Dynamically Adapting Page Migration Policies Based on Applications Memory Access Behaviors

ABSTRACT

There have been numerous studies on heterogeneous memory systems comprised of faster DRAM (such as 3D stacked HBM or HMC) and slower non-volatile memories (e.g., PCM, STTRAM). However, most of these studies focused on static policies for managing placement and migration of pages among the different memory devices. These policies are based on the average behavior across a range of applications. Results shows that these techniques do not always result in higher performance when compared to systems that do not migrate data across the devices: some applications show performance gains, but other applications show performance losses. In this paper we propose and evaluate techniques that dynamically adapt to the behavior of applications and either reduce (or increase) migrations or even halt migrations. Our adaptive techniques show performance gains for both migration friendly (on average of 71% over no migrations) and unfriendly workloads (by an average of 3%). Another contribution involves reverse migration of pages so that pages that are no longer heavily used may be sent back to their original place in slower memories, eliminating the need for address reconciliation (updating TLBs and PTEs with new physical addresses). We show performance gains of 59% with reverse migrations. Such reverse migrations can be useful when dealing with very large pages, whereby subpages of a page may be migrated instead of whole pages.

Key Words: Heterogeneous Memory Systems, On-the-fly page migration, Adaptive migration policies, Reverse migration, Address reconciliation.

1. INTRODUCTION

The need for larger, faster and lower power memory systems has been clear as more and more data-centric applications are becoming common place. Emerging data-centric applications need memory systems with very large capacities (100s of GBs to TBs), high bandwidth and energy efficiency [3], [16].

Architecture community has reacted to these needs with new memory technologies including 3D stacked DRAMs and very dense non-volatile memories. Different organizations for combining diverse memory technologies into a system architecture have been investigated, including hierarchical organizations (i.e., using 3D DRAM as LLC) or flat-address memories where 3D DRAM (for example, HBM[9], HMC [8]), DDR and non-volatile memories (NVM) devices (for example STT-RAM [20], PCM [27]) form a continuum of memory address space. To effectively reduce average memory access times, heavily accessed pages (hot pages) are migrated from slower memories (NVM) to faster memories (3D DRAM); cold pages are moved from faster memories to slower memories to make room for the hot pages. Pages can be migrated (or swapped) at regular intervals (epoch based) or individually (on-the-fly). The migration of pages between the memory systems incur execution and energy overheads. In addition to the cost of actual data movement between memory devices, OS tables (TLBs, page tables) must also be updated since physical addresses in such memory systems is based on the physical location of a page and a migration changes physical addresses (we call this process "address reconciliation", AR). There have been numerous designs that evaluated the efficiencies of different page migration techniques. One thing is clear from these studies: no single approach leads to consistent performance improvement for all applications. Some applications may actually see a performance degradation. It may be possible to perform off-line analysis of memory behaviors of applications and categorize them as migration friendly and migration unfriendly and use page migrations for only the migration friendly applications. However, off-line analyses are often coarse-grained and may not capture evolving behaviors of applications: applications may have both migration friendly and unfriendly phases. Our page migration techniques dynamically increases or reduces migrations and even turns-off migrations to adapt to changing behaviors of applications. The key contributions of our work are:

- On-the-fly page migration with hardware support for address reconciliation. Since we migrate a page as soon as it becomes "hot" (not wait for the end of an epoch and migrate several hot pages together) it is important to ensure that the migration does not halt user program execution, or incur costly OS intervention for address reconciliation. We use a small ReMap table to track physical locations of recently migrated pages. Periodically, older entries in ReMap table are deleted, after updating PTEs and TLBs, making room for new page migrations.

We use a specialized Migration Controller (MigC) hardware to handle page migration and address reconciliation.

- Adaptive migration polices. Previous page migration techniques relied on fixed hotness thresholds. In contrast, we control page migration policies based on applications memory access behaviors. Our technique increases or reduces the hotness thresholds to reduce or increase the number of pages migrated based on the observed benefits of page migrations.
- We also explored the benefit of reverse migrating pages to their original locations. The reverse migration eliminates the need for address reconciliation.

The rest of the paper is organized as follows. Section 2 includes the motivation for adaptive page migration techniques. The next section includes a description of our Migration Controller (MigC) as well as the migration and address reconciliation processes. Section 4 contains our experimental setup and the benchmarks used for evaluation. Section 5 includes an analysis of the results of our experiments. Section 6 includes a discussion of research that is closely related to ours and section 7 summarizes the conclusions of this study and further research that can be explored.

2. MOTIVATION

A number of heterogeneous memory investigations, such as [21], [35] and [26], show that not all applications benefit from page migrations since page migration incurs performance overheads due to extra data movement, as well as overheads for address reconciliation. It is possible to develop off-line analyses to categorize applications as "migration friendly" and "migration unfriendly", so that page migration is enabled only for migration friendly workloads. We developed one such off-line classification by analyzing memory accesses to main memory pages (when the accesses miss the cache hierarchy). We then create a histogram that shows how many pages received a certain number of accesses. Applications with an exponential shaped histogram indicates that very few pages receive most accesses and those applications benefit if these pages are either placed in the faster (HBM) memory at the start of execution, or migrated to HBM on demand. This is the case with mcf where just 3% of all pages contribute to 97% of memory accesses.[1] Thus mcf gains significant performance from page migration. On the other hand, applications exhibiting uniform shaped histograms indicate that most or all pages receive about the same number of accesses, implying that too many pages may be migrated and thus these applications do not benefit form page migration. This behavior is exhibited by milc: 65% of pages contribute to 82% of all accesses and this application does not benefit from page migration.

Friendliness	Benchmark
Very Friendly	mcf, mix1, mix2, mix4, mix5
Moderately Friendly	lbm, omnetpp, astar cactus, bfs, mix3, mix6
Least friendly or Unfriendly	milc, gems, zeusmp xalanc, braves, miniFE lulesh, xsbench, CoMD

Table 1: Migration Friendliness of Applications

Table 1 shows a possible classification of applications based such an analysis. With offline analysis, one could potentially determine an application either friendly or unfriendly. But for applications exhibiting both migration friendly and unfriendly phases, relying on off-line analysis limits the ability to benefit from the migration friendly phases of an application. A properly designed adaptive migration technique can turn-on or turn-off migration, or adjust the number and frequency of page migrations based on the evolving behavior of an application. Our results show that our adaptive techniques achieve this goal and result in better performance than static techniques.

A second motivation for this study is our desire to minimize OS overheads for address reconciliation (updating TLBs and page table entries when pages are relocated during migration). Large remap tables can be used to eliminate address reconciliation; however we use small remap tables and rely on hardware for address reconciliation to minimize OS intervention. We also explore the idea of reverse migrating previously migrated pages, when they are no longer heavily accessed, to eliminate address reconciliation, making the page migration transparent to OS.

3. ON-THE-FLY PAGE MIGRATION

Unlike approaches that rely on epochs (e.g., 10ms intervals) to track page access counts to determine which hot pages to migrate, we migrate a page as soon it receives a certain number of accesses (hotness threshold). We show that our "on-the-fly" (OTF) migration performs better than epoch-based page migration techniques since we migrate recent hot pages, instead of pages that accrued accesses over a long interval. Since page migration can take place at any time, it is important to ensure that a migration does not halt user program execution [2]. Moreover, OS based address reconciliation on each page migration is prohibitive for on-the-fly migration. We devise a specialized hardware (Migration Controller (MigC)) placed on the processor chip, which performs actions necessary for our on-the-fly page migration, including monitoring the number of pages

[1]We omit details of the analysis and our approach for classifying applications as migration friendly or not, since off-line analysis is not a key contribution of this paper. If this contribution is accepted, citation to reports that contain our off-line analyses will be provided.

[2]Epoch based approaches stop program execution and migrate several hot pages at the end of an epoch. OS manages the migration as well as updating TLBs and PTEs with new physical addresses.

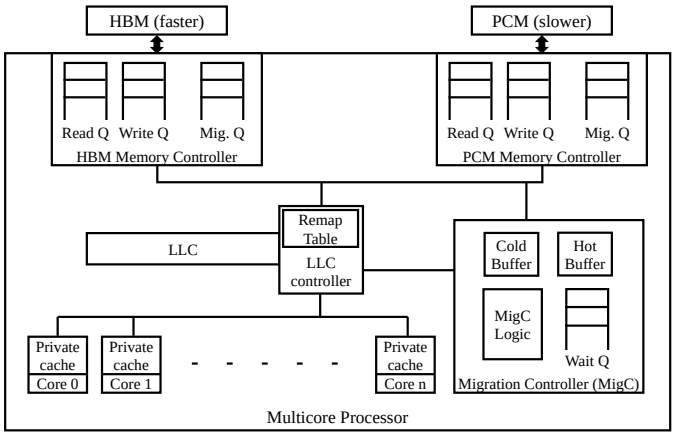

Figure 1: High-level system architecture

migrated and MBQ to adapt migrations to the behavior of applications. MigC performs limited communication with OS via reserved registers or memory locations [3]. 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. There is a Wait queue in MigC, which holds read/write requests from LLC for the currently migrating pages; these requests will be serviced by the hot/cold buffers. The memory controllers (MCs) are equipped with a separate Migration queues (Mig.Q) to service requests from MigC for the migrating pages. There is a small ReMap table which holds new physical page 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. The size of the ReMap table is kept small (e.g., 1024 entries) so that it can be placed on-chip. 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 3.2.

3.1 User Transparent Page Migration

We migrate a hot page from PCM into a free frame of HBM if available (one-way migration), or select an LRU HBM (cold) page and swap the hot and cold pages (two-way migration). Section 3.3 describes how hot and cold pages are selected. MigC performs one migration at a time. For explanation, we label the steps involved in a page migration process as M1, M2, etc.

M1) At time T1, MigC finds that a PCM (say page A with physical address, PA 8192) meets the hotness threshold for migration. MigC then finds a cold page from HBM (say page B with physical address, PA 0) to swap with the hot page. MigC inserts entries for pages A and B in the ReMap table with their current OS visible physical addresses (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 these pages are under migration and a Pair flag is set to 1 to indicate a two-way migration involving a hot and cold pair (this flag will be set to zero for one-way migration). The Pair flag will be checked during address reconciliation (AR) to update page table entries for both (or one) pages involved in the migration.

M2) MigC waits for any pending read requests to the pages involved in the migration that were already issued by time T1 to complete. Then, say at time T2, MigC starts reading hot and cold pages into their respective buffers. Any new requests (after time T1) for these pages from LLC 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.

M3) At time T3, MigC starts writing contents of buffers to their respective new page frames and Mig flag is set to 2. A bit vector kept in the ReMap table indicates which lines have already completed migration and any demand requests to migrated cache lines will be sent to appropriate memories.

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.

3.2 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. Removing entries from the ReMap table requires updates to physical addresses (i.e., address reconciliation) to reflect the new location of the page consistently throughout the system. We reconcile entries from the ReMap table pair-wise if the Pair flag with the entries is 1.

The following actions must be performed to ensure correct address reconciliation (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 (old) physical address (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

[3]Our hardware migration controller (MigC) can be configured as a memory mapped device and OS can communicate with MigC to obtain information about ReMap table entries for specific pages. Due to space limitations, we omit the details on the interactions between OS and MigC needed when OS wants to reserve pages that were migrated or for MigC to check if the pages considered for migration are not reserved by OS. We also omit detailed hardware complexity analysis in the contribution. If this contribution is accepted, citation to reports that detail the communication between MigC and OS will be provided.

invalidated.

3.2.1 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, (iii) *flush_tlb_page()* [23]. 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 4μs to flush cache lines of a page using CLFLUSH x86 instruction on a processor running at 2.26GHz. To update PTE, Linux acquires page table lock and changes PTE and also executes *flush_tlb_page()* to invalidate all TLBs (TLB shootdown) with old VA to PA translation. OS releases the lock upon completing these actions. The TLB shootdown is costly because it uses IPI (interprocess interrupt) to invalidate TLB entries in every core that contains an entry with old PA. The delay grows non-linearly with the number of cores [29, 36, 2]. As reported in [21], 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" 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 (instead of CLFLUSH instruction). More details are given below. 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 acknowledgements from MigC. For TLB-shootdown we rely on a shared TLB directory that contains all the private TLB entries along with process identifiers (i.e., address space identifier) and core residency information. MigC initiates TLB shootdown by sending the associated virtual addresses (VAs) to the shared TLB directory. The shared TLB directory then maps these VAs to necessary entries and requests cores to invalidate these TLB entries somewhat similar to that used in [36]. We envision that actual invalidation at each 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 invalidations is assumed to be a round-trip off-chip memory access latency [36].

3.2.2 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, our ReMap table contains only the original physical address (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. Linux keeps descriptors for every allocated physical page frame that maintains bookkeeping information on the number of PTEs referring to this page frame and pointers to such PTEs [5]. By using existing Linux reverse mapping function, we can obtain the list of PAs of PTEs which hold mappings to this specific page frame number and associated VAs with ASIDs [5]. We account for all the delays involved for these OS functions needed to implement our address reconciliation and page migrations, using previously reported numbers and actual experimental data on real system (see Table 3).

3.2.3 Hardware-Centric Address Reconciliation

Continuing with the example hot page A (PA:8192) and cold page B (PA: 0) we now present the steps involved in hardware based address reconciliation. The steps are labeled as R1, R2, etc.

R1) At time T5 (T5 is much greater than T4), MigC requests OS to send all VAs and ASIDs for the OS assigned PAs 8192 and 0. OS uses reverse mapping function [5], to find appropriate PA to VA mappings and return them to MigC. MigC will also instruct OS to update the associated PTEs and other necessary structures which keep information about page frames, with the new PAs and not to allow any further access to the associated PTEs. OS will lock the PTEs of A and B and then update them with new PAs 0 and 8192, respectively and block 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.

R2) At time T6, MigC will instruct the shared TLB directory to invalidate the VAs in question. The TLB directory will then initiate actions to invalidate all involved TLB entries by sending invalidation requests to cores. Each core will perform TLB invalidations and notify the shared TLB directory [36]. At this point of time, no new access requests to the pages being reconciled are allowed since the TLB entries are blocked, and will have to wait until AR completes.

R3) Next, at time T7, MigC will send "write-invalidate" coherence requests for all the cache lines of these pages so that these lines are invalidated (and written back to memory if dirty). The ReMap table still contains entries for pages A and B, 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 . Once OS completes unblocking OS will notify the MigC. This completes the address reconciliation process.

Since address reconciliation process freezes the VA to PA translation process for the pages under reconciliation, user processes cannot execute any new access requests to these pages. Therefore, when a page is hot we we will not 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.

3.3 Selecting Hot and Cold Pages

Instead of maintaining access counters for each page, we can maintain counters for PCM pages and store the counters in HBM; a small (16KB-32KB) on-chip cache for some counters can be used (similar to [15]). For a 16GB PCM we need 8MB storage for the counters (assumed page size is 4KB). It means that only 8MB of our 1GB HBM space is set aside for the counters. Our experiments show that using a 16KB or 32KB cache for some counters has negligible impact on page migration performance (up to 3% degradation), when compared to using access counts to all pages in on-chip tables.

Likewise, instead of maintaining accurate LRU information for HBM pages, one can use two bloom filters of 16,384 bits each (for a total of 4KB). The bloom filter keeps track of 4096 most recently used HBM pages, with 15% false negative probability [6]. We insert each accessed HBM frame address in both filters, but initially we start inserting in the second filter once 2048 entries have been inserted into the first one. We reset the filters when they are full, hence at any time at least one of the filters will retain information on the 2048 last accessed HBM pages. Altogether, the use of bloom filters along with 16KB or 32KB PCM page access counter cache has resulted in 6% and 5% page migration performance degradation respectively, when compared to using full PCM page access counts and accurate LRU information.

4. EXPERIMENTAL SETUP

4.1 Simulation Infrastructure

We model a 16-core system with a flat-address heterogeneous memory consisting of 1GB of HBM and 16GB of PCM using Ramulator [17]. 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 3.2GHz. The cores have private L1-D caches (32KB, 4-way, 2-cycles) and shared L2 (16MB, 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 [17] and for those of PCM on [24]. We modified Ramulator to support a flat-address heterogeneous memory system. A basic address mapping function is added to Ramulator to support this model; it allocates pages to frames of different memories in a round-robin fashion (viz., 4 pages to faster memory, then 4 pages 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 application span both memory devices and thus necessitating page migration considerations in our experiments.We also made sure that the memory footprints of our benchmarks are at least twice as large as the HBM capacity, requiring the use of both HBM and PCM.* We incorporate our MigC unit in Ramulator with all necessary details to perform functions as described in previous sections (Section 3). MigC also operates at 3.2GHz. We assume conventional 4KB pages. The ReMap table is implemented as a 1024 entry fully-associative table. We conservatively assumed an access latency of 10 CPU cycles (in some experiments we tested with larger ReMap tables and adjusted access times appropriately). We included all timing overheads (listed in Table 3 assuming 3.2GHz clock rate) for performing different HW/OS tasks described in the previous Section 3.1. We also implemented two previously known page migration studies, HMA-HS [21] and Mempod [26] for comparison purposes. We compare the performance of all these systems with a baseline that does not migrate pages.

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

Table 2: Baseline configuration

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 [36])
Page walk	150 cycles
OS reverse mapping	4480 cycles (measured using Ftrace [4] on a real machine running Linux)

Table 3: Timing parameters at 3.2GHz clock

4.2 Workloads

We use sixteen multi-programmed SPEC CPU2006 [34] workloads, four multi-threaded representative HPC benchmarks from the US Department of Energy (DOE) pro-

	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		
libq	2x		1x	3x		4x
mcf	3x	2x		1x	5x	
milc	2x		2x	1x		2x
omntpp	1x					3x
soplex	2x	3x		3x	5x	
sphinx	1x		2x			3x

Table 4: SPEC multi-programmed mix workloads

vided ECP Proxy Applications [10] as well as the BFS from Graph500 suite [14]. We selected SPEC benchmarks with large memory footprints, at least twice the capacity of HBM. SPEC benchmarks allow us to compare our work with other studies. We profile benchmarks using PinPlay kit [11] 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 multithreaded HPC proxy benchmarks by the US Department of Energy (DOE) that we used are- XSBench [1], LULESH [13], CoMD [22] and miniFE [12]. We ran each HPC benchmark in a 16-thread setup and collected 500M instruction traces for each of the threads using Pin tools [25]. 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 2GB to 11GB, ensuring that the workloads fit in physical memory and do not require access to secondary storage. They are large enough to cause migration but not unrealistically small.

5. RESULTS AND ANALYSES

In this section we present the experimental results using fixed thresholds for page migration as well as our adaptive migration and reverse migration techniques.

5.1 Analysis of Fixed Hotness Threshold Experiments

In the first set of experiments, we evaluated the performance of our on-the-fly page migrations (OTF) using fixed hotness thresholds. We experimented OTF without any Address Reconciliation (OTF_no_AR), assuming a sufficiently large on-chip ReMap table to track all pages migrated during a program executions. This is unrealistic but provides a data point for comparison. Then we evaluated OTF with OS-based AR (OTF_OS_AR) and OTF with our hardware-based AR (OTF_HW_AR). *This allows us to directly compare the benefits of using hardware instead of OS for address reconciliation.* We compared our OTF schemes with an epoch-based page migration study [21], which uses OS for address reconciliation; we refer to this as HMA_HS_OS_AR. We also compare our techniques with MemPod [26], where no explicit AR is necessary since they assume large ReMap tables; we refer to this as MemPod_no_AR. For all the OTF schemes presented in this section, we use a hotness threshold of 128. We experimented with different thresholds (e.g., 32, 64 and 128) for 4KB pages and settled on 128 since this threshold provided the best trade-off between the number of pages migrated and the cost of migrations. For both our OTF_OS_AR and OTF_HW_AR schemes, we use a 1024 entry ReMap table and start the address reconciliation process whenever the table is 50% occupied. We stop migration if the ReMap table does not contain free entries, and wait for address reconciliation to free up space. In our implementations of HMA_HS_OS_AR, we used a static threshold of 128 as it showed better performance than the threshold of 32 used in [21] and an epoch length of 100ms. In our implementation of MemPod_no_AR, given that our memory system is setup with 8 HBM and 2 PCM memory controllers (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 50μs [26]. Note MemPod migrates 2KB pages.

Figure 2 shows the results using static hotness thresholds for all our workloads for different page migration and AR policies. A positive (negative) y-axis value shows IPC improvement (degradation) as a percentage when compared to a baseline without any page migrations. We separated migration friendly and unfriendly workloads and used different scales for y-axis.

Overall, for page migration friendly workloads, our on-the-fly migration with hardware-based AR technique (OTF_HW_AR) results in 74% IPC improvement on average over the baseline system. *It also show 24% IPC improvement on average over on-the-fly page migration with OS-based address reconciliation (OTF_OS_AR).*

OTF_HW_AR) shows higher improvements over other page migration techniques - HMA_HS_OS_AR [21] (by 29%)and MemPod_no_AR [26] (by 13%). We included results for migration unfriendly workloads to show that all page migration techniques degrade performance for these workloads, not just our on-the-fly technique, and thus confirming our discussion regarding classifying applications as migration friendly and unfriendly in Section 2.

Delving more deeply, for 2/3rd of the page migration friendly workloads, OTF_no_AR performs better than MemPod_no_AR. For omnetpp, astar, xalancmbk, and mix3, MemPod_no_AR performs better mainly be-

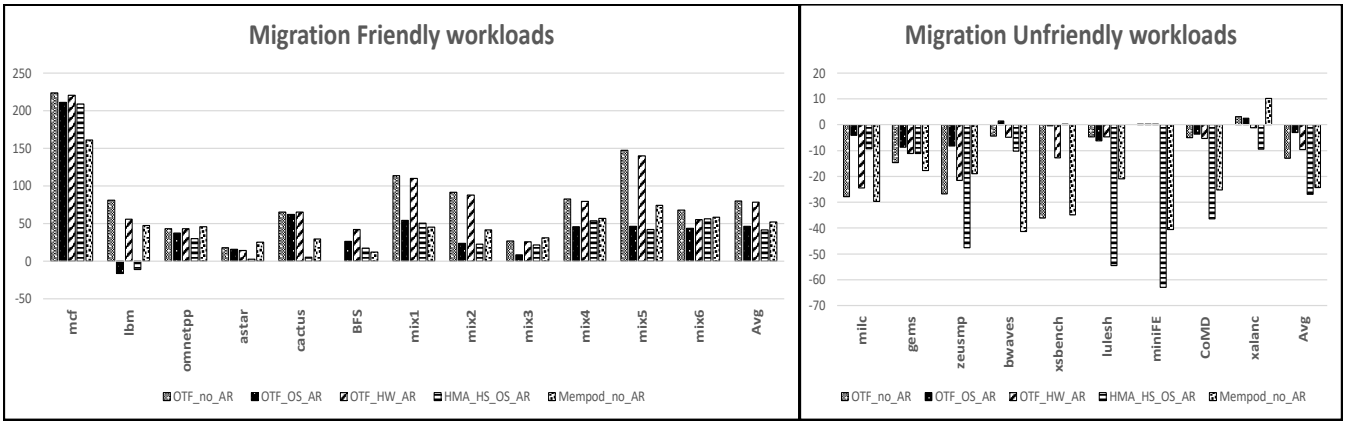

Figure 2: IPC improvement (%) of different page migration and address reconciliation policies over no-migration baseline with static hotness thresholds (negative y-axis shows degradation)

cause Mempod migrates at 2KByte granularity than at 4KB granularity used in OTF_no_AR. With smaller migration granularity (2KB), it is possible to only migrate only hot subpages of a larger (4KB) page). When 2K pages are used, our OTF_no_AR performs better than MemPod_no_AR for all workloads. It is important to note that, for more than half of the migration friendly workloads, even after accounting for all address reconciliation overheads (as discussed in Section 3), hardware-based AR, OTF_HW_AR performs better than MemPod_no_AR which perform no address reconciliations. Next, we compare HMA_HS_OS_AR that used OS based address reconciliations at each epoch with our on-the-fly approach. As shown in Figure 2, OTF_HW_AR performs better than HMA_HS_OS_AR for all the page migration friendly workloads except mix6. In this case, OTF_HW_AR migrated more pages than HMA_HS_OS_AR; the migration benefit of some of the pages is not high. In later sections we describe adaptive thresholds to monitor the migration benefit quotient (MBQ) to control the number of pages migrated.

As expected, hardware based AR (OTF_HW_AR) performs better than OS based AR (OTF_OS_AR). The only exception are astar and xalancmbk; the hardware based AR with smaller overheads is migrating more pages than the OS based AR methods (since the migrations are paused during AR), however, the additional migrations are not beneficial since these applications are classified as moderately friendly or unfriendly. In epoch based methods (e.g., [21]), since several pages are migrated at the end of an epoch, address reconciliation of all migrated pages can be completed together using OS, amortizing the cost of address reconciliation. Hardware based address reconciliation may only have minimal benefit for epoch based approaches. MemPod assumes very large Remap tables and thus requires no address reconciliation.

5.2 Adaptive Hotness Thresholds

In our next set of experiments, we dynamically change hotness thresholds to control the number of pages migrated in a window (we monitor pages migrated for every 4 million cycles). [4] We double the hotness threshold (but cap at a maximum of 256) if too many pages are migrated (more than 240 [5]); likewise we halve the threshold (but not below 64) when too few pages are migrated (less than 160). The MigC will monitor the number of pages migrated and sets new threshold values. MigC still performs address reconciliation when the ReMap table is more than half full. We compare the IPC using dynamic thresholds with the results obtained using static threshold of 128, OTF_HW_AR (previously shown in Figure 2). Figure 3 shows the IPC improvements using our adaptive control of hotness thresholds over the baseline with no page migration. The bars labeled OTF_HW_AR_adaptive_migcount refer to the results for our on-the-fly migrations using hardware address reconciliation and adapting hotness thresholds based on the number of pages migrated. Our adaptive technique results in 71% performance gains on average over the baseline for migration friendly workloads (the on-the-fly migration with static threshold shows 78% gains over the baseline) but shows on average 2% performance gains over the baseline even for unfriendly workloads (static threshold OTF shows a performance loss of 7%). The adaptive threshold generally reduces the number of pages migrated for migration unfriendly workloads (for example milc, gems, bwaves, lulesh, CoMD, xalancmbk) reducing the cost of migrations. On the other hand, applications that are friendly (e.g., mcf, lbm, BFS and some mixed workloads) may see some decrease in performance gains when compared to static threshold based migrations. This is likely because, when the MigC notices that very few pages are migrated in a window, the threshold is reduced to 64 to increase the migrations.

[4] All values used in our experiments are based on 4KB pages and the hardware organizations used (see Tables 2 and 3). These values may be different for other configurations.

[5] We observed that for migration friendly benchmarks, on average 240 pages were migrated in 4 million cycles with static thresholds.

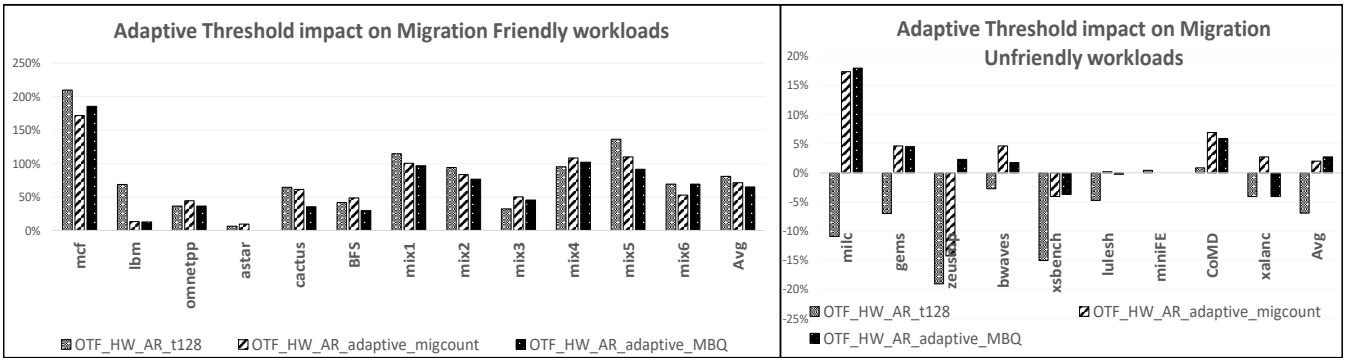

Figure 3: IPC improvement (%) using Adaptive migration policies over no-migration baseline (negative y-axis shows degradation)

However, these additional migrations do not contribute to performance gains. For example, as described in Section 2, for mcf, only 3% of pages receive large number of accesses, and migrating other pages with fewer access does not significantly contribute to performance gains, but adds to migration and address reconciliation costs. Another approach to dynamically adapting to an application is to rely on migration benefit quotient. Here we define MBQ as the average number of accesses to pages that were recently migrated to HBM. MigC counts all the accesses received by such pages in a window (every 4 million cycles). We decrease (halve) the hotness threshold when the MBQ is high (greater than 130 [6]), causing more pages to migrate. We increase (double) the threshold if the MBQ is low (less than 50), to reduce the number of pages migrated. We halt migrations if the MBQ is low for several consecutive windows (more than 25 windows). Figure 3 includes the results for MBQ-based adaptive threshold control: results labeled OTF_HW_AR_adaptive_MBQ, refer to our on-the-fly migrations using hardware for address reconciliation and adapting thresholds based on the average access counts to recently migrated pages (i.e., MBQ). On average, our MBQ based adaptive migration shows a performance gain of 65% over the baseline (compared 78% gain using static threshold) for migration friendly workloads, and a 3% performance gain on average (compared a performance loss of 7% with static threshold) for unfriendly workloads. The adaptive technique results in either some performance gains, or at least prevent performance losses for migration unfriendly benchmarks (for example, milc, gems, zeusmp, bwaves, CoMD). However, the adaptive MBQ technique results in smaller performance gains for some migration friendly benchmarks when compared to the results using static thresholds (e.g., mcf, lbm, cactus and some mixed workloads) for the same reasons outlined in the previous paragraph.

[6]Our analysis of average number of accesses with static thresholds indicate that a MBQ less than 90 did not yield performance gains. This number depends on the overheads due to page migration and address reconciliations.

To understand these behaviors better we analyzed migration patterns and accesses to migrated pages, see Figure 4. For each migration technique experimented in this study we show the average number of pages migrated in a window (4 million cycles) and the average number accesses to pages migrated to HBM in a window (or MBQ). Please note that the MBQ per window in Figure 4 is shown in $\log_2$ scale. Adaptive techniques migrate more pages for migration friendly workloads than static threshold technique, but this results in lower average MBQ. The exception is lbm. The static threshold technique migrated very large number of pages, while the adaptive techniques aggressively reduced the number of pages migrated and this in turn reduced overall performance gains. For migration unfriendly workloads, adaptive techniques migrated fewer pages and this in turn resulted in higher average MBQ than static thresholds. The exceptions are gems and bwaves. The number of migrations for gems is greatly reduced in adaptive migration techniques which improved the overall MBQ. For bwaves, the static threshold resulted in slightly better MBQ than adaptive techniques, but it migrated more than twice as many pages: the migration overheads defeat the MBQ advantage. These results directly translate into performance gains shown in Figure 3.

In summary, these results indicate that an increase in MBQ improves performance and that MBQ may be increased either by increasing the number of useful migrations or reducing migrations that are not useful. Our adaptive techniques achieve these goals without having to perform off-line analyses. Our adaptive techniques are very simple to implement. We either count the number of pages migrated in a given time window, or measure the number of accesses to pages recently migrated to HBM. It may be possible to investigate more intelligent adaptive techniques, but they will likely be complex to implement.

5.3 Reverse Migration of Pages

Next, we explored reverse migration of pages to eliminate address reconciliation. As the ReMap table fills up instead of reconciling addresses of pages and removing

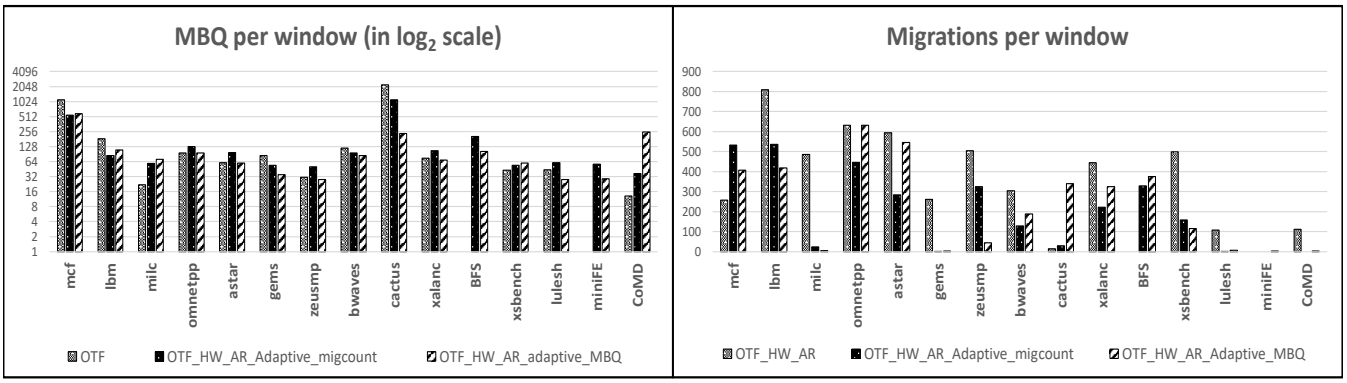

Figure 4: Traffic data (Left, MBQ per window. Right, Migration count per window)

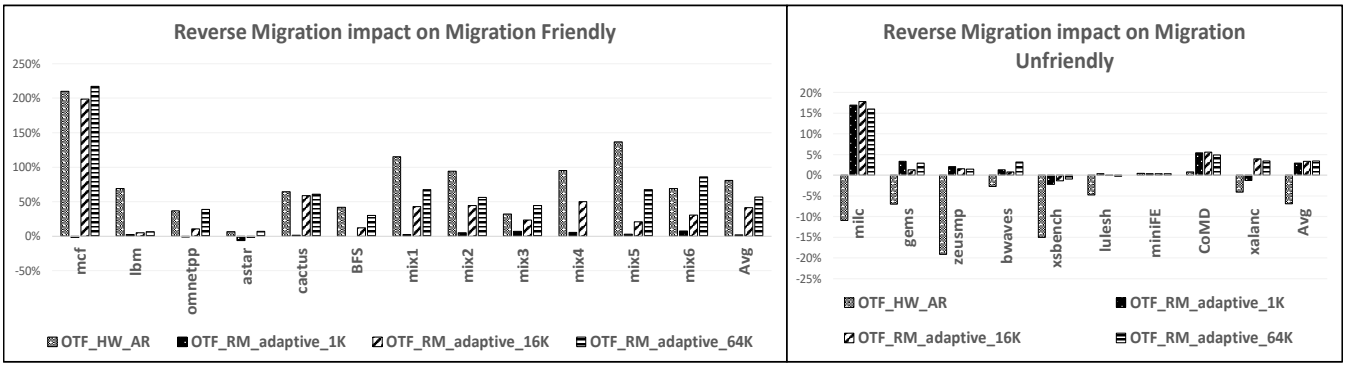

Figure 5: IPC improvement (%) of Reverse migration over no-migration baseline with different remap table sizes (negative y-axis shows degradation)

ReMap entries, we migrate pages back to their original locations and remove the corresponding ReMap entries. We select pages based on MBQ: pages with least number of accesses in a window are candidates for reverse migration, making sure that migrated pages remain in HBM for a minimum duration (in our case one million cycles) to avoid reverse migrating pages too early. Figure 5 shows the results. We still use adaptive thresholds relying on MBQ as described in Section 5.2. For comparison purposes, Figure 5 includes data for migration based on static threshold of 128 (previously shown in Figure 2: these results are labeled as OTF_HW_AR_t128 (with 1024 ReMap entries). For reverse mapping, we vary the ReMap table sizes between 1K and 64K entries. The results are labeled with ReMap table sizes. For example, OTF_RM_adaptive_1K refers to our on-the-fly migrations with reverse mapping, using adaptive MBQ based thresholds and a ReMap table with 1024 entries. For migration friendly workloads, on average the reverse migration technique shows performance gains of 2%, 41%, 57% with 1K, 16K, 64K ReMap tables respectively over the baseline (compared to 74% gains using static threshold). A smaller ReMap table causes frequent migrations and reverse migrations leading excessive data movement. Larger ReMap table results in performance gains for most applications when compared to the baseline with no page migrations. For some migration friendly benchmarks (for example lbm), reverse migration may cause heavily accessed pages to be migrated and reverse migrated several times. Address reconciliation eliminates such repeated migrations since heavily accessed pages are likely to be permanently moved to HBM (and physical addresses reconciled). For most workloads, larger ReMap tables provide sufficient space for heavily accessed pages to remain in ReMap table for longer periods of time, requiring fewer reverse migrations. Even for migration unfriendly workloads, reverse migration shows performance gains of about 3% for all ReMap table sizes (compared to 7% performance loss using static threshold). Our adaptive techniques either limit or stop page migrations for these migration unfriendly applications. Thus the size of the ReMap table has very little impact on the performance of migration unfriendly workloads. We feel that reverse migrations can be a viable option to HMA systems, particularly when the address reconciliation is costly. Larger ReMap tables are justified since reverse mapping eliminates the complexity and overheads of address reconciliations. A 64K ReMap table would require 1.3Mbytes This table can be placed in HBM while caching a small portion inside the MigC. This is similar to the studies reported in [32, 7, 26]; in those studies the ReMap tables are

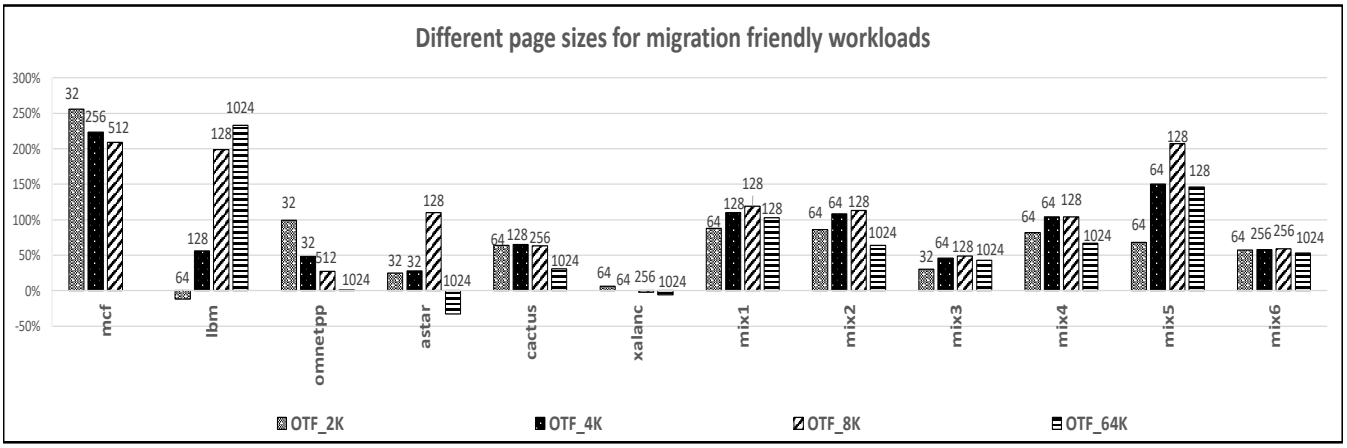

Figure 6: IPC improvement (%) of OTF over no-migration baseline with different page sizes showing the best hotness threshold on top of bar (negative y-axis shows degradation)

much larger than what we are proposing.

Although not actually implemented in this study, reverse migration may be useful in systems with very large pages. Larger pages can make page migration in HMA infeasible or expensive. The hotness thresholds will necessarily be very large limiting the number of pages migrated. We studied page migrations using static thresholds for different pages sizes and the results are shown in Figure 6[7]. The figure shows the threshold(s) that result in the best performance for different page sizes (each bar is labeled with page size). For most applications, the thresholds increase with page size (and likely to be very large for 1MB, 4MB or 1GB pages), leading to very few page migrations and insignificant performance benefits. As an aside, from Figure 6 we can also observe that some benchmarks show better performance with 8KB pages (over traditional 4KB pages) and one workload (lbm) shows better performance even with 64KB pages. Some applications, for example xalancmbk, performs better with 1KB or 2KB pages (or migrating subpages of a larger pages).

So, instead of migrating large pages, one may consider migrating only hot subpages of large pages. However, such subpage migrations require tracking the physical location of subpages. ReMap tables can be extended for such purposes: each entry in the ReMap table can contain a bit map to indicate if a subpage is migrated or not. But address reconciliation becomes very cumbersome; we need to migrate the remaining subpages before updating PTE and TLB entries. Instead, the use of reverse migration can provide an alternative to address reconciliation, making subpage migration a potentially viable method. Migration (and reverse migration) can take place at smaller subpage granularity (1KB, 2KB) as these result in better performance for some benchmarks.

[7]The page size for baseline is set to correspond the page sizes used in migrations. Even if the page size for the baseline is fixed at 4KB, our conclusions remain. We did not include these results due to space limitations.

6. RELATED WORK

There have been many studies on page migration techniques for flat-address heterogeneous memory systems (HMA), They propose different approaches to solve the general challenges associated with page migration, viz., selecting candidate pages to migrate, determining migration frequency and managing migration metadata. Here we will review only the works that are most closely related to our research. Meswani et al. [21] 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. There is no restriction on to where (i.e., which HBM page frame) a hot page can be migrated. Since OS-based address update incurs large overheads, the authors choose a longer epoch to reduce frequent OS interventions. However, it has been observed that page migration at shorter intervals is more beneficial than waiting for longer epoch times [32, 26], since, migrating hot pages sooner results in more beneficial accesses to the migrated pages.

Address reconciliation can be avoided using very large remap table that contains the new locations of migrated pages. The size and management of this remap table presents a new challenge. A number of different approaches have been proposed to keep this table in memory while using a small on-chip cache for recently accessed entries [32, 7, 26, 18]. Sim et al. used a Transparent Hardware based Management (THM) of flat-address memory [32]. In THM, restricts where a migrated page can be placed to reduce the remap table: a set of slow memory pages compete for a single fast memory page. This can reduce potential benefits since only one of the slow memory pages from a given set can be migrated to fast memory even if all of them are heavily accessed. Chou et al. proposed a somewhat similar idea of intra-set migration named CAMEO [7],

where the migration is done at finer granularity (cache line size) and the migration candidate is chosen on each slow memory access. While Chameleon [19], depending on application requirement, dynamically reconfigure the parts of stacked DRAM as flat-address memory or cache. Prodomou et al. proposed MemPod [26], which provides more flexibility on page relocation than [32, 7]. 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. A more recent work, PageSeer [18] proposes extensions to memory controllers that initiate page swaps between slow and fast memories based on TLB misses.To use the 3D-DRAM capacity efficiently, Ryoo et al., proposed a sub-block-based page migration and co-location of sub-blocks of two different pages in interleaved fashion [31]. However, this scheme only allows migration of sub-blocks within same congruence group and requires large SRAM tables to keep track of sub-blocks with bit vectors. In [30] the granularity of the data migrated depends on the contiguity of accesses in the virtual address space. In our work, migration granularity is fixed; however, we have included an evaluation of how the page size impacts the performance of page migrations.

In our proposal we migrate a page immediately when it receives sufficient number of memory accesses, unlike any epoch-based schemes described above. We allow full flexibility in page relocation like HMA-HS [21] and keep a small on chip ReMap table for address redirection. Similar approach has been proposed by Ramos et al. [28], which also performs on-the-fly type of migration with periodical reconciliation of remapping table entries and OS memory mappings. However, in [28] 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 concurrently. Furthermore, our migration candidate choice scheme is simpler than multi-queue scheme used by [28]. We also study dynamic adjustments to page migrations: we change hotness thresholds to reduce or increase the number of pages migrated or pause migrations when insignificant benefits are observed. We also explore reverse migration of pages to eliminate address reconciliations.

In [37], the authors propose a technique (called Banshee) 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-buffer fill up, the user programs are stopped and OS updates the TLB (and PTE) entries to reflect the new location of pages– again somewhat similar to our address reconciliation. However, unlike our hardware orchestrated approach, they rely on OS for address reconciliation. Moreover, 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.

Conventional NUMA (Non-Uniform Memory Access) systems with Multi-socket CPU and homogeneous memory emulate data migration between local and remote nodes. Accesses to data within local memory are faster compared to remote memory locations. The main difference between NUMA and HMA data migration is that, NUMA migration occurs when cores from different sockets need to work on the same data. This issue of on-demand migration is mitigated in these [33], [38] works by running multiple threads which share data on the same node, while in HMA, pages are migrated in a single-node system.

7. CONCLUSIONS

In this paper we presented a new Hybrid Memory Architecture (HMA) that relies on on-the-fly (OTF) migration of heavily accessed pages from slower memories to faster memories. We use a specialized hardware to assist in the processes needed for address reconciliation and page migrations. We divide our workloads into migration friendly and migration unfriendly. Our on-the-fly migration using fixed hotness thresholds outperformed epoch-based page migration technique that performs address reconciliation using a hardware/software mixed approach as proposed in HMA-HS study [21], by 29% for migration friendly workloads. When compared to a baseline system without any page migration, our technique results in 78% average IPC improvement for the same set of workloads.

We then tested two different adaptive migration techniques to adjust the page migrations to application behaviors. In one case, we increased (or decreased) the hotness threshold when too many (or too few) pages are migrated in a given interval (4m cycles) so that fewer (or more) pages will satisfy the hotness threshold in the future. This resulted in 71% performance gains on average over the baseline for migration friendly workloads, *but more importantly, our technique shows on average 2% performance gains over the baseline even for unfriendly workloads.* In as second adaptive method, we used the average number of accesses to pages that were recently migrated to HBM (we call this migration benefit quotient, MBQ) to reduce (or increase) hotness threshold if MBQ is high (or low). This resulted in a performance gain of 65% over the baseline for migration friendly workloads and 3% performance gain for migration unfriendly workloads.

We also experimented with reverse migration of pages, eliminating address reconciliation. ReMap entries are freed for new page migrations by reverse migrating pages with low MBQ back to their original locations. We feel that reverse migration can be used when dealing with very large pages: migrate subpages (either in 1K,2K or 4K granularity) and eliminate complex bookkeeping and address reconciliations by using reverse migrations.

8. REFERENCES

[1] "Proxy-Apps for Neutronics," https://cesar.mcs.anl.gov/content/software/neutronics.

[2] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 2017, pp. 273–287.

[3] C. Bendelac and P. Kokkalis, "SAP HANA Memory Usage Explained," https://www.sap.com/documents/2016/08/205c8299-867c-0010-82c7-eda71af511fa.html, 2017, [Online; accessed January-20-2019].

[4] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 47–54.

[5] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[6] Brilliant org., "Bloom Filter," https://brilliant.org/wiki/bloom-filter/, 2018.

[7] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 1–12.

[8] H. Consortium, "Hybrid Memory Cube Consortium," http://hybridmemorycube.org/, 2018, [Online; accessed July-27-2018].

[9] J. E. D. E. Council, "3D ICs," http://www.jedec.org/category/technology-focus-area/3d-ics-0, 2018, [Online; accessed July-27-2018].

[10] DOE, "US Department of Energy ECP Proxy Application Suite," https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/, 2018.

[11] Harish Patil, "PinPlay," https://software.intel.com/en-us/articles/program-recordreplay-toolkit, 2018.

[12] M. Heroux and S. Hammond, "Minife: finite element solver," 2015.

[13] R. Hornung, J. Keasler, and M. Gokhale, "Hydrodynamics challenge problem," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2011.

[14] Jewillco, "Graph500-v2-spec," https://github.com/graph500/graph500/tree/v2-spec, 2015.

[15] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[16] K. Keeton, "Memory-driven computing." in *FAST*, 2017.

[17] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," 2015.

[18] A. Kokolis, D. Skarlatos, and J. Torrellas, "PageSeer: Using page walks to trigger page swapps in hybrid memory systems," in *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2019.

[19] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "Chameleon: A dynamically reconfigurable heterogeneous memory system," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 533–545.

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

[21] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 126–136.

[22] J. Mohd-Yusof, S. Swaminarayan, and T. C. Germann, "Co-design for molecular dynamics: An exascale proxy application, 2013," 2013.

[23] D. Mosberger and S. Eranian, *IA-64 Linux kernel: design and implementation*. Prentice Hall PTR, 2001.

[24] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 309–319.

[25] Osnat Levi (Intel), "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2018.

[26] 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 *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 433–444.

[27] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase Change Memory: From devices to Systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[28] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 85–95.

[29] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "Unified instruction/translation/data (unitd) coherence: One protocol to rule them all," in *Proceedings-International Symposium on High-Performance Computer Architecture*, 2010.

[30] J. H. Ryoo, L. K. John, and A. Basu, "A case for granularity aware page migration," in *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*, 2018, pp. 352–362. [Online]. Available: https://doi.org/10.1145/3205289.3208064

[31] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "Silc-fm: Subblocked interleaved cache-like flat memory organization," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 349–360.

[32] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 13–24.

[33] F. Song, S. Moore, and J. Dongarra, "Analytical modeling and optimization for affinity based thread scheduling on multicore systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*, Aug 2009, pp. 1–10.

[34] spec, "SPEC CPU 2006," https://www.spec.org/cpu2006/, 2015.

[35] C. Su, D. Roberts, E. A. León, K. W. Cameron, B. R. de Supinski, G. H. Loh, and D. S. Nikolopoulos, "Hpmc: An energy-aware management system of multi-level memory architectures," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 167–178.

[36] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 340–349.

[37] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via

software/hardware cooperation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124555

[38] I. Åđtirb, "Numa-btlp: A static algorithm for thread classification," in *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, April 2018, pp. 882–887.