PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems

Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

***Abstract*—**
Hybrid main memories composed of DRAM and Non-Volatile Memory (NVM) combine the capacity benefits of NVM with the low-latency properties of DRAM. For highest performance, data segments should be exchanged between the two types of memories dynamically—a process known as segment *swapping*—based on the access patterns to the segments in the program. The key difficulty in hardware-managed swapping is to identify the appropriate segments to swap between the memories at the right time in the execution.

To perform hardware-managed segment swapping both accurately and with substantial lead time, this paper proposes to use hints from the page walk in a TLB miss. We call the scheme *PageSeer*. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to a subsequently-referenced page (i.e., its follower page), to potentially initiate swaps for the page and for its follower. We call these actions *MMU-Triggered Prefetch Swaps*. PageSeer also initiates other types of page swaps, building a complete solution for hybrid memory. Our evaluation of PageSeer with simulations of 26 workloads shows that PageSeer effectively hides the swap overhead and services many requests from the DRAM. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improves performance by 19% and reduces the average main memory access time by 29%.

***Keywords*-Hybrid Memory Systems; Non-Volatile Memory; Virtual Memory; Page Walks; Page Swapping.**

I. Introduction

Data-intensive applications demand large memory capacity and high bandwidth with low power consumption. While DRAM has been used as main memory for decades due to its relatively low latency and energy consumption, it is suffering from device scalability problems [1]. As a result, new memory solutions are required.

Non-Volatile Memory (NVM) technologies, such as PCM [2] and STT-RAM [3], show promise to satisfy the increasing memory capacity demands of workloads. These memories can attain high capacity at low cost [2], [4]. Memory vendors have announced the production of NVMs [5], and impending systems will include them.

Unfortunately, NVMs have drawbacks and cannot simply replace DRAM in their current form. Compared to DRAM, read and write accesses in NVMs have higher latencies and consume more energy. As a result, upcoming systems are likely to incorporate a hybrid main memory system composed of both DRAM and NVM.

The main challenge in a hybrid memory system is how to exploit the capacity benefits of NVM, while benefiting from the low latency of DRAM. Previous research has either used DRAM as a cache for the NVM [6] or used a flat address space configuration with both memories [7]. The latter organization provides both higher aggregate bandwidth and higher memory capacity. However, one needs to decide in what memory to place specific data structures.

Optimal placement of data structures in a hybrid memory system is hard to attain statically, as applications exhibit dynamic changes of behavior. It is better to dynamically move data segments between the two types of memory (i.e., to *swap* segments between the memories) based on the access patterns. Such a movement can be software- or hardware-managed.

Hardware-managed swap techniques [7]–[9] are generally preferred over software-managed ones because they induce much lower overhead. However, they require special hardware structures to track memory access activity, perform the data swaps, and record the remappings of segments between the memories. The efficient management of the relevant metadata is crucial for performance.

The key difficulty in hardware-managed swap techniques is to identify the appropriate segments to swap at the appropriate time in the execution. Aggressive schemes that move a segment to DRAM upon the first access to it introduce unnecessary traffic if the segment is not accessed much more after the swap. Alternatively, schemes that require a history of many accesses to the segment before moving it to DRAM may react too slowly and hence not improve performance.

To perform hardware-managed segment swapping both accurately and with substantial lead time, this paper proposes to use hints from the page walk in a TLB miss. We call the scheme *PageSeer*. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to a subsequently-referenced page (i.e., its *follower* page), to potentially initiate swaps for the page and for its follower. We call these actions *MMU-Triggered Prefetch Swaps*. They are transparent to the software. PageSeer also initiates other types of page swaps, building a complete solution for hybrid memory.

We evaluate PageSeer using simulations of 26 workloads. Our results show that PageSeer effectively hides the swap overhead, and services many requests from the fast memory. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improves performance by 19% and reduces the average main memory access time by 29%. Further, MMU-triggered prefetch swaps accurately predict future memory accesses. Overall, PageSeer efficiently manages a hybrid memory system.

II. Background & Motivation

A. Hybrid Memories

Emerging NVM technologies, such as 3D XPoint [5] and Phase Change Memory [2], have recently gained a lot of attention. The high bit density, low static power, and non-volatile aspects of these memories appear as a viable solution to the increasing memory demands of workloads and the slowdown of DRAM scaling. However, NVMs exhibit higher latencies than DRAM, and therefore cannot replace DRAM entirely without performance loss. For this reason, the combination of DRAM and NVM has been proposed as a method to efficiently increase system capacity, performance, and reliability [6]. A memory system that integrates both DRAM and NVM is typically called a hybrid memory system.

A hybrid memory system can be configured in one of two ways. In one configuration, the faster and smaller DRAM is a hardware-managed cache for the slower and larger NVM [10]–[15]. In the other, the DRAM and NVM are configured as a flat address space, where the OS is aware of both memories for page allocation [7]–[9], [16]–[18]. The first configuration has the advantage that it can be easily deployed and is transparent to the OS, with DRAM acting as an additional level of caching between the Last Level Cache (LLC) and main memory. However, it faces the challenge of efficiently storing and accessing a large amount of tags [11], [19]. Moreover, the overall capacity of the system decreases by a non-negligible amount, as long as the sizes of DRAM and NVM are comparable. Although previous work has shown performance improvements for latency critical applications [10], [13], capacity-limited applications do not benefit as much [16]. Also, the overall memory bandwidth is limited, since we cannot take advantage of the combined bandwidth of the two memories.

The flat address space configuration has the advantage that it provides both higher aggregate bandwidth and higher memory capacity. Moreover, there is no need for tag storage. However, it is challenging to decide the data placement and swapping of data between the two memories.

Data swaps between the two memories can be done either in software [18], [20], [21] or in hardware [7]–[9], [16], [17]. In both cases, we must identify data that are "hot" (i.e., accessed frequently) but reside in the slow memory, and swap them with data that are "cold" but reside in the fast memory. In a software-managed approach, the OS interrupts the processor, swaps the pages, performs a TLB shootdown to purge stale TLB entries, and continues execution. This procedure can take several microseconds [22], and constrains swaps to a coarse time granularity.

When data swaps are hardware-managed, they can happen at finer time granularity. However, there are several challenges in this method. The first one concerns the consistency between the OS view of memory and the data movements that the hardware has performed. Since the OS is not aware of any remapping, the hardware needs to keep track of the data remappings that have occurred. Second, we need dedicated hardware to decide when and what swaps to perform between fast and slow memory. This means that the hardware should be able to track memory activity, and trigger a swap accurately and promptly to tolerate the swap cost.

B. Hardware-Based Memory Management Techniques

Previous work has investigated hardware-only techniques for managing hybrid memory systems. Typically, these techniques rely on LLC misses to track main memory activity and determine data swaps between DRAM and NVM. The techniques differ in the size of the memory segments to swap and what triggers a swap.

Suppose that we perform a segment swap between the slow and the fast memory. Then, if a second segment from the slow memory needs to be moved to the exact same location in fast memory as the first one, it can do so with a *Slow* or a *Fast* swap. In a slow swap, the first swap is undone and then the second swap is performed. In a fast swap, only one swap is performed, exchanging the second segment from the slow memory with the segment currently in fast memory (which used to be in slow memory).

CAMEO [16] migrates data in 64B blocks, and a swap is triggered on every access to a block in slow memory. CAMEO restricts the swap flexibility by allowing a set of slow-memory blocks (which form a Swap Group) to be swapped only with a single block of fast memory. Also, only one of these slow-memory blocks can be in fast memory at a time. Further, a slow-memory block can reside anywhere within the slow-memory area assigned to its swap group. CAMEO uses fast swaps. While CAMEO keeps the required swap bandwidth low and is easy to implement, the small swap granularity requires substantial meta-data storage and misses the opportunity to take advantage of spatial locality. Moreover, the direct mapping of the swap groups to single fast-memory blocks may cause conflict misses.

PoM [7], [23] is similar to CAMEO, with the difference that swaps happen at the granularity of 2KB, and a swap is triggered when the number of accesses to a 2KB memory segment reaches a threshold. PoM is adaptive, and the swap threshold can change based on the program characteristics. PoM uses fast swaps and has direct-mapped swap groups.

SILC-FM [9] uses segments (e.g., 2KB), and optimizes the granularity of swaps, which can range from 64B sub-blocks to the whole segment. It supports sub-block interleaving, where two segments interleave data at sub-block granularity. SILC-FM also relaxes swap groups to be set-associative rather than direct-mapped. It uses slow swaps.

MemPod [8] further enhances swap flexibility by allowing any slow-memory segment to be swapped with any fast-memory segment within a Pod. This comes at the cost of a substantial increase in the metadata overhead. MemPod uses the Majority Element Algorithm [24] to identify memory segments that are to be accessed in the future, and migrates them to fast memory at 2KB granularity after predefined time intervals.

Other hardware schemes target different aspects of hybrid memory systems. For example, BATMAN [25] tries to optimize swaps so that the overall memory bandwidth utilization is maximized. ProFess [17] proposes a cost-benefit mechanism that decides swaps considering fairness between different programs that compete for fast memory.

One difficulty in this area is the need to make swap decisions early enough. Otherwise, it is likely that a swap for a memory segment will not be finished by the time memory requests for the segment arrive.

C. Page Walk

In x86, only 48 bits out of the 64-bit virtual address (VA) are used for addressing. Of those, the lower 12 bits are used for the offset within the 4KB page. When a virtual to physical page translation is not found in the TLB, the hardware initiates a page table walk. A page table walk consists of the hardware stepping over four levels of page tables (Figure 1): the Page Global Directory (PGD), the Page Upper Directory (PUD), the Page Middle Directory (PMD) and, finally, the Page Table Entry (PTE). The base of the PGD is obtained by using the CR3 register, which is unique to a process. Adding CR3 to bits 47-39 of the VA, we obtain a PGD entry, whose contents is the base of the PUD table. Adding this base to bits 38-30 of the VA, we obtain a PUD entry, whose contents is the base of the PMD table. This process repeats until we obtain the base of the requested physical page, which is finally added to the page offset.

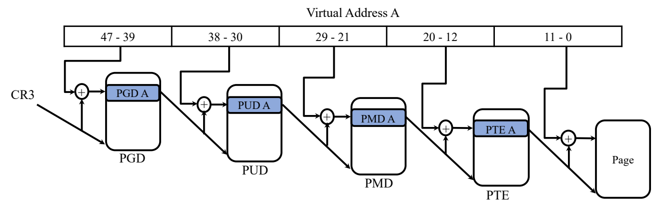

Figure 1: Page walk operation.

This process may require up to four memory accesses, to get the entries in the PGD, PUD, PMD, and PTE tables. To avoid main memory accesses, the data in these entries can be stored in the caches (except in L1), along with regular data. In addition, to further reduce the cost, modern processors have an intermediate translation cache called the Page Walk Cache (PWC), which stores a few entries per translation level (except for the PTE). The PWC is accessed before going to the L2 cache to obtain the entries. The four-step page walk and the PWC are in the core's Memory Management Unit (MMU).

III. Design of PageSeer

A. Main Idea

PageSeer is a hardware mechanism that initiates early page swaps — also called *Prefetch Swaps* — between slow and fast memory. Prefetch swaps are initiated before the main memory receives multiple requests for the page in slow memory to be moved to fast memory. To initiate prefetch swaps, PageSeer uses state stored in a hardware table called Page Correlation Table (PCT). Prefetch swaps can be triggered by one of two events: (i) a hint from the MMU (*MMU-triggered Prefetch Swaps*), or (ii) a regular memory access (*Prefetching-triggered Prefetch Swaps*).

PageSeer also supports regular swaps, which are initiated when the main memory receives a certain number of requests for a page. To initiate regular swaps, PageSeer uses state stored in a hardware table called Hot Page Table (HPT).

Most of PageSeer's hardware structures are placed in the memory controller, which we call *Hybrid Memory Controller* (HMC). In addition, PageSeer needs swap buffers in the DRAM and NVM memory modules, and slightly modifies the MMUs. Figure 2 shows the architecture of PageSeer, where the new or modified hardware structures are shown shaded, and the added connections are shown in lighter color. In the following, we first describe the communication between MMU and HMC (Section III-B), then the structures in the HMC (Section III-C), and then the operation of PageSeer (Section III-D).

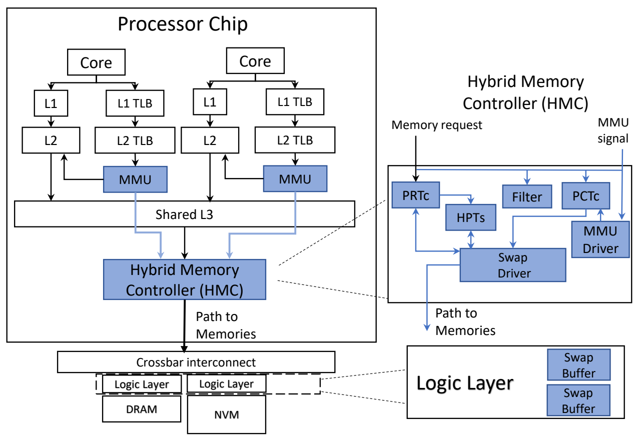

Figure 2: PageSeer architecture, where the new or modified hardware structures are shown shaded, and the added connections are shown in lighter color.

B. MMU-Triggered Prefetch Swaps

Memory-intensive applications that access many pages are likely to miss in the TLB. Further, after a page walk, applications with large working sets are unlikely to find the requested PTE entry in the caches, and are likely to have to go to main memory. Under such conditions, PageSeer uses the time that it takes to satisfy a TLB miss to potentially perform a prefetch swap of the requested page and one additional page — bringing them to fast memory in expectation that they will be referenced very soon. We call these actions *MMU-Triggered Prefetch Swaps*.

This operation requires some hardware modifications. In conventional systems, as soon as a page walk reaches the fourth translation level and the address of the memory line with the needed PTE entry is known, the MMU sends a request to the L2 cache. In PageSeer, the MMU additionally sends a signal to the MMU Driver in the HMC. This is shown as action ① in Figure 3.

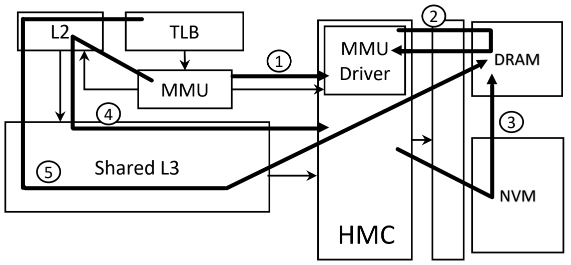

Figure 3: Performing MMU-triggered prefetch swaps in PageSeer.

When the MMU Driver in the HMC receives the signal, it sends a memory request to DRAM to obtain the PTE entry (action ② in Figure 3). When the HMC obtains the PTE entry, it extracts the Physical Page Number (PPN) from it. After that, based on the state of the PCT and other internal state, it decides whether to perform a prefetch swap of the requested page and one additional page (action ③ in Figure 3). In any case, the memory line with the needed PTE entry is cached in the MMU Driver of the HMC. Further, some internal state in the HMC is updated.

This design has two benefits. The first one occurs if, later, the request from the MMU to the L2 cache to obtain the PTE entry ends-up missing in both the L2 and L3. At that point, conventional systems send the request to the memory controller, which should initiate a main memory access (action ④ in Figure 3). However, since the MMU Driver in the PageSeer HMC does cache the line with the PTE entry (or, at least, it has already issued a request for it), the HMC provides the data faster. Note that the HMC needs to know that this is a request for a line with a PTE entry. To make this possible, PageSeer adds an identifying bit in the message that the MMU sends to the L2.

The second, more important benefit occurs when the TLB in updated with the new translation and the original memory request is replayed. If the request misses in the caches and is directed to a page that was in NVM, the prefetch swap triggered by PageSeer may have brought the page to DRAM. The result is a faster memory access for this request and potentially future ones (action ⑤ in Figure 3).

C. Structures in the Hybrid Memory Controller

In addition to the MMU signals described in the previous section, the HMC handles all the memory requests. The HMC includes some hardware structures that swap pages between the slow and fast memories, keep track of address re-mappings, monitor memory activity, and trigger swaps. We describe them in this section.

1) Page Re-mapping: PageSeer swaps pages without OS knowledge. As a result, the hardware needs to examine every request that reaches the HMC to determine if the location of the page has changed as a result of a swap. PageSeer accomplishes this with the *Page Remapping Table* (PRT). This hardware table is responsible for keeping information on all the current page re-mappings. The OS is oblivious to the re-mappings.

The information needed to keep track of all the current re-mappings is substantial. Moreover, the PRT is accessed on every main memory access and is on the critical path. Hence, we need to keep the access time to a minimum. Consequently, rather than having the whole PRT in the HMC, PageSeer saves storage and latency by keeping a cache of the PRT in the HMC, which holds only some of the PRT entries. We call it *PRTc*, for PRT cache. The rest of the entries are stored in DRAM, like in other designs [7]–[9], [17].

We design the PRT and the PRTc so that they can be accessed quickly and use the storage efficiently, minimizing the amount of metadata they need to hold. Specifically, we constrain the pairs of DRAM and NVM pages that can be swapped with each other. As shown in Figure 4, only DRAM and NVM pages of the same cache color can be swapped with each other. This means that an NVM page can only be swapped with DRAM pages that map to the same PRTc set.

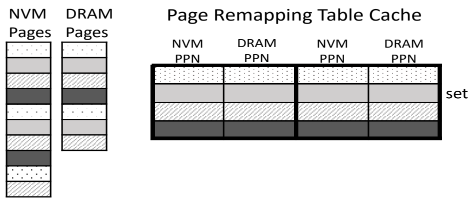

Figure 4: Page Remapping Table Cache (PRTc).

As shown in Figure 4, the PRTc is set-associative, and each entry has an NVM PPN and a DRAM PPN. The entry denotes that these two pages have been swapped — i.e., the NVM data is in DRAM, using the original location of the DRAM PPN, and vice-versa. With this design, PRTc queries are fast. When a request for a memory address arrives at the PRTc, the hardware extracts the address' PPN. Irrespective of whether this PPN is in the physical memory range of NVM or in the physical memory range of DRAM, the same PRTc set is accessed and the multiple entries are read out. Then, if this PPN is in the NVM range, the PPN is compared to the leftmost field in each of the selected entries; if this PPN is in the DRAM range, it is compared to the rightmost field in the entries.

This design requires that pages that are not currently swapped remain in their originally-assigned location. For example, an NVM page that is swapped to DRAM and then returns to NVM has to return to its original position. The same is true for a DRAM page. This design is very space efficient, as it requires minimal metadata. However, it cannot support fast swaps.

To reduce the cost of swaps, PageSeer uses what we call *Optimized* slow swaps. The idea is to reduce the number of read and write operations by temporarily keeping one of the pages in a swap buffer. Figure 5 shows an optimized slow swap. The figure considers three pages: DRAM page ① and NVM pages ② and ③. In the past, pages ① and ② have been swapped. As a result, the state of the memory is represented by the *dark* circles in the left figure labeled *Step 1*.

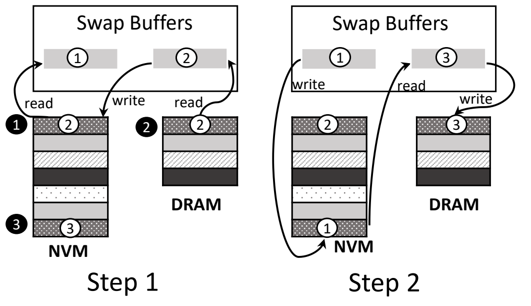

Figure 5: Optimized slow swap operation.

Suppose now that that we need to move page ③ to DRAM and, because of the state of the replacement algorithm, it has to go to the place currently occupied by page ②. A fast swap would simply swap pages ③ and ②, for a total of 2 page reads and 2 page writes. However, it would not bring ② to its original place in NVM (which is currently used by page ①). A slow swap, instead, would swap ① and ②, and then ③ and ①, for a total of 4 page reads and 4 page writes.

An Optimized slow swap leverages the swap buffers to only perform 3 reads and 3 writes. This is shown by the white circles. In *Step 1*, pages ① and ② are read into the swap buffers, and page ② is written to its original NVM location. Then, in *Step 2*, page ③ is read into a free swap buffer, and pages ① and ③ are written to NVM and DRAM, respectively.

2) Initiating Prefetching-triggered Prefetch Swaps: Besides the MMU hints, the other trigger of PageSeer actions is LLC misses. PageSeer has two hardware structures that track LLC miss information and initiate swaps. They are the *Page Correlation Table* (PCT), which initiates prefetching-triggered Prefetch Swaps (in addition to assisting in MMU-triggered Prefetch Swaps), and the *Hot Page Tables* (HPTs), which initiate Regular Swaps. In this section, we describe the PCT; in the next one, we describe the HPTs.

When a page *P1* is accessed, the main memory system often observes a flurry of LLC misses on *P1* in a short period of time, followed by a flurry of misses on another page *P2*, and so on. Further, later, when *P1* is accessed again, *P1* is often seen to cause a similar flurry of misses, again followed by a flurry of misses by follower *P2*. For a page like *P1*, a PCT entry saves the number of LLC misses observed when *P1* is accessed, the PPN of its follower page *P2*, and the number of misses observed on *P2*. Later, when *P1* is accessed and triggers its first miss, if its PCT entry's miss count is higher than a threshold, and *P1* is in NVM, PageSeer issues a prefetching-triggered prefetch swap for *P1*. Further, if *P1*'s follower *P2* has a miss count higher than the threshold, and *P2* is in NVM, PageSeer also issues a prefetching-triggered prefetch swap for *P2*. With these early swaps, PageSeer can avoid repeated, costly NVM accesses. The miss count threshold is set so that the cost of a swap is lower than the expected savings to be attained.

In practice, the PCT is too large to keep in the HMC. Consequently, the HMC keeps a *PCT cache* (PCTc). A PCTc entry contains the PPN of a leader page, the number of LLC misses on the page per invocation, the PPN of the follower page, and the number of LLC misses on the follower per invocation. This is shown in the top part of Figure 6.

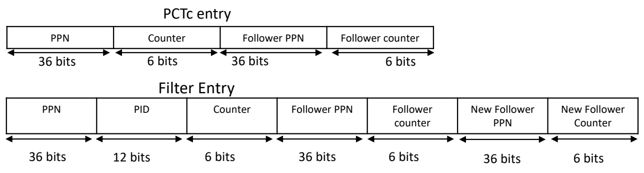

Figure 6: Structure of a PCTc and a Filter entry.

Since the miss patterns of a page change with time, PageSeer also uses a small hardware *Filter* table. Its purpose is to quickly update the information of a page's PCTc entry. The Filter table only has a few entries and works as follows. When the HMC observes an LLC miss for a page, it brings the page's PCTc entry (if it exists) into the Filter table. As execution proceeds, PageSeer recomputes a new miss count for the page by adding the number of misses observed in the current invocation plus half the value of the old miss count. This is done for both leader and follower pages. This approach is taken to reflect the new miss patterns, while retaining some history from past invocations. The new miss counts are stored back in the corresponding PCTc entry.

The structure of a Filter entry is shown in the lower part of Figure 6. In addition to the leader page's PPN and counter, and the follower page's PPN and counter, it has three more fields. One is the program identifier (PID) of the process accessing the leader and follower pages. This is needed so that, in a multi-program environment, PageSeer does not try to correlate pages that are accessed by different programs; we want to correlate leader-follower pages accessed by the same program.

The two additional fields are the PPN and counter for a new follower page. It is possible that, in this new use of leader page *P1*, *P1* is not followed by the use of page *P2*, but by the use of page *P3*. For this reason, the Filter entry has space to record the accesses to a new follower (*P3*). Later, when the Filter entry is to be saved into the PCTc, only the follower with the highest count is saved.

Each PCTc entry has one additional bit that indicates whether the entry's contents have effectively changed since it was last brought in from the PCT in main memory. An effective change is one that causes a different swap action for any of the pages involved. When a PCTc entry is evicted, it is written back to the PCT only if the change bit is set.

Finally, the PCTc state is also used for the MMU-triggered prefetch swaps of Section III-B. The difference is that the trigger that causes the PCTc look-up and potentially the two swaps is not an access to a page. Instead, it is a signal from the MMU Driver that was initiated by a TLB miss.

3) Initiating Regular Swaps: The *Hot Page Tables* (HPTs) are two small hardware tables, one for DRAM pages and the other for NVM pages, that record the pages that are being frequently accessed (i.e., are "hot"). Each HPT entry has a PPN and a counter of how many LLC misses have been recorded on this PPN. Every time that the HMC receives an LLC miss for a page, the corresponding counter is incremented. The counters are automatically halved at regular intervals. If the counter in an entry reaches zero, the corresponding page is removed from the HPT.

The goal of the DRAM HPT is to lock hot pages in DRAM. A DRAM page that appears in the HPT is being accessed a lot and, therefore, should not be swapped out of DRAM. The goal of the NVM HPT is to identify NVM pages that are becoming hot and should be swapped to DRAM. When the count in an entry of the NVM HPT reaches a swap threshold, the hardware starts a regular swap operation for the corresponding NVM page. Note that the NVM HPT complements the PCTc. The latter may fail to initiate a swap for the page, either because the PCTc does not yet have an entry for the page, or because the current count value is too low to initiate a prefetching-triggered prefetch swap. The NVM HPT has a lower count threshold to initiate a swap than the PCTc. Both the HPTs and the PCTc are off the critical path.

4) Other Structures: The two other HMC structures in Figure 2 are the *MMU Driver* and the *Swap Driver*. The former gets a signal from the MMU on a page walk. It then obtains the memory line with the relevant PTE entry — either from memory or from the small set of lines with PTEs that it caches. After that, it generates the page PPN and checks the PCTc to determine if an MMU-Triggered Prefetch Swap needs to be started for the page. The MMU Driver also intercepts LLC misses requesting lines with PTE entries.

The Swap Driver initiates all page swaps. It receives requests from either the PCTc or the HPT. It also checks all memory accesses, to ensure that those directed to pages being swapped get the data from the swap buffers.

D. Putting All Together: PageSeer Operation

After having described the HMC structures, we can explain PageSeer's operation. We divide it into three flows: (i) a regular memory request reaches the HMC, (ii) an MMU signal or an LLC miss requesting a PTE entry reaches the HMC, and (iii) a regular memory request reaches the HMC while a swap is in progress.

1) A Regular Memory Request Reaches the HMC: The PRTc is accessed to find out the correct address in case the page is remapped. In parallel, the PCTc and Filter receive the request. Note that PCTc and Filter use addresses before remapping, to be able to retain their state across remappings. If the PRTc or PCTc miss, memory requests are sent to DRAM to fetch the appropriate entries.

Immediately after the PRTc look-up, we have the correct address, and the request is sent to main memory (with a Swap Driver look-up). In parallel, the request is sent to the DRAM and NVM HPTs. After the Swap Driver receives signals from the HPTs and the PCTc, it knows whether the NVM HPT or the PCTc request a swap (for this page and/or its successor), and which pages cannot be swapped out of DRAM (from the DRAM HPT). If appropriate, the Swap Driver initiates page swap(s).

In a swap operation, as the hardware reads a page into a swap buffer, it starts with the requested cache line first. It also provides the requested line right away to the processor.

The Swap Driver may refuse to perform a swap if, due to a large number of requests directed to the DRAM, the DRAM bandwidth is saturated and the NVM bandwidth is under-utilized. Performance is usually higher if saturation of the DRAM links is avoided.

2) An MMU Signal or an LLC Miss Requesting a PTE Entry Reaches the HMC: As indicated in Section III-C4, the MMU Driver intercepts these two types of requests. On an MMU signal, the MMU Driver obtains the PTE, then fetches the needed PRTc and PCTc entries (if missing) and, finally, it checks the PCTc to determine whether an MMU-Triggered Prefetch Swap needs to be initiated. On reception of an LLC miss requesting a line with a PTE entry, the MMU Driver provides it from its cache.

3) A Regular Memory Request Reaches the HMC while a Swap Is in Progress: The Swap Driver checks whether the request targets a page that is participating in the swap. If it does not, the request proceeds normally. Otherwise, the request obtains the data from the appropriate swap buffer. This helps avoid stalls for requests directed to these hot pages. The swap buffers temporarily act as prefetch buffers for these pages.

E. Page Swaps between Memory and Disk

PageSeer is compatible with DMA engines that swap pages between memory and disk. All DMA requests go through the HMC, which may change the address if the page has been remapped. As soon as the HMC receives the first DMA request to read/write a line, the HMC completes any swap in progress for that page, then freezes the page (preventing future swaps), and then allows the DMA requests for that page to proceed. After the DMA is done, the page is unfrozen. There is no need to change the state of the page in the HMC structures; the state will dynamically evolve based on the miss patterns of the new page.

IV. Experimental Methodology

A. Evaluation Infrastructure

We use cycle-level simulations to model a server architecture with a 4-core multicore and a 4.5-GB main memory composed of 4 GBs of NVM and 512 MBs of DRAM. The architecture parameters are shown in Table I. Each core is an out-of-order core with private L1 and L2 caches, and a shared L3 cache. It has private L1 and L2 TLBs and page walk caches for intermediate translations. Each core has a page walker. We integrate the Simics full-system simulator [26] with the SST [27], [28] framework, and the DRAMSim2 [29] memory simulator, similar to [30]. To model NVM, we modified the DRAMSim2 timing parameters as shown in Table I, and disabled refreshes. We use CACTI [31] for energy and area analysis of the PageSeer structures. Additionally, we utilize Intel SAE [32] on top of Simics for OS instrumentation. The page walk is modeled after the x86 architecture, and leverages the 4-level page tables created and maintained by the OS to perform the page walk memory accesses. We accurately model the page swaps and the accesses to the HMC structures by issuing the appropriate read and write requests to memory. Our implementation is based on the Ubuntu 16.04 operating system.

Processor/MMU Parameters	
Cores; Frequency	4 out-of-order cores; 2GHz
Cache line	64B
L1 cache	32KB, 8-way, 2 cycles access latency (AL)
L2 cache	256KB, 8-way, 8 cycles AL
L3 cache	8MB, 16-way, 32 cycles AL, shared
L1 TLB	64 entries, 4-way, 1 cycle AL
L2 TLB	1024 entries, 12-way, 10 cycles AL
Main-Memory Parameters	
Capacity	DRAM: 512MB; NVM: 4GB
Channels	DRAM 4; NVM: 2
t_{CAS}-t_{RCD}-t_{RAS}	DRAM: 11-11-28; NVM: 11-58-80
t_{RP},t_{WR}	DRAM: 11,12 NVM: 11,180
Ranks per Channel	DRAM: 1; NVM: 2
Banks per Rank	DRAM: 8; NVM: 8
Frequency; Data rate	1GHz; DDR
Bus width	64bits per channel
Operating System: Ubuntu Server 16.04	

Table I: Configuration of the system evaluated.

B. Configurations

We compare our design to two state-of-the-art hardware-managed hybrid memory systems: PoM [7] and MemPod [8].

PoM: We configure PoM according to the specification given in previous work [7], but we change the architecture-related parameters to adjust it better to our configuration. In the PoM paper, the authors manage die-stacked and DRAM memories with different latencies than ours. Thus, we modify their `K` parameter to 12 to be consistent with our memory timing model. For the `SRC`, which is the equivalent of our PRTc, we use a 32KB cache similar to PageSeer.

MemPod: MemPod uses the MEA algorithm to decide on memory swaps. Both PoM and MemPod swap at the granularity of 2KB. For MemPod, we use 64 MEA counters and make swap decisions every 50 μs, as described in the original work [8]. We also use a 32KB cache for the remapping table. MemPod also requires an inverted map table, but since we lack details about its implementation, and to be optimistic in our evaluation, we assume a zero cycle latency for this structure.

PageSeer: The parameters of our design are shown in Table II. The goal is to keep the size of the entries in the PRTc and PCTc tables small. As a result, we can have more entries in the tables and increase their hit rate. The total size of the HMC structures is less than 72KB, which is very modest. We also need the full PRT and PCT tables in the DRAM, but they account for only 1% of our DRAM storage. In our experiments, we found that caching 16 lines with PTE entries in the MMU Driver is good enough. Doing so gives us a hit rate of over 99% for page walk requests that miss in the LLC and reach the MMU driver.

PageSeer Design Parameters	
Swap size	4KB (which is a page)
Counters	6 bits
MMU to HMC latency	2 cycles (at 2GHz)
PCTc prefetch swap threshold	14
HPT swap threshold	6
HPT counter decrease interval	50K cycles (at 1GHz)
PRT	4 way-set associative
PageSeer Hardware Structures	
PRTc and PCTc	32KB, 4-way, 1 cycle (at 1GHz)
HPT size (each table)	5.3KB,fully-assoc,4 cycle (at 1GHz)
Filter	2.2KB,fully-assoc,2 cycle (at 1GHz)
MMU Driver	16 lines with PTEs, 64B per line
PRTc,PCTc,HPT,Filter entry	3.5B, 10.5B, 5.25B, 17.25B
PageSeer Hardware Structures – Area and Energy per Access Area(A) $*10^{-3}mm^2$, Leakage(L) mW, Rd/Wr(R/W) pJ	
PRTc	A: 54.9, L: 11.4, R/W: 14.8/14.4
PCTc	A: 36.8, L: 11.4, R/W: 14.7/16.7
HPT	A: 23.7, L: 9.1, R/W: 1.8/2.6
Filter	A: 7.7, L: 2.3, R/W: 1.4/2.7
PageSeer Structures in DRAM	
PRT	426KB
PCT	7MB with follower 884.7KB without follower

Table II: PageSeer parameters.

C. Workloads

To evaluate the efficacy of our design, we run 20 different benchmarks organized into 26 workloads. They are shown in Table III, with the memory footprint for our simulated period when a *single* instance of the benchmark is running. We choose eight memory intensive benchmarks from the SPEC CPU2006 suite [33], six benchmarks from Splash-3 suite [34] and six benchmarks from CORAL [35], which are used for testing HPC systems. There are two types of workloads. The first twenty are unique-benchmark workloads, where we run multiple instances of the same benchmark on different cores. Typically, we run four instances. However, in cases where the memory footprint was not enough to stress our memory system, we increased the number of cores and run more instances of the same benchmark (see Table III). The next workloads are 6 mixes of benchmarks, where different benchmarks are running on different cores.

Workload	MB(Single)	Workload	MB(Single)
lbm×4	422	luNCon×4	520
milc×4	380	oceanCon×4	887
bwaves×4	385	barnes×8	250
GemsFDTD×4	502	radix×4	648
mcf×8	290	stream×4	457
libquantum×6	267	miniFE×4	480
omnetpp×8	164	LULESH×4	914
leslie3d×12	62	AMGmk×4	350
fft×4	768	SNAP×4	441
luCon×4	520	MILCmk×4	480
mix1: lbm-LULESH-SNAP-leslie3d			
mix2: AMGmk-luCon-radix-barnes			
mix3: miniFE-oceanCon-barnes-AMGmk			
mix4: LULESH-MILC-miniFE-stream			
mix5: luCon-radix-oceanCon-barnes			
mix6: libquantum-lbm-mcf-bwaves			

Table III: Workloads.

For the unique-benchmark workloads, we simulate 2 billion instructions per core, while for the mixed-benchmark workloads, we simulate until a core reaches 2 billion instructions, or a program terminates. In both cases, we perform 1.5 billion instructions of warm-up per core.

V. Evaluation

A. PageSeer Characterization

The goal of PageSeer is to identify pages that are "hot" and move them to DRAM as soon as possible, while preparing the HMC structures for accesses to these pages. As a result, PageSeer's effectiveness can be quantified by how accurately it recognizes present and future "hot" pages, and how fast it manages to move them to DRAM. In this section, our simulations do not take into account contention for main-memory system bandwidth. In reality, maximum performance will be obtained when some memory requests actually access the NVM rather than the DRAM, so that the overall bandwidth of both memories is effectively utilized. However, in this section, we want to know if PageSeer can identify the pages that are worth moving to DRAM.

In Figure 7, we present what fraction of the main-memory accesses were serviced from DRAM, NVM, or the swap buffers for the three configurations we are comparing. Each bar of the plot represents one of the three configurations (PoM, MemPod, and PageSeer), and each bar shows a breakdown of the memory requests serviced from each memory module. We present the results for each benchmark suite and our mixes. From this figure, we see that PageSeer directs a vast number of memory requests to DRAM (88.5% on average), a small but non-negligible number to the swap buffers (2.2% on average) and the rest to NVM. Compared to the other two schemes, PageSeer can better recognize and predict hot pages, and move them to fast memory on time.

The improvement over PoM and MemPod is mainly because of two reasons. First, PageSeer takes a swap decision ahead of time, so it does not wait until requests start hitting the NVM to initiate a swap. The second reason is that the MMU signal and the history of page accesses in the PCTc are an accurate indication of future accesses to a page. What

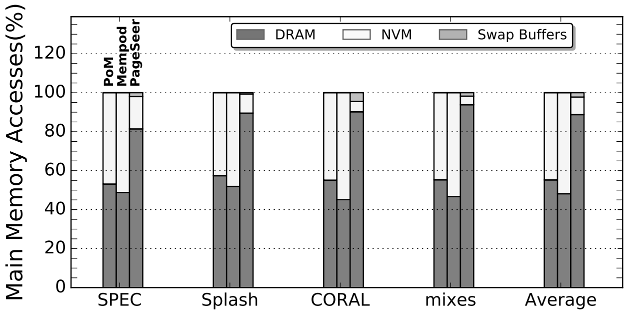

Figure 7: Percentage of main-memory accesses to each memory module for PoM, MemPod, and PageSeer.

is more, MemPod swaps pages at regular time intervals, which are not optimal for every application. In addition, all pages qualified for a swap start moving at the same time, causing swap bursts. As for PoM, it restricts its swap flexibility with a direct mapped re-mapping table, losing the opportunity to have multiple pages of the same swap group in DRAM.

Figure 8 depicts the result of the swaps for each configuration. The figure shows the positive, the negative, and the neutral main-memory accesses as a percentage of the total main-memory accesses for each configuration. We consider a main-memory access to be positive when it accesses DRAM instead of NVM thanks to a swap operation, and negative when the opposite happens. Neutral accesses are those that end-up accessing the same type of memory as a run without swaps. We see that, on average, PageSeer attains 16% and 13% more positive accesses than PoM and MemPod, respectively, and that it removes practically all of the negative accesses. To achieve that, it can be shown that PageSeer introduces 1% and 2.8% more swaps than PoM and MemPod, respectively.

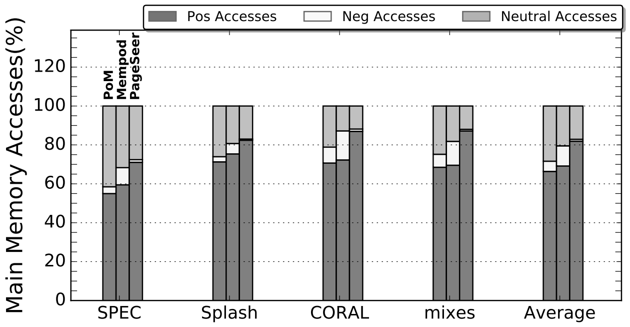

Figure 8: Characterization of swap effectiveness for PoM, MemPod, and PageSeer.

The takeaway is that PageSeer is capable of identifying hot pages and swapping them to fast memory. The result is a high percentage of positive accesses (81.3% on average) and only 1% of negative accesses on average.

Next, we present the accuracy and effectiveness of the prefetch swap mechanism alone. Recall from Section III-A that PageSeer supports regular swaps (initiated by the HPT) and prefetch swaps (initiated by the PCTc). The latter can be triggered by either a hint from the MMU (MMU-triggered Prefetch Swaps), or a regular memory access (Prefetching-triggered Prefetch Swaps). In this discussion, we focus on prefetch swaps only.

Figure 9 presents the accuracy of prefetch swaps. A swap is deemed accurate when the number of accesses to the swapped page in fast memory is enough to justify the page swap cost. In our experiments, we want to attain at least 14 positive accesses to a page to recognize the prefetch swap of the page as accurate. As we can see from the figure, our mechanism is accurate in the vast majority of times. It has an average accuracy of 86.7%. `GemsFTDT` is the only benchmark for which the accuracy of our mechanism is low (28.3%) and we also perform lots of prefetches (as we will see later). This occurs because the pattern of accesses to pages changes with time. `luCon` experiences relatively low accuracy but, as we will see, the total number of prefetches is not high enough to cause an application slowdown.

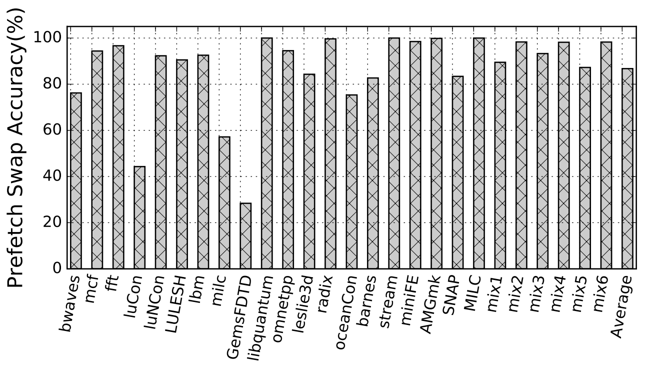

Figure 9: Accuracy of PageSeer's prefetch swaps.

In Figure 10, we present the percentage of swaps that are prefetch swaps. We break the prefetch swaps into prefetching-triggered and MMU-triggered. The remaining swaps to 100% in the figure are regular swaps. The benchmarks are organized into two groups. The group on the left contains those benchmarks for which PageSeer is not able to generate many prefetch swaps. This can happen because the pages for these benchmarks do not receive enough accesses to qualify for prefetching. Another reason may be that the highly-accessed pages of the application are moved to DRAM and the remaining pages are not worth swapping. However, even PageSeer's prefetch mechanism cannot find prefetch opportunities, PageSeer can still provide high performance through the use of the HPTs.

The group on the right contains those benchmarks for which PageSeer generates many prefetch swaps. We see that these are the most common benchmarks. Importantly, we see that MMU-triggered swaps are much more frequent than prefetching-triggered swaps. This shows the benefit of leveraging the MMU hints to initiate swaps.

Overall, on average for all the benchmarks, prefetch swaps account for 62.8% of all the swaps. In addition, 48.6% of

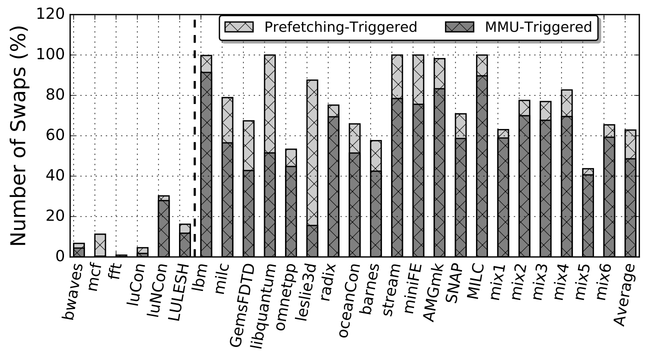

Figure 10: Percentage swaps that are prefetch swaps in PageSeer.

all the swaps are MMU-triggered swaps.

B. PageSeer Performance

In this section, we evaluate the performance of PageSeer. We consider a variety of metrics that give insights into PageSeer. Our simulations in this section take into account the contention for the main-memory system bandwidth.

In this environment, we want to avoid saturating the DRAM channels and not using the NVM channels. Consequently, our Swap Driver uses a simple heuristic to avoid the most extreme imbalanced conditions. Specifically, when the Swap Driver observes that the DRAM channels are saturated, it considers declining some of the incoming swap requests. Specifically, it declines to swap requests if over 95% of the main-memory requests up to this point in the application have been satisfied by DRAM. While this is a primitive heuristic, it is effective.

To assess the impact of this heuristic, in Figure 11, we show the rate of swaps in each benchmark suite, in swaps per kilo-instruction. Each suite has two bars. *PageSeer w/ BW-opt* corresponds to PageSeer; *PageSeer w/o BW-opt* corresponds to PageSeer without the Swap Driver heuristic described above to limit DRAM channel saturation. On average, these rates are 0.19 and 0.35 swaps per Kinstruction, respectively. The heuristic has an impact.

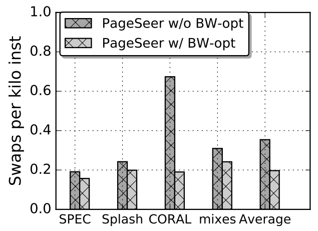

Figure 11: Swap rate.

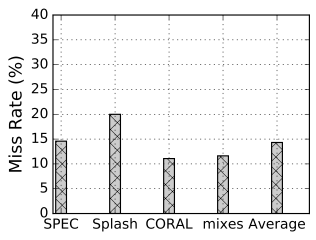

Figure 12: Cache miss rate of PTEs on a TLB miss.

We now examine one aspect of how PageSeer helps TLB misses. In Figure 12, we consider the TLB misses and record whether the resulting request for the PTE does miss in the caches (L2 or L3). The figure shows the miss rate of such requests. We see that, on average, 14.5% of these requests miss in the caches and reach the HMC. Fortunately, as indicated above, it can be shown that over 99% of these misses are satisfied by the cache in the MMU Driver. Consequently, PageSeer is effective at reducing the latency of such accesses.

The performance benefits of PageSeer come from three factors. First, PageSeer predicts that a page in NVM will receive a large number of main memory accesses in the near future, and moves the page to DRAM before the requests arrive. The result is an increase in the number of accesses that are satisfied by DRAM. Figures 7 and 8 showed that PageSeer is very effective at exploiting this factor.

The second factor is that PageSeer can satisfy main-memory requests for pages that are currently taking part in a swap. Specifically, the swap buffers can serve memory requests if they contain the data requested. However, we showed in Figure 7 that only a small percentage of the total main-memory accesses are serviced from the swap buffers.

The third factor is that a request in PageSeer spends relatively little time in the HMC structures, and particularly in the PRTc. This is because, as soon as the HMC receives an MMU hint, or information from the PCTc that a follower page will be prefetched, the PageSeer hardware starts fetching the corresponding PRTc and PCTc entries for the pages. It is especially important to load the PRTc as soon as possible, since it stands on the critical path of a memory request. Note that the time wasted in a PRTc miss is not negligible. On a PRTc miss, the hardware has to access DRAM. The earlier that we fetch these entries, the better.

To assess this factor, Figure 13 compares the total time that requests in PageSeer and PoM spend waiting at the PRTc to load PRTc entries. Specifically, the figure shows the reduction of the waiting time in PageSeer compared to PoM. We use the same size PRTc (32KB) for both PageSeer and PoM.

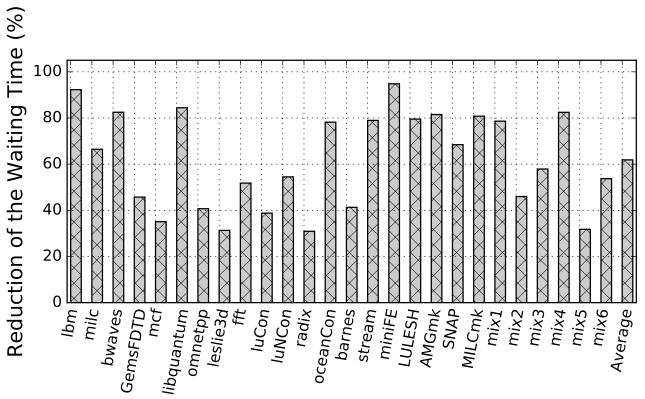

Figure 13: Reduction of the PRTc waiting time in PageSeer compared to PoM.

The figure shows that, on average, the total request waiting time for the PRTc in PageSeer is 61.8% lower than in PoM. Of course, many of the requests may be serviced in parallel. However, this is sizable number that affects the performance of the schemes. It can be shown that the magnitude of this waiting time is equal to 18% of the total execution time of the benchmarks in PoM, and to 12.8% of the total execution time in PageSeer. In addition, it can be shown that the hit rate in the PRTc is 3.5 points higher in PageSeer than in PoM.

Taking all this into account, in Figure 14, we compare the overall performance of PoM, MemPod, and PageSeer. The top graph in Figure 14 shows the IPC of each application in PoM and PageSeer normalized to MemPod. The bottom graph depicts the Average Main-Memory Access Time (AMMAT) of each application in PoM and PageSeer, also normalized to MemPod. As in previous work [8], AMMAT is calculated as the average time spent by a main-memory request to go from the memory controller to main memory and back to the memory controller.

The top graph shows that PageSeer outperforms MemPod and PoM. On average across all the benchmarks, the IPC in PageSeer is 28% and 19% higher than in MemPod and PoM, respectively. Furthermore, the bottom graph shows that the main-memory requests in PageSeer take less time to access main memory than in the other architectures. Specifically, on average across all the benchmarks, the AMMAT in PageSeer is 37% and 29% lower than in MemPod and PoM, respectively.

Even for benchmarks where PageSeer cannot perform many prefetch swaps (shown in the leftmost side of Figure 10), PageSeer still delivers a performance equivalent or better than PoM. The reason is two-fold. First, the NVM HPT in PageSeer still triggers swaps. Second, PageSeer has less stall due to PRTc misses than PoM for all the benchmarks, as shown in Figure 13. For example, consider `mcf`. While PageSeer is unable to perform many prefetch swaps (Figure 10), its lower PRTc stall time induces speedups over PoM.

Out of the 26 workloads that we tested, PoM has a higher IPC than PageSeer only in `milc` and `GemsFTDT`; MemPod never has a higher IPC than PageSeer. These two benchmarks experience lower than average prefetch swap accuracy, as shown in Figure 9. As a result, their performance is harmed. This occurs because pages experience different access patterns at different times.

Overall, our experimental results confirm that PageSeer efficiently manages a hybrid memory system, and that the communication between the MMU and the HMC increases performance.

C. Analysis of Page Correlation Prefetching

PageSeer uses page correlation prefetching to initiate some of the prefetch swaps. We want to understand the contribution of this technique to the overall performance. Consequently, we evaluate an architecture like PageSeer except that PCTc entries have no follower information. As a result, there is no correlation prefetching (*PageSeer-NoCorr*).

We find that, on average, PageSeer and PageSeer-NoCorr

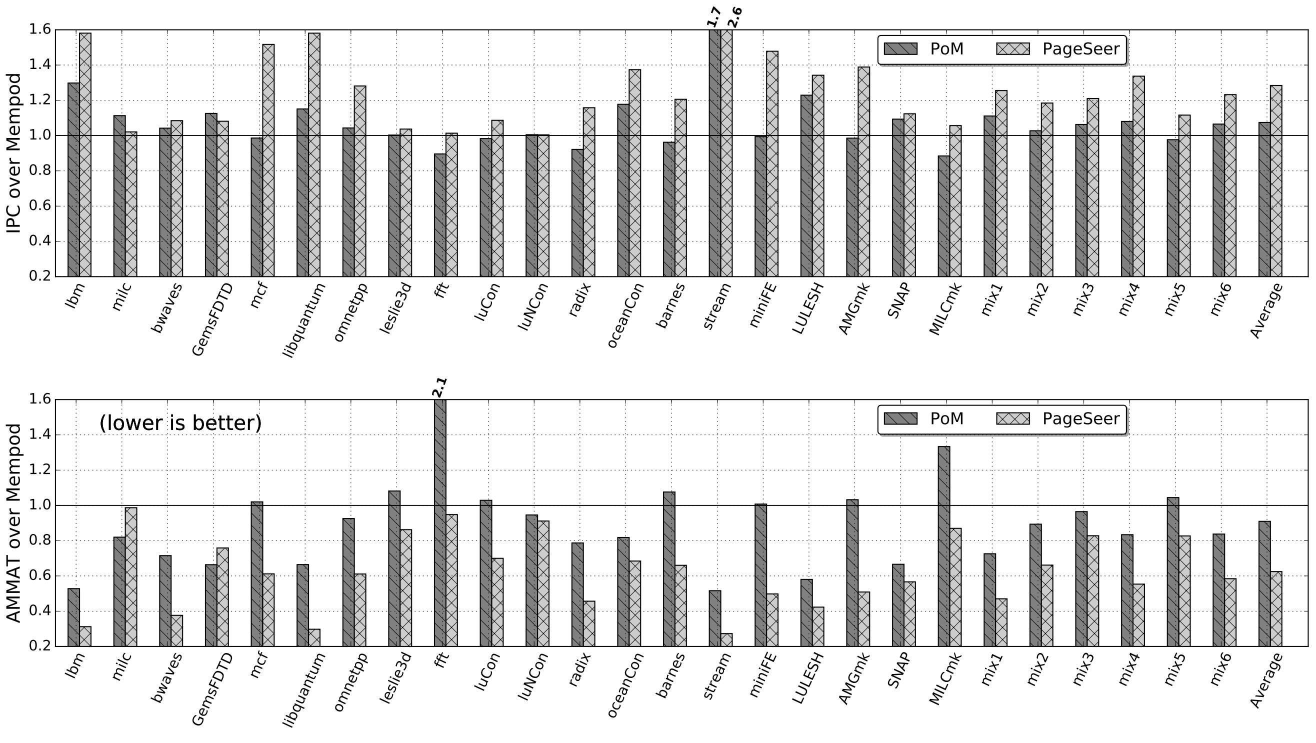

Figure 14: IPC and AMMAT in PageSeer and PoM normalized to the same measures in MemPod.

deliver similar performance. The reason is that, often, the MMU signal alone is able to notify the HMC about most of the future page accesses. Hence, the hardware is able to prefetch pages to DRAM without the correlation prefetching mechanism. However, the results vary across benchmarks. For example, it can be shown that `radix` shows a performance improvement of 11% with PageSeer over PageSeer-NoCorr, while `LULESH` shows 3% performance reduction. In general, supporting correlation prefetching is advantageous when the MMU signal cannot notify the HMC about the forthcoming pages. This occurs when there are few TLB misses. On the other hand, supporting correlation prefetching is undesirable when the page access patterns change frequently.

VI. Other Related Work

Bhattacharjee [36] proposes TEMPO, which involves using the virtual to physical address translation process to prefetch a data cache line into the LLC. As a page walk request for the PTE reaches the memory controller, the memory controller fetches the cache line with the PTE, and uses the PTE value to fetch the cache line that triggered the page walk into the LLC. It also prepares the DRAM row buffer for accesses to nearby cache lines. TEMPO is different from PageSeer in three ways. First, TEMPO targets cache line prefetching, while PageSeer targets optimizing hybrid memory systems. Second, TEMPO prefetches a cache line into the LLC, while PageSeer initiates page swaps between slow and fast memory. Finally, TEMPO only initiates prefetches for page walk requests that miss in the LLC, while PageSeer initiates an MMU hint for every single page walk.

Section II-B described hardware schemes for managing hybrid memory systems, either as a flat memory address space or as a DRAM cache for NVM. Apart from the different swap triggers that we discussed, these schemes tackle some other aspects that can be incorporated into our scheme without major modifications. For instance, SILC-FM [9] suggests a method to swap only a portion of a page. This method can be adopted by PageSeer and save memory bandwidth. A bitmap for a page can tell us which cache lines from a page are worth swapping, and avoid moving 4KB of data. MemPod [8] mentions a clustered architecture that groups together memory controllers to be more scalable. The same approach can be embraced by PageSeer.

In Section II-A, we mentioned software schemes for hybrid memory management. Besides the aforementioned methods, hybrid memory systems are an active area of research. There are proposals for hardware/sofware techniques for effective page placement [21], [37]–[39]. These techniques require either annotations to the applications and compiler support to identify "hot" data structures at compile time, or OS involvement to track memory activity and swap pages. The role of the hardware is to inform the OS about pages that need to be swapped. In other schemes [38], [40], the OS involvement is proposed as an optimization, to periodically update the page table entries, and relieve some pressure from the hardware remapping tables. The remapping table entries can be freed if the OS updates its page tables with the remapping information.

A well-known mechanism to hide memory latency is data cache prefetching. In theory, a data cache prefetcher could be used in PageSeer if it could help identify pages that will soon be accessed and, therefore, trigger DRAM-NVM swaps. However, data cache prefetches are not issued as early as PageSeer's MMU hints. Specifically, a regular data prefetcher will not issue any prefetches until the TLB has the translation for the page. On the other hand, our MMU hints give early information to the HMC about future accesses — before the TLB entry is filled and memory requests for the page are generated. Thus, PageSeer can start the swap earlier and prepare the HMC structures.

Researchers have examined the use of prefetching for hybrid memory systems [20], [41], [42], to prefetch pages from the slow to the fast memory. In our paper, we examine correlation prefetching. Correlation prefetching has been used in the past for cache lines [43], [44]. Here, we use it to prefetch pages. The intuition behind it is that, many times, a program accesses a set of pages multiple times during the execution, in the same or similar order. Keeping information about the correlation between these pages can help us swap future pages before we see any requests for these pages. Our mechanism has to identify the page patterns using only LLC misses, coming from shared caches in a possibly multi-programmed environment. Of course, the page access patterns may change during program execution. Thus, we need a page correlation mechanism that is able to identify access patterns adaptively across different programs.

VII. Conclusion

This paper presented *PageSeer*, a scheme that performs hardware-managed page swapping in a hybrid memory system using hints from the page walk in a TLB miss. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to its follower page to potentially initiate MMU-Triggered Prefetch Swaps for the page and for its follower. PageSeer also initiates other types of page swaps, as it builds a complete solution for hybrid memory. Our evaluation of PageSeer with simulations of 26 workloads showed that PageSeer effectively hides the swap overhead and services many requests from the DRAM. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improved performance by 19% and reduced the average main memory access time by 29%.

Acknowledgments

This work was funded by a Research Award from SK Hynix, Inc. Their support is gratefully acknowledged. We also thank Dr. Il Park for his advice.

References

[1] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and future directions for the scaling of dynamic random-access memory (DRAM)," *IBM Journal of Research and Development*, March 2002.

[2] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*. Morgan & Claypool Publishers, 1st ed., 2011.

[3] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2013.

[4] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *the 36th Annual International Symposium on Computer Architecture*, pp. 2–13, 2009.

[5] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," *Proceedings of the IEEE*, Sept 2017.

[6] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *the 36th Annual International Symposium on Computer Architecture*, 2009.

[7] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[8] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories," in *2017 IEEE International Symposium on High Performance Computer Architecture*, Feb 2017.

[9] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization," in *2017 IEEE International Symposium on High Performance Computer Architecture*, Feb 2017.

[10] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[11] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[12] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Computer Architecture Letters*, July 2012.

[13] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[14] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[15] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *2007 25th International Conference on Computer Design*, Oct 2007.

[16] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[17] D. Knyaginin, V. Papaefstathiou, and P. Stenstrom, "ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness," in *2018 IEEE International Symposium on High Performance Computer Architecture*, Feb 2018.

[18] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, Feb. 2015.

[19] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A Fully Associative, Tagless DRAM Cache," in *the 42nd Annual International Symposium on Computer Architecture*, 2015.

[20] M. Oskin and G. H. Loh, "A Software-Managed Approach to Die-Stacked DRAM," in *2015 International Conference on Parallel Architecture and Compilation*, Oct 2015.

[21] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[22] T. Straumann, "Open Source Real-Time Operating System Overview (Invited)," in *Accelerator and Large Experimental Physics Control Systems* (H. Shoaee, ed.), p. 235, 2001.

[23] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. Kandemir, "CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System," in *the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[24] R. M. Karp, C. H. Papadimitriou, and S. Shenker, "A Simple Algorithm For Finding Frequent Elements In Streams and Bags," *ACM Transactions on Database Systems*, vol. 28, p. 2003, 2003.

[25] C. Chou, A. Jaleel, and M. Qureshi, "BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM," in *the International Symposium on Memory Systems*, 2017.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, Feb 2002.

[27] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, Mar. 2011.

[28] A. Awad, S. D. Hammond, G. R. Voskuilen, and R. J. Hoekstra, "Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework," Tech. Rep. SAND2017-0002, Sandia National Laboratories, January 2017.

[29] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, Jan 2011.

[30] D. Skarlatos, N. S. Kim, and J. Torrellas, "PageForge: A Near-memory Content-aware Page-merging Architecture," in *the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[31] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, June 2017.

[32] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *the 2016 International Conference on Supercomputing*, 2016.

[33] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, Sept. 2006.

[34] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2016.

[35] "CORAL Benchmark Codes." https://asc.llnl.gov/CORAL-benchmarks/.

[36] A. Bhattacharjee, "Translation-Triggered Prefetching," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[37] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter," in *the 44th Annual International Symposium on Computer Architecture*, 2017.

[38] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *the International Conference on Supercomputing*, 2011.

[39] F. X. Lin and X. Liu, "Memif: Towards Programming Heterogeneous Memory Asynchronously," in *the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[40] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation," in *the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[41] M. Islam, S. Banerjee, M. Meswani, and K. Kavi, "Prefetching as a Potentially Effective Technique for Hybrid Memory Optimization," in *the Second International Symposium on Memory Systems*, 2016.

[42] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk Memory Access Prediction and Streaming," in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[43] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *29th Annual International Symposium on Computer Architecture*, 2002.

[44] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *28th Annual International Symposium on Computer Architecture*, 2001.