Superpage-Friendly Page Table Design for Hybrid Memory Systems

Xiaoyuan Wang[1,2,3,4], Haikun Liu[1,2,3,4](✉), Xiaofei Liao[1,2,3,4], and Hai Jin[1,2,3,4]

[1] National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan 430074, China
{xiaoyuanw, hkliu, xfliao, hjin}@hust.edu.cn

[2] Service Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China

[3] Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan 430074, China

[4] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Abstract. Page migration has long been adopted in hybrid memory systems comprising *dynamic random access memory* (DRAM) and *non-volatile memories* (NVMs), to improve the system performance and energy efficiency. However, page migration introduces some side effects, such as more *translation lookaside buffer* (TLB) misses, breaking memory contiguity, and extra memory accesses due to page table updating. In this paper, we propose superpage-friendly page table called SuperPT to reduce the performance overhead of serving TLB misses. By leveraging a virtual hashed page table and a hybrid DRAM allocator, SuperPT performs address translations in a flexible and efficient way while still remaining the contiguity within the migrated pages.

Keywords: Page table · Hybrid memory system · Page migration · Multiple page sizes · Address translation

1 Introduction

Recent years have witnessed many large-footprint applications. Traditional DRAM-based memory systems are unable to meet the ever-increasing memory demand due to the limited DRAM scaling in terms of memory density and power efficiency. The advent of *non-volatile memory* (NVM) technologies has attracted a lot of interests in constructing large-capacity and energy-efficient main memory systems with NVMs. However, since NVM cannot directly replace DRAM due to its shortcomings, such as lower performance and limited write endurance, hybrid memory systems composed of DRAM and NVM have been widely studied [1–4]. Most of these studies make efforts to improve system performance and save energy by using page migration [4, 5].

As the amount of memory required by applications increase significantly, the number of *page table entries* (PDEs) also grows rapidly. However, the capacity of *Translation Lookaside Buffer* (TLB) which is used to cache virtual-to-physical address

© Springer Nature Singapore Pte Ltd. 2020

J. Zeng et al. (Eds.): ICPCSEE 2020, CCIS 1257, pp. 623–641, 2020.
https://doi.org/10.1007/978-981-15-7981-3_46

translations cannot keep pace with the ever-increasing memory capacity due to access latency, energy consumption, and space constraints. The total address space that the TLBs can map directly, also known as TLB coverage, is far smaller than applications' footprint. In this scenario, the system performance is significantly degraded due to TLB misses. Previous studies show that the performance overhead is even up to 50% when running memory-hungry applications, i.e. applications with large footprints [6].

There have been a large body of studies on reducing the overheads serving TLB misses. These studies can be classified into two categories: one is to increase the TLB coverage (such as superpages, TLB coalescing, and range mapping), and another is to reduce the serving time of *page table walking* (PTW) which retrieves the *page table entries* (PTEs) to fill the TLBs upon TLB misses. Superpage [7–9] has long been used to reduce TLB misses. It can significantly improve the TLB coverage, e.g., a 2 MB superpage can enlarge the TLB coverage by 512 times when compared to a system in which memory pages are both managed and aligned in 4 KB. However, the using of superpage hinders lightweight page managements (such as page migration, page sharing) in hybrid memory systems [5]). TLB coalescing [10–12] and range mapping [13–15] are both practical and efficient schemes to improve TLB coverage. However, the frequent TLB updates due to page migrations limit the performance gain of these schemes.

Another direct and effective approach is to reduce the performance overhead of retrieving page tables. Since traditional binary-tree-based page table is very costly. For example, a TLB miss leads to four memory references in x86-64 architecture, and 24 memory references in virtualization environments [16–18]. Thus, it is essential to redesign the structure of page tables and the corresponding retrieving mechanism to reduce the cost of page table walking. *Hash page table* (HPT) leverages a hash function to map virtual addresses to physical ones in a constant time period. HPT significantly reduces the overhead of extra memory accesses caused by page table walking, at the expense of several undesirable advantages. For example, it is unable to support mixed page size and region mapping, and the cost of large page management is also extremely high. *Inverted page table* (IPT) is also developed to improve the physical-to-virtual address translation. By arranging an entry per memory page, IPT can significantly reduce the storage and the runtime overhead (such as searching), at the expense of lower performance of virtual-to-physical address translations. As a result, a hash function is usually used in IPT to speed up virtual-to-physical address translations, however, it is hard to support memory management at multiple page sizes.

We find that there is a remarkable contiguity in hot pages, which can be identified and migrated within a given monitoring period (10^8 cycles in our experiment) in hybrid memory systems. Previous works show that these contiguous pages can be leveraged by TLB coalescing [10–12]. In this paper, we study how to support fast virtual-to-physical and the reverse address translations while still remaining page contiguity between migrated pages. To accomplish this goal, several challenges should be addressed: 1) *identify page contiguity*, 2) *record page contiguity information in page tables, and* 3) *efficient virtual-to-physical and reserve address translations.*

To solve the above challenges, we propose superpage-friendly page table (called SuperPT), a novel page table design to support multi-grained page migrations in hybrid memory systems. SuperPT detects page contiguity within migrated pages and records them in a hash-based virtual page table. By leveraging this contiguity, multi-grained TLBs can deliver higher performance [10–12, 15]. What's more, even in systems without multi-grained TLB support, the system performance can also be improved by TLB prefetching [19, 20].

The remainder of this paper is organized as follows. Section 2 depicts the background and motivates of our design for multi-grained page migration in hybrid memory systems. Section 3 describes SuperPT designs in detail. Experimental results are presented in Sect. 4. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 Background and Motivation

We first introduce virtual memory and page tables. Next, we experimentally study memory access statistics of typical applications to motivate the design of SuperPT.

2.1 Virtual Memory and Page Table

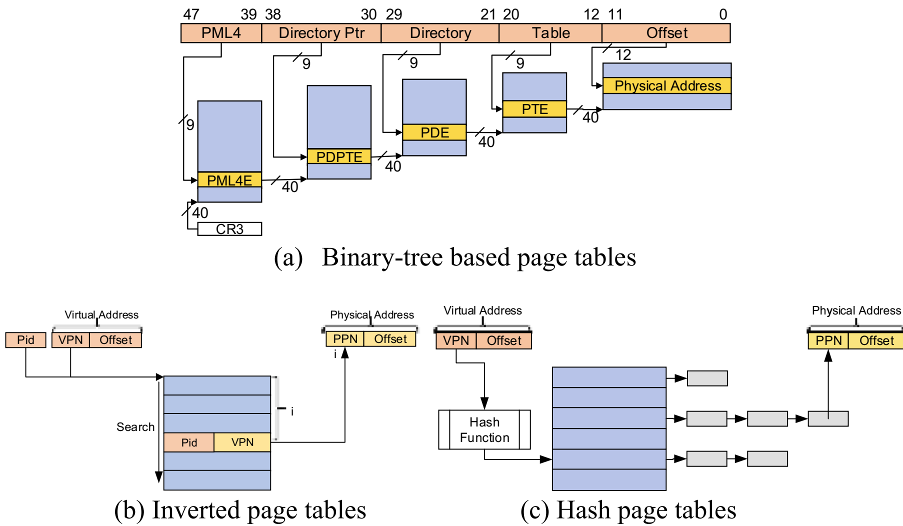
To extend the use of physical memory and enable memory protection, virtual memory is widely used in modern systems. There are two kinds of addresses in these systems, one for the virtual address space, and another for the physical or real address space. The virtual address is constructed by CPUs and used by processes, and the physical one is the real address space in memory systems. In order to accurately and conveniently perform the translation between virtual addresses and physical addresses, page tables are used to store and manage the virtual-to-physical address translations. To be specific, we classify most representative page table structures into the following categories.

Binary-Tree-Based Page Table. Figure 1 (a) shows the overview of binary-tree-based page table. Once a virtual-to-physical translation is required, the system looks up the corresponding page table entries layer by layer, thus four memory references are needed. In a virtualization environment, it even leads to 24 memory accesses per page table walking [16, 17, 21]. Therefore, this kind of page tables usually cause significant performance overhead [21]. Despite its high cost, it is widely adopted in modern computer systems, the reason is that this kind of page table is naturally friendly to cache locality because this mapping mechanism stores PTEs of adjacent pages in an adjacent manner.

Inverted Page Table. Figure 1 (b) shows the structure of *inverted page table* (IPT) [22]. IPT provides one-page table entry for each physical memory page. Each entry stores the *information of virtual address number (VPN)* and the corresponding *process ID* (PID). Thus, IPT is able to reduce the memory required to store the page tables since the number of IPT is equal to the number of physical memory pages. However, even when application memory requirement is low, the overhead of searching page tables is still very high upon a memory reference. Moreover, it also

poses other problems: ① Mixed page sizes are hard to support. ② Due to the lack of tree structure, operations on regions are extremely expensive.

Hashed Page Table. As shown in Fig. 1 (c), *VPN* is the input of the hashing function. Through hashing, an entry in the hash table is related to the VPN. If the first field of the three entries matches the *virtual page number* (*VPN*) of the desired page, we get the *physical page number* (*PPN*). If the *VPN* is not hit, we access the next entry in the linked list. This hash-function based page table design improves the efficiency of searching page tables, but it also brings some side effects. ① They are not able to support multiple page sizes. Since different pages are assigned to different table entries by the hash function, the page continuity is not guaranteed. ② They lead to poor cache locality and low performance of TLB prefetching mechanism. Because the address space is fragmented by hash function, the system cannot guarantee continuous response to the adjacent address space. In this case, both the page table cache and the TLB prefetching mechanism are inefficient. ③ Hash collisions are expensive. Since page table walking tends to be on the critical path of applications, page tables are more desired to be optimized for speed. Therefore, the system may suffer from high performance overhead due to hash collisions. If the *physical page number* (PPN) is not hit, the entries on the collision chain will be checked one by one, causing extremely high latency of page table retrieving.



(a) Binary-tree based page tables

(b) Inverted page tables (c) Hash page tables

Fig. 1. Layout of three kinds of page tables

2.2 Page Migration in Hybrid Memory Systems

Because NVM shows much lower energy consumption and higher density than DRAM, it has been studied by many works that prefer to use it to replace DRAM. However, NVM cannot directly replace DRAM for its shortcomings, such as limited

write endurance, high write energy consumption, and high access latency, especially for write operations [23]. Therefore, the heterogeneous memory system composed of DRAM and NVM has become a practical approach to the current dilemma. In order to improve the system performance of these systems, previous works generally leverage page migration to take advantages of the two storage medium, and overcome their shortcomings [2–4, 24, 25]. However, the use of page migration in heterogeneous memory systems brings additional performance overhead, because extra update operations are required for every page migration operation.

To evaluate the extra memory access times caused by page migration, we run several representative applications and profile their memory usage in an interval of 10^8 cycles. These applications are selected from SPEC CPU2006 [26], Parsec3.0 [27], NAS Parallel Benchmarks [28], and Graph500 [29]. CactusADM, Mcf, and Omnetpp are all chosen from SPEC CPU2006. Canneal, X264, Facesim, and streamcluster are all multi-thread applications that chosen from Parsec3.0. MG, UA, and SP are selected from NAS Parallel Benchmarks. Graph500 is designed to evaluate the performance of supercomputer by using large scale memory-intensive graph processing algorithms. All experiments are conducted in a simulated platform, as presented in Sect. 4.1. We have the following observations.

Observation 1: There is a large number of update operations on page tables caused by page migration in heterogeneous memory systems.

For each selected application, the extra memory access times caused by page migration is over 83% on average compare to the no-migration scenarios, as shown in Fig. 2. On one hand, for applications whose page table operations account for less memory access, the increment is more pronounced. For example, the memory access times in Canneal increases by more than two times. On the other hand, for applications whose page table operations account for most memory access, memory access growth is also considerable. For example, the memory access times in Omnetpp increases about five percent. Note that, page migration not only causes increase of page table operations, but also causes increase of total memory access times.

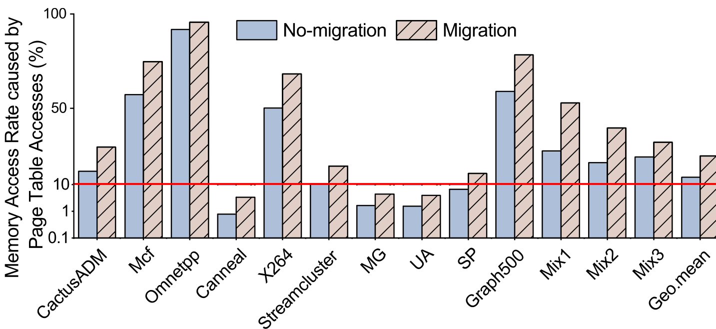


Fig. 2. Extra accesses caused by page migration in hybrid memory systems

Observation 2: There is a considerable continuity in migrated pages.

Figure 3 shows the *cumulative distribution function* (CDF) of the proportion of contiguous pages in the total number of migrated pages. For most applications, we find that almost 40% of the hot pages in the system are contiguous. For Graph500, the proportion of contiguous hot pages is over 90%. This implies that taking full advantage of the continuity of migrated pages can improve the efficiency of page table walking.

These above findings inspire us to conceive and model a new flexible and efficient memory management mechanism for hybrid memory systems that supporting multi-grained page sizes.

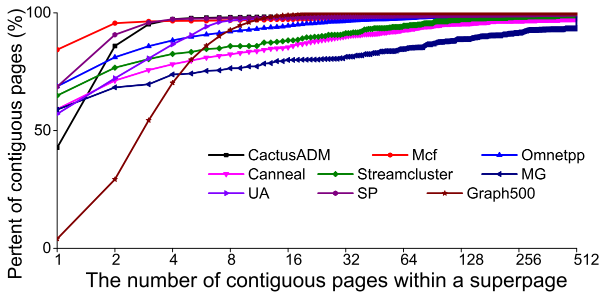


Fig. 3. Cumulative distribution function of contiguous hot pages within a superpage

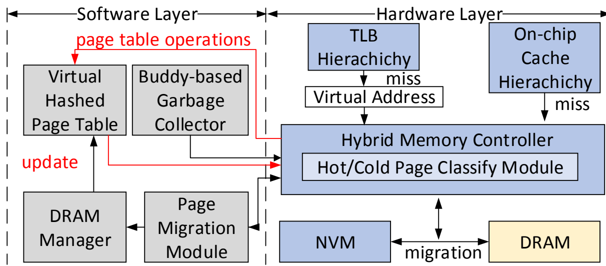


Fig. 4. Architecture of SuperPT

3 Design and Implementation

In this section, we first give an overview of SuperPT and then present the technical details of page table operations, hybrid memory allocator, buddy-based garbage collector, and memory fragmentation. At last, we describe some other implementation issues such as data consistency guarantee and page protection.

3.1 Architecture Overview

Figure 4 depicts the architecture of SuperPT. The hybrid memory controller counts and records the access information of each page in a given interval (10^8 cycles in our experiment). The hot/cold page classify module is added in the hybrid memory

controller, which analyses access counts of each page in the memory and selects the hot pages among them.

In the *operating system* (OS) level, we design a novel page table. Like HSCC [4], a DRAM cache filter with utility-based migration mechanism is adopted to improve the performance of DRAM cache. The DRAM cache manager module is responsible for page allocation and replacement. The migration module moves hot and cold pages between slow NVM and fast DRAM. To reduce the impact on system performance, SuperPT performs these modules periodically in the background.

Similar to CHOP [30], active pages in SuperPT are ranked according to the number of page accesses. We identify the top-*N* hot pages if the total accesses of them contribute 70% of the application's memory accesses in every period. When a NVM page is identified to be a hot page, it would be migrated from NVM medium to DRAM medium by the migration module. When the migration operation completes, the mapping of the page should be also updated.

3.2 Virtual Hashed Page Table

Figure 5 shows the structure of virtual hashed page table. The following terms are used: *virtual page number* (*VPN*), *virtual superpage number* (*VSN*), *index*, and *offset*. *Index* indicates the normal page number within a superpage, while *offset* indicates the real address within the normal page. When a virtual address comes, *VSN* is used as the key to find the location of superpage that the required page stays in. Note that, since the *virtual page number* is used as the key in the *Hash Function*, when a group of applications with the same or similar access patterns run together, the collision rate of the hash function increases significantly. To be more specifically, when two identical workloads running on the same machine, the hot pages they migrate will have the same virtual address and then be assigned to the same superpage by *Hash Function*, resulting in an increased collision rate.

To solve this challenge, the *process ID* (*PID*) is used as the key of the hash function along with the *virtual superpage number* (*VSN*). In this way, even the same application is allocated to different large pages, the use of PID avoids unnecessary hash conflicts.

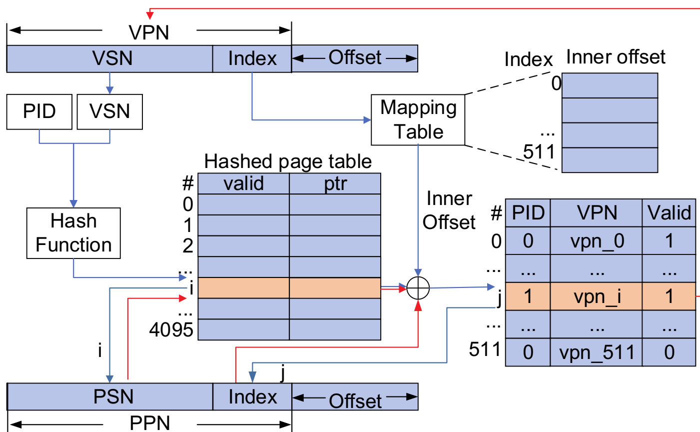


Fig. 5. Virtual hashed page table in SuperPT

3.3 Page Table Operations

Since physical address indexed cache is commonly used in modern computer systems, mapping virtual addresses to physical ones is on the critical path of applications. At this time, the system will first access TLBs. Upon a TLB miss, the page table walk operation is performed for virtual-to-physical address translation.

Lookup: ① Virtual-to-physical address translations. Upon a TLB miss, a hash query is done to find the corresponding superpage number according to the *virtual superpage numbe*r (VSN) and *process ID* (PID) by the *memory manage unit* (MMU). Meanwhile, the *index* is used as the key to lookup the *mapping tables* along with the value that the *hash function* gives. Note that, since *mapping table* and *hashed page table* are searched in parallel, the lookup overhead is acceptable. If the found hash table entry is valid, the *ptr* points to the starting address of the table that consists of 512 normal page table entries. If the entry is hit (the *valid* is true), the physical superpage address (*i*), the inner offset (*j*), and *offset* are used together to form the physical address. ② Physical-to-virtual address translation. Similarly, the physical address is also divided into three sections: *PSN*, *index,* and *offset*. By using *PSN* to search *hashed page table*, the starting address of normal page table is obtained, and then *index* is used to find the related internal normal page table of the large page to find the required *virtual page number* (*VPN*). At last, the complete virtual address is formed by combining it with the *offset*.

Insert: When a page or a set of pages are migrated from NVM to DRAM, the corresponding new page table entry is inserted into the *hash page table*. To be more particular, according to the virtual address information, the hash function is first used to find the physical superpage address. Second, the corresponding small page within the superpage is allocated to store the corresponding data. Finally, the corresponding hash page table, internal page table, and mapping table are updated.

Update: When there is insufficient free small pages within a superpage, or when the available free DRAM size is too small, the page write-back operation is triggered. Correspondingly, the content of *hash page table* also needs to be updated along with the page write-back. Specifically, SuperPT will find the corresponding physical superpage according to the *VSN*, and the corresponding small page is located through the *index* field. At last, the page is replaced through a LRU algorithm, and written back to NVM. Note that, SuperPT significantly reduces the address translation latency due to its good performance in physical-to-virtual address translation.

Delete: When a process finishes (normally or unexpectedly), its memory space is reclaimed and the relevant page table entries should be invalidated. SuperPT uses a lazy invalidation mechanism—reclaiming those pages when they are reallocated or during garbage collection. Specifically, the *PID* of the relevant page is checked at the time of memory allocation or garbage collection, and if its *PID* exists in the invalidation list, the relevant page table entry is invalidated. It is well known that in x86-64 platform, Linux uses 22 bits to identify process numbers (up to 2^{22} processes can be identified at the same time), and when the number of processes exceeds this threshold, the system reuses *PIDs* of destroyed processes for new PID allocation.

Page Sharing: Sometimes there may be a part of main memory that shared by more than one process. In this case, a single page table entry can be mapped to at least two virtual pages, and then SuperPT leverages a link pointer to bind the information of those virtual pages to the root page table.

Page Protection: Like traditional page table entry, the PTEs in SuperPT contain physical address and other various flags. The *present* bit reflects whether the page is already in memory or not. The *writable* bit reflects whether the page is allowed to write. The *user accessible* bit reflects whether the page can be accessed by the user mode code. The *write through caching* bit reflects whether the writes can be directly passed to the main memory. The *disable cache* bit reflects whether the page can be cached. The *accessed* bit reflects whether the page has been accessed yet, i.e., when the page is used, the CPU sets this bit. When a page is written, the *dirty* bit is set by the CPU. The *global* bit reflects whether the page can be flushed from caches or not when the content is switched. The *available* bit reflects if the page can be used freely by the OS or not. If the *no execute* bit is set, the system forbids executing code on this page. Since SuperPT supports range mapping, the *huge page/null* is overridden to reflect this is a regular page or a range page mapping.

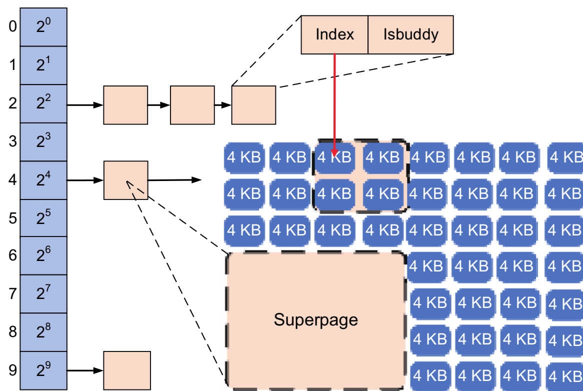


Fig. 6. Buddy-based garbage collator in SuperPT

3.4 DRAM Allocator

When a group of pages are identified as hot pages, the page migration module will migrate them to DRAM. With the virtual address and process number, the system first uses *Hash Function* to find the starting address of the corresponding superpage, and then carries out the corresponding normal page allocation operation. Finally, the system updates the corresponding page table entries.

Since SuperPT supports multi-granularity page migration and multi-granularity page mapping, at the end of each page monitoring period, a simple sort is made to merge the migration requests of small pages with adjacent addresses, making sure that the adjacent pages remain adjacently after migration as much as possible. However,

when a set of consecutive pages are migrated together, there may not be enough free pages for them. At this point, SuperPT splits these pages to fit the available memory space. This sacrifices some page continuity but has less impact on system performance.

3.5 Buddy Based Garbage Collection

To reduce external fragmentation, we leverage a buddy-system based garbage collector. It merges adjacent free blocks by keeping track of its neighbors. In particular, a bitmap is used to track whether neighbors are in use. What's more, since the memory blocks are aligned by power-of-two, they could be merged to construct a double-sized block.

As shown in Fig. 6, the space within a superpage is divided into 10 groups, each of which is a collection of exponential successive pages. For example, each element in group i represents a set of 2^i consecutive pages. In addition, for the convenience of management and statistics, any element in each group only records the first page address of the contiguous page addresses within the superpage.

3.6 Data Consistency

Page migrations may raise data inconsistency problems. We address those issues as follows.

Data Consistency Between DRAM and NVM. As mentioned before, there may be two replicas of the migrate pages, one is in DRAM and the other is resided in NVM. Conversely, the operation that mapping a virtual address to physical ones may be performed by both normal page table used for NVM and *Hashed page table* used for DRAM. To guarantee the consistency of the data, we extend the normal page table with a migration flag (*M*) by using reserved bit of PTE, identifying whether a page is migrated or not. Since the normal page table and *hashed page table* are searched in parallel, the more efficient hashed page table is always returned first if the page being searched is in DRAM, so the system always gives high priority to data that has been cached in DRAM. Even if the normal page table returns data first, the page migration status can be determined based on the migration flag bit in the PTE to prevent accessing to the wrong data.

Cache Consistency. To achieve higher performance, a large number of modern processors use write-back cache solutions, in which the data modify operations (i.e. writes) are directed to the cache without informing the main memory about these modifications. In this way, the memory is finally modified only when the cache is evicted. Since a set of cache lines may be mapped by a single page, once page migration occurs before all the cache lines are written back to memory, the stale data may be chosen for migration, resulting in data inconsistence problems. *Clflush* instructions is used in SuperPT to solve this problem. When pages are migrated, all cache lines corresponding to these pages are invalidated. Meanwhile, the invalidation operation would be broadcast to other parts in the same cache consistency domain. As a result, the relevant cache lines throughout the cache hierarchy is either be invalidated (clean pages) or written back (dirty pages). Finally, before a page is migrated, SuperPT writes back all the related dirty cache lines to the memory and invalidates all the corresponding clean cache lines.

4 Evaluation

4.1 Experimental Methodology

We implement SuperPT in a full-system simulator by integrating Zsim [31] and NVMain [33]. Zsim is based on Pin tools [32], and we leverage it to simulate on-chip systems, because it is fast and supports x86-64 multi-core and many-core architectures well. Meanwhile, we also add many OS-level functions to Zsim, such as memory allocator, *memory management unit* (MMU) for TLB, and page table simulation. As a widely studied cycle-accurate memory simulator, NVMain is used in SuperPT to simulate the hybrid memory system composed of DRAM and NVM in detail.

Table 1. System configuration of simulated platform

CPU	8 cores, 3.2 GHz, out-of-order	
TLB hierarchy	L1 DTLB	32 entries for 2 MB superpages, 64 entries for 4 KB small pages in each core, 4-way set-associate, 1 cycle per access
	L2 DTLB	512 entries for 2 MB superpages, 1024 entries for 4 KB pages, both can be used for Data and Instruction. 8-way set-associate, 8 cycles per access
Cache hierarchy	L1 Cache	Private 64 KB in each core, 4-way, split Data and Instruction, 3-cycles per access
	L2 Cache	Private 256 KB in each core, 8-way, set associate, 10-cycles per access
	L3 Cache	Shared 8 MB, 16-way set-associate, 34 cycles per access
DRAM	4 GB, channel-rank-bank-row-col: 1-4-32-32768-64, FR-FCFS, Bandwidth (GB/Sec): 10.7, Timing (cycles): (cas-red-rp-ras: 7-7-7-18), Read Delay (ns): 13.5, Write Delay (ns): 28.5	
PCM	32 GB, channel-rank-bank-row-col: 4-8-64-65536-64, FR-FCFS, Bandwidth (GB/Sec): 10.7, Timing (cycles): (cas-red-rp-ras: 9-37-100-53), Read Delay (ns): 13.5, Write Delay (ns): 171	

Configuration. Table 1 depicted the experimental platform and detailed configuration. We choose PCM as the representative storage medium of memory for it has been widely studied. The timing parameters are referred to previous works [4, 5, 34]. In addition, the latencies of manage mechanisms that associated with data consistency, such as *Clfulsh* and data migration are modeled in detail based on the timing parameters of on-chip systems and memory.

Table 2. Workloads for evaluation

Workloads	Applications
SPEC CPU 2006	CactusADM x8, Mcf x8, Omnetpp x8
Parsec 3.0	Canneal x8, X264 x8, Streamcluster x8
NPB	MG x8, UA x8, SP x8
Large footprints	Graph500 x8, GUPS x8
Mix1	CactusADM x2 + Mcf x2 + Canneal x2 + Omnetpp x2
Mix2	Canneal x2 + X264 x2 + SP x2 + GUPS x2
Mix3	Mcf x2 + X264 x2 + SP x2 + UA x2

Alternative Policies. To better evaluate the system performance, SuperPT is compared with several alternative page migration mechanisms for heterogeneous memories as follows.

①**Flat-static:** 4 GB DRAM and 32 GB NVM are both used as main memory in the same address space [4], and are managed at 4 KB granularity. Based on the capacity ratio of DRAM to NVM, the data is evenly distributed in the address space. Since the fast DRAM and slow NVM are used indiscriminately in this system, there is no page migration between DRAM and NVM. This system is used as a baseline for comparison.

②**HSCC:** This is a state-of-the-art hybrid main memory system with traditional four-level page tables [4]. HSCC leverages a utility-based page migration strategy to migrate hot pages between fast DRAM and slow NVM.

③**DRAM:** This is a memory system consisting of only 32 GB DRAM, which is used as the applications' performance upper bound.

Benchmarks. As shown in Table 2, we choose several representative applications from SPEC CPU2006 [26], Parsec [27], NAS Parallel Benchmarks [28], and Graph500 [29]. CactusADM, Mcf, and Omnetpp are chosen from SPEC CPU2006. Among them, CactusADM is designed as a computational kernel to represent many programs in numerical relativity. Mcf is designed to solve a scheduling problem in public mass transportation. Omnetpp is implemented to simulate discrete event of large Ethernet networks. Canneal, X264, and Streamcluster are all multiple thread workloads chosen from Parsec3.0. MG, UA, and SP are chosen from NAS Parallel Benchmarks. MG is a simple multiple-grid kernel that needs highly structured communication at long distance and used to evaluate data communication both for short and long distance. UA solves heat equation with convection and diffusion from moving ball. SP leverages scalar penta-diagonal to solve nonlinear PDEs problems. Graph500 is designed to evaluate the performance of supercomputer by using large scale memory-intensive graph processing algorithms. To verify the effectiveness of the system in running the same application, we ran eight instances for each application at the same time.

4.2 Extra Memory Accesses Time

Figure 7 shows the *memory access times* (MAT) caused by page migration of each workload in our experiment, and all the result are normalized to the baseline system (Flat-static). SuperPT significantly reduces the MAT of those applications with lower data locality and large footprint. For example, for Canneal and GUPS, SuperPT can reduce 24.3% and 23.2% access times, respectively. Meanwhile, the access counts of mixed workloads with different applications, such as Mix1, Mix2, and Mix3, are also significantly reduced. Compared to HSCC, SuperPT reduces 19.3% memory accesses on average. This indicates that SuperPT significantly reduces the extra memory accesses due to page migrations. Moreover, the well page contiguity deliver higher performance to the page table walker cache.

4.3 Application Performance

Figure 8 shows the *instructions per cycle* (IPC) of every workload, all normalized to the Flat-static system. For applications with poor locality and large footprint (such as Canneal and GUPS), SuperPT can significantly improve system performance. We also notice that for highly parallel applications, such as MG, UA, and SP, although SuperPT reduced the proportion of memory access times by a considerable amount, the performance improvement of SuperPT on such applications is not very significant due to the small proportion of memory access caused by page table access (Fig. 2). Overall, SuperPT improves system performance by 77.9% and 9.5% on average, compared to Flat-static and HSCC, respectively. The performance gap between SuperPT and the upper bound (DRAM) is only 6.8% on average.

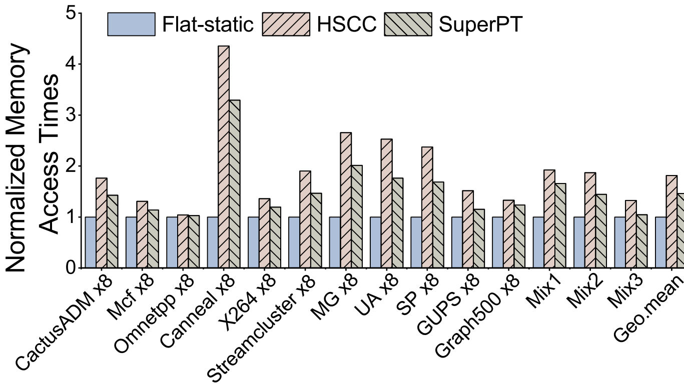


Fig. 7. Normalized memory access times relative to the flat system

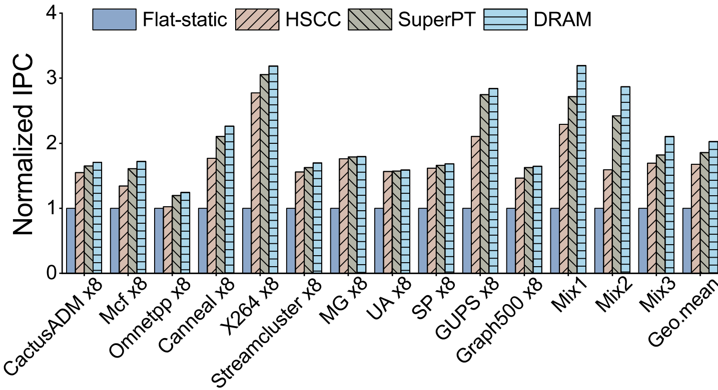


Fig. 8. Normalized IPC relative to the flat system

4.4 DRAM Allocation Collision

Figure 9 shows the hash conflict rate caused by DRAM allocation in SuperPT. To evaluate the performance of hash function in SuperPT, we compare SuperPT with several open-source hash functions, such as VHPT-remainder[1], VHPT-wyhash[2]. We have the following observations: ① SuperPT brings a very low conflict rate (less than 0.1% on average), especially for applications with small memory footprint, such as CactusADM. ② More efficient hash algorithms can significantly reduce the conflict rate, such as SuperPT-wyhash, which reduces the conflict rate by 84.4% on average, compared to SuperPT-remainder. Since a higher conflict rate is also associated with higher performance overhead, SuperPT adopts the hash function of SuperPT-wyhash in order to reduce system overhead.

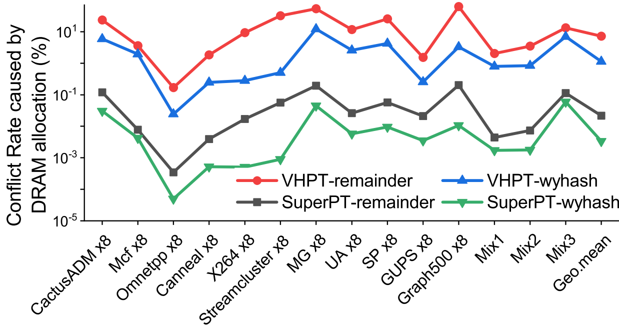


Fig. 9. Conflict rate of DRAM allocation

[1] Remainder is a simple hash function design with remainder operation.

[2] Wyhash [35] is a fast hash function on x86-64 without quality problems.

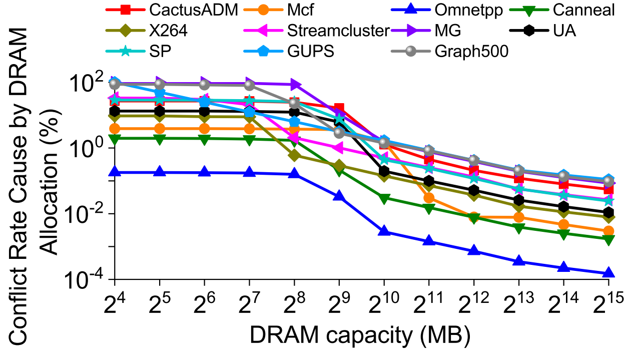


Fig. 10. DRAM collision rate sensitive to different DRAM size

To further study how the hash conflicting rate in SuperPT is sensitive to DRAM size, we run selected workloads with different DRAM capacity. As shown in Fig. 10, the conflict rate is calculated as the ratio of the number of conflicts to the total number of allocated pages. We find that, the conflict rate in SuperPT is less than 1% for most evaluated applications as long as the DRAM capacity is large than 4 GB. Moreover, the larger memory capacity leads to a lower collision rate.

4.5 Storage and Runtime Overheads

In common x86-64 systems, 46 bits are used for physical address and 48 bits for virtual address. Since 20 bits are needed for identifying the offset inner a superpage, the *virtual superpage number* (VSN) uses 28 bits. Similarly, 26 bits are needed for *physical superpage number* (PSN). Thus, to store the *virtual hashed page table* (VHPT), SuperPT consumes $\frac{4096\times(26+16)}{8\times1024} = 13.5\,\text{KB}$. The mapping table uses $\frac{9\times512}{8\times1024} = 0.56\,\text{KB}$. Meanwhile, to index the normal pages in the superpage, $\frac{(22+34+1)\times512\times4096}{8\times1024\times1024} = 14.25\,\text{MB}$ are needed to store the mappings between inner small pages and superpages. In all, the total memory usage is 14.25 MB. This storage overhead is negligible in modern computers which usually have several terabytes memory.

Figure 11 shows the breakdown of performance overhead caused by *page table walking* (PTW), DRAM mapping, Allocate pages, Confliction, Clflush, and TLB shootdown. All these operations are modeled by adding reasonable latencies in our simulator. We find that the runtime overhead of these selected programs varied greatly. For applications with large footprint and good hot page contiguity, such as MG, SP, mix2, and mix3, conflicts accounts for a larger partition of runtime overhead. In summary, the runtime performance overhead of SuperPT is 3.7% on average. This overhead is acceptable given the large performance benefits of employing multi-grained pages and efficient page table policy.

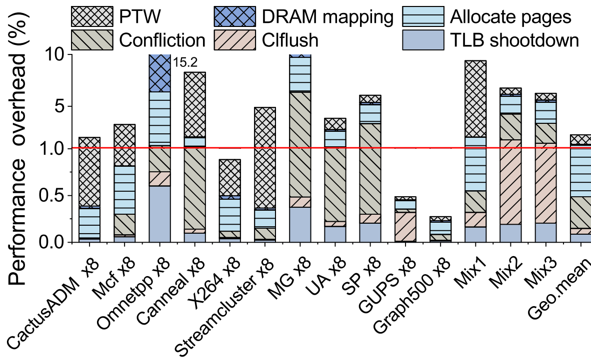


Fig. 11. Breakdown of running time overhead in SuperPT

5 Related Work

Page Contiguity. There has been a number of studies on exploring page contiguity to improve the TLB performance. Gorman *et al*. [36] design a new memory allocator in the operating system to mitigate memory fragmentation and promotes contiguity by aggregating pages based on the relocation information of them. Libhugetlbfs [37] exposes a user interface to programs to explicitly use huge pages. GLUE [12] leverages a single speculative superpage translation in the TLBs to map contiguous small pages. To reduce the overhead of page table walk, previous works [38, 39] allow the verification of speculative translations off the critical path of program execution. GTSM [40] leverages page contiguity at the hardware layer to construct superpages, even in system with retired bits. A large number of previous works focus on leveraging page contiguity to reduce TLB misses and to mitigate memory fragmentation (e.g., internal and external memory fragmentations). By using memory compaction, *contiguous memory allocators* (CMAs) [41] migrate memory fragmentation and offer a large contiguous memory space. By aggressively merging discrete physical frames into contiguous regions, Translation Ranger [14] enlarges the reach of contiguity-aware TLBs.

Contiguity-Aware TLBs. To leverage the page contiguity, TLB coalescing [10, 11, 42] and *memory management unit* (MMU) cache coalescing [6] are designed to increase the coverage of TLBs and MMU caches. To further enlarge TLB reach to cover the modern gigabyte-to-terabyte physical memory, direct segments [43] leverage a large segment to do fast address mapping between contiguous virtual address region and contiguous physical address region. *Redundant memory mappings* (RMM) [15] significantly enlarge TLB coverage by using range TLBs to map regions that both contiguous in virtual and physical addresses.

6 Conclusion

Energy efficiency in heterogeneous memory systems made up of DRAM and NVMs. However, page migration brings some side effects to the system, such as extra memory access due to page table modification and being hard to remain the contiguity of pages. In this paper, we propose a Superpage-friendly Page Table called SuperPT to reduce the overheads serving TLB misses. By leveraging a virtual hashed page table and a hybrid DRAM allocator, SuperPT conducts the address translation in a flexible and efficient way while rarely destroy the contiguity within the migrate pages. Experimental results show that SuperPT significantly reduces memory access times by 19.3% on average and thus improves system performance by 9.5% on average.

References

1. Dhiman, G., Ayoub, R., Rosing, T.: PDRAM: a hybrid pram and dram main memory system. In: Proceedings of the 46th Annual Design Automation Conference, pp. 664–469. ACM, New York (2009)
2. Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, pp. 24–33. ACM, New York (2009)
3. Ramos, L.E., Gorbatov, E., Bianchini, R.: Page placement in hybrid memory systems. In: Proceedings of the International Conference on Supercomputing, pp. 85–95. ACM, New York (2011)
4. Liu, H., et al.: Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In: Proceedings of the International Conference on Supercomputing, pp. 26:1–26:10. ACM, New York (2017)
5. Wang, X., et al.: Supporting superpages and lightweight page migration in hybrid memory systems. ACM Trans. Archit. Code Optim. **16**(2), 11:1–11:26 (2019)
6. Bhattacharjee, A.: Large-reach memory management unit caches. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 383–394. ACM, New York (2013)
7. Romer, T.H., Ohlrich, W.H., Karlin, A.R., Bershad, B.N.: Reducing TLB and memory overhead using online superpage promotion. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 176–187. ACM, New York (1995)
8. Talluri, M., Hill, M.D.: Surpassing the TLB performance of superpages with less operating system support. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 171–182. ACM, New York (1994)
9. Swanson, M., Stoller, L., Carter, J.: Increasing TLB reach using superpages backed by shadow memory. In: Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 204–213. IEEE Computer Society, Washington, DC (1998)
10. Pham, B., Vaidyanathan, V., Jaleel, A., Bhattacharjee, A.: Colt: coalesced large-reach TLBs. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 258–269. IEEE Computer Society, Washington, DC (2012)

11. Pham, B., Bhattacharjee, A., Eckert, Y., Loh, G.H.: Increasing TLB reach by exploiting clustering in page translations. In: Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture, pp. 558–567. IEEE Computer Society, Washington, DC (2014)
12. Pham, B., Veselý, J., Loh, G.H., Bhattacharjee, A.: Large pages and lightweight memory management in virtualized environments: can you have it both ways? In: Proceedings of the 48th International Symposium on Microarchitecture, pp. 1–12. ACM, New York (2015)
13. Gandhi, J., et al.: Range translations for fast virtual memory. IEEE Micro **36**(3), 118–126 (2016)
14. Yan, Z., Lustig, D., Nellans, D., Bhattacharjee, A.: Translation ranger: operating system support for contiguity-aware TLBs. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 698–710. ACM, New York (2019)
15. Karakostas, V., et al.: Redundant memory mappings for fast access to large memories. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp. 66–78. ACM, New York (2015)
16. Bhargava, R., Serebrin, B., Spadini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 26–35. ACM, New York (2008)
17. Gandhi, J., Basu, A., Hill, M.D., Swift, M.M.: Efficient memory virtualization: reducing dimensionality of nested page walks. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 178–189. IEEE Computer Society, Washington, DC (2014)
18. Yan, Z., Veselý, J., Cox, G., Bhattacharjee, A.: Hardware translation coherence for virtualized systems. In: Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture, pp. 430–443. ACM, New York (2017)
19. Kandiraju, G.B., Sivasubramaniam, A.: Going the distance for TLB prefetching: an application-driven study. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 195–206. IEEE, Anchorage (2002)
20. Saulsbury, A., Dahlgren, F., Stenström, P.: Recency-based TLB preloading, In: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 117–127. ACM, New York (2000)
21. Yaniv, I., Tsafrir, D.: Hash, don't cache (the page table). In: Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, pp. 337–350. ACM, New York (2016)
22. Stallings, W.: Operating Systems: Internals and Design Principles, 7th edn. Pearson/Prentice Hall, Upper Saddle River (2011)
23. Raoux, S., et al.: Phase-change random access memory: a scalable technology. IBM J. Res. Dev. **52**(4.5), 465–479 (2008)
24. Park, H., Yoo, S., Lee, S.: Power management of hybrid DRAM/PRAM-based main memory. In: Proceedings of the 48th Design Automation Conference, pp. 59–64. ACM, New York (2011)
25. Wei, W., Jiang, D., McKee, S.A., Xiong, J., Chen, M.: Exploiting program semantics to place data in hybrid memory. In: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation, pp. 163–173. IEEE Computer Society, Washington, DC (2015)
26. SPEC CPU2006. https://www.spec.org/cpu2006. Last Accessed 21 Nov 2019
27. Parsec. http://parsec.cs.princeton.edu/index.htm. Last Accessed 21 Nov 2019
28. Bailey, D., et al.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)

29. Graph500. http://graph500.org/. Last Accessed 21 Nov 2019
30. Jiang, X., et al.: CHOP: adaptive filter-based DRAM caching for CMP server platforms. In: Proceedings of the Sixteenth International Symposium on High-Performance Computer Architecture, pp. 1–12. IEEE Computer Society, Washington, DC (2010)
31. Sanchez, D., Kozyrakis, C.: ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, pp. 475–486. ACM, New York (2013)
32. Luk, C.K., et al.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200. ACM, New York (2005)
33. Poremba, M., Zhang, T., Xie, Y.: NVMain 2.0: a user-friendly memory simulator to model (non-)volatile memory systems. IEEE Comput. Archit. Lett. **14**(2), 140–143 (2015)
34. Lee, B.C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable DRAM alternative. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, pp. 2–13. ACM, New York (2009)
35. Wyhash. https://github.com/rurban/smhasher. Last Accessed 21 Nov 2019
36. Gorman, M., Healy, P.: Supporting superpage allocation without additional hardware support. In: Proceedings of the 7th International Symposium on Memory Management, pp. 41–50. ACM, New York (2008)
37. Huge Pages Part 2 (Interfaces). https://lwn.net/Articles/375096/. Last Accessed 21 Nov 2019
38. Barr, T.W., Cox, A.L., Rixner, S.: SpecTLB: a mechanism for speculative address translation. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, pp. 307–318. ACM, New York (2011)
39. Papadopoulou, M.M., Tong, X., Seznec, A., Moshovos, A.: Prediction-based superpage-friendly TLB designs. In: Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture, pp. 210–222. IEEE Computer Society, Washington, DC (2015)
40. Du, Y., Zhou, M., Childers, B.R., Mossé, D., Melhem, R.: Supporting superpages in non-contiguous physical memory. In: Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture, pp. 223–234. IEEE Computer Society, Washington, DC (2015)
41. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers: Where the Kernel Meets the Hardware. 3rd edn. O'Reilly Media, Sebastopol (2005)
42. Wang, X., Liu, H., Liao, X., Jin, H., Zhang, Y.: TLB coalescing for multi-grained page migration in hybrid memory systems. IEEE Access **8**, 66304–66314 (2020)
43. Basu, A., Gandhi, J., Chang, J., Hill, M.D., Swift, M.M.: Efficient virtual memory for big memory servers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, pp. 237–248. ACM, New York (2013)