Utility-Based Hybrid Memory Management

Yang Li[†] Saugata Ghose[†] Jongmoo Choi[‡] Jin Sun[†] Hui Wang[⋆] Onur Mutlu[†‡]
[†]*Carnegie Mellon University* [‡]*Dankook University* [⋆]*Beihang University* [†]*ETH Zürich*

While the memory footprints of cloud and HPC applications continue to increase, fundamental issues with DRAM scaling are likely to prevent traditional main memory systems, composed of monolithic DRAM, from greatly growing in capacity. Hybrid memory systems can mitigate the scaling limitations of monolithic DRAM by pairing together multiple *memory technologies (e.g., different types of DRAM, or DRAM and non-volatile memory)* at the same level of the memory hierarchy. *The goal of a hybrid main memory is to combine the different advantages of the multiple memory types in a* cost-effective *manner while avoiding the disadvantages of each technology. Memory pages are placed in and migrated between the different memories within a hybrid memory system, based on the properties of each page. It is important to make intelligent page management (i.e., placement and migration) decisions, as they can significantly affect system performance.*

In this paper, we propose utility-based hybrid memory management (UH-MEM), a new page management mechanism for various hybrid memories, that systematically estimates the utility *(i.e., the system performance benefit) of migrating a page between different memory types, and uses this information to guide data placement. UH-MEM operates in two steps. First, it estimates how much a single application would benefit from migrating one of its pages to a different type of memory, by comprehensively considering access frequency, row buffer locality, and memory-level parallelism. Second, it translates the estimated benefit of a single application to an estimate of the* overall *system performance benefit from such a migration.*

We evaluate the effectiveness of UH-MEM with various types of hybrid memories, and show that it significantly improves system performance on each of these hybrid memories. For a memory system with DRAM and non-volatile memory, UH-MEM improves performance by 14% on average (and up to 26%) compared to the best of three evaluated state-of-the-art mechanisms across a large number of data-intensive workloads.

1. Introduction

Modern large-scale computing clusters continue to employ dynamic random access memory (DRAM) as the main memory system within each server. However, as the amount of memory consumed by the applications running on these clusters (e.g., high-performance computing workloads, large-scale data analytics) grows, traditional DRAM-based memory systems are unlikely to be able to keep up with this growth. DRAM scaling is expected to become increasingly difficult [90, 91] due to increasing cell leakage current [42, 65, 66, 97], reduced cell reliability [46, 76, 91, 113], and increasing manufacturing complexity [37, 41, 46, 74, 90, 91, 96, 107]. As a result, other memory solutions have emerged to offer low-latency, low-power, or high-capacity substrates without heavily relying on DRAM scaling. New DRAM products such as 3D-stacked DRAM [3,45,60,61,99], reduced-latency DRAM (RLDRAM) [80], and low-power DRAM (LPDRAM) [82] make use of novel DRAM circuit design, architectures, and interfaces to better cater to applications such as scientific computing, data mining, network traffic, and mobile computing. In addition, emerging non-volatile memory (NVM) technologies (e.g., PCM [53, 54, 55, 104, 124], STT-RAM [52], ReRAM [68], and 3D XPoint [83]) have shown promise for future main memory system designs to meet increasing memory capacity demands of data-intensive workloads. With projected scaling trends, NVM cells can be manufactured more easily at smaller feature sizes than DRAM cells, achieving high density and capacity [14, 15, 52, 53, 54, 55, 68, 104, 107, 120, 124, 131].

However, these new memory technologies are unlikely to *fully* replace commodity DRAM in main memory systems. For example, 3D-stacked DRAM is limited in capacity [12]. RLDRAM has higher cost-per-bit than commodity DRAM [8,49,58,59]. Most NVMs incur high access latency and high dynamic energy consumption, and some NVM technologies have limited write endurance. To address these weaknesses, *hybrid memory systems* or *heterogeneous memory systems*, comprised of both commodity DRAM and one of these alternative memory technologies, have been proposed. A hybrid memory system aims to combine the benefits of both of its component memory types in a *cost-effective* manner [104,126]. For example, commodity DRAM is faster than NVM, but has a higher cost per bit. A hybrid memory with both commodity DRAM and NVM utilizes a small amount of DRAM and a large amount of NVM, to provide the *illusion* that the system has large memory capacity (of NVM), and that all data can be accessed at low latency (of DRAM). Hybrid memory systems can potentially meet both the performance and memory capacity (as well as memory energy efficiency) needs of large-scale computing clusters [4, 5, 31, 33, 64, 73, 75, 98, 100, 104, 126].

In order to successfully deliver high memory capacity at low latency, hybrid memory systems must make intelligent data placement decisions, choosing whether each page should be placed in the high capacity memory or in the fast memory. Previous data management proposals for hybrid memories consider only a *limited* number of characteristics, using these few data points to construct a placement heuristic that is specific to the memory types being used in the system. For example, the majority of prior work on hybrid DRAM–

NVM main memory systems either treats DRAM as a conventional cache [104] or places data with high access frequency, high write intensity, and/or low row buffer locality in DRAM [20, 39, 106, 126, 129], while placing the remaining data in NVM, as the access latency of NVM is generally higher than that of DRAM [53, 104]. A mechanism for combining commodity DRAM with 3D-stacked DRAM organizes the faster 3D-stacked DRAM as a page-granularity cache of the commodity DRAM, but identifies and places only the cache blocks that will be accessed in 3D-stacked DRAM [38]. Work on combining RLDRAM with commodity DRAM identifies and places only critical data words into the RLDRAM to reduce access latency [11].

These heuristic-based approaches do not directly capture the *overall system performance benefits* of data placement decisions (as we will show in Section 3). Therefore, they can only *indirectly* optimize system performance, which sometimes leads to sub-optimal data placement decisions. For example, let us consider a memory manager that migrates memory pages that are accessed frequently [39] and that inherently have a high access latency (i.e., they have low *row buffer locality*) [126] from the slower NVM to the faster commodity DRAM. A page migration based on only these two heuristics may not improve system performance, if, for instance, accesses to the page being migrated are completely overlapped with other requests from the same application that continue to access the slower NVM. In such a case, the latency reduction for accesses to the migrated page would *not* reduce the application's execution time, as the application still needs to wait for the accesses to the slower NVM to complete. The example memory manager is unable to capture this overlap with its simple heuristics, and thus incorrectly decides to migrate the page in this example.

Our goal in this work is to devise a *generalized* mechanism that *directly* estimates the *overall system performance benefit* of migrating a page between any two types of memory, and places only the performance-critical data in the fastest memory within the hybrid main memory system. To this end, we propose *utility-based hybrid memory management* (UH-MEM), a new hardware mechanism that estimates the *marginal performance utility* of each page (i.e., the *system performance benefit* of migrating the page to a faster memory type), and migrates only those pages with the greatest utility. UH-MEM employs two steps. First, it determines how much migrating a page belonging to that individual application would improve the application's performance. To do this, UH-MEM uses a new *performance model* that considers several factors, including how frequently each page is accessed, whether row buffer locality impacts the performance benefits of migration, and how much the page access latency is hidden by overlapping requests (i.e., the level of *memory-level parallelism*, or MLP [13, 57, 87, 92, 93, 94]). Second, UH-MEM estimates how much the improvement of a single application's performance benefits the overall system performance, as different workloads have different amounts of impact on overall system performance. UH-MEM migrates those pages with the greatest estimated *system-level performance benefit* from slow memory into fast memory.

Key Results. We extensively evaluate UH-MEM using a wide range of hybrid memory configurations, and show that it is effective at improving system performance over state-of-the-art hybrid memory managers. We quantitatively show that for a memory system with both conventional DRAM and NVM, UH-MEM improves system performance by 14% on average (and up to 26%) compared to the best of three state-of-the-art mechanisms that we evaluate (a conventional cache insertion mechanism [104], an access frequency based mechanism [39, 106], and a row buffer locality based mechanism [126]), for a large number of data-intensive workloads. We also show that the hardware cost of UH-MEM is very modest ($\sim$40KB in our baseline system).

In this paper, we make three **main contributions**:

- We propose the first general *utility metric* to estimate the potential system performance benefit of migrating a page between the different memories within a hybrid main memory system. This utility metric represents the system performance benefit as a function of (1) an application's stall time reduction if the accessed page is migrated to a faster type of memory, and (2) how an improvement to a single application's stall time impacts overall system performance.
- We propose *a new performance model* that can be implemented in hardware, which comprehensively considers the access frequency, row buffer locality, and MLP of a page to systematically estimate an application's stall time reduction from migrating the page. This is the first work to consider MLP in addition to access frequency, row buffer locality, and write intensity, and to model the interactions between them, for page placement decisions.
- Based on our new metric and new performance model, we propose the first utility-based hybrid memory management mechanism, UH-MEM, which selectively places pages that are most beneficial to *overall* system performance in fast memory within a hybrid memory system. Our mechanism is general, and works with a wide variety of memory types that can be used in a hybrid memory system. We quantitatively demonstrate that UH-MEM outperforms three state-of-the-art hybrid memory management techniques.

2. Background

In this section, we provide background on the organization and management of hybrid memory systems. Figure 1 shows an example hybrid memory system. This hybrid memory system has two different types of memory, which we call Memory A and Memory B. One of these memories (we arbitrarily choose Memory A) is faster than the other, while the other memory (Memory B) has a greater capacity due to its higher density. The goal of a hybrid memory system is to provide the large main memory capacity of Memory B, while providing the fast access latencies of Memory A for memory accesses that affect execution time.

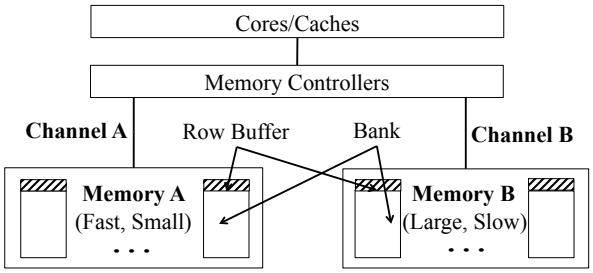

Figure 1: A typical hybrid memory system.

When a memory request is issued by a processor (e.g., the CPU), the *memory controllers* determine whether the request should be sent to Memory A or Memory B. Each memory has its own *memory channel* (i.e., a bus that connects the memory to its respective memory controller), and is internally organized similar to today's DRAM.[1] Each memory consists of multiple *banks*, where each bank is a two-dimensional array of memory cells organized into rows and columns. Each bank can operate in parallel, but all banks within a channel share the address, data, and command buses.

Within each bank, there is an internal buffer called the *row buffer*. When data is accessed from a bank, the entire row containing the data is brought into the row buffer. Hence, a subsequent access to data from the *same* row can be served from the row buffer and need not access the array. Such an access is called a *row buffer hit*. If a subsequent access is to data in a *different* row, the contents of the row buffer need to be written back to the row, and the new row's contents need to be brought into the row buffer. Such an access is called a *row buffer conflict* (or *row buffer miss*). A row buffer miss incurs a much higher latency than a row buffer hit. Previous works on hybrid memory systems observe that the latency of a row buffer hit is similar across memory types, while the latency of a row buffer conflict/miss is generally much higher in denser memories [53, 54, 55, 78, 79, 126]. The fraction of row buffer hits out of all memory accesses to a row is called *row buffer locality*. We can expect that migrating a page with *low* row buffer locality to the fast memory benefits performance, as a low-locality page experiences more row buffer misses, and such misses are serviced at a lower latency in the fast memory. Conversely, we can expect that migrating a page with *high* row buffer locality does *not* benefit performance much, as most of the accesses to such a high-locality page hit in the row buffer, and a row buffer hit has a similar latency in both the fast memory and the slow memory [126].

An important issue for a hybrid memory system is how to manage data stored in different memory devices. In our study, we adopt the configuration proposed by Qureshi et al. [104], and organize the fast, small memory (Memory A) as a cache for the pages in the large, slow memory (Memory B). We assume that all pages are initially in Memory B. Instead of unconditionally migrating a page when the page is accessed [69, 77, 102, 104], we *selectively* migrate pages into Memory A based on some metric, which is the utility of the page in our proposal. This migration may trigger the eviction of a victim page cached in Memory A, which is handled by the cache replacement policy of Memory A. We discuss our migration mechanism in Section 4.1. The migration process between memory devices is fully managed by hardware, and is transparent to the OS.

[1] We refer the reader to prior works for the detailed internal operation, organization, and control of DRAM [9, 10, 34, 46, 49, 59, 66, 88, 93, 111].

3. Motivation

In systems that can issue multiple memory requests in parallel (e.g., out-of-order execution processors, multicore processors, runahead processors), the number of cycles saved for a *single* memory request does *not* directly translate into a reduction in the application's execution time. In order to estimate the true *utility* of a page (i.e., the impact that migrating that page has on system performance), we need to estimate (1) by how much the latency reduction from migration would reduce the individual application's execution time (i.e., the application's *stall time reduction*), and (2) by how much the application's stall time reduction translates to an improvement in overall system performance (i.e., the *sensitivity* of overall system performance to each application's stall time). In this section, we first demonstrate that we need to *comprehensively* consider three major factors, i.e., access frequency, row buffer locality, and *memory-level parallelism* (MLP), to estimate the stall time reduction a page provides when migrated. These factors were not fully captured in prior works [20, 39, 106, 126, 129], none of which try to estimate the effect of migration on application or system performance. Then, we show that *overall* system performance exhibits different sensitivity to different applications' stall time reductions, and that we want to migrate pages from applications with high sensitivity to maximize overall system performance.

3.1. Comprehensive Stall Time Estimation of an Application

To the first order, an application's stall time reduction depends on two parts: (1) how much the latency for accessing the page can be reduced, and (2) how this latency overlaps with the latencies of other memory requests from the application. For the first part, since only the row buffer miss accesses can achieve shorter latency after the migration, we need to comprehensively consider access frequency and row buffer locality of the page (i.e., we can count the number of row buffer misses to the page) to estimate the latency reduction for the memory requests to the page. The second part depends on the parallelism of memory requests from an application (MLP). MLP is the number of concurrent outstanding requests (i.e., the in-flight memory requests that are yet to be completed) from the same application [13, 30, 87, 92, 93, 94]. In our mechanism, we consider the MLP for each page, and check how many concurrent requests from the same application typically exist when the page is accessed. If there are many concurrent requests, the access latency to the page is likely to overlap with the access latency to other pages, and therefore migrating the page to fast memory, while it may reduce its

access latency, will likely result in only a limited or small reduction in the application's stall time.

We illustrate this MLP effect using the conceptual example in Figure 2. Pages 0, 1, and 2 all have the same number of row buffer miss requests. Requests to Page 0 are not overlapped with other requests from the same application, while requests to Pages 1 and 2 are overlapped. We would like to see by how much the application's stall time would be reduced if we migrate each of these pages from slow memory to fast memory.

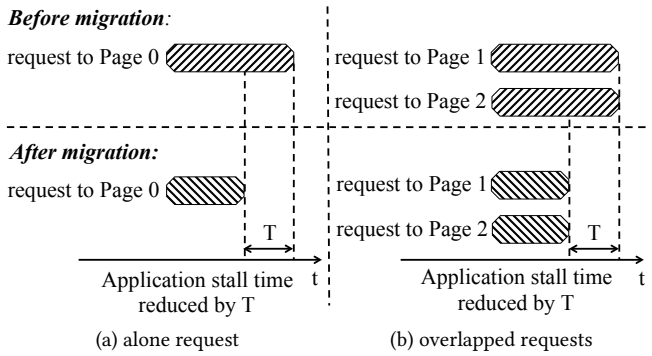

Figure 2: Conceptual example showing that the MLP of a page influences how much effect its migration to fast memory has on the application stall time.

Suppose we migrate Page 0 to fast memory (Figure 2a). As there is no other request that overlaps with the request to Page 0, the request to Page 0 is likely to be stalling at the head of the processor reorder buffer (ROB), which often stalls the entire application [29, 51, 87, 92, 94, 95, 103]. The requests to Page 0 will complete faster upon migration, thereby decreasing the application's stall time and thus being more likely to improve application performance [29, 51, 87, 92, 94, 95, 103]. On the other hand, if we migrate *both* Pages 1 and 2 to fast memory (Figure 2b), requests to both pages also complete faster, but the application's overall stall time will be reduced by roughly the same amount as that enabled by migrating *only* Page 0, since the access latencies to Pages 1 and 2 are overlapped. In other words, despite incurring double the number of migrations and consuming double the amount of limited fast memory capacity by migrating two overlapping pages (Pages 1 and 2), we achieve only the same performance benefit enabled by migrating only a single page that is serviced alone (Page 0). Unfortunately, without MLP, we are unable to build a comprehensive model that distinguishes between these two scenarios, and mechanisms that consider only row buffer locality and access frequency may migrate pages like Pages 1 and 2 that contribute less to reducing the application's stall time.[2]

Figure 3 shows the distribution of MLP across all memory pages for three representative benchmarks: *soplex*, *xalancbmk*, and *YCSB-B* [16, 35].[3] We can see that different pages within an application have very different MLP. Other benchmarks in our evaluation exhibit similar MLP diversity across their pages. Hence, we can take advantage of this diversity to optimize system performance.

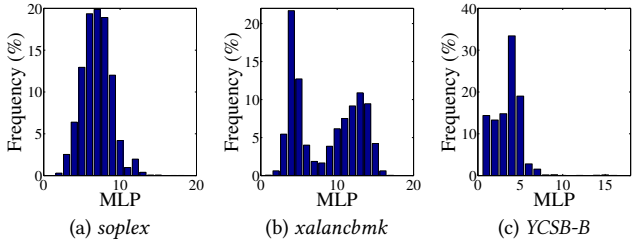

Figure 3: MLP distribution for all pages in three workloads.

In order to quantify the impact of different factors on an application's stall time, we measure the stall time contribution of each page (i.e., the time that the outstanding memory requests to the page cause the processor to stall) for every benchmark in our evaluation. Table 1 shows the correlation coefficients between the average stall time per page and three different page-level access characteristic metrics (i.e., access frequency, row buffer locality, and MLP, along with combinations of the three).[4] This shows that independently, access frequency, row buffer locality, and MLP all correlate *somewhat* with a page's stall time contribution. However, this correlation becomes *very strong* when we comprehensively consider *all three factors together* (correlation coefficient = 0.92). We see that the two factors considered together in prior work (access frequency and row buffer locality) [126] do not correlate nearly as strongly (correlation coefficient = 0.76). Therefore, we conclude that access frequency, row buffer locality, and MLP are all indispensable factors to comprehensively model the performance impact of data placement.

	AF	RBL	MLP
Correlation	0.74	0.59	0.54
	AF+RBL	**AF+MLP**	**AF+RBL+MLP**
Correlation	0.76	0.86	0.92

Table 1: Absolute Spearman correlation coefficients between the average stall time per page and different factors (AF: access frequency; RBL: row buffer locality; MLP: memory level parallelism). The correlation coefficients are between 0 and 1, where 0 = no correlation, and 1 = perfect correlation.

[2] In fact, if a mechanism migrates only one of the overlapping pages (either Page 1 or Page 2), it is unlikely that it will reduce stall time at all as the non-migrated page would still stall the CPU. A similar observation is made by Qureshi et al. in the context of caching [103].

[3] We run each workload separately on a system that is similar to the configuration shown in Section 5, though we use a single-core processor for the experiments shown here. When a page in the workload is accessed by a memory request, we measure how many outstanding memory requests with the same type (i.e., either read or write) exist in the workload, and use that number as the current MLP of the page. We then calculate the average MLP of each page, and report the distribution of average MLP across all of the pages in these figures.

[4] For each benchmark, we divide all of its pages into several bins, sorted by the values of the factors under consideration. We then calculate the average stall time per page for each bin. We analyze the correlation between the average stall time and the factors, and obtain the correlation coefficient. We report the average correlation coefficient over all of our benchmarks.

3.2. Estimating Effect on Overall System Performance

Prior proposals for hybrid memory page management that only use heuristics that are, as we have shown in Section 3.1, only somewhat correlated to application performance [11, 20, 38, 39, 104, 106, 126, 127, 128, 129] fail to capture how the stall time of a single application affects *overall system performance*. We find that this impact is *not uniform* across the applications within a multiprogrammed workload. There are several different metrics that can be used to express system performance, as has been discussed in a number of prior works [6, 26, 71, 112] (e.g., weighted speedup, harmonic speedup). These metrics express *overall* system performance by weighting the performance of each application within the workload differently, based on some application characteristics. For example, weighted speedup normalizes the performance of each application to its performance when running alone, in order to capture the effects of system interference between applications [26, 112]. For two applications with an equal amount of stall time reduction (in terms of absolute cycle count), the reduction for the application with a greater weight will result in a greater system performance improvement.

As prior page management mechanisms are oblivious to the unequal impact of application performance benefits on overall system performance, they can migrate pages that are less important for overall system performance into the fast memory. We, therefore, incorporate the relation between application performance and overall system performance directly into our mechanism, using application weighting to prioritize pages from applications that impact the overall system performance the most. In this work, we use *weighted speedup* [112], which has been shown to correspond to *system throughput* for multiprogrammed workloads [26]. However, system designers with other target objectives can use different system performance metrics, by simply modifying the system performance estimation hardware within our proposed mechanism.

4. UH-MEM: Utility-Based Hybrid Memory Management

In this section, we introduce *utility-based hybrid memory management* (UH-MEM). UH-MEM is a hardware mechanism that resides within the memory controller. It performs interval-based calculations to determine which pages should be migrated from slow memory to fast memory, where fast memory is treated as a set-associative (16-way) page cache with LRU cache replacement policy, similar to prior work [77, 104, 126]. During each interval (1 million cycles in our experiments, determined empirically), pages are selected for migration by UH-MEM, and a migration mechanism caches the data in the fast memory by copying the data first to the migration buffer in the memory controller, and then to the fast memory. Once a page is migrated to fast memory, it is inserted into a tag store within the memory controller. Whenever a request misses in the last-level on-chip cache, it looks up the tag store and the migration buffer, to see if the requested data resides in fast memory or in the migration buffer. The request is then dispatched to the appropriate location based on this lookup. As with on-chip caches, UH-MEM's operations are transparent to the OS.

4.1. Mechanism Overview

UH-MEM comprehensively estimates how the migration of each page would improve overall system performance, which we define as the *utility* of each page (see Section 3). The page utility calculation, as performed in hardware, is described in detail in Section 4.2. During each interval, when a page is accessed in slow memory, UH-MEM migrates the page to fast memory if its utility is greater than the *migration threshold*. It is not beneficial to move every accessed page into fast memory, because (1) migration operations take time to complete, and (2) doing so would cause the slow memory bandwidth to go unused. We include a mechanism to dynamically *set the migration threshold* at the end of each interval, which we discuss in Section 4.3.

When a page is selected for migration, we first check the tag store of the fast memory to see if we need to evict another page in the destination fast memory cache set. We implement a *migration buffer* within the memory controller to temporarily hold the migrating page(s). Each cache block in the buffer includes two migration status bits to determine where the cache block currently resides (i.e., in either of the memories, or in the buffer). The status bits allow UH-MEM to direct incoming memory requests for a migrating page to the correct place. After completing the data movement, the corresponding metadata information in the tag store is updated.

4.2. Computing Page Utility

The utility of a page depends on (1) the stall time reduction of an application due to migration of the page to the fast memory, and (2) the system performance sensitivity to the application.[5] Suppose that one page of Application i is migrated to fast memory, such that the application stall time is reduced by $\Delta Stall\ Time_i$. The utility of that page (U) can be expressed as:

$$U = \Delta Stall\ Time_i \times Sensitivity_i \quad (1)$$

4.2.1. Estimating Application Stall Time Reduction. The stall time reduction due to a page migration is dependent on two factors: (1) the access latency reduction for that page, and (2) the degree to which the page's access latency is masked (i.e., overlapped) by the access latency of other concurrent requests for the same application.

The degree to which a page's total access latency is reduced can be determined by using a combination of the page's access frequency and row buffer locality. If a page is migrated from slow memory to fast memory, the latency of row buffer

[5] Without loss of generality, we use the term "application" to refer to a hardware thread context executing an application.

misses decreases, while row buffer hits still achieve a similar latency. Therefore, the expected decrease in access latency is proportional to the total number of row buffer misses for that page, which is a function of access frequency and row buffer locality. We can estimate this decrease as:

$$\begin{aligned} \Delta Read\ Latency &= \#ReadMiss \times (t_{slow,read} - t_{fast,read}) \\ \Delta Write\ Latency &= \#WriteMiss \times (t_{slow,write} - t_{fast,write}) \end{aligned} \quad (2)$$

where $\#ReadMiss$ and $\#WriteMiss$ are the number of row buffer read and write misses, respectively, and $t_{fast,read}$, $t_{fast,write}$, $t_{slow,read}$, and $t_{slow,write}$ are the device-specific read/write latencies incurred on a row buffer miss for fast memory and slow memory, respectively.

In order to quantify the degree of access latency masking, we sample the total number of outstanding memory requests for that same application to model the "overlap effect." Specifically, we define the *MLP ratio* of an application to be the reciprocal of the outstanding memory request count.[6] Intuitively, if there are fewer outstanding requests, then there is less memory-level parallelism available to overlap the page's access latency. As such, we use the reciprocal of the number of outstanding memory requests so that the MLP ratio represents the fraction of the access latency that impacts the application's performance. During a sampling period t, the MLP ratio for an application with $N_{read,t}/N_{write,t}$ outstanding read/write requests is as follows, respectively for reads and writes:

$$MLPRatio_{read,t} = \frac{1}{N_{read,t}} \qquad MLPRatio_{write,t} = \frac{1}{N_{write,t}} \quad (3)$$

We can use the MLP ratio of the application to determine the MLP ratio for *individual pages*. For most applications, different pages do *not* typically have equal amounts of MLP. Therefore, we approximate an average MLP ratio *for each page* across all of the sampling periods that have taken place so far in the current interval. We compute two values, $PageMLPRatio_{read}$ and $PageMLPRatio_{write}$, which are the average MLP ratio of a page during the interval for outstanding read and write requests, respectively, to that page. We can model $PageMLPRatio_{read}$ and $PageMLPRatio_{write}$ as:

$$\begin{aligned} PageMLPRatio_{read} &= \frac{\sum_t MLPRatio_{read,t} \times m_{read,t}}{\sum_t m_{read,t}} = \frac{\sum_t \frac{m_{read,t}}{N_{read,t}}}{\sum_t m_{read,t}} \\ PageMLPRatio_{write} &= \frac{\sum_t MLPRatio_{write,t} \times m_{write,t}}{\sum_t m_{write,t}} = \frac{\sum_t \frac{m_{write,t}}{N_{write,t}}}{\sum_t m_{write,t}} \end{aligned} \quad (4)$$

To calculate $PageMLPRatio_{read}$, we start with the overall application MLP ratio at each sampling period t ($MLPRatio_{read,t}$). We determine the total contribution of the page to the application's MLP during sampling period t by multiplying $MLPRatio_{read,t}$ with the number of outstanding read requests during the sampling period to the page ($m_{read,t}$). We then sum up the page's MLP contributions over all of the sampling periods so far in the current interval, and divide it by the total number of outstanding read requests to the page during these sampling periods. This, in effect, gives us the average MLP contribution of each outstanding read request for the page. We repeat the same calculation for write requests.

We can now combine the latency reduction (Equation 2) and the average MLP ratio (Equation 4) to determine the stall time reduction for Application i as a result of migrating a particular page:

$$\begin{aligned} \Delta Stall\ Time_i = &\Delta Read\ Latency \times PageMLPRatio_{read} \\ &+ p \times \Delta Write\ Latency \times PageMLPRatio_{write} \end{aligned} \quad (5)$$

where p represents the probability that the write requests appear on the critical path. Prior work [130] has shown that this probability is dependent on an application's write access pattern, and is generally larger if the application has a large number of write requests. For simplicity, we choose to set $p = 1$, though using an online iterative approach to determine p [130] may yield better performance since it can enhance the accuracy of the stall time estimation.

Equation 5 shows that the stall time reduction due to a page migration from slow memory to fast memory can be determined by using a combination of access frequency, row buffer locality, and MLP for each page. Intuitively, a high access frequency and low row buffer locality increase the number of total row buffer misses, thus enlarging the benefits of migrating to fast memory. Likewise, poor MLP, with fewer concurrent outstanding requests, increases the average MLP ratio due to low likelihood of overlapping the request latency, and also increases the benefits from migration.

4.2.2. Estimating System Performance Sensitivity. For multiprogrammed workloads, we use the weighted speedup metric [27, 112] to characterize system performance.[7] For each application, the speedup component of Application i is the ratio of execution time when running alone, i.e., without interference from other applications ($T_{alone,i}$) to that when running together with other applications ($T_{shared,i}$):

$$System\ Performance = \sum_i Speedup_i = \sum_i \frac{T_{alone,i}}{T_{shared,i}} \quad (6)$$

[6] We calculate the MLP ratio separately for reads and writes, to account for their different behavior in main memory. While reads are often serviced as soon as possible (as they can fall along the critical path of execution), writes are deferred, and are eventually drained in batches [56, 110]. Distinguishing between reads and writes allows us to more accurately determine the MLP behavior affecting each type of request.

[7] UH-MEM can be adapted to use different system performance or fairness metrics [22, 24, 32, 47, 48, 86, 88, 93, 116, 117, 121, 125]. In order to support different system performance metrics, we can implement logic to estimate the sensitivity for each metric, and let the OS choose the most suitable metric to optimize based on the applications currently running within the system and the user's preferences.

When Application i migrates a page to fast memory, the speedup of that application improves by Δt:

$$Speedup_i' = \frac{T_{alone,i}}{T_{shared,i} - \Delta t} \tag{7}$$

Since the stall time reduction due to page migration is generally much smaller than the execution time ($\Delta t \ll T_{alone,i}$, $T_{shared,i}$), we can perform a Taylor expansion to find the change in speedup:

$$\begin{aligned} \Delta Speedup_i = Speedup_i' - Speedup_i &= \frac{T_{alone,i}\Delta t}{(T_{shared,i} - \Delta t)T_{shared,i}} \\ &\approx \frac{T_{alone,i}}{T_{shared,i}} \cdot \frac{\Delta t}{T_{shared,i}} = Speedup_i \times \frac{\Delta t}{T_{shared,i}} \end{aligned} \tag{8}$$

We defined the performance sensitivity of the system to an application in Section 3.1 as the measure of how the change in an application's stall time impacts the overall system performance. We can thus estimate it using Equation 9 (by plugging in Equation 8 at the appropriate place):

$$Sensitivity_i = \frac{\Delta Performance}{\Delta Stall\ Time_i} = \frac{\Delta Speedup_i}{\Delta t} = \frac{Speedup_i}{T_{shared,i}} \tag{9}$$

We calculate the performance sensitivity using an interval-based approach, where the speedup ($Speedup_i$) and execution time ($T_{shared,i}$) obtained in the last interval are used to estimate performance sensitivity in the current interval. The execution time of each application running on the system is equal to the length of an interval. We need to estimate the speedup of the application ($Speedup_i$) during the interval. This speedup estimate can be obtained by using prior proposals [22, 23, 84, 88, 118, 119]. These works consider the impact of memory interference and/or cache contention on the speedup of an application. In our implementation, we estimate speedup based on the approach in [88].

Equations 5 and 9 are combined using Equation 1 to give us the overall utility of migrating the page in question. A few measurements are required to obtain this utility calculation, and we discuss the implementation details of these mechanisms in Section 4.4.

4.3. Performing Page Migration

Algorithm 1 summarizes how UH-MEM decides which pages it should move to the fast memory. Whenever an outstanding memory request completes, UH-MEM (1) updates counters that hold statistics for the page accessed by the request, (2) recalculates the utility of the page, and (3) compares the calculated utility with the migration threshold. The page will only be migrated from slow memory to fast memory if the utility exceeds the migration threshold. At the end of each interval, UH-MEM adjusts the migration threshold to account for transient application behavior, and clears the page statistic counters.

Algorithm 1 Migrating pages with UH-MEM.

```
for every interval do
  for every completed memory request do
    Update the corresponding page's statistics counters
    Calculate the page's utility (Section 4.2)
    if the page's utility exceeds the migration threshold then
      Migrate the page to the fast memory
    end if
  end for
  if at the end of the interval then
    Adjust the migration threshold (Section 4.3)
    Estimate speedup for each application (Section 4.2.2)
    Reset all counters to zero
  end if
end for
```

A key question is how to determine this migration threshold. We choose to use a hill climbing based approach to determine this threshold dynamically, similar to the policy used by Yoon et al. [126]. We use the total stall time of all applications in each interval to reflect the system performance. At the end of each interval, the total stall time is recalculated. We then compare the current total stall time with the total stall time from the previous interval, and determine whether the previous threshold adjustment yielded a system performance improvement. If the total stall time of the current interval is lower (meaning that the threshold adjustment improved system performance), we continue to adjust the threshold in the same direction. Otherwise, since the previous adjustment degraded performance, we move the threshold in the opposite direction.

4.4. Hardware Structures

UH-MEM performs the calculations described in Section 4.2 in hardware. We first discuss the various hardware components required for UH-MEM to calculate the MLP ratios and page utility. Then, we summarize the total cost of the hardware.

4.4.1. MLP Ratio Calculation. To calculate the MLP ratios from Equation 4, we must maintain four temporary counters for every page with outstanding requests in the memory controller. Two of the counters, $MLPAcc_{read}$ and $MLPAcc_{write}$, accumulate the numerator from Equation 4, while the other two counters, $MLPWeight_{read}$ and $MLPWeight_{write}$, accumulate the denominator of the equation, as follows:

$$\begin{aligned} MLPAcc_{read} &= \sum_t \frac{m_{read,t}}{N_{read,t}} & MLPWeight_{read} &= \sum_t m_{read,t} \\ MLPAcc_{write} &= \sum_t \frac{m_{write,t}}{N_{write,t}} & MLPWeight_{write} &= \sum_t m_{write,t} \end{aligned} \tag{10}$$

For every sampling period (30 cycles in our experiments), we monitor both the outstanding read/write requests N_{read} and N_{write} for each application, as well as the outstanding requests m_{read} and m_{write} for each page, and update the corresponding counters.

When all the outstanding requests to a page have completed, the contents of the page's temporary counters are added to its corresponding counters in a statistics store (i.e., *stats store*), and are then reset. The stats store is a 32-way set-associative cache with LRU replacement policy, residing in the memory controller. Each stats store entry corresponds to a page, and consists of six counters that record the number of row buffer misses, the sum of weighted MLP ratios (*MLPAcc*), and the sum of weights for the MLP ratios (*MLPWeight*) for read/write requests. We can use the ratio of *MLPAcc* to *MLPWeight* to calculate the average MLP ratio of the page (*PageMLPRatio*), respectively for read and write requests. When a page in slow memory is accessed, if it has an existing entry in the stats store, the content of its entry is updated; otherwise, an entry is allocated, which may evict the entry of the least recently used page within the set. The access latency to the stats store is not on the critical path, as we update the stats store in the background.

When a system has multiple memory controllers, the stats store and the counters used to calculate MLP ratios need to be shared by these memory controllers. Different memory controllers need to communicate with each other to maintain the information, such as the number of outstanding requests, as done in prior works [17, 36, 47, 85, 86].

4.4.2. Utility Calculation for Shared Pages. For pages shared by multiple applications, we can use separate entries in the stats store to record the statistical information of the page with respect to each application. We can use our previous method to calculate the page utility for each application, and then add these utility values to obtain the aggregate utility for the page. The insight is that the total system performance improvement correlates with the sum of the performance improvement of each application. Therefore, summing up the page utility for each application (i.e., its performance improvement) should reflect the system performance improvement.

4.4.3. Hardware Cost. Table 2 describes the main hardware costs for UH-MEM. The largest component is the stats store. We use a 2048-entry stats store (organized as 32-way set-associative cache), as it leads to negligible performance degradation compared with an unlimited-size stats store. The main hardware cost of UH-MEM is 42.87KB,[8] which is only approximately 2% of our baseline system's L2 cache size.

UH-MEM also requires hardware logic to calculate the MLP ratios. For each page with outstanding requests in slow memory (96 at most; limited by the read request queue size and write buffer), we need to perform 4 25-bit additions and 2 fast divisions every 30 cycles to compute the MLP ratios.[9] We achieve this by pipelining the logic, and making it 3-way superscalar. We can implement fast division using a 32×32 ROM table that contains the precomputed results of the division, since both the numerator and denominator of the division are limited by the MSHR size of the last-level cache. As each quotient is 10 bits wide, the total size of such a ROM table is 1.25KB.

UH-MEM does *not* require any modifications to the operating system to support page migration. This is because UH-MEM does not use the virtual or physical address of a page to determine whether the page resides in fast memory or slow memory. Instead, UH-MEM uses a dedicated hardware tag store in the memory controller to determine whether the page has been migrated to the fast memory.

5. Evaluation Methodology

Similar to prior works [39, 104, 106, 126], we evaluate our proposed UH-MEM mechanism using a cycle-accurate x86 multicore simulator [2], whose front end is based on Pin [70]. We released our simulator [2, 109]. This in-house developed simulator is similar to Ramulator [1, 50], which is a widely-accepted open-source multicore simulator that models the main memory system in detail. In our simulator, page migrations between fast and slow memories are modeled as additional read requests to the memory device where the page is currently located, to read the entire page from it, followed by additional write requests in the destination memory device to write the entire page. The latency for determining whether a page resides in fast or slow memory is modeled as six cycles. Table 3 summarizes the major parameters of the baseline system consisting of DRAM and NVM in our

[8] This does not include the hardware used to determine whether a page resides in fast memory or slow memory, as this hardware is required by most hybrid memory management mechanisms [104, 106, 126], and the implementation of UH-MEM is orthogonal to the implementation of this structure.

[9] We determined all values empirically and did not optimize heavily. Reduction in hardware cost is possible with careful optimization.

Name	**Purpose**	**Structure** (number of bits in parentheses)	**Size**
Stats store	Tracks statistical information for recently-accessed pages	2048 entries; each entry consists of read row buffer miss count (14), write row miss count (14), $MLPAcc_{read}$ (30), $MLPAcc_{write}$ (30), $MLPWeight_{read}$ (21), $MLPWeight_{write}$ (21) and page number tag (30)	40.00KB
Counters for outstanding pages in slow memory	Records updates of *MLPAcc* and *MLPWeight* for pages with outstanding requests	For each page with outstanding requests in slow memory (96 at most), $MLPAcc_{read}$ (30), $MLPAcc_{write}$ (30), $MLPWeight_{read}$ (21), $MLPWeight_{write}$ (21) and page number (36)	1.62KB
ROM table for MLP ratios	Stores precomputed results of division used to calculate MLP ratios	32 x 32 entries; each entry consumes 10 bits	1.25KB
Total Hardware Cost (for our evaluated system in Table 3)			42.87KB

Table 2: Main hardware cost of UH-MEM.

Processor	8 cores, 2.67GHz, 3-wide issue, 128-entry instruction window
L1 Cache	32KB per core, 4-way, 64B cache block
L2 Cache	256KB per core, 8-way, 32 MSHR entries per core, 64B cache block
Fast Memory Controller	64-bit channel, 64-entry read request queue, 32-entry write buffer, FR-FCFS scheduling policy [108, 132]
Slow Memory Controller	64-bit channel, 64-entry read request queue, 32-entry write buffer, FR-FCFS scheduling policy [108, 132]
Baseline Fast Memory System	512MB DRAM, 1 rank (8 banks), t_{CLK}=1.875ns, t_{CL}=15ns, t_{RCD}=15ns, t_{RP}=15ns, t_{WR}=15ns, array read (write) energy = 1.17 (0.39) pJ/bit, row buffer read (write) energy = 0.93 (1.02) pJ/bit
Baseline Slow Memory System	16GB NVM, 1 rank (8 banks), t_{CLK}=1.875ns, t_{CL}=15ns, t_{RCD}=67.5ns, t_{RP}=15ns, t_{WR}=180ns, array read (write) energy = 2.47 (16.82) pJ/bit, row buffer read (write) energy = 0.93 (1.02) pJ/bit

Table 3: Baseline system parameters.

evaluation. The detailed DRAM and NVM timing and energy parameters are based on prior studies [53, 54, 78, 79, 81]. We calculate the static power of the hybrid memory system to be 5.6W [53].

In order to evaluate different types of hybrid memory systems, such as DRAM–RLDRAM and DRAM–NVM memories, we vary the size of the fast memory and the read/write write latency ratios of slow memory to fast memory. We also measure the performance of our evaluated page placement mechanisms under these different configurations.

5.1. Workloads

We use 30 benchmarks chosen from SPEC CPU2006 [35] and the Yahoo Cloud Serving Benchmark (YCSB) suite [16]. We classify them as memory-intensive or non-memory-intensive based on their last level cache misses per 1K instructions (MPKI) when running alone. Each experiment runs an eight-application *workload* on the system, with one application running on each core. The memory intensity category of the workload is determined by the percentage of memory-intensive benchmarks within the workload. For example, a workload has 75% intensity if it consists of six memory-intensive benchmarks and two non-memory-intensive benchmarks. We generate 40 workloads, eight for each category of workload memory intensity (0%, 25%, 50%, 75%, 100%). In each experiment, every benchmark was warmed up for 500 million instructions, and then executed for another 500 million instructions. A benchmark in a multiprogrammed workload is restarted after it completes until all the benchmarks in the workload complete once.

5.2. Metrics

We use weighted speedup (*WSpeedup*) [26, 112] and maximum slowdown (*MaxSlowdown*) [6, 17, 18, 43, 44, 47, 48, 86, 116, 117, 119, 121, 123] to evaluate system performance and unfairness, respectively, using the equations shown below. N is the number of cores; $IPC_{alone,i}$ and $IPC_{shared,i}$ are the instructions completed per cycle (IPC) when Application i is running alone and running with other applications, respectively. Weighted speedup (see Section 4.2) first weights the performance of each application (when it is running with others; $IPC_{shared,i}$) by the reciprocal of its performance while running alone ($IPC_{alone,i}$), reflecting the speedup of the application. Then, weighted speedup sums up the speedup of all the applications, reflecting the overall system performance. Weighted speedup is a widely-used multiprogrammed system performance metric in computer architecture evaluation [26]. It quantifies system throughput [26]. For unfairness, we use *maximum slowdown* to quantify the worst-case slowdown of any application in a multiprogrammed workload. Both weighted speedup and maximum slowdown use normalized IPC ratios, instead of the IPC itself, to avoid biasing either metric in favor of high-IPC or low-IPC applications.

$$WSpeedup = \sum_{i=0}^{N-1} \frac{IPC_{shared,i}}{IPC_{alone,i}}$$

$$MaxSlowdown = max\left(\frac{IPC_{alone,i}}{IPC_{shared,i}}\right)$$

6. Experimental Results

We evaluate our proposed UH-MEM mechanism across a wide variety of system configurations, covering several fast memory sizes and latency ratios of slow memory to fast memory. Throughout our evaluation, we compare UH-MEM to three other state-of-the-art mechanisms:

- *ALL*: a conventional cache insertion mechanism. This mechanism treats fast memory as a cache to slow memory, and inserts all the pages accessed in slow memory into fast memory using the LRU replacement policy. This is similar to the proposal by Qureshi et al. [104].
- *FREQ*: an access frequency based mechanism. This mechanism migrates pages with high access frequency to fast memory. It is similar to two proposals that try to improve the temporal locality in fast memory and reduce the number of accesses to slow memory [39, 106].
- *RBLA*: a row buffer locality based mechanism [126]. This mechanism migrates pages that have experienced a large number of row buffer misses in slow memory to fast memory. The intuition is that only the latency of row buffer miss requests can be reduced when the page is migrated to fast memory.

6.1. Results on the Baseline System Configuration

Figure 4 shows the normalized weighted speedup of the four evaluated mechanisms on the baseline system configuration, averaged for each workload intensity category. UH-MEM outperforms the best previous proposal, RBLA, in all workload categories with non-zero memory intensity. For the most memory-intensive category, UH-MEM provides a 14%

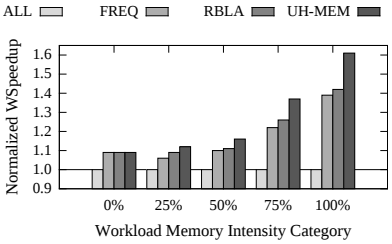

Figure 4: Normalized weighted speedup for the baseline configuration.

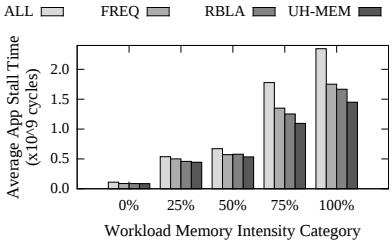

Figure 5: Average application stall time for the baseline configuration.

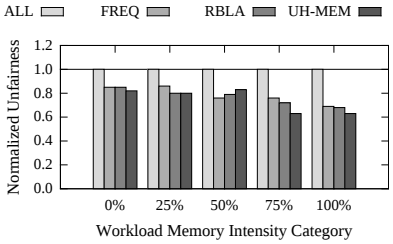

Figure 6: Normalized unfairness for the baseline configuration.

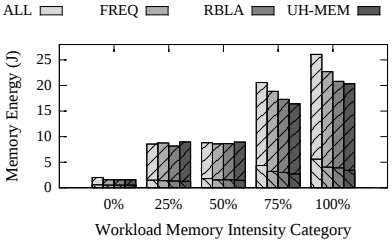

Figure 7: Memory energy consumption for the baseline configuration.

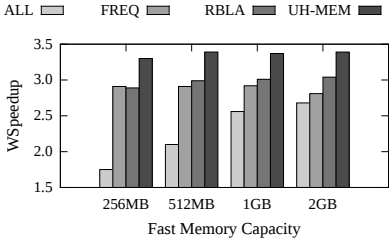

Figure 8: Weighted speedup for various fast memory sizes.

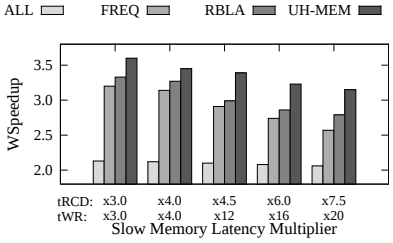

Figure 9: Weighted speedup for various slow-to-fast memory latency ratios for t_{RCD} and t_{WR}.

average performance improvement over RBLA. The maximum performance gain of UH-MEM over RBLA for a single workload is 26%. UH-MEM's performance advantage is twofold. First, UH-MEM not only considers the latency of each individual request (as FREQ and RBLA do), but also takes into account the memory-level parallelism between requests to estimate each request's individual contribution to the application's overall stall time. Therefore, UH-MEM can reduce stall time more effectively compared with those prior proposals by selecting and caching those pages that are more likely to stall the processor. This is demonstrated by Figure 5, which shows that each application within a workload stalls for less with UH-MEM than with RBLA. Second, UH-MEM is aware of which applications impact the system performance the most as it estimates system performance sensitivity to different applications, and prioritizes page migrations from those applications that are likely to benefit system performance the most. Figure 6 shows the normalized unfairness of the four evaluated mechanisms on the baseline system configuration. We can see that UH-MEM achieves equivalent or improved fairness compared to all prior proposals.

We also study the energy efficiency of the four mechanisms on the baseline system configuration. Figure 7 shows the memory energy consumption of the four mechanisms on workloads with varying memory intensities. We observe that energy consumption grows with the memory intensity of the workload. Compared to prior mechanisms, UH-MEM consumes similar energy for non-memory-intensive workloads, and uses less energy for memory-intensive workloads. For the memory-intensive workloads, UH-MEM reduces static energy consumption as a result of its shorter execution time. UH-MEM also reduces the dynamic energy consumed due to page migrations, as it selectively migrates the important pages to DRAM instead of migrating less important pages as the baseline mechanisms do.

We conclude that UH-MEM improves performance and lowers energy consumption compared to three state-of-the-art hybrid memory management mechanisms, because it can effectively gauge the system performance benefit of each page migration.

6.2. Sensitivity to Fast Memory Size

The fast memory size determines the room for performance optimization in hybrid memory systems. A larger fast memory can allow more pages to migrate from slow memory, thereby likely offering greater system performance. However, the fast memory size, in practice, cannot be too large, and therefore can limit the scalability of hybrid memory systems. In this section, we evaluate how each mechanism performs across a range of fast memory sizes (256MB, 512MB, 1GB, and 2GB).

Figure 8 shows the weighted speedup of workloads with 100% memory intensity under various fast memory sizes. We observe that system performance increases with fast memory size. Under the four evaluated sizes, UH-MEM outperforms RBLA by 14%, 14%, 12%, and 12%, respectively. Even for a 256MB fast memory, which offers less opportunity for optimization, UH-MEM achieves a weighted speedup of 3.30, which is larger than RBLA's weighted speedup of 3.04 *for a 2GB fast memory*. In other words, UH-MEM can exceed RBLA's performance even with only an eighth of the fast memory capacity. This implies that by estimating the system performance benefit of each page and selectively placing only critical pages in fast memory, UH-MEM can greatly shrink the fast memory size (while achieving higher performance), and thereby improve hybrid memory scalability.

6.3. Sensitivity to Slow-to-Fast Memory Latency Ratio

We vary the slow memory access latency to evaluate the sensitivity of our proposed mechanism to the latency ratio of slow memory to fast memory. In memory, row activation time t_{RCD} and write recovery time t_{WR} are two important timing parameters that determine the read/write access latency [8, 10, 49, 58, 59, 62, 79]. t_{RCD} specifies the latency between the row activate and buffer read/write commands, while t_{WR} specifies the latency between the array write and precharge commands. To evaluate the effectiveness of our mechanisms on hybrid memories with multiple types of DRAM, we set both the t_{RCD} and t_{WR} latencies of slow memory to be 3 and 4 times their latencies for fast memory, which is in the typical range of contemporary DRAM products [11, 49, 58, 80, 101]. To evaluate hybrid DRAM–NVM systems, we set the t_{RCD} latency of slow memory to be 4.5, 6, and 7.5 times that of fast memory latency, while setting its t_{WR} latency to be 12, 16, and 20 times, reflecting the generally more expensive write latency of NVM that is present in PCM [53] and STT-RAM [52].

Figure 9 shows the absolute weighted speedup under the different slow-to-fast memory access latency ratios. We make two observations from the figure. First, as t_{RCD} and t_{WR} increase, system performance gradually decreases. This is because the increased access latency increases the processor stall time, and in turn decreases system throughput. Second, the performance of ALL does not significantly change. This is because ALL tries to insert the whole working set into fast memory, which leads to very significant fast memory contention. Unlike the other mechanisms, this contention, and not the slow memory latency, is the bottleneck for ALL. For the other mechanisms, since they can perform some form of load balancing between fast and slow memory (through the dynamic adjustment of the migration threshold), their main bottleneck is the latency asymmetry between the different memory devices, and, as a result, their absolute performance improves when slow memory latency decreases. For our five latency configurations, UH-MEM improves weighted speedup by 8%, 6%, 14%, 13%, and 13%, respectively, over RBLA.

We conclude that UH-MEM provides performance improvements over state-of-the-art hybrid memory management techniques for a wide range of hybrid memory configurations, whether they be different types of DRAM or DRAM–NVM.

7. Related Work

To our knowledge, this work provides (1) the first utility metric for hybrid memory systems, which quantifies the system performance benefit of placing individual pages in fast memory; and (2) the first comprehensive performance model for doing so. The most closely related work is a set of proposals on data placement in DRAM–NVM or heterogeneous DRAM memory systems, which we briefly review in this section.

7.1. Hybrid DRAM–NVM Memory Systems

Prior works on hybrid DRAM–NVM memory systems propose to place pages that are recently accessed [104], or pages with high access frequency, high write intensity, and/or low row buffer locality [106,126,129] in DRAM. These prior works use only a few aspects of the memory characteristics of a page to construct a heuristic that optimizes access latency, instead of directly estimating the overall system performance benefit of migrating a page. As discussed in Section 3, improving the access latency of a page using the heuristics proposed by prior work does not necessarily lead to an improvement in system performance. In order to maximize system performance, it is important to estimate the likely performance benefit of placing each page in DRAM, which is demonstrated by our performance evaluation results of UH-MEM.

Agarwal et al. [5] propose a software-based approach to manage huge pages (e.g., 2MB pages) in hybrid memory systems. They propose to profile the memory access patterns of huge pages, and to use these patterns to guide page migration between DRAM and NVM. Although our work does not explicitly consider huge pages, our proposals can be extended easily to cover huge page migration, by treating each huge page as a series of regular-sized pages (e.g., 4KB pages), and collecting characteristics and making migration decisions for each of these regular-sized pages independently.

Dulloor et al. [21] propose a programmer-guided data placement tool. This tool requires (1) programmers to modify the source code, and (2) a representative profiling run of the application prior to making placement decisions. A major limitation of the tool is that it cannot be used when the source code is not accessible, or when a priori profiling is infeasible. Our software-transparent UH-MEM mechanism overcomes such limitations, and it can also potentially be used together with programmer-guided data placement.

Several works on hybrid memory management are orthogonal to our work. Peña and Balaji [100] propose a profiling tool to assess the impact of distributing memory objects across the multiple memory devices in a hybrid memory. Bock et al. [7] propose a scheme to migrate multiple pages concurrently between different memory devices without significantly affecting memory bandwidth. Gai et al. [28] propose a data placement scheme that optimizes the energy consumption of hybrid memory systems. Liu et al. [67] propose a scheme that jointly manages the cache, memory channels, and DRAM/NVM banks. Ideas from all these works can be combined with UH-MEM for better performance and efficiency.

7.2. Heterogeneous DRAM Memory Systems

Various techniques are proposed to combine multiple different types of DRAM for better performance, capacity, and efficiency [11, 12, 39, 72, 101, 127, 128]. Jiang et al. [39] propose to cache only hot pages in an on-chip DRAM cache, to overcome the off-chip DRAM bandwidth bottleneck. Chatterjee et al. [11] observe that the first word of cache blocks

is usually critical to the system performance, and propose to store only these words in fast DRAM. Luo et al. [72] propose heterogeneous-reliability memory, where multiple different types of DRAM with different reliability characteristics are used to improve system cost and efficiency. Chou et al. [12] and Yu et al. [127, 128] investigate how to utilize on-chip and/or in-package DRAM as part of the main memory address space. UH-MEM is complementary to these proposals.

Phadke and Narayanasamy [101] propose to classify applications as latency-sensitive, bandwidth-sensitive, or insensitive-to-both based on the MLP property of applications. To estimate MLP, they use an offline approach to profile applications in the compilation stage. Compared with this method, the MLP estimation approach in UH-MEM exhibits two major differences: (1) UH-MEM estimates MLP using an online approach that covers the dynamic events taking place during program execution; and (2) UH-MEM considers the MLP effects at a page granularity, and differentiates between pages with diverse MLP properties within the same application. Thus, UH-MEM works for a much broader range of applications, including those that we cannot profile a priori, and those that exhibit diverse MLP behavior across different pages.

7.3. Other Related Work

Several other works take advantage of the concept of memory-level parallelism to perform resource management [18, 19, 25, 57, 85, 87, 89, 92, 93, 103, 105, 122]. For example, Qureshi et al. [103] propose an on-chip cache replacement policy that tends to evict cache blocks that are serviced with larger MLP. The context of this work is different from ours: it targets on-chip cache replacement, while our work targets off-chip hybrid memory page placement. Hybrid memory placement is a more complex problem with a much larger design space, with two key differences from on-chip DRAM caching. First, for on-chip DRAM caches, retrieving data from the cache is clearly preferred over retrieving data from main memory, due to the high off-chip communication latency. If it were possible, those systems would prefer that *all data* be kept in the on-chip cache. In contrast, both our fast and slow memory are off-chip, with their row buffer hits having *identical* access latencies. Since the fast and slow memory have separate data channels, our partitioning mechanism also performs load balancing: some of our applications never fill up the fast memory in order to exploit the available slow memory bandwidth. Second, in some memories (e.g., PCM [53], STT-RAM [52]), writes require a longer latency than reads. Therefore, we need to consider the write intensity of a page when we make page placement decisions in hybrid memory, whereas on-chip DRAM cache management policies do not have to consider this due to DRAM's uniform read and write latencies. All these reasons make the decision space of hybrid memory much more complex than that of on-chip caches.

Prior work uses load criticality to make memory scheduling and caching decisions, by estimating or measuring the importance or processor stall time of each load request on processor performance [19, 29, 40, 114, 115]. These mechanisms are not designed for and hence do not consider the intricacies of hybrid memory systems. For example, characterizing stalls based only on load requests is problematic for hybrid memory management, as these mechanisms cannot correctly account for (1) stalls due to store requests (e.g., stalls due to a long memory write queue drain), or (2) the increased write latency in many types of NVM. As a result, load-criticality-based page placement may not correctly identify the pages that are most beneficial to migrate to fast memory. UH-MEM avoids this problem by directly estimating the performance impact of writes independently from the impact of reads.

8. Conclusion

Hybrid memory systems are a cost-effective approach to significantly increasing memory capacity and thus delivering high memory performance for data-intensive workloads. The ability to achieve high performance is highly dependent on data placement decisions made by a hybrid memory manager. We propose a utility-based hybrid memory management mechanism (UH-MEM), the first mechanism to *quantitatively estimate* the system performance benefit of migrating a page between different memory types within a hybrid memory system. UH-MEM consists of two major steps, which are two new performance models. First, it systematically estimates the application stall time reduction due to placing the page in fast memory versus slow memory. Second, it determines the sensitivity of system performance to each application, which represents the amount by which each application affects overall system performance. Based on the two models, UH-MEM migrates pages with high estimated system performance improvement to fast memory.

Our experimental results show that UH-MEM improves the system performance and reduces energy consumption over three state-of-the-art hybrid memory management proposals for a wide range of hybrid memory configurations. For a DRAM–NVM hybrid memory system, UH-MEM improves the system performance by 14% on average (and up to 26%) over the best of the three state-of-the-art management proposals. We conclude that the new utility metric and the new utility-based hybrid memory mechanism proposed in this paper can enable an effective approach to managing future hybrid memory systems, and hope that our proposal engenders more research in accurate performance estimation of such complex hybrid memory systems.

Acknowledgments

We thank the anonymous reviewers of IEEE Cluster 2017, ICCD 2016, ISCA 2016, PACT 2015, ISCA 2015, HPCA 2015, and MICRO 2014 for their comments. We especially thank the anonymous reviewers of IEEE Cluster 2017 for their constructive and insightful comments. An earlier version of this work was posted on arXiv in 2015 [63]. This work is supported in part by the NSF, SRC, ISTC-CC, and industrial partners of the SAFARI Research Group, especially Google, Intel, Samsung, and VMware.

References

[1] "Ramulator," https://github.com/CMU-SAFARI/ramulator, 2016.

[2] "Utility-Based Hybrid Memory Management Simulator," https://github.com/CMU-SAFARI/UHMEM, 2017.

[3] Advanced Micro Devices, Inc., "High-Bandwidth Memory (HBM): Revinventing Memory Technology," https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf, 2015.

[4] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *ASPLOS*, 2015.

[5] N. Agarwal and T. F. Wenisch, "Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory," in *ASPLOS*, 2017.

[6] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and Stretch Metrics for Scheduling Continuous Job Streams," in *SODA*, 1998.

[7] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, "Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory," in *ICCD*, 2016.

[8] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[9] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[10] K. K. Chang, A. G. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[11] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.

[12] C. C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *MICRO*, 2014.

[13] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.

[14] K. C. Chun, H. Zhao, J. Harms, T.-H. Kim, J.-P. Wang, and C. Kim, "A Scaling Roadmap and Performance Evaluation of In-Plane and Perpendicular MTJ Based STT-MRAMs for High-Density Cache Memory," *JSSC*, 2013.

[15] S. Chung, K.-M. Rho, S.-D. Kim, H.-J. Suh, D.-J. Kim, H. Kim, S. Lee, J.-H. Park, H.-M. Hwang, S.-M. Hwang, J. Y. Lee, Y.-B. An, J.-U. Yi, Y.-H. Seo, D.-H. Jung, M.-S. Lee, S.-H. Cho, J.-N. Kim, G.-J. Park, G. Jin, A. Driskill-Smith, V. Nikitin, A. Ong, X. Tang, Y. Kim, J.-S. Rho, S.-K. Park, S.-W. Chung, J.-G. Jeong, and S. J. Hong, "Fully Integrated 54nm STT-RAM with the Smallest Bit Cell Dimension for High Density Memory Application," in *IEDM*, 2010.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.

[17] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.

[18] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.

[19] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aérgia: Exploiting Packet Latency Slack in On-Chip Networks," in *ISCA*, 2010.

[20] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *DAC*, 2009.

[21] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *Eurosys*, 2016.

[22] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.

[23] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.

[24] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," in *MICRO*, 2009.

[25] S. Eyerman and L. Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," in *HPCA*, 2007.

[26] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.

[27] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *IEEE CAL*, 2014.

[28] K. Gai, M. Qiu, H. Zhao, and L. Qiu, "Smart Energy-Aware Data Allocation for Heterogeneous Memory," in *HPCC*, 2016.

[29] S. Ghose, H. Lee, and J. F. Martínez, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.

[30] A. Glew, "MLP Yes! ILP No," *ASPLOS WACI*, 1998.

[31] B. Goglin, "Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications," in *MEMSYS*, 2016.

[32] B. Grot, S. W. Keckler, and O. Mutlu, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-Effective QOS Scheme for Networks-on-a-Chip," in *MICRO*, 2009.

[33] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-Efficient and Heterogeneous Memory Systems," in *HPCA*, 2013.

[34] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.

[35] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006.

[36] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[37] ITRS, "Process Integration, Devices, and Structures," in *ITRS*, 2013.

[38] D. Jevdjic, S. Volos, and B. Falsafi, "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA*, 2013.

[39] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA*, 2010.

[40] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *SIGMETRICS*, 2016.

[41] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.

[42] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.

[43] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding Memory Interference Delay in COTS-Based Multi-Core Systems," in *RTAS*, 2014.

[44] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-Based Multi-Core Systems," *JRTS*, 2016.

[45] J.-S. Kim, C. S. Oh, H. Lee, D. Lee, H. R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park *et al.*, "A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM With 4 128 I/Os Using TSV Based Stacking," *JSSC*, 2012.

[46] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[47] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[48] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[49] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[50] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, 2016.

[51] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Checkpointed Early Load Retirement," in *HPCA*, 2005.

[52] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.

[53] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *ISCA*, 2009.

[54] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *Communications of the ACM*, 2010.

[55] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.

[56] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Tecas, HPS Research Group, Tech. Rep. TR-HPS-2010-002, 2010.

[57] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.

[58] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.

[59] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[60] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2A 1.2V 8Gb 8-Channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV," in *ISSCC*, 2014.

[61] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.

[62] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[63] Y. Li, J. Choi, J. Sun, S. Ghose, H. Wang, J. Meza, J. Ren, and O. Mutlu, "Managing Hybrid Main Memories with a Page-Utility Driven Performance Model," arXiv:1507.03303 [CoRR], 2015.

[64] F. X. Lin and X. Liu, "memif: Towards Programming Heterogeneous Memory Asynchronously," in *ASPLOS*, 2016.

[65] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[66] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[67] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A Full Hierarchy Hybrid Memory Management Framework," in *ICCD*, 2016.

[68] T. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang, T. Sasaki, A. Al-Shamma, C. Chen, M. Gupta, G. Hilton, A. Kathuria, V. Lai, M. Matsumoto, A. Nigam, A. Pai, J. Pakhale, C. H. Siau, X. Wu, Y. Yin, N. Nagel, Y. Tanaka, M. Higashitani, T. Minvielle, C. Gorla, T. Tsukamoto, T. Yamaguchi, M. Okajima, T. Okamura, S. Takase, H. Inoue, and L. Fasoli, "A 130.7mm^2 2-Layer 32Gb ReRAM Memory Device in 24nm Technology," *JSSC*, 2014.

[69] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *MICRO*, 2011.

[70] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[71] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *ISPASS*, 2001.

[72] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.

[73] K. T. Malladi, U. Kang, M. Awasthi, and H. Zheng, "DRAMScale: Mechanisms to Increase DRAM Capacity," in *MEMSYS*, 2016.

[74] J. Mandelman, R. Dennard, G. Bronner, J. DeBrosse, R. Divakaruni, Y. Li, and C. Radens, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," *IBM JRD*, 2002.

[75] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," in *HPCA*, 2015.

[76] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.

[77] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE CAL*, 2012.

[78] J. Meza, J. Li, and O. Mutlu, "A Case for Small Row Buffers in Non-Volatile Main Memories," in *ICCD*, 2012.

[79] J. Meza, J. Li, and O. Mutlu, "Evaluating Row Buffer Locality in Future Non-Volatile Main Memories," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. 2012-002, 2012.

[80] Micron Technology, Inc., "576Mb: x18, x36 RLDRAM 3," 2011.

[81] Micron Technology, Inc., "1Gb: x4, x8, x16 DDR3 SDRAM," 2013.

[82] Micron Technology, Inc., "LPDRAM for Mobile and Embedded Applications," 2015.

[83] Micron Technology, Inc., "Breakthrough Nonvolatile Memory Technology," https://www.micron.com/about/our-innovation/3d-xpoint-technology, 2016.

[84] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.

[85] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-Core DRAM Controllers," in *PODC*, 2008.

[86] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.

[87] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," *IEEE Micro*, 2006.

[88] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[89] O. Mutlu, "Efficient Runahead Execution Processors," Ph.D. dissertation, Univ. of Texas at Austin, 2006.

[90] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.

[91] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.

[92] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.

[93] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[94] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.

[95] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, 2003.

[96] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.

[97] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.

[98] M. Pavlovic, N. Puzovic, and A. Ramirez, "Data Placement in HPC Architectures with Heterogeneous Off-Chip Memory," in *ICCD*, 2013.

[99] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips*, 2011.

[100] A. J. Peña and P. Balaji, "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems," in *CLUSTER*, 2014.

[101] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.

[102] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-Off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *MICRO*, 2012.

[103] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.

[104] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.

[105] T. Ramirez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero, "Efficient Runahead Threads," in *PACT*, 2010.

[106] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.

[107] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-Change Random Access Memory: A Scalable Technology," *IBM JRD*, 2008.

[108] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *ISCA*, 2000.

[109] SAFARI Research Group GitHub Repository, "SAFARI Software Tools," https://github.com/CMU-SAFARI/.

[110] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.

[111] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[112] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[113] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *ASPLOS*, 2015.

[114] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson, "Locality vs. Criticality," in *ISCA*, 2001.

[115] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," in *MICRO*, 1998.

[116] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.

[117] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.

[118] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[119] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[120] Y.-H. Tseng, C.-E. Huang, C. H. Kuo, Y. D. Chih, and C.-J. Lin, "High Density and Ultra Small Cell Size of Contact ReRAM (CR-RAM) in 90nm CMOS Logic Technology and Circuits," in *IEDM*, 2009.

[121] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, 2016.

[122] K. Van Craeynest, S. Eyerman, and L. Eeckhout, "MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor," in *HiPEAC*, 2009.

[123] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *IEEE CAL*, 2011.

[124] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, 2010.

[125] X. Xiang, S. Ghose, O. Mutlu, and N.-F. Tzeng, "A Model for Application Slowdown Estimation in On-Chip Networks and Its Use for Improving System Fairness and Performance," in *ICCD*, 2016.

[126] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.

[127] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation"," in *MICRO*, 2017.

[128] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," arXiv:1704.02677 [CoRR], 2017.

[129] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT*, 2009.

[130] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mosse, "Writeback-Aware Bandwidth Partitioning for Multi-Core Systems with PCM," in *PACT*, 2013.

[131] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.

[132] W. Zuravleff and T. Robinson, "Controllers for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent No. 5,630,096, 1997.