Nimble Page Management for Tiered Memory Systems

Zi Yan
Rutgers University & NVIDIA
ziy@nvidia.com

Daniel Lustig
NVIDIA
dlustig@nvidia.com

David Nellans
NVIDIA
dnellans@nvidia.com

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

Abstract

Software-controlled heterogeneous memory systems have the potential to increase the performance and cost efficiency of computing systems. However they can only deliver on this promise if supported by efficient page management policies and mechanisms within the operating system (OS). Current OS implementations do not support efficient tiering of data between heterogeneous memories. Instead, they rely on expensive offlining of memory or swapping data to disk as a means of profiling and migrating hot or cold data between memory nodes. They also leave numerous optimizations on the table; for example, multi-threaded hardware is not leveraged to maximize page migration throughput, resulting in up to 95% under-utilization of available memory bandwidth.

To remedy these shortcomings, we propose and implement a general purpose OS-integrated multi-level memory management system that reuses current OS page tracking structures to tier pages directly between memories with no additional monitoring overhead. We augment this system with four additional optimizations: native support for transparent huge page migration, multi-threaded migration of a page, concurrent migration of multiple pages, and symmetric exchange of pages. Combined, these optimizations dramatically reduce kernel software overheads and improve raw page migration throughput over 15×. Implemented in Linux and evaluated on x86, Power, and ARM64 systems, our OS support for heterogeneous memories improves application performance 40% over baseline Linux for a suite of real-world memory-intensive workloads utilizing a multi-level disaggregated memory system.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304024

CCS Concepts • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Virtual memory**.

Keywords Page migration; Operating system; Heterogeneous memory management

ACM Reference Format:
Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304024

1 Introduction

Modern computing systems are embracing heterogeneity in their processing and memory systems. Processors are specializing to improve performance and/or energy efficiency, with CPUs, GPUs, and accelerators pushing the boundaries of instruction and data level parallelism. Memory systems are combining the best properties of emerging technologies that may be optimized for latency, bandwidth, capacity, or cost. For example, Intel's Knight's Landing uses a form of high bandwidth memory called multi-channel DRAM (MCDRAM) alongside DDR4 memory to achieve both high bandwidth and high capacity [27, 28]. Non-volatile 3D XPoint memory has been commercialized for next-generation database systems, and disaggregated memory may be a promising solution to capacity scaling for blade servers [41, 50]. Both CPUs and GPUs are embracing heterogeneous memory with IBM and NVIDIA having recently delivered supercomputers containing high-bandwidth GPU memories and high-capacity CPU memories [29, 38, 57, 58, 66, 67].

Figure 1 illustrates an abstract example of the memory systems architects and OS designers will likely have to consider in the future. These systems consist of a compute node (CPU, GPU, or both) connected to multiple types of memory with varying latency, bandwidth, and/or capacity properties. Of course, the particular configuration will vary by system.

The critical operating system support needed to enable the vision of efficiently moving data as programs navigate different phases of execution, each with potentially distinct

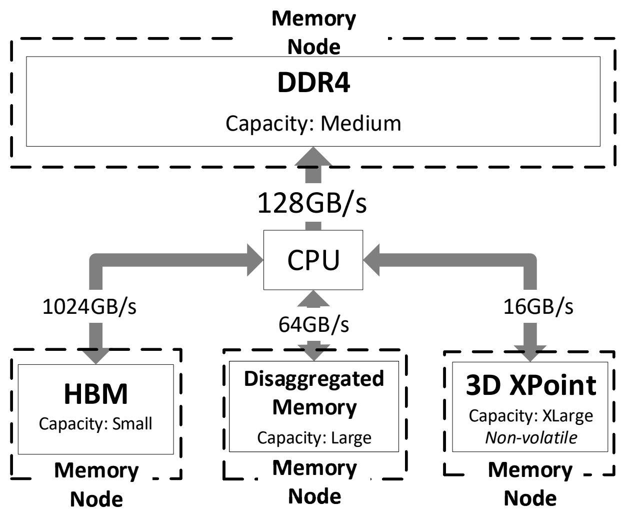


Figure 1. A hypothetical future multi-memory system with 4 technology nodes, all exposed as non-uniform memory nodes to the operating system.

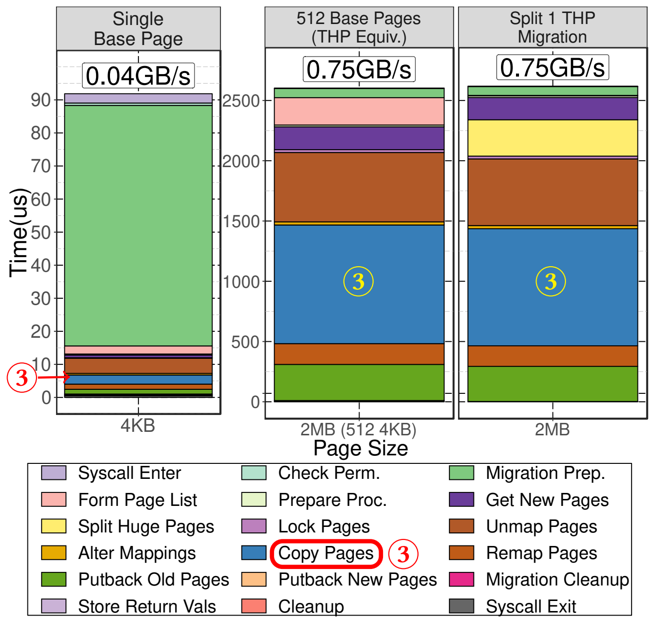


Figure 2. Page migration cost breakdown for migrating a single 4KB page, 512 consecutive base pages, and both splitting and migrating a 2MB THP. (Figure best viewed in color.)

working sets, is *efficient page management and migration*. Regardless of configuration, to optimize for performance, ideally the hottest pages will be placed in the fastest memory node (in terms of latency or bandwidth) until that node is full, the next-hottest pages will be filled into the second-fastest node up to its capacity, and so on. Then as programs execute, these pages must be constantly re-organized based on their hotness to retain maximum workload performance.

Unfortunately, page migration in today's systems has high overheads and is surprisingly inefficient. Past work and recent proposals mainly focus on reducing the overheads from the hardware side. Some increase TLB coverage to amortize the TLB miss penalty for migrated data [6, 13, 17, 19, 25, 33, 60–62, 65, 74] and others mitigate the performance ramifications caused by the correctness enforcement mechanisms (e.g., TLB coherence) of page migration [3, 5, 34, 71, 80]. We performed an experiment in which we moved pages between two memory nodes and where each local node has more memory bandwidth available than the inter-socket interconnect. After allocating memory from distinct memory nodes, we measured both the cost breakdown and the throughput of several types of cross-socket page migration. Figure 2 shows the cost breakdown and throughput achieved when migrating different page sizes on Linux today. For single base page migration, the majority of time is spent within kernel memory management and synchronization routines. Only a small fraction of time is consumed by the actual page copy. As a result, the effective migration throughput is just 40MB/s even though the hardware that has 19.2GB/s cross-socket memory bandwidth (see Table 1 for our experimental platform configuration). We also scaled the number of pages being migrated to 512, matching the huge page size (2MB), to amortize the software overhead across multiple migrations. In this case, throughput does scale up to 750MB/s, but this is still just 5% of peak hardware bandwidth. To investigate the potential improvement of page migration, we profile the data copy throughput of our system (described in Section 4.1) with different thread counts and data sizes including 2MB. Figure 3 shows that existing page migration throughput is 10× slower than what is achievable with a 2MB data size and the gap is bigger with larger transferred data sizes.

To eliminate the page migration bottleneck, we propose a set of four optimizations: *transparent huge page migration*, *parallelized data copy*, *concurrent multi-page migration*, and *symmetric exchange of pages*. On top of these mechanisms we build a holistic multi-level memory solution that directly moves data between heterogeneous memories using the existing OS active/inactive page lists, eliminating another major source of software overhead in current systems. The novel contributions of our work are as follows:

1. We show that breaking transparent huge pages (THPs), which are currently non-movable, into movable base pages reduces the effectiveness of THPs. We also demonstrate that existing base page migration only achieves throughput an order of magnitude lower than hardware line rate. We remedy this by implementing native huge page (THP) migration which improves migration throughput by 2.8× over the Linux baseline, while also having the side effect of improving TLB coverage.
2. Our additional page migration optimizations improve throughput by 5.2× over our native THP migration

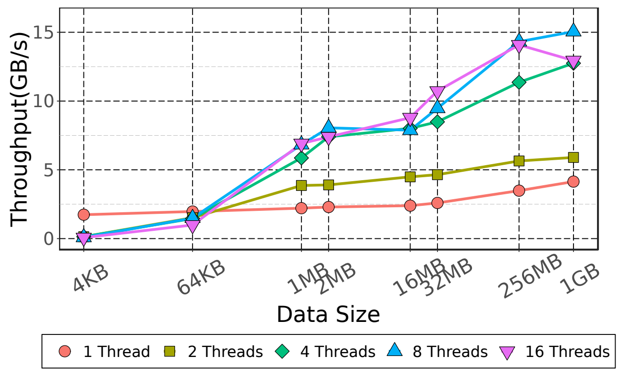


Figure 3. Impact of thread count and transfer size on raw data copy throughput (**higher is better**).

implementation alone. Together, they increase page migration throughput 15× over the state of the art today. By re-using existing OS interfaces, these optimizations are automatically inherited by any memory management policy using the standard Linux memory management APIs. Our claim of broad applicability is supported by the fact that some of our techniques have been adopted into mainstream Linux [53]. We have also publicly released the experimental kernel we implemented all our optimizations in and used for our evaluation[1].

3. We explore a simple end-to-end heterogeneous memory profiling and placement policy. Unlike existing implementations and other recent proposals, our system does not swap data out to disk, nor does it make portions of memory unavailable to profile accesses to them via page faults. Instead, it simply repurposes the existing OS active/inactive page lists. Therefore, our approach imposes no profiling overhead on systems which may not need its functionality.
4. We show that in a disaggregated memory system, an emerging class of important multi-tier memory system [22, 41, 42], our optimized OS support will improve application performance over 40%, on average, compared to current OS support for multi-level memories.

2 Background

Modern heterogeneous memory systems typically consist of low-capacity, high-bandwidth memory as well as high-capacity, low-bandwidth memory. Latency to large capacity memories is also expected to be higher due to longer, potentially multi-hop physical connections or differences in the underlying memory technology. Consequently, heterogeneous memory systems will often have non-uniform memory access (NUMA) properties for both latency and bandwidth. To ensure optimal performance, application developers will rely on both initial page placement policies and follow-up

[1]https://github.com/ysarch-lab/nimble_page_management_asplos_2019

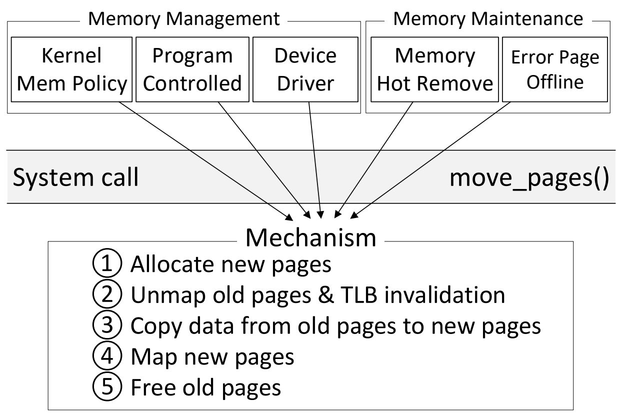


Figure 4. Separation of page migration policy and page migration mechanism in a multi-level memory system.

page migration policies to ensure that hot data pages remain within the highest bandwidth or lowest latency memory node during an application's runtime [12, 32, 56].

2.1 Page Management Policies and Mechanisms

Modern OSes are already NUMA-aware [11, 14, 37]. In fact, standards are already being introduced that allow heterogeneous memory properties (latency, bandwidth, durability) to be exposed to the operating system so that page location decisions can be optimized to maximize throughput [79, 83]. To explain holistic OS support for multi-level memory we must conceptually separate this memory management into two distinct components. Figure 4 shows that memory management *policy* decisions may be driven by the kernel, device drivers, programmer, or a system administrator. These policies are all built on top of a common *mechanism*, page migration, which performs the desired OS page movement operations. Policy decisions are thus separated from page migration mechanisms through a system call (or analogous kernel interface), such as `move_pages()` on Linux.

Figure 4 also shows that a generic page migration mechanism involves ① allocating a new page, ② unmapping the existing virtual to physical address translation (and, on many architectures, issuing a TLB shootdown), ③ copying data from the old physical page into the new physical page, ④ mapping the virtual address to the new physical page, and finally ⑤ freeing the old physical page. The actual copy between the old and new physical page occurs only in step ③. Steps ①–② and ④–⑤ are overheads needed to ensure both correctness and protection guarantees so that neither the old page nor the new page is accessed during the page migration process. This work investigates the natural evolution of this multi-step process to leverage modern hardware and improve migration throughput.

Today, Linux uses autoNUMA [12] to try and fairly balance memory and compute requirements between NUMA nodes. It relies on two basic techniques to do this: process migration and page migration. Unfortunately, process migration is not applicable to to heterogeneous memory systems, where multiple memory nodes are all connected to a single processor. Page migration is also currently limited in multi-level memory systems because autoNUMA can only migrate pages to a memory node with free space; otherwise, pages are swapped out to disk. This is at odds with the system's goal of placing as many pages in a (small) fast memory as possible. Additionally, to obtain page access information autoNUMA offlines pages for profiling, and such offlining causes unpredictable memory access latency and bandwidth. These two issues have spurred a range of academic work on two-level memories; however, prior studies have generally focused on page migration policy while assuming page migration mechanisms should be sufficient[2, 9, 15, 16, 24, 39, 46, 59, 69, 82].

2.2 Recent Developments

Because heterogeneous memories are only now being adopted commercially, multi-level memory paging remains an active area of research. Researchers have begun exploring hardware techniques to identify hot/cold pages and facilitate their movement among memory devices [10, 23, 30, 44, 68, 75]. This work has spurred further studies on software-based approaches that are better able to manage heterogeneous memories with complex topologies [1, 2, 22, 32, 59, 82]. While these studies have established the appeal of OS-managed multi-level memories, they mostly rely on trace-based simulation (which excludes OS effects) [1, 48, 82], use non-standard OS plumbing (such as the use of reserved page table entry bits) [2], hoist page migration out of the OS (due to the unavailability of the source code or the prohibitively large engineering work) [8, 11, 54, 55, 72], and/or simply assume that the OS can deliver the full hardware bandwidth (which we demonstrate is not possible) during page migrations [1].

Page migration mechanisms have not been studied as extensively as policies, despite being critical to performance in multi-level memory systems [20]. Combined, these seemingly subtle issues may hinder real-world adoption of many prior proposals because their conclusions may ultimately be shown to be incomplete. Solutions must be generally maintainable to be adopted in practice [21, 47]. However, this growing body of work does point to the need for both better page migration mechanisms and policies, in addition to system level evaluations of what effect the OS will have on multi-level memory systems, both of which we now address.

3 Native OS Support for Multi-Level Memories

Holistic support for multi-level memory systems includes both intelligent memory management policies and efficient page migration mechanisms. In this section we present our page migration mechanism improvements, which are independent of any one particular policy, as well as one specific low-overhead policy that we use in Section 4 to demonstrate the end-to-end benefits of our memory management system.

3.1 Optimizing Page Migration Mechanisms

Four critical issues need to be addressed within the operating system to implement an efficient page migration mechanism:

Larger data sizes: With larger data sizes, the software overheads of page migration can be amortized away.

Multiple threads: Today, page migration is single-threaded primarily for the sake of simplicity, but using multiple threads would speed up the copy time itself.

Concurrent Migrations: Performing multiple migrations concurrently can help us avoid the Amdahl's Law bottleneck seen in Figure 2. This problem cannot be solved by simply using larger pages, since architectures only support a very limited set of page sizes, and in general the largest page sizes (e.g., 1GB on x86) are not supported transparently.

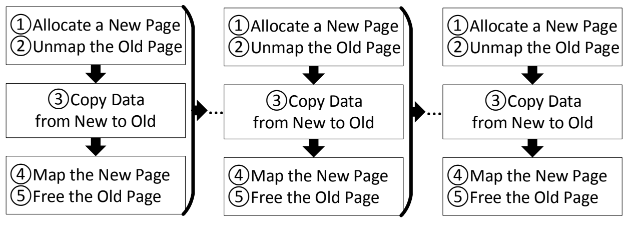
Efficient two-sided migration: When a page is migrated to fast memory, a victim page must be migrated to slow memory to make room. By eliding allocation/deallocation and simply exchanging pages, two-sided operations can be faster than the sum of two one-directional migrations.

We address each of the above issues in turn below.

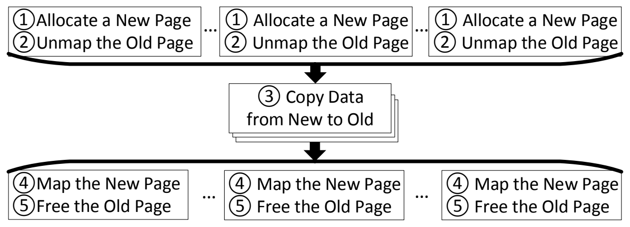
3.1.1 Native THP Migration

Our first optimization is to implement native THP migration. THP migration decreases both the hardware and software overhead of migrating THPs by a factor of 512, due to not splitting pages and reducing the number of required TLB invalidations and shootdowns. It also increases the amount of data migrated within a single migration operation. Although it may appear obvious, page migration support for THPs in Linux (and many OSes) is not yet mature, general, or high-performance. Page migration was originally proposed to enhance NUMA system performance and achieve memory hotplug functionally before THPs had even been introduced [4, 36].

Today, for example, Linux cannot migrate THPs in response to programmatic resource management requests like *mbind* and *move_pages*, which are designed to move data to specific memory nodes at the request of programmers. This is a serious shortcoming for heterogeneous memories since there are important situations where programmer-directed data placement enables good performance [59]. Similarly, Linux cannot directly migrate THPs in response to memory hot removal, soft off-lining, or *cpuset*/*cgroup*. In all these cases, when Linux is migrating a virtual memory range that contains transparent huge pages, it must split the THPs and migrate the constituent base pages instead, resulting in poor page migration performance and reduced TLB coverage.



(a) Linux Serialized Multi-Page Migration



(b) Proposed Concurrent Multi-Page Migration

Figure 5. Improvements to Linux multi-page migration to enable large transfers for improved copy bandwidth.

To implement THP migration, we augment the five steps shown in Figure 4 to be THP aware. Our THP implementation supports *all* resource management requests (i.e., *mbind*, *cpuset*, etc.) In addition, we adjust other THP-specific code paths in the kernel to account for the fact that THPs may be undergoing migration at the time of the call. We do this either by waiting for the end of the migration or by simply skipping the THPs (as is done for base pages). Other OSes can follow the same principle to realize THP migration to enable significantly higher-throughput page migration for a better support of heterogeneous memory systems.

3.1.2 Parallelized THP Migration

Currently, the Linux page migration routine is single-threaded and has a limited amount of data (usually the size of one base page) to transfer within a single migration. Motivated by Figure 3, we implemented a variable-thread count based copy subroutine within the page migration operation.

To enable multi-threaded page copies within the Linux `move_pages()` system call, we use kernel `workqueues` to spawn helper threads to copy data between arbitrary physical ranges. Our implementation calculates the amount of data to be copied via each parallel thread by dividing the page size (or, in Section 3.1.3, the aggregate of pages being migrated concurrently) by the number of worker threads.

Because the exact selection of thread location and thread count needed to maximize throughput is likely to differ among systems, we provide parameter configuration through the `sysfs` interface, so that system administrators can enable or disable multi-threaded CPU copy or change the number of CPUs involved in the data copy. We also augment the `move_pages()` system call with an optional parameter flag, `MPOL_MF_MT`, to enable migration policy engines to dynamically choose the level of parallelism on a per migration basis.

3.1.3 Concurrent Multi-page Migration

Multi-page migration is expected to be common in multi-level memory systems due to spatial locality and prefetching effects. The Linux `move_pages()` interface already supports migration of multiple pages with a single system call by passing in a list of pointers to the pages to be migrated between memory nodes. However, as shown in Figure 5a, the current implementation serializes the copies and performs them one page at a time.

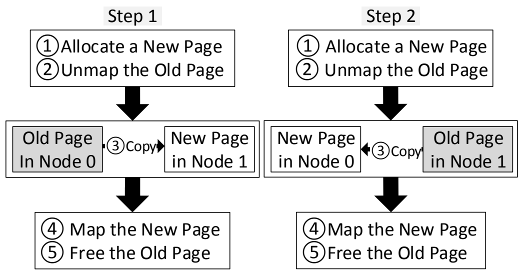
Our new page migration implementation concurrently migrates all pages in the list provided to `move_pages()` by aggregating all data copy procedures into a single larger logical step, as shown in Figure 5b. As an example, consider the case of migrating 16 THPs of 2MB. In the current Linux implementation, even using the parallel copy optimization, Linux will transfer 16 THPs of 2MB, with an implicit barrier between each parallel 2MB copy. In our concurrent migration optimization, each of the pages in the list is allocated and assigned a new page with matching size and then unmapped. Then, all pages in the list are distributed to the per-CPU `workqueues` according to the `sysfs` configuration.

If more parallel transfer threads are available than concurrent pages to migrate, our implementation uses multiple threads to copy different parts of a single page to maximize throughput. Once the concurrent page copy step is completed, the new pages are mapped onto the correct page table entries and the old pages are freed. It may also be possible to parallelize other steps of page migration. However, because there are strict correctness requirements for synchronization including architecture-dependent page table manipulation, failure recovery becomes complicated if optimizations become too aggressive.

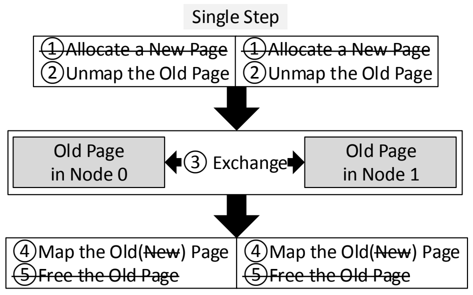
3.1.4 Symmetric Exchange of Pages

In multi-level memory systems, single-ended page migration is unlikely to be the common case. Higher-bandwidth memory is generally capacity-limited compared to larger lower-bandwidth memories. Therefore, when migrating pages into a higher-level memory node, at steady state, the page migration policy will need to migrate pages out of that node, so as to not exceed physical memory capacity. Therefore, when managing a high performance heterogeneous memory node as a software controlled cache, each hot-page insertion requires a symmetric cold-page eviction.

Naive two-step, one-way migration makes inefficient use of system hardware if the two copies are protected by locks and executed serially, as is the case in today's OSes. Figure 6a shows this common two-step page migration operation. First, the locking serialization limits the benefits of our previously discussed optimizations. Second, each migration operation must perform independent physical page



(a) Exchange Two Page Lists in Linux



(b) Proposed Exchange Page Migration

Figure 6. Exchanging pages improves efficiency by eliminating memory allocation and release when migrating symmetric page lists between memory nodes.

allocation and deallocation, and both are expensive software overheads (as shown in Figure 2).

To eliminate these overheads, we propose to combine the two one-way migration operations into a new single symmetric exchange operation. By exchanging pages, our implementation eliminates many of the kernel operations required in unidirectional page migration and reuses the existing physical pages instead of allocating new ones (as shown in Figure 6b).

We implement symmetric page migration in Linux by providing a new `exchange_pages()` system call that accepts two equal-sized lists of equal-sized pages. If the page lists do not meet the requirement, the caller must revert to a traditional two-step migration process. When called with two symmetric page lists, our exchange of pages implementation follows a similar path to unidirectional page migration. The differences are that that no new pages are allocated, and that instead of copying data into new pages, we transfer data between each pair of pages using copy thread(s) that use CPU registers as the temporary storage for in-flight iterative data exchange operations. This use of registers allows our mechanism to avoid allocating a complete temporary page.

Both our parallel page copy and concurrent page migration optimizations focus on improving the data copy itself while symmetric page exchange eliminates two of the most expensive software overheads that occur during page migration: page allocation and release. Because these kernel operations consume constant time irrespective of page size, page exchange improves the migration throughput of both base pages and transparent huge pages. Additionally, pairs of pages can also be exchanged using parallel exchange (Section 3.1.2), concurrent exchange (Section 3.1.3), or both, without extra locking as long as each parallel exchange thread operates in isolation.

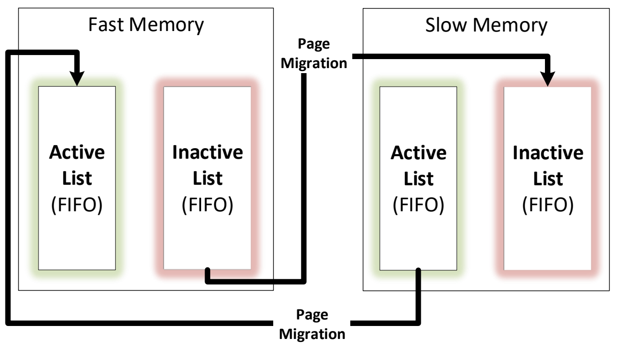
3.2 Optimizing Page Tracking and Policy Decisions

A multi-level memory paging policy and system needs to be sufficiently general and representative of real-world scenarios in order to be broadly useful on the diverse set of heterogeneous memory systems that are beginning to emerge. To this end, we explore a page migration policy that is simple, has been shown to work well in a wide range of environments [32], and adds negligible overhead to the baseline. As such, our implementation builds upon the existing Linux page replacement algorithm. We intentionally make as few changes as possible to keep our implementation maximally compatible with the upstream kernel.

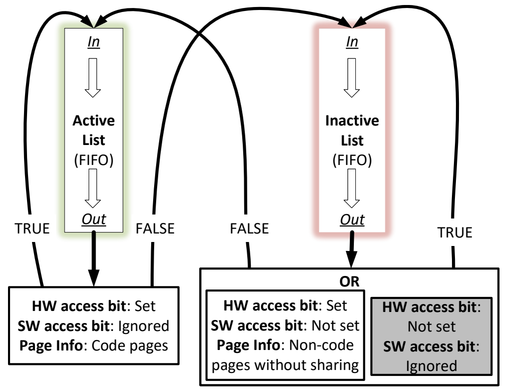
The goal of a page replacement algorithm is to identify hot and cold pages so that the page migration mechanism can migrate hot pages out of (historically) disk or (in our case) slow memory, and into fast memory. Linux already achieves this by separating hot and cold pages within each memory node into *active* and *inactive* lists, where a page can be in only one of these lists at a time. As hot pages become cold and vice versa, the kernel actively moves the pages between these lists as shown in Figure 7.

Like Linux's, our policy moves pages from one list to another by checking each page's state and two access bits, one in the page table entry pointing to the page and the other in the page metadata maintained by the kernel. We call the former the hardware access bit and the latter the software access bit. A page table entry's access bit is set by the hardware page table walkers on the first TLB miss to each virtual-to-physical translation corresponding to that page. Its software access bit is set by the existing Linux paging algorithm for each physical page. Both hardware and software access bits are inspected using the atomic operation `test_and_clear()`, except where marked "*Ignored*" in Figure 7, in which case the bit is neither checked nor cleared.

The key difference between our proposed paging policy and the standard Linux approach is graphically illustrated with the greyed box in Figure 7. In Linux today, cold pages that have not been accessed recently can be reclaimed (freed or paged to disk). However, heterogeneous memory systems aim to percolate such pages to the slower memory instead, to avoid the high cost of paging them back in from disk later. Consequently, our policy chooses to keep this page in the inactive list to make it a candidate for migration out of the fast memory. Similarly, if capacity is available in our fast memory (e.g., due to memory deallocations or migration of inactive pages), pages from the slow memory active list will



(a) Inter-memory-node page migration



(b) Intra-memory-node page list manipulation

Figure 7. Proposed native multi-level paging policy consisting of (a) inter-memory node page migration and (b) intra-memory-node page list manipulation. The former migrates hot pages (in the *active* list) from slow to fast memory and vice versa for cold pages (in the *inactive* list). The latter moves pages within a given memory node from one tracking list to the other.

be migrated into the fast memory. If the fast memory is full and does not contain inactive pages, no migration will occur.

Similar to traditional NUMA allocation policies, when new memory is allocated, it will occur in the fast memory if free space is available; otherwise, it will only occur in the slower memory node. We do not evict pages upon allocation so that we can keep page migration off the memory allocation path, which is performance critical. Finally, our system optimizes page locations only every 5 seconds throughout application runtime to minimize active process interference, based on profiling results.

4 Experimental Results

To quantify the utility of optimized page migration in a concrete scenario, we evaluate our approach on a disaggregated memory system due to the emerging importance of these systems in industry [22, 41].

4.1 Methodology

We emulate a disaggregated memory system using an experimental machine that has two memory nodes; we use one as fast local memory and the other as slow remote memory. We emulate slow memory by running one or more instances of memhog, an artificial memory-intensive workload that has been used in prior studies to load the system [59, 64]. The memhog instances, which run on otherwise-idle CPUs, inject extra memory traffic into the system. This has the effect of reducing remote memory bandwidth to one half of local memory bandwidth and increasing unloaded access latency to double that of local memory, which has been validated by Intel Memory Latency Checker [26]. Table 1 gives additional details of our setup.

To evaluate our OS optimizations, we integrate our proposed optimizations into Linux v4.14. The statistics of kernel modification given by `git diff` is: `23 files changed, 627 insertions(+), 114 deletions(-)`.

Intel Xeon Dual Socket System	
Processors	2-socket E5−2650v3
Memory	DDR4 — 2133MHz
Cross-socket QPI BW	19.2 GB/s
Memory BW	34.0 GB/s (per-socket)
Memory Latency	84.9 ns
OS & Kernel	Debian Buster — v4.14.0
Disaggr Mem BW (Emulated)	17.0 GB/s
Disaggr Mem Latency (Emulated)	199.2 ns

Table 1. Overview of experimental system.

For the evaluation itself, we perform a variety of experiments. First, we evaluate a set of microbenchmarks to measure the effect of each of our proposed optimizations in isolation and combination. Second, to get complete end-to-end performance numbers, we run workloads from SpecACCEL [31] and graph500 [52], and we show the performance across a range of fast memory oversubscription scenarios. Third, we sweep the design space to highlight the interesting behaviors that arise and to identify the configuration parameters that perform the best. Finally, we evaluate them on additional non-x86 architectures to prove the generality of our proposed enhancements.

4.2 Page Migration System Call Performance

To build intuition as to the sources of the overall performance improvements achieved by our page migration mechanisms, we first use microbenchmarks to tease out the relative benefits of the different (complementary) optimizations. Our experiments use the generic page migration interfaces, `move_pages()` (with our optimizations), and `exchange_pages()` (our newly proposed system call).

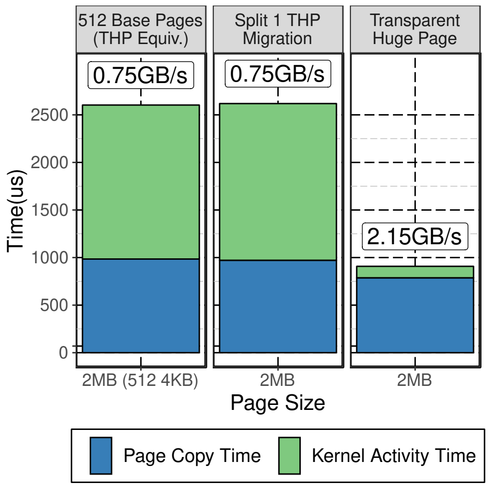


Figure 8. Cost breakdown (**lower is better**) of 512-base-page migration, THP-split migration, native THP migration.

4.2.1 Native THP Migration

Figure 8 contrasts the performance of a kernel that migrates pages in three ways. The leftmost bar is unmodified Linux migrating 512 4KB pages. The middle bar is Linux migrating a 2MB THP by splitting it into 4KB pages, which are then migrated. The right bar is our proposed native THP migration support.

Migrating a 2MB THP with splitting achieves virtually identical throughput as migrating 512 4KB pages. This is unsurprising, since they perform the same kernel operations for 512 pages plus one additional THP split. Our native THP migration improves throughput by 2.9× because it reduces kernel overhead by consolidating 512-page operations into a single page operation. Figure 8 also shows that page copy time decreases only marginally; i.e., there is still significant room for our additional optimizations to improve overall throughput.

4.2.2 Multi-threaded Transfers

Figure 9 shows the results of our second optimization, parallel (multi-threaded) page copying. We separate results for the cases where we migrate 2MB THPs (the graph on the top) and 4KB base pages (the graph on the bottom) also varying the number of threads used to perform the copies.

There are two primary observations from our multi-threaded copy results. First, parallel page copies are primarily beneficial when the page sizes are larger. For example, 2MB THP page migration time is sped up 2.8×, while parallel copies do not improve the throughput for 4KB pages, because the thread launch overhead cannot be amortized sufficiently. This overhead is likely the reason that the current Linux page migration has remained single-threaded regardless of base page sizes. Second, the graph at the top of Figure 9 shows that even though parallel migration is useful for 2MB

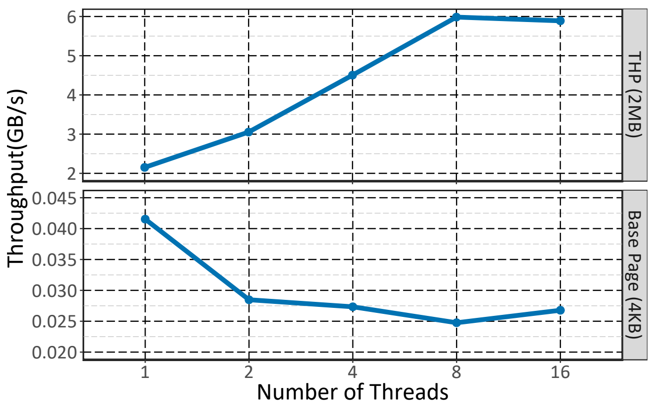


Figure 9. Throughput (**higher is better**) of multi-threaded single page migration for both base page (4KB) and THP (2MB).

THPs, overall throughput still remains well below the maximum cross-socket copy throughput of around 16 GB/s (see Figure 3), motivating the need for concurrent page migrations.

4.2.3 Concurrent Page Transfers

Figure 10 illustrates the performance advantage of using concurrent page migration optimization whenever possible. Results are again separated for 2MB THPs (the graph on the top) and 4KB base pages. For 4KB pages, parallel non-concurrent migration is always inferior to single threaded migration (due to the aforementioned parallelization overheads), and concurrent parallel migration only surpasses the baseline at sufficiently large page counts. As previously mentioned, the OS overheads are too large to overcome. However, utilizing both parallelism and concurrency are clear wins when transferring 2MB THPs, with a performance advantage (over parallelism alone) ranging from 10-25% depending on the number of pages transferred.

4.2.4 Symmetric Exchange Pages

Figure 11 shows the benefits of symmetric page exchange atop our prior optimizations. When using THPs, exchange page throughput follows similar trends as concurrent page migration, but with a performance improvement ranging from 10-50% depending on the number of pages exchanged. Interestingly, the largest fractional improvement is when exchanging small numbers of pages, because in these cases the software overhead remains a significant fraction of total transfer time. Removing the memory management overheads from the page migration process can improve the throughput of page migration to as high as 11.2GB/s when exchanging two lists of 512 2MB pages (1GB data on each list); this is very close to the best achievable copy throughput (excluding kernel overhead) of 11.7GB/s.

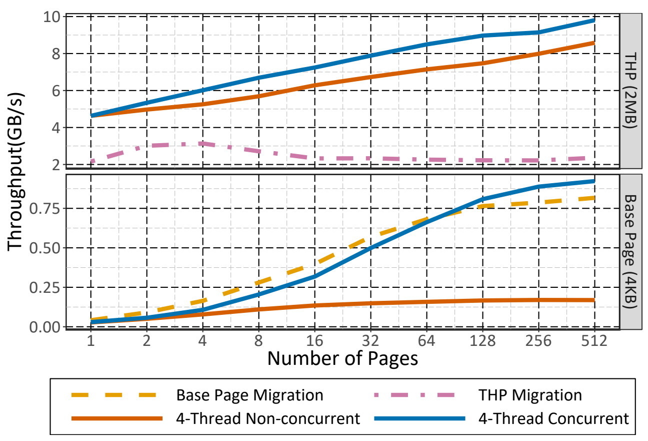


Figure 10. Throughput (**higher is better**) of concurrent page migration for both base page (4KB) and THP (2MB) with different numbers of pages under migration. 4-Thread Non-concurrent uses 4-thread data copy and 4-thread Concurrent adds concurrent page migration. Single-threaded Base Page Migration and THP Migration are shown for reference.

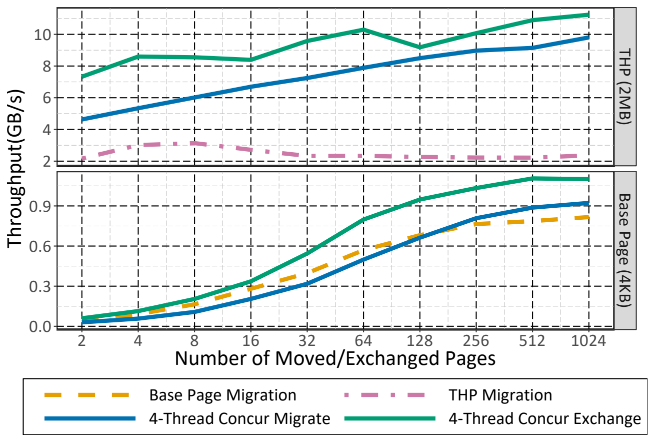


Figure 11. Throughput (**higher is better**) of page exchange vs. 2 page migrations for both base page (4KB) and THP (2MB) sizes while varying the number of pages exchanged. 4-Thread Concur Migrate and 4-Thread Concur Exchange use both concurrent and 4-thread parallel data copy. Single-threaded Base Page and THP Migration throughput are shown for reference.

Unlike our prior optimizations, when exchanging base pages (4KB) we also observe throughput improvements. When exchanging two lists of 512 4KB pages (2MB data on each list), we get 1.1GB/s throughput, or 37.5% more than the throughput of Linux's base page migration.

4.2.5 Microbenchmark Summary

When using only base pages and the three non-THP improvements (multi-threaded copy, concurrent copy, and two-way exchange), our system yields a 1.4× throughput improvement as compared to Linux's single threaded implementation. For THP migration, native THP migration alone (i.e., without our parallel/concurrent/exchange optimizations) delivers a 2.9× migration throughput improvement over Linux's state of the art. With our parallel copy and concurrent page migration optimizations added, we achieve a 4.6× throughput improvement over native-THP migration. With two-way exchange, we further improve throughput by 1.1×. The combined overall improvement for THP migration over the Linux baseline is 5.2× versus THP-splitting migration and 15× over base page only migration.

4.3 End-to-End Performance Results

Thus far, we have used microbenchmarks to quantify the performance benefits of our page mechanism optimizations. We now turn our attention to evaluating the end-to-end performance improvements that these optimizations, combined with our low overhead page management policy, can achieve. Our experimental testbed emulating a disaggregated memory is described in Section 4.1. No changes have been made to Linux's THP allocation policy and THPs are provided, when possible, on demand by the operating system.

We evaluate SpecACCEL and graph500, with the memory footprints scaled to 32GB. With these workloads running under our experimental setup, we find that over 90% of the pages for each workloads is typically backed by THPs, indicating that there is a significant negative impact of having to split THPs into base pages before migration.

We first run each workload in a disaggregated memory scenario that has 16GB local memory and 40GB remote memory. In this configuration, the local memory is only half the size of the workload memory footprint, while the remote memory can accommodate the entire workload footprint. We compare four different page migration mechanisms, along with an upper and lower bound for comparison:

1. **All Remote**, the lower bound, where workloads are run entirely from the 40GB remote memory
2. **Base Page Migration**, the Linux default (THPs are split before migration)
3. **Opt. Exchange Base Pages**, 4-threaded parallel copy and 512-page concurrent exchange (THPs are split before migration)
4. **THP Migration**, our native THP approach without parallel, concurrent, or exchange optimizations
5. **Opt. Exchange Pages**, THP migration, 4-threaded parallel copy and 8-page concurrent exchange
6. **All Local**, the upper bound, where workloads are run entirely from a 40GB fast local memory

In configurations 3 and 5, we use 4 threads for copying 512 and 8 pages respectively for our migration parameters. From our microbenchmark results, this presents the best configuration for both base and THP migrations. In Section 4.5 we present further sensitivity analysis to justify these selections.

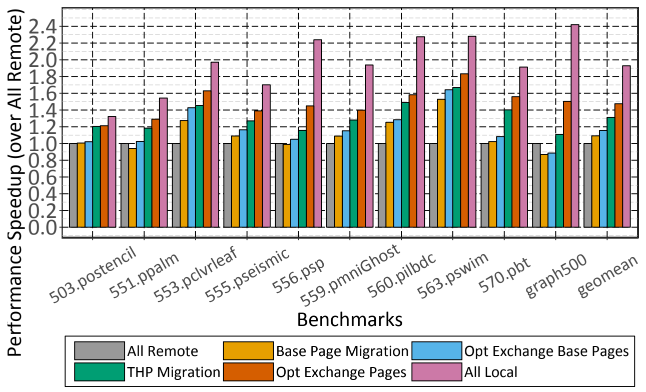


Figure 12. Benchmark runtime speedup (over All Remote, **higher is better**) with 16GB local memory. Base page migration and THP migration are single-threaded and serialized and shown for comparison, while Opt Exchange Base Pages uses 4-thread parallel and 512-page concurrent migration and Opt Exchange Pages use 4-thread parallel and 8-page concurrent migration.

Figure 12 shows the relative speedup of these six configurations over All Remote. All Local, which is our ideal case, on average achieves about 2× geomean speedup over All Remote, reflecting the local vs remote memory bandwidth and access latency difference in our system. Base Page Migration, which is our baseline for managing disaggregated memories, improves workload performance on average by 9%. However, some workloads (e.g., 551.ppalm, 556.psp, and graph500) perform worse than All Remote, meaning Base Page Migration is not always making good use of the 16GB local memory. Opt. Exchange Base Pages improves workload performance by 16% on average, and in this case, only graph500 does not take advantage of the 16GBs of local memory. Both of these results are testament to the notion that poor migration mechanisms can actually *degrade* performance despite the addition of a faster tier of memory.

Fortunately, enabling our native THP Migration can indeed harness the benefit of fast memory. Enabling THP migration improves the geomean workload performance by 31% over All Remote and achieving 68% of the geomean All Local performance, which is our ideal case. Our Opt. Exchange Pages (which can migrate both base pages and THPs) further improves average performance by 48% over All Remote and achieves 77% of the geomean ideal All Local performance.

4.4 Sensitivity to Local Memory Size

To further demonstrate the general applicability of our approach, we sweep the local memory size from 4GB to 28GB and show the geomean of all benchmark speedup over All Remote. Figure 13 shows that the performance trends are similar to those of the 16GB local memory case and we note four key observations First, Linux's Base Page Migration is not able to exploit the full potential of the disaggregated memories. When the local memory size (such as 4GB and 8GB) is much smaller than the workload's 32GB memory footprint, it degrades workload performance by 5% to 10% on average. Second, our Opt. Exchange Base Pages can help improve performance but is still limited by page migration throughput. It is however slightly better than Base Page Migration. Third, our THP Migration keeps improving performance; thus, it is able to make using disaggregated memory feasible without any performance loss. Fourth, our Opt. Exchange Pages improves performance substantially, exploiting most of the potential in the disaggregated memory system and outperform Linux's Base Page Migration 40% on average.

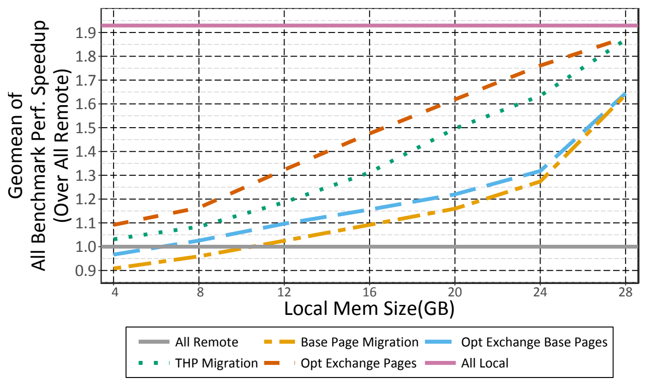


Figure 13. Geomean speedup (over All Remote, **higher is better**) over a sweep of local memory sizes, from 4GB to 28GB. Base page migration and THP migration are single-threaded and serialized, while Opt. Exchange Base Pages uses 4-thread parallel and 512-page concurrent migration, and Opt. Exchange Pages uses 4-thread parallel and 8-page concurrent migration.

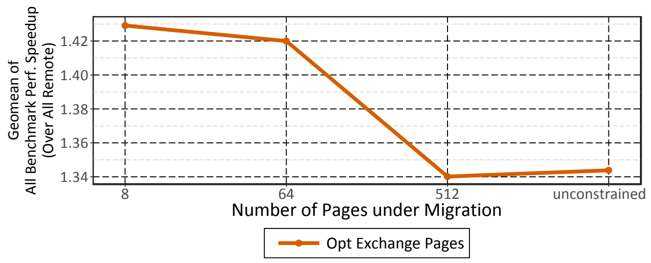


Figure 14. Geomean speedup (over All Remote, **higher is better**) given different numbers of pages under migration. Opt Exchange Pages use 4-thread parallel data copy. "*Unconstrained*" means we do not limit the number of pages under migration; instead we just migrate all pages at once. All use 16GB Local Memory.

4.5 Sensitivity to Tunable Parameters

Our page management system provides a number of user-tunable parameters. Here, we evaluate how sensitive the performance is to two of those parameters: the parallel copy thread count and the number of pages migrated concurrently.

4.5.1 Number of threads for parallel page migration

Because our system allows the number of threads used to copy data to vary, it is important to understand the importance of this tunable parameter. Using higher thread counts will improve copy throughput, but steals compute resources from the application itself, thus how to strike a balance between these factors is important to understand.

We swept the number of threads used by parallel page migration from 1 to 16. Using 4 copy threads was mostly the highest performing configuration, but that performance when using other thread counts had a maximum variation of just 4%, indicating that our proposed system is not overly sensitive to this parameter. Therefore, simply selecting a reasonable point (such as the 4 thread configuration we used in our end-to-end results) is a reasonable decision.

4.5.2 Number of pages being migrated concurrently

Similar to the variation possible in copy threads, we tested our system's sensitivity to the number of pages we migrate concurrently. As the number of migrated pages increases, so should throughput. However, beyond a certain point, migrating a high number of pages also can result in application stalls and decreased performance, because these pages are in-flight and unavailable to the user process.

Figure 14 shows the effect of varying the concurrent migration page count. We observe that limiting the number of concurrently migrated pages in the system is necessary, with 8 pages (32KB or 16MB of data respectively depending on page size) performing best. There is very little performance variation when using any value less than 64 pages, but performance does drop by almost 10% if the number is left unconstrained.

4.6 Generality Across Architectures

To explore the platform independence of our multi-level memory system and optimizations, we port them to other hardware platforms and compare the microbenchmarks used in Section 4.2. Our multi-level memory page tracking and policy implementation is architecture independent by design, because it is based on Linux's current architecture-independent active and inactive page list implementation. Thus we focus on platform sensitivity for our migration optimizations. We show results in Table 2 using optimal tunable parameters, as the best configuration for each platform varies due to available memory bandwidth and CPU performance.

While all platforms benefit from our optimizations, using Intel Xeons, we get a 2.9× improvement in migration bandwidth using native THP migration, an additional 4.6× by employing parallelization and concurrency optimizations, and another 1.1× by utilizing our new `exchange_page()` interface, which results in 15× total improvement. For Power, we achieve 7.0×, 2.7×,and 1.2× the throughput, respectively, with the same optimizations. This results in a 21.7× throughput improvement. For an NVIDIA TX1 (ARM64), we observe 2.1× throughput with THP migration as compared to base page migration. However due to lack of NUMA support on this hardware platform, we do not include concurrent THP migration and exchange of pages, as measuring results within a single socket may skew the results.

	Page Migration Type			
Platform	Base Page	**THP**	**Opt.**	**Opt. Exch.**
Intel Xeon	0.75	2.15	9.80	11.23
IBM Power 8	1.24	8.70	23.20	26.88
NVIDIA TX1	0.64	1.36	-	-

Table 2. Maximum achieved huge page migration (in GB/s) throughput based on architecture independent optimizations (bolded) shown across three architectures. When on NVIDIA TX1 (ARM64), due to platform constraints, we are only able to run THP migration.

On the Xeon platform there is variance in copy throughput depending on the x86_64 data copy instructions (integer vs. floating-point) used. However our testing finds that SIMD floating-point instructions, like SSE and AVX, no longer provide significantly higher copy throughput than `mov` on x86_64 due to aggressive linear prefetching within caches for easily identified memory patterns like page migrations.

On Power systems, a single-threaded data transfer achieves almost 10GB/s of copy bandwidth regardless of transfer size, but this is still less than 50% of the maximum achievable copy bandwidth when using multiple threads. The improved single-threaded throughput on Power arises comes from the integer vector move instruction that moves data at an efficient 16-byte granularity. CPU instructions and architectures that can improve single threaded copy bandwidth will ultimately help both base and transparent huge page migration, but it is unlikely that even vector instructions can achieve the 15× improvement needed to match the aggregate performance this work achieves through software only techniques.

4.7 Discussion of Experimental Results

To summarize, in heterogeneous memory systems such as the disaggregated memory system we use, low-throughput page migration mechanisms (e.g., Linux's baseline) do not allow workloads to exploit the benefits of fast memory. In fact, the existing mechanisms often even degrade performance to the point that they may result in runtimes that are worse than simply running on a system without any fast memory.

Realizing the potential of multi-tiered memory requires enabling our four key optimizations: THP migration, multi-threaded copy, concurrent migration of multiple pages, and two-way exchange. Our evaluation shows that the heterogeneous memory page management system that we propose is able to deliver significant speedup across multiple benchmarks, and our design space exploration shows that our techniques are flexible and general enough to apply across a range of architectures and memory system configurations.

We also varied the bandwidth and access latency of the slow memory relative to fast memory and measured performance changes. We found that the performance improvements offered by our optimizations enjoy similar trends as those seen in Sections 4.3 and 4.4. This observation further emphasizes the importance of improving page migration mechanisms in managing tiered memory systems.

5 Related Work

Heterogeneous Memory: Hybrid memory systems consisting of high-bandwidth, low-capacity memory (e.g. Hybrid Memory Cube (HMC) [49, 63], High Bandwidth Memory (HBM) [29]) and low-bandwidth, high-capacity memory (e.g. DDR, NVM [45]) are being widely adopted by vendors [27, 57]. Recent work has investigated architecting high performance memory in a hybrid memory system either as a hardware-managed cache [68, 76] or part of the OS-visible main memory system [10, 27, 51, 75]. When hybrid memory is OS-managed, performance is primarily dependent on the service rate from the high bandwidth, low capacity memory.

Page Placement Policies: To effectively utilize hybrid memory system performance, prior art focuses on page placement *policies.* Such policies use heuristics or hardware counters for page access profiling [70, 77], dynamic page access tracking [48], or bandwidth partitioning [1]. These page access profiling techniques either require specialized hardware (precluding policy portability) or incur high overhead, reducing the number of profiled pages. For example, consider that autoNUMA is carefully designed to balance the costs of profiling with the *accuracy* of profiling. Since autoNUMA uses page faults to sample data (which can consume ∼1000 cycles [78]), it limits its sampling rate to reduce the performance problems of excessive page faults. Thermostat [2] samples page hotness using page faults (via BadgerTrap [18]). This can cause ∼4× slowdown if all pages are profiled; consequently, Thermostat only profiles 0.5% of total memory.

HeteroOS [32] targets heterogeneous memory in virtualized environments. This work adapts Linux's page replacement algorithm as we do, but relies on page hotness tracking. This tracking mechanism can cause frequent and expensive TLB invalidations; consequently the authors are careful to limit aspects of their tracking mechanism. Like our work, HeteroOS shows that the high overhead of page migration makes heterogeneous memory system management suboptimal but does not address the issue further.

Page Migration Mechanisms: Similar in spirit to optimizing page migration, one recent study focuses on enhancements to the DRAM architecture to migrate pages within the memory controller without bringing data on-chip [73, 81]. This avoids cache pollution effects. Further, to avoid locking down page accesses during migration, other proposals pin data in caches, enabling pages to be accessed during migration [7]. Others have attempted to avoid the use of the in-kernel locks by freezing the related applications instead [40]. Lin et al. propose an asynchronous OS interface called *memif* for accelerating page migration with DMA devices [43]. Instead of using traditional Linux migration interfaces, it requires rewriting programs to adopt a new interface and is limited to only ARM processors due to an architectural dependence for race detection. More recently, Ryoo et al. discover that migrating larger numbers of pages (64KB or 2MB in groups of 4KB pages) could improve application performance in heterogeneous memory systems based on their leading-load model, which further supports our THP migration proposal [72].

Huge Page Management: Ingens [35] is a huge page management framework. They focus on principled ways to coordinate the construction of more THPs. Our work asks a complementary question: how can we preserve THPs? Our work can speed up their huge page promotion process and preserve THPs during migrations.

6 Conclusions

Current OS page migration and management frameworks were developed at at time when page sizes were small and page migration existed to support memory hotplug functionality rather than performance optimization. With the introduction of heterogeneous memory systems, pressure is being put on the OS to adapt to new hardware paradigms and efficiently support multi-tiered memory systems. This work implements a novel holistic high-performance page migration system which increases migration throughput from under 100MB/s to over 10GB/s, rendering it appropriate for a wide variety of future asymmetric memory studies. We also design a low overhead page tracking and migration policy, that significantly reuses pre-existing internal OS structures, thus having wide applicability to a range of anticipated multi-level memory systems. Using a disaggregated memory system as an example, our page migration enhancements along with this native two-level paging system result in a 40% end-to-end improvement in workload performance. Our work demonstrates that combining simple management policies with high performance page migration is critical to the future of high-performance heterogeneous memory systems yet achievable.

Acknowledgments

We thank the National Science Foundation, which has supported this work through grants 1253700, 1337147, and 1319755. Additionally, we thank Guilherme Cox and Jan Vesely for their input on early drafts, as well as Chris Rossbach, who helped inform our choice of case study to quantify the benefits of our optimizations.

References

[1] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 607–618.

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[3] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[4] Andrea Arcangeli. [n. d.]. RFC: Transparent Hugepage support. https://lwn.net/Articles/358904/. [Online; accessed 31-Jul-2018].

[5] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287. https://doi.org/10.1109/PACT.2017.38

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[7] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2014. Concurrent Page Migration for Mobile Systems with OS-managed Hybrid Memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, New York, NY, USA, Article 31, 10 pages. https://doi.org/10.1145/2597917.2597924

[8] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, USA, 12–24. https://doi.org/10.1145/195473.195485

[9] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 268–280. https://doi.org/10.1145/3132402.3132404

[10] 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 (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/MICRO.2014.63

[11] Julita Corbalan, Xavier Martorell, and Jesus Labarta. 2003. Evaluation of the Memory Page Migration Influence in the System Performance: The Case of the SGI O2000. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 121–129. https://doi.org/10.1145/782814.782833

[12] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709/. [Online; accessed 31-Jul-2018].

[13] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[14] Kathy Davies. 2016. What's new in Hyper-V on Windows Server 2016 Technical Preview. https://technet.microsoft.com/en-us/windows-server-docs/compute/hyper-v/what-s-new-in-hyper-v-on-windows. [Online; accessed: 31-Jul-2018].

[15] Peter J. Denning. 1967. The Working Set Model for Program Behavior. In *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*. ACM, New York, NY, USA, 15.1–15.12. https://doi.org/10.1145/800001.811670

[16] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.50

[17] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. 2015. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 223–234. https://doi.org/10.1109/HPCA.2015.7056035

[18] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News* 42, 2 (Sept. 2014), 20–23. https://doi.org/10.1145/2669594.2669599

[19] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (May 2016), 118–126. https://doi.org/10.1109/MM.2016.10

[20] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. 2014. Large Pages May Be Harmful on NUMA Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 231–242. https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud

[21] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall. https://books.google.com/books?id=ce1QAAAAMAAJ

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[23] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-Modal DRAM Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 38–50. https://doi.org/10.1109/MICRO.2014.36

[24] 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. https://doi.org/10.1145/2731186.2731191

[25] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[26] Intel. [n. d.]. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker. [Online; accessed 31-Jul-2018].

[27] Intel. 2016. Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor. http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf. [Online; accessed 31-Jul-2018].

[28] JEDEC. 2014. JESD79-4A: DDR4 SDRAM Standard. https://www.jedec.org/sites/default/files/docs/JESD79-4A.pdf. [Online; accessed 31-Jul-2018].

[29] JEDEC. 2015. High Bandwidth Memory(HBM) DRAM - JESD235A. http://www.jedec.org/standards-documents/docs/jesd235a. [Online; accessed 31-Jul-2018].

[30] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 25–37. https://doi.org/10.1109/MICRO.2014.51

[31] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. *SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance*. Springer International Publishing, Cham, 46–67. https://doi.org/10.1007/978-3-319-17248-4_3

[32] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245

[33] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[34] Mohan Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 651–664. https://doi.org/10.1145/3173162.3173198

[35] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[36] Christoph Lameter. [n. d.]. Swap migration V3: Overview. https://lwn.net/Articles/156603/. [Online; accessed 31-Jul-2018].

[37] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages. https://doi.org/10.1145/2508834.2513149

[38] Lawerence Livermore National Laboratory. 2016. CORAL/Sierra. https://asc.llnl.gov/coral-info. [Online; accessed 31-Jul-2018].

[39] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[40] 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.

[41] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 267–278. https://doi.org/10.1145/1555754.1555789

[42] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level Implications of Disaggregated Memory. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12.

[43] 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.

[44] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 454–464. https://doi.org/10.1145/2155620.2155673

[45] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. 2015. Exploiting nvm in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. ACM, 2.

[46] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*. ACM, New York, NY, USA, 162–. https://doi.org/10.1145/977091.977115

[47] Marshall Kirk McKusick and George V. Neville-Neil. 2004. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education.

[48] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, J ohn 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)*. 126–136.

[49] Micron 2015. Hybrid Memory Cube Specification 2.1. https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf. [Online; accessed 31-Jul-2018].

[50] Micron. 2016. 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. [Online; accessed 31-Jul-2018].

[51] Jeffery Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS'09)*. USENIX Association, Berkeley, CA, USA, 14–18. http://dl.acm.org/citation.cfm?id=1855568.1855582

[52] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. In *Cray User's Group*.

[53] Linux Newbies. 2017. Linux 4.14 Release Note. https://kernelnewbies.org/Linux_4.14#Memory_management

[54] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. A Case for User-level Dynamic Page Migration. In *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/335231.335243

[55] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing (ICPP '00)*. IEEE Computer Society, Washington, DC, USA, 95–. http://dl.acm.org/citation.cfm?id=850941.852887

[56] NVIDIA Corporation. 2013. Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. [Online; accessed 31-Jul-2018].

[57] NVIDIA Corporation. 2014. NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/. [Online; accessed 14-Aug-2016].

[58] Oak Ridge National Laboratory. 2018. Summit. https://www.olcf.ornl.gov/summit/. [Online; accessed 31-Jul-2018].

[59] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 188–200. https://doi.org/10.1109/PACT.2015.30

[60] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[61] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 210–222. https://doi.org/10.1109/HPCA.2015.7056034

[62] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. 2018. SEESAW: Using Superpages to Improve VIPT Caches. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 193–206. https://doi.org/10.1109/ISCA.2018.00026

[63] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[64] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[65] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[66] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[67] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[68] 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 International Symposium on Microarchitecture*. 12. https://doi.org/10.1109/MICRO.2012.30

[69] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[70] 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. https://doi.org/10.1145/1995896.1995911

[71] 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 *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416643

[72] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3205289.3208064

[73] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, , Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, and Michael A Kozuch. 2016. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 481–493.

[74] André Seznec. 2004. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Trans. Comput.* 53, 7 (July 2004), 924–927. https://doi.org/10.1109/TC.2004.21

[75] 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.

[76] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 247–257. https://doi.org/10.1109/MICRO.2012.31

[77] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware Monitors for Dynamic Page Migration. *J. Parallel Distrib. Comput.* 68, 9 (Sept. 2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[78] Linus Torvalds. 2014. Performance profiling on core kernel code. https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6. [Online; accessed 31-Jul-2018].

[79] UEFI.org. 2017. Advanced Configuration and Power Interface Specification, Version 6.2. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf. [Online; accessed 31-Jul-2018].

[80] 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 *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. https://doi.org/10.1109/PACT.2011.65

[81] Hao Wang, Jie Zhang, Sharmila Shridhar, Gieseo Park, Myoungsoo Jung, and Nam Sung Kim. 2016. DUANG: Fast and lightweight page migration in asymmetric memory systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 481–493.

[82] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 430–443. https://doi.org/10.1145/3079856.3080211

[83] Ross Zwisler. 2017. Surface Heterogeneous Memory Performance Information. https://lwn.net/Articles/727348/. [Online; accessed 31-Jul-2018].