise, the TLB entry maintains virtual-to-NVM page mapping. A page access counter is also added to reflect the hotness of a NVM page. HSCC increases the counter value when the requested data is not hit in LLC. When a TLB entry is evicted, HSCC writes back only one half of the counter value to the count in its PTE. For some architectures, the TLB entries have enough free bits to store the access count, and thus the hardware counters can be directly placed in the TLB entries by using the reserved bits, as shown in Figure 4.

3.3 DRAM Cache Management

Memory addressing and DRAM cache bypassing. Figure 5 depicts typical process of memory accesses in HSCC. When a machine is just booted up, the PTEs and TLBs only record virtual-to-NVM page mappings. On a TLB miss, a page table walk is performed for address translation, and then a new TLB entry is installed to set up the mapping from virtual page number to NVM page number. Also, the *"P"* flag is set to 0 and the page access counter increases. On a TLB hit, the *P* flag and *paddr* (ppn of NVM or DRAM pages) are delivered through the memory hierarchy till the required data is obtained. If the on-chip LLC misses, HSCC needs to check the value of flag *P* to determine whether the requested data is in DRAM cache or NVM. If $P = 1$, CPU fetches data from DRAM cache rather than NVM according to the virtual-to-DRAM mappings maintained in TLBs. Otherwise ($P = 0$), CPU directly fetches data from NVM main memory. For cold pages, this DRAM bypassing mechanism can reduce memory access latency by avoiding the cost of on-demand page fetching and data lookup in DRAM cache.

DRAM cache allocation and replacement. In HSCC, DRAM and NVM are managed in the granularity of page (4 KB), which is consistent with many OSes. To logically use DRAM as a cache of NVM, HSCC manages DRAM and NVM pages in separated address space. HSCC exploits buddy allocator of existing OSes for DRAM cache allocation. As the capacity of DRAM cache is much larger than LLC cache, traditional LRU cache replacement policy can yield significant performance overhead when it is implemented in software layer. HSCC maintains three lists to manage the DRAM cache pool, as shown in Figure 5. The clean and dirty lists maintain clean and dirty DRAM pages, respectively. Pages maintained in the victim list can be replaced when DRAM cache is fully utilized. Since NVM write is very expensive, we preferentially select clean and old pages for DRAM cache replacement. When the number of free pages and victim pages is less than a given threshold, some page descriptors from the head of clean list are moved to the victim list. If the clean list is empty, HSCC has to evict dirty pages. However, a

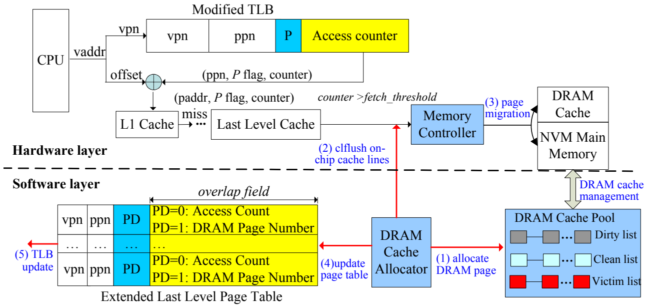


Figure 5: Detailed memory accesses in HSCC

dirty page in the victim list is reclaimed and written back to NVM only when another NVM page is about to replace this page.

Page migration. When the access count of a NVM page exceeds a given threshold, the page is considered as a hot page and should be fetched into DRAM cache. As shown in Figure 5, the page migration includes five steps: (1) allocate a DRAM page from the DRAM cache pool, (2) flush the on-chip cache lines corresponding to the NVM page, (3) copy the NVM page to DRAM cache, (4) update page tables, and (5) update TLB entries. However, page migration may result in data inconsistency problems. We discuss the details in the following.

On-chip cache consistency. A page may be referenced by many cache lines in on-chip cache. As some architectures adopt write-back cache techniques for low latency and high throughput, the dirty cache blocks are not written to main memory until they are evicted from the on-chip cache. To guarantee data consistency, a series of *clflush* operations are imposed on the dirty cache lines that corresponds to the NVM page. As a result, at any level of the cache hierarchy, the dirty cache lines are written to main memory before the corresponding page is migrated.

Page sharing. A page may be shared by several processes and each process has its own page tables. When a shared page is migrated, the OS needs to traverse other processes' page tables to update the virtual-to-physical address mappings. This is extremely costly for large programs due to a large number of PTEs. To reduce the cost of PTE update after page migration, we implement inverted page table technique [22], which uses a global inverted page table to map physical-to-virtual addresses. It is extremely useful for page migration because inverted page table can easily find the virtual addresses of other processes corresponding to the old physical page frame and update their PTEs with the new physical address.

TLB coherence. When a page has been migrated, the PTE and TLB should be updated correspondingly. Because a single page may be referenced by multiple cores, this may lead to TLB incoherence problem. If the stale mappings in other cores' TLBs are not synchronized, they may observe incorrect values. A simple solution is to exploit a TLB shootdown mechanism [21], which invalidates other cores' TLB entries when a processor changes a virtual-to-physical address mapping. The synchronization cost increases with the number of cores involved in the TLB shootdown [21]. TLB shootdown would increase TLB miss rate and has side effect on system performance. In HSCC, we improve TLB shootdown mechanism by updating the corresponding TLB entries in other cores with new virtual-to-physical address mappings rather than simply invalidating all the TLB entries. Since each core maintains a page access counter for a TLB entry, when a TLB entry is evicted, the counter value in the evicted TLB entry is written back to its corresponding PTE and added to the in-memory count.

4 UTILITY-BASED CACHE FILTERING

We first model the benefit of caching NVM pages in DRAM, and then present the dynamic cache filtering policy.

4.1 Benefit of Caching NVM Pages

Caching NVM pages in DRAM can improve memory access performance, however, page migration also leads to increased access latency of requested data. We should make a trade-off between the gained benefit of caching and the page migration cost. Let C_{read} and C_{write} denote total counts of reads and writes to a page in a time slot (10^8 cycles in our experiments), respectively. Let t_{nr} and t_{dr} denote the read latency of NVM and DRAM, respectively. Let t_{nw} and t_{dw} denote the write latency of NVM and DRAM, respectively. Let T_{fetch} represent cycles spent in fetching NVM pages to DRAM cache, and $T_{eviction}$ denote cycles spent in evicting dirty pages from DRAM cache to NVM in each time slot. As shown in Equation 1, the benefit of caching NVM pages is calculated as the total cycles saved by accessing data from DRAM cache against NVM minus total cycles spent in fetching and evicting pages.

$$\begin{aligned} caching\_benefit = (t_{nr} - t_{dr})C_{read} + (t_{nw} - t_{dw})C_{write} \\ - T_{fetch} - T_{eviction} \end{aligned} \tag{1}$$

For a given *fetching_threshold*, a positive value of *caching_benefit* implies that caching NVM pages in DRAM saves more data access time than the cost of page migrations.

4.2 Dynamic Threshold for Cache Filtering

We propose a filter-based cache fetching policy to improve the efficiency of DRAM cache. The basic idea is to place hot pages in DRAM cache while keeping cold pages in NVM. The filtering technique relies on a counter maintained in the TLB. The counter reflects the hotness of a NVM page. At first, the counter value is initialized to zero for any newly-allocated page in NVM. On each read/write to the page, the counter increases to enhance the page hotness according to the asymmetric read/write latencies of NVM. More specifically, assume the counter is increased by S_r when a NVM page is read, and then the counter is increased by $S_r * (t_{nw}/t_{nr})$ when the NVM page is written, where t_{nw} and t_{nr} denote NVM write and read latencies, respectively. We note that the page access counts are reset in each time slot (10^8 cycles in HSCC) to guarantee that the NVM pages fetched to DRAM are hot pages in the most recent period.

Once the counter value becomes larger than a given threshold, *fetching_threshold*, the corresponding NVM page is fetched to DRAM cache. DRAM cache allocator is invoked to allocate a free page, and then the NVM page is copied to this DRAM page. OS updates the extended last level PTE by setting *PD* flag to 1 and filling *overlap field* with physical address of the allocated DRAM page. Meanwhile, the TLB entry of the NVM page should also be updated accordingly.

We observe that *fetching_threshold* affects the effectiveness and efficiency of cache filtering significantly. As many applications

Algorithm 1 HSCC-Dyn: Fetching_threshold Adjustment

```
Procedure: Fetch_threshold_adjust (DRAM_utilization, ...)
$\Delta hotness_{dram} = current\_hotness_{dram} - previous\_hotness_{dram}$
if DRAM_utilization < DRAM_utilization_threshold then
    Climbing($\Delta hotness_{dram}$)
else
    if caching_benefit < 0 then        /*throttle page fetching*/
        if DRAM pages have not been evicted then
            fetching_threshold = 2 * fetching_threshold
        else
            assign the fetching_threshold an even larger value than 2 * fetching_threshold
    else
        Climbing($\Delta hotness_{dram}$)
Procedure: Climbing($\Delta hotness_{dram}$)
if $\Delta hotness_{dram} > 0$ then        /*DRAM hotness is increasing.*/
    if fetching_threshold increases in previous period then
        fetching_threshold++
    else
        fetching_threshold−−
else
    if fetching_threshold increases in previous period then
        fetching_threshold−−
    else
        fetching_threshold++
```

show diverse memory access patterns in different phases of execution, the hotness of pages may change in different time slots. We should adjust the *fetching_threshold* dynamically to fetch the hottest pages from NVM memory to DRAM cache as many as possible. However, since the capacity of DRAM cache is limited, indiscriminate NVM page fetching can lead to cache thrashing. Evicting a page from DRAM cache usually results in data written to NVM, which is even more costly than fetching pages from NVM, given that the overhead due to TLB and page table updates is not trivial. When the DRAM cache is under high pressure, it is more likely to evict pages from DRAM cache. To mitigate cache thrashing and adapt to varying page hotness, we propose dynamic threshold adjustment algorithm *HSCC-Dyn*, which adjusts *fetching_threshold* periodically according to memory utilization of DRAM cache and DRAM hotness at runtime, as described in Algorithm 1. Here, DRAM utilization denotes the percentage of pages in DRAM used to cache NVM pages. Because the read and write latencies of DRAM are symmetric, *DRAM hotness* is defined as average access counts of all DRAM pages in a time window, denoted by $hotness_{dram}$. For example, if D pages in DRAM are accessed for total N times in a time slot, $hotness_{dram}$ becomes N/D.

In the beginning, the memory utilization of DRAM cache is lower than a given threshold *DRAM_utilization_threshold*, and our goal is to adjust the *fetching_threshold* to cache more NVM pages in DRAM. We adopt hill climbing algorithm to adjust *fetching_threshold* at the end of each time slot (10^8 cycles in our experiments). Hill climbing algorithm is an efficient method to search local optimal solutions with low computational complexity. If the hotness of DRAM cache in current period is higher than that in previous period, implying that the adjustment is effective, *HSCC-Dyn* will take the same action to adjust *fetching_threshold*. In contrast, if the action in the previous period reduces the hotness of DRAM cache, an opposite *fetching_threshold* adjusting action is taken.

When the memory utilization of DRAM cache is higher than the given *DRAM_utilization_threshold*, meaning that high memory pressure may lead to more page evictions. We should increase *fetching_threshold* to only fetch hotter NVM pages into DRAM cache. If *caching_benefit* is less than zero, *HSCC-Dyn* doubles the current *fetching_threshold* value to throttle page fetching. If page evictions have begun, *HSCC-Dyn* performs cache replacement more carefully by increasing *fetching_threshold* to an even larger value.

Table 3: Detailed System Configuration of HSCC

Core	8 cores, 3.2 GHz, in-order
TLB	private 256 entries per core, split D/I, 1-cycle latency
L1 Cache	private 64 KB per core, 4-way, split D/I, 3-cycle latency
L2 Cache	private 256 KB per core, 8-way, 10-cycle latency
L3 Cache	shared 8 MB, 16-way, 34-cycle latency
DRAM Cache	1 GB: 1 channel, 1 rank, 8 banks, 32768 rows, 64 cols, Bandwidth: 10.7 GB/Sec, FR-FCFS request scheduling, Timing (tCAS-tRCD-tRP-tRAS) : 7-7-7-18 (cycles), 13.5 ns read latency, 28.5 ns write latency
PCM (Main Memory)	32 GB: 4 channels, 8 ranks, 8 banks/rank, 65536 rows, 32 cols, Bandwidth: 10.7 GB/Sec, FR-FCFS request scheduling, Timing (tCAS-tRCD-tRP-tRAS): 9-37-100-53 (cycles), 19.5ns read latency, 171 ns write latency
Power/Energy consumption	
DRAM	Voltage: 1.5V, Standby: 77 mA, Refresh: 160 mA, Precharge: 37 mA; Read and write on row buffer hit: 120 mA and 125 mA; Read and write on row buffer miss: 237 mA and 242 mA
PCM	Read/write on row buffer hit: 1.616 pJ/bit; Read and write on row buffer miss: 81.2 pJ/bit and 1684.8 pJ/bit

5 EVALUATION

5.1 Experimental Methodology

Implementation. We implement HSCC with zsim [13] and NVMain [14] simulators. Zsim is a fast x86-64 multi-core simulator. It exploits Intel Pin toolkit to collect traces of memory accesses for processes, and replays the traces in the zsim simulator. The simulator models multiple cores, hierarchical on-chip caches, detailed cache coherence protocol, on-chip/off-chip interconnect, and physical memories. NVMain is a cycle-accurate memory simulator, it models components of DRAM and NVMs, and memory hierarchy in detail. In our experimental setup, NVMain is used to simulate off-chip memories including PCM and DRAM, each has an individual memory controller and adopts channel-interleaving address decoding to exploit memory-level parallelism. Moreover, we have added TLB and memory management modules (such as buddy allocator, page tables) into zsim by referring to x86 architecture and Linux OS because the original zsim does not support OS-level simulations. More details can be found in our open-source code [23].

Experimental setting. Detailed configuration of our experiments is depicted in Table 3. As PCM is one of widely studied NVMs, we choose PCM as main memory in our experimental setup. Both DRAM and PCM are managed in a page granularity (4 KB). Timing and energy parameters of PCM are referred to [11]. The hybrid memory system is configured with 32 GB NVM and 1 GB DRAM. As the memory footprints of our selected applications range from 48 MB to 3.2G, such settings allow us to evaluate the effectiveness of HSCC under both high and low memory pressure conditions.

Alternative policies. We compare HSCC-Dyn with the following memory systems. (1) **Hardware-managed DRAM Cache (HDRC)** [5]. HDRC manages the DRAM as a 16-way set associative cache by hardware, and adopts an on-demand cache fetching policy. We implement and model HDRC in the same testbed [23]. The configuration of HDRC is the same as HSCC except the organization of DRAM cache. (2) **Flat-addressable hybrid memories (FLAT)**. NVM and DRAM are organized in a flat address space and managed by OS. Data is uniformly distributed in DRAM and NVM according to the ratio of their capacity. (3) **Row Buffer Locality-aware Caching for hybrid memories (RBLA)** [10], NVM rows

Table 4: Workloads for Evaluation

Workloads	Applications
SPEC CPU2006	mcf, bzip2, astar, omnetpp, sphinx3, leslie3d, soplex, milc
Parsec	Canneal, Blackscholes (B.S.)
PBBS	MSF, SetCover, KNN, DICT, ISORT, NBODY, BFS
mix1	milc+leslie3d+omnetpp+sphinx3
mix2	milc+leslie3d+soplex+sphinx3
mix3	mcf+leslie3d+sphinx3+SetCover
mix4	mcf+soplex+sphinx3+leslie3d+SetCover+BFS+NBODY+ISORT
mix5	mcf+soplex+milc+bzip2+SetCover+BFS+KNN+MSF

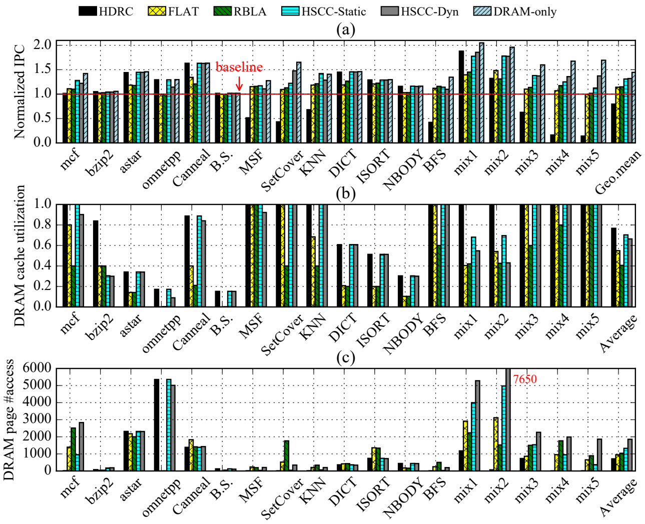


Figure 6: (a) IPC normalized to the baseline system, (b) DRAM memory utilization, and (c) average access frequency of DRAM pages

with high row buffer miss rates are cached in DRAM. (4) **HSCC-Static.** For comparison, we also evaluate HSCC with static *fetching_threshold*, which is set empirically according to offline profiling. (5) **PCM-only** and **DRAM-only**. We use a system with only 32 GB PCM as the baseline, and a system with only 32 GB DRAM as the upper bound of performance.

Benchmarks. We evaluate a number of workloads with different memory access patterns from SPEC CPU 2006 [16], Parsec [17], and Problem Based Benchmarks Suite (PBBS) [18]. Mcf, bzip2, astar, omnetpp, sphinx3, leslie3d, soplex, and milc are selected from SPEC CPU2006. Canneal and Blackscholes are multi-thread applications selected from Parsec. BFS, MSF, SetCover, KNN, DICT, ISORT, and NBODY are chosen from PBBS. BFS and MSF all solve graph problems. SetCover, KNN, and NBODY are computational biological problems. ISORT and DICT are integer sorting and dictionary matching algorithm, respectively. Detailed memory footprints of these applications are shown in Table 2. In addition, we evaluate five multi-programmed workloads, as shown in Table 4.

5.2 Performance Studies

Figure 6(a) shows the Instructions Per Cycle (IPC) of each application normalized to the baseline system (32 GB PCM-only). For all applications, HSCC-Dyn achieves 77.2%, 33%, 14.5%, and 13.8% performance improvement on average compared to HDRC, the baseline, FLAT, and RBLA, respectively. The performance gap between HSCC-Dyn and the upper bound (DRAM-only) is only 8.0% on average. We find that MSF, SetCover, KNN, BFS, mix4, and mix5 suffer significant performance degradation when running in HDRC.

Table 5: Percentages of Page Evicted in HSCC-Dyn and HDRC for Applications with Large Memory Footprint

Application	page eviction in HSCC-Dyn			page eviction in HDRC		
	clean	dirty	total	clean	dirty	total
mcf	0	0	0	99.77%	0.096%	99.87%
MSF	0	0	0	99.40%	0.15%	99.55%
SetCover	59.89%	17.58%	77.47%	99.86%	0.070%	99.93%
KNN	0.99%	0	0.99%	98.17%	0.29%	98.46%
BFS	0	0	0	99.56%	0.188%	99.75%
mix1	0	0	0	62.37%	0.026%	62.40%
mix2	0	0	0	98.09%	0.104%	98.19%
mix3	74.64%	8.79%	83.43%	99.83%	0.09%	99.92%
mix4	58.22%	10.36%	68.58%	99.88%	0.027%	99.91%
mix5	58.51%	17.66%	76.17%	99.93%	0.023%	99.95%

For mix5, HSCC even achieves up to 9.6X performance improvement compared to HDRC. To figure out the reason, we measure the utilization of DRAM cache and average access counts per DRAM page, as shown in Figure 6(b) and Figure 6(c).

For applications with large memory footprint, such as MSF, SetCover, KNN, BFS, mix4, and mix5, we find that their memory footprints are larger than the capacity of DRAM cache (1 GB), leading to approximate 100% DRAM utilization, as shown in Figure 6(b). However, the average DRAM page access frequency (hotness) is rather low for each of those workloads, as shown in Figure 6(c). We further take a look at the percentage of pages evicted from DRAM cache for these applications, as shown in Table 5. HDRC leads to more page evictions than HSCC for these applications. The reason is that HDRC adopts an on-demand cache fetching policy and organizes DRAM as a 16-way set-associative cache. HDRC needs to evict data blocks from a fully-filled set when fetching NVM pages to this set. Hence, HDRC can lead to severe cache thrashing because the on-demand cache fetching policy may evict a hot page and fetch it soon. As a result, HDRC decreases the access frequency of DRAM cache due to frequent page evictions, which lead to even worse application performance than the baseline. In contrast, HSCC actually enables a software-managed fully associative cache and increases the hotness of DRAM cache by filtering the cold NVM pages. As a result, HSCC uses DRAM cache more efficiently than HDRC when running workloads with large memory footprint.

For workloads with smaller memory footprint, such as astar, omnetpp, Canneal, DICT, NBODY, and mix1, HDRC achieves similar application performance to HSCC-Static and HSCC-Dyn because both DRAM cache utilization and access frequency of DRAM pages are almost equal in these systems. In addition, FLAT and RBLA show much lower DRAM cache utilization than HSCC for these applications. Because there is no page migration in FLAT, and the caching policy in RBLA is not based on hotness of NVM pages. As a result, HSCC-Dyn achieves up to 35% and 30.8% performance improvement compared to FLAT and RBLA, respectively.

Insight. *HSCC can improve application performance by even up to 9.6X compared to HDRC when the application's memory footprint is larger than the DRAM cache. When the applications have small memory footprint, HSCC achieves comparable performance with HDRC, and up to 35% and 30.8% performance improvement compared to FLAT and RBLA, respectively.*

5.3 Sensitivity to DRAM Cache Size

To study how the performance of HSCC-Dyn is sensitive to the capacity of DRAM cache, we run each selected workload with increasing capacity of DRAM cache. Figure 7(a) shows the IPC of each

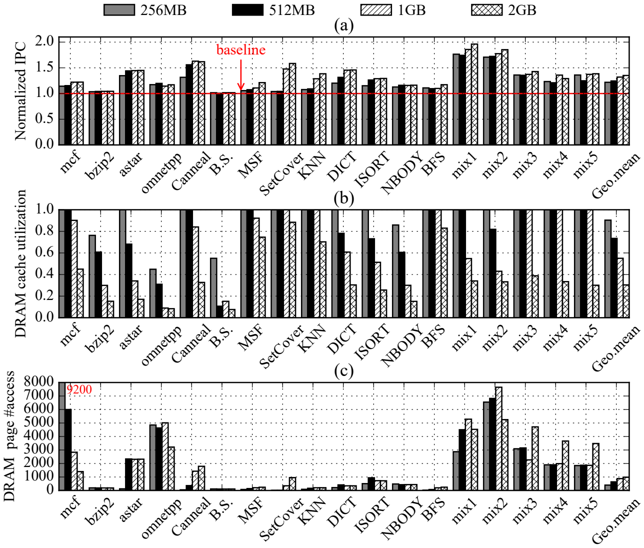


Figure 7: Sensitivity of HSCC-Dyn to different sizes of DRAM cache

application normalized to the baseline system. HSCC-Dyn with 256 MB, 512 MB, 1 GB, and 2 GB DRAM cache achieves 22.6%, 25.1%, 32.5%, and 35.8% performance improvement on average compared to the baseline system, respectively. To evaluate the efficiency of DRAM cache, we measure the DRAM cache utilization, as shown in Figure 7(b). We observe that the average utilization of DRAM cache in HSCC-Dyn declines with the increases of DRAM cache size. HSCC-Dyn using 256 MB, 512 MB and 1 GB DRAM cache achieves 90.3%, 92.1% and 97.6% performance of the 2 GB DRAM cache setting, respectively. Figure 7(c) shows the average access frequency of DRAM pages (DRAM hotness). For the multi-programmed workloads, HSCC-Dyn using larger DRAM cache exhibits higher hotness of DRAM pages due to less page evictions. HSCC-Dyn mitigates its sensitivity to the size of DRAM cache by only caching very hot NVM pages in DRAM, and evicting cold and clean pages from DRAM to NVM preferentially.

5.4 Energy Consumption

To evaluate the energy efficiency of HSCC, we measure energy consumption of HDRC, FLAT, RBLA, HSCC-Static, HSCC-Dyn in a system with 32 GB PCM and 1 GB DRAM cache. We also use a 32 GB DRAM-only system as a reference. As shown in Figure 8, all results are normalized to the baseline system (only 32 GB PCM).

Figure 8 shows that the energy consumption between HSCC-Static and HSCC-Dyn has insignificant difference. HSCC shows only 86% energy consumption of the baseline system, while HDRC leads to 31% more energy consumption. FLAT and RBLA show similar energy consumption as the baseline. For all hybrid memory systems, the energy consumed by DRAM cache is approximate 13%. For applications with small memory footprint such as bzip2, astar, Canneal, DICT, and ISORT, the energy efficiency in all hybrid memory systems are higher than the baseline system. For applications with large memory footprint such as MSF, SetCover, BFS, mix3, and mix5, HDRC consumes almost equal or even more energy than the 32 GB DRAM-only system. The reason is that HDRC leads to DRAM cache thrashing, where multiple NVM pages compete for the same set of DRAM cache, resulting in excessive page migrations between NVM and DRAM (costly in terms of energy and latency), especially that PCM consumes 20 times more energy per bit than DRAM on write operations. In contrast, HSCC adopts utility-based fetching policies to cache only hot pages in DRAM, and thus page migration operations are significantly reduced for large-footprint applications. As a result, HSCC reduces 34.3% and 65.6% energy consumption compared to HDRC and DRAM-only system, respectively.

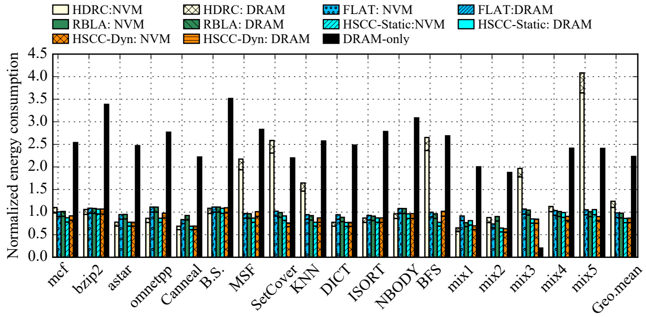


Figure 8: Normalized energy consumption in HDRC, FLAT, RBLA, HSCC-Static, HSCC-Dyn, and a 32 GB DRAM-only system

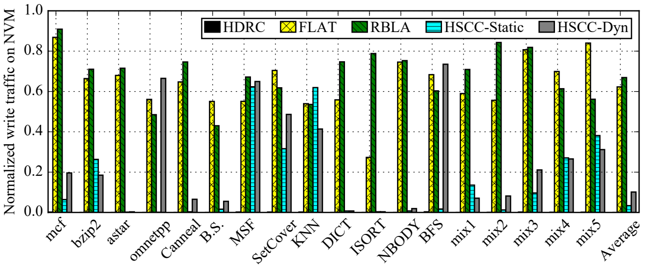


Figure 9: Total write traffic on NVM normalized to the baseline

5.5 NVM Write Endurance

In DRAM/NVM hybrid memory systems, it is essential to prolong the lifetime of NVMs (limited write endurance). To analyze the impact of DRAM caching on the endurance of NVM, we count the total write traffic on NVM for HDRC, FLAT, RBLA, HSCC-Static, and HSCC-Dyn, as shown in Figure 9. FLAT, RBLA, and HSCC deliver 67%, 72%, and 10% of the total memory write traffic to NVM, respectively. HSCC-Dyn significantly reduce the write traffic on NVM by 86% compared to RBLA because HSCC allows hot pages cached in DRAM. Amazingly, HDRC only causes negligible write traffic on NVM because HDRC actually implements a completely inclusive cache. Hence, a write operation on NVM occurs only when a dirty page is evicted from DRAM cache and should be written back to NVM. HSCC-Dyn needs a period of time to monitor the access counts of NVM pages before they are considered as hot pages, so there are still a portion of write traffic acted on NVM.

5.6 Overhead of HSCC

We analysis the storage overhead of HSCC and HDRC in a hybrid memory system composed of 1 GB DRAM and 32 GB PCM. HSCC can exploit the reserved bits in TLB to store the page access counts, and thus there is no additional storage overhead for TLB extension. HSCC only needs 32/64 = 50% additional storage to extend the PTEs, as shown in Figure 4. However, as the PTEs are stored in

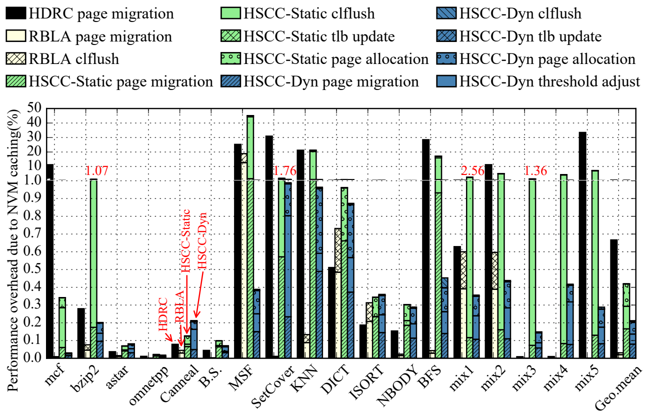


Figure 10: Breakdown of performance overhead due to page caching

main memory, the storage overhead of page table extension is trivial ($4B/4KB = 0.1\%$) for the hybrid memories. For comparison, HDRC requires 1 MB SRAM to store the tags of 1 GB DRAM cache (4B/page * 256K pages) [5]. To this end, HSCC significantly reduce the on-chip storage overhead compared to HDRC.

Figure 10 shows the breakdown of performance overhead due to NVM page caching for HDRC, RBLA, HSCC-Static, and HSCC-Dyn (the bars from left to right). For HDRC, the major cost is caused by page migrations, especially for applications with large memory footprints. For example, page migrations even result in 34% performance loss for mix5. For RBLA, the major performance overhead is caused by page migration and on-chip cache *clflush*. For HSCC-Static and HSCC-Dyn, the performance overhead due to NVM page caching mainly includes on-chip cache *clflush*, DRAM cache allocation, page migrations, and periodical execution of dynamic threshold adjustment algorithm. We model all the overhead in our simulator by adding reasonable delay accordingly. We find that these applications show significantly different performance overhead due to page caching, which is mainly determined by memory access patterns of applications. For example, page migration has a significant impact on the performance of MSF and BFS. However, for most workloads, the performance overhead are trial and can be offset by the benefit of selective page caching in HSCC.

6 RELATED WORK

Data placement/migration for flat-addressable hybrid memory systems. A simple architecture for hybrid memories is to organize both DRAM and NVM (stacked DRAM) as main memory in a single flat physical address space [4, 6, 7, 20, 24, 25]. In such kind of hybrid memory systems, the memory heterogeneity should be considered for data placement/migration schemes. Some studies propose OS-managed page migration to place hot data in DRAM and cold data in PCM [4, 7, 15]. Some other work propose hardware-assisted page migration policies, with memory controller monitoring page access patterns and migrating pages between hybrid memories [6, 26]. Ham *et al.* propose disintegrated memory controllers for a heterogeneous DRAM/PCM memory system, in which hardware-assisted page migration between heterogeneous memory devices is enabled [27]. Those studies share similar ideas to exploit page placement/migration policies in flat-addressable hybrid memory systems for high performance, energy efficiency, and endurance [6, 24, 28]. The difference between HSCC and those work is that HSCC *copies* the hot pages in NVM to DRAM and still maintains DRAM-to-NVM address mapping. Thus, when the DRAM becomes full and should be reclaimed for holding hotter NVM pages, HSCC only needs to reclaim clean pages in DRAM because there are still replicas in NVM. In contrast, the flat-memory approach needs to migrate pages from DRAM to NVM (costly and can reduce the lifetime of NVM).

Architecting hierarchical DRAM/NVM memories. Some DRAM/NVM hybrid memory systems organize DRAM as a cache to NVM [5, 8–10, 29]. Qureshi *et al.* propose a hybrid memory system using a small DRAM as a buffer for PCM-based main memory [5]. Park *et al.* propose a runtime-ware data decaying policy to improve power efficiency of hierarchical DRAM/PCM hybrid memory systems [9]. H. Yoon *et al.* propose to cache NVM rows with high row buffer miss rate in DRAM [10]. Since the DRAM cache is not part of the memory address space and is transparent to OSes, this kind of hybrid memory architectures needs additional hardware support for DRAM cache management. HSCC differs from those studies in three folds. First, HSCC is actually architected in a flat address space, however, it can logically support a cache/memory hierarchy with minimal hardware support. Second, HSCC can exploit the existing memory management modules in OSes to manage data in DRAM and NVM, with less intrusive modifications on the system software. Third, HSCC improves the previous work [5] by proposing a hardware/software cooperative caching mechanism and an utility-based cache filtering scheme.

Hardware-assisted cache management. Some previous work has explored Cache-Only Memory Architectures (COMA), which use memory as a hardware-managed cache [30–32]. Recently, there has been a large body of work on heterogeneous main memory design using in-package die-stacked DRAM as a large cache to off-package DRAM, such as [12, 26, 33–37]. A number of work propose block-granularity cache management and address the tag storing and lookup problems by co-locating a tag with each data block [12, 38]. Some other work manage stacked DRAM cache at the page granularity [34, 35]. Bi-Modal Cache manages cache blocks in flexible size according to data spatial locality [39]. Some proposals organize stacked DRAM as a hardware-managed cache while viewing the stacked DRAM in the memory address space [25, 33]. Our work differs from those studies as HSCC architects DRAM and NVM in a flat address space and manages DRAM as a logical cache in a software layer, significantly simplifying the hardware.

Cache bypassing. Cache bypassing is an effective technique to improve the LLC efficiency. There has been a large amount of work on cache bypassing policies for inclusive/exclusive on-chip LLCs [40–43]. Recently, a number of studies has focused on cache bypassing policies on GPU [44–46]. Sun *et al.* propose statistical cache bypassing for on-chip NVM-based cache [47]. Such work all relies on hardware support and provides limited flexibility. HSCC is inspired form those studies and further proposes hardware/software cooperative cache filtering schemes to improve the efficiency of cache-like DRAM in hybrid memories.

7 CONCLUSION

In this paper, we propose a hardware/software cooperative caching mechanism named HSCC, which organizes NVM and DRAM in a flat address space while logically utilizing DRAM as a cache to NVM. The proposed DRAM/NVM hybrid memory architecture can significantly simplify the hardware design and push DRAM cache management to the software layers. We propose utility-based cache filtering policies to improve the efficiency of DRAM cache. Extensive simulations show that HSCC improves system performance and energy efficiency significantly compared to a hardware-assisted DRAM/NVM memory system [11], and also achieves higher performance than flat-addressable DRAM/NVM memory systems and a RBLA caching policy [10].

Hugepage has become increasingly popular in big memory systems. One challenge of adopting hugepages in HSCC is that hugepage migration is extremely costly. An alternative solution is to enable re-sizable granularity of pages. For example, NVM is managed in hugepage (2 MB) and DRAM is used in typical pages (4 KB) as cache blocks of NVM hugepages. In the future, we will study hugepage migration and management techniques in hybrid memories.

ACKNOWLEDGMENT

Xiaofei Liao is the corresponding author of this paper. This work is supported jointly by National High-Tech Research and Development Program (863 Program) under grant No.2015AA015303, and National Natural Science Foundation of China (NSFC) under grants No.61672251, 61628204, 61433019.

REFERENCES

[1] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):1–55, 2015.

[2] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. Knowl. and Data Engin.*, 27(7):1920–1948, 2015.

[3] Krishna T Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C Lee, and Mark Horowitz. Rethinking DRAM Power Modes for Energy Proportionality. In *MICRO*, 2012.

[4] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: a Hybrid PRAM and DRAM Main Memory System. In *DAC*, 2009.

[5] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *ISCA*, 2009.

[6] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *ICS*, 2011.

[7] Wangyuan Zhang and Tao Li. Exploring Phase Change Memory and 3D Die-stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *PACT*, 2009.

[8] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling Efficient and Scalable Hybrid Memories using Fine-granularity DRAM Cache Management. *IEEE Comput. Archit. Lett.*, 11(2):61–64, 2012.

[9] Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. Power Management of Hybrid DRAM/PRAM-based Main Memory. In *DAC*, 2011.

[10] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[11] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *ISCA*, 2009.

[12] Gabriel H. Loh and Mark D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *MICRO*, 2011.

[13] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *ISCA*, 2013.

[14] Matthew Poremba, Tao Zhang, and Yuan Xie. NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems. *IEEE Comput. Archit. Lett.*, 14(2):140–143, 2015.

[15] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *EuroSys*, 2016.

[16] SPEC CPU 2006. *https://www.spec.org/cpu2006.*

[17] Parsec. *http://parsec.cs.princeton.edu/index.htm.*

[18] PBBS. *http://www.cs.cmu.edu/ pbbs/.*

[19] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms. In *HPCA*, 2010.

[20] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories. In *HPCA*, 2015.

[21] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *PACT*, 2011.

[22] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *ASPLOS*, 1987.

[23] SHMA. *https://github.com/CGCL-codes/SHMA.*

[24] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *ASPLOS*, 2015.

[25] Jaewoong Sim, Alaa R Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *MICRO*, 2014.

[26] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. Simple but Effective Heterogeneous Main Memory with On-chip Memory Controller Support. In *SC*, 2010.

[27] Tae Jun Ham, Bharath K Chelepalli, Neng Xue, and Brian C Lee. Disintegrated Control for Energy-efficient and Heterogeneous Memory Systems. In *HPCA*, 2013.

[28] Milan Pavlovic, Nikola Puzovic, and Adrian Ramirez. Data Placement in HPC Architectures with Heterogeneous Off-chip Memory. In *ICCD*, 2013.

[29] Alan Bivens, Parijat Dube, Michele Franceschini, John Karidis, Luis Lastras, and Mickey Tsao. Architectural Design for Next Generation Heterogeneous Memory Systems. In *IMW*, 2010.

[30] Sujoy Basu and Josep Torrellas. Enhancing Memory Use in Simple Coma: Multiplexed Simple Coma. In *HPCA*, 1998.

[31] Erik Hagersten, Anders Landin, and Seif Haridi. DDM-a Cache-only Memory Architecture. *Computer*, 25(9):44–54, 1992.

[32] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument for Simple COMA. In *HPCA*, 1995.

[33] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *MICRO*, 2014.

[34] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison Cache: A Scalable and Effective Die-stacked DRAM Cache. In *MICRO*, 2014.

[35] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *ISCA*, 2013.

[36] Mark Oskin and Gabriel H. Loh. A Software-Managed Approach to Die-Stacked DRAM. In *PACT*, 2015.

[37] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. A Fully Associative, Tagless DRAM Cache. *ACM SIGARCH Comput. Archit. News*, 43(3):211–222, 2015.

[38] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *MICRO*, 2012.

[39] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *MICRO*, 2014.

[40] Saurabh Gupta, Hongliang Gao, and Huiyang Zhou. Adaptive Cache Bypassing for Inclusive Last Level Caches. In *IPDPS*, 2013.

[41] Mazen Kharbutli and Yan Solihin. Counter-based Cache Replacement and Bypassing Algorithms. *IEEE Trans. Comput.*, 57(4):433–447, 2008.

[42] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-time Cache Bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, 1999.

[43] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and Insertion Algorithms for exclusive last-level caches. *ACM SIGARCH Comput. Archit. News*, 39(3):81–92, 2011.

[44] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive Cache Management for Energy-efficient GPU Computing. In *MICRO*, 2014.

[45] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *ICCAD*, 2013.

[46] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated Static and Dynamic Cache Bypassing for GPUs. In *HPCA*, 2015.

[47] Guangyu Sun, Chao Zhang, Peng Li, Tao Wang, and Yiran Chen. Statistical Cache Bypassing for Non-Volatile Memory. *IEEE Trans. Comput.*, 65(11):3427–3440, 2016.