On-the-fly Page Migration and Address Reconciliation for Heterogeneous Memory Systems

10

MAHZABEEN ISLAM, SHASHANK ADAVALLY, MARKO SCRBAK, and KRISHNA KAVI,
University of North Texas, USA

For efficient placement of data in flat-address heterogeneous memory systems consisting of fast (e.g., 3D-DRAM) and slow memories (e.g., NVM), we present a hardware-based page migration technique. Unlike epoch-based approaches that migrate heavily accessed ("hot") pages from slow to fast memories at each epoch interval, we migrate a page immediately when it becomes hot ("on-the-fly"), using hardware in user-transparent manner and with minimal OS intervention. The management of physical addresses due to page relocation becomes cumbersome and requires costly OS intervention. We use a small hardware remap table to keep track of new physical addresses of the migrated pages. This limits address reconciliation to occur only at periodic evictions of old remap entries. Also, we propose a hardware-orchestrated light-weight address reconciliation process. For our studied heterogeneous memory system, on-the-fly page migration with hardware-assisted address reconciliation provides 74% and 24% IPC improvements, on average for a set of SPEC CPU2006 workloads when compared to a baseline without any page migration and a system with on-the-fly page migration using OS-based address reconciliation, respectively.

Furthermore, we present an analytical model for classifying applications as page migration friendly (applications that show performance gains from page migration) or unfriendly based on memory access behavior.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**;

Additional Key Words and Phrases: Heterogeneous memory, flat address memory, page migration, page migration friendliness

ACM Reference format:
Mahzabeen Islam, Shashank Adavally, Marko Scrbak, and Krishna Kavi. 2020. On-the-fly Page Migration and Address Reconciliation for Heterogeneous Memory Systems. *J. Emerg. Technol. Comput. Syst.* 16, 1, Article 10 (January 2020), 27 pages.
https://doi.org/10.1145/3364179

1 INTRODUCTION

The demand for large, high bandwidth and efficient main memory systems has led to research on Heterogeneous Memory Architectures (HMAs). Such memory systems include diverse memory devices such as 3D-stacked DRAM (e.g., HBM, HMC), conventional DRAM (e.g., DDR4 DRAM) and non-volatile memories (e.g., PCM, STT-RAM, FeRAM). 3D-DRAM provides higher bandwidth

This research is supported in part by the NSF Net-Centric Industry/University Cooperative Research Center at UNT and its industrial members. The research is also supported by NSF grant #1828105.
Authors' addresses: M. Islam, S. Adavally, M. Scrbak, and K. Kavi, University of North Texas, 1155 Union Circle, Denton, TX, 76203; emails: MahzabeenIslam@my.unt.edu, {ShashankAdavally, MarkoScrbak}@my.unt.edu, Krishna.Kavi@unt.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1550-4832/2020/01-ART10 $15.00
https://doi.org/10.1145/3364179

at lower dynamic energy/bit than conventional DRAM; albeit with smaller capacities (10 s of GBs) [HMCC 2018; JEDEC 2018; Kim and Kim 2014]. Non-volatile memories (NVMs) are denser than DRAM, providing higher capacities and also consumng less static energy, but have higher access latencies and higher read/write energy overheads [Numonyx 2009; Qureshi et al. 2011]. Fast, high bandwidth but low capacity memory (viz., 3D DRAM) has been employed either as large last level cache (LLC) [Jevdjic et al. 2014, 2013; Kavi et al. 2015; Qureshi and Loh 2012; Yu et al. 2017] or as a part of larger main memory along with other memory devices to form a unified physical address space (viz., as flat-address memory) [Bock et al. 2014; Chou et al. 2014; Islam et al. 2017; Meswani et al. 2015; Prodromou et al. 2017; Ramos et al. 2011; Sim et al. 2014; Su et al. 2015] or as both cache and memory that is dynamically configurable at runtime [Kotra et al. 2018; Su et al. 2015]. In our study, we investigate a flat-address memory system involving faster 3D-DRAM (viz., High Bandwidth Memory—HBM) and slower NVM (viz., Phase Change Memory—PCM). The main challenges for such a memory system are to ensure placement of frequently accessed data in fast memory to improve overall performance and the management of information needed to keep track of page movement with minimal interruptions to user applications.

Most flat-address memory studies have relied on counting accesses to pages and migrating heavily accessed "hot" pages from slow to fast memory. Unlike approaches that rely on epochs (fixed window of time) to track page access counts to determine which hot pages to migrate, we migrate pages as soon as they become hot when they receive a certain number of accesses (hotness threshold). We show that our "on-the-fly" migration performs better than epoch-based page migration techniques, since we migrate recent hot pages. This is in line with the observations made by Prodromou et al. [2017], that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period. For example, consider an epoch of 10ms, and a hotness threshold of 64, and we find 200 pages meeting this threshold by the end of that epoch. An epoch-based approach migrates all the 200 pages at the end of the epoch, halting the user programs. In our on-the-fly migration, we do not wait till we reach an epoch boundary; rather, as soon as the first of the 200 pages reaches the hotness threshold, it will be migrated without halting the user programs. When one migration is done, the second migration will be initiated when another page meets the hotness criteria, and so on. In on-the-fly migration there is no notion of epochs. Since in our approach, migration can take place anytime, it is important to ensure that it does not require halting user program execution. We devise a migration technique that takes place in the background with the help of a specialized Migration Controller (MigC) hardware that it is transparent to the user program.

Because page migration changes physical addresses of migrated pages, we need to reflect the new physical addresses of those pages consistently throughout the system. This requires updating page table entries (PTEs) with new physical addresses, invalidating old translation look-aside buffer (TLB) entries (TLB shootdown), and flushing caches with old physical tags. We refer to this process of updating physical addresses as "address reconciliation" (AR). Usually these operations are performed by OS interrupting user application [Mosberger and Eranian 2001]. OS-based AR may be acceptable in epoch-based techniques [Meswani et al. 2015], since the overhead is amortized across several pages migrated at the end of an epoch, but not for on-the-fly migration, since we migrate one page at a time. TLB shootdown is the primary source of performance bottleneck due to the inter-process interrupt (IPI)-based mechanism used to perform the invalidations in different cores [Awad et al. 2017; Romanescu et al. 2010; Villavieja et al. 2011]. To minimize the impact of address reconciliation, we rely on an additional remapping table, that contains new physical address of a page after page migration so older physical address (as seen by the OS) need not be modified in PTEs and the remapping table is consulted on every memory access. However, the table can grow very large, since it must contain an entry for every migrated page. Some researchers

[Chou et al. 2014; Prodromou et al. 2017; Sim et al. 2014] proposed to keep the remap table inside main memory and use a separate on-chip cache for some entries. This introduces an extra memory access on remap cache miss and may degrade performance. This also reduces the available capacity of the smaller and faster memory in the system, since remap table is placed in faster memory. In our study, we investigate the use of a smaller remap table by periodically evicting the table entries. Instead of relying completely on OS to perform AR for the evicted entries, as done in [Ramos et al. 2011], we propose a hardware-based AR, where the MigC hardware initiates TLB shootdown and cache flushing without explicitly stopping the user program. We propose to employ shared directory-based TLB structure and necessary hardware to perform TLB invalidations [Villavieja et al. 2011]. The MigC also works as a pseudo-processor connected to the cache coherency network of the system and sends "write-invalidation" coherency messages for each cache line of the migrated pages so other caches perform necessary cache line invalidation and also write-back to memory if the line is dirty. Thus, cache flushing can be performed without OS intervention.

Like previous studies [Meswani et al. 2015; Prodromou et al. 2017; Su et al. 2015], we observe that not all applications benefit from page migration, since page migration incurs performance overheads due to extra data movement. We present an offline model for analyzing application's memory access behavior that classifies applications as either *page migration friendly* or not. The model works with the principle that, to get performance benefit, one should migrate the *smallest* set of pages from slow to fast memory that yields in the *largest increase* in memory accesses to fast memory to amortize the migration overhead. We profile memory accesses and evaluate memory localities to identify pages that contribute to majority of memory accesses, and the number of accesses these pages are likely to receive after migrating to fast memory. Depending on such analyses, we categorize workloads (we use application and workload interchangeably) as either page migration friendly or unfriendly. Such offline analyses can provide useful insight on how much performance benefit one may obtain by page migration.

To summarize, in this article, we present a Heterogeneous Memory Architecture (HMA) that relies on on-the-fly (OTF) migration of heavily accessed pages from slower memories to faster memories. Unlike epoch-based approaches, which migrate several pages together at regular intervals (or epochs), we migrate a page immediately when it becomes "hot" to achieve higher number of hits in fast memory. In OTF approach, migration can take place anytime, hence it is important to ensure that it does not require halting user program execution. Therefore, We devise a migration technique that takes place in the background with the help of a specialized hardware that is transparent to the user program and also OS. The physical address of a page depends on the physical location of the page, thus a migration changes the physical address. To locate the migrated pages, either we need some hardware address redirection table (remap table) or we need to update the new addresses of the migrated pages in all necessary system structures involved in virtual to physical address translation. Managing hardware remap table for large memory systems is cumbersome. Hence, page migration requires changes to PTEs to reflect new physical address mappings, invalidating TLB entries and cache entries. We call this process address reconciliation. This address reconciliation process introduces large overheads when managed by conventional OS-based approach. While this overhead may be acceptable for epoch-based approaches, it is not for on-the-fly migrations. Hence, in our study, we propose a low-overhead, efficient on-the-fly technique with the help of a special hardware migration controller that enables migration of pages while reconciling address for different pages in a cycle-interleaving fashion, obviating the need to perform these processes in separate phases. Such ability to perform on-the-fly migration of new pages and address reconciliation of older migrated pages simultaneously is a unique contribution to this field. For a set of page migration friendly SPEC CPU2006 workloads, our on-the-fly migration with hardware-based AR technique results in 74% instruction per cycle (IPC) improvement

on average over a baseline system without any page migration. It also showed 24% improvement when the baseline performs only on-the-fly page migration but employs OS for address reconciliation. Our proposed technique also outperforms state-of-the-art page migration techniques—one that performs AR in a hardware/software mixed approach as proposed by Meswani et al. [2015] by 29% and another one that completely relies on hardware remap table for address remapping as proposed by Prodromou et al. [2017] by 13%.

The key contributions of this article are:

(A) User transparent on-the-fly migration of pages in a flat-address memory (Section 3.2);
(B) An efficient hardware-based approach to manage relocation of pages (Section 3.3);
(C) Characterization of applications' memory access behavior as either page migration friendly or not (Section 4).

2 RELATED WORK

There have been many studies on page migration techniques for flat-address heterogeneous memory systems (HMA) [Chou et al. 2014; Meswani et al. 2015; Prodromou et al. 2017; Ramos et al. 2011; Sim et al. 2014]. They propose different approaches to solve the general challenges associated with page migration, viz., migration candidate tracking, migration frequency, migration metadata management. Meswani et al. [2015] presented a study where page migration in HMA is accomplished by a hardware/software (we refer to it as HMA-HS) mixed approach. The hardware keeps track of the page access counts over a fixed-length epoch, and at the end of each epoch, the hottest pages residing in slow memory are migrated to fast memory by OS, updating physical addresses of the migrating pages. HMA-HS provides full flexibility on page relocation, i.e., a slow memory page can be brought into any location of fast memory and vice versa. Since OS-based address update incurs high overhead, authors choose a longer epoch to reduce frequent OS interventions. However, it has been observed that page migration at smaller intervals is more beneficial than waiting for longer epoch times [Prodromou et al. 2017; Sim et al. 2014], since this can provide more hits in fast memory by migrating early and also adopts well with program phase changes. To minimize OS involvement in page migration, new physical addresses can be kept in an auxiliary remap table that is consulted on every LLC miss. The size and management of this table presents a new challenge. A number of different approaches have been proposed to keep this table in memory while using a smaller on-chip cache for it [Chou et al. 2014; Prodromou et al. 2017; Sim et al. 2014]. Sim et al. have used a Transparent Hardware–based Management (THM) of flat-address memory [Sim et al. 2014]. In THM, a more restricted page movement approach has been suggested to keep the remap table smaller and to facilitate a direct-mapped–like lookup to the table. Here, a fixed set of slow memory pages compete for a fixed, single slot in fast memory. Hence, it is not possible to bring in more than one slow memory page (from the competing set of slow pages) to fast memory, even when bringing all of them would result in better performance. A migration candidate is chosen using a competing counter per set. Kim et al. proposed a somewhat similar idea of intra-set migration named CAMEO [Chou et al. 2014], where the migration is done at finer granularity (cache line size) and the migration candidate is simply chosen on each slow memory access. Prodromou et al. [2017] proposed MemPod, which provides more flexibility on page relocation than Sim et al. [2014] or Chou et al. [2014]. On-chip memory controllers for fast and slow memories are grouped into "Pods" and only intra-Pod epoch-based page migration is allowed. A low cost counter is used to keep track of recently accessed hot pages. The assumption is to have large enough remap table per Pod containing one entry for each memory page of the pod, hence, no explicit address reconciliation is necessary.

In our proposal, we migrate a page immediately when it receives sufficient number of memory accesses, unlike any epoch-based schemes. We allow full flexibility in page relocation like HMA-HS [Meswani et al. 2015] and keep a remap table for address redirection. We keep this table small by periodically evicting entries and it is placed on-chip. A similar approach has been proposed by Ramos et al. [2011] that also performs on-the-fly type of migration with periodical reconciliation of remapping table entries and OS memory mappings. However, in Ramos et al. [2011] the migration and reconciliation processes are separate phases, since the reconciliation is completely handled by OS. In our proposal, we perform address reconciliation with help of specialized hardware, and hence these processes can progress simultaneously. Furthermore, our migration candidate choice scheme is simpler than multi-queue scheme used by Ramos et al. [2011].

In Banshee [Yu et al. 2017], the authors propose a technique for transferring pages between off-chip memory and on-chip DRAM cache. Instead of storing tags for DRAM cache, Banshee uses Tag-buffers, somewhat similar to our remap tables. As the Tag-buffers fill up, the user programs are stopped and OS will update the TLB (and PTE) entries to reflect the new location of pages, again somewhat similar to our address reconciliation. However, unlike our hardware-directed approach, they employ OS for address reconciliation. Finally, Banshee assumes that the two levels of memory (on-chip and off-chip DRAMs) have similar latencies but differ in bandwidth, while we assume memory technologies with significantly different latencies and bandwidths.

NVM such as PCM comes with limited write endurance of 10^6 to 10^8 program/erase cycles and much higher write latency and write energy than DRAM [Qureshi et al. 2011]; therefore, a number of works have investigated how to minimize number of writes to such NVM while using it as part of system main memory. In Qureshi et al. [2009], a small DRAM has been used as buffer for PCM-based main memory to enable lazy-write, line-level-write mechanisms to reduce number of writes to PCM. The Refinery swap [Chen et al. 2017] study proposed algorithms to efficiently swap pages between DRAM and NVM in flat-address memory by addressing the write limitations of NVM. In this study, it was found that there is write count disparity among pages of applications, which showed that most of the writes are generally condensed on a few pages. Therefore, the authors proposed to allow few direct writes to NVM for rarely written pages while employing swap-aware wear-leveling approach. In our work, we target to minimize overall number of accesses to NVM by migrating highly accessed pages to 3D-DRAM; however, we do not specifically focus on minimize writes to NVM. Adding any policy that prioritizes write-intensive pages to be placed in 3D-DRAM can be easily integrated with our hot/cold page selection policy in the future and will be orthogonal with our proposed mechanisms.

The UIMigrate [Tan et al. 2019] study presents an interesting and adaptive policy to choose hot and cold pages from NVRAM and DRAM, respectively, such that it minimizes the number of invalid migrations so pages that do not amortize the cost of migration by providing performance benefits are not migrated. However, this study does not discuss about underlying mechanisms that are needed to actually migrate the pages and find the new locations of the pages, which are among the main focuses of our work.

There are also some works that dynamically configure faster DRAM as cache or as part of flat-address memory as per application need. However, such reconfiguration flexibility comes at the cost of higher metadata overhead and/or additional constraints on page migration locations. In Su et al. [2015] researchers proposed a new memory controller, HpMC (Hybrid Policies Memory Controller), that dynamically analyzes the application behavior and chooses any one of the aforementioned cache or flat-address memory configuration to minimize overall energy consumption. Chameleon [Kotra et al. 2018] proposes a software-hardware co-design that allows the hardware to dynamically reconfigure the parts of 3D-DRAM to operate in flat-address memory or cache mode. The first variant, Chameleon, proposes confining usage of free space in 3D-DRAM to be

used as cache, whereas the second variant, Chameleon-Opt, additionally leverages the free space in off-chip DRAM by remapping segments of 3D-DRAM to off-chip memory to free-up space in 3D-DRAM to be used as cache.

Traditionally NUMA (non-uniform memory access) is observed in multi-socket systems—where each of the sockets has its own memory node (local node), when a thread running in one socket needs to access other sockets' local memory node, which is remote memory node for the requesting thread. Hence, the research on NUMA systems mainly focuses on how to facilitate that all threads can access their required data from their respective local nodes, which is challenging, since once the necessary data is migrated from a remote node to a local node as per a thread's request, the other threads running on the socket of the "remote node" (for the later set of threads that is the local node) will run into the similar issue when they need to access their required data, which is now moved to another node. This phenomenon will be seen when two or more threads running in different sockets need to access shared data. However, in flat-address memory, the non-uniform memory access happens due to the performance variability between the fast but small capacity memory and slow but large capacity memory—which are part of the same local node (from a socket point of view). Hence, in flat-address memory, ideally every thread running in the same socket would like to place their necessary data in the fast but small memory, which is challenging due to the limited capacity of the fast memory, rather than due to any data-sharing issue among threads [Meswani et al. 2015]. Hence, there are some differences between general NUMA system data movement challenges versus flat-address memory data movement challenges. In NUMA studies, there have been different approaches for facilitating proximity of threads and required data; for example, by data affinity—such as migrating necessary data to requesting threads through copy-on-touch policy [Mishra and Mehta 2013]; by thread affinity—such as running threads in the same node that share data [Song et al. 2009; Ştirb 2018]; or by managing both data and thread affinity [Dashti et al. 2013; Diener et al. 2014].

3 SYSTEM OVERVIEW

3.1 System Architecture

We propose a hardware unit, called Migration Controller (MigC), placed on the processor chip to perform actions necessary for our on-the-fly page migration. MigC performs limited communication with OS via reserved registers/memory locations. It swaps hot pages from a slow memory (e.g., PCM) with cold pages in a fast memory (e.g., HBM). Figure 1 shows the high-level system architecture. There are MigC resident hot and cold buffers to temporarily store data from pages as they are being migrated. We assume physically tagged write-back LLC. There is a small remap table that holds new physical addresses (i.e., frame addresses) of the migrated pages. The remap table is consulted on every LLC miss (or on a write-back), using the old physical address to find the new location (if any). The size of the remap table is kept small (e.g., 1,024 entries) so it can be placed on-chip for fast accesses. Whenever the table is full to a certain level, say 50%, address reconciliation process starts, i.e., entries from remap table are deleted and the new physical addresses are made visible to OS as discussed in Section 3.3. There is a Wait queue in MigC, which holds read/write requests from LLC for the currently migrating pages; these requests are likely to be serviced by the hot/cold buffers. The memory controllers (MCs) are equipped with a separate Migration queue (Mig. Q) to service read requests from MigC for the migrating pages.

3.2 User Transparent Page Migration

We assume a flat-address space where physical memory addresses are statically and linearly assigned across the different memory devices (viz., HBM and PCM), and thus the MigC can determine

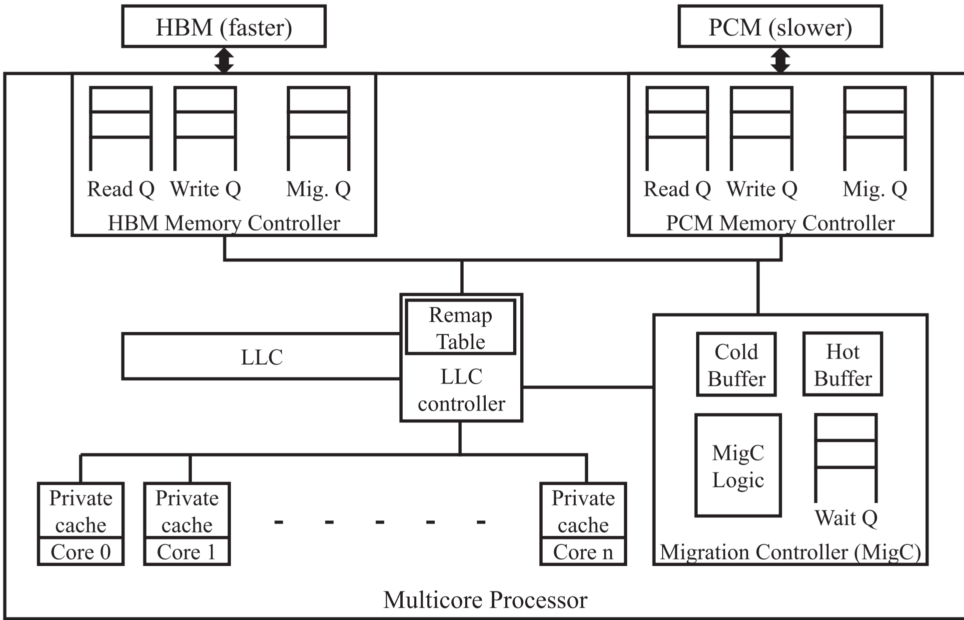


Fig. 1. High-level system architecture.

correspondence of physical address to a memory device. We migrate a hot page from PCM into a free frame of HBM (one-way migration); in case there is no free HBM frame, we choose an LRU HBM cold page and swap the hot and cold pages (two-way migration). MigC performs one migration at a time. We assume that MigC knows current access counts for all pages as well as free/LRU list of HBM frames. For explanation, we label the steps involved in a page migration process as M1, M2, and so on.

(M1) At time T1, MigC finds that a PCM page meets the hotness threshold for migration; for illustration, say a page with physical address (PA) 8192—let us call this hot page "A." Note that, in all our explanations when we mention a "page," we just mean the content of a physical frame: We do not have/keep any information about virtual addresses of the pages unless otherwise mentioned. MigC then finds a cold page from HBM to swap with the hot page. We call this cold page "B" and assume it has PA 0. MigC starts the migration process and notifies OS through reserved memory mapped registers not to replace/reclaim these pages. These interactions can be accomplished by the following means: MigC communicates the page frame number (pfn) of the pages that it is going to migrate to OS via reserved registers. OS then sets a flag in page frame metadata to indicate the page is being migrated by MigC. For example, Linux page frame descriptors include a set of flags to keep information about the frame [Bovet and Cesati 2005] and such descriptors can be used for this purpose. Linux macro *pfn_to_page(pfn)* can be used to obtain the address of the descriptor of page frame with number *pfn*. However, this communication is overlapped with other actions MigC performs preparing for the page migration. MigC also ensures that these pages are not already migrated, i.e., they are not present in the remap table. The same page can be migrated back and forth between memories as long as the page went through address reconciliation before being moved again. MigC inserts entries for pages A and B in the remap table with their current OS-visible PA and future new PA (after migration). Remap table is always looked up using OS-visible (original) PA. Mig flag is set to 1 to indicate that the pages are under migration and Pair flag is set to 1 to indicate the migration is taking place between a hot and cold pair. As the hot and cold page

OS visible PA	New PA	Mig	Pair	Buffer residency	Bit vector
8192	0	01	1	1 (hot)	00........00
0	8192	01	1	0 (cold)	00........00

Fig. 2. Status of remap table after migration step M1.

pair exchange locations, the partner in the pair can be looked up in the remap table using the New PA of current entry. The Pair flag will be checked later during the AR process to see if this entry was involved in a swap (Pair = 1) or just a one-way migration (Pair = 0). Buffer residency flag indicates the location of hot and cold buffers in MigC used during the migration. Figure 2 shows the state of the remap table at this point.

(M2) MigC waits for any pending read requests for page A that were already issued by time T1. After completion of these pending requests, say at time T2, MigC starts reading hot and cold pages into their respective buffers at cache line granularity. Any new requests (after time T1) for these pages from LLC (on miss or dirty eviction) will be held in MigC resident Wait queue and will be served from these buffers. Assume that the reading of these pages is completed by time T3. In between T2 and T3, each MC involved in page migration serves requests in migration queue (Mig.Q) with the highest priority.

(M3) At time T3, MigC starts writing contents of buffers to the page frames (new PAs) and Mig flag is set to 2. A bit vector kept in the remap table indicates which lines have already completed migration. At this point any new memory requests for these pages will be sent to new addresses (New PA + line offset) if the bit is 1 in the bit vector, else it will be served from buffers depending on the Buffer residency flag.

(M4) At time T4, when migrations are completed, the Mig flag for these pages will be reset. All future requests for these pages will be directed to proper new locations based on the remap table information.

3.3 Address Reconciliation

Tracking all migrated pages during the lifetime of an application can require a prohibitively large remap table. We use a small remap table and periodically evict old entries to make room for new entries for future migrations. To remove an entry from the remap table, it is necessary to reconcile physical addresses to reflect the new location of the page consistently throughout the system using "address reconciliation" process (AR). We reconcile entries from the remap table pairwise when we find the Pair flag is 1. Since the hot and cold page pair exchange physical locations during migration, the new physical addresses for them must be updated together in the system for correct cache accesses. The following actions must be performed to ensure correctness of AR: We use the same example hot and cold page pair, A and B, respectively. First, all cache lines from these pages, which are currently residing in the cache hierarchies and tagged with OS-visible PA, must be invalidated (and dirty lines written back), since the current OS-visible PA will be replaced with the new PA. All future accesses to these pages will only have access to the new PA. Next, corresponding page table entries (PTEs) for A and B need to be updated with new PAs. The TLB entries in all cores using the old PA must also be invalidated (as well as any other OS structures that contain the physical page addresses).

3.3.1 Address Reconciliation: OS vs. Hardware. Linux performs the following functions when the virtual to physical address mapping of a page is changed: (i) *flush_cache_page()*, (ii) change PTE, and (iii) *flush_tlb_page()* [Mosberger and Eranian 2001]. The function *flush_cache_page()* takes necessary parameters (a pointer to the process address space, the virtual address, and associated

page frame number) and writes back any dirty cache lines of that page to memory and invalidates the cache lines belonging to that page. This process halts the user program resulting in large overhead. We found that on average it takes 4 μs to flush cache lines of a page using CLFLUSH x86 instruction in a real machine running at 2.26 GHz. Next, to update PTE, Linux acquires page table lock and changes PTE and also executes *flush_tlb_page()* to invalidate all TLBs with old VA to PA translation. After successful TLB shootdown, it releases page table lock. The TLB shootdown is costly, because it uses IPI to invalidate TLB entries in every core that contains an entry with old PA. The delay grows non-linearly with the number of cores [Awad et al. 2017; Romanescu et al. 2010; Villavieja et al. 2011]. As reported in Meswani et al. [2015], TLB shootdown may take up to 4, 5, 8, and 13 μs for 4, 8, 16, and 32 cores, respectively, on an AMD 32-core system running Linux.

In our hardware-based approach, we configure the MigC as a pseudo-processor that can send "write invalidate" coherency requests over the coherency network for each of the cache lines of the pages under reconciliation, requiring all caches to write-back any dirty lines to memory and invalidate their cache lines for these pages. MigC will be configured such that it can send coherence requests to other caches and receive acknowledgments back from them; however, other caches will never send requests to, or wait for, any acknowledgments from MigC. Hence, it will not add to the traffic during regular coherence operations of the system. For efficient TLB-shootdown, we rely on a shared TLB directory that contains all the private TLB entries along with process identifier (i.e., address space identifier) and core residency information. MigC initiates TLB shootdown by sending the associated VAs to the shared TLB directory. The shared TLB directory then maps these VAs to necessary entries and send probes to cores to invalidate these TLB entries similar to what was proposed in Villavieja et al. [2011]. We envision that actual invalidation at the core will be carried out by a per-core hardware invalidation controller without interrupting the core, and the upper bound of time required for completing such invalidation at core is assumed to be a round-trip off-chip memory access latency [Villavieja et al. 2011].

3.3.2 One Final Issue in Address Reconciliation. To update PTE to reflect the new physical address of a migrated page and invalidate associated TLB entries, we need the virtual address (VA) of the page. However, the remap table contains only the original PA of a page and not its VA. Moreover, since the same page can be shared by multiple processes and each process may have a different VA corresponding to the PA of the page, we need to obtain all the possible VAs to perform necessary changes. Linux keeps descriptors for every allocated physical page frame that maintains bookkeeping information such as a set of flags describing the page frame's status, number of PTEs referring to this page frame, and indirect pointers to such PTEs [Bovet and Cesati 2005]. By using existing reverse mapping function Linux can provide a list of PAs of PTEs that hold mappings to this specific page frame number, and the associated VAs with ASIDs [Bovet and Cesati 2005]. We rely on these OS functions to obtain the VAs of the pages involved in AR (see Section 3.3.3). In our experiments, we have accounted for all the delays involved for these OS functions based on previously reported numbers and actual experimental data on real system.

3.3.3 Hardware-centric Address Reconciliation. Using our running example hot page A (PA:8192) and cold page B (PA: 0), we present the steps involved in AR. The steps are labeled as R1, R2, and so on. A high-level description can be found in Figure 3.

(R1) At time T5 (where T5>T4), MigC requests OS to send the associated VAs and ASIDs for the OS-assigned PAs 8,192 and 0. OS, using the reverse mapping function [Bovet and Cesati 2005], will find the appropriate PA to VA mappings and return them to MigC. MigC will also instruct OS to update the associated PTEs and other necessary structures that keep information about page frames, with the new PAs and not to allow any further access to the associated PTEs. OS will first lock the PTEs of A and B and then update them with new PAs 0 and 8,192, respectively, and block

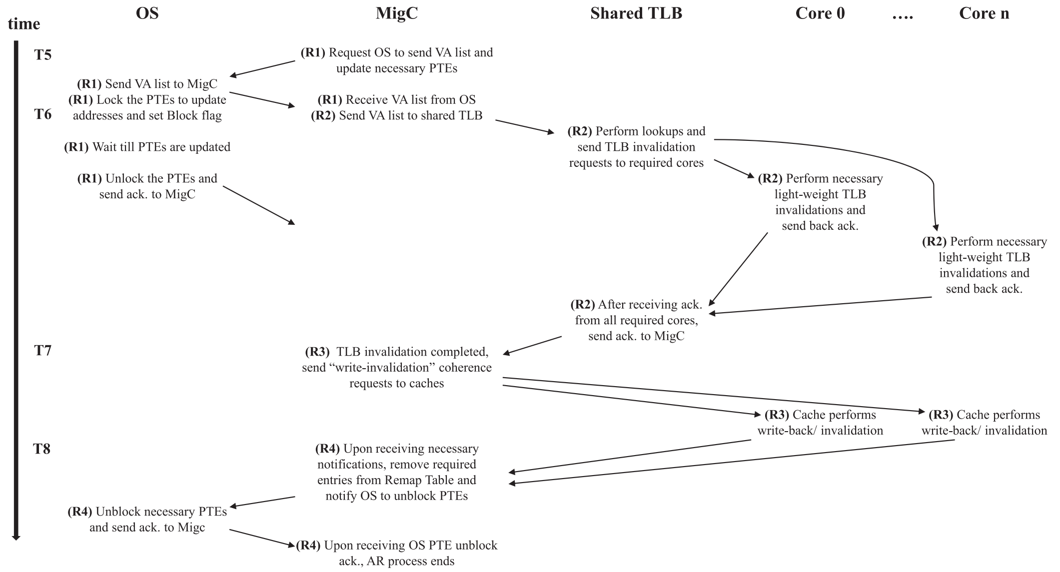


Fig. 3. High-level work flow of address reconciliation process (AR).

them from use. This blocking might be accomplished by setting a reserved bit in PTE to let both OS and hardware Memory Management Unit (MMU) know that the current translation is not valid and not to proceed with normal page fault operation. As a result, any new TLB fill requests for such associated virtual addresses will be pending as long as the PTE is blocked. Cores, which have valid TLB entries for pages A and B, will still be able to execute new memory requests for these pages.

(R2) At time T6, MigC will send the list of VAs along with ASIDs to the shared TLB directory and instruct it to invalidate TLB entries. The TLB directory will then initiate necessary actions to invalidate all involved TLB entries by sending invalidation requests to cores. The cores will then perform lightweight TLB invalidations in their private TLBs and notify the shared TLB directory [Villavieja et al. 2011]. At this point in time, we have stopped any new memory requests to be executed for pages A and B, since the VA to PA translation process for these pages is blocked. Hence, any core that needs to access the pages A or B at this point will have to wait till the process completes.

(R3) Next, at time T7, we are left with only the previously cached lines of the pages under AR, which are still tagged with OS-assigned PA (viz., cache lines of pages A and B, residing in the cache hierarchy, are still tagged with PAs 8,192 and 0, respectively). MigC will send "write-invalidate" coherence requests for all the cache lines of these pages so these lines are invalidated (and written back to memory if dirty). In our organization, we use a shared inclusive LLC, hence the coherence requests will be directed to the LLC. Note that, at this point the remap table entries for pages A and B are still present, hence the write-backs from the caches will be written to proper memory locations. Let us assume this step completes by time point T8.

(R4) At time T8, upon receiving notifications of completion of PTE updates from OS and cache invalidations from the coherency network, MigC removes the entries for pages A and B from the remap table. Then MigC notifies OS to unblock the corresponding PTEs that were blocked in step R1. Once OS completes unblocking, it will notify the MigC. Upon reception of this OS notification, the MigC completes the address reconciliation process.

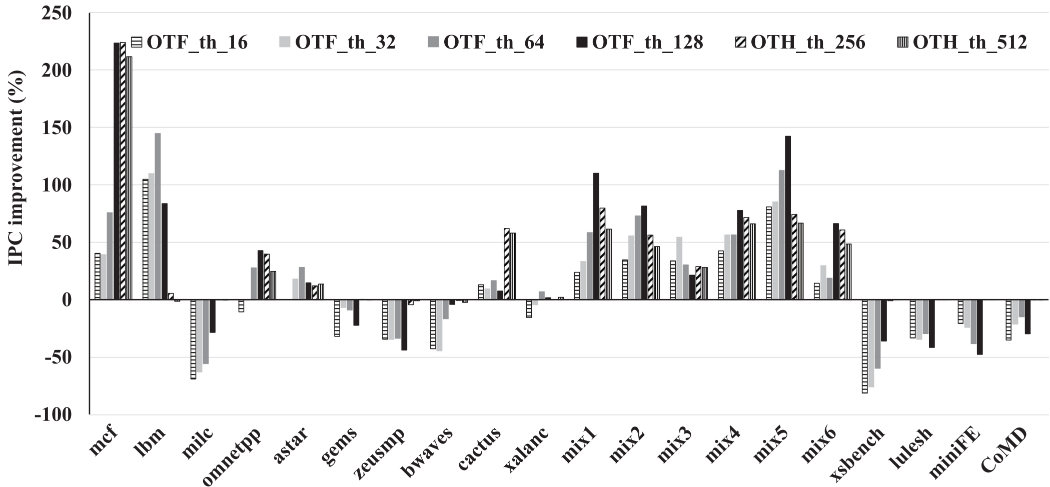


Fig. 4. Results of access-count based page-migration over no-migration baseline (negative y-axis shows degradation).

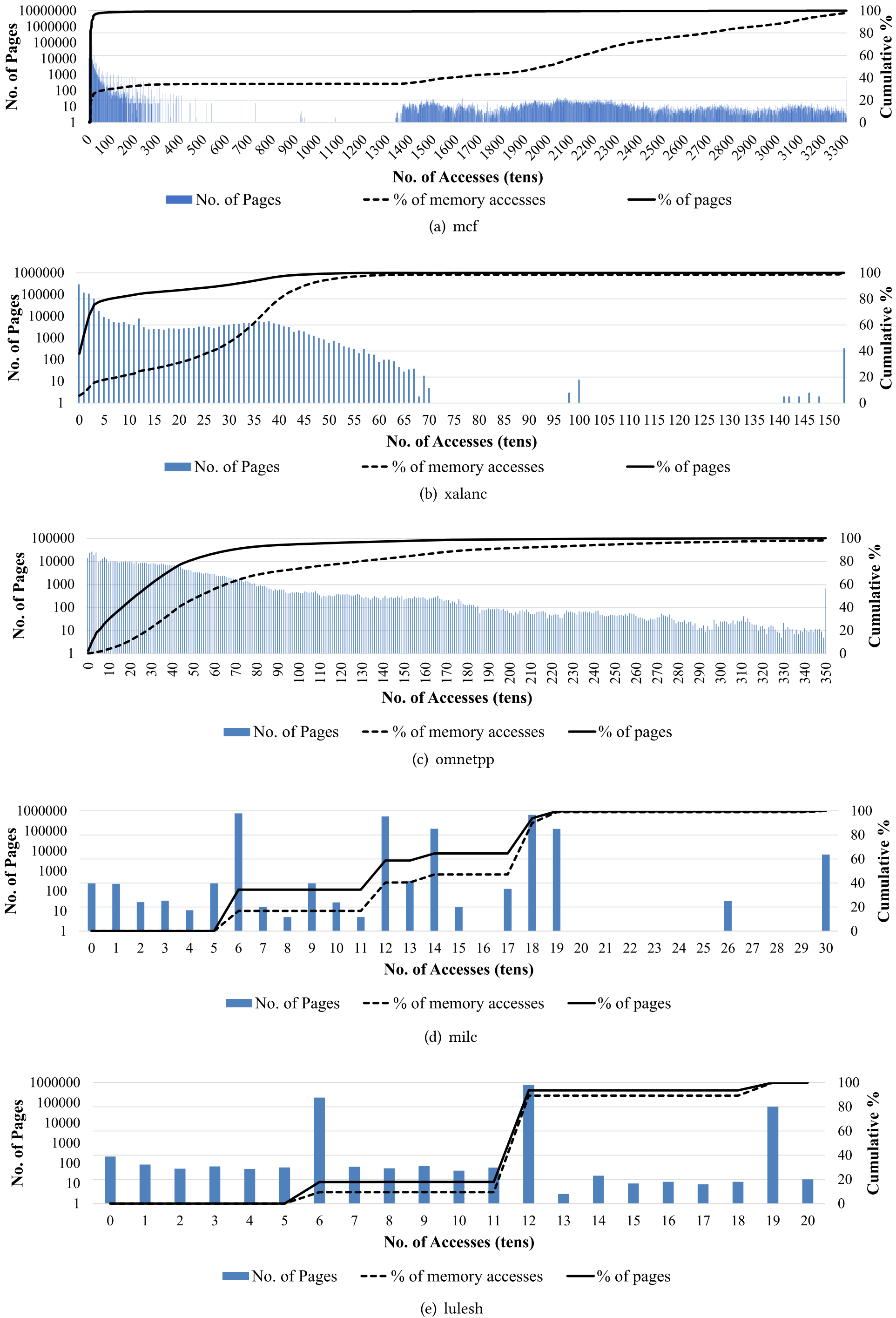
Since AR freezes the VA to PA translation process for the pages under reconciliation, user processes cannot execute any new read/write requests to these pages. Therefore, when a page is hot, we prefer not to start AR for that page. Note that, MigC performs page migration and address reconciliation in a cycle-interleaving fashion, but for different set of pages.

4 PAGE MIGRATION FRIENDLINESS

In Figure 4, we present our initial results of performing on-the-fly page migration (OTF) for a flat-address memory consisting of HBM and PCM for a number of different workloads (Section 5 provides details of our experimental setup and workloads). The positive (negative) y-axis values show relative improvement (degradation) of instruction per cycle (IPC) as a percentage when compared to the same flat-address memory but without any dynamic page migration. Here, we have used different static hotness threshold values (16, 32, 64, 128, 256, 512) to find the maximum improvement that may be achieved. For example, using a threshold of 256 accesses, mcf shows 224% IPC improvement. Here, we assume a sufficiently large on-chip remap table and hence, no address reconciliation is performed. We make this choice to determine only the impact of page migration. We observe that not every application benefits from migration, and even with those that do, there is a large variation in the improvements: We classify improvements as high ($max > 70\%$), medium ($30\% < max <= 70\%$), low ($5\% < max <= 30\%$), and negligible or negative ($max <= 5\%$). The specific ranges for high, medium, and low are based on our experimental setup that may be different under other system configurations.

To understand these variations and determine which applications benefit from page migration, we present an offline analytical model for memory access behaviors. Our model classifies applications as either “page migration friendly” or not. The model is based on empirical data that we gathered from our experiments and observations. The specific thresholds and cut-off points described below are derived from our experiments, and may be different for different system setups and migration techniques with different overheads.

In Figure 5, we present page access histograms for a subset of workloads from our study. The x-axis plots the total number of memory accesses to a page (in multiples of tens). The vertical bars



(a) mcf

(b) xalanc

(c) omnetpp

(d) milc

(e) lulesh

Fig. 5. Histogram graphs for memory access counts (primary y-axis) with cumulative percentage of memory accesses and page usage (secondary y-axis).

correspond to the primary y-axis, and each bar represents the number (count) of pages (in log10 scale) that received a given number of memory accesses. For example, in Figure 5(a), the very first bar indicates that the number of pages with 0–9 accesses is very large. The cumulative number of pages, i.e., the combined number of pages that received all the memory accesses to the left of a specified x-axis value, is also shown in the figure as a solid line and corresponds to the secondary y-axis. The cumulative number of pages is shown in percentage with legend entry "% of pages" in the figure. The cumulative percentages of memory accesses coming from these cumulative pages are shown as a dashed line and it also corresponds to the secondary y-axis. The legend entry for this is "% of memory accesses" in the figure.

The goal of any page migration in a flat-address memory is to achieve better performance by directing most accesses to the fast, high-bandwidth memory and fewer access to the slow memory by migrating heavily accessed pages to fast memories. But migrating pages between different memories incurs performance and energy overheads. An effective strategy is to migrate a small number of pages that results in a large increase in accesses to faster memory.

In our study, we look at the 80-percentile[1] accesses and identify the "set of top-accessed pages" that contribute to more than 80% of all memory accesses. The particular access count, which can separate such top-accessed pages from other pages, is referred to as "filter count." Note that, filter count indicates an upper bound for hotness threshold, but as we will see in Section 6, an ideal hotness threshold may differ from filter count due to other runtime characteristics of the workload. The size of the set of top-accessed pages is used to determine the application's memory access locality.

(A) Highly localized: set of top-accessed pages is 30% or fewer of the total pages. For example, in Figure 5(b) (xalanc), pages with fewer than 70 accesses (the bars corresponding to x-axis 0 to 6) contribute to only 19% of all accesses, and they account for 80% of the pages. That is, the remaining 20% of pages contribute 81% of memory accesses. Migrating this small number of pages may improve performance. Here the filter count is 70.

(B) Moderately localized: set size is 31% to 55% of the total pages. For example, in Figure 5(c) (omnetpp), using a filter count of 270 accesses, we can separate the top 45% of pages (the bars starting at x-axis 27 and onward have access count 270 or more) that contribute to 82% of all accesses. Overheads for migrating this many pages may cause only moderate performance gains.

(C) Least localized: set size is 56% to 70% of the total pages. In Figure 5(d), consider milc, using a filter count of 120 accesses, we can separate top-accessed 65% of pages that account for 83% of memory accesses. Migration overheads in such applications is likely to offset any benefits of page migrations.

(D) Distributed: set size is 71% (or more) of the total pages. In this case, memory accesses are more evenly distributed among pages. For example, consider lulesh in Figure 5(e). Here, using a filter count of 120, we can only separate top-accessed 82% of pages that account for 90% of memory accesses. Such applications do not benefit from page migration (or are migration unfriendly).

Next, we use a Migration Benefit Quotient (MBQ), by calculating the difference between the total number of accesses to any page and the filter count used in classifying applications' memory locality, as described. MBQ indicates how many of the future accesses of a page may go to fast

[1] As stated before, specific values and cutoff thresholds are based on our empirical observations, and different values and thresholds may need to be used under different setups.

Table 1. Classification of Page Migration Friendliness

MBQ / Locatity	High	Medium	Low
Highly Localized	Very friendly (high improvement: mcf, mix1, mix2, mix4, mix5)	Moderately friendly (-)	Less friendly (Low improvement: xalanc)
Moderately Localized	Moderately friendly (-)	Moderately friendly (Medium improvement: omnetpp, cactus, mix3, mix6)	Less or unfriendly (Negligible or no improvement: gems, xsbench, CoMD)
Least Localized	Less friendly (-)	Less or unfriendly (Low improvement: astar)	Unfriendly (Negligible or negative improvement: milc)
Distributed	Unfriendly (Negligible/negative improvement: zeusmp, bwaves, miniFe, lulesh)		

memory if the page were migrated from slow to fast memory using the filter count as a hotness threshold. In Figure 5(a), the sum of access counts of all pages with 32,909 accesses or less (the bars corresponding to x-axis 0 to 3,290) accounts for 98% of all accesses. For our purposes, we use this access count as a saturation count (or assume that the maximum number of accesses any page can receive). For each workload, we use the difference between the saturation count and the filter count to determine MBQ.

(A) Low MBQ, difference is less than 1K. For example, in Figure 5(b) xalanc, pages with memory access count 609 or less (the bars corresponding to x-axis 0 to 60) provide 98% of the memory accesses. Hence, the difference between saturation count (609) and filter count (70) is 539. These workloads may not achieve significant increase in accesses to fast memory after page migration.
(B) Medium MBQ, difference is in between 1K to K (e.g., Figure 5(c) omnetpp). These workloads may receive moderate benefits, depending the migration overheads.
(C) High MBQ, difference is more than 8K (e.g., Figure 5(a) mcf). These workloads are likely to receive higher hits in faster memory as a result of page migration.

We summarize our characterization of page migration friendliness in Table 1 and list the workloads in each category based on initial on-the-fly migration experiments. Using our static analysis of page access counts, we were able to correctly classify 19 out of the 20 workloads used in our study. The only outlier is lbm, which should be page migration unfriendly as per our analytical model. This is because, lbm is moderately localized and has low MBQ; however, our experiments show that it provides high performance improvements; it appears that lbm has some specific pattern of page accesses that appreciates our migration principle; for example, it accesses pages in bursts (or small reuse distance), so whichever page is brought into fast memory that can be fully exploited before being evicted as a cold page to make room for another hot page. Hence, we feel it is important to investigate application memory behavior from different perspectives to predict its page migration friendliness.

Our classification also largely holds for other access-count-based page migration techniques such as HMA-HS [Meswani et al. 2015] and MemPod [Prodromou et al. 2017] (details are provided in 6.1), and hence, we believe that our approach to understanding migration friendliness is likely to be true for other techniques that use page access counts to trigger migration.

Table 2. Baseline Configuration

Parameter	HBM	PCM
Channels, capacity	8, 1 GB (8 × 128 MB)	2, 16 GB (2 × 8 GB)
Memory Controller (MC)	1/channel	1/channel
Row buffer	2 KB	2 KB
Queue size/MC	RD 32, WR 32, Mig. 32 entries	RD 64, WR 256, Mig. 32 entries
Latency	tCAS-tRCD-tRP-tRAS: 14 ns-14 ns-14 ns-34 ns	Read 80 ns (7.5 ns tPRE + 62.5 ns tSENSE + 10 ns tBUS) Write 250 ns tCWL
Bus/channel	128 bit, 1 GHz	64 bit, 400 MHz

Table 3. Timing Parameters at 3.2 GHz Clock

Task	Time Requirement
Remap table lookup	10 cycles (after LLC)
Light-weight TLB invalidation at core	300 cycles (round trip latency to off-chip memory [Villavieja et al. 2011])
Page walk	150 cycles
OS reverse mapping	4,480 cycles (measured using Ftrace [Bird 2009] on a real machine running Linux)

5 EXPERIMENTAL SETUP

5.1 Simulation Infrastructure

We model a 16-core system with a flat-address memory consisting of 1 GB of HBM and 16 GB of PCM using Ramulator [Kim et al. 2015]. Ramulator is a trace-driven, cycle-level memory simulator with support for a simple multicore CPU model with cache hierarchies. Each core is 4-wide out-of-order issue with 128 ROB entries and operates at 3.2 GHz. The cores have private L1-D caches (32 KB, 4-way, 2-cycles) and shared L2 (16 MB, 16-way, 21-cycles) as LLC. All caches are physically tagged, write-back, and LLC is inclusive of the L1s. Ramulator does not model L1-I cache, and assumes non-load/store instructions are executed in one cycle. The memory system configuration is provided in Table 2; for timing parameters of HBM, we rely on Kim et al. [2015] and for PCM on Nair et al. [2015].

We modify Ramulator to support heterogeneous memories as a flat-address memory system. A basic address mapping function is added to Ramulator to support this model; it allocates addresses (i.e., pages) to frames of different memories in a round-robin fashion (viz., 4 to faster memory, then 4 to slower memory), as long as there are free frames in faster memory. When faster memory capacity is exhausted, only slower memory frames are assigned. This allocation ensures that pages for all applications span both memory devices and thus necessitating page migration considerations in our experiments. We incorporate our MigC unit in Ramulator with all necessary details to perform functions as described in previous sections. MigC also operates at 3.2 GHz. We assume conventional 4 KB pages.

The remap table is implemented as a 1,024 entry fully associative table (tables with 2 K or 4 K entries do not result in significant changes in performance). The remap table access latency and energy can be minimized using entry-prediction mechanisms such as bloom filters; however, in this study, we conservatively have assumed an access latency of 10 CPU cycles. We included all timing overheads (listed in Table 3 assuming 3.2 GHz clock rate) for performing different HW/OS

Table 4. SPEC Multi-programmed Mix Workloads

Benchmarks	mix1	mix2	mix3	mix4	mix5	mix6
astar	2x		1x			1x
bzip2		1x	1x	2x		
cactus		2x	2x	1x		
dealII		3x	1x	1x		
gcc	1x		2x	1x		3x
gems		2x	2x	1x		
lbm	2x	3x		1x	6x	1x
leslie			2x	1x		
libquantum	2x		1x	3x		4x
mcf	3x	2x		1x	5x	
milc	2x		2x	1x		2x
omnetpp	1x					3x
soplex	2x	3x		3x	5x	
sphinx	1x		2x			3x

tasks as mentioned in our on-the-fly model. We also implemented other page migration studies including HMA-HS [Meswani et al. 2015] and Mempod [Prodromou et al. 2017] for comparison purposes. We compare these systems with a baseline that does not migrate pages across HBM and PCM and uses the aforementioned round-robin address mapping policy.

5.2 Workloads

We use 16 multi-programmed SPEC CPU2006 [spec 2015] workloads and four multi-threaded representative HPC workloads using the ECP Proxy Applications [Energy 2018] provided by the US Department of Energy (DOE). All of our workloads have large memory footprints, at least twice the capacity of HBM. SPEC benchmarks allow us to compare our work with other studies. We profile several memory-intensive, single-threaded SPEC benchmarks using PinPlay kit [Harish Patil 2018] to collect a representative slice of 500M instructions from each of the applications. To make a multi-programmed workload, we run a 16-core Ramulator simulation where each core runs one of the SPEC traces to completion. We either run 16 copies of the same benchmark on 16 cores (each such workload is labeled by the benchmark name) or run a random mix of benchmarks on 16 cores (these workloads are labeled as mix1 to mix6 and described in Table 4). The publicly released multi-threaded HPC proxy benchmarks by the US Department of Energy (DOE) that we used are: XSBench [Tramm et al. 2014], LULESH [Hornung et al. 2011], CoMD [Mohd-Yusof et al. 2013], and miniFE [Heroux and Hammond 2015]. We ran each HPC benchmark in a 16-thread setup and collected 500M instruction traces for each of the threads using Pin tools [Osnat Levi 2018]. By running traces of the 16 threads of a HPC benchmark in Ramulator, we obtain a multi-threaded workload (each such workload is labeled with the name of the benchmark). The memory footprint of the workloads range between 2 and 11 GB, ensuring the workloads fit in physical memory and do not require access to secondary storage, yet create pressure on the HBM memory.

6 EVALUATION

We evaluated performance of our on-the-fly page migration (OTF) without any Address Reconciliation (OTF_no_AR), which assumes no AR is required, since there is a sufficiently large on-chip remap table. Then, we evaluated OTF with OS-based AR (OTF_OS_AR) and OTF with our proposed hardware-based AR (OTF_HW_AR). We compared OTF schemes with an epoch-based page

migration study HMA-HS [Meswani et al. 2015] that employs OS-based AR; we refer to this as HMA_HS_OS_AR. We also compare our technique with the MemPod study [Prodromou et al. 2017], where no explicit AR is necessary; since they assume large remap tables, we refer to it as MemPod_no_AR. For all the OTF schemes presented in this section, we use a hotness threshold of 128. We will show results for different thresholds in our sensitivity analysis in Section 6.3.

We keep count of memory accesses to each PCM page. For OTF schemes, whenever we find a page with access count >= hotness threshold, we start migrating the page and reset its access count to 0. Previously, we have shown the results of using different static hotness thresholds (e.g., 32, 64, 128) in Figure 4. However, we observe that by simply using one more comparator, we can get even better results. As each migration takes several hundreds of cycles, we use this span to select the PCM page with the highest access count satisfying the hotness threshold, i.e., we now select the PCM page with access count >= hotness threshold and access count >= current highest count. We reset the current highest count to hotness threshold at the beginning of each migration and keep updating it with the access count of any PCM page satisfying the condition until we start another migration. Results, using this approach to select the next PCM page with current highest hotness for on-the-fly migration, are presented in Section 6.1.

For both OTF_no_AR and MemPod_no_AR, we assume sufficiently large on-chip remap tables and hence no AR is required. HMA_HS_OS_AR does not need any remap table, since OS performs necessary AR at each epoch boundary when the pages are migrated. For both our OTF_OS_AR and OTF_HW_AR schemes, we use a 1,024 entry remap table and start the AR process whenever the table is 50% occupied. We stop migration if the remap table does not contain free entries, and wait for AR to free up space. This may happen when the rate at which MigC performs page migration is more than the rate of AR. Such a scenario takes place when there is more memory pressure, i.e., more hot pages are found. We insert at the tail and evict from the head, treating the remap table as a circular buffer. We start evicting when the table is 50% full, thus not evicting entries corresponding to recently migrated pages.

In our implementations of HMA-HS [Meswani et al. 2015] and MemPod [Prodromou et al. 2017], we mostly used the best configuration parameters as recommended in the studies. In HMA_HS_OS_AR, we used a static threshold of 128 (as it showed better performance than the recommended threshold 32) and epoch length of 100 ms. In HMA-HS, migrations are performed at typical 4 KB page granularity. Here, to minimize the delay overheads, we have assumed all the page migrations and AR steps run in parallel at each epoch interval. We assumed no additional overhead for hot page sorting. In our implementation of MemPod_no_AR, given our memory system setup with 8 HBM MCs and 2 PCM MCs, we created two "Pods" each clustering 4 of the HBM MCs and 1 PCM MC. In each of the Pods, we used 64 MEA counters, and the epoch length is set to 50 μs [Prodromou et al. 2017]. MemPod migrates 2 KB pages.

6.1 Performance Analysis

In this section, we assume that we have accurate information about access counts per page while choosing a migration candidate (hot page) and a complete LRU list of frames for fast memory for choosing an eviction candidate (cold page). The other studies we have compared here are assumed to have similar advantages. We understand that maintaining such accurate information is not practical, so in Section 7, we will present performance results using approximate information that is more practical. At this point, our aim is to study the maximum performance gains possible without such limitations, yet accounting for migration and AR overheads.

In Figure 6, we show the results for all our workloads for different page migration and AR policies, when compared to a baseline without any page migration and any AR. The positive (negative) y-axis shows IPC improvement (degradation) as a percentage. The page migration friendly

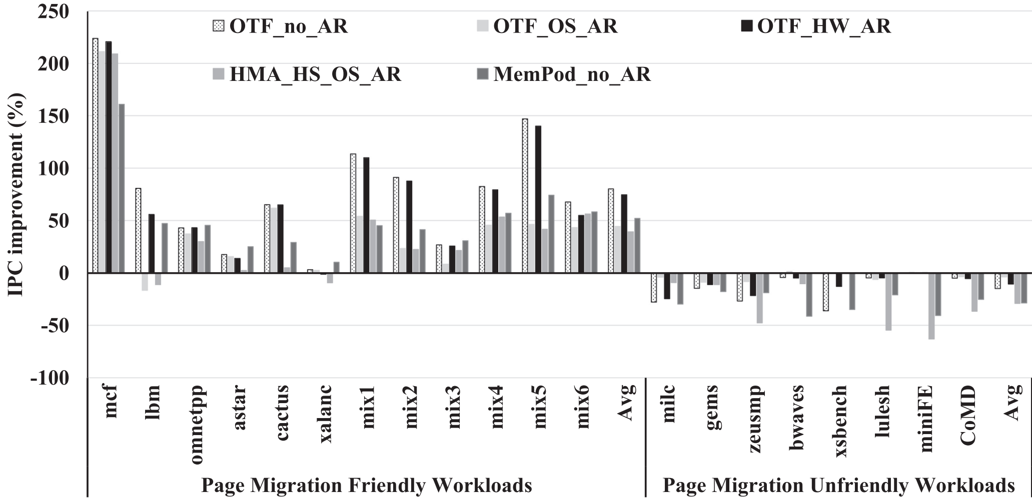


Fig. 6. IPC improvement (%) of different page migration and address reconciliation policies over no-migration baseline (negative y-axis shows degradation).

workloads are mcf, lbm, omnetpp, astar, cactus, xalanc, and mix1 to mix6, and page migration unfriendly workloads are milc, gems, zeusmp, bwaves, xsbench, lulesh, miniFE, and CoMD.

In summary, we find that for a set of page migration friendly workloads, our on-the-fly migration with hardware-based AR technique (OTF_HW_AR) results in 74% of IPC improvement on average over a baseline system without any page migration. It provides 24% IPC improvement on average over on-the-fly page migration with OS-based address reconciliation (OTF_OS_AR). It also provides higher performance improvements on average over other page migration techniques—HMA_HS_OS_AR (by 29%) [Meswani et al. 2015] and MemPod_no_AR (by 13%) [Prodromou et al. 2017]. Also, we have included results for even the migration unfriendly workloads to show that all HMA techniques degrade performance for these workloads, not just our on-the-fly technique, thus confirming that the characterization of application as migration friendly and unfriendly applies to most page migration techniques.

First, we compare the different page migration policies where no AR is performed, viz., OTF_no_AR and MemPod_no_AR. For two-thirds of the page migration friendly workloads, OTF_no_AR performs better than MemPod_no_AR. For the workloads omnetpp, astar, xalanc, and mix3, MemPod_no_AR performs better mainly due to having the benefit of migrating data at smaller granularity, which is 2 KB as compared to migration of 4 KB sized pages by OTF_no_AR. We have also evaluated OTF_no_AR with 2 KB page migration granularity, and found that it performs better than MemPod_no_AR for most of the workloads. Figure 7 shows performances for OTF_no_AR and MemPod_no_AR when both use 2 KB pages compared over a baseline using 2 KB pages without any migration. With smaller migration granularity (2 KB), there is more flexibility to migrate only the "hot" subpages in memory that are getting more accesses, when there is not enough spatial locality among the hot regions. Whereas with larger granularity (4 KB), we may have to migrate pages that are only partially hot. However, we have chosen to migrate in typical page sizes so we can be able to perform address reconciliation at OS level, which obviates the need of a huge remap table. It is important to note that, for more than half of the migration friendly workloads, even with all address reconciliation overheads accounted for hardware-based AR in OTF_HW_AR, it performs better than MemPod_no_AR, which does not pay any cost for address reconciliation assuming a large on-chip remap table.

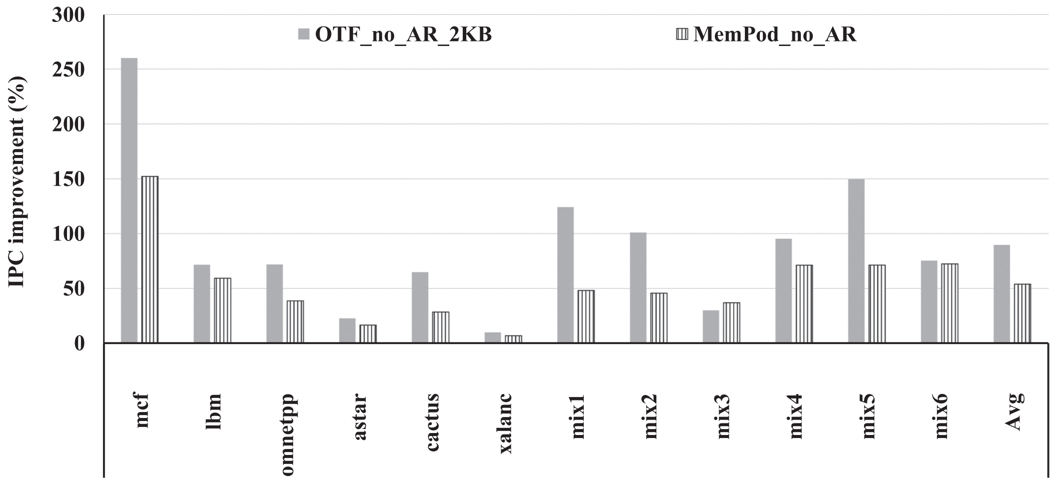


Fig. 7. Performance comparison when OTF_no_AR also migrates at 2 KB granularity as MemPod_no_AR.

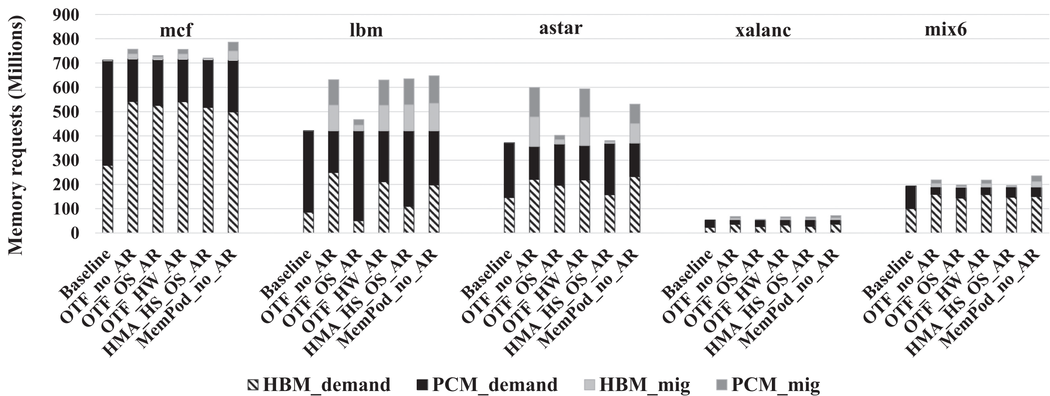


Fig. 8. Demand and page migration traffic to memory for different policies.

Next, to assess our hardware-based AR over OS-based AR. We compare OTF_HW_AR with OTF_OS_AR, both of which have the same underlying OTF migration scheme. For all the page migration friendly workloads, except astar and xalanc, OTF_HW_AR performs better than OTF_OS_AR, because of the lower overheads of hardware-based scheme, as expected. Since our hardware-based AR has less overhead than OS-based AR, it generally allows more page migrations and thus provides better performance improvements. However, for these two workloads, more migrations accomplished by OTF_HW_AR were not helpful, because they are less migration friendly workloads as found by our analytical model in Section 4. astar is least localized, which means it has a large number of pages that only provide moderate number of accesses. However, xalanc has low Migration Benefit Quotient (MBQ), i.e., after crossing the thresholds the migrated pages are not heavily accessed. In Figure 8, we show the demand and migration data traffic analysis for these workloads. We will explain this figure next in 6.2.

Next, we compare where both page migration and address reconciliation are done in an epoch-based approach, HMA_HS_OS_AR to our on-the-fly approach. As shown in Figure 6, OTF_HW_AR performs better than HMA_HS_OS_AR for all the page migration friendly workloads except mix6. As we analyze the data traffic behavior for mix6 in Figure 8, we again find that

OTF_HW_AR migrated a large number of pages more than HMA_HS_OS_AR, which in return could not maximize the benefit.

OTF migration is a dynamic process—it is not bound by number of epochs; therefore, it may end up migrating a large number of pages that might not provide the optimal benefit after paying the migration and address reconciliation overheads. Since OTF migration chooses a page for migration whenever it becomes hot, i.e., it crosses a certain threshold, we need to adjust the threshold dynamically to control the rate of migration so it provides optimal benefit. As future work, we plan to explore changing the threshold dynamically during the execution of an application.

6.2 Data Traffic Analysis

In Figure 8, we show the breakdown of the data traffic from (to) memory to (from) processor for a number of workloads we found interesting. For each of the workloads, we analyze the demand and migration traffic generated by our evaluated policies. The y-axis shows the total number of read/write requests (64-byte sized) served by the different memories, viz., HBM and PCM.

For mcf, all of the policies have similar behavior in that they were able to identify and migrate the highly localized part of the memory footprint and hence, we see HBM demand traffic almost doubled as compared to the baseline, providing high IPC improvements. For lbm, OTF_HW_AR migrates almost the same number of pages as OTF_no_AR, but it fails to do it in a timely manner due to extra time it needs to spend in the AR process. Hence, we do not see as many hits in HBM, resulting in performance degradation. As we defined MBQ in Section 4, we found lbm has low MBQ, and hence it is important to migrate the pages early.

As we mentioned, astar falls into least memory localized category (see Section 4), hence there is a large number of pages that only provide moderate number of accesses. With our static threshold of 128, OTF_no_AR and OTF_HW_AR end up migrating ~7× more pages than OTF_OS_AR for astar. OTF_OS_AR also uses the same threshold but due to large time overhead involved in the OS-based AR process there is more pressure on the small remap table and therefore the number of migrations is fewer. As a result, although OTF_no_AR and OTF_HW_AR get more demand hits to HBM than OTF_OS_AR, they fail to get better performance than the latter due to large data migration overhead. For astar, MemPod_no_AR provides slightly better HBM demand hits as compared to OTF_no_AR and OTF_HW_AR, but with fewer migration traffic, because MemPod migrates in smaller granularity. Therefore, MemPod_no_AR provides better performance for astar than others. Similarly, for xalanc, only MemPod_no_AR could provide better performance, because it could migrate in smaller granularity.

HMA_HS_OS_AR uses longer epoch length, hence in most of the cases it did not provide as much performance gain as others; only for mix6, we see that it provides similar or slightly better gain, since with minimal data migration it provided similar HBM demand hits as others.

6.3 Sensitivity Analysis

6.3.1 Threshold Sensitivity. In Figure 9, we show how our OTF_HW_AR policy performs with different hotness threshold values of 16, 32, 64, 128, 256, and 512 for the page migration friendly workloads as compared to a baseline system without any page migration and address reconciliation. For half of the workloads threshold 128 provides better performance than others, and on average it provides the best performance from the set of different static thresholds we have evaluated. Threshold 64 provides the next best performance after threshold 128. The higher threshold 128 provides the improvement primarily because fewer pages are being migrated than with threshold 64. Hence, we choose 128 to be the optimal threshold for our study.

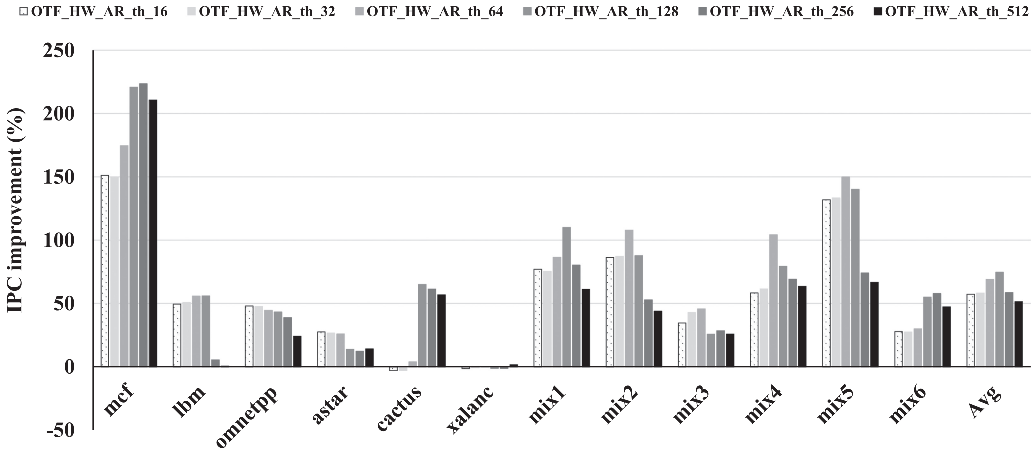


Fig. 9. Threshold sensitivity for OTF_HW_AR policy over no-migration baseline.

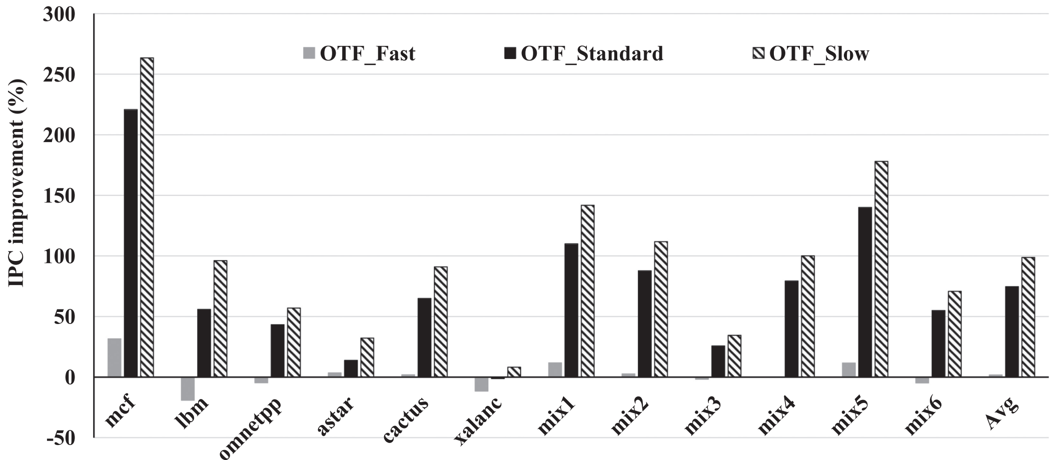


Fig. 10. NVM timing sensitivity for OTF_HW_AR policy over no-migration baseline.

6.3.2 Memory Timing Sensitivity. We also have analyzed the impact of timing sensitivity of OTF_HW_AR when we replace PCM having timing configuration in Table 3 either by faster DDR4-DRAM (DDR4-2400) or by 2×-slower PCM, which is a hypothetical model, to see the impact of having a much slower NVM in the system. As shown in Figure 10, OTF_Fast and OTF_Slow represent IPC performances of flat-address systems when PCM is replaced by faster and slower memories, respectively. As a comparison point, we also show our actual results for OTF_HW_AR as OTF_Standard. We use the same HBM configuration for all of them. We only show the IPC improvements of the page migration friendly workloads, since the unfriendly workloads did not show any improvements as before. Each of the different configurations presented here are compared with the baseline having respective timing configurations.

Use of DDR4 instead of PCM in our system bridges the gap in access latency between "slow" and "fast" memory, therefore, we observe marginal improvement from page migration in case of OTF_fast. However, using an even slower NVM (2× slower than PCM) amplifies the benefit of page migration from slow to fast memory, though it also takes more time to migrate a page. The

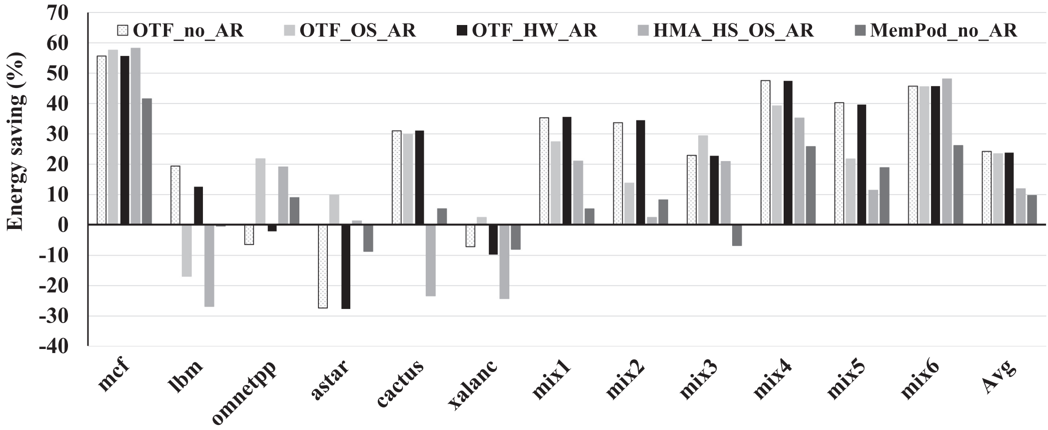


Fig. 11. Energy saving (%) of different page migration and address reconciliation policies over no-migration baseline (negative y-axis represents excess energy spent).

Table 5. Memory Energy Parameters

Memory	Access energy
HBM	3.92 pj/bit
PCM	Read 42 pj/bit Write 140 pj/bit

migration cost is amortized by large time savings due to hits in HBM. OTF_HW_AR provides an average IPC improvement of 2% and 98% for the page migration friendly workloads while using DDR4 and 2× slower PCM, respectively.

6.4 Energy Analysis

We analyzed the dynamic energy consumption of the flat-addressable main memory system for the different page migration policies discussed in this article. Figure 11 shows the energy savings for the page migration friendly workloads by the policies when compared to the baseline, which does not migrate any pages. The y-axis shows the percentage of energy saved or more energy expended. The energy parameters for HBM and PCM are listed in Table 5; for energy number of HBM, we rely on O'Connor et al. [2017]; and for PCM, on Borkar [2013], Qureshi et al. [2009], and Numonyx [2009]. HBM consumes lower dynamic energy than conventional DRAM, whereas PCM consumes slightly higher read and much higher write access energy than conventional DRAM. We calculate the dynamic energy consumed using the demand and migration traffic statistics gathered from the experiments and the parameters listed in Table 5 and also CACTI [Muralimanohar et al. 2009] generated numbers to model SRAM-based remap table, and hot and cold buffer access energy when these structures are accessed by different page migration polices.

On average, for the page migration friendly workloads, OTF_HW_AR provides 24% energy savings over the baseline (with no page migration); whereas HMA_HS_OS_AR and MemPod_no_AR result in 12% and 10% energy savings, respectively. For OTF_HW_AR, the energy spent to access the remap table and hot/cold buffers is only 0.07% of the total dynamic energy spent on average; hence, we emphasize that our proposed hardware is energy-efficient.

While looking at the individual workloads, it is important to understand that higher performance gain does not necessarily result in higher energy-efficiency; rather, energy saving depends

on how well the extra energy spent for migrating pages between memories is amortized by getting more accesses to the pages that are migrated to the energy-efficient memory—in our case, that is HBM. We observe that for less-friendly workloads—astar and xalanc—OTF_HW_AR consumes more energy than the baseline, since for these benchmarks, the extra energy consumed by the migration traffic was not effectively offset by the number of accesses (hits) to the migrated pages after migration due to low memory locality or low MBQ of the workloads (Table 1). For these these two workloads MemPod_no_AR also consumes higher energy than baseline but lower than OTF_HW_AR. As we can see from Figure 8, in both cases, MemPod_no_AR migrates fewer pages than OTF_HW_AR. We also observe from Figure 8 that for astar, HMA_HS_OS_AR migrates far fewer pages than other policies, and hence it provides slight energy saving over baseline. For one moderately friendly workload, omnetpp, we also observe that OTF_HW_AR spends slightly more energy (2%) than baseline primarily due to large amount of migration traffic to PCM; although this large number of page migrations essentially provided 43% of performance improvement. However, OTF_OS_AR migrates ~7× fewer pages than OTF_HW_AR and as a consequence, this technique provides lower performance benefit than OTF_HW_AR but it results in energy savings of 22% over baseline (due to fewer page migrations). Hence, one might choose throttling page migration for marginally migration friendly workloads to tradeoff performance against energy consumption. We plan to investigate such throttling mechanisms in the future.

7 MIGRATION CONTROLLER HARDWARE COST ANALYSIS

As stated in Section 6.1, initially, we assumed that we have accurate information about access counts per page for selecting a candidate (hot page) for migration and a complete LRU information on pages in fast memory for selecting a cold page that can be swapped with the hot page. The research studies that we compared in Section 6.1 assumed similar information for their page migrations. However, maintaining such accurate information is not practical; so, here, we discuss how to minimize the overhead for keeping necessary knowledge on hot and cold pages, while not sacrificing performance benefits of our OTF migration designs.

How to choose hot page with low overhead? As proposed in Jiang et al. [2010], we can maintain 16-bit counters for each PCM page and store the page access counters in HBM while maintaining a small on-chip cache for such counters. For 16 GB PCM, we need 8 MB storage for the counters (assumed page size is 4 KB). It means that 8 MB of HBM space is set aside for the counters, reducing the available HBM memory space by less than 1%. Our experiments show that using counter cache with size 16 KB or 32 KB (and full counters in HBM) has negligible impact on performance (up to 3% degradation), when compared to OTF_HW_AR with accurate access count and LRU information (see Figure 6).

How to choose cold page with low overhead? When it is not practical to maintain an accurate LRU information on every HBM page, we can use two bloom filters of 16,384 bits each (for a total of 4 KB). This bloom filter keeps track of 4,096 most recently used HBM pages, with 15% false negative probability [Brilliant org. 2018]. We insert each accessed HBM frame address in both filters, but initially, we start inserting in the second filter once we have already inserted 2,048 entries in the first one. We reset the filters when they are full (inserted 4,096 distinct entries), hence at any time at least one of the filters will retain information of last accessed 2,048 HBM frames. Altogether, the use of bloom filters along with counter cache has resulted in 6% and 5% performance degradation for cache sizes of 16 KB and 32 KB, respectively, as compared to OTF_HW_AR with accurate access counts and LRU information.

The total storage requirement for the on-the-fly migration controller (MigC) is below 70 KB–16 KB for remap table (for 1,024 entries), 4 KB each for hot and cold buffers, ~2.25 KB for wait queue (32 entries, each 72 bytes), 32 KB for counter cache and 4 KB for bloom filters, and 2.5 KB

for migration queues in memory controllers (we assume 10 memory controllers and 256 bytes required for each migration queue). We feel that this is a small storage overhead and feasible to place on the processor chip.

Our system is scalable to multicore and manycore systems, as we envision the MigC hardware to be local per multicore processor chip or tile; that is, connected to the local memory node containing heterogeneous memory modules. For larger shared memory systems with a number of processors, there will be multiple heterogeneous NUMA (non-uniform memory access) nodes. In such systems, each MigC will always choose a pair of hot and cold pages from heterogeneous modules in the local node. We assume that DMA reads/writes generated by IO devices will be checked against the remap table before accessing memory; therefore, the temporary address redirection introduced by the migration controller will not affect DMA.

8 DISCUSSION AND CONCLUSIONS

In this article, we have presented "on-the-fly" page migration policy that migrates a page from slow NVM to fast 3D-DRAM whenever the page becomes "hot" without waiting for any specific time interval, thus migrating more recent hot pages and increasing the probability of such pages remaining hot after migration. A small on-chip remap table keeps track of locations of the migrated pages, and the remap table periodically evicts entries to make room for new page migration entries. Address reconciliation is then performed for the evicted entries, but rather than depending solely on OS, we have also presented a light-weight hardware-assisted address reconciliation process. Both page migration and address reconciliation processes are orchestrated by a special hardware migration controller. This migration controller allows migration of new pages and address reconciliation of older migrated pages simultaneously, which is unique in this field.

We analyze our on-the-fly migration technique with hardware-assisted address reconciliation and without hardware (that is, using OS), and compare our techniques against two known state-of-the-art page migration techniques. For a set of page migration friendly SPEC CPU2006 workloads, our proposed on-the-fly page migration policy provides 17% IPC improvement, on average, over hardware-managed page migration policy proposed in the MemPod study [Prodromou et al. 2017], where both policies ignore address reconciliation overheads, relying on large unrealistic remap tables. When we integrate our hardware-assisted address reconciliation with on-the-fly page migration policy, the average performance improvement reaches 96% of that by on-the-fly page migration policy without any address reconciliation, indicating hardware-assisted address reconciliation does not add significantly to the overhead. Whereas, if we use conventional OS-based address reconciliation with on-the-fly page migration policy, the average performance improvement is 81% of the performance of on-the-fly page migration without any address reconciliation, indicating higher overheads when relying on OS for address reconciliation. Thus, we show that both our on-the-fly migration and hardware-assisted reconciliation mechanisms are beneficial in heterogeneous memory architectures. Our technique also outperformed the epoch-based page-migration technique that performs address reconciliation using a hardware/software mixed approach as proposed in the HMA-HS study [Meswani et al. 2015], by 29% for the same set of workloads. When compared to a baseline system without any page migration, our technique results in 74% average IPC improvement.

In this article, we have used conventional page size (4 KB) and different static thresholds. In the future, we would like to adjust the threshold dynamically during the execution of an application to adapt better with application behavior. For systems with huge page sizes such as 2 MB, 1 GB, and so on, it might not be beneficial to migrate such a large amount of data among different memories when only part of such pages—"subpages"—are hot. Migrating only hot subpages of huge pages would require more complex data management and bookkeeping for address reconciliation. In

such case, address reconciliation can be avoided using reverse migration of subpages back to their original locations. In the future, we will explore such alternatives for huge pages.

To determine which benchmarks are likely to benefit from any page-migration technique, we present an offline analytical model. In this model, by static analysis of memory access counts to pages, we classify applications as migration friendly and migration unfriendly. Our analytical model correctly classifies 19 out of the 20 workloads evaluated in our study as page migration friendly or not when we match the recommended classifications against results provided by our on-the-fly page migration policy, and also two different hotness-based page migration policies studied in this article. We found only one outlier, lbm, which appears to have some specific pattern of page accesses (for example, access pages in bursts or small reuse distances) that appreciates hotness-based migration principle, which is fast enough to start migration in a timely manner. In the future, we plan to integrate memory access patterns to pages, such as reuse distance of pages, memory-level parallelism, and so on, as input metrics to our analytical model to achieve more complete classifications.

ACKNOWLEDGMENTS

The authors would like to thank Nuwan Jayasena and Mike Ignatowski from AMD Research for their reviews and suggestions.

REFERENCES

Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. 2017. Avoiding TLB shootdowns through self-invalidating TLB entries. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*. IEEE, 273–287.

Tim Bird. 2009. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*. Citeseer, 47–54.

Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2014. Concurrent page migration for mobile systems with OS-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 31.

Shekhar Borkar. 2013. Exascale computing—A fact or affliction. In *Proceedings of the IPDPS (keynote presentation)*.

Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly Media, Inc.

Brilliant.org.2018. Bloom Filter. Retrieved from https://brilliant.org/wiki/bloom-filter/.

Xianzhang Chen, Edwin H.-M. Sha, Weiwen Jiang, Chaoshu Yang, Ting Wu, and Qingfeng Zhuge. 2017. Refinery swap: An efficient swap mechanism for hybrid DRAM–NVM systems. *Fut. Gen. Comput. Syst.* 77 (2017), 52–64.

Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1–12.

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. *ACM SIGARCH Comput. Archit. News* 41, 1 (2013), 381–394.

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 277–288.

US Department Of Energy. 2018. ECP Proxy Application Suite. Retrieved from https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/.

Harish Patil. 2018. PinPlay. Retrieved from https://software.intel.com/en-us/articles/program-recordreplay-toolkit.

M. Heroux and S. Hammond. 2015. MiniFE: Finite element solver. https://portal.nersc.gov/project/CAL/designforward.htm#MiniFE.

HMC Consortium HMCC. 2018. Hybrid Memory Cube Consortium. Retrieved from http://hybridmemorycube.org/.

R. D. Hornung, J. A. Keasler, and M. B. Gokhale. 2011. *Hydrodynamics Challenge Problem*. Technical Report. Lawrence Livermore National Lab (LLNL), Livermore, CA.

Mahzabeen Islam, Krishna M. Kavi, Mitesh Meswani, Soumik Banerjee, and Nuwan Jayasena. 2017. HBM-resident prefetching for heterogeneous memory system. In *Proceedings of the International Conference on Architecture of Computing Systems*. Springer, 124–136.

Joint Electron Devices Engineering Council JEDEC. 2018. 3D ICs. Retrieved from http://www.jedec.org/category/technology-focus-area/3d-ics-0.

Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 25–37.

Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache. *ACM SIGARCH Comput. Archit. News* 41, 3 (2013), 404–415.

Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*. IEEE, 1–12.

Krishna Kavi, Stefano Pianelli, Giandomenico Pisano, Giuseppe Regina, and Mike Ignatowski. 2015. Memory organizations for 3D-DRAMs and PCMs in processor memory hierarchy. *J. Syst. Archit.* 61, 10 (2015), 539–552.

Joonyoung Kim and Younsu Kim. 2014. HBM: Memory solution for bandwidth-hungry processors. In *Proceedings of the Symposium on High Performance Chips (HC'14)*.

Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. DOI: 10.1109/LCA.2015.2414456

Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. Chameleon: A dynamically reconfigurable heterogeneous memory system. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 533–545.

Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 126–136.

Vipul Kumar Mishra and D. A. Mehta. 2013. Performance enhancement of NUMA multiprocessor systems with on-demand memory migration. In *Proceedings of the 3rd IEEE International Advance Computing Conference (IACC'13)*. IEEE, 40–43.

Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C. Germann. 2013. Co-design for molecular dynamics: An exascale proxy application. https://www.lanl.gov/orgs/adtsc/publications/science_highlights_2013/docs/Pg88_89.pdf.

David Mosberger and Stephane Eranian. 2001. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall PTR.

Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* 27 (2009), 28.

Prashant J. Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K. Qureshi. 2015. Reducing read latency of phase change memory via early read and Turbo Read. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 309–319.

Numonyx. 2009. Phase Change Memory. Retrieved from http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. IEEE, 41–54.

Osnat Levi. 2018. Pin—A Dynamic Binary Instrumentation Tool. Retrieved from https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M. Tullsen. 2017. MemPod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 433–444.

Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. Phase change memory: From devices to systems. *Synth. Lect. Comput. Archit.* 6, 4 (2011), 1–134.

Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 235–246.

Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Comput. Archit. News*, Vol. 37. ACM, 24–33.

Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*. ACM, 85–95.

Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 2010. UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent hardware management of stacked dram as part of memory. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 13–24.

Fengguang Song, Shirley Moore, and Jack Dongarra. 2009. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–10.

spec. 2015. SPEC CPU 2006. Retrieved from https://www.spec.org/cpu2006/.

Iulia Ştirb. 2018. NUMA-BTLP: A static algorithm for thread classification. In *Proceedings of the 5th International Conference on Control, Decision and Information Technologies (CoDIT'18)*. IEEE, 882–887.

ChunYi Su, David Roberts, Edgar A. León, Kirk W. Cameron, Bronis R. de Supinski, Gabriel H. Loh, and Dimitrios S. Nikolopoulos. 2015. HpMC: An energy-aware management system of multi-level memory architectures. In *Proceedings of the International Symposium on Memory Systems*. ACM, 167–178.

Yujuan Tan, Baiping Wang, Zhichao Yan, Qiuwei Deng, Xianzhang Chen, and Duo Liu. 2019. UIMigrate: Adaptive data migration for hybrid non-volatile memory systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 860–865.

John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench—The development and verification of a performance abstraction for Monte Carlo reactor analysis. In *Proceedings of the Conference on the Role of Reactor Physics toward a Sustainable Future (PHYSOR'14)*.

Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 340–349.

Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. 2017. Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 1–14. DOI: https://doi.org/10.1145/3123939.3124555

Received March 2019; revised August 2019; accepted September 2019