Revamping Storage Class Memory With Hardware Automated Memory-Over-Storage Solution

Jie Zhang[1], Miryeong Kwon[1], Donghyun Gouk[1], Sungjoon Koh[1], Nam Sung Kim[2]
Mahmut Taylan Kandemir[3], Myoungsoo Jung[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], University of Illinois Urbana-Champaign[2]
Pennsylvania State University[3]
http://camelab.org

***Abstract*—Large persistent memories such as NVDIMM have been perceived as a disruptive memory technology, because they can maintain the state of a system even after a power failure and allow the system to recover quickly. However, overheads incurred by a heavy software-stack intervention seriously negate the benefits of such memories. First, to significantly reduce the software stack overheads, we propose HAMS, a hardware automated Memory-over-Storage (MoS) solution. Specifically, HAMS aggregates the capacity of NVDIMM and ultra-low latency flash archives (ULL-Flash) into a single large memory space, which can be used as a working memory expansion or persistent memory expansion, in an OS-transparent manner. HAMS resides in the memory controller hub and manages its MoS address pool over conventional DDR and NVMe interfaces; it employs a simple hardware cache to serve all the memory requests from the host MMU after mapping the storage space of ULL-Flash to the memory space of NVDIMM. Second, to make HAMS more energy-efficient and reliable, we propose an "advanced HAMS" which removes unnecessary data transfers between NVDIMM and ULL-Flash after optimizing the datapath and hardware modules of HAMS. This approach unleashes the ULL-Flash and its NVMe controller from the storage box and directly connects the HAMS datapath to NVDIMM over the conventional DDR4 interface. Our evaluations show that HAMS and advanced HAMS can offer 97% and 119% higher system performance than a software-based NVDIMM design, while costing 41% and 45% lower energy, respectively.**

I. Introduction

Recently, persistent memories such as PRAM [41] and 3D XPoint [48] have received a considerable attention as their non-volatile intrinsic, high density and low power consumption can benefit modern datacenters and high-performance computers. For such systems, back-end storage is required for recovering from system failures and crashes. Since the persistent memories can spontaneously and instantaneously recover all memory states, they can eliminate a large number of accesses to the back-end storage and associated runtime overheads [38], [49], [61]. Besides, enterprise workstations and servers employ the persistent memory with DirectAccess (DAX) [18], [67], which brings the advantages of unprecedented levels of performance and data resiliency [57].

There are three standard persistent memory types (i.e., NVDIMM-N/F/P). NVDIMM-F directly integrates flash into a dual-inline memory module (DIMM) to provide a high capacity similar to storage. However, NVDIMM-F cannot simply replace DRAM, as it only exposes a block interface. NVDIMM-P such as Optane DC PMM is byte-addressable, but its performance using the app direct mode to support data persistence is yet 6× worse than DRAM [26], [29]. In contrast, NVDIMM-N aims to offer "byte-addressable" persistency with DRAM-like performance [54]. NVDIMM-N generally consists of a small flash device and multiple DRAM modules with a battery. NVDIMM-N can be useful for a wide range of data-intensive applications such as database management system (DBMS) [2], transaction processing [45], [63], data analytics [8], and checkpointing [12]. However, the memory space of NVDIMM-Ns (*e.g.*, 4GB ∼ 64GB) is considerably smaller than that of NVDIMM-P and persistent storage devices such as solid state drives (SSDs). Furthermore, the capacity of DRAM in NVDIMM-Ns is constrained by poor scaling of battery that needs to supply the power for DRAM backup operations when a power failure occurs [5], [71]. For example, for the past two decades, the storage density of DRAM has increased by many orders of magnitude, whereas the energy density of lithium-ion battery has only tripled [34].

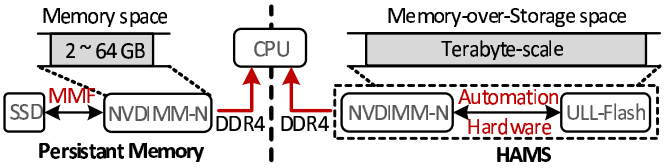


Fig. 1: NVDIMM-N vs. HAMS.

A possible solution to build a large and scalable, yet persistent memory space is to use NVDIMM-N together with SSD and memory-mapped files (MMFs), which can be implemented in an OS memory manager or a file system. This allows data-intensive applications to access a large storage space using conventional load/store instructions. However, we observe that such MMF-assisted persistent memory can degrade the application performance at the user-level by 48%, on average, compared to an NVDIMM-N-only solution (cf. Section III-B). Such severe performance degradation is caused by not only the long stall latency in accessing SSD but also the software overhead and frequent data copies between the user and system memory spaces in a conventional storage stack.

Tackling the aforementioned limitations, we propose HAMS, a **H**ardware **A**utomated **M**emory-over-**S**torage (MoS)

solution that aggregates the memory capacity of NVDIMM-N and the storage capacity of new ultra-low latency flash archives, referred to as *ULL-Flash* [37], [68], into a single large memory space (*cf.* Figure 1). The large monolithic memory space of HAMS can be used as a working memory or a persistent memory expansion. Our HAMS resides in the memory controller hub, and manages its MoS address pool by leveraging the conventional DDR4 and NVMe interfaces. To this end, HAMS employs a simple hardware cache to handle all the memory requests from the host memory management unit (MMU) by mapping the storage space of ULL-Flash to the memory space of NVDIMM-N. In case of an NVDIMM-N cache miss, HAMS internally manages the NVMe commands and I/O request queues while hiding all the NVMe protocol and interface management overheads from the OS, such that data requested by MMU are always served by NVDIMM-N.

While the "baseline" design of HAMS can offer a 20GB/s peak bandwidth, it can still yield sub-optimal system performance, especially when running large-scale data-intensive applications due to some inefficiencies, described subsequently. First, handling NVDIMM-N cache misses requires data transfers between NVDIMM-N and ULL-Flash. That is, HAMS needs to go through both DDR4 and PCIe interfaces, including physical layers, controllers and protocol managers, to handle NVDIMM-N cache misses. However, the PCIe bandwidth is insufficient to expose the full potential of ULL-Flash to HAMS. Consequently, the data transfers to handle frequent NVDIMM-N cache misses for large data-intensive applications can contribute to as high as 47% of the total memory access latency of HAMS. Second, some data may redundantly exist in the internal DRAM of both NVDIMM-N and ULL-Flash, used as cache and/or buffer. For example, most modern SSDs, including ULL-Flash, employ large internal DRAMs, to buffer/cache all incoming I/O requests to hide the long latency of the underlying flash. This would help SSDs improve performance when employed in a block-storage file system, but it wastes power and increases the internal complexity of SSDs when employed for a MoS-based solution.

To address these limitations, we also propose to aggressively integrate HAMS into existing computer systems by modifying its datapath and hardware modules. This makes the baseline solution more energy-efficient and reliable, as far as data persistency is concerned. This "advanced HAMS" unleashes ULL-Flash and its NVMe controller from the storage box and directly connects their datapath to NVDIMM-N. To this end, we propose to slightly modify the NVMe controller within ULL-Flash, by incorporating a new register-based interface and tightly integrating the interface with the DDR4 interface of HAMS. This aggressive integration allows ULL-Flash to access the DRAM devices in NVDIMM-N without any intervention from HAMS, and removes the DRAM buffer from ULL-Flash while enabling full NVMe functionality.

Our evaluation results show that HAMS and advanced HAMS provide 97% and 119% higher system performance, than the MMF-based NVDIMM-N+SSD hybrid design, while consuming 41% and 45% less system energy, respectively.

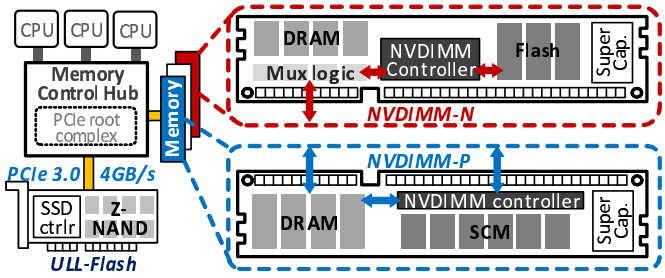


Fig. 2: Persistent memory and storage.

Types	Capacity	OS intervention	Performance	Byte-addressable
NVDIMM-N [31]	Low	No	DRAM-like	Yes
NVDIMM-F [54]	High	Yes	Slow	No
NVDIMM-P [16]	Medium	Yes	Medium	Yes
HAMS	**High**	**No**	**DRAM-like**	**Yes**

TABLE I: Feature comparison across different persistent memories and HAMS.

II. Background

In this section, we first describe the key hardware components of persistent memory and the storage stack for heterogeneous memory expansion. Next, we give the hardware and firmware details of ULL-Flash.

A. Persistent Memory and Storage

Figure 2 depicts a high-level view of a system architecture that includes NVDIMM-N/P and ULL-Flash. NVDIMM-N/P is attached to the memory controller hub (MCH) through a DDR memory bus, and operates as a standard Registered DIMM (RDIMM) for the CPU, whereas ULL-Flash is connected to MCH via a PCIe root complex as block storage.

Persistent memory. There are three standard incarnations of persistent memory, NVDIMM-N [31], -F [54] and -P [16]. Table I summarizes the key differences between these three types of persistent memory and our design. NVDIMM-N is a JEDEC standard for a persistent memory module, which includes DRAM devices, a supercapacitor, multiplexers, and a small flash device. The supercapacitor is used as an energy source for the DRAM backup operations when a power failure occurs. The multiplexers are located between the DRAM and the standard DIMM connector to a memory bus, and they isolate the DRAM from the memory bus when backup and restore operations take place. The flash, as a backup storage medium, has the same capacity as the DRAM, and it is invisible to users. While the host directly accesses the DRAM of NVDIMM-N, its controller internally migrates DRAM data to flash upon a power failure and this migration typically takes tens of seconds [31]. The controller restores the data from flash to the DRAM on the next boot, thereby providing non-volatility. In contrast, NVDIMM-F consists of multiple flashes without DRAM. Since it is normally used as block storage, NVDIMM-F requires both file system and OS support, similar to conventional SSDs. NVDIMM-P combines the design strategies of NVDIMM-N and NVDIMM-F, and employs a byte-addressable interface. However, NVDIMM-P such as Optane DC PMM exhibits 6× lower performance than DRAM, and does not allow direct access to its internal DRAM as well as requires OS-level support to enable persistent memory

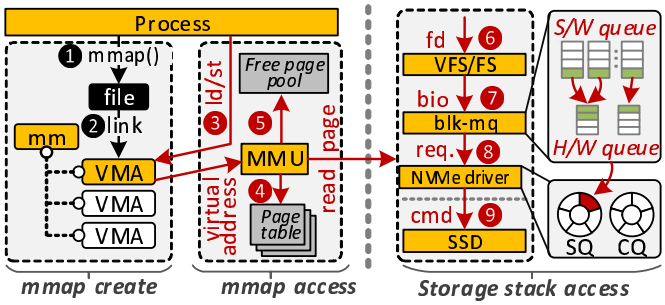


Fig. 3: Software support.

accesses. This by far makes NVDIMM-N the only persistent memory that supports DRAM performance with the byte-addressability. Considering this aspect, in this paper, we use the terms "NVDIMM" and "NVDIMM-N" interchangeably.

Storage. All the high-performance SSDs, including ULL-Flash, are connected to another part of the MCH, PCIe root complex. A PCIe lane is also treated as a memory bus in modern computer systems, but it transfers 4KB or larger data packets between the CPU and the SSD for I/O transactions. Since the granularity of I/O accesses is a page or a block, user applications can only access the underlying SSD by going through the entire storage stack of the OS, which includes an I/O runtime library, file system, and block layer, atop an NVMe driver. The NVMe driver manages the transfers of data packets over PCIe, and communicates with the NVMe controller in the SSD through the PCIe *baseline address registers* (BARs), including doorbell registers, queue attributes, target addresses for each queue, and the NVMe controller information [10]. The internal hardware details of ULL-Flash will be explained in Section II-C.

B. Support for Persistent Memory Expansion

Figure 3 illustrates the software support and storage stack that user applications require for expanding NVDIMM with SSD. The *memory-mapped file* (MMF) module in Linux, also referred to as `mmap`, can be used to expand the persistent memory space of NVDIMM with SSD. If a process calls `mmap` with a file descriptor (`fd`) for SSD (❶), the MMF creates a new mapping in its process address space, represented by a memory management structure (`mm_struct`), by allocating a virtual memory area (VMA) to the structure (❷). In other words, the MMF links `fd` to the VMA, by establishing a mapping between the process memory and the target file. When the process accesses the memory designated by the VMA (❸❹❺), this triggers a page fault (if the data is not available in NVDIMM).

When a page fault occurs, the page fault handler is invoked and allocates a new page to the VMA. Since the VMA is linked to the target file, the page fault handler retrieves the file metadata (`inode`) associated with `fd` and acquires a lock for its access (❻). The MMU interacts with a fault handler of the file system to read a page from the SSD. The file system initializes a block I/O request structure, called `bio`, and submits it to the multi-queue block I/O queueing (*blk-mq*) layer, which schedules I/O requests over multiple software queues (❼). Depending on the design of the target system, one or more software queues can be mapped to a hardware dispatch queue (❽), managed by the NVM controller that exists within the SSD (❾). Once the service of the I/O request (i.e., `bio`) is completed, and the actual data is loaded into a new region of the allocated page memory, the page fault handler creates a page table entry (PTE), records the new page address in the PTE, and resumes the process.

The MMF can be used to expand the persistent memory space of NVDIMM with SSDs. However, this approach can potentially negate most of the benefits that would be brought by ULL-Flash, due to the high overheads caused by page faults, file systems, context switching, and data copies.

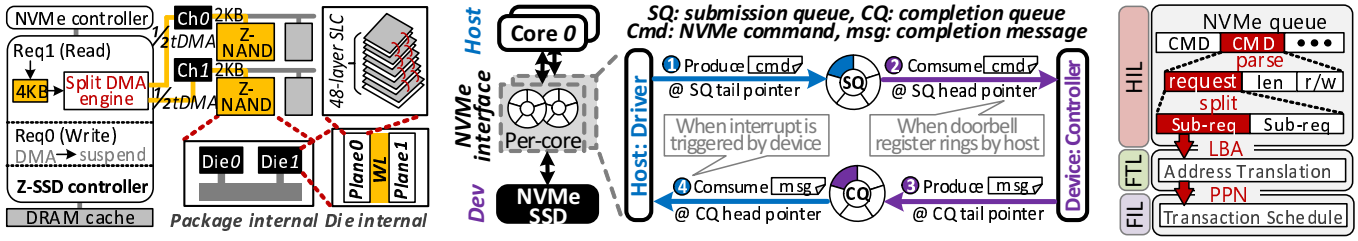
C. ULL-Flash

Hardware details. All state-of-the-art SSDs typically employ a large number of flash packages and connect them to multiple system buses, referred to as *channels*. Each flash package contains multiple dies and planes for fast response time and low latency, as illustrated in Figure 4a. To deliver massive parallelism and hence high I/O performance, an SSD spreads a given set of I/O requests from the host across multiple channels, packages, dies, and even planes.

ULL-Flash also adopts this multi-channel and multi-way architecture, but it optimizes the datapath and channel stripping [9]. More specifically, ULL-Flash splits a 4KB I/O request from the host into two operations and issues them to two channels simultaneously; doing so can effectively reduce the DMA latency by half. In addition, while most high-performance SSDs employ multiple-level cell (MLC) or triple-level cell (TLC), ULL-Flash employs a new type of flash medium, called *Z-NAND* [9]. Z-NAND leverages a 3D-flash structure to provide a single-level cell (SLC) technology and optimizes the I/O circuitry and memory interface to enable short latency. Specifically, Z-NAND uses 48 stacked word-line layers, referred to as the vertical NAND (V-NAND) architecture, to incarnate an SLC memory. Thanks to its unique flash architecture and advanced fabrication technology, the read and write latencies of Z-NAND (i.e., $3\mu s$ and $100\mu s$) are $15\times$ and $7\times$ lower than the V-NAND flash memory, respectively [55].

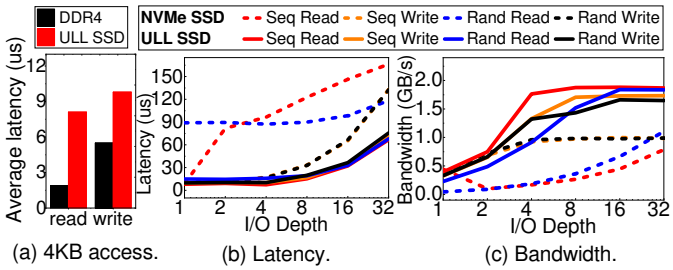
ULL-Flash employs large DRAM in front of its multiple channels and exposes its internal parallelism, low latency, and high bandwidth (through the NVMe interface), which are managed by multiple interface controllers and firmware modules. Note that the DRAM management is tightly coupled with handling the NVMe protocol. Based on the definition of NVMe, the same data can be in both the host-side DRAM and the SSD-internal DRAM after the underlying ULL-Flash controller or firmware performs DMA for data transfer.

I/O connection to CPU. Figure 4b illustrates the per-core NVMe queue and communication protocol. An NVMe queue consists of a pair of *submission queue* (SQ) and *completion queue* (CQ), each with 64K entries [22]. These are simple FIFO queues, and each entry is referenced by a *physical region page* (PRP) pointer [10]. If the request size is larger than a



(a) SSD internal parallelism. (b) NVMe queue and protocol management. (c) Flash firmware.

Fig. 4: Overview of ULL-Flash and NVMe datapath.



(a) 4KB access. (b) Latency. (c) Bandwidth.

Fig. 5: ULL-Flash versus NVMe SSD.

4KB NVMe packet, the data can be referenced by a list of PRP pointers instead of a single PRP pointer. When a request arrives at the SQ, the host increments its tail (pointer) and rings the corresponding doorbell of ULL-Flash, so that the NVMe controller can synchronize the storage-side SQ, which is logically paired with the host-side SQ. Since the data for each entry exist in the host-side DRAM (pointed by a PRP pointer), the ULL-Flash handles the DMA for the I/O request, and then the underlying Z-NAND and firmware serve the request. Once the service is completed, the NVMe controller moves the tail of the CQ (paired with the SQ) and informs the host of the event over a message signaled interrupt (MSI). The host then jumps to an interrupt service routine (ISR) and synchronizes the CQ tail. The ISR completes the request, advances the head of the CQ (and releases the buffer data), and rings the doorbell to notify the ULL-Flash of the completion of the host-side I/O processing. Finally, the NVMe controller releases the internal data and advances the head pointer of the CQ. The NVMe interface has no knowledge of the data cached in the host-side DRAM, while the data for each I/O request can reside in the host-side DRAM. Therefore, even if I/O requests can be served by the host-side DRAM, the NVMe interface obliviously enqueues the requests and processes them.

Firmware. Figure 4c shows a general firmware architecture implemented in ULL-Flash. At the top of the firmware layers within the ULL-Flash, the *host interface layer* (HIL) is responsible for parsing the NVMe commands and managing the queues by collaborating with the internal NVMe controller [32]. This layer also splits an I/O request, which can be of any length, into sub-requests. The size of a sub-request matches the unit I/O size that the underlying firmware module manages. The parsed separate requests are forwarded to the *flash translation layer* (FTL) [35]. The FTL translates a given *logical block address* (LBA) to a *physical page number* (PPN). After translating the address of each sub-request into a PPN, the flash interface layer (FIL) submits the request and manages its transactions, which constitutes multiple flash commands such as row/column addresses, I/O commands, administrative commands, and DMA transfers. During this I/O processing, FTL/FIL can stripe the requests across multiple internal resources (*e.g.*, channels, packages, dies, planes, etc.), achieving both low latency and high bandwidth.

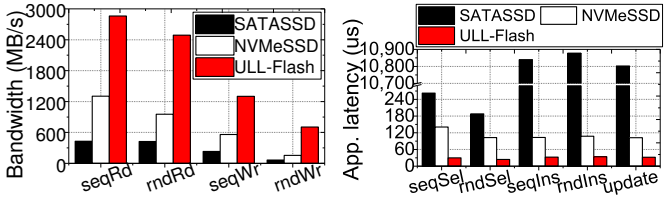
III. Motivation and Challenges

In this section, we explain why ULL-Flash can be used for a large working memory solution, and discuss what challenges the conventional software-assisted solutions face to expand the persistent memory by integrating NVDIMM with ULL-Flash.

A. ULL-Flash Performance Characterization

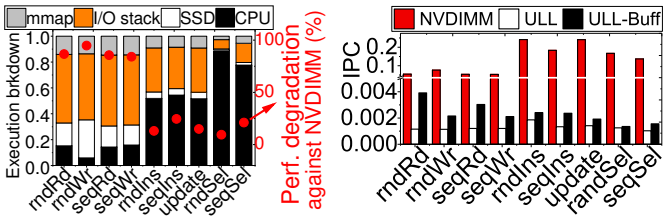
We evaluated a real 800 GB Z-SSD *prototype* ([56] as ULL-Flash) and analyzed its performance characteristics. We then compared the performance characteristics of ULL-Flash with those of a high performance NVMe SSD (Intel NVMe 750 [25]) using a Flexible I/O Tester [3]. Both the devices use four PCIe3.0 lanes (1GB/lane) and are evaluated by a system that has a single 4GHz CPU [23]. The collected performance characteristics are plotted in Figure 5, under the sequential and random read/write accesses. We also evaluated the performance with varying I/O queue depths (1∼32). The request sizes equal to that of the NVMe packet payload (4KB).

As shown in Figure 5a, we observe that ULL-Flash exhibits 8 μs and 10 μs for 4KB read and write latencies with 1∼4 queue depth at the user-level. That is, such read and write latencies of ULL-Flash are only 3.3× and 79% longer than the real read/write latencies (4KB-sized) of single DDR4-2133 DIMM [64] on the same testbed. This significant latency advantage makes ULL-Flash a promising replacement for conventional SSD to expand the persistent memory space of NVDIMM with storage. As shown in Figure 5b, ULL-Flash maintains such latency characteristics under different I/O depths in a predictable and sustainable manner, while NVMe SSD experiences significantly increased latencies as the I/O depth increases (up to 155 μs). Figure 5c compares the bandwidth trends of ULL-Flash with those of NVMe SSD. For read and write accesses, ULL-Flash offers 115% and 137% higher average bandwidth than NVMe SSD. These plots also indicate that ULL-Flash reaches its peak bandwidth with only a few NVMe commands, whereas an NVMe SSD does not achieve such peak bandwidth for random read accesses, even



(a) mmap-bench. (b) SQLite.

Fig. 6: MMF-based system performance.



(a) Software overheads. (b) IPC of bypassing.

Fig. 7: Challenges of using ULL-Flash.

if we increase the queue depth to 32 entries. We also observe that *the number of requests in ULL-Flash waiting queue is 4 for most accesses (cf. Figure 5c)*. ULL-Flash can support only 16 outstanding requests for random read accesses. We believe that this characteristic can make the NVMe queue management simple and amenable to be implemented in hardware.

B. Software-Based Memory Expansion

To evaluate the performance of an existing software-based memory expansion, we configure an MMF-based host system with the real devices. Our evaluation system integrates three SSDs (including a SATA SSD [24], in addition to ULL-Flash and NVMe SSD) and employs two 8GB DRAM ranks, each with eight banks operating at 1.6GHz. The ULL-Flash is used to expand memory space over `mmap`.

Benchmarks. We use `mmap-benchmark`, which is designed to evaluate the performance of `mmap` with a set of microbenchmarks [33]. Each of `seqRd` and `seqWr` creates a single thread, and then performs sequential read and write operations. In contrast, each of `rndRd` and `rndWr` creates four threads, each simultaneously performing random read and write operations. We also tested `SQLite-benchmark`, which is a benchmark for a widely-used DBMS (`SQLite`) [14]. The workload details will be explained in Section VI-A.

Performance. Figures 6a and 6b show the performances of `mmap-benchmark` (bandwidth) and `SQLite` (transaction latency), respectively. As shown in Figure 6a, ULL-Flash exhibits 399% and 118% higher bandwidth than SATA and NVMe SSDs, respectively, in the MMF-system. `seqRd` and `seqWr` exhibit the performance near the peak bandwidth of the SSDs [24], [25], [56], but they significantly degrade performance while executing `rndRd` and `rndWr`. This is because `seqRd` and `seqWr` pull the data in a sequential manner to the file system's buffer cache, and this helps us hide the performance degradation of SSDs for byte-based I/O accesses. In addition, the average I/O queue depths of `mmap-benchmark` and `SQLite` are one and four, respectively, which can better leverage the benefits of read ahead. Similarly, Figure 6b shows that the average latency of ULL-Flash for `SQLite` (per transaction) is lower than those of SATA and NVMe SSDs by 95% and 72%, respectively.

Analysis on ULL-Flash overhead. Figure 7a further decomposes the total execution time of user applications into a mmap processing time (i.e., context switch and page fault handling), an I/O stack time (i.e., filesystem, blk-mq layer, and NVMe driver), a ULL-Flash access time, and an application computation time. For better understanding, the figure also analyzes how much the ULL-Flash-based MMF system degrades the overall performance, compared to the NVDIMM-based system. Since `rndSel` and `seqSel` spend most of their execution on the DBMS-side computation, their CPU cycles account for 83% of the total execution time, on average. However, the remaining workloads in `mmap-benchmark` and `SQLite` take 13% and 53% of the execution, respectively, while ULL-Flash only accounts for 13% of the total latency, on average. Note that the system overhead imposed by MMF (`mmap` and I/O stack) accounts for 69% of the total execution time. This is because MMF is involved in many software operations including multiple page fault handling, context switches, address translations (i.e., page table, filesystem and FTL), boundary checks, and permission checks [20]. The context switches are one of the main contributors to increase I/O latency [40]. On the other hand, the queuing mechanism and NVMe communication protocol in I/O stack are optimized for throughput rather than I/O latency [68]. The software operations of MMF consume 15∼20 us [20], which is around 6× longer than ZNAND access latency (3 us).

C. Software Overhead

To remove the software overheads brought by MMF, we can *bypass* the entire storage stack and simulate the underlying ULL as a memory module to directly serve the load/store instructions. Figure 7b shows the CPU-side performance for three different bypass strategies: (1) NVDIMM only (`NVDIMM`), (2) ULL-Flash (`ULL`), and (3) ULL-Flash with a page buffer, which is essentially a small DRAM (`ULL-buff`). For this evaluation, we use the same workloads used in Section III-B. The results collected with `mmap-benchmark` indicate that the average instructions per cycle (IPC) values for the `ULL`- and `ULL-buff`-based systems are only 0.001 and 0.003, respectively, while the `NVDIMM`-based system offers an average IPC of 0.06 (i.e., 98% and 95% degradation).

When evaluating `SQLite`, we observed that `ULL` and `ULL-buff` degrade IPC compared to `DRAM` by 140× and 101×, respectively. The load and store instructions take 51% of the total number of executed instructions, and all these load/store instructions for the workloads we tested are due to the relatively long ULL-Flash operations. Note that the Z-NAND latency ($3\mu s$) is much shorter than that of conventional flash, but it is 3.3× longer than the latency of NVDIMM for 4KB access. While a page cache can potentially hide the page access delay, we observe that a large fraction of the load/store instructions suffer from the page cache misses, due to the poor data locality exhibited by `mmap-benchmark` and `SQLite`.

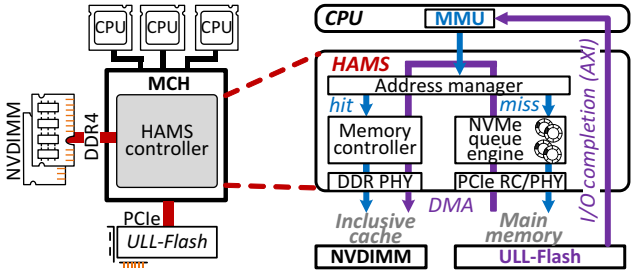


Fig. 8: Overview of baseline HAMS.

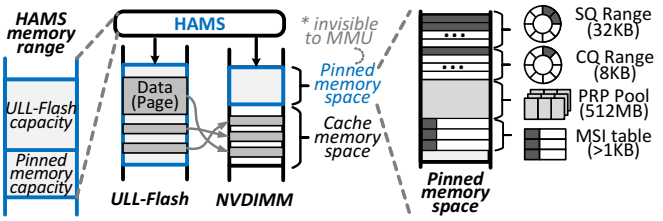


Fig. 9: Address management.

The goal of HAMS is to (1) remove the `mmap` and storage stack overheads from the MMF-system and (2) reduce the number of stalled instructions by caching the memory references in NVDIMM directly and by automating the mapping between ULL-Flash and NVDIMM.

IV. Memory Over Storage

HAMS is aimed to automate all necessary hardware for the expansion of the persistent memory by integrating NVDIMM and ULL-Flash, while reducing the energy consumption and maintaining the consistency on the resulting heterogeneous memory space. In this section, we give an overview of the baseline design and aggressive integration of HAMS.

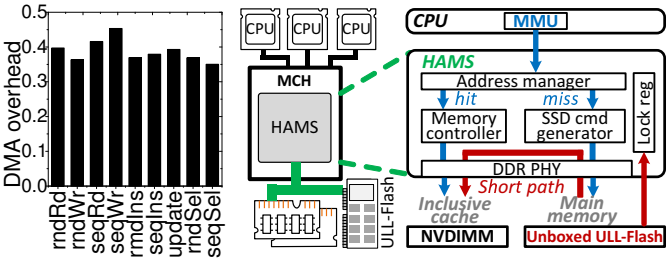
A. HAMS Overview

Figure 8 shows the baseline architecture of HAMS. HAMS resides in MCH, which implements an address manager, an NVDIMM memory controller and PCIe root complex. The address manager offers a 64-bit byte-addressable address space by exposing the storage capacity of ULL-Flash to MMU. It also utilizes a memory space of NVDIMM as an inclusive cache for ULL-Flash with an integrated tag-array. To implement MoS, the address manager employs a simple hardware cache logic that coordinates NVDIMM and ULL-Flash to serve incoming memory requests. Note that a memory request can be generated by either MMU or ULL-Flash, and thus, they should be processed differently. When the NVMe controller (in ULL-Flash) generates a memory request, the NVMe controller extracts the NVDIMM address of the target data by referring to PRP(s) that the address manager handles and records it in the request. HAMS then directly forwards the request to access NVDIMM based on the recorded NVDIMM address. On the other hand, HAMS checks the memory address of MMU's request by examining its MoS tag-array. If the requested memory address hits in the MoS tag-array, the request is directly served by the data from NVDIMM. Otherwise, HAMS secures an NVDIMM space to accommodate the incoming request by evicting data to ULL-Flash. HAMS also fetches target data from ULL-Flash to NVDIMM for read requests. Once the data transfer from ULL-Flash to NVIDMM (or vice versa) is completed, HAMS informs MMU of the completion so that MMU can retry the stalled instruction.

B. ULL-Flash Archive Management

The power failure management for persistency control is central to a key design of HAMS. While NVDIMM's data is stored and restored by its on-board NVM controller, NVMe storage employs a different mechanism to handle power failures. Specifically, the data persistency and I/O atomicity of an SSD are guaranteed by a file system. The file system and other kernel related components in typical support persistency using journaling [65]. Since HAMS removes the MMF and file system support, the data in the SSD-internal DRAM can be lost upon a power failure. While HAMS can enforce data persistency by tagging force unit access (FUA) per request [60], doing so can significantly degrade the ULL-Flash performance. This is because FUA enforces serializing all outstanding requests to be written to the underlying flash media. Another issue in the design of HAMS is related to hardware implementations of the NVMe protocol. Since NVMe data structures, including SQs, CQs, and BARs, are mapped to a memory region of NVDIMM, all data and queue information can be unintentionally overwritten by any of user applications or the OS. This can make the hardware-based NVMe management in HAMS vulnerable. In addition, the data in NVDIMM can be inconsistent if HAMS and ULL-Flash simultaneously access the same page frame.

We propose a set of designs to address the aforementioned challenges. Specifically, to protect data against power failure, we integrate super-capacitors in ULL-Flash to flush data in the volatile DRAM buffer to the persistent flash media. We also utilize the NVMe data structure to recover the I/O requests, which are corrupted by power loss. On the other hand, to resolve the vulnerability issue of the NVMe data structure, we pin a specific memory region of NVDIMM to store the NVMe data structure and make it invisible to the MMU. As shown in Figure 9, this pinned memory region includes the ring buffers for the SQs and CQs, PRP pool and MSI table, and it is mapped to the upper memory part of NVDIMM (around 512MB in our design). On the other hand, the remaining NVDIMM memory area is mapped to the MoS address space by HAMS. During the initialization process, HAMS reviews the pinned memory region, in particular the SQ and CQ buffers including the head and tail pointers. If there is no power failure, the SQ and CQ tail pointers should refer to the same offset of their queue entries to avoid a violation of NVMe queue management and consistency [21]. However, if a power failure occurs, HAMS is able to detect all pending requests by checking the offset differences between the SQ/CQ tail pointers in the MMU-invisible space of NVDIMM (cf. Figure 9). During the power restoration, HAMS needs to reissue all pending requests to the underlying ULL-Flash for data persistency and consistency. To protect the memory to which



(a) DMA overhead. (b) Aggressive integration.

Fig. 10: Challenges and aggressive integration.

the data is being transferred, HAMS keeps track of the DMA status by configuring a bit per each entry of the MoS tag-array, which is referred to as *busy bit*. This bit is set to 1 whenever the NVMe engine issues a command, and it can be cleared when HAMS updates the CQ's head pointer. Thus, if the busy bit is set, HAMS will exclude the corresponding page from being evicted. This guarantees that the data is consistent when ULL-Flash accesses the page frame via PRP.

C. Aggressive Integration of HAMS

The baseline design of HAMS explained so far includes a hardware automation of cache logic in the MCH by leveraging the conventional DDR and PCIe controllers, thus offering a large working memory space. While this design strategy does not require any modification to the existing storage and memory devices, it brings two inefficiencies from an architectural perspective: (1) the overheads imposed by data transfer and (2) the energy inefficiency brought by the SSD-internal DRAM. First, in case of a cache miss, the target data needs to go through the DDR4 module (e.g., the memory controller and DDR4 PHY) and the PCIe module (root complex, transaction layer, data link layer and physical layer). While the peak bandwidth of DDR4 [59] is 20 GB/s per channel, ULL-Flash (including most NVMe SSD products) uses PCIe 3.0 with 4 lanes, which makes the maximum bandwidth of NVMe 4 GB/s. Thus, in case of a cache miss, the performance of HAMS can be capped by the peak PCIe bandwidth. In addition, the raw data of NVDIMM should be encoded and encapsulated into a PCIe packet, which also makes the HAMS latency longer in case of a cache miss.

Figure 10a shows the fraction of data movement latency in the average memory access time (AMAT) under the execution of the workloads selected from Section III. It can be observed that the interface latency taken by moving data between the NVMe controller and the DDR4 controller constitutes 39% of the total AMAT, which can degrade the HAMS performance. Another drawback of the baseline design of HAMS is that, even if HAMS already holds the data in NVDIMM, the data will still be copied to the SSD-internal DRAM. While this would improve performance under the block storage use-case (with a file system), it would also introduce extra energy consumption and increase the internal complexity of the SSD. Note that the SSD-internal DRAM requires 17% more power than a flash complex consisting of 32 flash chips.

To address these challenges, we propose to remove the SSD-internal DRAM that is used for data buffering, introduce a new register-based interface (instead of doorbell registers and PCIe BARs), and connect ULL-Flash to DRAM PHY (instead of PCIe). Note that writes to ULL-Flash are already reduced without employing the SSD-internal DRAM as the incoming data are buffered/cached by NVDIMM. Similarly, the address mapping table is also buffered in the NVDIMM. Accessing the mapping information only consumes a *tCL* and a few *tBURST* periods (less than 20ns), which is ignorable compared to the long ULL-Flash access latency. As shown in Figure 10b, this aggressive integration of NVDIMM and ULL-Flash, which we call advanced HAMS, allows the NVMe controller to directly access the DRAM modules over the DRAM interface. Specifically, to be compatible with the synchronous DDR4 interface, the NVMe controller avoids unpredictable delay of the underlying Z-NAND accesses by employing a set of registers to buffer the command, address, and data. For communications, the address manager employs an SSD command generation logic that writes a set of registers capturing the source and destination addresses and I/O command, based on the I/O request that HAMS needs to initiate. The NVMe controller fetches (or pulls) the target data from the source address of NVDIMM (written to the address register via the DRAM interface) and then forwards it to flash firmware so that it can be programmed into flash media.

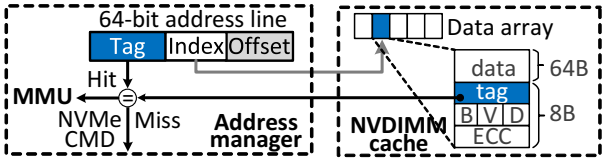


Fig. 11: MoS tag-array design in NVDIMM cache.

While allocating multiple DDR4 channels to connect each pair from HAMS controller, NVDIMM and ULL-Flash can parallelize the MMU operations and ULL-Flash read/write accesses, this design also makes DDR4 channels underutilized. To avoid wasting the channel resources, we propose to connect ULL-Flash and one/multiple NVDIMMs to HAMS controller via the same DDR4 bus. However, one of the key design issues is that NVDIMM can be accessed by both the HAMS controller and NVMe controller in our design. To avoid simultaneous accesses from these two, this aggressive integration also introduces a *lock register*, which indicates that the NVMe controller is in the process of accessing DDR4 and NVDIMM for data transfers.

V. Implementation Details

A. NVDIMM Cache and Bus Integration

HAMS address management. An SRAM-based MoS tag-array can expose a significant circuit area cost to the HAMS controller and raise the concern of metadata persistency when a power failure occurs. Instead, we configure MoS' NVDIMM cache as direct mapped and integrate its tag information along with ECC bits in each NVDIMM cache line, which is similar to the MCDRAM configuration of Intel Knights Landing processor [58]. Figure 11 shows details of our MoS tag-array in the NVDIMM cache. Each entry of the MoS tag-array

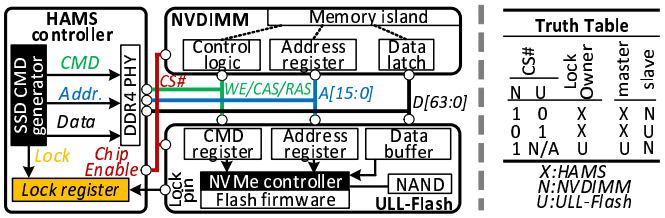


Fig. 12: The details of the register-based interface.

contains all metadata of the cache, such as the tag, busy bit (B), valid bit (V) and dirty bit (D). When there is an incoming memory request, its address is decomposed into the tag, index, and offset fields. HAMS address manager then retrieves the tag-array entry and the data block from the NVDIMM cache by using the decomposed index. A comparator pulls the stored tag from the retrieved tag-array entry and compares it with the tag of the corresponding memory request. If the two tags match, the fetched data can be directly served from HAMS controller. On the other hand, if the two tags mismatch, HAMS composes two NVMe commands, one for a read that fills data from ULL-Flash to the NVDIMM cache entry, and another for a write that evicts the data from NVDIMM to ULL-Flash. Once the target data are available in the NVDIMM cache, HAMS places it on the system bus, and notifies the completion to CPU by setting the MMU's command and address buses.

Register-based interface. Figure 12 illustrates how our advanced HAMS controller communicates with the underlying NVMe controller through DDR4. In our design, to send an I/O request to NVMe controller, the HAMS controller firstly deselects the NVDIMM by toggling its *CS#* strobe to high voltage. In the next clock cycle, the HAMS controller configures the write command in the DDR channel by toggling the *WE#*, *CAS#* and *RAS#* strobe to low, low and high voltage, respectively. Following the write command, the I/O request, which is packeted as a 64B NVMe command, is transferred to the data buffer registers of ULL-Flash via the *D[63:0]* strobes in 8-cycle data burst. The NVMe controller then extracts and parses the request information (i.e., request type, source/destination addresses and data length) from the data buffer registers, similar to most NVMe SSDs. Note that, unlike their original purpose, the address strobes *A[15:0]* deliver no information during the communication between HAMS controller and NVDIMM controller.

After a given number of cycles for processing the NVMe write command or fetching data from the flash media based on NVMe read command, HAMS sets the lock register to 1, which indicates that the NVMe controller can take over the control as a bus master. If the lock register is configured, the NVMe controller initializes the DMA procedure between ULL-Flash and NVDIMM based on the timing sequence of the DDR4 interface. After transferring data, the NVMe controller releases the lock register by resetting it to 0. HAMS cache logic uses the lock register for NVDIMM accesses, which helps us avoid a case where both NVMe controller and memory controller use the bus at the same time.

B. NVMe and Hazard Management

The I/O requests for each NVMe queue entry can be simply composed by filling the information fields of the NVMe command structure. HAMS writes the opcode field for a given request (read/write), and fills the NVDIMM address, SSD address, and page size (4KB) into the corresponding PRP, LBA, and length fields, respectively. The generated NVMe command is enqueued in the SQ by the HAMS NVMe engine. This engine writes the doorbell to inform the ULL-Flash of a request. Whenever the interrupt is delivered from the ULL-Flash controller of HAMS, the NVMe engine synchronizes the corresponding CQ and clears the target entries of CQ and SQ.

There are two issues associated with this NVMe management, as NVDIMM is used both as a cache and as a PRP target: (1) *eviction hazard* and (2) *redundant eviction*. The eviction hazard occurs when the NVMe controller and HAMS cache logic access the same NVDIMM location, whereas the redundant eviction arises when the cache logic generates an eviction command, which is already being issued. Consider the example illustrated in Figure 13. The MMU requests a read at `0xF0` of the MoS address space, the index and tag of which are `0x0` and `0xF`, respectively. Since a cache miss occurs, the HAMS cache logic evicts the exiting page (`0xE0`) to ULL-Flash, and requests a data read at `0xF0`. In the meantime, the MMU accesses `0xF0` to update the data. This makes the cache logic evict the same data again, because the evicted request is still serviced by ULL-Flash (i.e., redundant eviction). Now, the HAMS NVMe engine contains three NVMe commands (`CMD1/2/3`). These commands are processed by the NVMe controller in a FIFO order based on the NVMe specification. However, I/O completions within ULL-Flash can be out-of-order, due to SSD-internal tasks. More importantly, the NVMe controller transfers the data to NVDIMM based on the order of completion, which can cause an eviction hazard.

To prevent these hazards and redundant evictions, HAMS employs two techniques. When the NVMe engine issues commands, HAMS isolates the target contents from the corresponding NVDIMM cache entry by cloning the corresponding page into the PRP pool allocated in the pinned memory (Figure 9). It then updates the PRP value with the location of the cloned page so that the underlying NVMe controller does not make the data inconsistent during DMA. Further, we add a wait queue to the pinned memory, and make HAMS always refer to a busy bit (cf. Section IV-B) of the MoS tag-array, whenever a cache miss occurs. The HAMS cache logic sets the bit to 1 and then resets it to 0, when the NVMe engine completes the request. Figure 14 shows an example that illustrates how the eviction hazard and redundant eviction issues are handled. When a cache miss occurs (read at `0x0E`), the cache logic toggles the busy bit of the target tag-array's entry and copies the target page to a PRP pool entry. During this process, HAMS replaces the reference to PRP with the PRP pool entry and submits it to the NVMe engine. Upon the next cache miss (write to `0xF0`), the cache logic realizes that the entry is in an eviction process, and puts the request into

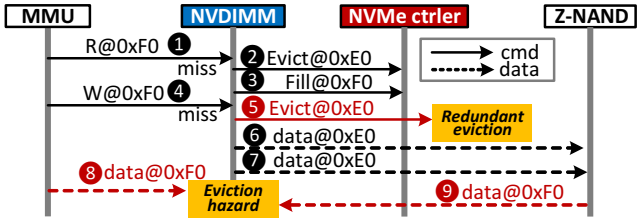


Fig. 13: Challenges with the baseline HAMS.

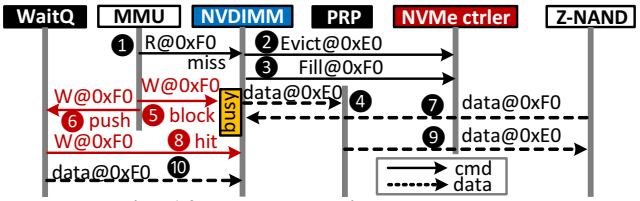


Fig. 14: Hazard avoidance methods.

the wait queue. After the I/O services of the NVMe commands are completed, the busy bit is cleared, and the request that sits in the wait queue is issued again. In this way, the eviction hazard and redundant eviction issue with the wait queue and busy bit can be avoided.

C. Persistency Control Upon Power Failure

A target system can benefit from a large memory space, if it utilizes HAMS as a working memory expansion, which expands the address space by combining NVDIMM and ULL-Flash. However, it needs a guarantee for *data persistency*, as MoS address space is considered as a persistent memory expansion. Thus, HAMS requires to flush the NVMe request whenever its cache logic needs to update data in ULL-Flash. To address this shortcoming, we add a *journal tag* to each SQ's NVMe command entry by utilizing the reserved area in the NVMe command format. This journal tag keeps information that indicates whether the corresponding request is completed by ULL-Flash. Whenever the NVMe engine sends a request to ULL-Flash, it sets the tag to 1. Once the interrupt arrives, HAMS clears the tag associated with the I/O completion.

Figure 15 gives an example that illustrates how HAMS utilizes the journal tag information. In the first phase of this example, HAMS issues all the commands in the SQ to ULL-Flash, and `CMD1`, `CMD3` and `CMD4` are processed as the tail and head pointers refer to the same location in the SQ/CQ, which clears the corresponding journal tags to 0. Upon a power failure at the end of the first phase, ULL-Flash and HAMS cannot finish `CMD2`. Since the pinned memory space of NVDIMM holds the data of the SQ region in our design, HAMS first checks the SQ region on power-up to determine if there is any command whose journal tag is 1. If there is one, HAMS pulls the command and creates a pair of SQ and CQ for the I/O service in second phase. HAMS then restores it to the SQ, increases the SQ's tail pointer, and rings the doorbell register, so that the outstanding request issued at the moment of a power failure can be served appropriately.

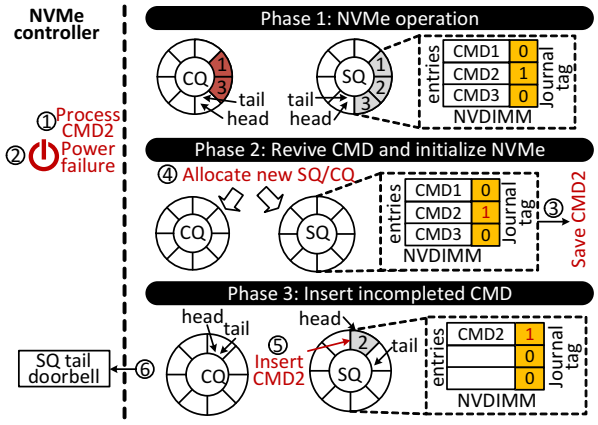


Fig. 15: Power failure recovery procedure.

VI. Evaluation

A. Experiment Setup

Simulation model. To explore the full design space of the HAMS enabled systems from both the software and hardware perspectives, we first replace the existing main memory implementation in a full system simulator (gem5 [6]) with the latency model of an 8GB DRAM-based NVDIMM [11]. We then model an ULL-Flash archive and integrate it into gem5 by revising the memory controller and I/O bridge model[1]. The storage-side components of the proposed simulator are configured as ULL-Flash instances by leveraging an existing SSD simulator, Amber [15], which is highly reconfigurable (being aware of the details of flash internals, SSD internals and parallelism-related design parameters) and detailed (implementing a full firmware stack and an actual NVMe interface). Our simulator has been verified with an actual 800GB ULL-Flash prototype [55]. Note that this proposed simulation framework enables the execution of data-intensive applications on a real Linux, while allowing us to investigate the full design space on the datapath from top to bottom. The details of our simulation environment are given in Table II.

Experiment precondition and energy profiling method. To guarantee the consistency of our experimental results, we completely wrote all data-blocks into the flash-media, and flushed/cleaned up the internal-DRAM in a *warm-up phase* before performing our evaluations. The energy estimation of each component in the full-system platform is performed based on its power model; more specifically, the power models of ULL flash and NVDIMM are derived based on NAND flash datasheets and MICRON SDRAM power calculator [47], which will be available for download (along with our simulator), while the energy consumptions of core and cache are measured by leveraging McPAT [43].

Benchmarks. We evaluate 12 data-intensive workloads from MMF microbenchmark [33], Rodinia [7], and SQLite [13] benchmarks. While MMF microbenchmark is memory-intensive, Rodinia benchmark requires high computation. In addition, MMF microbenchmark accesses the persistent mem-

[1]All source codes of our full-system simulation that integrates high-fidelity SSD storage models will be made available for download in public domain.

OS	Linux 4.9, Ubuntu 14.10
CPU	quad-core, ARM v8, 2GHz
Cache	64KB L1I/64KB L1D/2MB L2
memory	NVDIMM, DDR4, 8GB, 128KB page
storage	ULL-Flash, 512MB buffer, 800GB
flash	3us read, 100us write

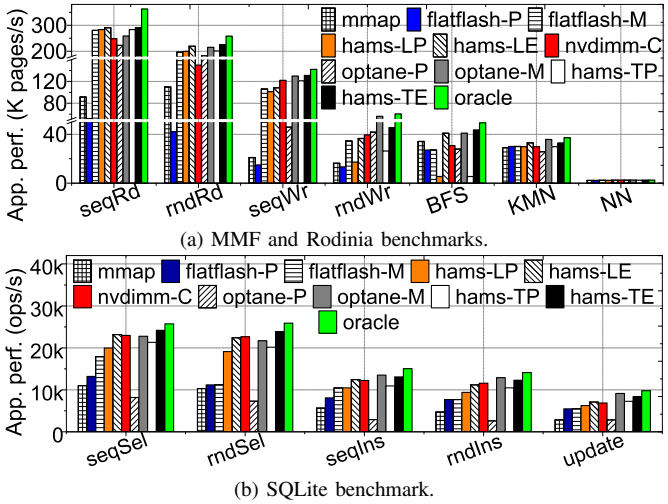
TABLE II: Gem5 specification.

Benchmark	Microbenchmark				SQLite benchmark					Rodinia		
Workloads	seqRd	rndRd	seqWr	rndWr	seqSel	rndSel	seqIns	rndIns	Update	BFS	KMN	NN
# of inst.	67G	69G	67G	69G	213G	213G	40G	44G	244G	192G	38G	145G
load inst. ratio	0.28	0.27	0.28	0.27	0.26	0.26	0.25	0.25	0.26	0.21	0.27	0.16
store inst. ratio	0.43	0.37	0.43	0.37	0.20	0.20	0.21	0.21	0.20	0.04	0.03	0.05
Data sets	16GB	16GB	16GB	16GB	11GB	11GB	11GB	11GB	11GB	9GB	5GB	7GB

TABLE III: Workload characteristics.

ory system in a coarse-granular fashion (i.e., by pages). In contrast, the other workloads generate fine-granular memory accesses ranging from 8B to 100B. In our experiments, the datasets to be tested initially reside in either ULL-Flash or HAMS. To access data, these workloads are structured to support memory-mapped file I/O via the POSIX-compliant system call `mmap`. Table III tabulates the important characteristics of our benchmarks such as the total number of instructions, fraction of load/store instructions, and dataset sizes.

Simulation platforms. We configured a traditional computer system, called `mmap`, as our baseline for evaluation. `mmap` employs an ULL-Flash and a DDR4 DRAM as its storage and memory media, respectively. Table II shows important parameters of our system configurations. By default, the baseline accesses data directly from the persistent storage by using the MMF module. We also built five computing platforms employing the existing memory expansion techniques [1], [42], [66] and four different systems that implement our `HAMS` model. Specifically, (1) `optane-P` [29] employs 512GB Optane DC PMM as main memory. To guarantee data persistency, Optane DC PMM operates in App Direct mode that serves all memory requests without DRAM cache. (2) `optane-M` [29] employs 8GB DRAM as the cache of Optane DC PMM, which can improve the performance but sacrifice the data persistency. (3) `flatflash-P` [1] allows the applications to directly access a cache line from ULL-Flash via MMIO [4] thereby guaranteeing the data persistency. (4) Compared to `flatflash-P`, `flatflash-M` [1] selectively buffers hot pages in 8GB host-side memory for fast accesses. (5) `nvdimm-C` [42] connects ULL-Flash to DRAM PHY thereby sharing the memory channel with DRAM. `nvdimm-C` uses DRAM as a cache of ULL-Flash. However, data migration between DRAM and ULL-Flash is only allowed during DRAM refresh periods. (6) A loosely-coupled HAMS system, which connects to 8GB NVDIMM and 800GB ULL-Flash via a memory channel and PCIe links, respectively, is referred to as `hams-L`. `hams-LP` is the loosely-coupled HAMS system, which works in a "persist mode" to persistently store data. `hams-LP` tags FUA per I/O request and enforces at most a single I/O request on-the-fly. (7) `hams-LE` is also the loosely-coupled `HAMS` system, but it operates in an "extend mode". In particular, `hams-LE` leverages the NVMe protocol to enable parallel accesses to ULL-Flash. To guarantee the data persistency, it also employs our proposed persistency control to manage power failure. (8) An advanced HAMS system with aggressive integration is referred to as `hams-T`. `hams-TP` employs such HAMS system, which works on persist mode. Lastly, (9) `hams-TE` employs `hams-T`, but the extend mode. Lastly, we configure an `oracle` platform that employs a 512GB NVDIMM to serve the evaluated workloads.



(a) MMF and Rodinia benchmarks.

(b) SQLite benchmark.

Fig. 16: Application performance.

B. System Performance Analysis

Application-level performance. Figures 16 plots the performance for microbenchmark, Rodinia, and SQLite benchmark. `mmap` achieves 43K pages/s for the microbenchmark and graph workloads, and 6905 SQL ops/s for the SQLite workloads, on average, which are, respectively, 2.54x and 1.37x worse than `hams-TE`. This is because, the I/O requests in `mmap` go through a complex system software stack before finally reaching the storage, which introduces a substantial overhead. Although `flatflash-P` allows CPU to directly access data from the storage, it consumes 4.8us for 64B data access, which is over 40 times longer than DRAM access latency [1]. Thus, `flatflash-P` degrades performance by 75% in the workloads of MMF microbenchmark, compared to `mmap`. Since `flatflash-M` can hide the long storage access latency by buffering hot pages in host-side memory, `flatflash-M` outperforms `flatflash-P` by 136%, on average, in all evaluated workloads. However, `flatflash-M` accesses the storage via MMIO rather than the NVMe protocol, which loses the opportunity of utilizing the plenty of queue and flash parallelisms. In contrast, `hams-LE` implements NVMe protocol in our HAMS controller to enable parallel accesses to the underlying ULL-Flash. In addition, `hams-LE` mitigates the storage access overhead from OS by offloading the task of page access to hardware. Therefore, `hams-LE` improves the performance by 26% in all the workloads, on average, compared to `flatflash-M`. `nvdimm-C` further improves the efficiency of the storage access by directly connecting ULL-Flash to the host-side DRAM via the same memory channel. However, to prevent the memory controller and SSD controller from competing for the memory channel, `nvdimm-C` constrains the data migration between DRAM

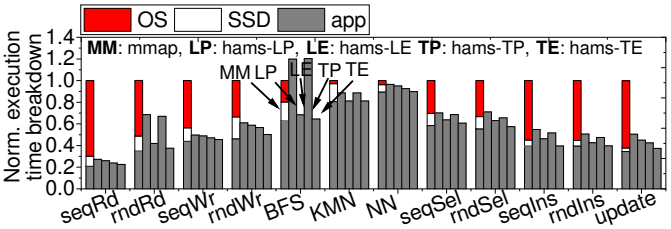


Fig. 17: System-level execution time breakdown.

and ULL-Flash in the period of DRAM refresh operations. Although fetching a single page from ULL-Flash costs 3us, moving data from ULL-Flash to DRAM consumes upto 48us [42]. Considering such long latency, it is difficult to execute latency-critical applications in `nvdimm-C`. In contrast, HAMS can fit to a wider range of applications, especially the ones with large memory footprint. `optane-P` outperforms `mmap` by 121% in microbenchmark, as all data initially reside in the persistent memory, which eliminates the overhead of moving data between memory and storage. However, the performance of `optane` is unfortunately not promising in the workloads with fine-granular memory accesses (i.e., Rodinia and SQLite benchmark). This is because the memory request size is much smaller than the internal block size (256B) of Optane DC PMM, which wastes the memory bandwidth. `optane-P` resolves the mismatch between the memory request size and the Optane internal block size by employing DRAM as the cache of Optane DC PMM. Such design improves the performance by 142%, compared to `optane-P`. On the other hand, `hams-T` serves the memory requests from NVDIMM, whose bandwidth is not constrained by the internal block size. In addition, `hams-T` enables direct access between NVDIMM and ULL-Flash, which eliminates the redundant data copies. Thus, `hams-TE` improves the application's performance by as high as 12%, compared to `optane-M`. Lastly, as the data migration latency cannot be fully overlapped with computation time in data-intensive workloads (e.g., *seqRd* and *seqWr*), `hams-TE` performs worse than `Oracle` by 14%.

Execution time breakdown. We now break down the execution time of our workloads from the view point of system software, to analyze the critical factors that can impact the overall performance. Figure 17 shows the execution time breakdown. As shown in the figure, in `mmap`, a large fraction of the execution time is consumed by the "OS" and "SSD" accesses. The overheads brought by the "OS" and "SSD" accesses cannot be hidden by application execution, as the application is always stalled until the OS fetches data from storage and prepares it in the main memory. Since we are using an ultra-low latency SSD (ULL-Flash), the overheads brought by the storage accesses are not the main factor that degrades the overall performance. Instead, as current Linux kernel is *not* optimized for ULL-Flash, it becomes a performance bottleneck in the baseline platform. On the other hand, the overheads brought by "OS" and "SSD" can be ignored in `HAMS`, as `HAMS` hybrids NVDIMM and ULL-Flash in the main memory, and directly accesses ULL-Flash as memory without any OS intervention. Note that the storage-access times are excluded from "app" and separately presented with the labels of "OS"

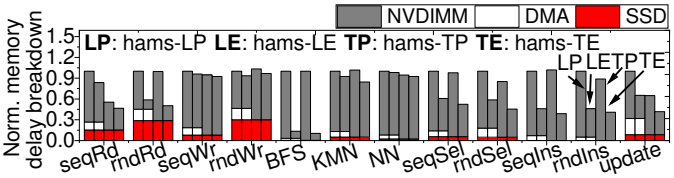


Fig. 18: Memory access delay breakdown.

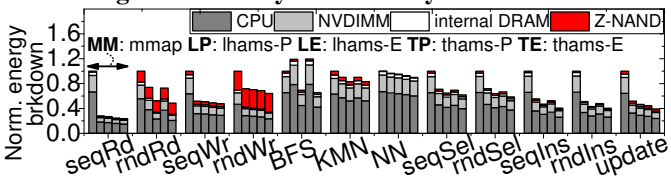


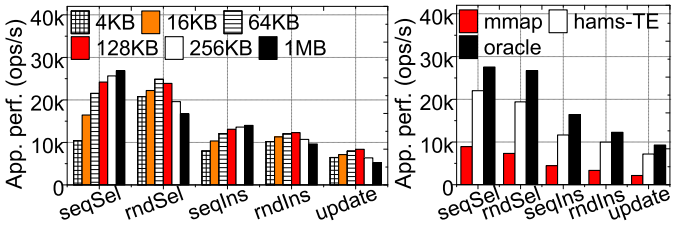
Fig. 19: Energy breakdown normalized to `mmap`.

and "SSD", whereas the storage-access times of `HAMS` are included in "app" (as they are classified as the latencies of LD/ST instructions). The "app" time of `hams-TE` is as short as that of `mmap`, indicating that `hams-TE` can *fully hide* the OS and SSD access overhead.

C. Detailed Analysis

Memory latency analysis. We collect the statistics from the memory-side and present the hardware performance in terms of memory stalls in Figure 18. As our HAMS employs a large NVDIMM as cache (i.e., 8GB in Table II) to accommodate most memory requests, the cache hit rate of NVDIMM reaches 94%, on average, in all the tested workloads. Thus, NVDIMM accesses account for 79% of the total memory delay in `hams-LP`. `hams-T` (including `hams-TP` and `hams-TE`) reduces the total memory stalls by 16%, compared to `hams-L`. This is because `hams-T` leverages the DDR4 interface to directly transfer data between NVDIMM and SSD while `hams-L` uses different interfaces (DDR and NVMe/PCIe) for NVDIMM and ULL-Flash, and always requires extra time to transform data format. On the other hand, the persist mode generates 34% more memory delay than the extend mode, on average. This is because, the persist mode only allows one memory access at a time, which means serializing the executions of instructions that experience cache misses. For `hams-L`, NVMe-DMA contributes to 18% of the memory delay in data-intensive workloads such as *rndRd*, *rndWr*, *seqRd*, *seqWr* and *update*. This is because, the PCIe link used by NVMe SSD is mainly designed for peripheral devices and provides much lower bandwidth compared to the DDR4 interface. Thus, transferring data via PCIe costs much longer time than the DDR4 interface. On the other hand, for other workloads that do not intensively access the storage, `hams-L` and `hams-T` have similar memory delays.

Energy analysis. Figure 19 plots the energy consumption of the *whole system* including CPU, system memory (DRAM), ULL-Flash internal DRAM, and Z-NAND chips. As shown in the figure, `hams-LP`, `hams-LE`, `hams-TP`, and `hams-TE` reduce the system-level energy by 31%, 41%, 34%, and 45%, respectively, compared to `mmap`. Specifically, the combined energy of CPU and system memory in `mmap` is 89% higher than that of `hams`, as `mmap` spends more time for I/O responses, which also costs more CPU and memory idle



(a) Various page sizes. (b) Larger footprints.

Fig. 20: Performance impact of different page sizes and large memory footprints.

energy. On the other hand, the persist mode and extend mode do not impact the energy consumption of NVDIMM. This is because, the persist mode only constrains the number of memory requests on the fly, but it does not impact the total number of NVDIMM accesses. In contrast, `hams-L` consumes 8% more NVDIMM energy than `hams-L`. This is because, `hams-T` directly transfers data between NAND flash and NV-DIMM without any redundant data copies, whereas `hams-L` employs NVDIMM and SSD internal DRAM to buffer data and this introduces redundant data copies. `HAMS` also reduces the energy of accessing SSD by 11%, compared to `mmap`, on average. This is because, `mmap` needs to periodically flush data from the main memory to SSD for persistency.

Overhead analysis. Compared to the existing memory controller, `HAMS` requires NVMe queue engine, SSD command generator and lock register. In our design, the core logic of the NVMe queue engine and SSD command generator employ thousands of gates, which are negligible compared to the billion transistors in modern CPUs. While HAMS enables ULL-Flash to share the DDR4 channels with the NVDIMMs to avoid extra usage of channel resources, ULL-Flash can occupy one DIMM slot. However, considering the fact that LRDIMM supports up to 24 DIMM slots, HAMS only reduces the maximal memory capacity by 4%.

D. Sensitive Testing

Various page sizes. We evaluate the performance of SQLite benchmark with various page sizes, and the results are shown in Figure 20a. While 4KB and 1MB are the default page sizes in Linux Kernel 4.9, we also select intermediate page sizes such as 16KB, 64KB, 128KB and 256KB. As a small page size incurs frequent TLB misses and cannot utilize ULL-Flash internal parallelism, `4KB` achieves poor performance in workloads *seqSel* and *seqIns*. On the other hand, employing a large page size increases data migration overhead when cache misses in NVDIMM. Therefore, `1MB` achieves poor performance in workloads of random accesses (e.g., *rndSel* and *rndIns*). In our evaluation, configuring the page size as `128KB` can achieve the best performance in most workloads.

Large memory footprints. We perform a stress testing on NVDIMM by increasing the data set size to 44 GB, and the results are shown in Figure 20b. `hams-TE` degrades the performance by 24% compared to `Oracle`, owing to the frequent data migration between NVDIMM and ULL-Flash. Nevertheless, `hams-TE` still outperforms `mmap` by 181%.

VII. Related Work and Discussion

Recently, Intel has released a byte-addressable PRAM-based NVDIMM (i.e., Optane DC PMM) as a replacement for the main memory [27]. However, unlike DRAM, user applications *cannot* directly access persistent data from the proposed hardware using load/store instructions without customized software stack. Specifically, OS needs a series of Intel-custom software support, including a block driver, persistent-memory-aware filesystem and *Direct Access* (i.e., DAX) [46] to directly map the Optane DC PMM (as memory) to a userspace. The existing applications also require modifications to be compatible with Intel runtime libraries [28] built on DAX. Further, Optane DC PMM also has several drawbacks from a hardware design angle. Specifically, it exhibits much lower storage capacity than ULL-Flash based HAMS (i.e., 512 GB/DIMM vs. 2.3 TB/DIMM) [19]. With the same number of memory packages per standard unit size, the aggregated throughput of Optane DC PMM is 4.5× lower than that of ULL-Flash [29], [70]. It also faces the challenges of addressing the long PRAM write latency issues. [66] reports that Optane DC PMM integrates a 16KB XPBuffer to accommodate the write requests. However, as the XPBuffer size is fixed and relatively small, [29] observes that NVDIMM-N outperforms Optane DC PMM (as persistent memory) by 5.72x in write-intensive workloads.

Several prior studies [17], [30], [42], [51], [62] propose to integrate DRAM and flash into a single system memory. Similar to Optane DC PMM, memory requests need to go through multiple software layers, including NVML libraries and a specific HybriDIMM driver [52], before accessing data from HybriDIMM. In addition, when configuring HybriDIMM as persistent memory, its internal DRAM buffer is *disabled*, which directly exposes the long flash latency to system [50].

Abulila *et al.* proposes FlatFlash [1], which utilizes NAND flash to expand the memory space. Specifically, FlatFlash directly exposes ULL-Flash to the host as a byte-addressable device by leveraging the SSD internal DRAM as cache. However, a large portion of the SSD internal DRAM is used to store the address translation table [69]. The remaining DRAM space is much smaller than the host-side DRAM, which can be insufficient to accommodate the whole working set. In addition, as FlatFlash employs MMIO rather NVMe protocol to access the underlying ULL-Flash, it cannot benefit from the SSD internal parallelism thereby exhibiting lower device-level throughput. While migrating hot pages to the host-side memory can improve the overall performance, FlatFlash cannot guarantee the data persistency in such case.

In contrast, our HAMS expands the capacity of main memory without modifying the traditional filesystem or user applications. Specifically, just like DRAM, it directly exposes the address space of ULL-Flash to MMU, while leveraging our HAMS controller to manage data movements between NVDIMM and ULL-Flash, making it transparent to OS. To the best of our knowledge, such architectural design has not been discussed in the literature before. While HAMS can also be implemented as a kernel module, it requires OS to respond

to every cache miss in NVDIMM (i.e., page fault), which incurs the overhead of context switch and page fault handling. Such software overhead is undesirable when large working sets incur frequent page swapping between NVDIMM and ULL-Flash (cf. Figure 7a). Furthermore, HAMS outperforms other DRAM+NVM approaches by maximizing the throughput of both NVDIMM and ULL-Flash (cf. Section VI-B). Our persistency control design can also guarantee data persistency without sacrificing ULL-Flash's performance.

A set of prior work propose disaggregated memory solutions to expand the memory capacity [36], [39], [44], [53]. For example, [53] explores the feasibility of constructing a large memory pool across 1,000 servers via Ethernet. However, this design suffers from a low network bandwidth and high total cost of ownership (TCO). [44] partially addresses the aforementioned challenges. [44] improves the network throughput by employing PCIe interface and reduces the cost of hardware infrastructures by deploying more DRAM DIMMs in customized blade servers. Unfortunately, it is still challenging to adopt this design, owing to the high cost (i.e., price and power consumption) of DRAM DIMMs. [36] further reduces TCO by replacing DRAM with NAND flash. However, accessing flash from remote servers increases the I/O latency by 10$\sim$15 us, which is 5$\times$ longer than ZNAND access latency (i.e., 3us). On the other hand, [36] requires source-level modifications to the running applications, which exposes huge overheads to the users. In contrast to the above solutions, HAMS is a scale-up solution, which aggregates the capacities of local NVDIMM and ULL-Flash as a single memory space. HAMS, therefore, saves the huge cost of constructing many blade servers and purchasing expensive DRAM DIMMs. In addition, as HAMS builds TB-scale persistent memory in an OS-transparent manner, executing applications in HAMS requires no changes to the existing programming models.

VIII. Conclusion

We proposed HAMS to aggregate the storage capacities of NVDIMM and ULL-Flash into a single large memory space, which can be used either as a working memory expansion or as a persistent memory expansion. We also optimized HAMS by modifying its datapath and hardware modules, which guarantees data persistency and makes HAMS more energy efficient and reliable. Our HAMS and advanced HAMS architectures improve MIPS by 97% and 119%, respectively, compared to the software-based hybrid NVDIMM design, while saving 41% and 45% energy, respectively.

IX. Acknowledgement

This research is mainly supported by NRF 2021R1AC4001773 and IITP 2021-0-00524. The work is also supported in part by KAIST start-up package (G01190015), NRF 2016R1C182015312, and MemRay grant (G01190170). Dr. Kandemir is supported in part by NSF grants 1908793, 1629129, 2028929, and 1931531. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Myoungsoo Jung is the corresponding author.

References

[1] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 971–985.

[2] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1753–1758.

[3] J. Axboe, "Flexible io tester," *https://github.com/axboe/fio*, 2017.

[4] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2b-ssd: the case for dual, byte-and block-addressable solid-state drives," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 425–438.

[5] I. Bhati, Z. Chishti, and B. Jacob, "Coordinated refresh: Energy efficient techniques for dram refresh scheduling," in *ISLPED*, 2013.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH*, 2011.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[8] R. Chen, Z. Shao, and T. Li, "Bridging the i/o performance gap for big data workloads: A new nvdimm-based approach," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 9.

[9] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu *et al.*, "A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*. IEEE, 2018, pp. 338–340.

[10] J. Ellefson, "Nvm express: Unlock your solid state drives potential," *Flash Memory Summit*, 2013.

[11] H. P. Enterprise, "Hpe 8gb nvdimm single rank x4 ddr4-2133 module," *https://www.hpe.com/us/en/product-catalog/servers/server-memory/pip.hpe-8gb-nvdin* 2018.

[12] S. Gao, B. He, and J. Xu, "Real-time in-memory checkpointing for future hybrid memory systems," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 263–272.

[13] Google, "Leveldb sqlite benchmark," *https://github.com/google/leveldb/tree/master/doc/bench*, 2014.

[14] ——, "Leveldb," *http://leveldb.org/*, 2017.

[15] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," *arXiv preprint arXiv:1811.01544*, 2018.

[16] V. Guddekoppa, "Method and system providing file system for an electronic device comprising a composite memory device," 2016, uS Patent App. 15/390,021.

[17] S. D. Hammond, A. F. Rodrigues, and G. R. Voskuilen, "Multi-level memory policies: what you add is more important than what you take out," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 88–93.

[18] HPE, "Using nvdimm persistent memory server technology with linux," *https://h20195.www2.hpe.com/v2/getpdf.aspx/a00036172enw.pdf*, 2017.

[19] J. Hruska, "Optane dc persistent memory offers up to 512gb per dimm," *https://www.extremetech.com/computing/288854-optane-dc-persistent-memory-offers-u* 2019.

[20] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped ssds with flashmap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015, pp. 580–591.

[21] A. Huffman, "Nvm express, revision 1.0 c," *Intel Corporation*, 2012.

[22] A. Huffman and S. P. Engineer, "Nvm express overview & ecosystem update," *Proceedings of Flash Memory Summit*, 2013.

[23] Intel, "Intel core i7-4790k processor," *https://ark.intel.com/products/80807*, 2014.

[24] ——, "Intel ssd 535 series," *https://ark.intel.com/products/series/86825/Intel-SSD-535-Series*, 2015.

[25] ——, "Intel ssd 750 series," *https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series.html*, 2015.

[26] ——, "Intel optane memory," *https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html*, 2017.

[27] ——, "Intel optane dc persistent memory," *https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html*, 2018.

[28] ——, "Persistent memory programming," *https://pmem.io/*, 2019.

[29] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," *https://arxiv.org/abs/1903.05714*, 2019.

[30] J. Jayaraj, A. F. Rodrigues, S. D. Hammond, and G. R. Voskuilen, "The potential and perils of multi-level memory," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 191–196.

[31] JEDEC, "Ddr4 nvdimm-n design standard," 2016.

[32] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "Simplessd: modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 37–41, 2018.

[33] M. Jurik, "mmap-benchmark," *https://github.com/exabytes18/mmap-benchmark*, 2014.

[34] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 613–626.

[35] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[36] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash local flash," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 345–359, 2017.

[37] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," in *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[38] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 399–411, 2016.

[39] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 417–427.

[40] D. Le Moal, "I/o latency optimization with polling," in *Vault Linux Storage and Filesystems Conference*, 2017.

[41] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA*, 2009.

[42] C. Lee, W. Shin, D. J. Kim, Y. Yu, S.-J. Kim, T. Ko, D. Seo, J. Park, K. Lee, S. Choi *et al.*, "Nvdimm-c: A byte-addressable non-volatile memory module for compatibility with standard ddr memory interfaces," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 502–514.

[43] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.

[44] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 267–278, 2009.

[45] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 329–343, 2017.

[46] C. Mellor, "Do optane's prospects look dimm? chip chap has questions for intel," *https://www.theregister.co.uk/2018/07/26/david_kanter_optane_dimms/*, 2018.

[47] MICRON, "Tn-40-07: Calculating memory power for ddr4 sdram," *https://www.micron.com/resource-details/868646c5-7ee2-4f6c-aaf4-7599bd5952d*, 2017.

[48] I. Micron Technology, "3d xpoint technology," *https://www.micron.com/products/advanced-solutions/3d-xpoint-technology*, 2015.

[49] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 401–410, 2012.

[50] NETLIST, "Hybridimm," *http://s2.q4cdn.com/000096926/files/doc_downloads/hybridimm/Netlist-FMS2017-HybriDIMM-modes-of-operation-v2.81-080717.pdf*, product brief, 2017.

[51] ——, "Hybridimm: Storage at memory speeds, memory at storage capacities," *http://www.netlist.com/products/Storage-Class-Memory/HybriDIMM/default.aspx*, 2017.

[52] ——, "Netlist demonstrates the first storage class memory running real-world applications with hybridimm," *http://s2.q4cdn.com/000096926/files/doc_downloads/hybridimm/Netlist-FMS2017-Demonstration-Brief-v3.1-FINAL.PDF*, 2017.

[53] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[54] A. Sainio, "Nvdimm: Changes are here so what's next," *In-Memory Computing Summit*, 2016.

[55] Samsung, "Advancements in ssds and 3d nand reshaping storage market," in *Flash Memory Summit*, 2017.

[56] ——, "Ultra-low latency with samsung z-nand ssd," *https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf*, 2017.

[57] SNIA, "Persistent memory in windows," *https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Tom_Talpey_Persistent_Memory_in_Windows_Server_2016.pdf*, 2017.

[58] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[59] D. S. STANDARD, "Jesd79-4," *JEDEC*, 2012.

[60] C. E. Stevens, "Information technology-at attachment 8-ata/atapi command set (ata8-acs)," *ANSI, revision*, vol. 18, 2005.

[61] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[62] G. Voskuilen, A. F. Rodrigues, and S. D. Hammond, "Analyzing allocation behavior for multi-level memory," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 204–207.

[63] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[64] Wikipedia, "Ddr4 sdram," *https://en.wikipedia.org/wiki/DDR4_SDRAM*, 2014.

[65] D. Woodhouse, "Jffs: The journalling flash file system," in *Ottawa linux symposium*, 2001.

[66] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 169–182.

[67] ZDNet, "Windows leaps into the nvm revolution," *https://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/*, 2017.

[68] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim *et al.*, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 477–492.

[69] J. Zhang, M. Kwon, H. Kim, H. Kim, and M. Jung, "Flashgpu: Placing new flash next to gpu cores," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[70] J. Zhang and J. Myoungsoo, "Zng: Architecting gpu multi-processors with new flash for scalable data analysis," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[71] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," in *ICCAD*, 2012.