Supporting Superpages and Lightweight Page Migration in Hybrid Memory Systems

XIAOYUAN WANG, HAIKUN LIU, XIAOFEI LIAO, JI CHEN, HAI JIN, YU ZHANG, and LONG ZHENG, Huazhong University of Science and Technology, China
BINGSHENG HE, National University of Singapore, Singapore
SONG JIANG, University of Texas, Arlington, USA

Superpages have long been used to mitigate address translation overhead in large-memory systems. However, superpages often preclude lightweight page migration, which is crucial for performance and energy efficiency in hybrid memory systems composed of DRAM and non-volatile memory (NVM). In this article, we propose a novel memory management mechanism called *Rainbow* to bridge this fundamental conflict between superpages and lightweight page migration. Rainbow manages NVM at the superpage granularity, and uses DRAM to cache frequently accessed (hot) small pages within each superpage. Correspondingly, Rainbow utilizes split TLBs to support different page sizes. By introducing an efficient hot page identification mechanism and a novel NVM-to-DRAM address remapping mechanism, Rainbow supports lightweight page migration without splintering superpages. Experiment results show that Rainbow can significantly reduce applications' TLB misses by 99.9%, and improve application performance (in terms of IPC) by up to $2.9\times$ (45.3% on average) when compared to a state-of-the-art memory migration policy without a superpage support.

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Software and its engineering** → **Virtual memory**; • **Computer systems organization** → *Multicore architectures*;

Additional Key Words and Phrases: Superpage, hybrid main memory, page migration, non-volatile memory

ACM Reference format:

Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. 2019. Supporting Superpages and Lightweight Page Migration in Hybrid Memory Systems. *ACM Trans. Archit. Code Optim.* 16, 2, Article 11 (April 2019), 26 pages.
<https://doi.org/10.1145/3310133>

This work is supported jointly by National Key Research and Development Program of China under grant No. 2017YFB1001603, and National Natural Science Foundation of China (NSFC) under grants No. 61672251, 61732010, 61825202, 61628204.

Authors' addresses: X. Wang, H. Liu (corresponding author), X. Liao, J. Chen, H. Jin, Y. Zhang, and L. Zheng, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; emails: {xiaoyuanw, hkliu, xfliao, chenji, hjin, zhyu, longzh}@hust.edu.cn; B. He, Department of Computer Science, School of Computing, National University of Singapore, Singapore; email: hebs@comp.nus.edu.sg; S. Jiang, Department of Computer Science and Engineering, University of Texas, Arlington, Arlington, Texas; email: song.jiang@uta.edu. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/04-ART11

<https://doi.org/10.1145/3310133>

1 INTRODUCTION

Today's data-intensive applications impose heavy demands on memory capacity. DRAM scaling, however, is unable to meet the increasing memory requirement for petabyte-scale big data processing. Emerging byte-addressable non-volatile memory (NVM) technologies, such as phase change memory (PCM) and 3D XPoint [1] offer high memory density, low cost per bit, and near-zero standby power consumption, at the expense of low performance and limited write endurance [31, 62]. Despite the promising features of NVM, it cannot entirely take the place of the DRAM. Instead, it is more practical to use NVM in conjunction with DRAM in hybrid memory systems [31, 48, 62, 63, 74].

With the continuously increasing application footprints and a corresponding growth of memory capacity, the virtual-to-physical memory address translation tends to be a new bottleneck of system performance [18, 32, 38]. Modern computer systems typically employ translation lookaside buffer (TLB) as a cache to store the recent virtual-to-physical address translations for faster retrieval later on. Upon each memory reference, the TLB is consulted first. If the requested address is not present in the TLB (i.e., a TLB miss), then the CPU needs to retrieve the absent address translation through hardware page table walking, which incurs a significant performance penalty due to multiple memory references (four in x86-64 systems) [77]. Previous studies have shown that TLB misses may significantly degrade system performance by up to 50% when the application's memory footprint becomes very large [19, 21, 23, 33]. The overhead can be as high as 90% of total system runtime in virtualization environments, since two layers of page table walking results in 24 rather than 4 memory references [16, 17, 22, 35, 76, 77].

There has been a large body of work on improving TLB coverage, i.e., the total memory address space that can be directly translated through TLBs. As the number of TLB entries does not scale up well due to speed, power, and space constraints, superpages have been widely exploited to increase TLB coverage [15, 30, 33, 65, 71]. A superpage refers to a large virtual page that maps to a number of continuous physical small (base) pages. The use of superpages can significantly broaden the TLB coverage (by a factor of 512 for typical 2MB superpages compared to 4KB small pages). However, the side effect is that superpages can hamper lightweight and agile memory management, such as page migration.

However, hybrid DRAM/NVM memory systems often rely on page migrations to achieve higher performance [48, 62, 63, 74] and energy efficiency [48, 55, 66, 74]. This in turn requires lightweight page migration schemes to move the frequently accessed (hot) pages from the slow NVM to the fast DRAM. However, page migration at the superpage granularity (e.g., 2MB) can incur unbearable performance overhead due to a vast waste of DRAM capacity and bandwidth if most memory references are distributed in a small region of the superpage (see Section 2.2). The cost may be even larger than the benefit of superpage migration. This presents a dilemma for the use of superpages, since the lightweight page migration can outweigh the benefits of extended TLB coverage.

In this article, we study how to exploit superpages for wide TLB coverage while supporting lightweight page migration in hybrid memory systems. To achieve this goal, several challenging issues should be addressed. (1) *Identification of lightweight hot pages*: to support lightweight page migration, a large body of work advocates monitoring memory accesses through the memory controller [55, 63]. However, using access counters at per-page granularity (i.e., 4KB) leads to prohibitively high storage overhead when the capacity of main memory becomes large (e.g., 1TB memory needs a total of $\frac{1TB}{4KB} \times 2B = 512MB$ storage on a 2B-per-page basis). Storing those records with on-chip SRAM is impractical. Another alternative is to store them in main memory. However, this would lead to higher memory access latency and additional record lookup overhead in main memory for each memory reference. (2) *Impact of lightweight page migration on TLB coverage*: page migrations often fragment superpages and thus break the physical address continuity.

Previous work has advocated splintering superpages to enable lightweight memory management such as page migration and sharing, while sacrificing the performance of address translation [37, 58]. It is still a challenge to retain the improved TLB coverage when the hot small pages within superpages are migrated to the DRAM. (3) **Efficiency of hot pages addressing**: as hot pages contribute to a major portion of applications' memory references, it is essential to further reduce the overhead of address translation for those hot pages in the DRAM.

To address the above problems, we propose *Rainbow*, a novel memory management mechanism to bridge the gap between superpaging and lightweight page migration for hybrid DRAM/NVM memory systems. *Rainbow* manages NVM and DRAM with different page granularities. Correspondingly, *Rainbow* exploits the available hardware feature of split TLBs [2, 7, 30, 52] to support different page sizes, with one TLB for addressing the superpages and another TLB for small pages. *Rainbow* migrates hot small pages within superpages to the DRAM without compromising the integrity of superpage TLB. As a result, *Rainbow* actually architects the DRAM as a cache to the NVM. *Rainbow* has the following novel designs to address the aforementioned challenging issues to support both superpages and lightweight page migrations:

- To mitigate the storage overhead of counting fine-grained page accesses, we propose to conduct the counting in two stages. In a given time interval, *Rainbow* first counts NVM memory accesses at the superpage granularity, then selects the *top N* hot superpages as targets. In the second stage, we only monitor those hot superpages at the small pages (4KB) granularity to identify hot small pages. This history-based policy avoids monitoring the sub-blocks (4KB pages) within a large number of cold superpages, and thus significantly reduces the overhead of hot page identification.
- We adopt split TLBs to accelerate the address translation performance for both DRAM and NVM references. To keep the integrity of superpages' TLB when some small pages are migrated to the DRAM, we use a bitmap to identify the migrated hot pages in the memory controller without splintering the superpages.
- We propose a physical address remapping mechanism to access the migrated hot pages in the DRAM, without suffering costly page table walks for addressing a DRAM page. To achieve this goal, we store the migrated hot pages' destination address in its original residence (the superpage). Once the hot pages' corresponding TLB misses, the DRAM page addressing should resort to an indirect access of the superpage. This design logically leverages the superpage TLBs as the next-level cache of the 4KB page TLBs. Because the superpage TLB hit rate is usually very high, *Rainbow* can significantly speed up the DRAM page addressing.

Putting those design components all together, we implement *Rainbow* within an integrated simulator based on *zsim* [67] and *NVMain* [60]. To the best of our knowledge, this is the first kind of work that supports superpages and lightweight page migration in hybrid memory systems. We compare *Rainbow* with several alternatives using a wide range of workloads. Experiment results show that *Rainbow* can significantly reduce the address translation overhead for applications with large memory footprints, and improve application performance by up to 2.9× (43.0% on average) compared to a hybrid memory migration policy without a superpage support. *Rainbow* also demonstrates higher energy efficiency than other policies.

The remainder of this article is organized as follows: Section 2 introduces the background and motivates our design for DRAM/NVM hybrid memories. Section 3 gives the detailed design of *Rainbow*. Experiment results are presented in Section 4. We discuss the related work in Section 5 and conclude in Section 6.

2 BACKGROUND AND MOTIVATION

We first introduce superpages and split TLBs. Next, we experimentally study memory access statistics of typical applications to motivate the design of Rainbow.

2.1 Superpage and Split TLBs

The evidence of performance degradation due to address translation has been well corroborated [19, 21, 23]. Modern data-centric applications are characterized with large memory footprint and lower data locality and are expected to incur even higher address translation overheads due to TLB misses. However, emerging NVM technologies are much denser and cheaper than the conventional DRAM and, consequently, we expect a rapid growth of memory capacity in the near future. The trends of big memory systems make the address translation problem become more urgent than ever before.

TLB misses can be mitigated by improving TLB coverage (or TLB reach). There are two fundamental ways to enlarge the TLB coverage, either by using more TLB entries or letting each entry map a larger memory page. Increasing TLB size implies larger die space area, higher energy consumption, and access latency. Another alternative is to use superpages, which have been proposed to improve TLB coverage since the 1990s [65, 72]. Most modern computer systems support superpages at both hardware and software levels. For example, x86-64 supports 4KB, 2MB, and 1GB page sizes, and processor vendors provide split TLBs for different page sizes correspondingly [2, 7, 30, 52]. A virtual address can be consulted in all split TLBs in parallel to shorten the address translation latency. Although split TLBs are simple for implementation and offer good performance, they would be underutilized without judicious allocation of superpages at different sizes. For example, if the OS only allocates 4KB pages, the 2MB superpage TLBs are wasted.

2.2 Memory Access Analytic of Superpages

Emerging NVM offers higher density than DRAM, but at the expense of higher access latency and lower bandwidth. Particularly, for the write operations, NVM is about 5–10× slower than DRAM and consumes up to 10× more energy than DRAM [64]. As a result, page migration is widely utilized to improve performance and energy efficiency in hybrid memory systems [48, 55, 62, 63, 74]. However, the use of superpages in hybrid memory systems precludes lightweight page migrations, because a superpage is required to be contiguous and aligned in both physical and virtual address spaces. Fine-grained page (e.g., 4KB) migration breaks the continuity of physical address space and thus splinters the superpages. Page migration at the superpage granularity (e.g., 2MB) can still retain the advantages of wide TLB coverage; however, it is prohibitively costly.

To evaluate the side effects of superpage migrations, we run several representative applications using 2MB superpages and profile their memory usage at the granularity of 4KB pages in an interval of 10^8 cycles. These applications are selected from SPEC CPU2006 [13], Parsec [10], Problem Based Benchmarks Suit (PBBS) [11], WhiteDB [14], Redis [12], Graph500 [4], Linpack [8], NPB-CG [9], and HPC Challenge Benchmark GUPS [5]. CactusADM, mcf, and soplex are selected from SPEC CPU2006. CactusADM is a computational kernel representative of many applications in numerical relativity. Soplex solves a linear program using the simplex algorithm. Mcf is a program used for single-depot vehicle scheduling in public mass transportation. Canneal, bodytrack, and streamcluster are multi-thread applications selected from Parsec. DICT, BFS, setCover, and MST are selected from PBBS. Both BFS and MSF all solve graph problems. SetCover is a computational biological problem. DICT is a dictionary matching algorithm. Both WhiteDB and Redis are in-memory database, and all use the dataset of Yahoo! Cloud Serving Benchmark (YCSB [29]). Linpack is a traditional supercomputer benchmark performing numerical linear algebra. Graph500

Table 1. Hot Page (4KB) Access Statistics

Application	Page access statistics (10^8 cycles)			Total memory footprint
	Hot page min# accesses	Working set size	Hot page percentage	
cactusADM	64	74.6MB	4.71%	776MB
mcf	30	1,089MB	2.36%	1,698MB
soplex	51	70.9MB	19.63%	1,888MB
canneal	2	891.6MB	8.52%	972MB
bodytrack	19	16.2MB	1%	620MB
streamcluster	10	105.5MB	27.60%	150MB
DICT	53	20.3MB	37.20%	384MB
BFS	30	404.1MB	20.51%	3,718MB
setCover	34	49.8MB	37.53%	2,520MB
MST	35	121.2MB	32.42%	6,660MB
WhiteDB	38	136.9MB	2.65%	3.6GB
Redis	7	12.4GB	1.26%	29.8GB
Graph500	64	7.20MB	6.35%	27.4GB
Linpack	63	40MB	21.19%	23.9GB
NPB-CG	64	40.9MB	24.7%	22.9GB
GUPS	4	7.6GB	5.8%	8.06GB

Fig. 1. The cumulative distribution function of superpages versus the number of touched 4KB small pages within a superpage in a given interval (10^8 cycles).

is a supercomputer benchmark based on large-scale data-intensive graph analysis. NPB-CG measures irregular memory access and communication performance. GUPS measures the rate of integer random updates of memory. These workloads cover a wide range of memory access patterns, and their memory footprints are shown in Table 1. All experiments are conducted in a simulated platform, as presented in Section 4.1. We have the following observations:

Observation 1: For most applications, only a small portion of 4KB pages are actually touched in each superpage during each sampling interval. Figure 1 shows the cumulative distribution function (CDF)

Table 2. Distribution of Hot 4KB Pages on Superpages

Application	Percent of superpages that only contain # hot 4KB pages					
	#1-32	#33-64	#65-128	#129-256	#257-384	#385-512
cactusADM	28.01%	34.1%	29.32%	0.65%	7.45%	0.47%
mcf	57.56%	16.48%	10.84%	9.95%	4.78%	0.39%
soplex	45.69%	10.88%	22.76%	9.28%	6.77%	4.62%
canneal	62.18%	15.86%	8.9%	11.57%	0.91%	0.58%
bodytrack	83.19%	6.01%	7.66%	2.18%	0.63%	0.33%
streamcluster	23.77%	30.55%	14.38%	13.71%	17.5%	0.09%
DICT	23.86%	14.53%	28.27%	22.14%	11.06%	0.14%
BFS	3.94%	18.19%	57.42%	6.35%	5.6%	8.5%
setCover	16.26%	24.28%	27.58%	17.36%	7.5%	7.02%
MST	13.44%	21.28%	21.77%	25.8%	16.31%	1.4%
WhiteDB	28.04%	53.14%	18.82%	0%	0%	0%
Redis	88.33%	6.67%	3.33%	1.67%	0%	0%
Graph500	61.48%	38.46%	0.06%	0%	0%	0%
Linpack	22.21%	14.71%	29.18%	16.3%	9.64%	7.96%
NPB-CG	0.05%	96.29%	2.66%	1.0%	0%	0%
GUPS	95.5%	4.5%	0%	0%	0%	0%

of superpages as a fraction of the touched small pages in one superpage. For many applications, we find that almost 80% of superpages' memory accesses are distributed on only a few small pages in a given interval. For cactusADM, the total number of touched small pages is even less than 100 for all superpages. This indicates that the migration of a whole superpage often results in wasted DRAM bandwidth and CPU time, and inefficient use of the limited DRAM capacity. The cost may be even higher than the benefit of superpage migrations.

Observation 2: Most applications' memory references are mainly distributed within a small portion of 4KB hot pages. Similar to CHOP [39], the hot pages are classified as the top N pages ranked by number of accesses, and the total accesses of these pages constitute 70% of the application's memory accesses. For each application, Table 1 shows the minimum access counts for these hot pages in working sets measured every 10^8 cycles, as well as the applications' total memory footprints. The hot page percent denotes the ratio of total volume of hot small pages to the working set in the sampling interval. Given the small fraction of touched small pages in each superpage, the proportion of hot small pages is even much smaller for many applications, such as mcf, canneal, bodytrack, WhiteDB, Redis, and GUPS. Take GUPS as an example: A page is classified as a hot page only if the number of memory references to this page exceeds 4 and only 5.8% of the total pages in the working set are classified as hot pages. We further analyze the distribution of these hot small pages on superpages. Table 2 shows the proportion of superpages that only contain a number of hot small pages. For many applications, we find that most superpages' memory references are mainly distributed on less than 128 hot small pages. This is extremely clear for data-intensive benchmarks. For GUPS and Graph500, even 95.5% and 61.48% of superpages only contain less than 32 hot small pages. This implies the hot small pages are distributed sparsely on the superpages, and thus it is more beneficial and lightweight to migrate only the hot small pages rather than the whole superpages from NVM to DRAM.

The above observations motivate us to design a new memory management mechanism that supports both superpages and fine-grained page migration for hybrid memory systems.

Fig. 2. Architecture of Rainbow.

3 DESIGN AND IMPLEMENTATION

In this section, we first give an overview of Rainbow and then present the technical details of hot page identification, utility-based page migration, and split TLBs. At last, we describe some other implementation issues such as cache/TLB consistency guarantees.

3.1 Architecture Overview

Figure 2 depicts the architecture of Rainbow. It only allocates 2MB superpages in the NVM and uses the DRAM to cache the hot NVM pages of 4KB size. Correspondingly, each processor uses two split TLBs to accelerate the address translation of superpages and small pages. In the memory controller, we design a lightweight page access monitoring mechanism to identify the hot small pages in the NVM. Also, we use a migration bitmap to flag the migrated small pages on a one-bit-per-page basis.

In the operating system (OS), we develop three modules. The hot page identifier periodically reads the page access counts in the memory controller and identifies the hot small pages within the monitored superpages. The page migration controller exploits a utility-based scheme to migrate small pages when the migration benefit is expected to be larger than the migration cost. The DRAM manager is responsible for page allocation and replacement. The three modules are performed periodically by special kernel modules in the background.

We adopt the buddy allocator used in many OS's for DRAM memory allocation. As DRAM is managed in 4KB pages and is used as a cache to the NVM superpages, we use a table to maintain the physical address mappings between NVM pages and DRAM pages. In this DRAM-to-NVM mapping table, for each DRAM page, we also use a few bits to record the protection and status bits, such as *page_present*, *page_rw*, *page_accessed*, *page_dirty*. Similar to the status bits in the superpage table entries, these bits are updated by the hardware and read by the OS.

When a DRAM page is written back to the NVM, the DRAM-to-NVM mapping table is used to retrieve the target NVM address. As the DRAM capacity is often much larger than the on-chip cache, conventional LRU-based replacement policies can cause significant performance penalty when they are implemented in the software layer. Like HSCC [48], we use three lists to manage

Fig. 3. The two-stage memory access monitoring for small hot page identification in *Rainbow*.

the DRAM memory: a free list to maintain unused pages, a clean list for unmodified pages, and a dirty list for dirty (modified) pages. Because the dirty pages should be written back to the NVM (costly), *Rainbow* preferentially selects free and clean pages for DRAM replacement at first. When the free and clean lists all become empty, *Rainbow* has to replace the dirty pages finally. Most of the time, page migrations would not disrupt the applications' execution. However, because a page should not be written during its migration, it may stall memory accesses to the on-the-fly pages. To mitigate the potential high latency of DRAM and NVM swapping, we proactively reclaim some dirty pages when the number of clean pages is smaller than a given threshold. This mechanism creates more free pages in the background and thus reduces the cost of DRAM page allocation when the DRAM is under high pressure.

3.2 Lightweight Hot Page Identification

Memory access monitoring at the page granularity (4KB) is costly when the memory size becomes very large. For example, if we use only 2 bytes to record the access counts of a 4KB page, 1TB memory requires a total of $\frac{1TB}{4KB} \times 2B = 512MB$ storage. It is impractical to store those records in on-chip SRAM.

We propose a two-stage memory access monitoring mechanism to mitigate the storage overhead. As shown in Figure 3, *Rainbow* divides the process of memory access monitoring into two phases. In stage 1, *Rainbow* first counts NVM data accesses at the superpage granularity (①). We use 2 bytes to store the access counts for each superpage in an interval of 10^8 cycles. For each data access to a superpage on the NVM, the memory controller determines which superpage corresponds to the physical address and updates the page access counts. We note that NVM write operations have a higher weighting of the counter value than NVM read operations. After a given time interval for superpage access counting, *Rainbow* then selects the *top N* hot superpages to further monitor them at the granularity of 4KB pages (②). Even though application footprints may be very large, their working sets in a short interval is often much smaller. Thus, the superpages' sorting latency is acceptable through a software approach. In stage 2, *Rainbow* monitors the small pages within those hot superpages and records their memory access counts (③) in a small table. As shown in Figure 4, we need 4 bytes to record the physical superpage number and 2 bytes to record the access counts for each small page. Note that the access counter uses 15 bits to store the data values and 1 bit as the overflow flag. An overflow implies that the superpage is definitely hot. Thus, monitoring a hot superpage requires $4B + 512 \times 2B = 1028$ bytes of total storage in a fine-grained manner. At last, *Rainbow* classifies those split small pages into hot and cold pages via threshold based classification (④). A page is determined as a hot page only when its migration benefit exceeds a given threshold (see Section 3.3). This history-based policy avoids monitoring

Fig. 4. The data structure for small page access counting.

Table 3. Parameters for Evaluating Page Migrations

Notations	Descriptions
C_r, C_w	total number of reads and writes on a page in a time interval
t_{nr}, t_{nw}	NVM read and write latencies
t_{dr}, t_{dw}	DRAM read and write latencies
T_{mig}	cycles spent in migrating a page from NVM to DRAM
$T_{writeback}$	cycles spent in writing a dirty DRAM page to NVM

cold superpages at the small page granularity, and thus significantly reduces the overhead of hot page identification.

3.3 Utility-based Hot Page Migration

Page migration from NVM to DRAM can improve memory access performance and energy efficiency. However, it also incurs increased access latency of requested data. Moreover, indiscriminate page migration can result in page thrashing between DRAM and NVM when memory pressure in DRAM becomes high. Like HSCC [48], we make a trade-off between the gained benefit and cost of page migrations. Table 3 presents the parameters for evaluating the benefits of page migrations in a time interval (10^8 cycles in our experiments).

When the DRAM has free pages to cache a NVM page, we should check whether the benefit of page migration is larger than the cost of page migration. We assume the migrated page will be read and written for C_r and C_w times in the next interval. The benefit of page migration is calculated as the total cycles saved by accessing data from DRAM against NVM. The total cycles spent in a page migration can be deemed as a constant, as shown in Equation (1):

$$Benefit_{mig} = (t_{nr} - t_{dr})C_r + (t_{nw} - t_{dw})C_w - T_{mig} \quad (1)$$

When the DRAM utilization becomes high, Rainbow may need to reclaim DRAM pages for holding the newly migrated pages. Rainbow would preferentially reclaim clean pages. However, if there are no clean pages, then Rainbow needs to evict dirty pages to the NVM. This results in bidirectional page migrations and less migration benefit. Assume a DRAM page $p1$ is evicted to hold a newly migrated NVM page $p2$, the total migration benefit should be offset by the cost due to page swapping, as illustrated in Equation (2). To mitigate the cost of page swapping, we monitor the data traffic of bidirectional page migrations and dynamically increase the threshold of migration benefit to select hotter small pages within each superpage:

$$\Delta Benefit_{mig} = (t_{nr} - t_{dr})(C_r^{p2} - C_r^{p1}) + (t_{nw} - t_{dw})(C_w^{p2} - C_w^{p1}) - T_{mig} - T_{writeback} \quad (2)$$

Fig. 5. The migration bitmap cache in Rainbow.

3.4 A Small Cache for Page Migration Bitmap

When a page is migrated to the DRAM, Rainbow sets the corresponding bit in the migration bitmap to identify whether this page has been migrated to the DRAM. For each 2MB superpage, we need a 512-bits bitmap to record the migration flags for all 4KB small pages. The storage overhead ($\frac{1}{4096 \times 8}$) is acceptable for a moderate-sized NVM device, and thus the migration bitmaps of all superpages can be placed in the memory controller (SRAM) for high performance. However, for large-capacity memory systems, it is impractical to store all migration bitmaps in SRAM. For example, 1TB NVM leads to 32MB bitmap data. For better scalability, we design a small cache in the memory controller to store the migration bitmaps of recently accessed superpages, while the whole migration bitmaps are still stored in the main memory.

The migration bitmap cache is implemented as an 8-way set-associative cache, as shown in Figure 5. The physical superpage number (PSN) is used to index the migration bitmap of a superpage, and the middle 9 bits (12 to 20) are used to index the migration flag of a small page in the bitmap. It requires only a number of bit shifting operations to locate the migration flag. Due to space constraints in the memory controller, Rainbow only uses 4000 entries to cache the page migration flags of total 8GB memory. Each cache entry requires 4 bytes for the PSN and 512 bits for the migration bitmap. The total storage overhead is only 272KB SRAM. Generally, the migration bitmap cache is filled accompanying with a superpage TLB miss. As the number of migration bitmap cache entries is much larger than the superpage TLB entries in Rainbow, the hit rate of migration bitmap cache is also higher than that of superpage TLBs. We further evaluated the timing parameters of the bitmap cache by using CACTI 3.0 [3]. It only leads to 9 cycles latency (similar to the L2 cache latency) before accessing the NVM, which is one order of magnitude lower than the inherent NVM access latency.

3.5 Split TLBs and Address Remapping

Once a page has been migrated to the DRAM, Rainbow stores the destination address (DRAM page number) in the page's original place. More specifically speaking, Rainbow overwrites the beginning 8 bytes data with the page's new physical address, which points to a DRAM page. Meanwhile, we set the corresponding flag in the migration bitmap. When the DRAM page is written back, if the data is not modified (clean), then we only need to write back the first 8 bytes of the page.

Rainbow leverages split TLBs cooperatively to accelerate virtual-to-physical address translations for both DRAM and NVM. Upon a memory reference, the split TLBs are consulted in parallel, as shown in Figure 6. Generally, we have the following four cases: (1) 4KB page TLB hit and superpage TLB hit; (2) 4KB page TLB hit and superpage TLB miss; (3) 4KB page TLB miss and superpage TLB hit; (4) 4KB page TLB miss and superpage TLB miss.

Fig. 6. Four cases of memory addressing in Rainbow.

For the first and second cases, Rainbow chooses the physical address that the 4KB page TLB returns (path ① in Figure 6). These two cases imply that the accessed data has been cached in the DRAM, and the stale data in the NVM is invalid.

For the third case, Rainbow needs to check the migration bitmap with the returned physical address. If the corresponding migration bit is set, meaning that the small page within the superpage has been cached in the DRAM, then Rainbow reads the first 8 bytes of this page in the NVM to obtain the target physical address, which points to a page in the DRAM (② in Figure 6). At the same time, Rainbow inserts a new TLB entry associated with the DRAM page in the 4KB TLBs. If the small page is not migrated, then Rainbow gets the physical address translated by the superpage TLB (③ in Figure 6). At last, Rainbow sends the translated physical address to on-chip cache or main memory (upon LLC misses) to access the requested data.

For the fourth case, Rainbow performs hardware page table walking for the superpage address translation (④ in the Figure 6). When the page tables return the physical address, Rainbow also needs to check the migration bitmap, and the following operations are the same as the third case.

We note that some CPU architectures (e.g., Intel Skylake [7]) use a unified L2 TLB for both 4KB and 2MB page translations. Our split TLB design only requires a little modification to adapt to this TLB organization. A simple approach is to add a tag bit (i.e., *sp_bit*) for each L2 TLB entry to identify whether it is used for a superpage translation or a small page translation. For each TLB lookup, if *sp_bit* is set, meaning this TLB entry maps to a superpage, then only the virtual superpage number is compared with the TLB tag. Otherwise, the virtual (small) page number is used to look up the small page TLB.

As illustrated in Figure 6, although the hot small pages are migrated between DRAM and NVM, Rainbow does not need to splinter the NVM superpages and the corresponding TLBs. Any memory references to a migrated hot page are redirected to the DRAM through only one access to the superpage. This address remapping mechanism guarantees the transparency of hot page migration from the view of applications.

In the following, we analytically compare the cost of DRAM page addressing in Rainbow with the traditional page table walking mechanism [77]. We assume that the page tables of DRAM and NVM are stored in the fast DRAM. Upon the 4KB page TLB miss, page table walks result in four memory references to the four-level page tables, and thus the address translation overhead is $4 \times t_{dr}$. For Rainbow, we need to read the DRAM page's physical address from the corresponding superpage in NVM. Since the superpages have only three-level page tables, there are three memory

references to the page tables and one memory access to NVM for reading the DRAM page address. Assuming the hit rate of superpage TLBs is R_{hit} , the DRAM page addressing overhead becomes

$$\begin{aligned} Cost_{DRAM-addr-trans} &= R_{hit} \times t_{nr} + (1 - R_{hit}) \times (3t_{dr} + t_{nr}) \\ &= 3(1 - R_{hit}) \times t_{dr} + t_{nr}. \end{aligned} \quad (3)$$

Assuming the NVM read latency t_{nr} is twice as much as the DRAM read latency t_{dr} , we deduce that Rainbow leads to lower DRAM page addressing overhead than the page table walking mechanism only if $R_{hit} > 33.4\%$, and reduces DRAM page addressing overhead by 46.25% when $R_{hit} = 95\%$. Since the hit rate of superpage TLBs is rather high for many applications (over 99% in our experiments), Rainbow is able to significantly reduce the overhead of DRAM page addressing. To some extent, Rainbow essentially enables the superpage TLB to be another larger cache to the 4KB page TLBs. Moreover, we can deduce that the DRAM address remapping mechanism is better than the page table walking mechanism if the ratio of NVM read latency to DRAM read latency is lower than four.

3.6 Data Consistency

Data Consistency between DRAM and NVM. As mentioned above, the hot data has two replicas, one in DRAM and one resident in NVM. Correspondingly, a virtual address may be presented in both superpage TLBs and 4KB page TLBs. To guarantee the data consistency, we use a migration bitmap in the memory controller to mark the migrated hot pages. For each memory reference on the NVM, the migration flag is first checked to make sure that Rainbow always accesses the data cached in the DRAM.

Cache Consistency. Since some processors leverage write-back cache solutions, where the write operations are directed to cache and the completion is immediately confirmed to the host CPU. The dirty data blocks are written to main memory only at specified intervals or under the condition of cache evictions. This mechanism often brings higher performance but may result in inconsistency problems. Because a page may be referenced by a set of cache lines in on-chip cache, a page migration may copy the stale data to another place, allowing a portion of data inconsistent with the replica in on-chip caches. Rainbow utilizes *clflush* instructions to address this problem. To be more specific, the *clflush* instruction invalidates all cache lines associated with the migrated page from all levels of the processor's cache hierarchy. The invalidation is broadcast throughout the cache coherence domain. If a cache line contains modified (dirty) data at any level of the cache hierarchy, then the cache line is written back to the main memory before invalidation. In this way, when a page is migrated, the corresponding dirty cache lines are written to main memory and clean cache lines are invalidated.

TLB Consistency. Similar to the cache consistency issue, page migration may also cause TLB inconsistency problems [44, 48] because a page may be referenced by multiple cores' TLB entries. A simple solution is to adopt the TLB shutdown mechanism [26, 44, 48]. That is, when a processor's TLB changes an address translation, the same TLB entries in other cores should be invalidated. However, in Rainbow, an NVM-to-DRAM page migration does not cause a TLB inconsistency problem. As mentioned above, our address remapping mechanism is able to logically guarantee the contiguity of superpages, and thus the hot page migration is not perceived by the superpage TLB. Also, because a migrated page in DRAM is not necessarily accessed in the immediate future, the 4KB page TLB associated with the DRAM page will be constructed on its first access. When a DRAM page is written back to the NVM, we adopt the TLB shutdown mechanism [26] to invalidate all cores' 4KB TLB entries associated with the DRAM page, and the record of DRAM-to-NVM page mapping is removed from the mapping table. For DRAM pages and NVM superpages, their status bits such as *accessed* and *dirty* bits are stored in the DRAM-to-NVM mapping table

Table 4. System Configuration in Rainbow

CPU	8 cores, 3.2GHz, out-of-order
L1 TLB	32 Data TLB entries for 2MB superpages, and 64 Data TLB entries for 4KB small pages per core, 4-way, 1-cycle latency
L2 TLB	512 unified TLB entries for 2MB superpages, and 1,024 unified TLB entries for 4KB small pages, 8-way, 8-cycle latency
L1 Cache	private 64KB per core, 4-way, split D/I, 3-cycle latency
L2 Cache	private 256KB per core, 8-way, 10-cycle latency
L3 Cache	shared 8MB, 16-way, 34-cycle latency
Bitmap Cache	272KB, 8-way, 9-cycle latency
DRAM	4GB: 1 channel, 4 rank, 32 banks, 32,768 rows, 64 cols, Bandwidth: 10.7GB/Sec, FR-FCFS request scheduling, Timing (tCAS-tRCD-tRP-tRAS) : 7-7-7-18 (cycles), 13.5ns read latency, 28.5ns write latency
PCM	32GB: 4 channels, 8 ranks, 8 banks per rank, 65,536 rows, 32 cols, Bandwidth: 10.7GB/Sec, FR-FCFS request scheduling, Timing (tCAS-tRCD-tRP-tRAS): 9-37-100-53 (cycles), 19.5ns read latency, 171ns write latency
Power/Energy consumption	
DRAM	Voltage: 1.5V, Standby: 77mA, Refresh: 160mA, Precharge: 37mA; Read and write on row buffer hit: 120 and 125mA; Read and write on row buffer miss: 237 and 242mA
PCM	Read/write on row buffer hit: 1.616 pJ/bit; Read and write on row buffer miss: 81.2pJ/bit and 1,684.8pJ/bit

and superpage table entries, respectively, as described in Section 3.1. Page migrations between DRAM and NVM only lead to updating the status bits in different places. They do not cause data consistency issues among small page TLBs, superpage TLBs, and superpage table entries.

4 EVALUATION

4.1 Experimental Methodology

We implement Rainbow in an integrated simulator based on zsim [67] and NVMain [60]. Zsim is a fast x86-64 multi-core simulator built on Pin [49]. We extend zsim to support many OS-level functions, such as buddy allocator, page tables, and TLB management operations. NVMain is a cycle-accurate memory simulator that can model both DRAM and NVM in detail. In our experiment, NVMain is used to simulate the hybrid main memory composed of DRAM and NVM, each with an individual memory controller. NVMain can also simulate memory-level parallelism by using channel/rank/bank interleaving address decoders. As a page only logically presents a contiguous physical address space, the memory controller can still infer which page a physical address maps to and updates the page access counts, no matter which memory decoding scheme is used.

Configuration. The platform and the detailed configuration in our experiments are depicted in Table 4. PCM [40, 75] is chosen as the storage medium of main memory, as it is a widely studied NVM. Timing and energy parameters of PCM are referred to in References [46, 48]. We also model the latencies of clflush, TLB shutdown, and data move-in details according to the timing parameters of CPU and DRAM/NVM.

Alternative policies. We compare Rainbow with several alternative page migration policies for hybrid memories as follows:

Table 5. Workloads for Evaluation

Workloads	Applications
SPEC CPU2006	cactusADM, mcf, soplex
Parsec	canneal, bodytrack, streamcluster
PBBS	DICT, BFS, setCover, MST
In-memory database	WhiteDB, Redis
Large footprints	Graph500, Linpack, NPB-CG, GUPS
mix1	cactusADM+soplex+setCover+MST
mix2	setCover+BFS+DICT+mcf
mix3	canneal+DICT+MST+soplex

- **Flat-static:** 4GB DRAM and 32GB NVM are organized in a flat address space [48] and are managed in 4KB small pages. Data is evenly distributed in DRAM and NVM according to the ratio of DRAM to NVM capacity (1:8). There is no page migration between NVM and DRAM. We use this system as a baseline for comparison.
- **HSCC-4KB-mig:** HSCC is a state-of-the-art hybrid memory system that supports utility-based page migration at the granularity of 4KB page [48]. The major difference between Rainbow and HSCC is the support of superpages. This comparison is made to evaluate the effectiveness of using superpages in hybrid memory systems.
- **HSCC-2MB-mig:** We modify HSCC to support superpages and memory migration at 2MB superpages granularity. This comparison is made to evaluate the performance and energy penalties of superpage migrations.
- **DRAM-only:** This is a system with only 32GB DRAM and supports only 2MB superpages. We use it as the applications' performance upper bound because they can benefit from superpages without any page migrations.

Benchmarks. We evaluate a number of workloads with different memory access patterns from SPEC CPU 2006 [13], Parsec [10], Problem Based Benchmarks Suite (PBBS) [11], WhiteDB [14], Redis [12], Graph500 [4], Linpack [8], NPB-CG [9], and GUPS [5], as listed in Table 5. Detailed memory access patterns of these applications are shown in Table 1. In addition, we evaluate three multi-programmed workloads, as shown in Table 5.

4.2 Address Translation Overhead

Figure 7 shows superpages significantly reduce TLB misses per kilo instructions (MPKI) by several orders of magnitude on average. Although Rainbow supports different page sizes, it shows almost similar TLB performance with HSCC-2MB-mig and DRAM-only (2MB). The reason is that Rainbow logically uses the superpage TLBs as a larger next-level cache to the 4KB page TLBs. For applications with small memory footprints, such as bodytrack and streamcluster, the MPKI is significantly reduced because of the wider TLB coverage (1GB) offered by the superpages. As shown in Table 1, GUPS and canneal are memory intensive benchmarks and show very large working sets in a short sampling interval. As a result, GUPS and canneal show relatively high MPKI even using superpages. Mix2 shows both a large working set and large memory footprint, leading to a large amount of page swapping between DRAM and PCM in HSCC-2MB-mig. Thus, HSCC-2MB-mig incurs a lot of TLB shutdown operations, causing a relatively high MPKI. In contrast, Rainbow does not cause TLB shutdown when migrating hot small pages within superpages to DRAM.

Figure 8 shows the percent of execution cycles spent on servicing TLB misses for different applications. When the memory is managed in 4KB small pages, the TLB miss overhead even

Fig. 7. MPKI of applications.

Fig. 8. Percent of total cycles spent on servicing TLB misses. The very small values show TLB miss overheads in Rainbow.

Fig. 9. The breakdown of detailed address translation overheads in Rainbow.

approximates to 60% of total execution cycles for soplex, Redis, Graph500, and mix2. For mcf, canneal, Redis, GUPS, and mix3, since their working sets approximate or exceed the superpage TLB coverage, they cause relatively high address translation overheads even using superpages. Overall, superpages are able to significantly reduce TLB miss overhead by 99.9% on average.

We further study the detailed address translation overheads in Rainbow. Figure 9 shows the breakdown of execution cycles spent on split TLB hits, bitmap cache hits/misses, superpage table walks (SPTWs), and address remapping. The overall address translation causes 32.4% performance overhead on average. Since split TLBs are on the critical path of address translation, they introduce 64.6% of total address translation overhead, although the TLB latency is very short. The bitmap cache costs near 17.5% of total address translation overhead, because it should

Fig. 10. Normalized IPC relative to the Flat-static system.

be consulted for each access to the NVM. To address DRAM pages when the corresponding 4KB page TLB misses, the address remapping mechanism leads to 13.2% of total address translation overhead on average. The bitmap cache miss can result in relatively higher latency; however, we observed trivial bitmap cache misses even for applications with very large footprints, such as Graph500, NPB-CG, and Linpack. As the superpage hit rate even exceeds 99.9% on average, the average cost on superpage table walks is as low as 4.5%. We only observe remarkable cost on SPTWs for canneal, GUPS, and Redis, because their working sets are extremely large, resulting in frequent superpage TLB thrashing and high TLB miss rates.

4.3 Application Performance

Figure 10 shows the *instructions per cycle* (IPC) of each application normalized to the baseline system (Flat-static). Rainbow achieves 85.1%, 45.3%, and 24.5% performance improvement on average, compared to Flat-static, HSCC-4KB-mig, and HSCC-2MB-mig, respectively. The performance difference between Rainbow and the upper bound (DRAM-only) is only 13.7%, on average.

Compared to HSCC-4KB-mig, Rainbow can significantly improve the IPC of mcf, soplex, and Graph500 by 2.1 $\times$, 1.2 $\times$, and 2.9 $\times$, respectively. For mcf, since superpages reduce the MPKI by approximately 99%, they deliver 2.1 $\times$ performance improvement compared to HSCC-4KB-mig. This suggests that superpages can significantly reduce the address translation overheads for memory-intensive applications. Soplex, SetCover, GUPS, and Graph500 all show rather poor data locality. However, they also show significant performance improvement against the systems without superpage support. This implies that applications with poor data locality can also benefit from superpages because of the improved TLB coverage.

We also find that HSCC-2MB-mig may result in lower application performance than HSCC-4KB-mig, such as cactusADM, streamcluster, DICT, setCover, and MST. This implies that page migrations at the superpage granularity are extremely costly. The benefit of using superpages is significantly offset by the cost of superpage migrations. In contrast, Rainbow explores the advantages of both superpages and lightweight page migrations, and thus achieves much better application performance. For the DRAM-only system with 2MB superpages supported, it shows the best performance against other policies because it takes full advantage of superpages without any memory migrations. We also note that it is not a completely fair comparison, since DRAM-only uses more DRAM.

Insight. (1) For applications with intensive memory accesses or poor data locality, Rainbow can significantly improve application performance by up to 2.9 $\times$. (2) For other applications, the cost of superpage migrations can offset the advantages of superpages. Using the proposed lightweight page

Fig. 11. Normalized page migration traffic relative to the applications' total memory footprints.

migration scheme without splintering superpages, Rainbow can still improve application performance by 37.4%.

4.4 Page Migration Traffic

Figure 11 shows the ratio of migration traffic to total memory footprint for each application. Generally, HSCC-2MB-mig shows larger migration traffic than other migration policies because of the large granularity of page migrations. As a result, superpage migrations lead to wasted memory bandwidth on copying the cold data within superpages. Rainbow can reduce page migration traffic by 55.8% on average compared to HSCC-2MB-mig. For memory-intensive applications, such as solex, canneal, and Graph500, HSCC-4KB-mig shows more migration traffic than Rainbow, because the page access counting scheme in HSCC is implemented in TLB and does not filter the memory references in on-chip caches, and thus more pages are migrated to the DRAM. For MST, GUPS, and Linpack, because their memory footprints are larger than the capacity of DRAM (4GB), HSCC-2MB-mig leads to a large amount of page swapping between DRAM and PCM. Thus, the migration traffic is even larger than their total memory footprints. In contrast, Rainbow only selects the hot small pages for migration and thus significantly mitigates the frequent page swapping. We also observe that page migrations only consume 1.35% of total memory bandwidth at most. Thus, page migrations lead to trivial memory bandwidth contention with these applications.

4.5 Energy Consumption

DRAM consumes a large amount of energy due to periodic refreshing, while NVM leads to near-zero static energy consumption. To evaluate energy efficiency of Rainbow, we compare energy consumption of those migration schemes using Flat-static as a baseline, as shown in Figure 12. Generally, the DRAM-only system shows much more energy consumption than the hybrid memory systems on average. Rainbow is able to reduce energy consumption by 49.2% and 70.5% on average, compared to the Flat-static and DRAM-only (2MB), respectively. Although Flat-static does not introduce additional energy consumption due to page migrations, it causes more energy consumption than HSCC and Rainbow. The reason is that a large amount of memory references are distributed on the PCM, resulting in higher active energy consumption of PCM. This is because write operations on PCM consume 20 times more energy per bit than on DRAM [46]. This phenomenon is more clear for mcf, which shows that the misuse of hybrid memories causes higher energy consumption compared to the DRAM-only system. In contrast, Rainbow and HSCC migrate hot pages to the DRAM, which services a large portion of memory accesses with higher energy efficiency.

Fig. 12. Normalized energy consumption relative to the baseline system.

Fig. 13. Migration traffic and IPC vary with the sampling intervals in Rainbow.

4.6 Sensitivity Studies

To study how the application performance in Rainbow is sensitive to the time interval for hot page monitoring, we run selected applications with different sampling intervals. Figures 13(a) and 13(b) show how the normalized migration traffic and application IPC are sensitive to the sampling interval, respectively. All experimental results are normalized to the sampling interval of 10^5 cycles. Generally, a longer sampling interval usually cause less software overhead for hot page identification. However, we find that less hot pages are migrated to the DRAM when the sampling interval exceeds 10^8 cycles. Correspondingly, the applications' IPC also decreases with the growth of sampling interval. As a result, we set the sampling interval as 10^8 cycles in Rainbow for better performance.

To evaluate how the number of selected top N hot superpages can affect the page migration traffic and application performance in a given time interval (10^8 cycles), we run some memory-intensive applications with different settings of N . Figure 14(a) shows that there is trivial growth of migration traffic for those applications when the number of selected hot superpages exceeds 50. Figure 14(b) also shows that those applications' IPC become stable when the value of N is larger than 50. This suggests that the majority of hot small pages of applications are distributed on only a few hot superpages. As a result, we prudently set N to be 100 in Rainbow. We argue that the top 100 hot superpages are enough for hot small pages identification, because many applications' working sets are much less than 200MB in each sampling interval.

Fig. 14. Migration traffic and IPC vary with the number of top N hot superpages in Rainbow.

Table 6. Storage Overhead of Rainbow with 1TB PCM

Data Structure	Storage overhead
Migration bitmap (1 bit per 4KB-sized page)	272KB SRAM
Access counters for superpages in PCM (2 Byte per 2MB)	1MB SRAM
Physical Superpage Number (PSN) of top N hot superpages (4 Byte per superpage)	$4N$ Bytes SRAM
Access counters for split small pages in the top N hot superpages ($2B \times 512 = 1KB$ per hot superpage)	N KB SRAM
Total storage overhead (if $N = 100$)	1.372MB SRAM

We have also studied the sensitivity of other settings in Rainbow. Due to space limitations, we only describe the results here. The first one is the threshold for hot page identification. We find that less hot pages are migrated to DRAM when the threshold increases. Correspondingly, the applications' IPC also become lower. We have also studied the impact of different NVM access latencies on the effectiveness of page migration. When the NVM read/write latencies increase linearly, a little more pages are migrated to DRAM because the migration benefit increases according to Equations (1) and (2). However, the applications' performance degrades, because a large portion of cold data on the NVM introduces higher accumulative access delay.

4.7 Storage and Runtime Overhead

We analyze the storage overhead of Rainbow in a hybrid memory system comprising of 1TB PCM. The storage overheads mainly come from migration bitmaps and superpage access counters. We list all costs in Table 6. For the migration bitmaps, 1TB PCM needs a total of $\frac{1TB}{4KB \times 8} = 32MB$ storage to store all superpages' migration bitmaps. We put the whole bitmaps in the main memory and cache only a portion of recently accessed ones (272KB) in the memory controller.

We use 2 bytes to record both superpages and small pages' access counts. There are a total of 512K superpages for 1TB PCM, and thus consume 1MB SRAM for superpage access counters. As described in Section 2, although many applications may have a very large memory footprint during execution, they show relatively small working sets in the sampling time interval (10^8 cycles). As a result, we only select the top 100 hot superpages for fine-grained page access counting at the second stage, and thus only require $1.004 \times 100 = 100.4KB$ of additional storage. Overall, Rainbow only causes 1.372MB SRAM storage overhead for a big memory system with 1TB PCM, and the hardware die area overhead modeled by CACTI [3] is only 7%.

Fig. 15. Breakdown of runtime overhead in Rainbow.

Fig. 16. Performance overhead of page access accounting in Rainbow, all relative to applications' total execution time.

Figure 15 shows the breakdown of performance overhead due to the address remapping mechanism, bitmap cache, page migration, TLB shutdown, and cflush. We model all these operations in our simulator by adding reasonable latencies accordingly. We find that these applications show significantly different performance overhead at runtime. For soplex, mix2 and mix3, DRAM page addressing accounts for the majority of runtime overhead, implying that these workloads show relatively a high miss rate of 4KB page TLBs. DICT, BFS, setCover, MST, and Graph500 all spend a relatively large portion of time accessing the bitmap cache, suggesting that many memory accesses are distributed on NVM due to poor data locality of these workloads. MST, Linpack, and NPB-CG show very large memory footprints, and thus a large fraction of execution time is spent on page migrations. Overall, the runtime performance overhead of Rainbow is 8.9% on average. However, it can be offset by the significant benefit of using superpages and lightweight page migrations.

We also evaluate software overhead due to hot page identification in Rainbow. The performance overhead is mainly attributed to sorting the numbers of page accesses by software. We compare the two-stage page access accounting mechanism (tracking top N hot superpages) with a simple approach that always tracks all 4KB split pages in each sampling interval. Figure 16 shows that

our hot page monitoring mechanism in Rainbow only introduces less than 2.4% (0.6% on average) performance overhead, all normalized to the applications' total execution time. Because Rainbow can significantly reduce the number of pages tracked in each sampling interval, particularly for applications with large working sets such as GUPS, it can reduce the performance overhead by up to 98.4% (86.3% on average) compared to the approach that tracks all 4KB pages. We note that the hot page identification is not on the critical path of applications; it can be performed by a specific CPU core to avoid the performance interference.

5 RELATED WORK

We summarize the related work in the following categories:

Superpages and TLBs. There have been many studies on mitigating the performance overhead of virtual-to-physical address translations [21, 24, 25, 57, 70]. Due to energy and latency constraints on TLB designs, a majority of studies have focused on superpages for improving TLB coverage. Talluri et al. discuss the challenges and tradeoffs to support superpages in hardware [73]. Libhugetlbfs [6], Ingens [45], and Illuminator [51] are OS-level supports for hugepage management. TLB coalescing [54, 56, 57] and MMU cache coalescing [23] have been proposed to increase the coverage of TLB and MMU cache. Redundant memory mappings (RMM) [41, 42] extends TLB coverage by mapping ranges of virtually and physically contiguous pages in a range TLB. SpecTLB [20] offers speculative address translations to hide the latency of many TLB misses. Interestingly, a recent work SEESAW [53] shows superpages can be leveraged to improve the performance and energy efficiency of L1 VIPT caches. Moreover, the existing CPU MMU ideas, including TLBs, page table walker, and shared page walk cache, have been successfully adopted for GPU MMU designs [59, 61, 69]. These studies all focuses on using superpages and TLBs to improve the performance of virtual address translations. Most of those technologies, such as TLB coalescing, are complementary to our work for further reducing address translation overhead.

Many studies have focused on improving the availability of superpages. Navarro et al. propose reservation-based allocation and deferring promotion [50] to support superpages in the OS layer. Gorman et al. [36] propose a physical page allocator to mitigate memory fragmentation and promote page contiguity. Zhang et al. proposed Enigma to map superpages to discontinuous physical pages using a intermediate address (IA) space [78]. GLUE [58] groups contiguous, aligned small page translations under a single speculative huge page translation in the TLB. GTSM [33] leverages contiguity of physical memory extents to construct superpages even when pages have been retired due to bit errors. Skewed associative TLB [68] supports concurrent use of multiple page sizes in a single process. MIX TLB [30] exploits superpage allocation patterns to concurrently support multiple page sizes. Those studies are orthogonal to our work, as the design space is different. *Rainbow* mainly aims to address a thorny problem of enabling lightweight page migration in a superpage-supported hybrid memory system.

Page Migration in Hybrid Memory Systems. There have been a number of studies on page migration for hybrid memory systems [31, 34, 48, 63]. Both PDRAM [31] and CLOCK-DWF [47] migrate frequently written NVM pages to DRAM while leaving read-intensive pages in the NVM. CAMEO [28] only considers data recency to cache the recently accessed data line in stacked memory by swapping data lines between stacked memory and off-chip memory regions. RaPP [63] ranks pages according to the access frequency and recency and migrate the top-ranked NVM pages to DRAM. Bock et al. proposed CMMP [27] to concurrently migrate multiple pages. PageSeer [43] exploits hints from page table walks in a TLB miss to direct hardware-assisted page swapping. Those studies have assumed that the hybrid main memories are uniformly managed at the granularity of 4KB pages, and thus naturally support lightweight page migration [62]. The context of

Rainbow is different from those studies, mainly focusing on supporting lightweight page migration in hybrid memory systems while still preserving the benefit of superpages.

Although HSCC [48] and Rainbow all support a cache/memory hierarchy and hot page migration in hybrid DRAM/NVM memory systems, there are several significant differences between HSCC and Rainbow: (1) HSCC only supports a single size of pages for DRAM and NVM, while Rainbow supports different page sizes to manage the hybrid memories; (2) HSCC performs page access accounting through a moderate extension of TLBs and page tables, without considering the impact of cache filtering on actual data accesses to main memory. In contrast, Rainbow monitors page accesses in the memory controllers of DRAM and NVM. Particularly, to mitigate the storage overhead of fine-grained page access accounting in Rainbow, we propose two-stage memory access accounting; (3) HSCC maintains the DRAM-to-NVM address mappings in the extended page tables. In contrast, Rainbow accesses DRAM pages through split TLBs and an NVM-to-DRAM address remapping mechanism. Moreover, Rainbow logically uses the superpage TLBs as a next-level cache of the 4KB page TLBs, and thus can significantly reduce the TLB miss rate.

Probably the most relevant work to this article, Thermostat [15] supports page migration at the granularity of 2MB or 4KB pages for a two-tiered hybrid memory system. Rainbow is different from Thermostat in two ways. First, to migrate small pages, Thermostat needs to splinter the corresponding superpages and then migrate the small pages. In contrast, Rainbow supports lightweight page migration without splintering superpages, and thus preserves the benefit of superpages on TLB performance. Second, Thermostat exploits an OS-level extension for intercepting TLB misses to estimate access counts at the 4KB page granularity. The software overhead is usually rather high, and thus Thermostat makes a tradeoff between the precision of hot page monitoring and the performance penalty. In contrast, the two-stage page access counting mechanism in Rainbow is more precise and efficient than Thermostat, and thus leads to lower page migration cost.

6 CONCLUSION

Superpages are able to significantly improve TLB coverage and reduce address translation overhead. Hybrid memory systems composed of DRAM and NVM usually can provide very large memory capacity, and thus are more eager for the support of superpages. However, superpages often preclude lightweight page migration, which is a key technique for improving performance and energy efficiency in hybrid memory systems. In this article, we propose a novel hybrid memory management mechanism named Rainbow to support both superpages and lightweight page migration. Rainbow manages the NVM at the superpage granularity and uses the DRAM to cache frequently accessed (hot) small pages in the superpages. Correspondingly, Rainbow utilizes split TLBs to support different page sizes. We propose an NVM-to-DRAM address remapping mechanism to identify the migrated small pages without splintering the superpages. Experiment results show that Rainbow can significantly reduce the address translation overhead for applications with large memory footprints and improve application performance by up to $2.9\times$ (45.3% on average) compared to a state-of-the-art memory migration policy without a superpage support.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

- [1] Micron Corp. and Intel Corp. 2015. 3D XPoint. Retrieved from <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [2] AMD Corp. 2011. AMD Family 10h and 12h Processors. Retrieved from <http://support.amd.com/TechDocs/40546.pdf>.
- [3] HP Labs. 2001. CACTI 3.0. Retrieved from <http://www.hpl.hp.com/research/cacti/>.

- [4] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, Jeremiah Willcock, Anton Korzh, and Marcin Zalewski. 2010. Graph500. Retrieved from <http://graph500.org/>.
- [5] Piotr Luszczek, David Bailey, Jack Dongarra, Jeremy Kepner, Robert Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. Retrieved from <http://icl.cs.utk.edu/hpcc/>.
- [6] Mel Gorman. 2010. Huge Pages Part 2 (Interfaces). Retrieved from <https://lwn.net/Articles/375096/>.
- [7] Intel Corp. 2015. Intel Skylake. Retrieved from <http://www.7-cpu.com/cpu/Skylake.html>.
- [8] Jack Dongarra, Jim Bunch, Cleve Moler, and Pete Stewart. 1984. Linpack. Retrieved from <http://www.netlib.org/benchmark/>.
- [9] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. The NAS Parallel Benchmarks. Retrieved from <https://www.nas.nasa.gov/publications/npb.html>.
- [10] Christian Bienia. 2011. Benchmarking Modern Multiprocessors. Retrieved from <http://parsec.cs.princeton.edu/index.htm>.
- [11] Julian Shun, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Aapo Kyrola, Kanat Tangwonsan, and Harsha Vardhan Simhadri. 2012. Problem Based Benchmark Suite. Retrieved from <http://www.cs.cmu.edu/~pbbs/>.
- [12] Salvatore Sanfilippo. 2015. Redis. Retrieved from <https://redis.io/>.
- [13] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006. Retrieved from <https://www.spec.org/cpu2006>.
- [14] WhiteDB team. 2013. WhiteDB. Retrieved from <http://whitedb.org/>.
- [15] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 631–644.
- [16] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 476–487.
- [17] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2015. Fast two-level address translation for virtualized systems. *IEEE Trans. Comput.* 64, 12 (Dec. 2015), 3461–3474.
- [18] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-it-yourself virtual memory translation. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 457–468.
- [19] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: Skip, Don't walk (the page table). In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 48–59.
- [20] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 307–318.
- [21] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 237–248.
- [22] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM, New York, NY, 26–35.
- [23] Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 383–394.
- [24] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Washington, DC, 62–63.
- [25] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 359–370.
- [26] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. 1989. Translation lookaside buffer consistency: A software approach. *SIGARCH Comput. Archit. News* 17, 2 (Apr. 1989), 113–122.
- [27] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mosse. 2016. Concurrent migration of multiple pages in software-managed hybrid main memory. In *Proceedings of the 34th IEEE International Conference on Computer Design (ICCD'16)*. IEEE Computer Society, Washington, DC, 420–423.
- [28] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE Computer Society, Washington, DC, 1–12.
- [29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 143–154.

- [30] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 435–448.
- [31] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Design Automation Conference (DAC'09)*. ACM, New York, NY, 664–469.
- [32] Xiaowan Dong, Sandhya Dwarkadas, and Alan L. Cox. 2016. Shared address translation revisited. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 18, 15 pages.
- [33] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. 2015. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE Computer Society, Washington, DC, 223–234.
- [34] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. 2012. DI-MMAP: A high performance memory-map runtime for data-intensive applications. In *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis (SC'12)*. IEEE Computer Society, Washington, DC, 731–735.
- [35] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE Computer Society, Washington, DC, 178–189.
- [36] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. ACM, New York, NY, 41–50.
- [37] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively breaking large pages to improve memory overcommitment performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*. ACM, New York, NY, 39–51.
- [38] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing memory in heterogeneous systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 637–650.
- [39] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*. IEEE Computer Society, Washington, DC, 1–12.
- [40] Li Jing and Lam Chung. 2011. Phase change memory. *Sci. China Inform. Sci.* 54, 5, Article 1061 (May 2011), 1061–1072 pages.
- [41] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 66–78.
- [42] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. 2016. Energy-efficient address translation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE Computer Society, Washington, DC, 631–643.
- [43] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using page walks to trigger page swaps in hybrid memory systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE Computer Society, Washington, DC.
- [44] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy translation coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 651–664.
- [45] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 705–721.
- [46] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 2–13.
- [47] Soyeon Lee, Hyokyung Bahn, and Sam H. Noh. 2014. CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Trans. Comput.* 63, 9 (Sept. 2014), 2187–2200.
- [48] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing (ICS'17)*. ACM, New York, NY, Article 26, 10 pages.
- [49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200.
- [50] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 89–104.

- [51] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 679–692.
- [52] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE Computer Society, Washington, DC, 210–222.
- [53] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. 2018. SEESAW: Using superpages to improve VIPT caches. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA'18)*. IEEE Press, Piscataway, NJ, 193–206.
- [54] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 444–456.
- [55] Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. 2011. Power management of hybrid DRAM/PRAM-based main memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, New York, NY, 59–64.
- [56] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE Computer Society, Washington, DC, 558–567.
- [57] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced large-reach TLBs. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, 258–269.
- [58] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, 1–12.
- [59] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 743–758.
- [60] Matthew Poremba, Tao Zhang, and Yuan Xie. 2015. NVMain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Comput. Archit. Lett.* 14, 2 (July 2015), 140–143.
- [61] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE Computer Society, Washington, DC, 568–578.
- [62] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 24–33.
- [63] Luiz E. Ramos, Eugene Gorbato, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 85–95.
- [64] 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. 2008. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* 52, 4 (Jul. 2008), 465–479.
- [65] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 176–187.
- [66] Reza Salkhordeh and Hossein Asadi. 2016. An operating system level data migration scheme in hybrid DRAM-NVM memory architecture. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE'16)*. EDA Consortium, San Jose, CA, 936–941.
- [67] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 475–486.
- [68] André Seznec. 2004. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Trans. Comput.* 53, 7 (July 2004), 924–927.
- [69] Seunghye Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling page table walks for irregular GPU applications. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA'18)*. IEEE Press, Piscataway, NJ, 180–192.
- [70] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for high performance address translation in chip multiprocessors. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE Computer Society, Washington, DC, 313–324.

- [71] Mark Swanson, Leigh Stoller, and John Carter. 1998. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA'98)*. IEEE Computer Society, Washington, DC, 204–213.
- [72] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. ACM, New York, NY, 171–182.
- [73] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA'92)*. ACM, New York, NY, 415–424.
- [74] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Washington, DC, 163–173.
- [75] Cui Xiaole, Cheng Zuolin, Lee Chunglen, Lin Xinnan, Wei Yiqun, Chen Xiaogang, and Song Zhitang. 2016. A snake addressing scheme for phase change memory testing. *Sci. China Inform. Sci.* 59, 10, Article 102401 (Feb. 2016), 11 pages.
- [76] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 430–443.
- [77] Idan Yaniv and Dan Tsafir. 2016. Hash, Don't cache (the page table). In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*. ACM, New York, NY, 337–350.
- [78] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and operating system support for reducing the impact of address translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, New York, NY, 159–168.

Received July 2018; revised December 2018; accepted January 2019