A Case for Granularity Aware Page Migration

Jee Ho Ryoo*
ARM Inc.
San Jose, California
jeeho.ryoo@arm.com

Lizy K. John
The University of Texas at Austin
Austin, Texas
ljohn@ece.utexas.edu

Arkaprava Basu†
Indian Institute of Science
Bangalore, India
arkapravab@iisc.ac.in

ABSTRACT

Memory is becoming increasingly heterogeneous with the emergence of disparate memory technologies ranging from non-volatile memories like PCM, STT-RAM, and memristors to 3D-stacked memories like HBM. In such systems, data is often migrated across memory regions backed by different technologies for better overall performance. An effective migration mechanism is a prerequisite in such systems.

Prior works on OS-directed page migration have focused on *what data to migrate* and/or on *when to migrate*. In this work, we demonstrate the need to investigate another dimension – *how much to migrate*. Specifically, we show that the *amount* of data migrated in a single migration operation (called "migration granularity") is vital to the overall performance. Through analysis on real hardware, we further show that different applications benefit from different migration granularities, owing to their distinct memory access characteristics. Since this preferred migration granularity may not be known a priori, we propose a novel scheme to infer this for any given application at runtime. When implemented in the Linux OS, running on a current hardware, the performance improved by up to 36% over a baseline with a fixed migration granularity.

ACM Reference Format:
Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *ICS '18: International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3208064

1 INTRODUCTION

Scaling of DRAM capacity has slowed down due to challenges in designing charge-based memories with smaller transistors. Fortunately, several non-volatile memory (NVM) technologies like Phase Change Memory (PCM) [55], STT-RAM, memristor [15], and MRAM are emerging as potential supplements to the DRAM [50]. NVMs are denser (higher capacity) and scale well with smaller transistors [27]. In an orthogonal technology trend, 3D integration techniques have allowed the stacking of DRAM chips to give rise to the high-bandwidth memories [17]. Each of these disparate technologies has different access latency, bandwidth, energy dissipation, and density. Consequently, memory systems of future computers are becoming increasingly heterogeneous.

Irrespective of the combination of memory technologies deployed, a common aspect of any system with heterogeneous memory is that the technology with relatively superior access characteristics (e.g., lower latency, more bandwidth) is also more capacity constrained. For example, in a system with DRAM and PCM [42], DRAM is faster, but the PCM has higher capacity. Alternatively, if the memory system is composed of 3D-stacked memory like high bandwidth memory (HBM) and conventional DRAM [31], then HBM provides much higher bandwidth, but has a comparatively smaller capacity. For the ease of reference, we will refer the memory region backed by the technology with superior access characteristics as the *fast memory* and the rest as the *slow memory*. For example, HBM is the fast memory in a system with DRAM and HBM while DRAM is the fast memory in a system with DRAM and PCM.

Servicing application memory accesses from the fast memory is preferable for better performance. However, if the memory footprint of an application exceeds the capacity of the fast memory, then migrating data between the fast and slow memory is critical to service larger fraction of accesses from the fast memory (referred as the fast memory hit ratio) [18]. Thus, an effective migration mechanism is essential for any heterogeneous memory system.

Data can be migrated under the hardware or the software control. Hardware-managed migration allows fine-grain migrations (e.g., 64B cache-line size) [31] with low overheads and does not require software modifications. But they require intrusive hardware modifications and adds significant hardware overheads (~13% capacity loss of fast memory [18]). Importantly, the jury is still out on what hardware is essential for migration, and consequently, the proposed hardware enhancements are yet to become commercially available. Alternatively, an application-managed migration [28] can avoid hardware modifications and utilize application-specific knowledge, but needs to re-write applications. A third approach is an OS-directed page migration [39] that avoids both custom hardware and application modifications. Instead, the OS initiates a migration as and when deemed necessary. However, the migration has to happen at a coarser granularity of pages (e.g., 4KB) and thus, also called page migration. Importantly, previous works [39] have shown that OS-directed migration can add significant overhead due to long-latency TLB shootdown operations. Shootdown is necessary to alter the virtual-to-physical address mapping of a page after the migration [39]. In this work, we show how the efficacy of OS-directed migration mechanisms can be improved through the novel use of adaptive granularity of migration.

*The author contributed while he was an intern at AMD Research.
†The author contributed while he was a member of technical staff at AMD Research.

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3208064

Prior works [31] on page migration have focused on primarily two aspects – what page to migrate and when to migrate. In this work, we demonstrate the need to explore another dimension, – *how much to migrate*. We find that the *migration granularity* significantly impacts the performance of a migration scheme. We define the migration granularity as the size of a *contiguous virtual memory region* [1] in an application's address space that is migrated in one migration operation. A larger granularity amortizes the overheads of migration better. For example, only a single TLB shootdown is needed to complete a migration, irrespective of its granularity. However, precious capacity in the fast memory is wasted if an application does not utilize the larger chunk of data migrated into the fast memory due to the larger granularity. This could lower fast memory hit ratio and increase the total number of migrations.

Our empirical analysis performed, on current hardware running Linux®, showed that *the migration granularity that leads to the best performance for a given application varies across applications*. Specifically, we studied three different migration granularities as an example – 4KB, 64KB and 2MB. We found that applications like `xsbench`, which demonstrate near-random memory access patterns, perform best with the smallest granularity (here, 4KB). Applications with a streaming behavior (e.g., `lulesh`) prefer a large migration granularity (here, 2MB). They benefit from the better amortization of migration overheads and the implicit prefetch of useful data to the fast memory. Interestingly, for applications like `graph500`, use of smallest granularity adds to migration overheads, but the performance drops with the largest granularity due to extraneous migration. Such applications have some spatial reuse across, but not as much as a streaming application.

Unfortunately, the preferred migration granularity for an application is not known a priori. Applications also demonstrate phase behaviors, and thus, may prefer different granularities during different phases of execution. We thus, propose a dynamic scheme that adjusts the migration granularity at runtime based upon application behavior. We monitor page-grain meta-data (e.g., access bits) in the OS to estimate application's spatial access locality across different migration granularities. This information is then used to employ the largest granularity that is estimated to contain *enough* spatial locality. This ensures that the overhead of migration is amortized as much as possible without wasting fast memory capacity. An implementation of this scheme in Linux® is projected to improve application performance by up to 36% and by 11% on average, over a baseline page migration scheme using a fixed granularity.

Our contributions in this paper are as follows:

- We demonstrate the need to consider the migration granularity in designing OS-directed page migration scheme.
- By analyzing a wide range of applications on real hardware, we show that different applications perform better with different page migration granularities.
- We propose and evaluate a novel dynamic scheme in the OS to adapt migration granularity at runtime.

[1] Not necessarily contiguous in physical memory, unlike large pages. More details are in Section 7.1

2 BACKGROUND

We now discuss technologies behind the emergence of heterogeneity in the memory and approaches to managing this heterogeneity.

2.1 Emerging Memory Technologies

Scaling of DRAM technology is reaching its limit due to physical design constraints. This can stifle the growth of memory capacity in DRAM-only systems. Fortunately, emerging NVM technologies are denser than DRAM and scale better with shrinking transistor size. However, they are often slower than the DRAM and provide limited write endurance. Thus, NVMs are emerging as *supplement* rather than alternative to DRAM [31]. Consequently, memory subsystems of future computers are likely to have both DRAM and NVMs.

Beyond NVMs, 3D-stacked DRAM technologies like High Bandwidth Memory (HBM) is adding to the heterogeneity [39, 40, 48]. HBM is even more capacity-constrained than DRAM but provides higher bandwidth [34]. HBM's limited capacity means that the memory system cannot solely comprise of HBM. It needs to be complemented by DRAM and/or NVM.

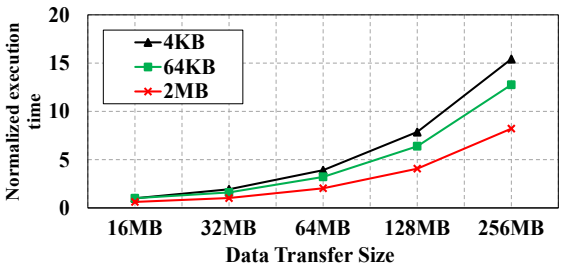
2.2 Managing Heterogeneous Memory

A common characteristic of any heterogeneous memory system is that the fast memory is always relatively more capacity constrained, irrespective of the specific technologies employed. Thus, a goal of any such system is to service a larger fraction of an application's memory accesses from the fast memory for better overall performance – i.e., to achieve higher fast memory hit ratio (henceforth, referred as *FM-hit ratio*). However, an application's memory footprint may not fit in the capacity-constrained fast memory. Thus, an effective migration mechanism between the fast and slow memory is key to achieving a high FM-hit ratio.

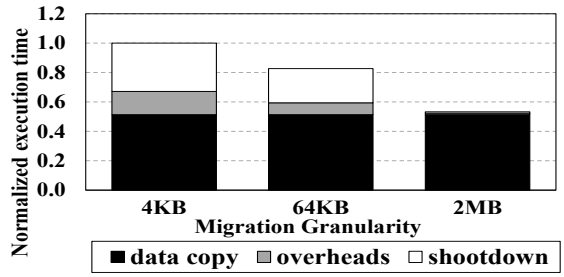
From software's perspectives, it is best if hardware manages the fast memory as a cache [16, 18–20, 31, 32, 41, 47]. This requires no software modifications but comes with typical costs of hardware caching. Significant hardware resources have to be devoted to bookkeeping (e.g., tags). Moreover, it hides the capacity of fast memory from the software. Thus, this approach is applicable only when the fast memory capacity is very limited.

On the other extreme, an application writer can be tasked to manage the heterogeneity in the memory [5, 28]. Here, the application writer can choose to place data in either fast or slow memory through specific APIs (e.g., "malloc-fast" and "malloc-slow") and/or request to migrate data [28]. While this approach requires no hardware modification, rewriting applications is a herculean task.

Alternatively, the OS can dynamically migrate data between the fast and slow memory based on application's (perceived) needs [11, 14, 22, 36, 39]. However, this has its own challenges. First, since the OS has no visibility to application's loads/stores, it is a challenge to decide which data is most beneficial to migrate. In one approach, Oskin et al. [39] proposed to monitor loads/stores to slow memory by generating page faults each such access. Unfortunately, a page fault requires several microseconds to service. Alternatively, Meswani et al. [35] proposed to modify the hardware to extend each Page Table Entry (PTE) to keep access counts to identify frequently accessed pages. It then migrates these hot pages to the fast memory. Unfortunately, such hardware does not exist today.



(a) Latency to migrate contiguous virtual memory.



(b) Breakdown of migration latency for different granularities.

Figure 1: Impact of migration granularity.

Researchers have also proposed to use already-existing *access bits* in PTEs to approximate the frequency of access [35]. Modern commercial processors set this access bit (if unset) in the PTE whenever the corresponding page is accessed. This approach does not require any hardware modifications. We use this approach to identify hot pages and therefore, do not require custom hardware.

The other challenge for OS-directed migration is that it can add significant overheads due to associated slow operations like TLB shootdown (e.g., 13.2 μsec) [39]. Migration alters the virtual-to-physical address mapping. However, address mapping could be cached in CPU's Translation Lookaside Buffers [2] (TLBs). TLB shootdown is the process of purging such stale mapping from TLBs. A shootdown typically involves slow operations such as issuing inter-process-interrupts (IPIs) by the OS. We empirically found that such TLB shootdowns can dominate the cost of a page migration (Section 3), corroborating previous work [28, 39].

In summary, OS-directed page migration requires no application or hardware modifications. However, it can add significant performance overhead. We explore ways to reduce this overhead.

3 THE NEED TO CONSIDER MIGRATION GRANULARITY

Migration is anything but a free lunch. We consider any time spent beside copying of the data from one type of memory to the other as the overhead of migration. For example, TLB shootdowns and the software cost of invoking migration [35, 39, 53] is pure overhead.

We quantify the impact of granularity on migration overhead on a x86-64 hardware running Linux (details in Table 1). Figure 1a shows the normalized time spent to migrate a varying amount of data at different granularities across two NUMA memory zones. We used *numa_move_pages()* to migrate a desired number of pages between different NUMA nodes. The x-axis shows the amount of data being migrated. The y-axis shows the time required to migrate. The migrated data is allocated contiguously in the virtual memory, which is key to leverage larger granularities. Three lines in the graph represent three different granularities of migration. We observe that as the amount of data being migrated increases, the larger granularity takes less time to migrate a given amount of data by amortizing overheads.

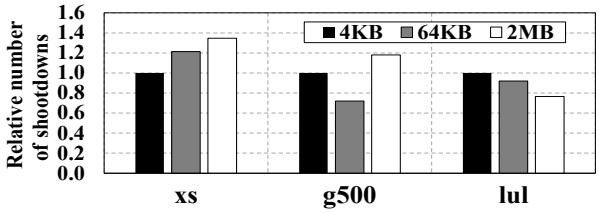


Figure 2: Number of TLB shootdowns (lower is better).

Figure 1b breaks down normalized execution time (normalized to 4KB granularity) while migrating a fixed amount data (here, 256MB) to depict sources of overhead. Each bar shows the time spent on copying the data, TLB shootdown and other software overheads such as initiating the process of migration. We observe that the cost of copying data remains same irrespective of migration granularity since the total amount of data copied remains unaltered. We note that TLB shootdowns are important contributors to migration overheads. Importantly, overheads due to TLB shootdown and other sources scale down with increasing migration granularity. This is expected since the number of shootdowns decreases in proportion to the increasing migration granularity as long as the data being migrated is contiguous in the virtual memory. For example, migrating a given amount of virtually contiguous data using 2MB granularity incurs 512× fewer shootdowns than 4KB granularity. The key is to define the migration granularity as a contiguous *virtual* memory region instead of a contiguous physical memory region. This allows a single TLB shootdown to complete a migration irrespective of its granularity since a shootdown invalidates a contiguous virtual address range, by definition.

However, a larger migration granularity does not always improve an application's performance. A larger granularity is beneficial if an application demonstrates mostly streaming behavior since they benefit from migrating contiguously allocated data together (i.e., larger migration granularity). It decreases the total number of migrations, and consequently, the number of TLB shootdowns [3]. Other migration overheads decrease too. Additionally, larger granularity implicitly *prefetches* data to be accessed next by the application into the fast memory. However, a larger granularity can waste precious fast-memory capacity for applications with low spatial reuse. Additional data migrated due to larger granularity can evict otherwise useful data from the fast memory. The evicted data may later need

[2] TLBs are hardware structures in processors that cache recently used PTEs to make address translation process fast. Typically, each CPU core has its own TLB hierarchy

[3] In all our experiments, the number of TLB shootdowns are exactly twice the number of migrations, and thus, the number of shootdowns is a good estimator of the relative overhead of migration both due to shootdown and other software invocations.

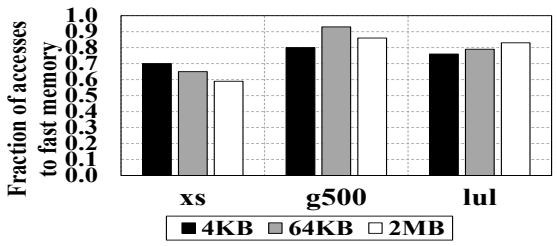


Figure 3: Fast memory hit ratio (higher is better).

to be migrated back again. This could increase the total number of migrations (thus shootdowns), and consequently, add to overheads.

To empirically understand how an application's performance is impacted by migration granularity, we measured the number of TLB shootdowns and the fast memory hit ratio. The number of shootdowns is a good indicator of the migration overheads (lower is better) while the fast memory hit ratio captures usefulness of migration (higher is better). Together, they provide an effective estimate of how migration aids (or hurts) application-performance in a heterogeneous memory system. We counted the number of migration (shootdowns) inside the Linux and measured the fast-memory hit ratio using Instruction Based Sampling (IBS) [8] feature in AMD processors. Methodology is detailed in Section 5.

Figure 2 shows the relative number of migrations (thus TLB shootdowns) for three representative applications with different migration granularities (4KB, 64KB and 2MB). Each subgraph represents one application, and three bars in each subgraph represents the number of migrations under three different granularities. The number of migrations is normalized to that with the default 4KB granularity for each application. Figure 3 shows similarly structured graph for the fast-memory hit ratio (henceforth, *FM-hit ratio*).

We observe that the application `xsbench` ("xs") shows the least number of migrations (Figure 2) while experiencing a better FM-hit ratio (Figure 3) with the smallest migration granularity (4KB). As the granularity increases, both metrics worsen for `xsbench`. In hindsight, this is expected since `xsbench` demonstrates near-random access patterns. On the other end of the spectrum is `lulesh` ("lu"). It incurs the least number of migrations and the highest FM-hit ratio with the largest granularity (2MB). This workload demonstrates streaming behavior, and thus, benefits by migrating larger chunks of contiguous virtual memory together. Application `graph500` ("g500") falls somewhere in the middle where it performs best with 64KB granularity. It has spatial locality that benefits from a larger granularity, but the locality falls off beyond 64KB region.

Figure 4 shows the normalized projected runtime (lower is better) of these example applications with different migration granularities (Section 5 details methodology) in a system with heterogeneous memory. The ratio of the fast to slow memory latency is 1:3 in these experiments. The figure is arranged similarly to previous ones. The y-axis represents projected runtime of each application normalized to application's runtime when the fast memory capacity is configured to fit the entire memory footprint (i.e., when *no* migration is incurred). We observe that the migration granularity could significantly impact the performance of an application running on a system with heterogeneous memory (e.g., 23% for `lulesh`). As expected from the data presented in the Figure 2 and Figure 3, application `xsbench` performs the best with the smallest migration granularity of 4KB, `lulesh` prefers 2MB granularity, and `graph500` works the best with 64KB granularity.

Summary: ① By defining migration granularity as contiguous *virtual* memory region rather than physical memory, we enable potential amortization of migration overheads with a larger granularity. ② We demonstrate that the migration granularity is a key factor in an OS-directed page migration mechanism. ③ The preferred migration granularity can differ across applications.

4 DYNAMICALLY SELECTING MIGRATION GRANULARITY

While the migration granularity is important, an application's preferred granularity may not be known a priori. Further, applications are known to demonstrate phase behaviors [45], and thus, may prefer different granularities during different execution phases. Therefore, we propose a novel dynamic scheme that adapts the migration granularity depending upon memory access pattern of an application. This can enhance the efficacy of, not just ours, but any, OS-directed page migration scheme.

Our goal is to conceive a simple scheme, implementable in the OS, to adapt the migration granularity for an application at runtime to: ① reduce the number of TLB shootdowns (thus reduce the migration overhead), ② increase the FM-hit ratio (thus enhancing the usefulness of a migration). We achieve this by employing the largest migration granularity as long as the migrated data is used *enough* by the application. This amortize overheads of migration without migrating useless data into the capacity-constrained fast memory. A way to infer if a larger migration granularity could be helpful is to monitor application's spatial access locality across contiguous virtual memory regions (i.e., migration granularities). If there exists substantial locality within a larger contiguous virtual memory region, then we use a larger migration granularity. However, if we observe low spatial locality within a given virtual memory region of an application, then we fall back to using a smaller granularity.

We keep a per-application variable that records the migration granularity at any given time. Its value could be 4KB, 64KB or 2MB in our experiments. However, our proposal works with any granularity that is a multiple of 4KB (base page size). Our dynamic algorithm alters the value of this variable at runtime based on the application behavior as follows.

For ease of explanation, we divide our proposal in *move-in* and *move-out* path. In the move-in path, data is migrated from the slow to fast memory while, in the move-out path, migration happens in the opposite direction. The decision on whether to increase the migration granularity is taken in the move-in path, and whether to decrease the migration granularity is decided in the move-out path.

The move-in and move-out mechanisms are executed as part of two independent OS threads. Both threads independently perform the periodic scanning of *contiguous virtual memory ranges* to ascertain application's access locality over the scanned region and then decide on migration. Scanning is performed by examining access bits of PTEs of 4KB (base) pages in a given region to identify which of those pages are accessed (Figure 5). This is a key innovation of our proposal – scanning on virtual memory for deciding what to migrate, instead of physical memory, unlike previous works

FN Size
425MB

benchmark	Baseline	4KB	64KB	2MB		4KB	64KB	2MB
g500	36.05	39.11	36.12	40.34		1.084882	1.001942	1.119001
xs	137.94	159.39	169.23	186.32		1.155502	1.226838	1.350732
lul	29.45	40.68	38.73	33.6		1.381324	1.31511	1.140917

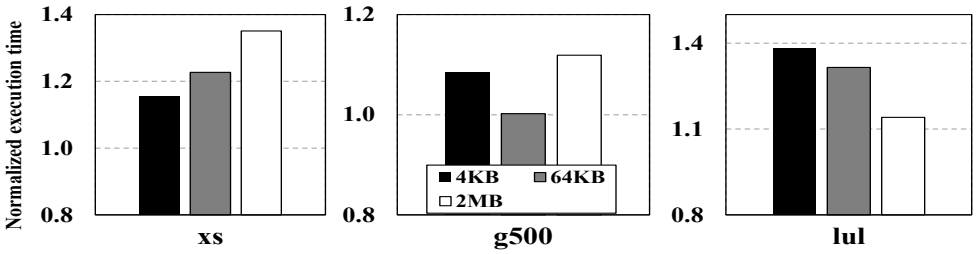


Figure 4: Impact of migration granularity on application performance.

(e.g., [14, 22, 35]). This helps to ascertain an application's access locality accurately by ignoring the mapping between virtual and physical memory which is unrelated to application behavior.

Each periodic scan starts on a physical page frame just after where the previous scan stopped. In case of the move-in path, this will be the page frame in the slow memory. It then reverse-maps the physical address to the virtual page address and starts scanning contiguously on the virtual address space starting from that address. The scanning continues until a pre-configured number of pages that maps to the slow memory is scanned (a typical value is 4,096). During the scan, it examines the access bit of 4KB pages in the region. It uses this information to ascertain how many 2MB contiguous virtual memory regions in the scanned memory have more than a threshold number of 4KB pages with access bit set (`thr_high_2MB`). This essentially counts the number of 2MB contiguous virtual memory regions with *enough* spatial locality. Similarly, it also collects how many 64KB contiguous regions that have more than a threshold number of 4KB pages (`thr_high_64KB`) with the access bit set. At the end of the scan, if the number of such 2MB regions with enough spatial locality is above a given threshold (`thr_high_num_reg_2MB`), then it sets the migration granularity to 2MB. Otherwise, if there is more than a threshold number of such 64KB regions (`thr_high_num_reg_64KB`) with high spatial locality, then the granularity is set to 64KB. Typical values of thresholds are – `thr_high_2MB`: 480, `thr_high_64KB`: 10, `thr_high_num_reg_2MB`: 3, `thr_high_num_reg_64KB`: 3.

The move-out path similarly scans contiguous virtual memory but starts with a physical page frame in the fast memory instead of slow memory. Similar to the move-in path, it also scans contiguous virtual memory regions. Differently, though, it ascertains

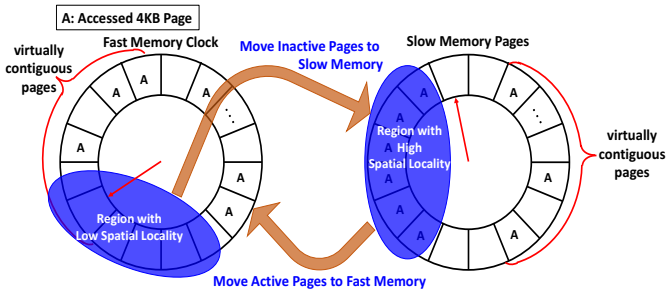


Figure 5: Overview of dynamic scheme for assessing locality.

Processor	Values
Number of Cores	4
Frequency	4.1 GHz
Caches	**Values**
L1 I-Cache	128 KB, 2 way
L1 D-Cache	64 KB, 4 way
L2 Cache	4 MB, 16 way
Fast Memory	**Values**
Type	DDR3-1866
Capacity	400 MB
Slow Memory	**Values**
Type	PCM (emulated)
Capacity	8 GB

Table 1: System parameters

which 2MB and 64KB regions have *not* experienced enough spatial access locality. As before, we use access bits in PTEs of 4KB pages within scanned regions for this purpose. Specifically, during a scan, it counts the number of 2MB regions that have fewer than a threshold number of 4KB pages with their access bits set (`thr_low_2MB`). Similarly, we track access locality in 64KB regions using another threshold `thr_low_64KB`). At the end of a scan, if the number of sparsely accessed 2MB and 64KB regions are above two thresholds, `thr1_downgrade_2MB` and `thr_low_num_reg_64KB` respectively, then the migration granularity is changed to 4KB. If the number of sparsely accessed 64KB regions is below the threshold `thr_low_num_reg_64KB`, and the current granularity is 2MB, then the granularity is set to 64KB. Typical threshold values are as follows – `thr_low_2MB`: 480, `thr_low_64KB`: 10, `thr_low_num_reg_2MB`: 3, `thr_low_num_reg_64KB`: 3.

Note that since we scan the virtually contiguous memory, the 4KB pages in the scanned regions can fall either in the fast or slow memory. However, in the move-in path, we should consider only pages in the slow memory, and in the move-out path, we need to consider only pages in the fast memory. Therefore, our scanning algorithm in the move-in path ignores the access bit information of any pages mapped to the fast memory. In the move-out path, it ignores pages mapped to slow memory.

5 IMPLEMENTATION AND METHODOLOGY

We now describe how we prototyped the above-mentioned granularity-aware migration mechanism in Linux. We then present our evaluation methodology. Finally, we briefly describe the baseline and the workloads.

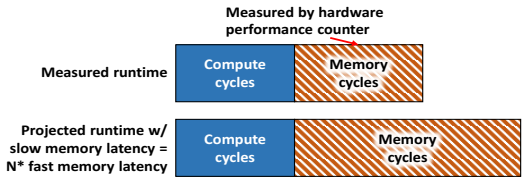


Figure 6: Leading load performance model for system without heterogeneous memory.

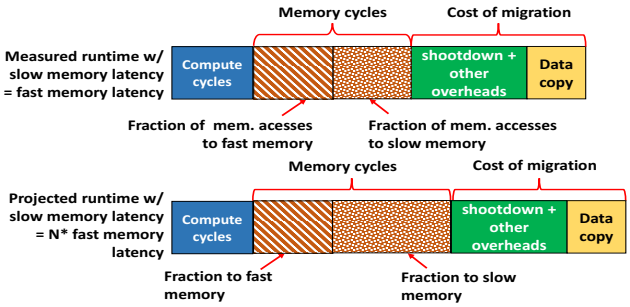


Figure 7: Leading load performance model extended for system with heterogeneous memory and page migration.

Implementation: We evaluate our proposal on an AMD A10-6800K® processor running Linux (kernel version 3.16.36). Table 1 lists the configuration of our experimental platform. Our goal is to evaluate the granularity-aware migration scheme in a heterogeneous memory system with DRAM and NVM. While non-volatile memory technologies like Intel 3DXpoint are emerging, they are still neither commercially prevalent nor easily affordable. Therefore, we emulate the heterogeneous memory with a DRAM-only system as done in almost all recent work [14, 22].

We use Linux's NUMA emulation feature to divide the aggregate physical memory available in the system (32GB) into equally sized contiguous physical memory zones. Pages can then be migrated across these zones. We then make one memory zone to act as the fast memory and another as the slow memory. We needed to ensure that applications' memory footprint surpasses the capacity of the fast memory in order to observe any meaningful number of migrations. We utilized Linux's Memory HotPlug feature *offline* part of physical memory in the fast memory node as if it did not exist. This way we limited the capacity of the fast memory to 400MB in the baseline configurations to emulate the real-world scenario where the fast memory is a fraction of application's memory footprint. We list memory footprint of our workloads in Table 2. However, we also varied this fast memory capacity in sensitivity studies (Section 6.1).

We then extended Linux to support three different migration granularities – 4KB, 64KB, and 2MB. Each migration operation requires one TLB shootdown irrespective of its granularity. However, each shootdown routine is slightly different depending upon the migration granularity. For 4KB granularity, the shootdown routine executes x86-64's *invlpg* instruction to invalidate the stale address mapping in local TLBs of each (receiving) core (background in Section 2). For 64KB, the shootdown routine at each core executes 16 *invlpg* instructions to invalidate all sixteen 4KB pages that the 64KB region maps to. For 2MB pages, instead of looping over 512 pages, it writes to the local core's *cr3* register. Writing to *cr3* flushes all entries belonging to the user applications from the local TLBs.

We separately implemented a driver for dynamically altering the migration granularity. The driver monitors access patterns of applications and alters the page migration granularity by setting a variable that encodes the migration granularity (description in Section 4).

Evaluation methodology: To evaluate the granularity-aware migration mechanism, we collect two key statistics –(1) the number of TLB shootdowns and the time spent on them, and (2) the fast-memory hit ratio (FM-hit ratio). The first metric is a good indicator of the overheads of migration, while the later estimates the usefulness of migrations (Section 3). While the shootdown information is easily obtained by minimally instrumenting the kernel, the FM-hit ratio is not readily available since the OS has no visibility to individual memory accesses by an application. To address this, we utilize a hardware feature called Instruction Based Sampling (IBS) [8] available in AMD CPUs. IBS can track every n^{th} instruction (user settable) that goes through CPU core's pipeline. As the instruction flows through the CPU pipeline, information about several events caused by that instruction is gathered by the hardware. Then, when a tracked instruction completes, the collected information is logged in a kernel buffer. Each sample includes, among others, the type of the instruction, whether it missed processor caches, and virtual/physical address of the access (if load/store). We use the open source IBS driver for Linux [12] to sample one in every one thousand instructions to collect a trace of sampled instructions. We post-process this trace to find the fraction of physical addresses accessed by an application (after missing in processor caches) falls in NUMA zone of the fast memory. This estimates the FM-hit ratio.

The measured numbers of TLB shootdowns, the number of migrations, and estimated FM-hit ratio determine both the overheads and usefulness of any OS-directed migration scheme. However, this does not immediately allow us to estimate the performance of an application with migrations and with different access latencies of fast and slow memory.

We, therefore, used a previously validated *leading-load model* for projecting performance under varying CPU core frequency and memory latency [49]. The key idea of this model is to break the

Notation	Suite	Workloads	Input	RSS
g500	-	graph500	-	761MB
gups	HPC	gups	-	2.00GB
str	HPC	stream	-	1.08GB
cg	NAS	cg	C	890MB
ft	NAS	ft	B	1.26GB
is	NAS	is	C	1.03GB
mg	NAS	mg	B	490MB
ua	NAS	ua	C	483MB
can	PARSEC	canneal	native	939MB
freq	PARSEC	freqmine	native	678MB
lul	CORAL	lulesh	-	696MB
mini	Mantevo	minife	-	642MB
xs	CORAL	xsbench	large	5.55GB

Table 2: Workload descriptions

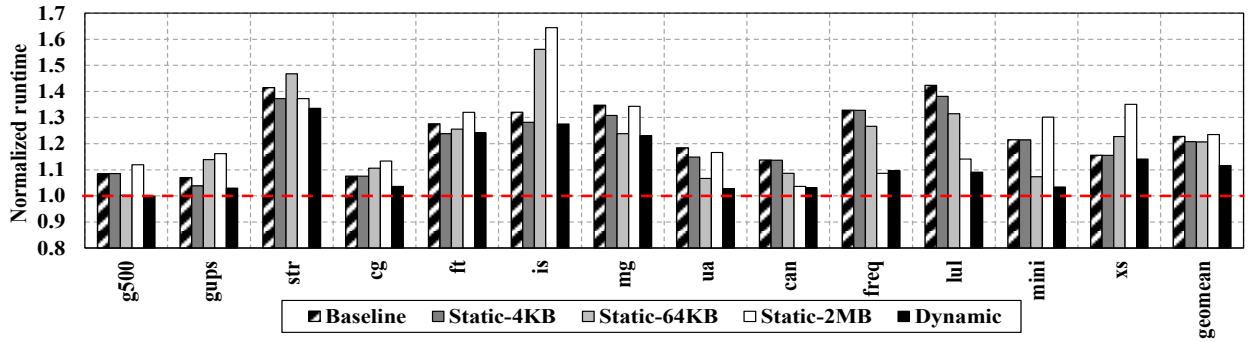


Figure 8: Normalized to execution time with no migration (entire memory footprint in fast memory, lower is better).

execution time of an application in two parts as shown in the top part of Figure 6: ① compute cycles – the portion of execution cycles that scales with CPU core frequency, and ② memory cycles – the portion of execution cycles that is dependent on memory latency. In general, memory cycles are the cycles during which the CPU is blocked waiting for memory accesses [4]. The lower part of Figure 6 shows how this model can predict the performance of an application with different memory latency by scaling the memory cycles proportionally while keeping compute cycles unaltered. This model is shown to predict performance with an average error under 2.7% across a range of workloads under varying core frequencies and memory latencies [49]. We directly measure the memory cycles using a performance counter as in the previous work [49].

The above model, however, works only if the latency of all memory accesses is increased or decreased equally. We thus extended this leading load model as shown in the Figure 7. From the IBS, we measured the FM-hit ratio that helps identify the fraction of accesses to the fast versus slow memory. We used this fraction to partition the measured memory cycles into two: those due to accesses to fast memory and those to slow memory. We then scaled the slow memory cycles by the ratio of the slow to fast memory latency (Shown in the bottom part of Figure 7). We assumed the slow memory is 3× slower but we also vary this for sensitivity studies in Section 6.1.

Further, the original leading-load model did not account for the overheads of migration. We measured TLB shootdown overheads in the kernel, and this remains unaltered with a larger number of accesses to the slow memory [5]. We also measured the time needed to copy the data between the fast and slow memory. The data copy latency is scaled in proportion to the ratio of the fast to slow memory latency. These two are shown in the green and yellow portion (rightmost two partitions) in Figure 7. Finally, our workloads (described next) are multi-threaded. We apply the above-mentioned model for each application thread that is bound to a core using Linux® taskset utility. The projected application runtime is then maximum of the projected runtime of each of its thread.

Baseline: We implemented an OS-directed baseline migration scheme on existing hardware by modifying Linux. Like most previous work [14, 22], our baseline migrates frequently accessed pages from the slow to fast memory. Cold (less frequently accessed) pages are migrated from the fast to slow memory to make space.

[4]These stalling instructions are generally referred as *leading load*.
[5]We assume the kernel memory resides in the fast memory zone.

We made the baseline as similar as possible to our granularity-aware migration scheme; except where our key innovations lie (i.e., altering the migration granularity and scanning in virtual memory). This provides a fair estimation of benefits from our innovations.

Similar to our dynamic scheme, two independent OS threads perform the periodic scanning in the fast and slow memory for finding the access frequencies to pages. However, the scanning operation in the baseline examines the access bits in PTEs of a batch of *physically* contiguous pages. This is different from our dynamic scheme where *virtually* contiguous pages are scanned. We set the scanning frequency to 100μs across all experiments. During a scanning pass in the slow memory, any page with its access bit set is selected as a candidate for migration to the fast memory. In contrast, for the fast memory, any page with its access bit unset is selected as a candidate to be migrated back to the slow memory to make space (if needed). Most importantly, the baseline always migrates 4KB of memory (default page size) at a time.

Workloads: We evaluated our proposal using a wide range of workloads drawn from Parsec [4], NAS Parallel benchmarks [3], HPC Challenge [33], Mantevo [23], CORAL [1, 51], and graph500 [38]. Table 2 lists the individual applications used from these benchmark suites. The table also lists the working/resident set size (RSS) of each application as reported by Linux. This will help the reader to understand the memory footprint of each application in relation to the emulated fast memory capacity. All applications are multi-threaded and made to use at least 8 threads. Workloads from Parsec use pthread while all others use OpenMP for multi-threading.

6 RESULTS

We evaluate our proposal through a quantitative analysis to following questions. ① How does a migration granularity affect the efficacy of page migration? ② How does the proposed dynamic scheme for finding the preferred granularity, perform? ③ What are the sources of improvements (degradations)? ④ How sensitive are the results to different fast memory capacity and access latencies?

Performance: Figure 8 shows the normalized execution time (lower is better) of the baseline, migration scheme under three different statically fixed migration granularities and our dynamic scheme. The height of each bar is normalized to the execution time of the application running on a system with the fast memory capacity exceeding the application's memory footprint, and thus, incurring *zero migrations*. Each application has five bars – the first bar is the baseline as described in previous section. The next three bars

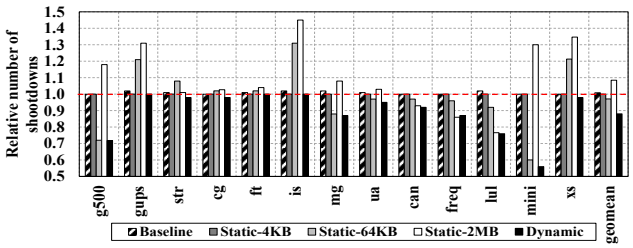


Figure 9: Relative number of shootdowns (lower is better).

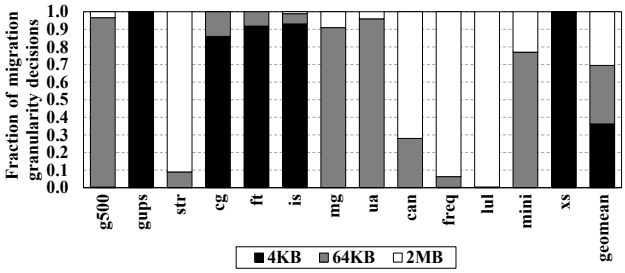


Figure 11: Breakdown of dynamic migration granularity alterations.

are for static migration granularities where the scheme works as described in Section 4 except that the migration granularity is fixed at 4KB, 64KB, and 2MB, respectively. The last bar represents our dynamic scheme that alters the migration granularity at runtime.

First, from the first two bars we observe that the baseline performs close to or slightly less than the static migration granularity of 4KB. The only difference between these two is that the `4KB-static` scheme is able to exploit application's spatial access locality by scanning *virtually* contiguous pages, instead of scanning the physical memory. Also, among static migration granularities, we observe that five out of thirteen applications studied (`gups, cg, ft, is,` and `xsbench`) perform best with 4KB. Four applications (`graph500, mg, ua,` and `minife`) prefer the 64KB static granularity while the rest (`stream, can, luesh,` and `freqmine`) prefer 2MB granularity. This demonstrates the preferred granularity of migration varies across applications.

Most importantly, we find that our dynamic scheme yields the runtime close to or even slightly better than the best performing static migration granularity across the board. The dynamic scheme was able to achieve this by altering the migration granularity for an application at runtime based on application's execution phases. Specifically, the dynamic scheme performs the best for eight applications (`stream, cg, mg, ua, canneal, lulesh, miniapp` and `xsbench`), and close to the best static granularity for the rest.

Analysis: Two metrics capture the efficacy of any OS-directed migration scheme –(1) the number of TLB shootdowns, and (2) the fast memory hit ratio While the former is an indicator of the migration overheads, the later captures the usefulness of migration.

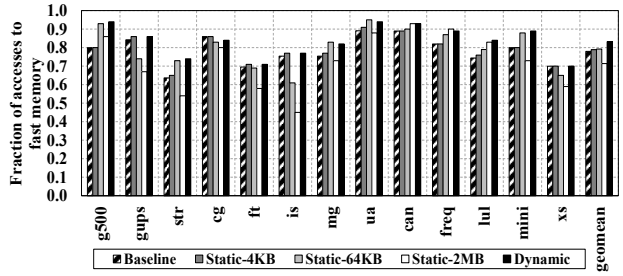


Figure 10: Fast memory hit ratio (higher is better).

Figure 9 shows the number of TLB shootdowns for the baseline, three static, and the dynamic scheme. Similar to the previous figure, the first bar is for baseline, next three bars are for static granularities, and the last one is for the dynamic scheme. The height of each bar is normalized to the number of shootdowns with static 4KB granularity. First, the baseline and the static-4KB scheme has a similar number of shootdowns, as expected. We observe that the applications that performed the best with 4KB granularity (in Figure 8) also, have the lowest number of TLB shootdowns with 4KB. The same is also true for the other two migration static granularities. Importantly, the number of shootdowns in our dynamic scheme remains close to that in the best-performing static granularity.

Figure 10 shows the fast-memory hit ratio (FM-hit ratio) measured by processor's IBS feature. Here also, we observe that the dynamic scheme yields the FM-hit ratio very close to the best performing static migration granularity for each application. These two metrics clearly show (1) why the migration granularity makes a significant impact on the performance of a migration scheme and (2) how the proposed dynamic scheme greatly improves the efficacy of migration by bringing *useful* data into the fast memory.

We further analyze the proposed dynamic scheme by measuring the breakdown of migrations across three different possible granularities in Figure 11. Each application has a stacked bar, which shows the fraction of migrations with 4KB, 64KB, and 2MB granularity. We observe that the dynamic scheme is able to pick the migration granularity that generally matches the best-performing static migration granularity for an application. For example, `graph500` is known to prefer 64KB migration granularity, and the breakdown in Figure 11 shows that the dynamic scheme employed 64KB granularity in more than 95% of migrations. In Figure 8, we saw that the dynamic scheme performed even better than the best performing static granularity for `cg`. In Figure 11, we observe that the dynamic scheme employs the granularity of 4KB for around 85% of migrations while using 64KB for the rest. We found that `cg` is composed of computation involving sparse matrix and data communications. In the computation phase, there is little locality, and thus, 4KB is preferable. For the communication phase though, 64KB helps as it benefits from relatively more locality. Adapting the granularity across phases benefits the overall performance and can sometimes outperform static granularity schemes. We observe the similar behavior in case of application `ua` too.

6.1 Sensitivity studies

In this section, we study the sensitivity of our dynamic scheme.

Capacity sensitivity: Hitherto, all results are performed with the fast memory capacity set to 400MB. We choose this size such that every applications' working set substantially exceeds the fast memory capacity (the working set size in Table 2). Otherwise, there is not enough number of migrations to study. However, to understand the robustness of our proposal, we evaluated both the static and

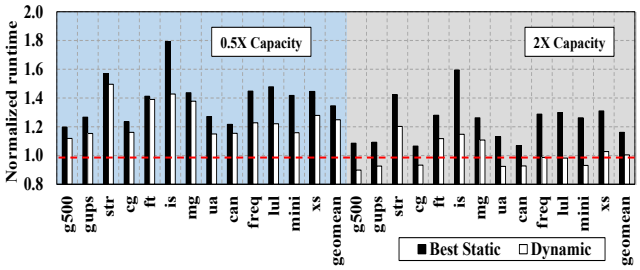


Figure 12: Execution time of varying capacity normalized to execution time with no migration (lower is better).

dynamic schemes with the fast memory capacity of 200MB (0.5×) and 800MB (2×).

Figure 12 shows the normalized execution times with different fast memory capacity. As earlier, the height of each bar is normalized to the execution time with zero migrations. The left half of the figure shows measurements with 200MB fast memory capacity and the right half shows that with 800MB capacity. For each application, there are two bars as in Figure 8. The left bar for each application represents the best performing static granularity for that application (it can be 4KB, 64KB or 2MB depending upon the application). The right bar represents the dynamic scheme. We

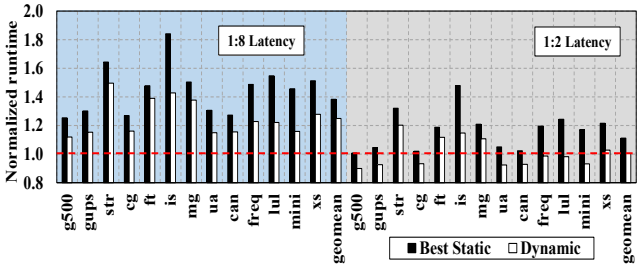


Figure 13: Execution time of varying latency normalized to execution time with no migration (lower is better).

Latency sensitivity Hitherto, our runtime projections assumed the ratio of the fast to slow memory access latency to be 1:3. Here, we vary this ratio to 1:2 and to 1:8. The left half of Figure 13 shows the normalized execution time of each application with the access latency ratio of the fast to slow memory being 1:8 and the right half shows the same when the ratio is 1:2. Regardless of the specific latency ratio, we observe that the dynamic scheme performs close to or better than the best performing static granularity. Intuitively, this result is expected as our dynamic scheme depends on application's memory access patterns.

Summary: We quantitatively demonstrate that (1) the dynamic scheme was able to adapt the migration granularity at runtime which resulted in achieving the performance that is close to or even better than the best-performing static migration granularity across thirteen workloads, and (2) the same conclusion holds across varying fast memory capacity and latencies.

7 DISCUSSION

Here, we discuss a few subtle but related topics.

7.1 Large pages vs Migration granularity

We define the migration granularity as a contiguous chunk of *virtual memory*. Unlike OS pages, the migration granularity does not necessarily map to a contiguous chunk of physical memory. For example, a 2MB migration granularity represents a contiguous 2MB chunk in the virtual address of an application, but could be mapped to 512 non-contiguous 4KB physical page frames.

We use the migration granularity instead of large pages for two important reasons. (1) Large pages are few and far in between. For example, x86-64 supports only 4KB, 2MB, and 1GB page sizes. If large pages were used instead, then it would have unnecessarily limited the choices of granularities. (2) The purpose of large pages is to reduce TLB miss overheads, and this is orthogonal to migration. A single 2MB TLB entry can map to 512 times more memory than a 4KB TLB entry. If we used large pages, then our scheme would have benefited more from fewer TLB misses, and also, it would have muddled our measurements. However, the conclusion about the importance of the granularity of migration holds even if large pages are used as a larger granularity.

7.2 Applicability to other migration policies

In this work, we chose a page migration scheme that is inspired by Linux's clock algorithm that utilizes the access bits in PTEs. However, our contribution of demonstrating the importance of migration granularity is not limited to any specific migration scheme. For example, our proposed dynamic granularity aware scheme can be easily extended to the next touch algorithm [11], which migrates a single 4KB page when the page is accessed more than twice in the slow memory. Here, instead of migrating 4KB upon a second access, a larger migration granularity can be chosen when high spatial locality is detected. Likewise, we can extend our proposal to the work by Meswani et al. [35], but the work requires additional per-page access counters, which are not available in current hardware. Oskin et al. [39] proposed to incur page faults on every access to the slow memory. A modified page fault handler then brings the data to the fast memory. Our granularity-aware migration scheme can be implemented in this page fault handler.

7.3 Heterogeneous memory system vs NUMA

The heterogeneous memory systems may look similar to conventional NUMA systems in that different parts of the physical memory have different access latencies. In a typical NUMA system, the data placed in *local node's* memory can be accessed faster by *local* threads running in the same node. Accessing the same data from *remote* threads running in another node takes longer. In a heterogeneous memory system, however, there are no local or remote threads. A given memory is equally distant from *all* threads. Therefore, a page migration in a heterogeneous memory system affects all threads *equally*, which is different from the NUMA system. Consequently,

many NUMA-specific optimizations (e.g., avoiding false sharing) do not apply to heterogeneous memory systems and vice-versa.

8 RELATED WORK

Data Management in Heterogeneous Memory Systems: A significant amount of previous work explored managing the fast memory entirely in hardware [6, 13, 18, 21, 25, 31, 37, 41, 44, 46, 52, 54]. Alloy Cache [41] migrates data at 64B cacheline granularity for fine-grain control. However, they do not exploit the spatial locality beyond 64B, and thus, a few proposals [16, 18, 43] keep access histories to migrate previously accessed lines together with a demanded cacheline, if they fall in a 2KB physical memory region. They rely on application's repetitive access patterns to achieve higher performance. Different from above proposals, we do *not require any hardware modifications*. Our scheme, however, bears similarities to the access pattern detection in the above-mentioned schemes. However, different from them, our proposal migrates a contiguous chunk of *virtual memory* and is not limited to migrating only cachelines that were accessed temporarily close to each other. Consequently, we improve performance even in workloads with low temporal locality, such as streaming.

Previous software managed proposal focused on detecting what to migrate to the fast memory. Meswani et al. [35] introduced per-page access counters to migrate pages whose counts are higher than a threshold at regular time intervals. The next-touch algorithm proposed by Goglin et al. [11] and the one proposed by Oskin et al. (our baseline) [39] use the demand request to trigger a migration. Kannan et al. [22] showed that under a virtualized environment, it is important for the guest-OS to be aware of memory heterogeneity to identify the "right" memory to migrate. Our proposal is orthogonal to these proposals; all these schemes can benefit from varying the migration granularity explored in this work.

NUMA Policies: Determining what data to migrate has been actively explored [7, 9, 10, 24, 26] in the context of NUMA. Lepers et al. [26] monitored the available bandwidth at each node and use it to trigger the migration of all pages of a process to a node with higher available bandwidth. Gaud et al. [10] use access counters to collocate and/or replicate pages to the local node. Recent work [9] showed that using larger pages can hurt performance in NUMA systems due to increased contention for bandwidth and false sharing. Our proposal is orthogonal to prior proposals in two aspects. First, the system of interests in our paper is the heterogeneous memory system and not NUMA (Refer to Section 7.3). Thus, many observations are not applicable; e.g., false sharing across local and remote threads do not occur in our context. Second, we demonstrated the importance of varying the migration granularity rather than on page placement.

User Guided Migration: Other prior work [5, 28, 36] explored the application/user feedback to guide the page migrations. Lin et al. [28] proposed asynchronous migration triggered by user request. Similarly, Meswani et al. [36] explicitly managed the fast memory under application's direction while Cantalupo et al. [5] proposed an explicit user level heap manager. Different from these proposals, ours does *not* require application modifications. Importantly, unlike any of prior work, we have explored the migration granularity.

TLB Shootdown Optimizations: Recent work [2] reduced the overheads of a single TLB shootdown, but this still does not solve the problem of invoking multiple shootdowns upon a large page migration. Some linux patches [29, 30] enable batching multiple TLB shootdowns, yet their batching is limited to physical address space. Our TLB shootdown mechanism exploits the spatial locality of virtually contiguous pages, which are very closely related to application behaviors.

9 CONCLUSION

We demonstrated that the OS-directed migration scheme in heterogeneous memory systems must take another dimension into account, *migration granularity*. We proposed a dynamic granularity aware migration schemes that can detect spatial locality and dynamically change the migration granularity. The scheme achieves an overall system performance improvement up to 36% over the baseline scheme that uses a single migration granularity.

10 ACKNOWLEDGEMENT

This research was (some of the authors were) supported in part by National Science Foundation under grants 1745813 and 1725743. Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors. AMD, the AMD Arrow logo, Radeon® and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.
[2] Nadav Amit. 2017. Optimizing the TLB shootdown algorithm with page access tracking. In *Proc. USENIX Ann. Conf.* 27–39.
[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. DOI: http://dx.doi.org/10.1145/125826.125925
[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.
[5] Christopher Cantalupo, Vishwanath Venkatesan, Jeff R Hammond, K Czurylo, and S Hammond. 2015. User extensible heap manager for heterogeneous memory platforms and mixed memory policies. *Architecture document* (2015).
[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] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS-International Conference on Architectural Support for Programming Languages and Operating Systems*.
[8] P. Drongowski, Lei Yu, F. Swehosky, S. Suthikulpanit, and R. Richter. 2010. Incorporating Instruction-Based Sampling into AMD CodeAnalyst. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 119–120. DOI: http://dx.doi.org/10.1109/ISPASS.2010.5452049
[9] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 231–242.
[10] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of memory management on modern NUMA systems. *Commun. ACM* 58, 12 (2015), 59–66.

[11] Brice Goglin and Nathalie Furmento. 2009. Memory migration on next-touch. In *Linux Symposium*.

[12] Joseph Greathouse. 2017. AMD IBS toolkit. (2017). https://github.com/jlgreathouse/AMD_IBS_Toolkit

[13] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. 2014. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 38–50.

[14] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 79–92. DOI: http://dx.doi.org/10.1145/2731186.2731191

[15] Y Huai, M Pakala, F Albert, T Valet, and P Nguyen. 2005. Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions. *Appl. Phys. Lett.* 84, cond-mat/0504486 (2005), 3118–3120.

[16] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. 2016. Efficient footprint caching for Tagless DRAM Caches. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 237–248.

[17] JEDEC. 2015. JEDEC 235A: High Bandwidth Memory (HBM) DRAM. (2015).

[18] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 25–37.

[19] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM.

[20] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[21] Lizy Kurian John. 1996. VaWiRAM: a variable width random access memory module. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*. IEEE, 219–224.

[22] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *ISCA*.

[23] Sandia National Laboratories. 2017. Improving Performance via Mini-applications. (Aug. 2017). https://mantevo.org

[24] J Laudon and D Lenoski. 1997. The SGI Origin: A ccnuma Highly Scalable Server. In *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*. IEEE, 241–251.

[25] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. 2015. A fully associative, tagless DRAM cache. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 211–222.

[26] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and memory placement on NUMA systems: asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 277–289.

[27] Zhongqi Li, Ruijin Zhou, and Tao Li. 2013. Exploring high-performance and energy proportional interface for phase change memory systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 210–221.

[28] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards programming heterogeneous memory asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 369–383.

[29] Linux Mailing List. 2015. Batch TLB flushing when unmapping pages for migration. (2015). https://lkml.org/lkml/2015/4/15/184

[30] Linux Mailing List. 2015. TLB flush multiple pages per IPI. (2015). https://lkml.org/lkml/2015/7/6/438

[31] Gabriel H Loh and Mark D Hill. 2012. Supporting very large dram caches with compound-access scheduling and missmap. *IEEE Micro* 32, 3 (2012), 70–78.

[32] Gabriel H Loh, Nuwan Jayasena, K McGrath, M O'Connor, S Reinhardt, and J Chung. 2012. Challenges in heterogeneous die-stacked and off-chip memory systems. In *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*.

[33] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Citeseer, 213.

[34] Joe Macri. 2015. AMD's next generation GPU and high bandwidth memory architecture: FURY. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–26.

[35] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 126–136.

[36] Mitesh R Meswani, Gabriel H Loh, Sergey Blagodurov, David Roberts, John Slice, and Mike Ignatowski. 2014. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*. IEEE, 9–16.

[37] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Comput. Archit. Lett.* 11, 2 (July 2012).

[38] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray UserâĂŹs Group (CUG)* (2010).

[39] Mark Oskin and Gabriel H Loh. 2015. A Software-managed Approach to Die-stacked DRAM. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 188–200.

[40] J Thomas Pawlowski. 2011. Hybrid memory cube: breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem. In *Proceedings of the 23rd Hot Chips Symposium*.

[41] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[42] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *In International Symposium on Computer Architecture*.

[43] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95.

[44] 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.

[45] A. Sembrant, D. Black-Schaffer, and E. Hagersten. 2012. Phase behavior in serial and parallel applications. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. 47–58.

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

[47] Jaewoong Sim, Jaekyu Lee, Moinuddin K Qureshi, and Hyesoon Kim. 2012. FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 321–332.

[48] Avinash Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–24.

[49] Bo Su, Joseph L. Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. 2014. Implementing a Leading Loads Performance Predictor on Commodity Processors. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association.

[50] Said Tehrani, JM Slaughter, E Chen, M Durlam, J Shi, and M DeHerren. 1999. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics* 35, 5 (1999), 2814–2819.

[51] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*.

[52] Ahsen J. Uppal and Mitesh R. Meswani. 2015. Towards Workload-Aware Page Cache Replacement Policies for Hybrid Memories. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 206–219. DOI: http://dx.doi.org/10.1145/2818950.2818978

[53] 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.

[54] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu. 2012. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 337–344.

[55] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *In International Symposium on Computer Architecture*.