Hardware Memory Management for Future Mobile Hybrid Memory Systems

Fei Wen, Mian Qin, Paul V. Gratz, *Senior Member, IEEE*, and A. L. Narasimha Reddy, *Fellow, IEEE*

***Abstract*—The current mobile applications have rapidly growing memory footprints, posing a great challenge for memory system design. Insufficient DRAM main memory will incur frequent data swaps between memory and storage, a process that hurts performance, consumes energy, and deteriorates the write endurance of typical flash storage devices. Alternately, a larger DRAM has higher leakage power and drains the battery faster. Furthermore, DRAM scaling trends make further growth of DRAM in the mobile space prohibitive due to cost. Emerging nonvolatile memory (NVM) has the potential to alleviate these issues due to its higher capacity per cost than DRAM and minimal static power. Recently, a wide spectrum of NVM technologies, including phase-change memories (PCMs), memristor, and 3-D XPoint has emerged. Despite the mentioned advantages, NVM has longer access latency compared to DRAM and NVM writes can incur higher latencies and wear costs. Therefore, the integration of these new memory technologies in the memory hierarchy requires a fundamental rearchitecting of traditional system designs. In this work, we propose a hardware-accelerated memory manager (HMMU) that addresses in a flat address space, with a small partition of the DRAM reserved for subpage block-level management. We design a set of data placement and data migration policies within this memory manager such that we may exploit the advantages of each memory technology. By augmenting the system with this HMMU, we reduce the overall memory latency while also reducing writes to the NVM. The experimental results show that our design achieves a 39% reduction in energy consumption with only a 12% performance degradation versus an all-DRAM baseline that is likely untenable in the future.**

***Index Terms*—FPGA accelerator, heterogeneous memory system, nonvolatile memory (NVM).**

I. Introduction

As THE demand for mobile computing power scales, mobile applications with ever-larger memory footprints are being developed, such as high-resolution video decoding, high-profile games, etc. This trend creates a great challenge for current memory and storage system design in these systems. The historical approach to address memory footprints larger than the DRAM available is for the OS to swap less used pages to the storage, keeping higher locality pages in memory. Given the latencies of modern storage systems (even "high" performance SSDs [1]–[3]) are several orders of magnitude higher than DRAM, however, allowing any virtual memory swapping to storage implies incurring a severe slowdown. Thus, the mobile device manufacturer rapidly expanded the DRAM size for the worst case possible memory footprint. For example, the DRAM capacity of the flagship phones from the Samsung Galaxy S series has expanded by 16 × over the past ten years. While this approach has been largely successful to date, the size of DRAM is constrained by both cost/economics and energy consumption. Unlike data centers, mobile devices are highly cost sensitive and have a highly limited energy budget. Moreover, the DRAM technology has a substantial background power, constantly consuming energy even in idle due to its periodic refresh requirement, which scales with DRAM capacity. Therefore, a larger DRAM means a higher power budget and a shorter battery life, particularly given recent hard DRAM VLSI scaling limits. The approach of provisioning more DRAM is not sustainable and hard limits will soon be hit on the scaling of the future mobile memory system.

The emergence of several nonvolatile-memory (NVM) technologies, such as Intel 3-D Xpoint [4], memristor [5], and PCM [6], provides a new avenue to address this growing problem. These new memory devices promise an order of magnitude higher density [7] per cost and lower static power consumption than traditional DRAM technologies, however, their access delay is significantly higher, typically also within one order of the magnitude of DRAM. Furthermore, these new technologies show significant overheads associated with writes and are nonvolatile. Thus, these emerging memory technologies present a unique opportunity to address the problems of growing application workload footprints with hybrid memory systems composed of both DRAM and emerging NVM memories. To exploit these new memory devices effectively, however, we must carefully consider their performance characteristics relative to existing points in the memory hierarchy. In particular, while memory access and movement in prior storage technologies, such as flash and magnetic disk are slow enough that software management via the OS was feasible. With emerging NVM memory accesses at within an order of the magnitude of DRAM, relying on traditional OS memory management techniques for managing placement between DRAM and NVM is insufficient as illustrated in Fig. 1.

In Fig. 1, a subset of benchmarks from the SPEC CPU2017 benchmark suite is executed in a system where around 128 MB

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by the National Science Foundation under Grant I/UCRC-1439722 and Grant FoMR-1823403; in part by DellEMC; and in part by Hewlett Packard Enterprise. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Fei Wen.)*

The authors are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: fei8wen@gmail.com; celery1124@tamu.edu; pgratz@gratz1.com; reddy@tamu.edu).

Digital Object Identifier 10.1109/TCAD.2020.3012213

TABLE I
APPROXIMATE PERFORMANCE COMPARISON OF DIFFERENT MEMORY TECHNOLOGIES [13]–[15]

Technology	HDD	FLASH	3D XPoint	DRAM	STT-RAM	MRAM
Read Latency	5ms	$100\mu s$	50 - 150ns	50ns	20ns	20ns
Write Latency	5ms	$100\mu s$	50 - 500ns	50ns	20ns	20ns
Endurance (Cycles)	$> 10^{15}$	10^4	10^9	$> 10^{16}$	$> 10^{16}$	$> 10^{15}$
$ per GB	0.025-0.5	0.25-0.83	6.5 [16]	5.3-8	N/A	N/A
Cell Size	N/A	$4 - 6F^2$	$4.5F^2$ [7]	$10F^2$	$6 - 20F^2$	$25F^2$

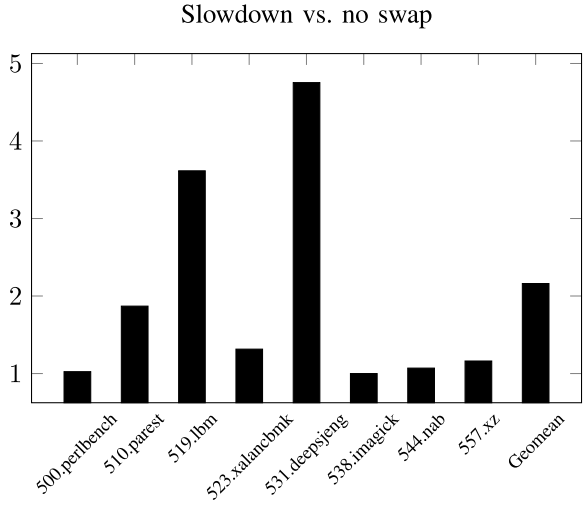

Fig. 1. Performance impact of OS memory management.

of the application's memory footprint is able to fit in the system DRAM directly. A ramdisk-based swap file is set up to hold the remainder of the application memory footprint. Since this ramdisk swapfile is implemented in DRAM, it represents an upper bound on the performance for pure software swapping. The results shown are normalized against a system where sufficient DRAM is available to capture the entire memory footprint. As we see, in this arrangement, the cost of pure OS managed swapping to NVM would be quite high, with applications seeing an average of ∼2× slowdown versus baseline. As we will show, a significant fraction of this overhead comes explicitly from the costs of the required page fault handling.

Some existing work has begun to explore system design for emerging hybrid memories. Broadly this prior work falls into one of two categories, first, some advocate using DRAM as pure hardware managed cache for NVM [8], [9]. This approach implies a high hardware cost for metadata management and imposes significant capacity and bandwidth constraints. Second, some have advocated for a pure software, OS managed approach [10]–[12]. As we discussed previously, this approach implies significant slowdowns due to software overhead of the operating system calls.

Here, we propose a new, hardware managed hybrid memory management scheme which retains the performance benefits of caching, without the high metadata overhead such an approach implies. Compared to previous work, our project has the following advantages.

1) With a ratio of 1/8 DRAM versus 7/8 NVM, we achieved 88% of the performance of an untenable full DRAM configuration while reducing the energy consumption by 39%.
2) Compared to inclusive DRAM caches, we preserve the full main memory capacity for the user applications.
3) Parallel access to both the DRAM and NVM is supported, rendering a higher effective memory bandwidth. This also helps to suppress the excessive cache insertion/replacements and prevent cache thrashing.
4) The data placement and migration are executed by hardware. This eliminates the long latency incurred by the OS managed virtual memory swap process.
5) To obtain optimal performance with applications with various localities, we created a combined management policy that addresses data at page and subpage block granularity, dependent on locality.

II. BACKGROUND AND MOTIVATION

With emerging NVM technologies providing more memory system capacity, density, and lower static power, they have the potential to meet the continuously increasing memory usage of mobile applications. Given their different characteristics from traditional DRAM and storage, however, the design of systems comprising these new technologies together with traditional DRAM and storage is an open question. Here, we examine the characteristics of this new memory technologies and the existing proposals to date on how to leverage them in system designs.

A. Nonvolatile Memory Technology Characteristics

Table I shows the relative characteristics of several emerging NVM technologies against traditional DRAM and storage [13], [14], [17]. HDD and Flash have 100k and 2k times slower read access than DRAM, respectively, while the emerging NVM technologies have read access latencies typically within one order of magnitude of DRAM. Meanwhile, emerging NVM technologies provide higher memory system capacity, density, and lower static power. Furthermore, we note that in these new technologies writes are often more expensive that reads both in terms of latency as shown and endurance/lifetime cost, as well as energy consumption for writing.

The relative closeness in performance and capacity to traditional DRAM of emerging NVM technologies argues for a different approach to memory management than traditional, OS or hardware-cache-based approaches. In the remainder of this section, we examine the prior work approaches to the design of hybrid memory systems.

B. Operating System-Based Memory Management

Hassan *et al.* [10] and Fedorov *et al.* [11] proposed to leverage the OS to manage placement and movement between

NVM and DRAM. They treat NVM as a parallel memory device on the same level as that of DRAM in the memory hierarchy. They argue that this approach can yield better utilization of the large NVM capacity without wasting the also relatively large DRAM capacity. Their approach is similar to the traditional approach of using storage as a swap space to extend the DRAM main memory space. The direct application of this approach to NVM creates some difficulties, however. When a given requested data are found to be in the swap space on the NVM, a page fault occurs which must be handled by an operating system. The latency of this action is not only comprised of the device latency itself but also the induced OS context switch and page fault handling. While in traditional storage systems with ms-level latencies, that cost is negligible, with the latency of SSD and other NVM devices significantly decreased, the OS management overheads come to dominate this latency, as discussed previously and indicated in Fig. 1.

C. Hardware-Managed DRAM Caches and Related Approaches

Other groups have proposed using DRAM as the cache/buffer for NVM, and thus, turning DRAM into the new last-level cache [8]. Similar schemes have also been applied to other memory devices with latency discrepancy in the heterogeneous-memory system (HMS). For instance, 3-D-stacked DRAM was proposed as a cache for off-chip DRAM in the works [18]–[21]. A common theme in all these designs is the difficulty in lookup and maintenance of the tag storage since the number of tags scales linearly with the cache size. Assuming the cache block size is 64 and 8 B of tag for each block, then a 16-GB DRAM cache requires 2 GB for the tag storage alone. That is, much too large to fit in a fast, SRAM tag store. Much of the prior work explores mechanisms to shrink the tag storage overhead [22]. Some researchers explored tag reduction [23]. Others aimed to reconstruct the cache data structure. For instance, some works combine the tag or other metadata bits into the data entry itself [19], [24].

Another issue these works attempt to address is the extended latency of tag access. DRAM devices have significantly greater access latency than SRAM. Additionally, their larger cache capacity requires a longer time for the tag comparison and data selection hardware. If the requested data address misses in the TLB, it takes two accesses to the DRAM before the data can be fetched. Lee *et al.* [25] attempted to avoid the tag comparison stage entirely by setting the cache block size to equal the page size and converting virtual addresses to cache addresses directly in a modified TLB. This approach, however, requires several major changes to the existing system architecture, including requiring extra information bits in the page table, modifying the TLB hardware, and an additional global inverted page table.

Broadly, several issues exist with the previously proposed hardware-based management techniques for future hybrid memory systems.

1) As with traditional processor cache hierarchies, every memory request must go through the DRAM cache before accessing the NVM. The prior work shows that this approach is suboptimal for systems, where bandwidth is a constraint and where a parallel access path is available to both levels of memory [26]. Furthermore, given the relatively slow DRAM access latency requiring a miss in the DRAM before accessing the NVM implies a significantly higher overall system latency.
2) These works largely assume an inclusive style caching. Given the relative similarity in capacity between DRAM and NVM, this implies a significant loss of capacity.
3) Given the capacities of DRAM and NVM versus SRAM used in processor caches, a traditional cache style arrangement implies a huge overhead in terms of cache metadata. This overhead will add significant delays to the critical path of index search and tag comparison, impacting every data access.

Liu *et al.* [27] proposed a hardware/software collaborative approach to address the overheads of pure software approaches without some of the drawbacks of pure hardware caching. Their approach, however, requires modifications both to the processor architecture as well as the operating system kernel. These modifications have a high NRE cost and hence, are difficult to be carried out in production. Ramos *et al.* [28] proposed a combined OS/hardware scheme where page migrations are performed in hardware, at the direction of the OS. Here, the OS maintains the page tables and other data structures.

In this article, we propose a hardware-based hybrid memory controller that is transparent to the user and as well as the operating system, thus, it does not incur the overheads of management of OS-based approaches. The controller is an independent module and compatible with existing hardware architectures and OSs. The controller manages both DRAM and NVM memories in flat address space to leverage the full capacity of both memory classes. Our approach also reserves a small portion of the available DRAM space to use as a hardware-managed cache to leverage spacial locality patterns seen in real application workloads to reduce writes to the NVM.

III. Design

Here, we describe the proposed design of our proposed hardware memory management for future hybrid memory systems. Based on the discussion in Section II and cognizant of the characteristics of emerging NVM technologies, we aim to design a system in which the latency overheads of OS memory management are avoided while hardware tag and meta-data overheads of traditional caching schemes are minimized.

A. System Architecture Overview

Fig. 2 shows the system architecture of our proposed scheme. The data access requests are received by the hybrid memory management unit (HMMU), if they miss in the processor cache. These are processed based on the built-in data placement policies and forwarded with address translation to either DRAM or NVM. The HMMU also manages the migration of data between DRAM and NVM by controlling the high-bandwidth DMA engine connecting the two types of memory devices.

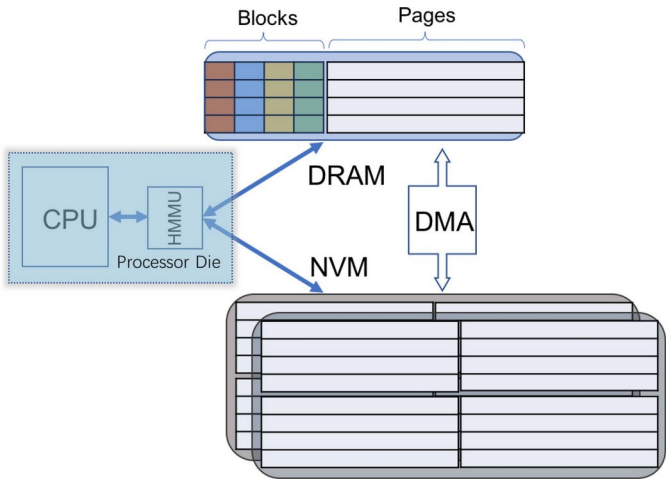

Fig. 2. System architecture overview.

B. Data Management Policy

A key component of the proposed HMMU design is its data management policy, i.e., the policy by which it decides where to place and when to move data between the different memory levels. Traditionally, in processor caches and elsewhere, cache blocks are managed with 64-B lines and policies such as set associative are used to decide what to replace upon the insertion of new lines into a given cache level. While this approach yields generally good performance results in processor caches, there are difficulties in adapting it for use in hybrid memories. As previously discussed in Section II-C, for a hybrid memory system of 16 GB comprised of 64-B cache lines, the tag store overhead would be an impractically large 2 GB. Extending the block size up to 4 KB to match the OS page size would significantly reduce the overheads of the tag store, bringing it down to 4 MB for a 16-GB space. Since the host operating system primarily uses 4-KB pages, using any larger size than 4 KB for block management, however, risks moving a set of potentially unrelated pages together in a large block, with little, if any spatial locality between different pages in the block. This is particularly true because the addresses seen in the HMMU are "physical addresses," thus, physically colocated pages may come from completely different applications, with no spatial relationship.[1] As we will discuss, however, even managing blocks on a page granularity will yield greater than optimal page movements between "fast" (DRAM) and "slow" (NVM) memory levels, due to the fact that only subsets of the page are ever touched in many applications. Thus, we will examine a hybrid scheme in which most of the fast memory is managed on a page basis, lowering tag overheads, while a small fraction is managed on a subpage basis to reduce page movement when only small portions of each page are being used at a given time.

In terms of organization and replacement, using traditional processor cache policies of set associativity and LRU replacement become unwieldy for a memory system of this size. The practical implementation of such a set-associative cache requires either a wide/multiported tag array (which becomes untenable for large SRAM structures) or multiple cycles to retrieve and compare each way in the set sequentially. The prior work from the OS domain [29], [30] shows that with a large number of pages to choose from, set associative, LRU replacement is not strictly necessary. Inspired by that we first developed a simple counter-based page replacement policy.

1) Counter-Based Page Management: Rather than implementing a set-associative organization with the drawbacks described above, we instead propose to implement a secondary, page-level translation table internal to the HMMU as illustrated in Fig. 3. The internal page table provides a one-to-one remapping, associating each CPU-side "physical" page number in the host address space to a unique page number in the hybrid memory address space, either in the fast or slow memory. Thus, any given host page can be mapped to any location in either fast or slow memory.

While this design gives great flexibility in mapping, when a slow memory page must be moved to fast memory (i.e., upon a slow memory reference we move that page to fast memory) it requires a mechanism by which to choose the fast memory page to be replaced. Inspired by prior work in the OS domain [29], [30], we designed the counter-based replacement policy for this purpose.

a) Algorithms and design: The counter-based replacement policy only requires one counter to keep track of the currently selected fast memory replacement candidate page, thus, it has minimal resource overhead and can efficiently be updated each cycle. The counter value is passed through a hash function and a modulo function to generate an index into the internal page table. If the retrieved page number turns out to be in slow memory, the counter increments by one and the hash function generates a new index for the next query to the page table. Such process loops until it finds a page in the fast memory, which becomes the candidate destination for the page swapping. The chance that a recently accessed page gets replaced is very rare because: 1) the total number of pages is very large and 2) the counter increases monotonically. To further reduce the possibility of evicting a recently touched page, however, we implemented a lightweight bloom filter that tracks the last 2048 accessed pages. Since checking against the bloom filter is parallel to normal page scan process and is also executed in background, it adds no extra delay to data accesses. Algorithm 1 shows the details of the algorithm.

Further details of the counter-based page management policy are as follows.

1) Current requests are processed at top priority under all circumstances. Except for rare cases when a given write request conflicts with ongoing page movement, we always process the current request first. As for those rare cases, since all write requests are treated as non-blocking, the host system shall not suspend for them to complete. Therefore, our design does not add overhead to the critical path of request processing.
2) Due to the parallel nature of the hardware, we search for free pages in fast memory in the background, without interference to host read request processing.

[1]While many systems do allow a subset of pages to be managed at larger granularities, the HMMU has no visibility to this OS-level mapping, thus, we conservatively assume 4-KB pages.

Algorithm 1: Counter-Based Page Relocation

```
Function unsigned pgtb-lookup (address) is
    return page_table[address/page_size];
Function unsigned search-free-fast-page() is
    while pointed_page ∈fast memory or
     pointed_page ∈bloom filter do
        counter++;
        pointed_page = pagetable[Mod(Hash(counter))];
    set candidate page as ready;
    return pointed_page;
Function counter-based-page-move(address) is
    pointed_page = pgtb-lookup (address);
    if pointed_page ∈fast memory then
        directly forward the request to DRAM
    else
        if candidate page is available then
            initiate to swap the content between requested
              page and candidate page.;
            Call page-swap();
        else
            Forward the current request to NVM;
Function page-swap (source_page, target_page) is
    while page swap is not completed do
        if new requests conflict with pages on flight then
            Froward the requests to the corresponding
              device depending on the current moving
              progress
        Continue the page swap;
    Update the corresponding entries in page table.;
```

Algorithm 2: Subpage Block Management

```
Function subpage block management(address) is
    pointed_page = pgtb-lookup(address);
    if pointed_page ∈fast memory then
        directly forward the request to DRAM
    else
        if the count of cached blocks > threshold value
          then
            if candidate free page available then
                initiate to swap the content between
                  requested page and candidate page.;
                Call page-swap();
            else
                Forward the current request to NVM;
        else
            initiate moving the block to cache zone
```

3) Page swap is initiated by the HMMU, however, it is executed by a separate DMA hardware module. Thus, it does not impact other ongoing tasks. In some very rare cases, the write requests are held until the current page copy is finished.
4) Data coherence and consistency are maintained during page movements.

2) Subpage Block Management: Various applications could have widely different data access patterns: those with high spatial locality may access a large number of adjacent blocks of data; while others may have a larger stride between the requested addresses. For applications with weak or no spatial locality, there is very limited benefit to moving the whole page of data into fast memory as most of the nontouched data may not be used at all. Based on this observation, we propose a scheme for a subpage size block management, which manipulates the data placement and migration in finer granularity. Our design supports flexible block size, ranging from the regular cache line size of 64 B, up to 1024 B. After comparing the results by sweeping all possible block sizes, we found the optimal choice to be 512 B.

a) Data migration policy: We set aside a small fraction of the fast memory and manage that area in a cache-like fashion with subpage sized blocks. The basic algorithm used in shown in Algorithm 2. Upon the first accesses to a slow memory page, instead of moving the whole page into fast memory, we will only move the requested block of that page into the "cache" zone in fast memory. We then keep track of the total number of cached blocks belonging to every page. Only after the count of cached blocks meets a certain threshold will we swap the whole page to fast memory.

Fig. 3 illustrates a walkthrough of the comprehensive data relocation policy. The memory controller receives a request to host a physical address 0x124000a200. In the first cycle, both the page table and cache metadata are checked in parallel to decide the target memory device. If the data are found only in the slow memory, the memory controller will trigger the data relocation process. [2] In the second cycle, we check the 4-b vector counter in the page table entry, which monitors the number of subpage blocks that have been cached for the current page. Comparing the vector against the preset threshold, it determines whether to start a full-page swap or a subblock relocation. If the vector value is smaller than the threshold value, then only that particular block containing the requested data (0x40027200 to 0x4002727f) will be copied to the cache. It is possible that the data might be found in both slow memory and the cache at the same time. To enforce data consistency, we always direct the read/write request to the copy in cache. This dirty data will be written back to the slow memory upon eviction.

In the other case when the vector value is bigger than the threshold value, the HMMU directs the DMA engine to start the full-page swapping process between the requested page (40027) and the destination page in fast memory. Here, as described in Algorithm 1, the fast memory pages to be replaced is selected via the replacement counter, i.e., page 00038 in this example. Once the data swapping is completed, the memory controller updates the new memory addresses of the two swapped pages in the internal page table.

[2]Note that the request is serviced immediately, directly from the slow memory, while the data migration happens in the background.

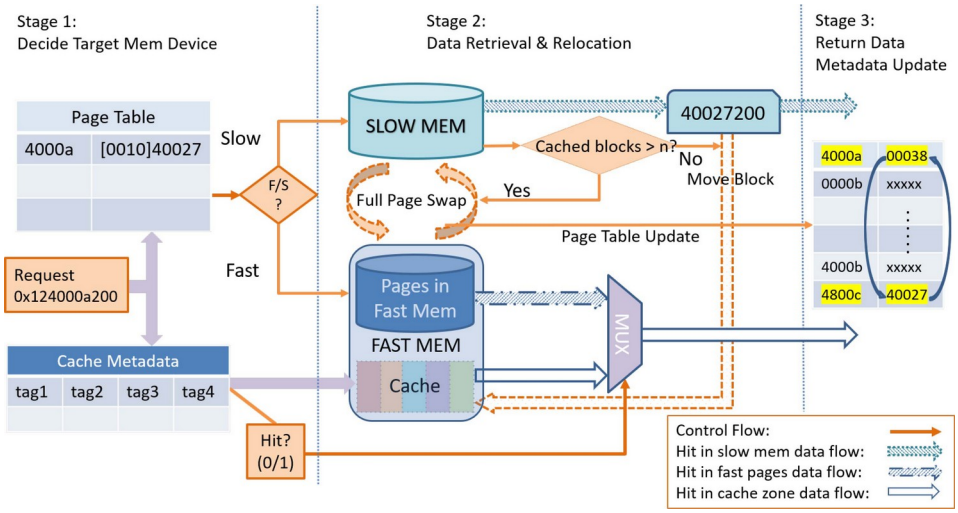

Fig. 3. Data relocation policy.

3) Hardware Cost and Overhead: Each page table entry takes $\log_2(\texttt{Memory Space}/\texttt{Page Size})$ bits to represent the page address. In addition, we need some bits for statistical metadata such as the counter of misses occurring to the page. In our sample design, the memory space is 2 GB and the page size is 4 KB, thus, the hardware cost per page could be rounded to $\log_2(2\ \texttt{GB}/4\ \texttt{KB}) + 5\ \texttt{bits} = 3\ \texttt{bytes}$, and the total cost is 1.5 MB. The page table cost scales linearly with memory size whereas the cost per entry only grows logarithmically. The metadata for each cache set is comprised of three parts: 1) four tags (8 bits × 4); 2) pseudo-LRU bits (3); and 3) dirty bits (4), which adds to 39 b. The total cost is $39\ \texttt{bits} \times 2^{16} \approx 312\ \texttt{KB}$. Since the cache is read and check parallel to the access to the page table, there is no additional timing cost for handling regular requests. The DMA provides the nonconflict data relocation for the subpage block level as same as that of the page relocation.

4) Static Versus Adaptive Caching Threshold: With both page and block migration available, a new question arises, how to choose wisely between these two policies for optimal results? We note that these policies have different characteristics as follows.

1) With page migration, the data are exclusively placed between NVM and DRAM devices. Thus, larger memory space is available to applications, and the bandwidth of both devices is available.
2) Subpage-block migration is done in an inclusive cache fashion, thus, avoids the additional writes to NVM when the clean data blocks are evicted from DRAM.

For applications with a strong spatial locality, whole page migration maximizes performance because the migration cost is only incurred once, and the following accesses hit in the fast memory. Alternately, subpage block promotion benefits applications with less spacial locality because it limits writes to NVM incurred by full-page migration. We further note that application behavior may vary over time with one policy being better in one phase and another better during another.

We, therefore, include in the page translation table an 8-bit bitmap for tracking accesses to each subpage block of the given page. This allows measurement of the utilization rate of promoted pages. If a large portion of blocks were revisited, then we lower the threshold to allow more whole page migration. Alternately, if few blocks were accessed we suppress the whole page promotion by raising the threshold value, decreasing the rate at which full pages are migrated.

IV. Evaluation

In this section, we present the evaluation of our proposed HMMU design. First, we present the experimental methodology. Then, we discuss the performance results. Finally, we analyze some of the more interesting data points.

A. Methodology

1) Emulation Platform: Evaluating the proposed system presents several unique challenges because we aim to test the whole system stack, comprising not only the CPU but also the memory controller, memory devices, and the interconnections. Furthermore, since this project involves hybrid memory, accurate modeling of DRAM is required. Much of the prior work in the processor memory domain relies upon software simulation as the primary evaluation framework with tools, such as Champsim [31] and gem5 [32]. However, detailed software simulators capable of our goals impose huge simulation time slowdowns versus real hardware. Furthermore, there are often questions about the degree of fidelity of the outcome of arbitrary additions to software simulators [33].

Another alternative used by some prior work [11] is to use an existing hardware system to emulate the proposed work. This method could to some extent alleviate the overlong the simulation runtime, however, no existing system supports our proposed HMMU.

Thus, we elected to emulate the HMMU architecture on an FPGA platform. FPGAs provide flexibility to develop and test sophisticated memory management policies while its hardware-like nature provides near-native simulation speed. The FPGA communicates with the ARM CortexA57 CPU via a high-speed PCI Express link and manages the two memory

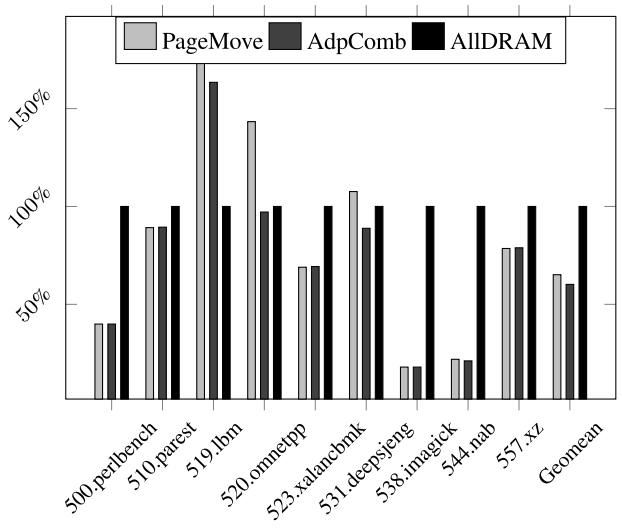

Fig. 4. Energy consumption comparison.

TABLE II
EMULATION SYSTEM SPECIFICATION

Component	Description
CPU	ARM Cortex-A57 @ 2.0GHz, 8 cores, ARM v8 architecture
L1 I-Cache	48 KB instruction cache, 3-way set-associative
L1 D-Cache	32 KB data cache, 2-way set-associative
L2 Cache	1MB, 16-way associative, 64kB cache line size
Interconnection	PCI Express Gen3 (8.0 Gbps)
Memory	128MB DDR4 + 1GB NVM
OS	Linux version 4.1.8

modules (DRAM and NVM) directly. The DRAM and NVM memories are mapped to the physical memory space via the PCI base address register (BAR) window. From the perspective of the CPU, they are rendered as available memory resource same as other regions of this unified space.

Our platform emulates various NVM access delays by adding stall cycles to the operations executed in FPGA to access external DRAM. The platform is not constrained to any specific type of NVM, but rather allows us to study and compare the behaviors across any arbitrary combinations of hybrid memories. In the following sections, we would show the simulation results with different memory devices. The detailed system specification is listed in Table II.

2) NVM Emulation: We measured the round trip time in FPGA cycles to access external DRAM DIMM first, and then scaled the number of stalled cycles according to the speed ratio between DRAM and future NVM as described in Table I. Thus, we have one DRAM DIMM running at full speed and the other DRAM DIMM emulating the approximate speed of NVM memory.

3) Workloads: We use applications from the recently released SPEC CPU 2017 benchmark suite [34]. To emulate memory-intensive workloads for future mobile space, we selected only those SPEC CPU 2017 benchmarks that require a larger working set than the fast memory size in our system. The details of tested benchmarks are listed in Table III.

TABLE III
TESTED WORKLOADS [34]

Benchmark	Description	Memory footprint
Integer Application		
500.perlbench	Perl interpreter	202MB
520.omnetpp	Discrete Event simulation - computer network	241MB
523.xalancbmk	XML to HTML conversion via XSLT	481MB
531.deepsjeng	Artificial Intelligence: alpha-beta tree search (Chess)	700MB
557.xz	General data compression	727MB
Float Point Application		
510.parest	Biomedical imaging: optical tomography with finite elements	413MB
519.lbm	Fluid dynamics	410MB
538.imagick	Image Manipulation	287MB
544.nab	Molecular Dynamics	147MB

To ensure that application memory was allocated to the HMMU's memory, the default Linux malloc functions are replaced with a customized jemalloc [35]. Thus, the HMMU memory access was transparent to the CPU and cache, and no benchmark changes were needed.

4) Designs Under Test: Here, we test the following data management policies developed for use with our HMMU.

1) *Static:* A baseline policy in which host requested pages are randomly assigned to fast and slow memory. This serves as a nominal, worst case, memory performance.
2) *PageMove:* The whole 128-MB DRAM is managed on the granularity of 4k pages. When a memory request is missed in fast memory, the DMA engine will trigger a page relocation from slow memory to fast memory, as described in Section III-B1.
3) *AdpComb:* Here, 16 MB out of the 128-MB DDR3 is reserved for subpage block relocation, managed in the cache-like fashion, as described in Section III-B2. The remainder of the DRAM is managed on a full page basis. An adaptive threshold is used to determine when the full page should be moved.
4) *AllDRAM:* Here, we implement a baseline policy in which there is sufficient fast memory to serve all pages in the system and no page movement is required. This serves as a nominal, best case but impractical memory performance design.

B. Results

1) Energy Saving: Emerging NVM consumes minimal standby power, which could help save energy consumption on mobile computation. We evaluated and compared the energy spent in running SPEC 2017 benchmarks between the full DRAM configuration and our policies. We referred to Micron DDR4 technical spec [36] for DRAM and recent work on 3DxPoint [37] for NVM device power consumption, respectively (Table IV). We normalize the energy consumption of our policies to that of the AllDRAM configuration and present them in Fig. 4. In the figure, we see that both techniques save a substantial amount of energy. On average, the AdpComb adaptive policy only consumes 60.2% energy as compared to the AllDRAM configuration, while the PageMove is at

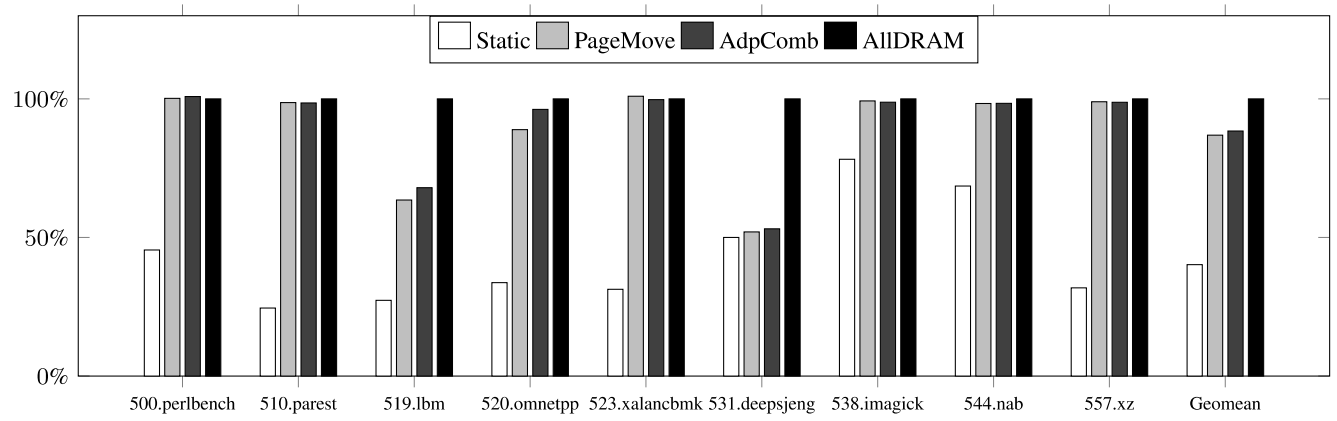

Fig. 5. SPEC 2017 performance speedup.

TABLE IV
POWER CONSUMPTION OF DDR4 AND 3-D-XP OINT

Technology	DDR4	3Dxpoint
Read Energy	4.2nJ	1.28nJ
Write Energy	3.5nJ	8.7nJ
Background Power	30mW/GB	~ 0

65.1%. That said, several benchmarks see energy consumption increases under the PageMove policy. AdpComb, while also seeing increased energy consumption under 519.lbm, shows better energy consumption than the other two policies for all other cases.

Further investigating the distribution of energy consumption, we track the DRAM background power and the number of DRAM read/writes and NVM read/writes. Since 7/8 of the memory was replaced with NVM, the standby power shrinks significantly. Although write operations to NVM dissipate more energy than DRAM, the AdpComb policy avoids most of this increase by absorbing many writes in DRAM. Our policies saw the greatest energy efficiency improvement with applications imagick and nab, which spent 17.9% and 21.1% energy compared to full DRAM. We find that these two applications have high processor cache hit rates and spent the most time in computation. Thus, they have few references to the memory, and the largest portion of energy was spent on DRAM background power. Thus, AptComb policy's advantage of having much lower DRAM static power is best exploited. Our policies did pretty well with all benchmark applications except lbm, which spent 63% more energy. This application incurred a massive number of cache block writebacks to NVM. We investigated the case and found lbm has the highest percentage of store instructions among all benchmark applications [38]. This creates many dirty blocks, and thus, writebacks are expected when blocks are later evicted. The amount of writes is also amplified by the writebacks of cache blocks.

2) Runtime Performance: Fig. 5 shows the speedup attained by different designs under test for the various benchmarks in the SPEC CPU 2017 benchmark suite. Here, all the results are normalized to the runtime of the ideal, AllDRAM, memory configuration. We see that the average performance of AdpComb is 88.4%, while the random static allocation "Static" only yields 40% of the AllDRAM performance. Thus, the adaptive policy achieves more than 2× performance benefit versus the worst case, static allocation policy under the same memory resource. Generally, the AdpComb policy outperforms the other two policies we propose, though interestingly, for many benchmarks, including perlbench, parest, xalancbmk, xz, imagick, and nab, PageMove comes within 5% of the performance of AllDRAM.

C. Analysis and Discussion

The adaptive AdpComb policy successfully reduces energy by 40%, with a modest 12% loss of the performance versus an unrealistic and unscalable AllDRAM design. AdpComb attempts to make the optimal choice between the full page and the block migration. In the remainder of this text, we will further analyze the experiment results.

1) PageMove Policy Performance: The PageMove policy has similar average runtime performance (86.9%) to the adaptive AdpComb policy (88.4%). Fig. 6 shows the breakdown of memory requests that hit in the fast pages and slow pages, respectively. When compared to the speedup in Fig. 5, we see the benchmarks in which PageMove policy works best (500.perlbench, 510.parest, 523.xalancbmk, 538.imagick 544.nab, and 557.xz) have more than 95% of their memory requests hitting in the fast pages, while the hit rate in slow pages becomes negligible. This provides a large performance boost considering that the system's slow memory is 8× slower than the fast memory.

The PageMove policy performs worst on the benchmark 531.deepsjeng, with a slowdown of 52% versus AllDRAM. We divided the number of hits in fast memory by the occurrences of page relocation, and found that deepsjeng has the lowest rate (0.03) across all the benchmark applications (Geomean is 3.96). This suggests that when a page is relocated from slow memory to fast memory, the remainder of that page is often not extensively utilized. Furthermore, we also see an exceptionally high ratio of blocks moved to cache versus page relocation. The geometric mean of all benchmarks is 10.5 while deepsjeng marks 397. This is a sign that in most cases, the page is only visited for one or two lines, and never accumulates enough cached blocks to begin a whole page relocation. To sum up, deepsjeng has a sparse and wide-range memory access pattern,

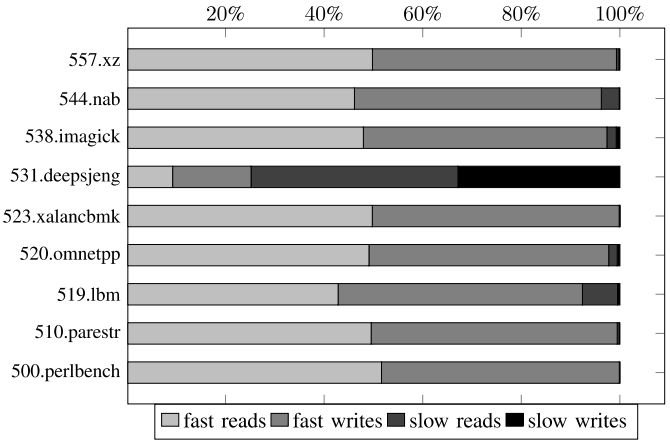

Fig. 6. Memory accesses breakdown of PageMove policy.

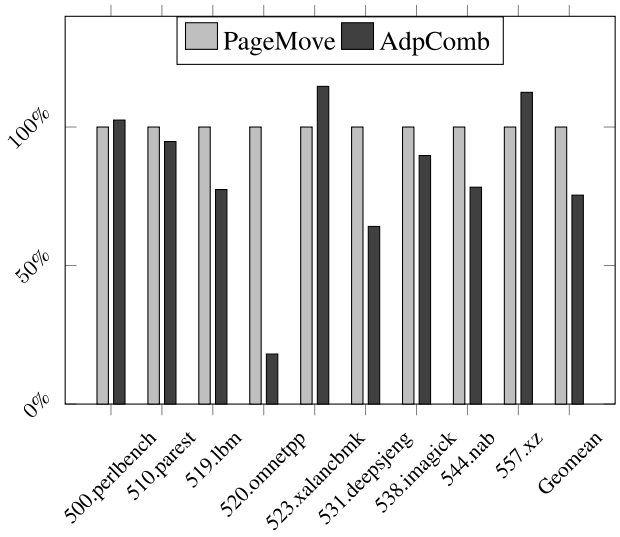

Fig. 7. Writes to NVM.

which is quite difficult to prefetch effective data or improve performance.

519.lbm presents another interesting case, since its performance is also poor. Similar to deepsjeng, the hit rate in fast memory is low in contrast to the number of page relocations. However, a key difference is that over 60% of the cached blocks were evicted after its underlying pages relocated to fast memory. This indicates that lbm walks through many blocks of the same page and triggers the whole page relocation quickly. On that account, we deduce that this benchmark will benefit from a configuration with more fast pages and a smaller cache zone. We reran this benchmark with a cache size of 8 MB and the threshold value of 1, and found a supportive result of 8% performance gain on top of the default threshold value of 4.

2) Writes Reduction and NVM Lifetime Saving: Unlike the traditional DRAM, emerging NVM technologies have different characteristics for reads and writes. Write operations dissipates more than 8× the energy of reads [13]. Moreover, NVM technologies often have limited write endurance, i.e., the maximum cycles of writes before they wear out. Hence, if we could reduce the amount of writes, we could greatly save energy consumption and extend the lifetime of the NVM device. Fig. 7 shows the percentage of writes to slow memory for both techniques, normalized against the number of writes seen in the PageMove policy. Note that we measure not only the direct writes from the host but also the writes induced by page movements and subpage block writebacks to slow memory. In the figure, we see that our combined policy has an average of 20% fewer writes than the PageMove policy. While several benchmarks benefit from the subpage block cache, this advantage is strongest with omnetpp, with a drop of 86%. We reran the tests with different static page relocation thresholds and examined the changes in runtime and total numbers of writes to NVM. The runtime varied according to the same trend as the number of writes, and the threshold value of 4 turned out to be the overall sweet spot. Both metrics started to deteriorate rapidly when the threshold value shifted. Then, we measured the number of writes to NVM incurred by page relocation and block relocation, respectively. The data showed that more pages were relocated when the threshold was lowered. On the other hand, the amount of block migration grew rapidly as the threshold increased. The tradeoffs reached perfect balance at the value of 4, which had a slightly more page moves than that of value 5, yet significantly fewer block migrations.

3) Adaptive Policy: The analysis above showed that the whole page promotion policy favors certain benchmark applications in which most blocks were revisited on the promoted pages. Meanwhile, other applications benefit from subpage block promotions as only a subset of blocks were reutilized. If we could always choose the correct policy for each application, then we could expect the optimal results for overall performance. These results reinforce the reasoning behind our AdpComb policy's adaptive threshold, wherein for applications where pages are mostly utilized full page movement is completed quickly, while for applications where accesses are sparse, page movement is postponed till most of the page has been touched once.

V. Conclusion

A wide spectrum of NVM technologies are emerging, including PCMs, memristor, and 3-D XPoint. These technologies look particularly appealing for inclusion in the mobile computing memory hierarchy. While NVM provides higher capacity and less static power consumption, than traditional DRAM, its access latency and write costs remain problematic. The integration of these new memory technologies in the mobile memory hierarchy requires a fundamental rearchitecting of traditional system designs. Here, we presented an HMMU that addresses both types of memory in a flat address space. We also designed a set of data placement and data migration policies within this memory manager such that we may exploit the advantages of each memory technology. While the page move policy provided good performance, adding a subpage-block caching policy helps to reduce writes to NVM and save energy. On top of these two fundamental policies, we built an adaptive policy that intelligently chooses between them, according to the various phases of the running application. The experimental results show that our adaptive policy can significantly reduce power consumption by almost 40%.

With only a small fraction of the system memory implemented in DRAM, the overall system performance comes within 12% of the full DRAM configuration, which is more than 2 × the performance of random allocation of NVM and DRAM. By reducing the number of writes to NVM, our policy also helps to extend device lifetime.

REFERENCES

[1] Intel Corporation. (2015). *Intel 750*. [Online]. Available: https://ark.intel.com/products/86740/Intel-SSD-750-Series-400GB-12-Height-PCIe-3_0-20nm-MLC

[2] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "STRATA: A cross media file system," in *Proc. 26th Symp. Oper. Syst. Principles (SOSP)*, 2017, pp. 460–477.

[3] J. Zhang *et al.*, "FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 477–492.

[4] Intel Corporation. (2016). *Intel Optane Technology*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[5] K. Eshraghian, K.-R. Cho, O. Kavehei, S.-K. Kang, D. Abbott, and S.-M. S. Kang, "Memristor MoS content addressable memory (MCAM): Hybrid architecture for future high performance search engines," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 8, pp. 1407–1417, Aug. 2011.

[6] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 465–479, Jul. 2008.

[7] J. Choe. (2017). *Intel 3D Xpoint Memory Die Removed From Intel Optane PCM*. [Online]. Available: https://www.techinsights.com/blog/intel-3d-xpoint-memory-die-removed-intel-optanetm-pcm-phase-change-memory

[8] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 24–33.

[9] 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 *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2014, pp. 1–12.

[10] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid DRAM/NVM main memory," in *Proc. 12th ACM Int. Conf. Comput. Front. (CF)*, 2015, pp. 1–8.

[11] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future NVM storage," in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, 2017, pp. 399–410.

[12] Z. Wang, Z. Gu, and Z. Shao, "Optimizated allocation of data variables to PCM/DRAM-based hybrid main memory for real-time embedded systems," *IEEE Embedded Syst. Lett.*, vol. 6, no. 3, pp. 61–64, Sep. 2014.

[13] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electron.*, vol. 125, pp. 25–38, Nov. 2016.

[14] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, Jul. 2016.

[15] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nat. Nanotechnol.*, vol. 8, pp. 13–24, Dec. 2012.

[16] A. Shilov. (2019). *Pricing of Intel's Optane DC Persistent Memory Modules*. [Online]. Available: https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks

[17] ITRF Semiconductors. (2015). *More Moore*. [Online]. Available: https://www.semiconductors.org/resources/2015-international-technology-roadmap-for-semiconductors-itrs/

[18] N. Madan *et al.*, "Optimizing communication and capacity in a 3D stacked reconfigurable cache hierarchy," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit.*, Feb. 2009, pp. 262–274.

[19] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective DIE-stacked dram cache," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2014, pp. 25–37.

[20] 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 *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2012, pp. 235–246.

[21] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked DRAM as part of memory," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2014, pp. 13–24.

[22] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE Comput. Architect. Lett.*, vol. 11, no. 2, pp. 61–64, Jul.–Dec. 2012.

[23] C.-C. Huang and V. Nagarajan, "ATCACHE: Reducing DRAM cache latency via a small SRAM tag cache," in *Proc. 23rd Int. Conf. Parallel Architect. Compilation (PACT)*, 2014, pp. 51–60.

[24] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE Comput. Archit. Lett.*, vol. 11, no. 2, pp. 61–64, Jul.–Dec. 2012.

[25] Y. Lee *et al.*, "A fully associative, tagless DRAM cache," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 211–222.

[26] X. Wu and A. L. N. Reddy, "Managing storage space in a flash and disk hybrid storage system," in *Proc. IEEE Int. Symp. Model. Anal. Simulat. Comput. Telecommun. Syst.*, Sep. 2009, pp. 1–4.

[27] H. Liu *et al.*, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *Proc. Int. Conf. Supercomput. (ICS)*, 2017, pp. 1–10.

[28] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 85–95.

[29] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. VLDB*, 1994, pp. 439–450.

[30] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 1993, pp. 297–306.

[31] ChampSim. (2016). *Champsim*. [Online]. Available: https://github.com/ChampSim/ChampSim

[32] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[33] T. Nowatzki, J. Menon, C. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, Nov. 2015.

[34] SPEC. (2017). *SPEC CPU2017 Documentation*. [Online]. Available: https://www.spec.org/cpu2017/Docs/

[35] J. Evans. (2016). *Jemalloc*. [Online]. Available: http://jemalloc.net/

[36] "Calculating memory power for DDR4 SDRAM," Micron, Boise, ID, USA, Rep. TN-40-07, 2017.

[37] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 2–13.

[38] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2018, pp. 149–158.

Fei Wen is currently pursuing the Ph.D degree in computer engineering with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA.

He conducted research on interconnect network design and modeling for exascale systems as Research Associate with HP Labs, Palo Alto, CA, USA, where he has been a Research Assistant since 2016. He has expertise across the hardware/software stack, in RTL design, FPGA development, kernel programming, and architecture performance modeling. His current research interests include computer architecture, memory systems, and FPGA accelerator.

Mian Qin received the B.S. degree in electrical engineering and the M.S. degree in information and communication engineering from Shanghai Jiao Tong University, Shanghai, China, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree in computer engineering with Texas A&M University, College Station, TX, USA.

Since 2016, he has been a Research Assistant with the Electrical and Computer Engineering Department, Texas A&M University. From 2012 to 2016, his research interests include high-speed, multichannel data acquisition and signal processing, RTL design, and FPGA development. His current research interests include memory/storage system, computer architecture, and hardware/software co-design.

Paul V. Gratz (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from the University of Florida, Gainesville, FL, USA, in 1994 and 1997, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, TX, USA, in 2008.

He is an Associate Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA. From 1997 to 2002, he was a Design Engineer with Intel Corporation, Santa Clara, CA, USA. His research interests include efficient and reliable high performance computer architecture, processor memory systems, and on-chip interconnection networks.

Dr. Gratz received the "Excellence Award in Teaching" from the Texas A&M College of Engineering in 2017 and the "Distinguished Achievement Award in Teaching—College Level" from the Texas A&M Association of Former Students in 2016. His paper, "Synchronized Progress in Interconnection Networks: A New Theory for Deadlock Freedom," was selected as a Top Pick from the architecture conferences in 2018 by IEEE Micro. His papers "Path Confidence Based Lookahead Prefetching" and "B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors" were nominated for best papers at MICRO'16 and MICRO'14, respectively. At ASPLOS'09, he received a best paper award for "An Evaluation of the TRIPS Computer System."

A. L. Narasimha Reddy (Fellow, IEEE) received the B.Tech. degree (Hons.) in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1985, and the M.S. and Ph.D. degrees in computer engineering form the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1987 and 1990, respectively.

He is currently a J. W. Runyon Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA, as well as an Associate Dean for Research with Texas A&M Engineering Program, and an Assistant Director of Strategic Initiatives and Centers, Texas A&M Engineering Experiment Station. From 1990 to 1995, he was a Research Staff Member with IBM Almaden Research Center, San Jose, CA, USA. He holds five patents and was awarded a technical accomplishment award while at IBM. His research interests are in computer networks, storage systems, and computer architecture.

Prof. Reddy received an NSF Career Award in 1996. His honors include an Outstanding Professor Award by the IEEE student branch at Texas A&M University from 1997 to 1998, the Outstanding Faculty Award by the Department of Electrical and Computer Engineering from 2003 to 2004, the Distinguished Achievement Award for teaching from the Former Students Association of Texas A&M University, and the citation "for one of the most influential papers from the first ACM Multimedia Conference."