Dynamically Adapting Page Migration Policies Based on Applications Memory Access Behaviors

SHASHANK ADAVALLY, University of North Texas
MAHZABEEN ISLAM, University of North Texas
KRISHNA KAVI, University of North Texas

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 (such as PCM, STT-RAM). However, most of these studies focused on static policies for managing data placement and migration among the different memory devices. These policies are based on the average behavior across a range of applications. Results show 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. It is possible to utilize off-line analyses to identify which applications benefit from page migration (migration friendly) and use page migration only with those applications. However, we observed that several applications exhibit both migration friendly and migration unfriendly behaviors during different phases of execution supporting a need for adaptive page migration techniques. We introduce 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 81% over no migrations) and unfriendly workloads (by an average of 3%): it should be remembered that previous migration techniques resulted in performance losses for unfriendly workloads.

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

Additional Key Words and Phrases: Heterogeneous memory systems, flat address memory, dynamic page migration, reverse migration

ACM Reference Format:
Shashank Adavally, Mahzabeen Islam, and Krishna Kavi. 2020. Dynamically Adapting Page Migration Policies Based on Applications Memory Access Behaviors. 1, 1 (December 2020), 24 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

High performance applications as well as emerging data-centric applications need memory systems with very large capacities (100s of GBs to TBs), high bandwidth and energy efficiency [3], [15]. For example, SAP HANA in-memory database system requires 256GB to multiple terabytes (TBs) per host [3]; Spark in-memory analytics provides higher performance when run with 12TB memory [15]. These applications need memory systems with very large capacities (100s of GBs to TBs), high bandwidth and energy efficiency [3], [15]. Also, the value of these large amounts of gathered data depends on how fast the data can be analyzed to make decisions [15].

Authors' addresses: Shashank Adavally, ShashankAdavally@my.unt.com, University of North Texas; Mahzabeen Islam, University of North Texas, MahzabeenIslam@my.unt.edu; Krishna Kavi, University of North Texas, Krishna.Kavi@unt.edu.

Manuscript submitted to ACM

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 these diverse memory technologies into a system architecture have been investigated, including hierarchical organizations (i.e., using 3D DRAM as Last Level Cache, LLC) or flat-address memories where 3D DRAM (for example, HBM[8], HMC [7]), DDR and non-volatile memories (NVM) devices (for example STT-RAM [19], Phase Change Memories or PCM [27]) form a single memory address space. In such flat-address systems, 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 are based on the physical location of pages and a migration changes physical addresses: we call this process of changing physical addresses and updating system tables "address reconciliation" or 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 due to page migrations [26], [20]. It may be possible to perform off-line analysis of memory access behaviors of applications and categorize them as migration friendly (applications that show performance gains from page migration – for these applications, performance gains outweigh migration overheads), and migration unfriendly (applications that do not show performance gains – overheads outweigh performance gains from migration) 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. It is necessary to design adaptive migration techniques to dynamically increases or reduces migrations and even turns-off migrations to adapt to changing behaviors of applications. We propose such adaptive page migrations techniques in this paper. The key contributions of our work are:

- Adaptive migration polices. Previous page migration techniques relied on fixed hotness thresholds: a page is migrated from slow memories to faster memories when the number of times that page was accessed exceeds the hotness threshold. 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 either the number of pages migrated over a window of observation or based on the observed benefits of page migrations (were pages accessed after the migration to faster memories).
- Address reconciliation overheads can defeat the benefits of page migration. To eliminate address reconciliation, we explore the benefit of reverse migrating pages to their original locations, particularly when the migrated pages are no longer heavily accessed. Reverse migration makes page migration invisible to OS. However, reverse migrations can result in excessive data movement between slow and fast memories. In this paper we evaluate the effectiveness of reverse migration technique.

Epoch-based approaches migrate "hot" pages at the end of an epoch. In such systems, some hot pages may have exhausted their usefulness by the time they are migrated. Our previous research [13] and the MemPod study [26] have observed that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period (i.e., an epoch). We migrate pages as soon as they become hot (we call this "on-the-fly" or OTF migration). Epoch-based approaches rely on OS for address reconciliation, which can be excessive.

In our previous study [13] we used a special hardware Migration Controller (MigC) that includes a small remap table (or ReMap table in our system) to track physical locations of recently migrated pages to aid in redirecting accesses to correct page locations. Periodically, older entries in ReMap table are deleted, after MigC updates PTEs and TLBs, making room for new page migrations (that is, after address reconciliation). We based our adaptive migration techniques and reverse migration techniques on our previous design. We extended MigC to monitor applications' memory access behaviors to dynamically adapt page migrations.

The rest of the paper is organized as follows. Section 2 includes the motivation for adaptive page migration techniques. We also include a description of our Migration Controller (MigC) as well as the migration and address reconciliation processes. We include this information (previously reported in [13]) to make this contribution self-contained. Section 3 describes our adaptive migration techniques as well as the reverse migration technique. Section 4 contains our experimental setup and the benchmarks used for evaluation. Section 5 includes an analysis of the results from our experiments. We include both performance and energy results when using our adaptive migration policies. 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 BACKGROUND AND MOTIVATION

A number of heterogeneous memory investigations, such as [20], [35] and [26], and our previous study [13] show that not all applications benefit from page migrations since page migrations incur 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" (applications that show performance gains) and "migration unfriendly" (applications that show performance losses), so that page migration is enabled only for migration friendly workloads. In our previous work [13] we developed one such off-line classification by analyzing memory accesses to main memory pages (when the accesses miss the cache hierarchy). We then created a histogram that shows how many pages received a certain number of accesses. We discovered that an exponential shaped histogram indicates that very few pages receive most accesses and those applications benefit by either placing those few pages in the faster (HBM) memory at the start of execution, or migrated to HBM on demand. This is the case with mcf (one of the benchmarks) where just 3% of all pages cause 97% of memory accesses[1]. Thus mcf gains significant performance from most page migration policies and it is classified as a migration friendly application. 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 if a fixed hotness threshold is used for migrating pages, and the migration overheads outweigh performance gains. This behavior is exhibited by milc (another one of our benchmarks): 65% of pages contribute to 82% of all accesses and this application does not benefit from page migration and will be classified as migration unfriendly workload.

We also tracked the usefulness of pages that were recently migrated to faster memory. Migration of pages to faster memories result in performance gains if those pages continue to be heavily used, because these accesses will be satisfied by faster memories. We defined Migration Benefit Quotient (MBQ) to measure the average usefulness of recently migrated pages. Relying on the histograms and MBQ, we classified applications as very (migration) friendly, moderately friendly and migration unfriendly. Table 1 shows a possible classification of our benchmarks based on such an analysis.

[1]We omit details of the analysis and our approach for classifying applications as migration friendly or not friendly, since off-line analysis is not a key contribution of this paper and were previously published in [13].

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

2.1 On-The-Fly Page Migration

In this section we will include a brief description of our previous work [13] to provide sufficient details needed to understand the contribution of our paper. This section also provides the motivation for our adaptive migration techniques.

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 call this On-The-Fly (OTF) migration technique. OTF migration performs better than epoch-based page migration techniques since we migrate recent hot pages [13]. This is inline with the observations made by [26] that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period. Since in OTF migration, a 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. To mitigate these issues, we devised a special hardware called MigC, placed on the processor chip, which performs actions necessary for our OTF page migration.

Figure 1 shows a high-level system architecture of MigC. There are 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 Last Level Cache (LLC) for the currently migrating pages; these requests will be serviced from 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 2.3 and in [13].

2.2 User Transparent Page Migration

We migrate a hot page from slower PCM into a free frame of HBM if available (one-way migration), or select an HBM page that has not been accessed recently (LRU Cold page) and swap the hot and cold pages (two-way migration). Consider that MigC finds that a PCM page (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 (namely 8192

[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 page table entries with new physical addresses.

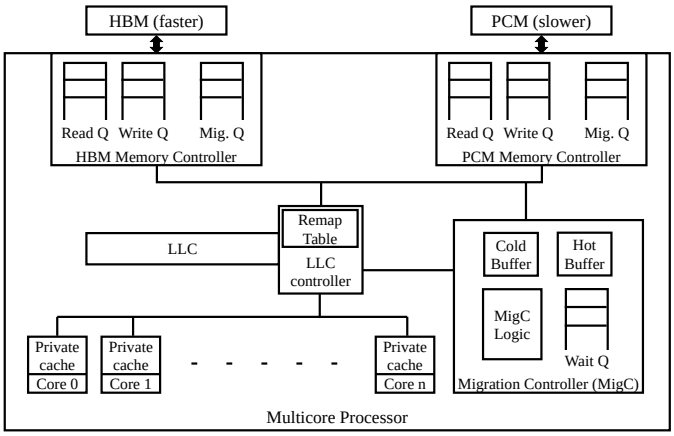

Fig. 1. High-level system architecture

and 0) and future PA (after migration, namely 0 and 8192). ReMap table is always looked up using OS visible (original) PA. Mig flag is set to 1 when these pages are being migrated 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.

MigC waits for any pending read requests to the pages involved in the migration that were already issued to complete. Then, MigC starts reading hot and cold pages into their respective buffers (inside MigC). Any new requests (after the migration is initiated) for these pages from LLC will be held in MigC resident Wait Queue and will be served from these buffers. After completely reading the migrating page contents into the buffers, MigC starts writing contents of buffers to their respective new page frames. When migrations are completed, the Mig flag for these pages will be reset (to indicate completion of migration). All future requests for these pages will be directed to proper new locations based on the ReMap table information.

2.3 Address Reconciliation

Address reconciliation can be eliminated (and make page migration transparent to OS) by using very large ReMap tables, sufficient to track all migrated pages during the lifetime of an application. However, this is not practical for emerging systems with very large memories (several hundred giga bytes to tera bytes). We use a very 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, and making the new physical addresses visible to OS. We reconcile entries from the ReMap table pair-wise if the Pair flag is 1, thus updating the physical addresses of the pages swapped during the migration. When the Pair flag is 0 then we perform AR only for that entry. The following actions must be performed to ensure correct address reconciliation. We use the same example hot and cold page pair, A (PA=8192) and B (PA=0), respectively. First, all cache lines from these pages, which are currently residing in the cache hierarchies and tagged with OS visible (old) physical address, 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 (known as TLB shootdown).

2.3.1 Address Reconciliation: OS vs. Hardware. Linux [3] 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()* [22].

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 number of cores [2, 29, 36]. As reported in [20], 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). 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].

2.3.2 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, 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). More details on how our MigC performs address reconciliations can be found in in [13].

[3] We use Linux based systems in all our experiments which makes it easier to compare our results with those reported in the literature.

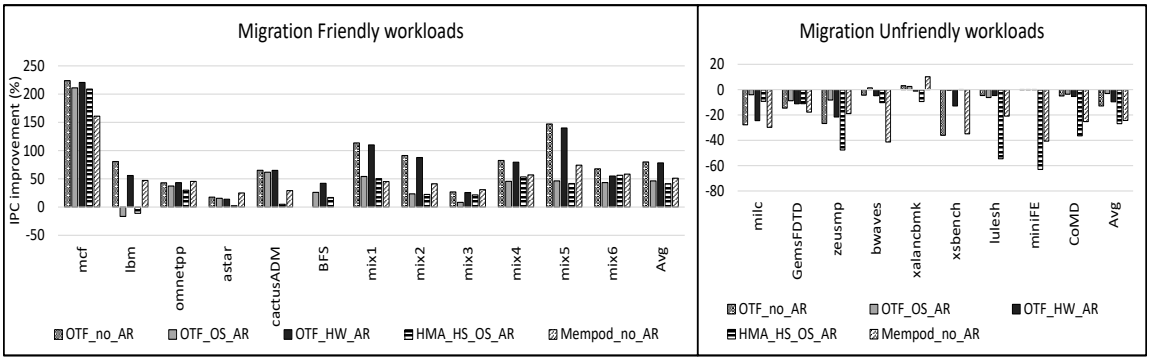

Fig. 2. IPC improvement (%) of different page migration and address reconciliation policies over no-migration baseline with static hotness thresholds (negative y-axis shows degradation)

2.4 Analysis of Fixed Hotness Threshold Experiments

As a motivation for the adaptive migration techniques presented in this paper, we reproduced some results from [13]. We include results using On-the-Fly (OTF) migrations without any Address Reconciliation (labeled as 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. We also included OTF with OS-based AR (labeled as OTF_OS_AR) and OTF with our hardware-based AR (labeled as 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 [20], which uses a combination of hardware and OS for address reconciliation (we refer to this as HMA_HS_OS_AR), and with MemPod [26], with no AR as it assumes large ReMap tables (we refer to this as MemPod_no_AR). For all the OTF schemes presented in this section, we used a fixed 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 migration experiments, one that performs address reconciliations using OS (OTF_OS_AR) and the other which uses our MigC hardware (OTF_HW_AR), 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. More details of the experimental parameters can be found in [13]

Figure 2 shows the results using static hotness thresholds for all our workloads [4] for different page migration and address reconciliation techniques. A positive y-axis value shows performance improvement in terms of Instructions Per Cycle or IPC as a percentage when compared to a baseline without any page migrations. Likewise, a negative y-axis value indicates a performance (or IPC) loss as a percentage compared to the baseline. We separated migration friendly and unfriendly workloads and used different scales for y-axis to make the graphs clearer.

[4]Experimental setup and workloads are described in Section 4.

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 shows 24% IPC improvement on average over on-the-fly page migration with OS-based address reconciliation (OTF_OS_AR).*

Our hardware based migration, OTF_HW_AR, shows higher improvements over other page migration techniques as well - HMA_HS_OS_AR [20] (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 justifying our discussion regarding classifying applications as migration friendly and unfriendly in Section 2. 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 2.1), hardware-based AR, (OTF_HW_AR) performs better than MemPod_no_AR even when MemPod performs 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, our hardware based address reconciliation (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, within our on-the-fly techniques, hardware based AR (OTF_HW_AR) performs better than OS based AR (OTF_OS_AR). The only exception are astar and xalancbmk; 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. Epoch-based approaches migrate "hot" pages at the end of an epoch, and some hot pages may have exhausted their usefulness by the time they are migrated. This is one of the reasons for our on-the-fly technique outperforming HMA_HS_OS_AR. But, in epoch-based methods (e.g., [20]), 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. Detailed discussions on the performance results can be found in [13].

2.5 Motivation

The performance results (shown in figure 2) for different page migration techniques that use static values for "hotness" thresholds supports our classification of applications as migration friendly and unfriendly shown in Table 1 in Section 2. The very migration friendly applications do show significant performance gains while unfriendly applications show performance losses. With off-line analysis such as ours (as described in Section 2 and in [13]), one could potentially classify an application as either friendly or unfriendly and use page migrations only for migration friendly applications. But some applications exhibit both migration friendly and unfriendly behaviors during different phases of execution. For such applications, relying on off-line analysis limits the ability to benefit from page migrations during migration friendly phases of an application.

Consider the behavior of one benchmark *xalancbmk* shown in figure **??**. The left side of the figure shows the variation of MBQ and the number of pages migrated over the course of the benchmark execution, using static "hotness" threshold (in this case 128) in determining when a page is migrated. Both the number of pages migrated and MBQ varies during the execution of the application. Although a large number of pages were migrated during the windows between 70 and 180, the MBQ did not show a significant increase. *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.* The right side of the figure **??** shows the result of using one of our adaptive migration techniques (as described later

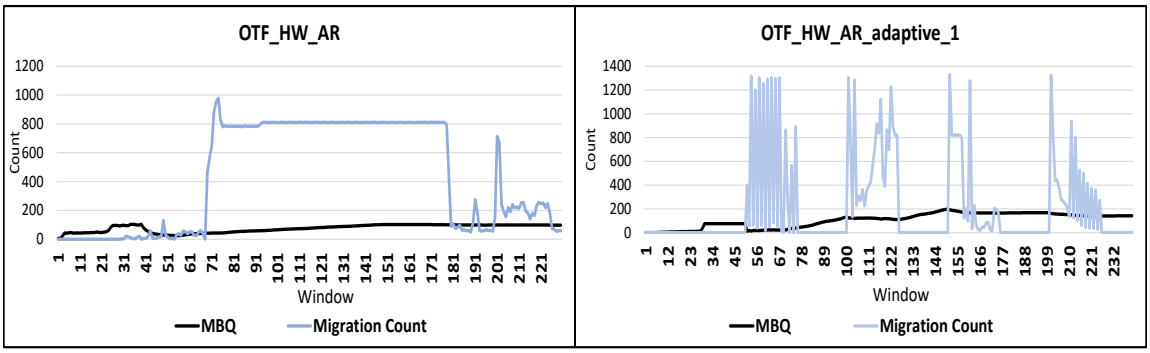

Fig. 3. MBQ and Migration count variation of xalancbmk workload

in section 3.2). The adaptive technique controls the number of pages migrated based on MBQ. The figure shows an overall increase in MBQ (which corresponds in the performance gains).

A second motivation for this study is our desire to minimize overheads due to address reconciliation (updating TLBs and page table entries when pages are relocated during migration). Some prior research (e.g., [26]) used very large remap tables [5] to eliminate address reconciliation. However, the remap tables for emerging systems with 100's of GB to terabytes of memory can become impractical, or require additional techniques for the management of large remap tables (for example, caching a small number of remap entries while keeping rest in DRAM). We use small remap tables and rely on hardware for address reconciliation to minimize OS intervention. As an alternative to using address reconciliation, we also explore the idea of reverse migrating previously migrated pages, particularly when they are no longer heavily accessed, making the page migration completely transparent to OS.

3 ADAPTIVE MIGRATION

The results shown in figure 2 in section 2.4 clearly indicate that page migration techniques that use static or fixed "hotness" threshold for deciding when a page is considered for migration can lead to performance loss for some applications. In this section we describe our adaptive migration techniques that adjust the number of pages migrated and even turn off migration by observing the benefits of page migration, eliminating the need for off-line analysis for classifying applications as migration friendly and unfriendly. We track the number of pages migrated in a window *twindow* and determine if too many or too few pages are migrated in the window. We also measure the Migration Benefit Quotient (MBQ) which is the average number of accesses to pages recently migrated to faster memories. Using these metrics we propose two adaptive techniques described in Algorithm 1 and Algorithm 2. Values shown for different variables in these algorithms depend on the page size (4KB in our experiments) and system parameters such as memory latencies and overheads due address reconciliation. However, they will be constant for a specific system configuration.

3.1 Adaptive Migration Based on Number of Pages Migrated

Algorithm 1 presents an overview of our first adaptive migration technique. We monitor the page migration behaviors over a window, or a threshold_window (i.e. twindow), and dynamically increase or reduce hotness thresholds to be

[5]Note that remap tables keep track of the new locations of migrated pages.

Algorithm 1 Adaptive Migration_count technique

```
1:  function ADAPTIVE-MIGRATION(migclk, MBQ, threshold)
2:      migwindow = 100000000;
3:      twindow = 4000000
4:      min_MBQ = 50
5:      max_MBQ = 70
6:      min_mig_count = 160
7:      max_mig_count = 240
8:      min_threshold = 64
9:      max_threshold = 256
10:     if migclk mod twindow == 0 then
11:         if mig_count ≥ max_mig_count && threshold < max_threshold then
12:             threshold *= 2;
13:         else if mig_count ≤ min_min_count && threshold > min_threshold then
14:             threshold /= 2;
15:     if migclk mod migwindow == 0 then
16:         if MBQ ≤ min_MBQ then
17:             pausemigration = true;
18:         else if pausemigration && MBQ ≥ max_MBQ then
19:             pausemigration = false;
20:
```

used by our on-the-fly migration techniques. The change is based on how many pages have been migrated during this window. If the count is high (too many pages have been migrated), we double the hotness threshold to reduce future migrations; likewise, if too few pages have been migrated in a *twindow*, we halve the hotness threshold to increase future migrations. In our experiments we used 4 million cycles as our *twindow* [6]. We also limit the hotness threshold variations between 64 and 256. We increase threshold if more than 240 pages have been migrated in a window and reduce the threshold if fewer than 160 pages have been migration in a window. These numbers are based on our observations from our experiments of systems we simulated. These numbers will likely be different for other systems and can be determined from experimental evaluations. We also pause migrations or resume migrations using a Migration Benefit Quotient (MBQ). We define MBQ as the average number of accesses to pages that were *recently migrated to HBM*. If the MBQ is less than a threshold (min_MBQ), then migrations are halted; migrations are resumed if the MBQ is greater than another threshold (max_MBQ). The decisions regarding pausing and restarting migrations are made only after observing MBQ values over several *twindows*; we refer to this as migration_window (*migwindow*). Our experiments indicated that an average MBQ of less than 90 did not result in performance gains. We use this value to pause migrations. This number (indicating the number of accesses to recently migrated pages in a *twindow*) depends on the overheads to page migrations.

3.2 Adaptive Migration Based on MBQ

Migration of a page from a slow memory to a faster memory is valuable only if the page receives sufficiently large number of accesses after migration to offset the cost of migration (and address reconciliation). Figure **??** in section 2.5 illustrated that the usefulness of migrated pages (or MBQ) plays a critical role in the overall performance gains

[6]We selected this value to minimize overheads when changing the hotness thresholds. Smaller window results in more rapid adaptation, which in turn causes some overheads in changing the configuration of MigC structures that identify pages ready for migration. Larger windows may be too slow to adapt.

Algorithm 2 Adaptive MBQ technique

```
1:  function ADAPTIVE-MIGRATION(migclk, MBQ)
2:      migwindow = 100000000;
3:      twindow = 4000000
4:      min_MBQ = 50
5:      max_MBQ = 70
6:      min_threshold = 64
7:      max_threshold = 256
8:      if migclk mod twindow == 0 then
9:          if MBQ ≤ max_MBQ && threshold < max_threshold then
10:             threshold *= 2;
11:         else if MBQ ≥ min_MBQ && threshold > min_threshold then
12:             threshold /= 2;
13:     if migclk mod migwindow == 0 then
14:         if MBQ ≤ min_MBQ then
15:             pausemigration = true;
16:         else if pausemigration && MBQ ≥ max_MBQ then
17:             pausemigration = false;
18:
```

achieved. Thus, another approach to dynamically adapt to an application's behavior is to rely on MBQ as shown in Algorithm 2. Our hardware MigC counts all the accesses to recently migrated pages in a window (*twindow* or 4 million cycles) and calculates the average MBQ. We decrease (halve) the hotness threshold when the MBQ is high (greater than 130), 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). We remind the reader that these specific numbers are based on our experiments and the systems on which we conducted our experiments and the numbers may be different for different systems.

3.3 Reverse Migration

In most page migration techniques, either epoch-based or on-the-fly migrations, the OS is eventually notified of the migration to modify page table entries and TLBs to reflect new physical addresses of the migrated pages (since physical addresses are based on their location). We referred to this process as address reconciliation. As described in section 2.3, and reported in [13], address reconciliation can be very expensive, particularly if performed in software by the Operating System. The address reconciliation can be completely eliminated by using very large remap tables, as done in [26]. However, the remap tables for emerging systems with 100's of GB to terabytes of memory can become impractical, or require additional techniques for the management of large remap tables. Instead of using very large remap table, we propose to use small remap tables but "reverse migrate" pages back to their original locations, thus making room for more recent hot pages. It should be noted that reverse migration may involve "pairwise" migration if originally a page from faster member was moved to slower memory to make room for a hot page from slower memory.

The reverse migration process is shown in figure 4. The top left quadrant of the figure shows the identification of hot pages and the top right quadrant shows the original migration, where a PCM page with a PA of 8192 is swapped with an HBM page with a PA of 0. The ReMap table maintains entries for this pair of pages so that CPU requests are correctly directed to their new locations. Previously migrated PCM page to HBM with PA 8192 has turned cold in the bottom left quadrant, and the page is reverse migrated as shown in the bottom right quadrant, where HBM page which

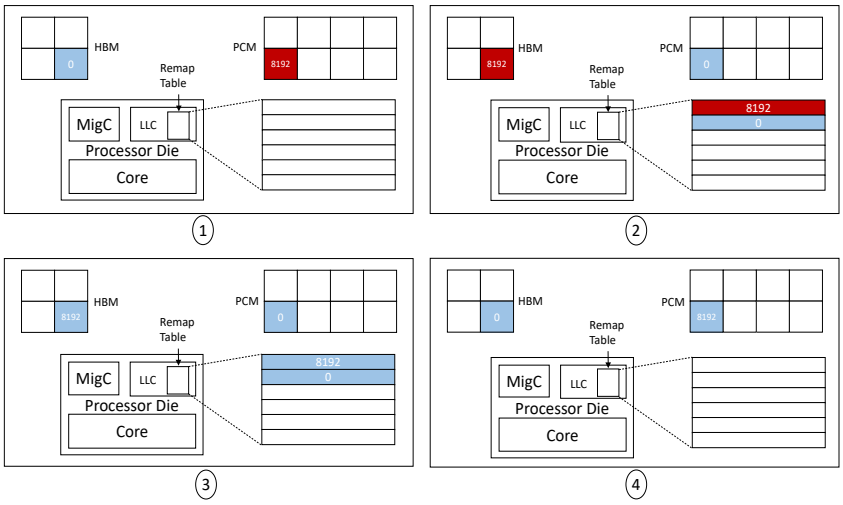

Fig. 4. Reverse Migration Model

contains the page with the PA of 8192 is swapped with the PCM page with the PA of 0. We rely on MigC hardware for both forward and reverse migrations. The ReMap table entries for these pages will be deleted upon completion of the process (as shown in the bottom right quadrant).

Pages selected for reverse migration can be based several different criteria, such as LRU, or MBQ (pages with smaller MBQ are reverse migrated), and the reverse migration can take place on demand or when the ReMap table fills up. Elimination of address reconciliation simplifies the complexity of our MigC hardware, since the hardware no longer needs to initiate cache invalidation, shootdown of TLBs and lock user processes during the reconciliation.

4 EXPERIMENTAL SETUP

In this section we will describe our simulation set up and the benchmarks we used. This set up and workloads are the same that we used in our previous work [13] and the same for the experimental results shown previously in figure 2 in Section 2.4.

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 [16]. Ramulator is a trace driven, cycle-level memory simulator with support for a simple multi-core CPU model with cache hierarchies. Each core is 4-wide out-of-order issue with 128 Reorder Buffer (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. 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 [16] and for PCM timing 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

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

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 2.1). 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 for ReMap table (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 2.2. Finally, we used CACTI [23] to model the power rating of the MigC components.

4.2 Workloads

We use sixteen multi-programmed SPEC CPU2006 [34] workloads, four multi-threaded benchmarks from the US Department of Energy (DOE) provided ECP Proxy Applications [9] 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 [10] to collect a representative

	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

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 in our graphs) 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 [1], LULESH [12], CoMD [21] and miniFE [11]. 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 for our adaptive migration and reverse migration techniques [7].

5.1 Adaptive Hotness Thresholds

As discussed in section 2.4, techniques that use fixed hotness thresholds do not perform well for some applications. In this section we will evaluate our adaptive page migration techniques described in section 3. As described in that section, we monitor the number of pages migrated in a window: page migration overheads can defeat benefits of migrations if too many pages are migrated. We also monitor the benefits of page migration (or MBQ) by tracking the average number of memory accesses to migrated pages in a window. A low MBQ may indicates that the migrated pages are not heavily accessed and thus the migration was not beneficial. This may be the case for applications that are migration unfriendly. Using these measures we either change hotness thresholds (to either increase or decrease the number of pages migrated) or completely stop migrations for a duration.

[7] In Section 2.4 we reported the performance results for migration techniques that use static or fixed hotness thresholds.

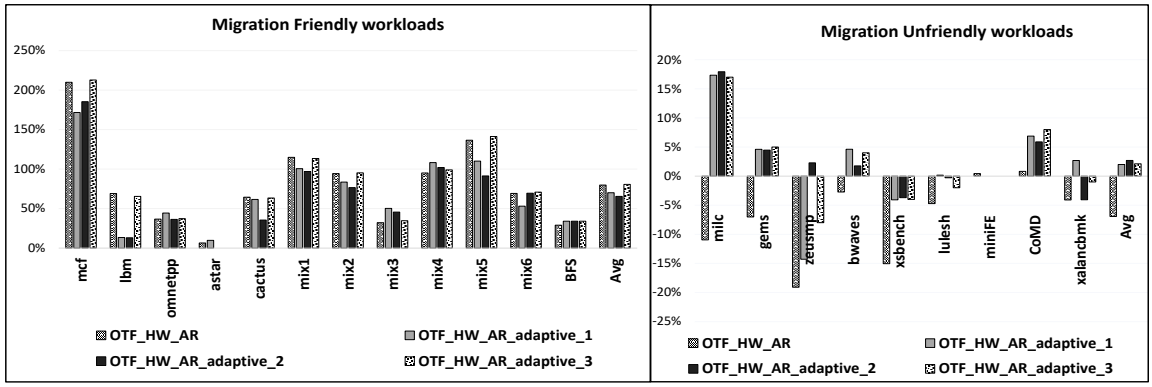

Fig. 5. IPC improvement (%) using Adaptive migration policies over no-migration baseline (negative y-axis shows degradation)

5.2 Evaluation of Adaptive Control Based on Number of Pages Migrated

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 5 shows the IPC improvements using our first adaptive technique (as described in Algorithm 1) over the baseline with no page migration. The bars labeled OTF_HW_AR_adaptive_1 refer to the results from 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 70% performance gains on average over the baseline for migration friendly workloads (the on-the-fly migration with static threshold shows 80% 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 because, when the MigC notices that very few pages are migrated in a window, the threshold is reduced to 64 to increase the migrations. However, these additional migrations do not contribute to performance gains. For example, as described in Section 2.5, for mcf, only 3% of pages receive large number of accesses, and migrating other pages with fewer accesses will not contribute to performance gains, but adds to migration and address reconciliation costs. It may also be the case that for moderately friendly applications, the adaptive technique may aggressively increases hotness threshold and reduce the number of pages migrated, even when the application is benefiting from the page migrations. We will explore this later in Section 5.4.

5.3 Evaluation of Adaptive Control Based on Migration Benefit Quotient

Figure 5 includes the results for our second adaptive technique which uses MBQ to control hotness thresholds, as described in Algorithm 2. The results labeled OTF_HW_AR_adaptive_2 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 66% over the baseline

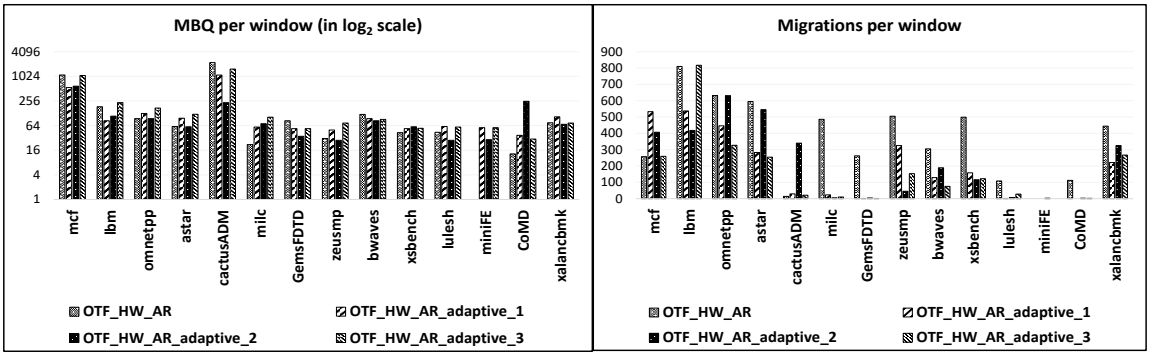

Fig. 6. Traffic data (Left, MBQ per window. Right, Migration count per window)

(compared 80% 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 static threshold technique (e.g., mcf, lbm, cactus and some mixed workloads) for the same reasons outlined in the previous section (Section 5.2.

To understand the performance of our adaptive algorithms better we analyzed migration patterns and accesses to migrated pages, see figure 6. We show the average number accesses to pages migrated to HBM in a window of 4 million cycles (or MBQ) on the left and the average number of pages migrated in a window on the right. Please note that the MBQ per window in Figure 6 is shown in $\log_2$ scale. For most of the migration friendly applications 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. Note that our adaptive techniques try to reduce the number of pages migrated to minimize migration overheads. The exception is lbm. As reported in our previous work [13] and listed in Table 1, lbm is actually classified as only moderately friendly application. 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 5.

5.4 Evaluation of Adaptive Migrations Based on Combined Technique

Based on these observations regarding the two adaptive techniques presented thus far, we explored a third adaptive technique that combines these two methods. This is shown in Algorithm 3. In this method, when the number of pages migrated exceeds the threshold, but the MBQ is high, we will not increase the hotness threshold (which in turn reduces the number of pages migrated), since the high MBQ indicates that page migration is beneficial even when a large number of pages are migrated.

Algorithm 3 Combined Adaptive technique (Initialized values depend on the page size)

```
1:  function ADAPTIVE-MIGRATION(migclk, MBQ)
2:      migwindow = 100000000;
3:      twindow = 4000000
4:      min_MBQ = 50
5:      max_MBQ = 70
6:      upper_MBQ = 100
7:      min_threshold = 32
8:      max_threshold = 256
9:      if migclk mod twindow == 0 then
10:         if (mig_count ≥ max_mig_count) && (threshold < max_threshold) && (MBQ ≤
    upper_MBQ) then
11:             threshold *= 2;
12:         else if (mig_count ≤ min_mig_count) && (threshold > min_threshold) && (MBQ ≤
    max_MBQ) then
13:             threshold /= 2;
14:     if migclk mod migwindow == 0 then
15:         if MBQ ≤ min_MBQ then
16:             pausemigration = true;
17:         else if pausemigration && MBQ ≥ max_MBQ then
18:             pausemigration = false;
19:
```

The results of this combined adaptive technique are also included in Figure 5 which are labelled as OTF_HW_AR_adaptive_3. The figure shows that the combined approach achieves almost the same level of performance as static threshold technique for (very) migration friendly applications. On the other hand, the combined adaptive technique outperforms static threshold and the other adaptive migration techniques for moderately friendly and unfriendly workloads. On average, our combined adaptive migration shows a performance gain of 81% over the baseline (compared 80% gain using static threshold) for migration friendly workloads, and a 2% performance gain on average (compared a performance loss of 7% with static threshold) for unfriendly workloads.

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.

5.5 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 ReMap entries, we migrate pages back to their original locations and remove the corresponding ReMap entries, as described in section 3.3. For our experiments, we select pages based on MBQ: pages with least number of accesses in a window are candidates for reverse migration, after 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. If no pages are suitable for reverse migration, page migration is temporarily delayed. Figure 7 shows the results from our reverse migration experiments.

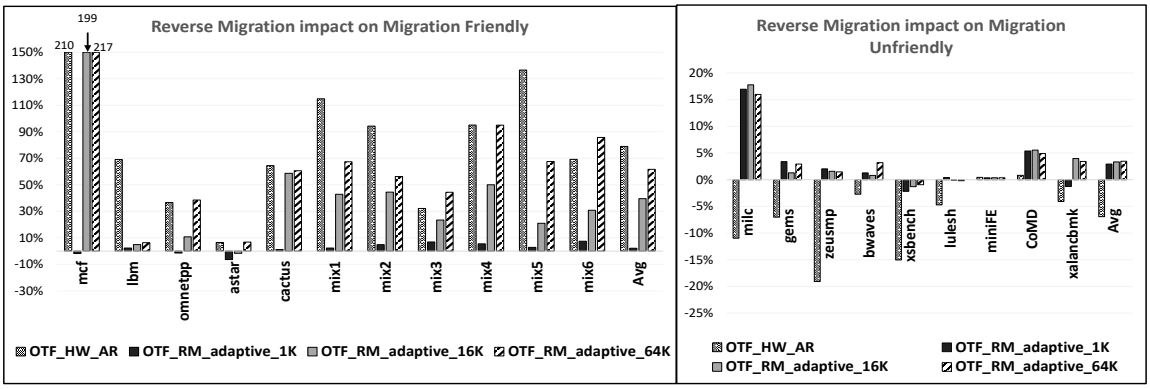

Fig. 7. IPC improvement (%) of Reverse migration over no-migration baseline with different remap table sizes (negative y-axis shows degradation)

We still use adaptive thresholds relying on MBQ as described in Algorithm 2. For comparison purposes, Figure 7 includes data for migration based on static threshold of 128 (previously shown in Figure 2: these results are labeled as OTF_HW_AR (with 1024 ReMap entries). For reverse migration experiments, 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 on-the-fly migrations with reverse migrations, 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 to 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. For example, consider mcf, based on our characterization [13], only 3 percent of the pages account for 90 percent of the memory accesses. So, having such large ReMap table sizes helped in reducing reverse migrations. Except for some very friendly workloads like mix1, mix2, mix5 and lbm, reverse migration (OTF_RM_adaptive_64K) performs on par or even better than OTF_HW_AR. 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 about 1.3Mbytes storage. This table can be placed in HBM while caching a small portion inside the MigC. This is similar to the studies reported in [6, 26, 32]; in those studies the ReMap tables were much larger than what we are proposing.

5.6 Subpage Migration

Although not actually implemented in this study, reverse migration may be useful in systems with very large pages. As the physical memory sizes increase, traditional 4K byte pages also increase the sizes of page tables and TLBs. Page table and TLB sizes can be reduced by using larger pages, say 64KB, 1MB, 2MB or even 1GB pages.

However, it should be noted that only small portions of a large (or huge) page is likely to be "hot" while other portions are not heavily accessed. It will be wasteful to migrate the entire page in such cases. So, instead of migrating large pages, one may consider migrating only hot subpages of large pages. Such subpage migrations require tracking the physical location of subpages. ReMap tables can be extended for this 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 rest of the 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. We will explore reverse migration of subpages for systems using huge pages in future.

5.7 Energy Consumption

Figure 8 shows the dynamic energy savings that are achieved using Adaptive and Reverse migration techniques when compared with the baseline (with no page migrations). For baseline system, we measure energy for all demand requests to faster and slower memory. Energy consumption for our techniques account for energy for demand requests, energy consumed for migration of pages (which requires additional accesses to memory) and energy consumed by the migration controller hardware to manage migrations and address reconciliation. It should be noted that the HBM energy consumption is much lower than PCM as shown in table 5. As the number of accesses to the HBM increases with the decrease in the accesses to PCM when pages are migrated (even accounting for the migration cost itself), noticeable energy savings can be observed. All our on-the-fly migration techniques show energy savings for migration friendly workloads: the hardware address reconciliation technique (OTF_HW_AR) consumes on average 27 percent less energy, while the adaptive techniques (OTF_HW_AR_adaptive_1, OTF_HW_AR_adaptive_2 and OTF_HW_AR_adaptive_3) consume 25, 22 and 29 percent less energy than baseline respectively. As shown in a previous Section 5.1, the execution performance of adaptive techniques (OTF_HW_AR_adaptive_1, OTF_HW_AR_adaptive_2) is slightly lower that the OTF_HW_AR, which uses static threshold. This in turn also results in slightly higher energy consumption for adaptive techniques, while the OTF_HW_AR_adaptive_3 performs slightly better than OTF_HW_AR, and consumes 2% less energy than OTF_HW_AR. The reverse migration techniques with 1K, 16K and 64K ReMap table sizes (OTF_RM_adaptive_1K, OTF_RM_adaptive_16K, OTF_RM_adaptive_64K) show 0, 21 and 29 percent energy savings respectively over the baseline with no page migrations. As stated, OTF_RM_adaptive_1K causes frequent migrations and reverse migrations even for heavily accessed pages which otherwise would have been reconciled. Thus the benefits of the migrations are negated by the reverse migrations. For migration unfriendly workloads, our adaptive techniques saves power up to 15% compared to OTF_HW_AR technique on average. In the case of reverse migration technique with 16K and 64K ReMap tables (OTF_RM_adaptive_16K and OTF_RM_adaptive_64K), ReMap table sizes are large enough to hold hot pages for longer periods, minimizing the need for reverse migration. This in turn results in improved energy savings. In some cases, the benefit of large ReMap table size is sufficient to show energy savings that equal the improvements achieved with the best adaptive technique (OTF_HW_AR_adaptive_3). In our energy computations, we have accounted for the additional power needed for large ReMap table sizes.

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

Table 5. Memory Energy parameters

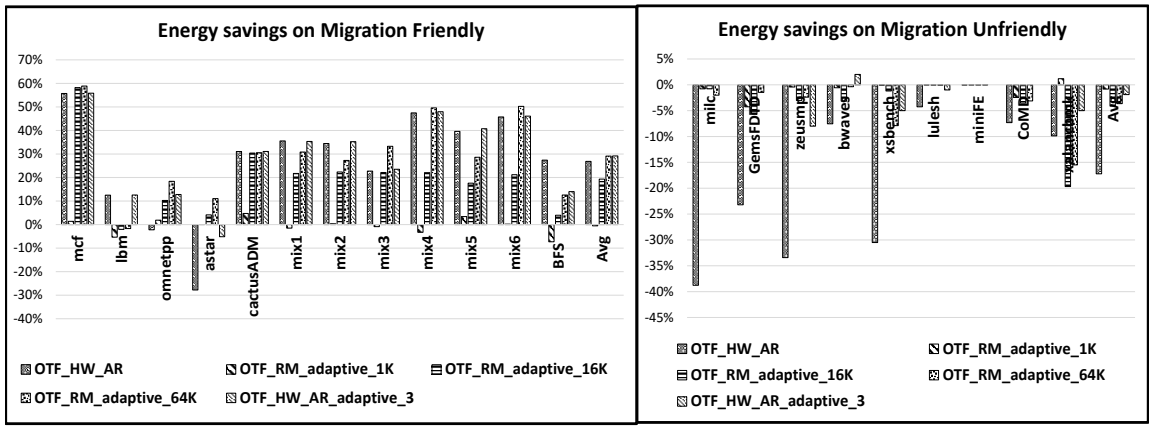

Fig. 8. Energy savings (%) of Adaptive and Reverse migration over no-migration baseline (negative y-axis shows degradation)

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. [20] 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 [26, 32], 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 [6, 17, 26, 32]. Sim et al. used a Transparent Hardware based Management (THM) of flat-address memory [32]. 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 [6], where the migration is done at finer granularity (cache line size) and the migration candidate is chosen on each slow memory access. While Chameleon [18], 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 [6, 32]. 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 [17] 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 [20] 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 AND FUTURE WORK

In this paper, we extended our previous study [13] of on-the-fly (OTF) page migration technique that migrates a page from slow non-volatile memory (NVM) to fast memory (such as HBM) as soon as the page becomes "hot". For this contribution we employ adaptive migration techniques that dynamically change the frequency and the number of pages migrated, and pause or resume migrations based on the memory access behavior of applications. We use three different

adaptive techniques. (1) We monitor the number of pages migrated in a given observation window (mig_count_adaptive). If the number exceeds a threshold, we increase the hotness threshold so that the number of migrations are reduced (likewise, reduce the threshold if the number of pages migrated is below a threshold). (2) We monitor the *Migration Benefit Quotient* (the average number of accesses to page after migration, which indicates the usefulness of a migration, MBQ_adaptive) and either increase or reduce migrations based on the MBQ. We pause migrations temporarily if the MBQ is too low, and resume migrations if the MBQ increases. (3) We explore a combination of the previous two approaches: we monitor the number of pages migrated in a window and increase hotness threshold if too many pages are migrated and the MBQ is low. Likewise, we reduce the threshold when the number of pages migrated is low and MBQ is high. The first technique has resulted in an average of 71% IPC improvement over the baseline for migration friendly workloads, but more importantly, our technique has shown on average 2% performance gains over the baseline even for unfriendly workloads. In terms of energy consumption, this technique has achieved 25% energy savings compared to the baseline for migration friendly and but has consumed 3% more energy than the baseline for unfriendly workloads. This should be compared with static threshold on-the-fly migration technique which consumes 17% more energy than the baseline. The second technique that monitors MBQ, has achieved an average performance gain of 65% over the baseline for migration friendly workloads, 3% gains for migration unfriendly workloads and has resulted in 22% energy savings for migration friendly, while consuming 2% more energy for migration unfriendly applications over the baseline. The third technique which combines the previous two adaptive techniques resulted in 81% performance gains on average over the baseline for migration friendly workloads and 2% gains even for migration unfriendly workloads. Our adaptive techniques eliminates the need for off-line profiling of applications to categorize them as a migration friendly or unfriendly. We feel that additional adaptive techniques may further improve the performance gains when relying page migrations in multi-level heterogeneous memories.

We also experimented with reverse migration of pages, eliminating address reconciliation (making the page migration completely transparent to OS). When ReMap table is almost full, ReMap entries are freed for new page migrations by reverse migrating pages with low MBQ back to their original locations. We tested with ReMap table sizes of 1K, 16K and 64K entries. The reverse migration technique has achieved performance improvements of 2%, 41% and 57% respectively for migration friendly workloads and 3%, 3% and 3% respectively for migration unfriendly workloads. Regarding energy savings, reverse migration technique has achieved 0%, 21% and 29% for migration friendly and has consumed 1%, 4% and 4% respectively for migration unfriendly workloads. It can be observed that the static OTF technique (OTF_HW_AR) achieved 28% energy savings, that is lower than reverse migration technique with 64K ReMap table entries by 1% but require more area for the extra ReMap table. We feel that reverse migration can be used when dealing with very large pages. Since migrating huge pages can be very expensive, one can consider migrating only hot subpages of a huge page and reverse migrate the subpages as they become cold. We plan to explore such subpage migrations in the future.

8 ACKNOWLEDGEMENTS

This research is supported in part by NSF Awards #1828105 and #1361806. The authors would also like to acknowledge Nuwan Jayasena and Mike Ignatowski of AMD for their feedback and suggestions throughout the conduct of this research.

REFERENCES

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

[2] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. 2017. Avoiding TLB shootdowns through self-invalidating TLB entries. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 273–287.

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

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

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

[6] 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 Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1–12.

[7] HMC Consortium. 2018. Hybrid Memory Cube Consortium. http://hybridmemorycube.org/. [Online; accessed July-27-2018].

[8] Joint Electron Devices Engineering Council. 2018. 3D ICs. http://www.jedec.org/category/technology-focus-area/3d-ics-0. [Online; accessed July-27-2018].

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

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

[11] M Heroux and S Hammond. 2015. MiniFE: finite element solver.

[12] RD Hornung, JA Keasler, and MB Gokhale. 2011. *Hydrodynamics challenge problem.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[13] Mahzabeen Islam, Shashank Adavally, Marko Scrbak, and Krishna Kavi. 2020. On-the-Fly Page Migration and Address Reconciliation for Heterogeneous Memory Systems. *To Appear in ACM Journal on Emerging Technologies in Computing Systems* (2020). http://csrl.cse.unt.edu/kavi/Research/JETC-2019.pdf

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

[15] Kimberly Keeton. 2017. Memory-Driven Computing. USENIX Association, Santa Clara, CA.

[16] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49.

[17] A. Kokolis, D. Skarlatos, and J. Torrellas. 2019. 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.

[18] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 533–545. https://doi.org/10.1109/MICRO.2018.00050

[19] E. Kultursay, M. Kandemir, Sivasubramaniam, and O. A., Mutlu. 2013. 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.

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

[21] Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C Germann. 2013. Co-design for molecular dynamics: An exascale proxy application, 2013.

[22] David Mosberger and Stephane Eranian. 2001. *IA-64 Linux kernel: design and implementation.* Prentice Hall PTR.

[23] N. Muralimanohar, Rajeev Balasubramonian, and N. Jouppi. 2007. CACTI 6 . 0 : A Tool to Understand Large Caches.

[24] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 309–319.

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

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

[27] Moinuddin K Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. Phase Change Memory: From devices to Systems. *Synthesis Lectures on Computer Architecture* 6, 4 (2011), 1–134.

[28] 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.

[29] 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-International Symposium on High-Performance Computer Architecture*.

[30] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*. 352–362. https://doi.org/10.1145/3205289.3208064

[31] Jee Ho Ryoo, Mitesh R Meswani, Andreas Prodromou, and Lizy K John. 2017. SILC-FM: Subblocked interleaved cache-like flat memory organization. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 349–360.

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

[33] F. Song, S. Moore, and J. Dongarra. 2009. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*. 1–10. https://doi.org/10.1109/CLUSTR.2009.5289173

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

[35] 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 2015 International Symposium on Memory Systems*. ACM, 167–178.

[36] 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 *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 340–349.

[37] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3123939.3124555

[38] I. Ştirb. 2018. NUMA-BTLP: A static algorithm for thread classification. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*. 882–887. https://doi.org/10.1109/CoDIT.2018.8394925