HME: A Lightweight Emulator for Hybrid Memory

Zhuohui Duan, Haikun Liu*, Xiaofei Liao, Hai Jin
Services Computing Technology and System Lab/Cluster and Grid Computing Lab/Big Data Technology and System Lab
School of Computing Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {zhduan, hkliu, xfliao, hjin}@hust.edu.cn

***Abstract*—Emerging *non-volatile memory* (NVM) technologies have been widely studied in recent years. Those studies mainly rely on cycle-accurate architecture simulators because the commercial NVM hardware is still unavailable. However, current simulation approaches are either too slow, or cannot simulate complex and large-scale workloads. In this paper, we propose a DRAM-based hybrid memory emulator, called *HME*, to emulate the performance characteristics of NVM devices. HME exploits hardware features available in commodity *Non-Uniform Memory Access* (NUMA) architectures to emulate two kinds of memories: fast, local DRAM, and slower, remote NVM on other NUMA nodes. HME can emulate a wide range of NVM latencies by injecting software-created memory access delays on the remote NUMA nodes. To evaluate the impact of hybrid memories on the application performance, we also provide application programming interfaces to allocate memory from NVM or DRAM regions. We evaluate the accuracy of the read/write delay injection models by using SPEC CPU2006 and compare the results with a state-of-the-art NVM emulator *Quartz*. Experimental results demonstrate that the average emulation errors of NVM read and write latencies are less than 5% in HME, which is much lower than *Quartz*. Moreover, the application performance overhead in *HME* is one order of magnitude lower than *Quartz*.**

I. Introduction

The advent of NVM technologies has increasingly attracted research interest in hybrid memory systems. Emerging NVM technologies, such as Intel/Micron's 3D XPoint memory [1] offers higher memory density, much lower cost per bit and standby power consumption than DRAM. However, because NVM has limited write endurance, and higher write latency and power consumption, a more practical way of using NVM is to organize it in conjunction with commodity DRAM.

There are many open issues about hardware/software design for hybrid memory systems, such as, memory hierarchy [2]–[5], hybrid memory management [6]–[8], and memory allocation schemes [9], [10] from the perspective of computer architectures, operation systems, and programming models, respectively. Moreover, the most interesting fact programmers are expected to know is that how the NVM latencies and bandwidth characteristics affect application performance. This knowledge is especially useful for directing the memory system designs and data placement policies.

Because NVM hardware is still not commercially available, most system researchers mainly rely on cycle-accurate architecture simulators [11]–[13] to investigate how a new algorithm or system design could improve the application performance under different settings of NVM latencies and bandwidth [6], [7], [9], [10]. However, the simulation approaches have many limitations. First, full-system simulators such as Gem5 [11] are extremely slow for simulating large workloads since the application performance slowdown may be more than five orders of magnitude. Second, some trace-driven simulation approaches such as Zim [14] run faster than the full-system simulators, but only support very limited applications. Many widely studied parallel/distributed applications (e.g., Hadoop and Spark) are not able to execute in the simulation platforms. Moreover, it would take a lot of effort to adapt simulators for supporting NVMs in some mushrooming software platforms such as, virtualization, machine learning, and graph computing. Third, many memory simulators [12], [13] can not model the effects of cache evictions, out-of-order instruction execution, and memory-level parallelism.

Despite the cycle-accurate architecture simulators, there have been some emulation approaches for NVMs [7], [15]. Intel *Persistent Memory Emulation Platform* (PMEP) [7] emulates NVM read latency by injecting additional CPU stall cycles in each sampling interval, and emulates NVM bandwidth by throttling the maximum number of DRAM access requests through a programmable feature in memory controller. Unfortunately, Intel PMEP needs special CPU microcode and customized platform firmware/hardware, and this in-house NVM emulator have limited accessibility. Quartz [15] is another persistent memory emulator proposed by HP lab. Quartz also models application perceived NVM access latency by injecting software created delays periodically. One limitation of Quartz is that it does not support NVM write latency emulation. This may result in high emulation errors when estimating the performance of memory-intensive applications with a high ratio of memory writes to memory reads. Generally, those emulation approaches are much faster (several orders of magnitude) than cycle-accurate architecture simulators.

In this paper, we present a lightweight hybrid memory emulator, called HME. HME exploits existing hardware features in commodity Intel CPUs to emulate performance characteristics of NVM using DRAM. Generally, application perceived performance difference between NVM and DRAM is mainly determined by two performance characteristics (i.e., memory latency and bandwidth). We do not mimic the details of each memory access operation accurately, instead we focus on emulating the NVM latency and bandwidth to make it close to a given setting in a period of time.

Similar to PMEP [7] and Quartz [15], we exploit the DRAM thermal control interface provided by commodity Intel CPUs

* Haikun Liu is the corresponding author.

978-3-9819263-0-9/DATE18/©2018 EDAA

to limit the maximum memory bandwidth. It is more difficult to emulate NVM latency because current commodity hardware does not provide any knob to control the memory latency. Fortunately, we can program hardware performance counters to monitor the number of *Last Level Cache* (LLC) misses in a small time window (called epoch). The CPU stall time on a LLC miss can reflect the memory access latency. Thus, we inject additional software created delay (the difference between NVM and DRAM latencies) periodically to emulate the NVM latency. In this way, we slow down DRAM to mimic the performance characteristics of NVM.

We implement HME on NUMA based Intel Xeon processors. For a given processor, we use DRAM on a remote NUMA node to emulate the higher-latency and lower-bandwidth NVM. To facilitate programming on hybrid memory systems, we also redesign the memory allocator to explicitly support memory allocation from DRAM or NVM regions. We modify Linux kernel to identify memory zones of NVM, and extend the Glibc library to support *nvm_malloc* interface.

We evaluate the emulation accuracy of HME through a set of applications in SPEC CPU2006 benchmark. The experimental results show that the average emulation error of NVM read in HME is lower than 1%. HME also demonstrates higher emulation accuracy and lower application performance overhead than the state-of-the-art NVM emulator *Quartz*.

The remainder of this paper is organized as follows. Section II introduces NVM access latency and bandwidth emulation models in HME. Section III describes the detailed implementation of HME. Section IV presents an application programming interface for memory allocation in hybrid memories. Experimental results are presented in Section V. We discuss the related work in Section VI and conclude in Section VII.

II. NVM Emulation Models

In this section, we present the models for emulating NVM using DRAM in a NUMA architecture.

A. Latency Emulation

As there is no programming interface in commodity hardware to directly control memory access latency, we adopt a software approach to emulate the NVM latency. The basic idea is to inject software-generated additional delays for the applications in a fixed time interval.

Let RD_i denote the total read delay that should be injected in a given time interval i, NVM_r and $DRAM_r$ denote the average NVM and DRAM read latencies, respectively, and M_i denote the total number of memory accesses in the interval i. Equation 1 describes the basic model for computing the addition delay RD_i, where M_i is the only key parameter we need to count in the time interval i.

$$RD_i = M_i * (NVM_r - DRAM_r) \tag{1}$$

Generally, a processor needs to access memory only when the required data misses in the last level cache. The CPU stall time due to a LLC miss is proportional to the memory access latency. The current commodity CPUs provide hardware performance counters to monitor the LLC misses. At the end of each given interval, HME reads the hardware performance counters and calculates the additional delay, and then halts the CPUs of the applications to inject the software-created delay.

In practice, a LLC miss often accompanies a page table walk, which consumes even more time to access the page table entries in the main memory. As a result, we refine the NVM latency emulation model in Equation 2:

$$RD_i = \delta * M_i * (NVM_r - DRAM_r) \tag{2}$$

where δ is a coefficient mainly correlated with the LLC miss rate of the application. Its value can be estimated by executing a large amount of experiments on a wide range of workloads, and using advance learning techniques such as linear regression to model it.

As memory write operations are not on the critical path of applications' execution, they are not perceived by the applications due to the write-back cache and speculative instruction execution on modern CPUs. Generally, data write operations can be categorized into *write through* and *write back*. For the former approach, data is written to the main memory through the cache immediately. For the latter approach, data is first written to the cache and then written to the main memory when the corresponding cache lines are evicted. However, the memory write operations still consume memory bus bandwidth and can postpone the memory read requests. Those delays should not be ignored at all.

To emulate the NVM write delay, we monitor the numbers of NVM write-through and write-back by employing the *Performance Monitor Unit* (PMU) and *Model-Specific Registers* (MSR) in Intel Xeno CPUs, and count the CPU stall time caused by those memory writes. Let WD_i denote the total write delay that should be injected in a given time interval i, M_i and N_i denote the total number of memory write-through and write-back operations on the NVM in the interval i, Δ_{wtl} and Δ_{wbl} denote the injected write-through and write-back delays (the difference between NVM and DRAM write latencies), respectively. The injected delay of NVM write can be calculated as follows:

$$WD_i = M_i * \Delta_{wtl} + N_i * \Delta_{wbl} \tag{3}$$

B. Bandwidth Emulation Model

Like the previous approaches [7], [15], we emulate NVM bandwidth by limiting the maximum available DRAM bandwidth. The bandwidth throttling is achieved by leveraging a DRAM thermal control interface provided in commodity Intel Xeon processors [16]. Equation 4 describes the bandwidth control model:

$$B_N = \varepsilon * B_D \tag{4}$$

where B_N and B_D denote the bandwidth of NVM and DRAM, respectively, and ε denotes a user-defined bandwidth throttling ratio. We use Intel Xeon Processor Uncore Performance Monitor [16] to limit DRAM bandwidth for each channel separately.

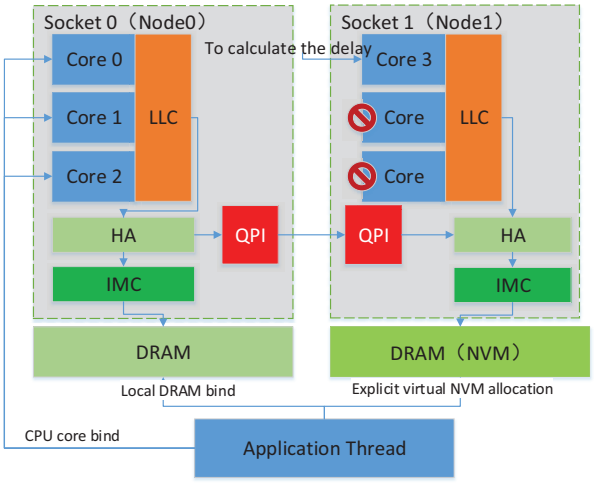

Fig. 1: HME uses the NUMA architecture to emulate the hybrid memory comprising of NVM and DRAM on a two-socket server

More specifically, we throttle the number of memory requests from the memory controller of a channel in a given interval.

III. HME Implementation

In this section, we present the implementation details of HME [17].

We employ the NUMA architecture to emulate a hybrid memory system on a two-socket server. The NUMA architecture enables commonly use of the local DRAM while offering higher DRAM access latency on the remote socket. We use DRAM on the remote NUMA node to emulate the higher-latency and lower-bandwidth NVM. As shown in Figure 1, we select a core at the remote socket (Node 1) to calculate the additional delay that should be injected to the applications running on the local socket (Node 0). The other cores on the remote socket are not used to avoid performance interference on the computation core. HME uses *Home Agent* (HA) in uncore PMU (*Performance Monitoring Units*) of Intel Xeon processors [16] to record the number of memory read accesses (LLC misses) and write accesses to the DRAM on Node 1, and these memory requests should be issued from applications running on Node 0. HME injects additional delays to the cores of the Node 0 through *inter-processor interrupts* (IPIs). In this way, HME increases the remote memory access latencies to emulate the NVM latency.

The process of read delay injection is shown in Figure 2. The program runs on Core 0 of Node 0, while the Core 3 of Node 1 emulates the NVM. When the program accesses the remote DRAM (i.e., emulated NVM), the Core 3 calls a poll function. It periodically reads the PMU registers on Node 1 and gets the number of remote memory accesses by Core 0 in the polling period. After that, Core 3 calculates the emulated latency and injects the delay to Core 0 through IPIs. Core 0 handles the IPI requests, which result in CPU spinning to create software delays for emulating the NVM read latency.

We rely on a specific counter in the *Model Specific Registers* (MSR) [18] and PMU tool [16] to emulate NVM write delay.

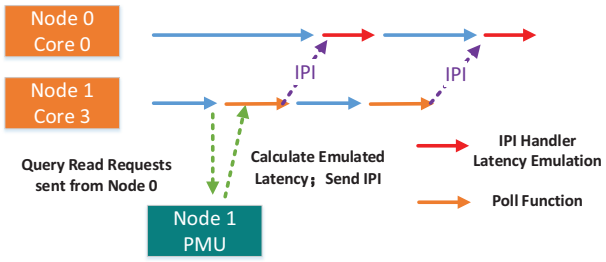

Fig. 2: The process of read delay injection in HME

We can get the numbers of write-back and write-through operations during a polling period, and then we use the delay injection model as mentioned in Section II to emulate NVM write delay. Because each core's MSR cannot be accessed by other cores, the applications' cores are periodically interrupted to read their MSR counters, and then send this information to the remote socket for calculating the write delay. These interrupts may slow down application's execution. Moreover, because the hardware counter only records the total number of LLC evictions on a socket, we could not know exactly which core the write-back LLC eviction comes from. To this end, we apportion the write delay on each core evenly. We think this is also a limitation of this approach. This problem can be addressed if processor vendors provide more plentiful performance counters in the future.

We use *Integrated Memory Controller* (IMC) uncore PMU and the model described in Section II-B to emulate NVM bandwidth. We limit the number of memory access instructions that the memory controller can process in a time slot to control the NVM bandwidth. We also provide a set of bandwidth control APIs for users.

IV. NVM Programming Interface

We develop new memory allocation APIs to facilitate the programming on HME, named AHME. It includes two kinds of hybrid memory allocation policies: (1) using *numactl* [19] to achieve coarse-grained hybrid memory allocation; (2) using *malloc* and *nvm_malloc* interfaces for fine-grained memory allocation in DRAM and NVM, respectively.

Because HME is a NUMA-based NVM emulator, we can directly allocate memory using NUMA APIs [19]. There are three data placement strategies:(1) "*numactl-membind*=0" (all data on the DRAM); (2) "*numactl-membind*=1" (all data on NVM); (3) "*numactl-interleave*" (data interleavingly placed in DRAM and NVM). Obviously, the coarse-grained *numactl* does not meet the needs of on-demand memory allocation from DRAM or NVM regions.

We develop a new API for more flexible and fine-grained hybrid memory allocation. As shown in Figure 3, we extend the *Glibc* library to provide a *nvm_malloc* function, so that the application can allocate hybrid memories through *malloc* or *nvm_malloc*. In order to support the *nvm_malloc* function, we modify Linux kernel to provide a branch for *nvm_mmap* memory allocation. The branch handles the procedure *alloc_page* in the *nvm_malloc* function. Meanwhile, the NVM pages are

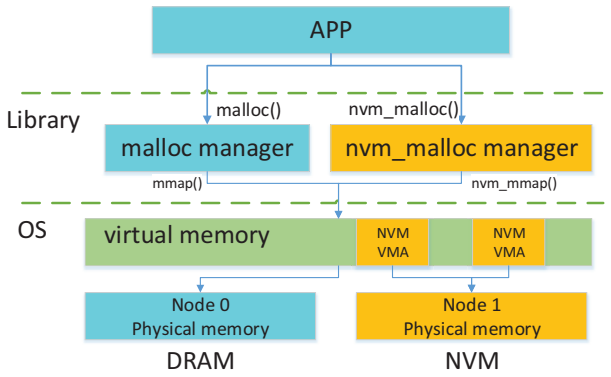

Fig. 3: Using AHME to achieve fine-grained memory allocation

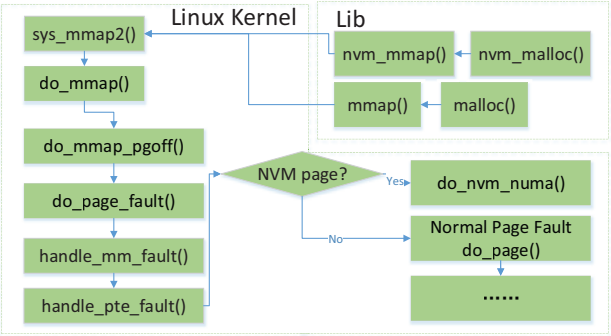

Fig. 4: Flow chart of AHME memory allocation in Linux kernel

differentiated from the DRAM pages in the VMA (virtual address space). AHME marks the NVM flags in VMA, and calls a specified *do_nvm_page_fault* function on a page fault to allocate the physical address space on the remote NUMA nodes.

The NVM memory allocation branch is shown in Figure 4. When *nvm_mmap()* from the extended *Glibc* is called, the kernel calls the *do_mmap()* function, and sets a *NVM_VMA* flag on the VMA structure when the *do_mmap_pgoff()* function applies for the VMA.

When a NVM page is accessed for the first time, it triggers a page fault and the kernel calls *handle_mm_fault()* function to handle the page fault. If the *NVM_VMA* flag is matched, the *do_nvm_numa()* function is called to allocate a physical NVM page, which is actually a DRAM page in a remote NUMA node. If the page fault does not refer to a NVM page, the kernel uses normal *do_page()* to allocate a DRAM page in the local node.

We implement the *nvm_malloc* function in Glibc library and add a new syscall *nvm_mmap*. The *nvm_malloc()* calls the *nvm_mmap()* to pass the *mmap* parameters to the modified Linux kernel. After that, the above NVM allocation is performed by the kernel.

V. Evaluation

In this section, we evaluate the emulation accuracy of our models using a variety of applications, and compare the emulation errors with Quartz.

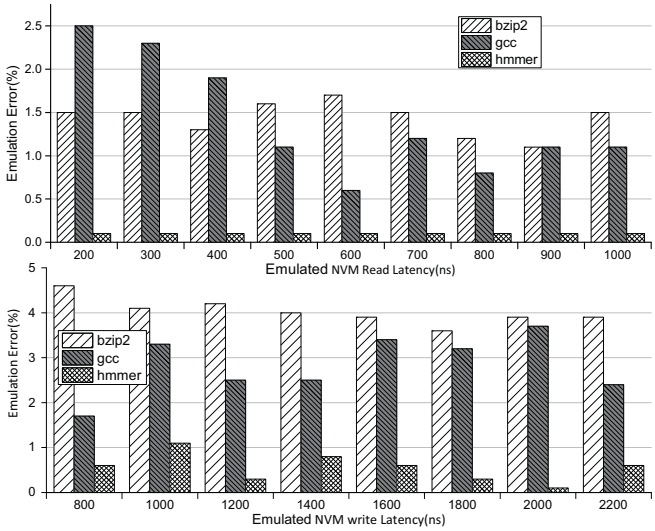

Fig. 5: Emulation errors of NVM read and write latencies

A. Experimental Setup

Testbed. Our emulator is implemented and evaluated on NUMA-based servers, which are equipped with two-socket octa-core Intel Xeon E5-2650 2.0 GHz processors and 64 GB DRAM. According to [15], the local and remote average memory access latencies are approximate 120 ns and 175 ns, respectively. The servers all run Linux kernel 3.11.0 with PMFS supported.

Methodology. We use applications from SPEC CPU2006 [20] to evaluate the accuracy of NVM latency emulation model. First, we run each application in a DRAM-only system, and use the application execution time T as a baseline. Second, we run each application in our emulator with a specific NVM delay setting, and get the application execution time $T^{'}$. We also count the total number of memory accesses to the remote DRAM, and calculate the total delay D we should inject for emulating NVM accesses. Finally, we compare $(T^{'} - T)$ with D to quantify the NVM latency emulation errors.

B. NVM Emulation Accuracy

We disable the NVM write and read delay injection to evaluate the NVM read and write latency emulation, respectively. As shown in Figure 5, the emulation errors do not exceed 3% for a wide range of NVM read latency settings. We find that the emulation errors for hmmer are extremely low and stable with the increase of read latency. The reason is that hmmer is a CPU-bound application for searching patterns in DNA sequences, and there is only a very small fraction of time consumed in memory accesses. As NVM write latency is several times larger than NVM read latency, we change the write latency setting from 800 ns to 2200 ns. The emulation errors of NVM write are a little larger than that of NVM read, because memory write operations do not necessarily cause CPU stalling while memory read latency is always perceived by applications. Overall, the write delay emulation errors in HME are less than 5%.

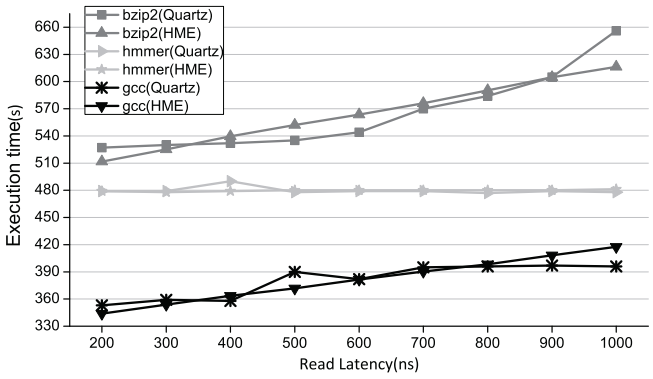

Fig. 6: Application execution time varies with NVM read latencies

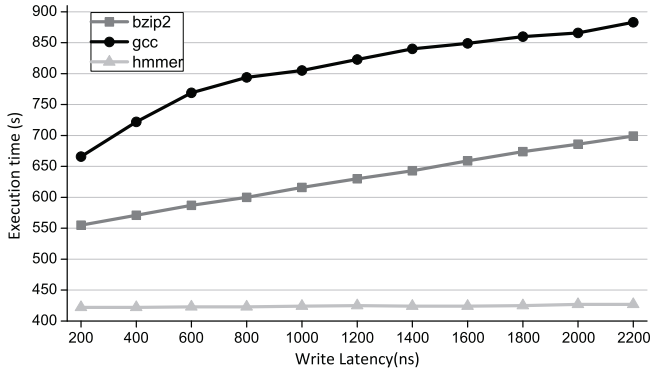

Fig. 7: Application execution time varies with NVM write latencies

Figure 6 shows how application performance is affected by the increase of NVM read latency. For bzip2 and gcc, the application execution time is exactly linear with the increase of NVM read latency in HME. The results have validated the accuracy of our read delay injection model described in Section II. Quartz also shows the trend of increasing application execution time, but with some fluctuations. This implies that HME can achieve a higher accuracy of NVM latency emulation than Quartz. For hmmer, we find that its performance is not sensitive to the memory access latency because the memory accesses only consume a very small portion of its total execution time. As Quartz does not support NVM write delay emulation, Figure 7 only shows the write emulation results of HME. We find that the application execution time is roughly linear with the growth of NVM write latency. This result also demonstrates the accuracy of our model for NVM write latency emulation.

Figure 8 shows the emulation errors of NVM read latency in HME and Quartz. For all applications in SPEC CPU2006, the average emulation errors in HME and Quartz are 0.7% and 2.1%, respectively. HME significantly reduces the emulation errors by 2X compared to Quartz. We believe the reason for higher emulation errors in Quartz is due to its delay injection framework. When Quartz emulates NVM read latency, it disables all computation cores on the remote socket and uses a local core to sample the PMU counters. Also, this emulation framework may have impact on application performance due to inter-application interference on CPU

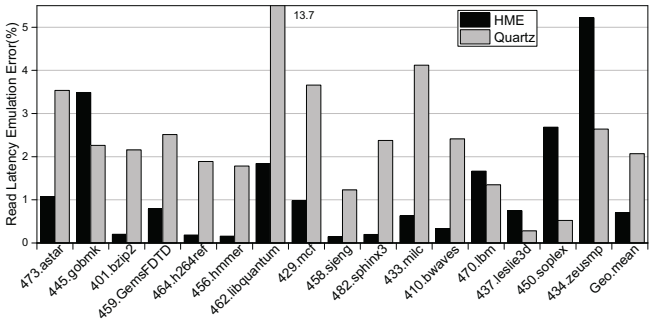

Fig. 8: Emulation accuracy of NVM read latency in HME and Quartz

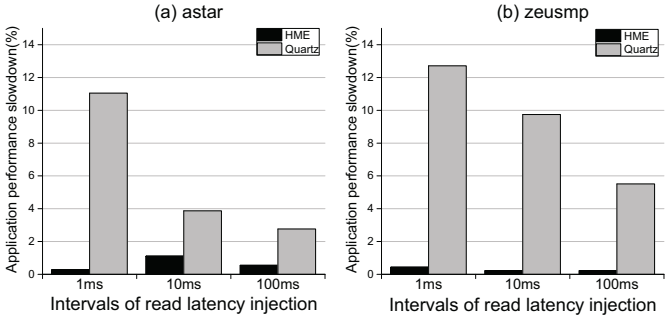

Fig. 9: Application performance degradation varies with the different intervals of delay injection in HME and Quartz

resource. Figure 9 shows that the application performance slowdown is higher when the periodical sampling interval becomes shorter in Quartz. In contrast, HME performs the delay injection operations by a processor on the remote node, so that the delay injection in HME has very small effect on the application performance. As shown in Figure 9, the application performance degradation in HME is not sensitive to the intervals of delay injection.

C. Performance of Programming APIs

Figure 10 shows the performance of NVM allocation APIs in HME and Quartz. All results are normalized to the baseline Glibc. As the *nvm_malloc()* in AHME is an extension of *malloc()* in Glibc, the normalized NVM allocation latency is almost equivalent to Glibc. For Quartz, the NVM allocation is based on *libnuma*. There are some additional operations such as *mbind()* before the actual memory allocation *numa_alloc_onnode()*. As a result, this function is approximate 20% slower than the *malloc()*. For all allocators, allocating a larger memory region usually costs more time, although it does not linearly increases with the requested memory sizes.

VI. Related Work

There have been a number of studies on NVM emulation with DRAM. Pelley *et al*. [21] propose offline analysis with PIN to calculate the average number of cache misses referenced by a program. They use these statistics to introduce additional delays when the application actually runs on bare metal servers. LEFF [22] supports both NVM emulation and simulation in the same framework. The trace-driven model

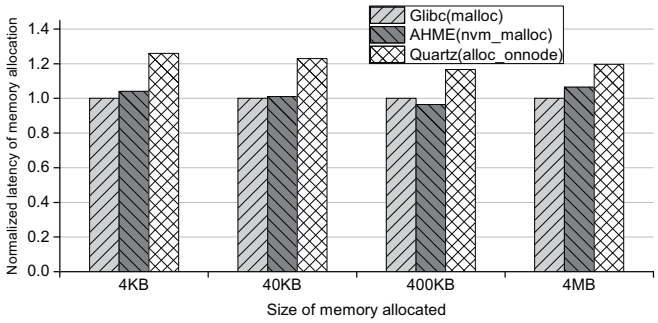

Fig. 10: Performance of memory allocation for different approaches

usually takes much time on profiling and analysis. Instead, HME emulates NVM read/write latency in an online manner.

Volos *et al*. [10], [23] achieve NVM write latency emulation by injecting software generated delays. This progress needs programmers to explicitly flush cache lines out of the last level cache. HME extends this model by further categorizing memory writes into write-back and write-through operations and injecting different delays correspondingly.

PMEP is an in-house NVM emulator, which emulates NVM read latency and bandwidth by using Intel's special CPU microcode and custom platform firmware [7]. Lee *et al.* [24] design FPGA-based hybrid memory prototypes with DRAM and PCM. In contrast, HME is an open source hybrid memory emulator, which is implemented by using the available hardware features of commodity Intel processors.

Quartz [15] perhaps is the most similar work to HME. It also emulates NVM read latencies using a delay injection model on a NUMA-based architecture. However, the following key features of HME differentiate it from Quartz. First, Quartz does not support write delay emulation while HME provides a reasonable approach to NVM write emulation with a good accuracy. Second, Quartz uses the local cores to sample the PMU counters and performs the delay injection. This results in application performance degradation and higher emulation errors. In contrast, HME uses a dedicated processor on the remote socket to handle the event counting of MSR and PMU when emulating the NVM. As a result, our model does not interfere applications' execution and archives a high accuracy of NVM emulation.

VII. Conclusion

We presented a hybrid memory emulator, called HME. It can effectively emulate a wide range of NVM read/write latencies and bandwidth constraints on commodity hardware, with trivial effect on application performance. Moreover, we also develop an user-friendly API to facilitate the programming on our emulator. The API is an extension of Glibc and has comparable memory allocation performance with Glibc. Experimental results show that the average emulation errors of NVM read/write latencies are lower than 5% in HME. It also shows higher emulation accuracy and lower application performance overhead than the state-of-the-art Quartz.

Acknowledgment

This work is supported by National Key Research and Development Program of China under grant No. 2017YFB1001603, and National Natural Science Foundation of China under grant No.61672251, 61732010, 61628204.

References

[1] Intel Optane Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.
[2] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *DAC*, 2009.
[3] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *ISCA*, 2009.
[4] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.
[5] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *ICS*, 2017.
[6] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *SOSP*, 2009.
[7] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *EuroSys*, 2014.
[8] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *ASPLOS*, 2017.
[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS*, 2011.
[10] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS*, 2011.
[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
[12] Matthew Poremba, Tao Zhang, and Yuan Xie. NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.
[13] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, January 2011.
[14] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. *ACM SIGARCH Computer Architecture News*, 41(3):475–486, 2013.
[15] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Middleware*, 2015.
[16] Intel Xeon Processor E5-Product Family. https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors.html.
[17] HME. *https://github.com/CGCL-codes/HME*.
[18] Intel Architecture Instruction Set Extension Programming Reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.
[19] NUMACTL. *http://linux.die.net/man/8/numactl/*.
[20] SPEC CPU 2006. *https://www.spec.org/cpu2006*.
[21] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.
[22] Guoliang Zhu, Kai Lu, Xiaoping Wang, Xu Zhou, and Zhan Shi. Building Emulation Framework for Non-Volatile Memory. *IEEE Access*, 5:21574–21584, 2017.
[23] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *EuroSys*, 2014.
[24] Taemin Lee, Dongki Kim, Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. FPGA-based Prototyping Systems for Emerging Memory Technologies. In *RSP*, 2014.