HAMS: Hardware Automated Memory-over-Storage for Large-scale Memory Expansion

Jie Zhang[1], Miryeong Kwon[2], Donghyun Gouk[2], Sungjoon Koh[2], Nam Sung Kim[3]
Mahmut Taylan Kandemir[4], Myoungsoo Jung[2]
Computer Architecture and Memory Systems Laboratory,
Peking University[1], Korea Advanced Institute of Science and Technology (KAIST)[2]
University of Illinois Urbana-Champaign[3], Pennsylvania State University[4]
http://camelab.org

I. Introduction

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, the existing persistent memories either suffer from the poor performance or are constrained by poor scaling. One may leverage the existing OS memory management to construct a large persistent memory space by hybriding NVDIMM and SSD. Unfortunately, overheads incurred by a heavy software-stack intervention seriously negate the benefits of such designs.

Tackling the aforementioned limitations, 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 (*cf.* Figure 1), 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.

II. Related Work and Challenges

Baseline architecture for persistent memory expansion. One common solution to build a large and scalable, yet persistent memory space is to use a type of NVDIMM (*i.e.*, NVDIMM-N) together with SSD and memory-mapped files (MMFs), which can be implemented in an OS memory manager or a file system (*cf.* Figure 1). Figure 2 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 (❼❽❾). Once the actual data is loaded into a new region of the allocated page memory,

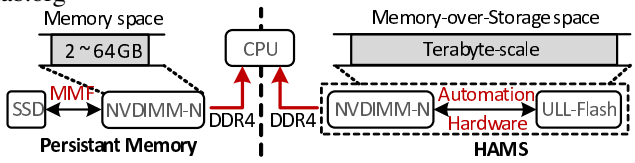


Fig. 1: NVDIMM-N vs. HAMS.

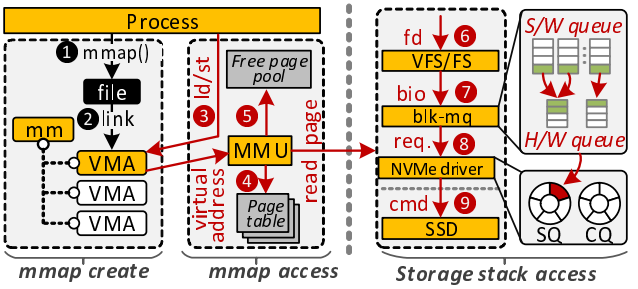


Fig. 2: Software support.

the page fault handler creates a page table entry (PTE), records the new page address in the PTE, and resumes the process.

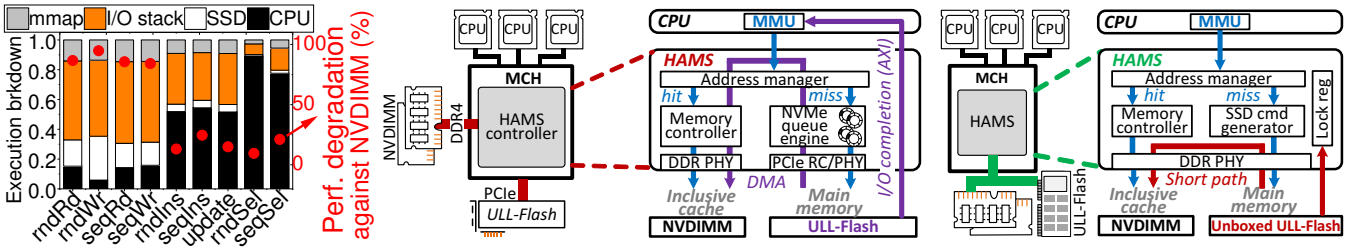
Challenges. To evaluate the performance of an existing software-based memory expansion, we configure an MMF-based host system with the real devices. The SSD (*i.e.*, ULL-Flash) is used to expand memory space over `mmap`. Figure 3a decomposes the execution time of user applications into a mmap processing 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. 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 [3]. The context switches are one of the main contributors to increase I/O latency [5]. On the other hand, the queuing mechanism and NVMe communication protocol in I/O stack are optimized for throughput rather than I/O latency [6]. The software operations of MMF consume 15∼20 us [3], which is around 6× longer than ZNAND access latency (3 us).

III. Memory Over Storage

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.

A. Overview of HAMS Designs

Figure 3b 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



(a) Software overheads. (b) Overview of baseline HAMS. (c) Aggressive integration.

Fig. 3: Software overheads of using MMF, the overview of baseline HAMS and the aggressive integration.

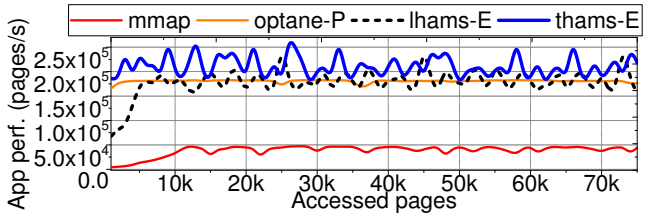


Fig. 4: Performance analysis.

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. In case that a memory request generates an NVDIMM cache miss, HAMS migrates the target data from ULL-Flash to NVDIMM cache by managing the NVMe commands and I/O request queues internally. This design can hide all the NVMe protocol and interface management overheads from the OS. 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. Aggressive Integration of HAMS

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 the following inefficiencies: (1) the overheads imposed by data transfer and (2) the energy inefficiency brought by the SSD-internal DRAM. 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 3c, 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.

(3) `lhams-E` employs the designs of baseline HAMS. (4) `thams-E` is an advanced HAMS system with aggressive integration.

IV. Evaluation and Conclusion

Experiment. We use a full system simulator (gem5 [1]) and an SSD simulator (Amber [2]) to explore the design space of the HAMS enabled systems. We then build four computing platforms for evaluation: (1) `mmap` employs an ULL-Flash and a NVDIMM as its storage and memory media, respectively. It accesses data directly from the persistent storage by using the MMF module. (2) `optane-P` [4] employs 512GB Optane DC PMM as main memory.

Performance. Figure 4 plots the application-level performance variation when executing workload "seqRd". In this graph, the x-axis shows the number of accessed pages, and the y-axis shows the performance in terms of pages per second. `mmap` exhibits poor performance at the beginning of the execution, and it takes a long time to reach its best performance. This is because, all pages initially reside in SSD and `mmap` fetches the pages on demand. Thus, it takes a lot of time to fully fetch all the pages in the working set from the NVMe SSD into the main memory. In contrast, as all data reside in the persistent memory, `optane` does not suffer from cold misses. `lhams-E` experiences poor performance at the beginning of the execution. However, unlike `mmap`, it takes a much shorter time for `lhams-E` to reach the full bandwidth. This result indicates that the storage access has less penalty to `lhams-E`, as it eliminates the OS intervention in accessing the storage. `thams-E` achieves promising performance throughout the execution, as it can successfully hide the penalty of cold misses in the NVDIMM caches. Note that the best performance of `thams-E` is better than `lhams-E` and `mmap` by 16% and 4.8x, respectively.

In conclusion, we proposed HAMS to aggregate the storage capacities of NVDIMM and ULL-Flash into a single large memory space, which can be used as a memory expansion. We also optimized HAMS by modifying its datapath and hardware modules, which makes HAMS more energy efficient and reliable. Our HAMS and advanced HAMS architectures improve the performance by 97% and 119%, respectively, compared to the software-based hybrid NVDIMM design.

V. Original Publication

J. Zhang J, M. Kwon, D. Gouk, S. Koh, NS Kim, MT Kandemir and M. Jung. 2021. Revamping storage class memory with hardware automated memory-over-storage solution. IEEE ISCA. *https://arxiv.org/pdf/2106.14241.pdf*

VI. Acknowledgement

This research is mainly supported by NRF 2021R1AC4001773 and IITP 2021-0-00524 & 2022-0-00117. 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] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH*, 2011.
[2] D. Gouk *et al.*, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," *IEEE MICRO*, 2018.
[3] J. Huang *et al.*, "Unified address translation for memory-mapped ssds with flashmap," in *IEEE ISCA*, 2015.
[4] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *https://arxiv.org/abs/1903.05714*, 2019.
[5] D. Le Moal, "I/o latency optimization with polling," in *Vault Linux Storage and Filesystems Conference*, 2017.
[6] J. Zhang *et al.*, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *OSDI*, 2018.