Transparent Hardware Management of Stacked DRAM as Part of Memory

Jaewoong Sim* Alaa R. Alameldeen† Zeshan Chishti† Chris Wilkerson† Hyesoon Kim*

*Georgia Institute of Technology †Intel Labs
jaewoong.sim@gatech.edu hyesoon@cc.gatech.edu {alaa.r.alameldeen,zeshan.a.chishti,chris.wilkerson}@intel.com

***Abstract*—Recent technology advancements allow for the integration of large memory structures on-die or as a die-stacked DRAM. Such structures provide higher bandwidth and faster access time than off-chip memory. Prior work has investigated using the large integrated memory as a cache, or using it as part of a heterogeneous memory system under management of the OS. Using this memory as a cache would waste a large fraction of total memory space, especially for the systems where stacked memory could be as large as off-chip memory. An OS-managed heterogeneous memory system, on the other hand, requires costly usage-monitoring hardware to migrate frequently-used pages, and is often unable to capture pages that are highly utilized for short periods of time.**

This paper proposes a practical, low-cost architectural solution to efficiently enable using large fast memory as Part-of-Memory (PoM) seamlessly, without the involvement of the OS. Our PoM architecture effectively manages two different types of memory (slow and fast) combined to create a single physical address space. To achieve this, PoM implements the ability to dynamically remap regions of memory based on their access patterns and expected performance benefit. Our proposed PoM architecture improves performance by 18.4% and 10.5% over static mapping and an ideal OS-based migration, respectively.

***Keywords*-Stacked DRAM, Heterogeneous Memory, Hardware Management, Die-Stacking, Two-Level Memory**

I. Introduction

With the continuing advancements in process technology, there is a clear trend towards more integration in designing future systems. In the memory subsystem, with smaller form factors and the quest for lower power, a part of memory has also started being integrated on-die or as a die-stacked DRAM. Embedded DRAM (eDRAM) has already been used in some commercial systems [1,2], and die-stacked memory is also gaining momentum [3, 4]. This trend is expected to scale by integrating larger memory capacities across market segments from mobile to server.

Integrated memory structures have often been exploited in prior work as hardware-managed last-level caches [5–10]. In such a cache design, allocating a cache line involves making a local (i.e., redundant) copy of the data stored in main memory. In this case, cache capacity is invisible to system memory, but applications can experience reasonable performance benefits without modifications to the operating systems (OS) or running software. With conventional cache size of a few megabytes per core (or tens of MBs per core as in today's eDRAMs), the opportunity costs of losing overall memory capacity to cache copies are insignificant. However, the integrated memory structures driven by die-stacking technology could provide hundreds of megabytes of memory capacity *per core*. Micron already offers 2GB Hybrid Memory Cube (HMC) samples [11]. By integrating multiple stacks on a 2.5D interposer, it is also plausible to integrate even tens of gigabytes of memory on package. For some market segments, depending on its deployment scenarios, making the integrated memory invisible to overall system memory (i.e., used as a cache) could lead to a non-negligible loss of a performance opportunity.[1]

An alternative to using on-die memory as a cache is to use it as part of an OS-managed heterogeneous system, as in non-uniform memory architectures (NUMA) [13–15]. NUMA systems were quite popular in designing large-scale high-performance computers even without on-die memory integration. NUMA allows processors fast access to data in memory that is closer in proximity to the processor. With careful OS-managed page migration and/or replication policies, processors could get most of the data they need in near memory. However, performing migration under OS control implies a high latency overhead since it could only happen through OS code. Furthermore, OS-managed migration could only happen at coarse-grained intervals since the OS routines cannot be called frequently. This could miss many opportunities to improve performance by migrating pages that are highly utilized for short periods of time.

In this paper, we propose architectural mechanisms to efficiently use large, fast, on-die memory structures as part of memory (PoM) seamlessly through the hardware. Such a design employing effective management would achieve the performance benefit of on-die memory caches without sacrificing a large fraction of total memory capacity to serve as a cache. As we discuss in Section III, the main challenge for the hardware-based management is to keep the hardware cost of meta-data and other structures in check. Our PoM architecture provides unique designs and optimizations that become very effective in our problem space, which we cover in Section IV. In contrast to OS-managed policies, our approach is transparent to software and achieves higher performance due to its ability to adapt and remap data at a fine granularity.

[1]Some industry architects consider providing the option to use gigabytes of on-die memory as part of system-visible memory in addition to a large cache configuration [12].

This paper makes the following contributions:

- We propose a Part-of-Memory (PoM) architecture that efficiently manages a heterogeneous memory system by remapping data to fast memory without OS intervention. To our knowledge, this is the first work that provides a practical design of hardware-based management.
- We propose two-level indirection with a remapping cache and memory permutation for hardware-based PoM management to balance the additional latency for indirection and on-die storage overheads.
- We propose a competing counter-based page activity tracking and replacement mechanism that is suitable to implement for the PoM architecture. Our mechanism provides a responsive remapping decision while being area-efficient, which is necessary for hardware-based PoM management.

Our results show that when these techniques are used in concert they achieve an 18.4% performance improvement over static mapping, and a 10.5% improvement over an *ideal* OS-based dynamic remapping policy.

II. Background and Motivation

A. Heterogeneous Memory System

We define a *heterogeneous memory system* as a memory subsystem in which some portions of memory provide different performance/power characteristics than other portions of memory in the same node. We project that many of future memory systems will exploit such *heterogeneity* to meet a variety of requirements imposed on today's memory architectures. A good example is a memory subsystem composed of stacked DRAM and off-chip memory [6, 16], where DRAM stacked on the processor provides both higher bandwidth and faster access time as compared to the off-chip memory. Another example is the recently proposed tiered-latency DRAM architecture, where a DRAM array is broken up into *near* and *far* segments that have *fast* and *slow* access times, respectively [17]. Such heterogeneous memory is often equipped with a few gigabytes of fast memory, so a key to achieving more system performance is how to make an efficient use of such large fast memory. In Sections II and III, we discuss the opportunities and challenges of architecting such heterogeneous memory systems.

B. Architecting Fast Memory as Part-of-Memory (PoM)

Heterogeneous memory can be managed in a number of ways. The most common approach is to manage it as a cache [5,6]. Generally, in this approach, allocating a block in the fast memory entails the duplication of the slow memory block into the fast memory. Although such duplication results in capacity loss, it makes block allocations simple and fast. However, in cases where the capacity of the fast memory is comparable to that of the slow memory, the capacity lost in duplication may be unacceptable. In these cases, both fast and slow memory may be combined into a single flat address space. We refer to this as a *PoM (Part-of-Memory)* architecture. The simplest PoM architecture resembles typical homogeneous memory architectures, with a portion of the address space mapped statically to the fast memory whereas the remainder mapped to the slow memory. To maximize the performance benefits of fast memory, the operating system (OS) could allocate heavily used pages to the portion of the physical address space mapped to the fast memory.

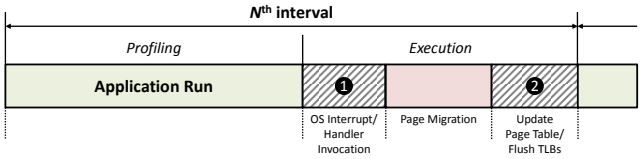

Figure 1. A high-level view of an OS-based PoM management.

C. Dynamic PoM Management

The key advantage of the PoM architecture is the ability to increase overall memory capacity by avoiding duplication. However, its performance may suffer relative to a simple cache. The PoM architecture is at a disadvantage for two reasons. First, the performance benefits of the fast memory will depend on the operating system's ability to identify frequently used portions of memory. Second, even if the most frequently used portions of memory can be successfully identified, a replacement policy that relies on frequency of use may underperform the typical cache recency based replacement algorithm [18].

Figure 1 shows an overview of an OS-based PoM management. At a high-level, such dynamic management consists of two phases of *profiling* and *execution*. At every interval, we first need to collect information that helps determine the pages to be mapped into fast memory during run-time (Application Run). The operating system generally has a limited ability to obtain such information; a reference bit in page tables is mostly the only available information, which provides a low resolution of a page activity. Thus, richer hardware support for profiling may be desirable even for an OS-based PoM management. A typical way of profiling such as used in [19] has hardware counters associated with every active page and increments the counter for the corresponding page on a last-level cache (LLC) miss. The profiled data is then used to perform page allocations for the next interval during the *execution* phase. In an OS-based management, the *execution* is costly since it involves an OS interrupt/handler invocation, counter sorting, page table modification, and TLB flushing in addition to the *actual* page allocation cost. As such, the OS-involved execution must be *infrequent* and thus often fails to exploit the full benefits of fast memory.

D. Potential of Hardware-Managed PoM Architecture

By managing the PoM architecture without an involvement of the operating system, we can eliminate the overhead of an OS interrupt/handler invocation (❶) in the execution phase. More importantly, we do not need to wait for an OS quantum in order to execute page allocations, so the

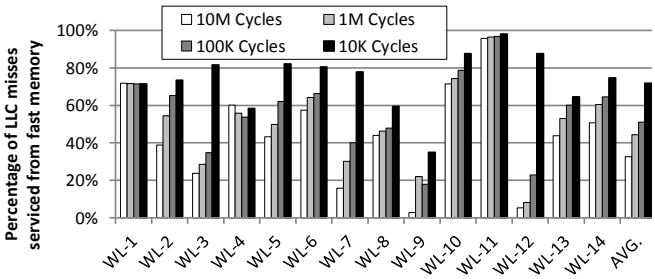

Figure 2. Percentage of LLC misses serviced from fast memory across different intervals.

execution can happen at any rate. Therefore, the conventional approach of (long) interval-based profiling and execution is likely to be a less effective solution in the hardware-managed PoM architecture. To see how much benefit we can approximately expect by exploiting recency, we vary the interval size in a frequency-based dynamic mechanism similar to that described in Section II-C.

Figure 2 presents the percentage of LLC misses serviced from fast memory while varying the interval size from 10M cycles to 10K cycles (see Section V for our methodology). As the interval size decreases, the service rate from fast memory significantly increases for many workloads. This implies that we could miss many opportunities for a performance improvement if a tracking/replacement mechanism in the PoM architecture fails to capture pages that are highly utilized for short periods of time. The potential of the hardware-managed PoM architecture could be exploited only when we effectively deal with the cost of hardware management, which we describe in the next section.

III. Challenges of Hardware-Managed PoM

The high-level approach of *profiling* and *execution* remains the same in the hardware-managed PoM architecture. However, the hardware-managed PoM architecture introduces the following new challenges.

A. Hardware-Managed Indirection

Dynamic PoM management performs relocating pages into the memory space that is different from what OS originally allocated to, thus the hardware-managed PoM must take responsibility for maintaining the integrity of the operating system's view of memory. There are two ways this could be achieved. First, PoM could migrate memory regions at the OS page granularity, update the page tables, and flush TLBs to reflect the new locations of the migrated pages. Unfortunately, this method is likely infeasible in many architectures since it would require the availability of all the virtual addresses that map to the migrating physical page in order to look up and modify the corresponding page table entries. In addition, the OS page granularity could be too coarse-grained for migration, and updating page tables and flushing TLBs (❷) still need to be infrequent since they are expensive to perform, thereby leading to lack of adaptation to program phase changes. Therefore, using this method in the hardware-managed PoM is unattractive.

The other approach is to maintain an indirection table that stores such new mapping information and to *remap* memory requests targeting the pages that have been relocated into the non-original memory space. The remapping table, however, could be too large to fit in on-die SRAM storage. For example, 2GB of fast memory managed as 2KB segments[2] will require a remapping table consisting of 1M entries *at least* to support the cases where all the segments in the fast memory have been brought in from slow memory. Note, in this approach, that every memory request that missed in the LLC must access the remapping table to determine where to fetch the requested data (i.e., whether to fetch from the original OS-allocated address or from the hardware-remapped address). Thus, in addition to the concern of providing such large storage on-chip, the additional latency of the indirection layer would be unmanageable with a single, large remapping table, which is another critical problem.

To overcome the problem of a single, large remapping table, we propose a PoM architecture with *two-level* indirection in which the large remapping table is embedded into fast memory, while only a small number of remapping entries are cached into an on-die SRAM structure. Although the two-level indirection may make the PoM architecture more feasible in practice, naively designing the remapping table makes the caching idea less effective. Section IV describes the remapping table design that is suitable for such caching yet highly area-efficient.

B. Swapping Overhead

A key distinction between PoM and cache architectures is the need to *swap* a segment to bring it to fast memory, rather than just *copy* a memory block when allocating it to the cache. PoM also differs from an exclusive cache hierarchy since caches are backed up by memory (i.e., a clean block in any level of an exclusive cache hierarchy is also available in memory). Conversely, only one instance of each segment exists in PoM, either in slow or in fast memory.

Swapping a segment differs from allocating a cache block since the segment allocated to fast memory replaces another segment that occupied its new location, and the swapped-out segment needs to be written back to slow memory. Therefore, every allocation to fast memory requires a write-back of the evicted data to slow memory. This swapping overhead could be significant depending on the segment size and the width of the channel between fast and slow memory. A small segment size reduces the swapping cost of moving large blocks between large and slow memory, and provides more flexibility in the replacement policy. However, a small segment size such as 64B or 128B (typical in caches) reduces spatial locality, incurs a much higher storage overhead for the remapping table, and suffers from a higher access latency due to the large remapping table size. In Sections IV-C and IV-D, we explore different designs to balance between minimizing swap overhead and remapping table size.

[2]We use the term *segment* to refer to the management granularity in our PoM architecture.

C. Memory Activity Tracking and Replacement

Providing efficient memory utilization tracking and swapping mechanisms specifically tailored to the hardware-managed PoM architecture is another major challenge. If we simply use the mechanism similar to that in Section II-C, we need to maintain a counter per active page, so we could need as many counters as the number of page table entries in the worst case. In addition, due to the required large interval, each counter needs to have a large number of bits to correctly provide the access information at the end of each interval. For example, with a 4GB total memory with 2KB segments, we need to track as many as 2M entries; then, assuming that the size of each entry is 16 bits (in order not to be saturated during a long interval), the tracking structure itself requires 4MB storage. Having shorter intervals could help mitigate the storage overhead a bit by reducing the number of bits for each counter, but comparing all the counters for shorter intervals would greatly increase the latency and power overhead. Furthermore, the storage overhead would still be bounded to the number of page table entries, which may be undesirable for scalability.

To make a responsive allocation/de-allocation decision with a low-cost tracking structure, we propose competing counter-based tracking and swapping for our PoM architecture in which a single counter is associated with multiple segments in fast and slow memory. Section IV-F discusses how we reduce the number of counters as well as the size of each counter while providing *responsiveness*.

D. Objective and Requirements

The primary objective of this work is to efficiently enable the PoM architecture in heterogeneous memory systems. For this purpose, we need to address the previous challenges. The hardware-managed indirection needs to be fast and area-efficient. The swapping cost needs to be optimized. The memory utilization tracking structure also needs to be small in area while being designed to provide *responsive* swapping decisions. In the next section, we describe our PoM architecture and how it addresses these main challenges.

IV. A Practical PoM Architecture

A. Design Overview

In a conventional system, a virtual address is translated to a physical address, which is then used to access DRAM. In contrast, our system *must* provide the ability to remap physical addresses in order to support the transparent swapping of memory blocks between fast and slow memory. Starting with the physical address retrieved from the page tables (Page Table Physical Address, *PTPA*), we must look up a remapping table to determine the actual address of the data in memory (DRAM Physical Address, *DPA*). Unfortunately, as discussed in Section III-A, such single-level indirection with a large remapping table not only has a non-negligible storage overhead but also incurs a long latency penalty on every access.

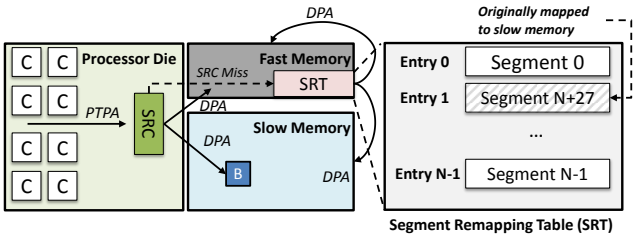

Figure 3. Overview of the PoM architecture.

Figure 3 presents the overview of our PoM architecture. One of the key design points in our PoM architecture is *two-level* indirection with a remapping cache. Each request to either slow or fast memory begins by looking for its remapping information in the segment remapping cache (SRC). If the segment remapping cache does not contain an appropriate remapping entry (*SRC Miss*), then the memory controller retrieves the remapping entry from the segment remapping table (SRT) located in fast memory and allocates it in the SRC. Once the remapping entry has been fetched, the location of the request (i.e., DPA) is known, and the data can be fetched.

At a minimum, a remapping entry needs to indicate which segment is currently located in *fast* memory. For example, with 4GB fast/16GB slow memory and 2KB segments, the maximum number of segments that fast memory can accommodate is 2M (out of total 10M segments). In this configuration, the *minimum* number of remapping entries required for the SRT would be 2M. With the minimal remapping table design, when a segment originally allocated to slow memory by the operating system is brought into one of the locations in fast memory, the corresponding remapping entry is modified to have new mapping information, such as Entry 1's "Segment N+27" in Figure 3; then, "Segment 1" is stored in the original OS-allocated location of "Segment N+27". Note that, even with the simplest design, the size of the remapping table is bounded to the number of segments in the fast memory, so the storage and latency overheads would still be high to use an on-chip SRAM structure for the remapping table. We discuss the implementation of the remapping table in more detail in Section IV-D.

B. Segment-Restricted Remapping

At first blush, it seems that we can simply cache some of the SRT entries. However, our caching idea may not be easily realized with the SRT described in the previous section due to a huge penalty on an SRC miss. On an SRC miss, we need to access fast memory to retrieve the remapping information. In the above SRT design, however, since the remapping information can be located anywhere (or nowhere) for a miss-invoked memory request, we may need to search all the SRT entries in the worst case, which could require thousands of memory requests to fast memory just for a single SRC miss.

Our solution to restricting the SRC miss penalty within a single fast memory access is *segment-restricted* remapping, as shown in Figure 4. Each entry in the remapping table

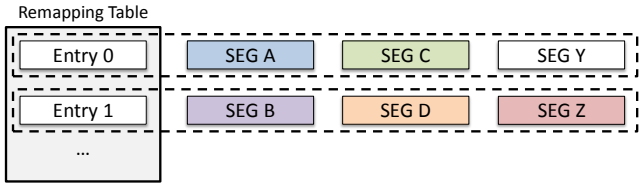

Figure 4. Segment-restricted remapping.

owns some number of segments where the number can be determined by the total number of segments over the number of SRT entries in the simplest case. A segment is restricted to be swapped only for the segments that are owned by the same entry. This is a similar concept to direct-mapping in cache designs (but is easily extensible to set-associative segment-restricted designs). For example, segments A, C and Y are only allowed to be swapped for the segments owned by Entry 0, whereas segments B, D and Z are swapped only for the segments owned by Entry 1. In this segment-restricted remapping, even if the remapping information for segment A is not found in the SRC, we can retrieve the remapping information with a single access to fast memory since it is only kept in Entry 0. To determine the SRT entry to which a segment is mapped, we simply use a few bits from the page table physical address (PTPA), which is good enough for our evaluated workloads.

C. Segment Allocation/De-allocation: Cache vs. PoM

In this section, we compare a cache allocation and the swap operation required by our PoM architecture. Throughout the examples, segments X, Y and Z are all mapped to the same location in fast memory (as in the segment-restricted remapping), and non-solid segments in slow memory represent the segments displaced from their original OS-allocated addresses.

First, Figure 5 shows a cache line allocation (segment Z) under two different conditions. In the example on the left (Clean), segment Z is brought into a local buffer on the CPU (❶) and simply overwrites segment Y (❷). Before Z is allocated, both fast memory and slow memory have identical copies of segment Y. As a result, allocating Z requires nothing more than overwriting Y with the contents of the newly allocated segment Z. The right example (Dirty) illustrates a case in which Y is modified and thus the copy of Y held in fast memory is unique. Allocating Z, in this case, requires reading Y from fast memory (❶) and Z from slow memory (❷) simultaneously. After storing them in buffers in the memory controller, Z is written back to fast memory (❸), and Y is written back to slow memory (❹).

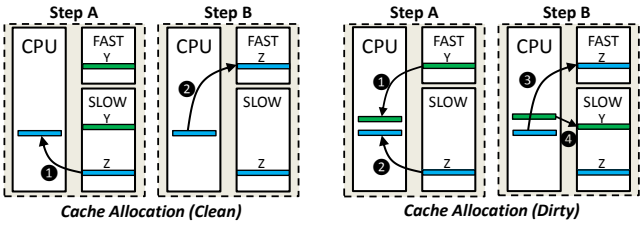

Figure 5. Cache allocation.

In contrast to the cache allocation, the PoM architecture makes all of both fast and slow memory available to the running software. To prevent duplicates and to ensure that all data is preserved as data is moved back and forth between fast and slow memory, PoM replaces the traditional cache allocation with a swap operation. The PoM swap operation, illustrated in Figure 6, differs depending on the contents of the segment displaced from fast memory. The PoM swap operation on the left (*PoM Fast Swap1*) occurs when the displaced segment X was originally allocated by the operating system to fast memory. In this case, a request to segment Z in slow memory requires segments X and Z to be read simultaneously (❶,❷) from fast and slow memory into on-chip buffers. The swap completes after copying Z from the on-chip buffer to fast memory (❸) and copying X from the buffer to slow memory (❹).

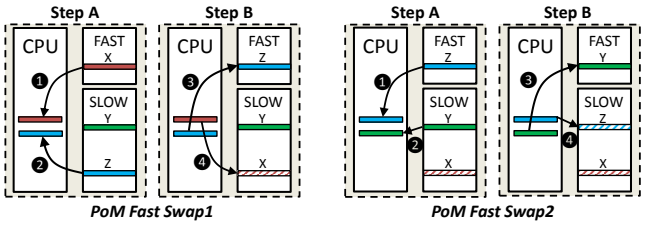

Figure 6. Fast swap operation in the PoM architecture.

As different segments from slow memory are swapped into fast memory, the straightforward swap operation previously described will result in segments migrating to different locations in slow memory, as illustrated in Figure 6 (*PoM Fast Swap2*). In the example, a request to segment Y causes a second swap after Swap1. The second swap (Swap2) simply swaps segment Y with segment Z, resulting in segment Z assuming the position in slow memory that was originally allocated to segment Y. With more swaps, all slow memory segments could end up in locations different than their original location. This segment motion between different locations in slow memory implies that the remapping table must not only identify the current contents of fast memory, but must also track the current location of *all* segments in slow memory. Note that recording only the segment number brought into fast memory, as the remapping entry shown in Figure 3, would not allow this fast swap in most cases. The ability to support segment motion throughout slow memory increases the size and complexity of the remapping table, but greatly simplifies the swapping operation.

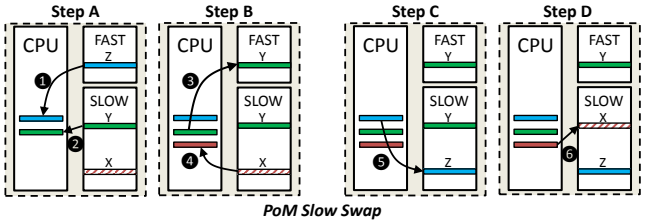

Figure 7. Slow swap operation in the PoM architecture.

An alternative approach to remapping segments requires segments to always return to their original position in slow memory. In this approach, the positions of all segments in

slow memory can be inferred from their page table physical address, with the exception of segments currently mapped to fast memory. To ensure this, we can employ a second swapping algorithm depicted in Figure 7 (*PoM Slow Swap*). In the example, as in *PoM Fast Swap2*, a request to segment Y causes a swap with segment Z, currently in fast memory. In this case, however, rather than perform a simple swap between Z and Y, we restore Z to its original position in slow memory, currently occupied by X. We accomplish this in four steps: (A) Fetching Z and Y simultaneously (❶,❷); (B) Writing Y to fast memory (❸) and simultaneously fetching X from slow memory (❹); (C) Freeing X's location then writing Z back to its original location (❺); (D) Writing X to Y's previous location in slow memory (❻). The slow PoM swap generally requires twice as much time as the fast PoM swap with each of the four steps requiring the transfer of a segment either to or from slow memory.

D. Segment Remapping Table (SRT)

The segment remapping table (SRT) size depends on the swapping type we support. The PoM slow swap ensures that all data in slow memory is stored at its original location as indicated by its page table physical address. As a result, the SRT can include remapping information only for the segments in fast memory. Conversely, PoM fast swap allows data to migrate throughout slow memory; thus, the remapping table *must* indicate the location of each segment in slow memory. For instance, consider a system consisting of 1GB of fast memory and 4GB of slow memory divided up into 2KB segments. This system would require a remapping table with the ability to remap 512K (i.e., 1GB/2KB) segments if it implemented slow swaps, and 2M (4GB/2KB) segments if it implemented fast swaps. The discussion in the remainder of this section will focus on a remapping table designed to support fast swaps.

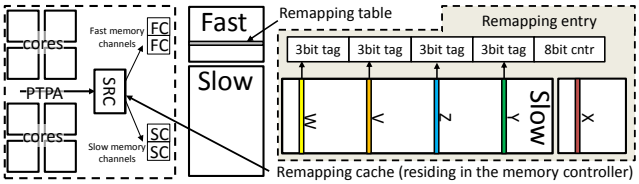

Figure 8. Remapping table organization.

In the case of fast swaps with the previous system configuration, a total of five possible segments compete for a single location in fast memory, and the other segments will reside in one of the four locations available in slow memory. The SRT needs to record which of the five possible segments currently resides in each of five possible locations. Figure 8 illustrates the organization of the remapping table with the five segments, V, W, X, Y and Z competing for a single location in fast memory. Each remapping entry in the SRT contains tags for four of the five segments; the contents of the fifth segment can be inferred from that of other segments. In addition to the four 3-bit tags, the remapping table contains a shared 8-bit counter used to determine when swapping should occur (Section IV-F). Co-locating the tags for conflicting segments has two advantages. First, since all swaps are performed between existing segments, this organization facilitates updates associated with the swap operation. Second, it facilitates the usage of the shared counter that measures the relative usage characteristics of the different segments competing for allocation in fast memory.

E. Segment Remapping Cache (SRC)

The remapping cache must be designed with two conflicting objectives in mind. On the one hand, a desire to minimize misses provides an incentive to increase capacity and associativity. On the other hand, increasing capacity can increase access latency, which negatively impacts the performance of both hits and misses. To strike this balance, we choose a 32KB remapping cache with limited (4-way) associativity. On an SRC miss, we capture limited spatial locality by fetching both of the requested remapping entry and the second remapping entry that cover an aligned 4KB region. It is worth noting that with a protocol similar to DDR3, our fast memory will deliver 64B blocks. Although a single 64B block would contain tens of remapping entries, we found that SRC pollution introduced by allocating a large number of remapping entries outweighed the spatial locality we could harvest. Since we modeled 4KB OS pages, any spatial locality that existed beyond a 4KB region in the virtual address space could potentially have been destroyed after translation to the physical address space. It is also noted that an SRC hit for a given memory request does not guarantee that the requested data is found in fast memory.

F. Segment Activity Tracking

We previously discussed in Section III-C that a conventional segment tracking/replacement mechanism is not suitable for a hardware-managed PoM architecture, and a tracking mechanism for PoM needs to respond quickly with a low storage overhead (e.g., a small number of counters, fewer bits per counter). In this section, we discuss the tracking/replacement mechanism for our PoM architecture.

1) Competing Counter: To make a segment swapping decision, we need to compare the counter values of all involved segments at a decision point (e.g., sorting). Here, the information of interest is in fact the one relative to each other rather than the absolute access counts to each segment. For example, assume that one slot exists in fast memory with both segments A and Y competing for the slot, which is currently taken by segment Y. To decide which segment should reside in fast memory, we allocate a counter associated with a segment in fast memory (segment Y) and another segment in slow memory (segment A). During an application run, we decrement the associated counter on an access to segment Y, and increment it on an access to segment A. By having this *competing counter* (CC), we can assess which of the two segments has been accessed more during a certain period, which would be useful for swap decisions.

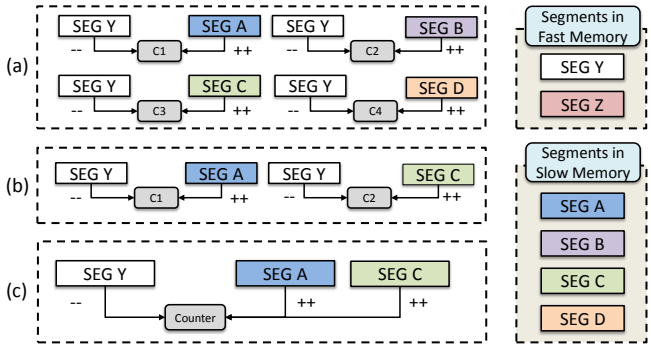

Figure 9. Competing counters.

Figure 9 illustrates a general case in which multiple slots exist in fast memory, while also a number of segments are competing for the slots. At first blush, the CC-based approach seems to incur high overhead since the competing counters need to be allocated to all combinations of the segments in fast and slow memory, as shown in Figure 9(a) (counters shown only for segment Y due to space constraints), if segments are allowed to be mapped into *any* location in fast memory. However, thanks to the segment-restricted remapping described in Section IV-B, the number of competing counters required is in fact bounded to the number of segments in slow memory, as shown in Figure 9(b). Although this already reduces the storage overhead compared to the tracking structure in Section III-C while providing more responsiveness, we can further reduce the storage overhead by *sharing* a single counter between competing segments, as shown in Figure 9(c). This reduces the number of counters to a single counter for each segment in fast memory. In this *shared* counter case, the segment that has just triggered swapping is chosen for allocation in fast memory.

Sharing the competing counters between competing segments provides us with two benefits. First, it reduces the overall memory capacity required by the segment remapping table (Section IV-D). Second, and more importantly, it reduces the size of each SRC entry by a little more than 50%, allowing us to effectively double its capacity. Furthermore, sharing counters between competing segments seems to have little to no effect on the performance of our replacement algorithm. Theoretically, references to segment A could have incremented the shared counter just below a threshold, and segment C could cause the counter to reach the allocation threshold and is chosen for allocation. In practice, however, we found this to be rare since the usage of different segments among competing segments tends to be highly asymmetric. Even though this rare case could happen, it is likely to have a temporary effect, and the highly-referenced segment would end up residing in fast memory soon afterwards.

2) Swapping Operation: Swapping occurs when the counter value is greater than a threshold, which implies that the segment currently residing in fast memory may not be the best one. For example in Figure 9(c), when an LLC miss request to segment A increments its associated counter, if the resulting counter value is greater than a threshold, segments A and Y will be swapped and their associated counter will reset.

An optimal threshold value would be different depending on the application due to the different nature of memory access patterns. To determine a suitable swapping rate for different applications, the PoM architecture samples memory regions. The locations in fast memory are grouped into 32 distinct regions in an *interleaving* fashion, and four regions are dedicated to sampling, while other 28 regions follow the threshold decision from sampling. The segments in the sampling regions modify the remapping table/cache when their counter values are greater than the assigned thresholds, but the actual swapping is *not* performed for the segments restricted to the sampling region. For the memory requests that target sampling regions, we simply get the data with static mapping without looking up in the remapping table (i.e., DRAM PA = Page Table PA). In order to drive the suitable swapping rate, we collect the following information for each sampling region:

- N_{static}: # of memory requests serviced from fast memory with static mapping
- N_{dynamic}: # of memory requests *expected* to be serviced from fast memory when swapping with a given threshold
- N_{swap}: # of *expected* swaps for a given threshold.

For each of four sampling regions, we then compute the expected benefit (B_{expected}) using Equation (1) and choose the threshold used in the sampling region that provides the highest *non-negative* B_{expected} value at every 10K LLC misses. In the case where such B_{expected} does not exist (i.e., all negative), the following regions do not perform any swapping operations.

$$B_{\text{expected}} = (N_{\text{dynamic}} - N_{\text{static}}) - K \times N_{\text{swap}}. \quad (1)$$

N_{dynamic} is counted just by checking the remapping table for the requests to the segments in fast/slow memory dedicated to the sampling regions. N_{static} is counted when the requests target the segments originally assigned to the fast memory. K is the number of extra hits required for a swapped-in segment (over a swapped-out segment) to compensate for the cost of a single swap. K differs depending on the relative latency of fast and slow memory. In our configuration (see Table I), the cost of a single fast swap is about 1200 cycles, and the difference in access latency between fast and slow memory is 72 cycles.[3] Thus, in general, the swapped-in segment needs to get at least 17 more (future) hits than the swapped-out segment for swapping to be valuable. K is computed in hardware at boot time. Note that the memory controller knows all the timing parameters in both fast and slow memory. In our evaluations, we use 1, 6, 18, and 48 for the thresholds in four sampling regions, and we use K = 20.

[3](11 ACT + 11 CAS + 32×4 bursts) × 4 (clock ratio of CPU to DRAM) = 600 cycles. Fast swapping requires two of these (Section IV-C).

V. Experimental Methodology

Simulation Infrastructure: We use a Pin-based cycle-level x86 simulator [20] for our evaluations. We model die-stacked DRAM as on-chip fast memory, and we use the terms of fast memory and stacked memory interchangeably in our evaluations. The simulator is extended to provide detailed timing models for both slow and fast memory as well as to support virtual-to-physical mapping. We use a 128MB stacked DRAM and determine its timing parameters to provide the ratio of fast to slow memory latency similar to that in other stacked DRAM studies [5–10, 16]. Table I shows the configurations used in this study.

Table I
Baseline configuration used in this study

CPU	
Core	4 cores, 3.2GHz out-of-order, 4 issue width, 256 ROB
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	8-way, private 256KB (8-cycle)
L3 cache	16-way, shared 4MB (4 tiles, 24-cycle)
SRC	4-way, 32KB (2-cycle), LRU replacement
Die-stacked DRAM	
Bus frequency	1.6GHz (DDR 3.2GHz), 128 bits per channel
Channels/Ranks/Banks	4/1/8, 2KB row buffer
tCAS-tRCD-tRP	8-8-8
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11

Workloads: We use the SPEC CPU2006 benchmarks and sample one-half billion instructions using SimPoint [21]. We selected memory-intensive applications with high L3 misses per kilo instructions (MPKI) since other applications with low memory demands have very little sensitivity to different heterogeneous memory management policies. To ensure that our mechanism is not harmful for less memory-intensive applications, we also include two applications that show intermediate memory intensity. We select benchmarks to form rate-mode, where all cores run separate instances of the same applications, and multi-programmed workloads. Table II shows the 14 workloads evaluated for this study along with L3 MPKI of a single instance in each workload as well as the speedup of the all-stacked DRAM configuration where all the L3 miss requests are serviced from stacked DRAM instead of from off-chip DRAM. For each workload, we simulate 500 million cycles of execution and use weighted speedup [22, 23] as a performance metric.

Table II
Evaluated multi-programmed workloads

Mix	Workloads	L3 MPKI (single)	All-stacked
WL-1	4 × mcf	71.48	1.88x
WL-2	4 × gcc	12.13	1.27x
WL-3	4 × libquantum	35.56	2.12x
WL-4	4 × omnetpp	7.30	1.47x
WL-5	4 × leslie3d	16.83	1.68x
WL-6	4 × soplex	31.56	1.81x
WL-7	4 × GemsFDTD	12.15	1.33x
WL-8	4 × lbm	32.83	3.37x
WL-9	4 × milc	18.01	1.75x
WL-10	4 × wrf	6.28	1.49x
WL-11	4 × sphinx3	11.89	1.52x
WL-12	4 × bwaves	19.07	2.00x
WL-13	mcf-lbm-libquantum-leslie3d	N/A	1.87x
WL-14	wrf-soplex-lbm-leslie3d	N/A	1.77x

VI. Experimental Evaluations

A. Performance Results

Figure 10 shows the performance of our proposed scheme and a few other static mapping/OS-based dynamic remapping policies for comparisons. We use a baseline where no stacked DRAM is employed, and all performance results are normalized to the baseline.

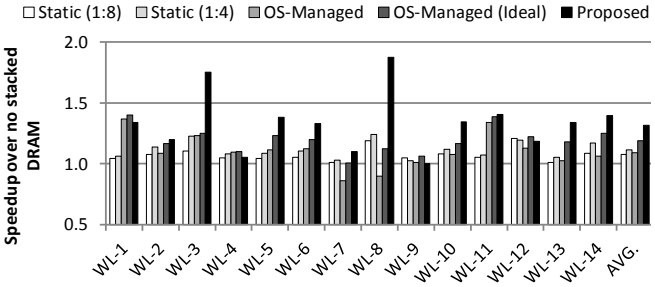

Figure 10. Speedup with our proposed mechanism compared to other schemes (normalized to no stacked DRAM).

First, `static(1:8)` and `static(1:4)` show the speedups when we assume that the OS would allocate memory pages such that one ninth or one fifth of the total pages are placed in fast memory for each workload. Static mapping results show the performance improvement due to having a part of memory that is accessed quickly without making any changes to hardware or OS page allocation policies. On average, we achieve a 7.5% (11.2%) speedup over the baseline with this 1:8 (1:4) static mapping.

Our proposed scheme achieves a 31.7% performance improvement over the baseline on average, and also shows substantial performance improvements over static mappings. Compared to `static(1:4)`, our scheme improves performance by 18.4% on average, and many of the evaluated workloads show huge speedups due to serving more requests from fast memory (see Section VI-B). On the other hand, a few other workloads show performance similar to that of `static(1:4)`. This happens because of one of two reasons. First, our dynamic scheme could determine that the cost of page swapping would outweigh its benefit for some workloads (e.g., WL-4 and WL-9), so the original page allocation by the OS remains the same in both fast and slow memories, and none (or only a small number) of the pages are swapped in-between. Second, some workloads are not as sensitive to memory access latency as others, so their performance improvement due to our mechanism is limited by nature (e.g., WL-2 and WL-7).

Next, `OS-Managed` and `OS-Managed(Ideal)` are OS-based migrations *with* and *without* remapping overhead, respectively. We use a mechanism similar to that used in prior work [19]. In the mechanism, the OS first collects the number of accesses (LLC misses) to each 4KB OS page during an interval. Then, at the end of the interval, the most frequently accessed pages are mapped into the fast memory for the next interval. To mitigate remapping overheads, the mechanism uses a 100M cycle interval (31.25 ms on a 3.2GHz processor) and also does not select the pages with

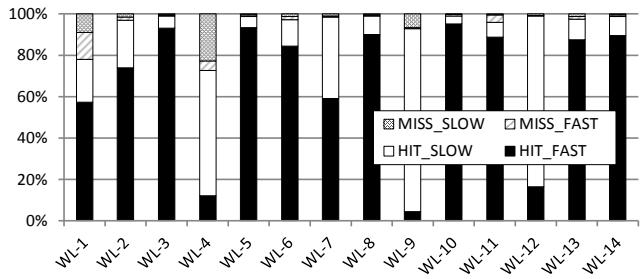

Figure 11. Physical address translation breakdown.

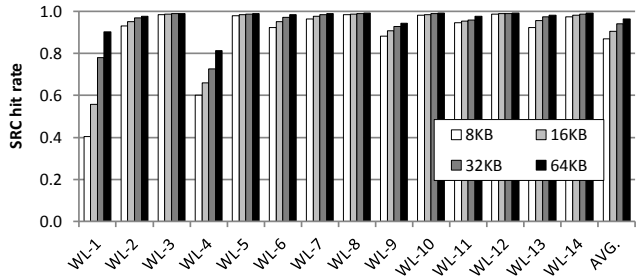

Figure 12. SRC hit rate across different cache size.

fewer than 64 LLC misses.[4] Also, if selected pages for the next interval were already present in fast memory, the pages are not moved. On average, the OS-based migration achieve a 9.3% (19.1% without overhead) speedup over the baseline.

Compared to `OS-Managed`, our PoM achieves 20.5% performance improvement. In our proposed scheme, we start out with the same page allocation as in `OS-Managed`. However, we continuously adapt and remap data into fast memory at a *fine* granularity during the workload runtime, whereas `OS-Managed` adapts to the working set changes at a *coarse* granularity; so, the service rate from fast memory of `OS-Managed` would be quite lower than our mechanism. As a result, even assuming zero-cost overheads, such as `OS-Managed(Ideal)`, the OS-based mechanism performs worse than ours.

B. Effectiveness of Remapping Cache

The effectiveness of the remapping cache is very crucial to our two-level indirection scheme. Figure 11 shows the percentage of memory requests serviced from fast memory along with the source of the translation (i.e., whether translations are obtained from the remapping cache or the remapping table).

The HIT_FAST bar represents the percentage of requests whose translations hit in the remapping cache and the corresponding data are serviced from fast memory. The HIT_SLOW bar represents the percentage of requests whose translations hit in the remapping cache but are serviced from slow memory. MISS_FAST and MISS_SLOW represent remapping cache misses that are serviced from fast and slow memory, respectively. Note that the hit rate of the remapping cache is independent of swapping schemes since it is a function of an LLC-filtered memory access stream.

Our remapping cache is shown to be quite effective with more than 95% hit rate for most workloads. The high hit rate is due to the spatial locality in the lower levels of the memory hierarchy. WL-1 and WL-4 are the only benchmarks that show slightly lower hit rates due to the low spatial locality.

C. Sensitivity to Remapping Cache Size

One of the main performance drawbacks of remapping is that address translation latency is now added to the overall memory latency. For an effective design, most of translations need to hit in the remapping cache to minimize the cost of access serialization. This section analyzes the effectiveness of different sizes of the remapping cache. Figure 12 shows the hit rate of the remapping cache when we change its size from 8KB to 64KB. Our workloads show high hit rates on average even with the 8KB remapping cache. Only a few workloads (e.g., WL-1) experience a bit higher number of remapping cache misses with small size caches since their accesses are spread out across large memory regions. Note that we used simple LRU replacement for the remapping cache. If needed, other replacement policies or techniques (e.g., prefetching) could be used to improve the hit rate.

D. Swapping Overhead

Figure 13 shows the swapping overhead of our proposed mechanism compared to the unrealistic *ideal* case in which swapping has no overhead; i.e., swapping does not generate any memory requests, and updates the swapping table automatically without using up any latency or bandwidth.

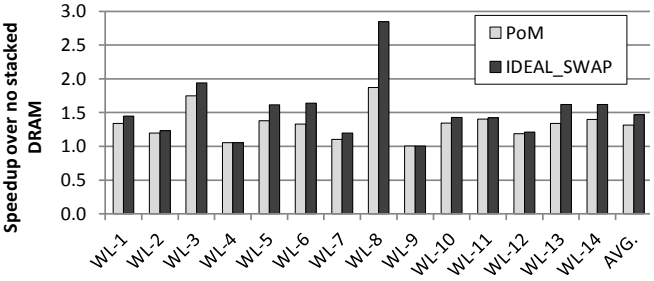

Figure 13. Comparison with ideal swapping.

Most of our evaluated workloads show reasonable swapping overhead since our mechanism enables swapping for small data segments (i.e., 2KB segments) and also attempts to avoid unnecessary swaps that are not predicted to improve performance. Some workloads such as WL-4 and WL-9 show no swapping overhead. This is because our dynamic scheme determines that the swapping would not be beneficial compared to static mapping as discussed in Section VI-A.

Intuitively, the performance cost of page swapping diminishes as the number of hits per swap increases. This can be achieved by swapping in more *guaranteed-to-be-hit* pages into fast memory. However, such a conservative decision is likely to reduce the amount of the memory requests serviced from fast memory. Thus, we first need to make a careful decision on when to swap in order to optimize the swapping cost without sacrificing the fast memory service rate.

[4]We have also performed experiments with shorter intervals and different thresholds, but most of the workloads showed significantly lower performance since the remapping overheads were too large to amortize.

The swapping overhead may also be partially hidden by performing swap operations in a more sophisticated fashion. In our design, we already defer the write-backs of *swapped-out* pages since they are likely to be non-critical. More aggressively, we may delay the entire execution of swapping operations until the memory bus is expected to be idle (for a long enough time for swapping). We may also employ a mechanism similar to the critical block first technique; only the requested 64B block out of a 2KB segment is first swapped into fast memory, and other blocks are *gradually* swapped in when the bus is idle. These are all good candidates to further alleviate the swapping cost although they need more sophisticated hardware structures.

E. Sensitivity to Swap Granularity

In our mechanism, we manage remapping at a granularity of 2KB segments. This is the same as the row size in the stacked memory. This section discusses the sensitivity of PoM as the granularity varies from 128B to 4KB.

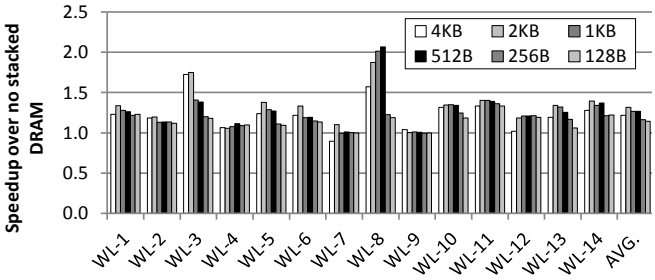

Figure 14. Speedup across different segment granularity.

Figure 14 shows the speedup across different segment granularity. Using a smaller segment size has the advantage of reducing the overhead of any individual swap. This allows for more frequent swaps and increases the rate of serving requests from fast memory. On the other hand, using a smaller segment size loses the benefits of prefetching that a larger granularity can provide. For the applications that have spatial locality, employing a larger swapping granularity may allow fast memory to service more requests, and the overall swapping overhead can also be smaller due to high row buffer hit rates. In addition, as the segment size decreases, the overhead of the remapping table increases since it needs to have more entries. This, in turn, reduces the effective coverage of the remapping cache. When deciding on the segment size, all these factors need to be considered. We chose a 2KB segment size since it achieved a decent speedup, while its overhead is still manageable.

F. Energy Comparison

Figure 15 shows the energy consumption of OS-managed and our proposed heterogeneous memory systems compared to no stacked DRAM. We compute the off-chip memory power based on the Micron Power Calculator [24] and the Micron DDR3 data sheet (-125 speed grade) [25]. Since no stacked DRAM data sheet is publicly available, we compute the stacked memory power based on the access energy and standby power numbers reported in [26].[5] The results show that PoM reduces energy per memory access by an average of 9.1% over the evaluated OS-managed policy. In general, PoM migrates more data blocks between fast and slow memory than OS-based migration, which increases the amount of energy used for data migration. However, the increased energy could be amortized by the increased number of hits in fast memory. More importantly, PoM's performance improvement reduces static energy in the memory system, which leads to significant energy savings.

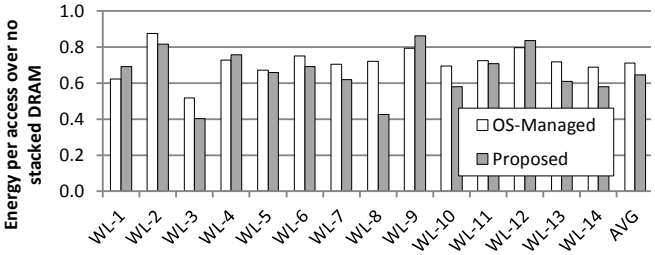

Figure 15. Energy comparison between OS-based migration and our PoM.

G. Comparison to Hardware-Managed Cache

Figure 16 compares two different DRAM cache implementations with two different implementations of PoM including a naive version of PoM (`Unmanaged`) and our proposal. `LH-Style` uses 64B lines similar to [6] but includes improvements found in subsequent work such as a direct-mapped organization [7] and a hit-miss predictor [8]. The `PoM-Style` cache uses 2KB cache lines and an improved replacement policy similar to our PoM proposal (Section IV-F). The performance benefits we observe for the `PoM-Style` cache over `LH-Style` result from the additional prefetching due to the large 2KB lines and our effective replacement policy.[6]

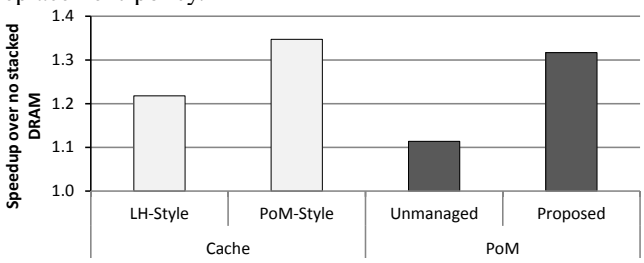

Figure 16. Comparison to hardware-managed caches.

The last two bars depict alternative implementations of PoM, both delivering the same capacity benefits. `Unmanaged` depicts a naive implementation of PoM without the benefits provided by the remapping cache and modified allocation algorithm (competing counters). The performance of our proposal is depicted on the far right. Since our experiments do not account for the performance impact of page faults, the best we can expect from our proposal is to match the performance benefits of `PoM-Style`. In fact, our proposal delivers

[5] Although our stacked memory is not identical to the one used in [26], we have verified that the conclusion remains the same across reasonably different power/energy numbers expected for die-stacked DRAM.

[6] The LH-style cache can be managed similarly to PoM-Style using prefetching and bypass techniques, thereby providing better performance.

a speedup close to what is achieved with PoM-Style, and it does this while avoiding data duplication and allowing running software to use all available fast and slow memory; thus, for running software that does not fit in slow memory, PoM would provide much higher speedups than caches when page fault costs are considered.

H. Sensitivity to Fast to Slow Memory Capacity Ratio

Figure 17 shows the average speedup of static mapping and our proposed scheme over no fast memory across different ratios of fast to slow memory capacities. As the ratio becomes larger, the percentage of memory requests serviced from fast memory is likely to increase, and we observe a large performance improvement even with static mapping. Although, the potential of our dynamic scheme would decrease as the ratio of fast to slow memory capacity increases, our results show that the proposed scheme still leads to non-negligible performance improvements over static mapping (a 13.8% improvement for a 1:1 ratio). For smaller ratios, we achieve much higher speedups compared to static mapping (e.g., we achieve a speedup 20% over static mapping when the ratio is 1:8).

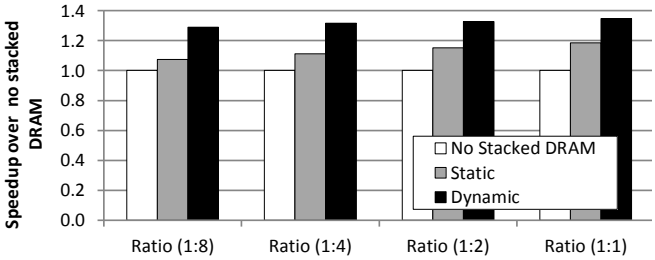

Figure 17. Speedups of static mapping and our scheme across different ratios of fast to slow memory.

I. Transparency to Virtual Memory Subsystem

PoM is transparent to current virtual memory/TLB subsystems and also ensures that a functionally correct memory image is maintained in the context of OS-invoked page migrations, copy-on-write, TLB shoot-downs, I/O requests, etc. This is because in today's systems all memory requests (even from other devices such as disk and network) must go through the system-on-chip memory controller to access memory. In PoM, these requests must all look up the SRC/SRT as they pass through the memory controller and before they access DRAM physical memory. For example, OS-invoked physical page migrations may result in page table updates and TLB shoot-downs, but since these involve the OS-maintained virtual-to-physical mappings and not the PoM-maintained physical page to segment mappings, these would be handled in a PoM system just like they would in a system without PoM. These events might make the physical segments allocated in stacked memory no longer hot (even dead); then, PoM would have no knowledge of the migration. However, since PoM uses a dynamic replacement algorithm and the competing counters are constantly comparing eviction and allocation candidates, the cold segments would quickly be replaced by hotter segments.

VII. Related Work

A large body of research has investigated non-uniform memory access (NUMA) architectures. Cache-coherent NUMA (CC-NUMA) architectures reduce traffic to remote nodes by holding remote data in the cache of each processing node [13]. Cache-only memory architectures (COMA) [27] use memory as a hardware-managed cache, like the ALL-CACHE design in the KSR1 system [28]. An S-COMA system allocates part of the local node's main memory to act as a large cache for remote pages [29]. Falsafi and Wood proposed Reactive NUMA (R-NUMA) that reacts to program behavior and enables each node to use the best of CC-NUMA or S-COMA for a particular page [15]. The Sun WildFire prototype showed that an R-NUMA-based design can significantly outperform a NUMA system [30]. Although the heterogeneous memory system analyzed in our paper has properties similar to a NUMA system (with variable memory latencies), our work differs from traditional NUMA research since we treat both the fast and slow memory as local to a node.

Many recent papers on heterogeneous memory systems have investigated the use of fast memory as a cache for the slow memory [5–10, 31]. The key difference between PoM and all previous work on DRAM caches is that PoM provides higher total memory capacity as compared to DRAM caches. Enabling PoM to maximize memory capacity requires quite different design approaches from previously described DRAM cache architectures, including support for complex swapping operations (Section IV-C) and memory permutation (Section IV-D for fast-swap) that results when different memory locations are swapped. The benefits of the additional memory capacity provided by PoM extend beyond the performance benefits harvested through reduced disk swapping, directly impacting one of the attributes consumers use when making a purchase decision.

Some prior work has explored managing heterogeneous memory systems using software. Loh et al. [19] studied the benefits and challenges of managing a die-stacked, heterogeneous memory system under software control. The authors discussed that even OS-managed heterogeneous memory systems require non-negligible hardware/software overheads for effective page activity monitoring and migration. RAMpage [32] proposed managing an SRAM-based last-level cache by software as the main memory of a machine, while using DRAM as a first-level paging device (with disk being a second-level paging device). Machanick et al. [33] showed that a RAMpage system can outperform a traditional cache system when the speed gap between the processor and DRAM increases. However, the RAMpage approach presents a significant practical challenge for operating systems that have minimum memory requirements, whereas our PoM approach maximizes the amount of memory that could be allocated by the OS. Lee et al. [17] proposed Tiered-Latency DRAM, which provides heterogeneous access latency, and introduced its use cases (caches or OS-visible memory). However, the use case as OS-visible memory was

briefly mentioned without details. As shown in other work, it is not trivial to effectively enable software-/hardware-managed heterogeneous memory systems, which we address in this work. Ekman and Stenstrom [34] also discussed a two-level main memory under software management. In contrast to our approach, these software-managed memory systems are less responsive to program phase changes.

Some papers have investigated hardware implementations for supporting page migrations in heterogeneous memories [16, 35]. The work that is most closely related to our proposal is the one from Dong et al. [16]. Their hardware-only implementation maintains a translation table in the memory controller, which keeps track of all the page remappings. To keep the table size small, their implementation uses large 4MB pages, which incurs both high migration latencies and increased bandwidth pressure on the slow memory. In comparison, our approach supports small page sizes by keeping the remapping table in the fast memory and caching the recent remapping table accesses in a small remapping cache. Ramos et al. [35] propose hardware support for OS-managed heterogeneous memories. In their work, the page table keeps a master copy of all the address translations, and a small remapping table in the memory controller is used to buffer only the recent remappings. Once the remapping table becomes full, the buffered remappings need to be propagated to the page table, requiring the OS to update the page table and flush all the TLBs. Thus, their approach requires costly OS interventions, which our technique avoids by maintaining page remappings in a dedicated hardware-managed remapping table in the fast memory.

VIII. Conclusion

Heterogeneous memory systems are expected to become mainstream, with gigabytes of fast memory on-die. For some market segments, using large, fast memory as a cache may be undesirable. This paper presents a Part-of-Memory (PoM) architecture that effectively combines slow and fast memory to create a single physical address space in an *OS-transparent* way. PoM employs unique designs (i.e., two-level indirection, permutation-based management, and competing counters) and provides substantial speedups over static mapping and alternative OS-based dynamic remapping. There likely remain other research opportunities in this domain. For example, to strike the right balance between capacity and bandwidth for running applications as in [36], we may want to dynamically configure fast memory between caches, PoM, or even software-managed memory. Studies on how to support such flexibility in heterogeneous memory systems could be one of the good research directions.

Acknowledgments

We thank Shih-Lien Lu, the anonymous reviewers, and the HPArch members for their useful feedback. Part of this work was conducted while Jaewoong Sim was on an internship at Intel Labs. We acknowledge the support of Intel, Sandia National Laboratories, and NSF CAREER award 1054830.

References

[1] R. Kalla *et al.*, "Power7: IBM's Next-Generation Server Processor," *IEEE Micro*, vol. 30, no. 2, 2010.
[2] N. Kurd *et al.*, "Haswell: A Family of IA 22nm Processors," in *ISSCC*, 2014.
[3] J. T. Pawlowski, "Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem," in *Hot Chips*, 2011.
[4] B. Black, "Keynote: Die Stacking is Happening," in *MICRO*, 2013.
[5] X. Jiang *et al.*, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA*, 2010.
[6] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *MICRO*, 2011.
[7] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-offs in Architecting DRAM Caches," in *MICRO*, 2012.
[8] J. Sim *et al.*, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *MICRO*, 2012.
[9] D. Jevdjic *et al.*, "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA*, 2013.
[10] J. Sim *et al.*, "Resilient Die-stacked DRAM Caches," in *ISCA*, 2013.
[11] Micron Technology, www.micron.com/products/hybrid-memory-cube.
[12] R. Hazra, "Accelerating Insights in the Technical Computing Transformation," in *ISC*, 2014.
[13] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *ISCA*, 1997.
[14] E. Hagersten *et al.*, "Simple COMA Node Implementations," in *HICSS*, 1994.
[15] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," in *ISCA*, 1997.
[16] X. Dong *et al.*, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *SC*, 2010.
[17] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[18] D. Lee *et al.*, "LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, 2001.
[19] G. H. Loh *et al.*, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *SHAW*, 2012.
[20] H. Kim *et al.*, *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*, Georgia Institute of Technology, 2012.
[21] T. Sherwood *et al.*, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.
[22] A. Snavely and D. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor," in *ASPLOS*, 2000.
[23] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, 2008.
[24] Micron Technology, "Calculating Memory System Power for DDR3," 2007.
[25] Micron Technology, "1Gb: x4, x8, x16 DDR3 SDRAM," 2006.
[26] B. Giridhar *et al.*, "Exploring DRAM Organizations for Energy-efficient and Resilient Exascale Memories," in *SC*, 2013.
[27] E. Hagersten *et al.*, "DDM - A Cache-Only Memory Architecture," *IEEE Computer*, vol. 25, 1992.
[28] S. Frank *et al.*, "The KSR 1: Bridging the Gap between Shared Memory and MPPs," in *Compcon Spring '93, Digest of Papers.*, 1993.
[29] A. Saulsbury *et al.*, "An Argument for Simple COMA," in *HPCA*, 1995.
[30] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," in *HPCA*, 1999.
[31] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *PACT*, 2014.
[32] P. Machanick, "The Case for SRAM Main Memory," *Computer Architecture News*, vol. 24, no. 5, 1996.
[33] P. Machanick *et al.*, "Hardware-software Trade-offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy," in *ASPLOS*, 1998.
[34] M. Ekman and P. Stenstrom, "A Cost-Effective Main Memory Organization for Future Servers," in *IPDPS*, 2005.
[35] L. E. Ramos *et al.*, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.
[36] J. Sim *et al.*, "FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion," in *ISCA*, 2012.