Towards Write-back Aware Software Emulator for Non-Volatile Memory

Atsushi Koshiba*† 1), Takahiro Hirofuchi†, Soramichi Akiyama†, Ryousei Takano†, Mitaro Namiki*

*Tokyo University of Agriculture and Technology

†National Institute of Advanced Industrial Science and Technology (AIST)

Email: koshiba@namikilab.tuat.ac.jp, {t.hirofuchi, s.akiyama, takano-ryousei}@aist.go.jp, namiki@cc.tuat.ac.jp

***Abstract*—Non-volatile memory (NVM) such as phase change memory is a promising technology for future low-energy and high-capacity memory systems. One of the well-known issues of NVM is its fundamental characteristics that are different from common memory subsystems with DRAM. In particular, the NVM write latency is much higher than DRAM while the NVM read latency is almost the same as DRAM. The latency asymmetry affects the performance of applications significantly. For analyzing behavior of applications running on NVM environments, most researchers use emulation tools due to the limited number of commercial NVM products. However, these existing tools are too slow to emulate a large-scale workload or too simplistic to emulate the application behavior on NVM with asymmetric read/write latencies. This paper therefore proposes a new NVM emulation model that is not only light-weight but also aware of the NVM read/write latency gap. We implemented our prototype on a commercial Intel processor. We also evaluated its accuracy and performed case studies for practical benchmarks. The results show that our prototype can emulate the execution time of practical workloads according to their write behavior, while an existing light-weight emulation model over-estimates the execution time.**

I. Introduction

Recent trends of high-speed and many-core processors lead to an increasing demand for larger memory capacity. Modern computer systems use DRAM for main memory while scaling up DRAM capacity is becoming difficult due to its refresh energy. Because a DRAM cell holds its data as electric charge on a capacitor, periodically refreshing the cell is necessary to prevent a data miss. This energy overhead rapidly increases as DRAM scales up its capacity. It is predicted that the refreshing energy occupies 50% of the overall power consumption in 64 GB DRAM modules [1]. It is also reported that a server computer with 128 GB DRAM consumes more than 40% of its energy consumption for its main memory [2]. This energy-greedy characteristic of DRAM is an obstacle for future large capacity memory systems.

Non-Volatile Memory (NVM) is one of the solutions to overcome this energy constraint. NVM read/write latencies are in the order of tens of nanoseconds, therefore it has a potential to be used as main memory. In addition, NVM does not require refreshing to keep its data unlike DRAM. This non-volatility prevents memory subsystems from wasting a large amount of energy. Recent NVM technologies have attracted a lot of attention not only in academia but also in industry; new NVM products such as 3D-Xpoint are being developed [3]. For these reasons, NVM products are expected to achieve high-capacity and energy-efficient memory systems.

Although NVM is effective for energy reduction, current applications and system software for DRAM machines may not be appropriate for future NVM systems due to the performance characteristics of NVM such as latency and bandwidth. In particular, the latency gap between reading and writing NVM is not negligible. For example, phase change memory (PCM) [4] represents High and Low by changing its cell phase either of two phases: amorphous phase (Low) and crystalline phase (High). Read operations to PCM just measure the cell resistance, while write operations apply an electrical pulse to the cell to heat it and change its phase. Particularly, PCM recrystallization (changing from amorphous phase to crystalline phase) requires long duration of pulsing. Writing PCM therefore requires much longer latency than reading. The ITRS roadmap [5] reports that the write latency of PCM is about 10X higher than the read latency in 2013. It also forecasts that writing PCM will be still 5X slower than reading it in 2026. This gap possibly leads to performance degradation of write-intensive application programs. For example, the results of our preliminary experiments (shown in Section IV-C in this paper) exhibit that libquantum, a write-intensive workload, becomes nearly 3X slower on an NVM environment than on a DRAM environment.

To make use of future main memory with NVM, several researchers have tackled to find out new system software support and memory subsystems which are appropriate for NVM characteristics [6], [7], [8]. However, the lack of available NVM devices makes it difficult to analyze the performance of applications running on NVM machines. Memory emulation tools are therefore essential for researchers to analyze/evaluate the performance of their proposals without actual NVM devices. Although there are several existing emulation tools for NVM devices, their functions are limited in terms of taking the higher write latency into account. Cycle-accurate simulators [9], [10] are widely used among researchers. While they can set read and write latencies independently in nanoseconds, these simulators are not appropriate for large-scale workloads because they are time-consuming. In contrast to heavy-weight simulators, Volos et al. propose Quartz, which is a software emulator for NVM devices [11]. Quartz runs on common DRAM machines and estimates additional memory stall cycles on NVM machines using CPU performance counters. It then delays the execution of processes by the stall cycles. Such an emulation model based on software calculations is light-weight, however, it is unaware of the NVM read/write latency gap. Because common processors use a write-back caching system, cache controllers hide writing to the main memory from running processes and CPU cores. The fact makes it difficult for software emulators to distinguish read and write access to the DRAM memory modules.

To overcome these shortcomings of existing emulation tools, this paper presents a light-weight NVM emulator that takes the performance gap between reading and writing into account. Unlike Quartz approach, our emulator classifies cache misses of the process into two types: *read-only* and *write-back*. The former performs only reading data from NVM, and the latter performs both reading and writing. It assumes that write-back cache misses lead to additional CPU stall

This work was supported by JSPS Grant KAKENHI 16K00115.

1) Work done during his internship in AIST.

cycles than the other on NVM systems. To estimate the number of write-back cache misses, our emulator monitors not only CPU cache misses, but also the behavior of other components (pre-fetchers and cache controllers). The emulator then calculates the additional delays caused by the two types of cache misses (read-only and wrire-back) independently based on the emulated NVM read/write latencies. This write-back aware approach enables an accurate emulation of NVM devices such as PCM.

To clarify the effectiveness of our approach, we developed a prototype of the proposed emulator on the Intel SandyBridge-E architecture. We then evaluated the accuracy of the prototype and performed case studies using SPECCPU 2006 benchmarks. We found that our prototype emulates the NVM write latency with errors of 12.1% to 28.5%. We also found that existing light-weight emulation models such as Quartz can over-estimate the execution time of benchmark programs due to their unawareness of the latency gap. These results indicate that our write-back aware approach is helpful for emulating NVM devices whose read/write latencies are asymmetric.

II. Motivation

In this section, we mention the impacts of the NVM latency gap on common write-back cacheable memory systems.

A. Memory Access Mechanism

We first describe a memory access mechanism on commodity computers. We assume that NVM modules are byte-addressable as with DRAM modules, and running processes can access to the NVM modules with load/store instructions. We also assume that both NVM and DRAM environments are write-back cacheable, which hold modified data in cache lines and do not write the data to the memory modules until the lines are evicted.

On such common memory systems, memory references reaching to the main memory mostly occur when load/store instructions cause last level cache (LLC) misses. When a CPU core executes a load or store instruction, the CPU refers to its local cache. If the data do not exist on the local cache, it then refers to larger caches (L2, L3, ...). If the data do not exist even on the LLC, the core causes an LLC miss and fetches a chunk of memory including the target data from the memory module. When an LLC miss occurs, a cache controller selects an LLC line where new data should be loaded according to a certain cache management scheme (e.g., n-way set associative). At the same time, the old data on the selected LLC line is evicted to make room for new data.

The procedure of an LLC miss differs depending on the state of the evicted LLC line. If the state of the line is *clean* or *invalid*, the cache controller reads the new data from the memory module and stores it on the LLC. On the other hand, if the state of the line is *modified*, the controller not only reads new data from the module, but also writes the modified line to the module in order to reflect the change to the main memory. Therefore, we can find two types of LLC misses; one that just reads data from the memory module, and the other that induces a write-back. We define the former and the latter as a *read-only* LLC miss and a *write-back* LLC miss, respectively. We describe the performance difference between the two on NVM systems.

B. Impacts of Higher Write Latency

As mentioned above, there are two types of LLC misses on common write-back cacheable systems. The two types of LLC misses lead to the same latency on common DRAM systems because reading a new line and writing an old line are executed in parallel [12].

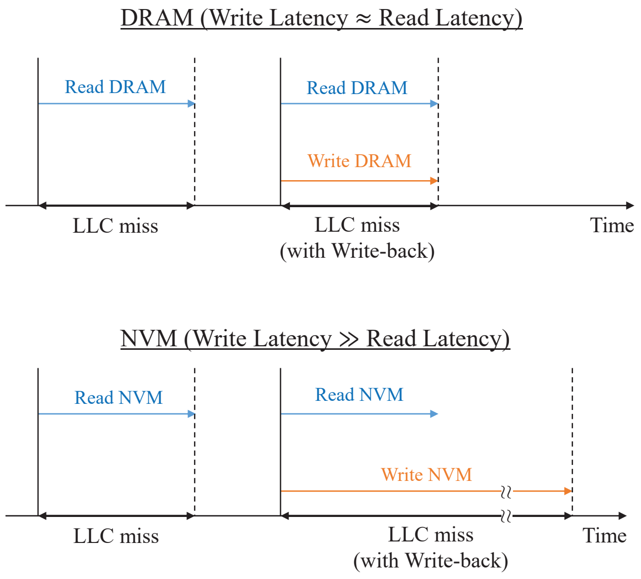

Fig. 1. Penalty time of LLC misses on DRAM and NVM environment.

However, if NVM devices such as PCM are used for main memory, the additional duration will be necessary when a modified cache line is evicted due to its higher write latency. Fig. 1 shows the difference of penalty time per one LLC miss between DRAM and NVM. The upper part of Fig. 1 shows the DRAM case where the write latency is almost the same as the read latency. As write-back operations are completely hidden behind reading on DRAM environments, both the two types of LLC misses result in the same penalty time, irrespective of the occurrence of write-backs. On the other hand, the lower part of Fig. 1 shows the NVM case where the write latency is much longer than the read latency. We assume that write-back LLC misses on NVM environments require longer time in order to wait for the evictions of old lines. Although memory controllers can temporarily hold write requests in a request queue to prevent write requests from interfering read requests, the queue will not work well for write-intensive applications because of its limited size. Thus, if write-back LLC misses occur frequently, the CPU core that causes a write-back is forced to keep stalling until the old data eviction is completed. This problem possibly influences the processing speed of application programs depending on their memory-access behavior. For instance, our experimental results in Sec. IV-C show that the execution time of libquantum, a write-intensive benchmark, becomes nearly 3X slower on NVM than on DRAM.

C. Problem of Related Work

As described above, the asymmetric read/write latencies of NVM have an impact on applications performance. Analyzing the effects on performance is therefore indispensable for developing future NVM systems. Because there are few number of commercial NVM products, researchers are forced to use emulation/simulation tools for their experiments. However, there are some issues in existing tools to emulate the read/write latency gap. The most common tool is cycle-accurate simulators. These simulators are used with other CPU simulators and simulate full system behavior with NVM per CPU cycle [13], [9]. This approach can set NVM read/write latencies independently, while it is too slow to emulate large-scale workloads. For instance, the simulation environment of NVMain [9]

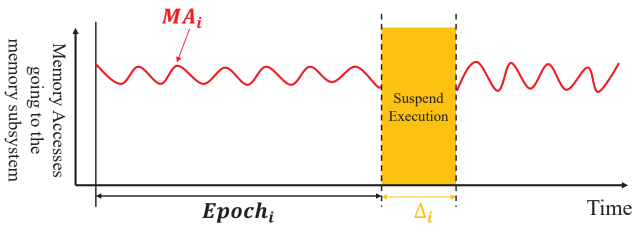

Fig. 2. Quartz approach [11]

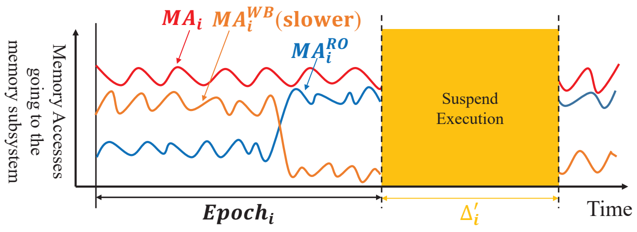

Fig. 3. Our approach

with gem5 [14] takes about eight hours to simulate a program whose execution takes only one second on a real machine.

On the other hand, Quartz [11] is proposed as a light-weight emulation approach making use of Performance Monitoring Counters (PMC) with which each CPU core of Intel processors is equipped. Quartz monitors the number of DRAM accesses of the target process during the process execution and inserts additional delays based on the obtained information and a given NVM access latency. Fig. 2 shows the Quartz emulation model. Quartz measures the number of DRAM accesses caused by the target process using PMCs at a specific interval named $Epoch$. It then calculates the additional delay, Δ, that is expected to occur when the process runs on slower NVM devices. Δ_i, the additional delay in $Epoch_i$, is defined as Eq. (1):

$$\Delta_i = MA_i \times (NVM_{lat} - DRAM_{lat}) \tag{1}$$

where MA_i is the number of LLC misses causing stalls on the target process, and NVM_{lat} and $DRAM_{lat}$ are NVM access latency and DRAM access latency, respectively. Quartz uses POSIX signals to suspend the process execution for Δ_i and to resume it. This approach causes only a slight emulation overhead which is negligible.

Although the Quartz approach has an advantage on the processing overhead over cycle-accurate simulators, it does not take the read/write latency gap into account. One of the difficulties to adapt it to the latency gap is that current PMCs do not support monitoring the number of write-backs per CPU core. As mentioned above, when a CPU core causes an LLC miss, whether a write-back occurs is determined by the state of the cache line selected in accordance with the cache coherent scheme. Because cache controllers are responsible for managing states of LLC lines and synchronizing LLC coherency among the cores, a CPU core itself and its counters do not know when a line is evicted to the main memory. In addition, LLC misses are caused by not only CPU cores, but also hardware pre-fetchers. Modern processors have multi-cores and each core has a pre-fetcher. Since hardware pre-fetching is performed asynchronously, pre-fetchers cause write-backs without being noticed by CPU cores. These facts make it difficult for the emulation approach using PMCs to measure the actual number of write-back LLC misses caused by a certain process. To overcome this issue, our emulation model estimates per-core write-back LLC misses using the performance monitoring feature of cache controllers in addition to PMCs.

III. Write-back Aware NVM Emulator

This section proposes a light-weight emulation model that distinguishes write-back LLC misses and read-only LLC misses.

A. Basic Idea

We assume that write-back LLC misses lead to longer CPU stalls than read-only LLC misses. To take the difference between read and write latencies into account, our emulation model monitors two types of LLC misses independently unlike Quartz approach. Our approach allows users to evaluate applications performance with NVM devices whose access latency is asymmetric.

Our emulator injects delays into the target process depending on the number of LLC misses like Quartz. However, unlike Quartz, our model divides LLC misses into two types; one just reads data from memory modules (read-only) and the other induces both reading and writing (write-back) as shown in Fig. 3. MA_i^{WB} in Fig. 3 is the number of write-back LLC misses, and MA_i^{RO} is the number of read-only LLC misses within $Epoch_i$. These two types of LLC misses satisfy the following condition:

$$MA_i = MA_i^{WB} + MA_i^{RO} \tag{2}$$

We assume that MA_i^{WB} makes CPU cores stall for longer time than MA_i^{RO}. Let NVM_{lat}^{Write} be the average NVM write latency and let NVM_{lat}^{Read} be the average NVM read latency ($NVM_{lat}^{Write} \gg NVM_{lat}^{Read}$), our model represents the additional delay Δ_i as follows:

$$\begin{aligned} \Delta_i' = MA_i^{WB} \times (NVM_{lat}^{Write} - DRAM_{lat}) \\ + MA_i^{RO} \times (NVM_{lat}^{Read} - DRAM_{lat}) \end{aligned} \tag{3}$$

To calculate the value of Δ_i', the emulator needs to monitor MA_i^{WB} and MA_i^{RO} of the target process periodically at run-time. However, PMCs cannot measure the number of write-back LLC misses directly because of their functional limitation. We therefore present a way to estimate the number of write-back LLC misses and achieve a write-back aware NVM emulator.

B. Run-time Estimation of Read-only/Write-back Memory Accesses

This section describes how to calculate the two types of LLC misses (MA_i^{RO} and MA_i^{WB}) independently. Our emulation model enables the calculation by making use of a function of cache controllers in addition to information obtained from PMCs. We assume that processors have cache pre-fetchers (PF) and out-of-order pipelining to accelerate main memory accesses. We also assume that both our emulator and the emulated process are running on the same multi-core processor during the emulation.

In addition to the PMCs' unawareness of write-backs, overlapping LLC misses by memory-level parallelism techniques of common processors is also a problem to calculate MA_i^{WB} and MA_i^{RO}. Quartz approach has already proposed a solution for the LLC miss overlapping and our model follows their model, while the two models differ in terms of the write-back awareness. Because of the PFs and out-of-order pipelining, two or more LLC misses are sometimes performed concurrently and CPU cores avoid long stalls to access to the memory modules. If the emulator does not take overlapping LLC misses into account, it over-estimates additional delays. To exclude the number of LLC misses executed in parallel, our model measures the number of CPU stall cycles caused by LLC misses. Overlapped

TABLE I
PERFORMANCE EVENTS OF SANDY BRIDGE-E FAMILY

Non-Architectural Performance Events [15]	
$L2_{stalls}$	CYCLE_ACTIVITY:STALLS_L2_PENDING
LLC_{hit}	MEM_LOAD_UOPS_RETIRED:LLC_HIT
LLC_{miss}, LLC_{miss,cpu_i}	MEM_LOAD_UOPS_MISC_RETIRED:LLC_MISS
LLC_{miss,PF_i}	OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_MISS.DRAM_N & OFFCORE_RESPONSE.ALL_PF_DATA_RD.LLC_MISS.DRAM_N
Uncore Performance Events for CBo [16]	
WB	LLC_VICTIMS.M_STATE

LLC misses are included in the number of LLC misses counted by PMCs, while they do not increase CPU stall cycles. Our model therefore counts the core stall cycles caused by LLC misses in an epoch and then divides the cycles by the DRAM access latency of the machine. In this way, it estimates the number of LLC misses that actually suspend the CPU execution. Our model defines MA_i^{WB} and MA_i^{RO} as shown in Eq. (4):

$$MA_i^{WB} = \frac{MA\_STALL_i^{WB}}{DRAM_{lat}}, \quad MA_i^{RO} = \frac{MA\_STALL_i^{RO}}{DRAM_{lat}} \tag{4}$$

where $MA\_STALL_i^{WB}$ and $MA\_STALL_i^{RO}$ are the total cycles of core stalls with MA_i^{WB} and with MA_i^{RO}, respectively.

To calculate $MA\_STALL_i^{WB}$ and $MA\_STALL_i^{RO}$, our model extends the equation provided in the documentation of Intel CPUs [12]. The documentation provides the equation to calculate the total time of core stall cycles caused by LLC misses, which makes no distinction between read-only and write-back LLC misses, as follows:

$$MA\_STALL_i = L2_{stalls} \times \frac{W \times LLC_{miss}}{LLC_{hit} + W \times LLC_{miss}} \tag{5}$$

where $L2_{stalls}$ is the total number of core stall cycles caused by L2 cache misses, and LLC_{hit} and LLC_{miss} are the numbers of LLC hits and LLC misses of a core, and W is the ratio of the LLC miss latency (DRAM access latency) to the LLC hit latency. In Eq. (5), LLC_{miss} can be classified into two types (write-back and read-only) as we have already described in Sec. II-A. Our model then defines $MA\_STALL_i^{WB}$ and $MA\_STALL_i^{RO}$ as Eq. (6) and Eq. (7):

$$MA\_STALL_i^{WB} = L2_{stalls} \times \frac{W \times LLC_{miss}^{WB}}{LLC_{hit} + W \times LLC_{miss}} \tag{6}$$

$$MA\_STALL_i^{RO} = L2_{stalls} \times \frac{W \times (LLC_{miss} - LLC_{miss}^{WB})}{LLC_{hit} + W \times LLC_{miss}} \tag{7}$$

where LLC_{miss}^{WB} is the total number of write-back LLC misses.

Due to the lack of performance events of PMCs, LLC_{miss}^{WB} cannot be counted directly. Our model therefore estimates LLC_{miss}^{WB} using other available monitoring functions. To estimate LLC_{miss}^{WB}, there are two key factors: (1) the number of write-backs within a certain period, and (2) the degree of contribution of the process to these write-backs. To measure the factor (1), our model uses an uncore performance counter implemented on the cache controller. Intel processors such as Intel Xeon have LLC coherency engines (CBo) that maintain the coherency among CPU cores [16]. Because CBo counters monitor the number of cache lines written back to the memory modules, they enable our model to measure the factor (1) directly. Next, to estimate the factor (2), our model measures the number of all LLC misses caused by CPU cores and their PFs in the system. We expect that the degree of contribution of a certain CPU core to write-backs can be estimated based on the proportion of its LLC misses to the whole. For example, when the total number of LLC misses in an epoch is 200,000 and the number of write-backs is 50,000, the total number of write-back LLC misses in the epoch is 50,000. In the epoch, when the number of LLC misses caused by a certain core occupies 20% of all the LLC misses, the number of write-backs caused by the core is expected to be also 20% of all the write-backs. Thus, the number of write-back LLC misses of the core is expected to be 10,000. Based on these considerations, our model estimates LLC_{miss}^{WB} with Eq. (8):

$$LLC_{miss}^{WB} = WB \times \frac{LLC_{miss}}{\sum_{i=0}^{n-1} LLC_{miss,cpu_i} + \sum_{i=0}^{n-1} LLC_{miss,PF_i}} \tag{8}$$

where WB is the total number of write-backs, n is the number of CPU cores of a processor, $\sum_{i=0}^{n-1} LLC_{miss,cpu_i}$ is the sum of the numbers of LLC misses caused by every CPU core, $\sum_{i=0}^{n-1} LLC_{miss,PF_i}$ is the sum of the numbers of LLC misses caused by every PF. Eq. (8) calculates the ratio of LLC misses of the target process to LLC misses of the whole system, and then multiplies the ratio and the number of write-backs. Thus, the equation gives us the estimated number of write-back LLC misses caused by a specific process.

C. Applying to Intel SandyBridge-E Architecture

To ensure that our emulation model is applicable to commercial processors, we implemented a prototype of our emulator for Intel SandyBridge-E architecture. Table I shows the performance counter events corresponding with the variables of the above equations [15], [16]. Here, $DRAM_{lat}$ and W are static values relying on the specific performance of a given machine and can be measured using a tool such as Intel Memory Latency Checker [17]. Our prototype is portable because other Intel processor families are equipped with performance counters that support the equivalent events.

IV. EVALUATION

To show the effectiveness of our approach, we evaluated our prototype of the proposed emulator on an Intel Xeon E-2650 machine which has the Intel SandyBridge-E Architecture. Table II shows the detail of our evaluation environment. We used Intel Memory Latency Checker to measure the values of $DRAM_{lat}$ and W.

A. Wbbench: Tool to Measure Write-back Latency

To evaluate the precision of our model emulating the NVM write latency, we developed a tool named *wbbench* that measures the

TABLE II
OUR EVALUATION ENVIRONMENT

Processor	Intel Xeon E5-2650
OS	Debian 8.5 (Linux 3.18.5)
$Epoch$	20 ms
$DRAM_{lat}$	90.7 ns
W	4.5

```
wbbench (){
  memory_region = malloc(line_count * 64);
  generate_random_address_list(memory_region, line_count);
  struct cacheline *clp = get_nextline_from_list();

  start_time = get_time();
  while(clp != NULL){
    clp->value = 0xFFFF;                    // modify line
    clp        = get_nextline_from_list();  // load next line (cause write-back)
  }
  end_time = get_time();

  return wb_latency = (end_time – start_time) / line_count;
}
```

Fig. 4. Pseudo code of wbbench

average write-back LLC miss latency. Fig. 4 shows a pseudo code of wbbench. First, wbbench calls malloc() to reserve a certain amount of memory region. It then calls generate_random_address_list() to generate a list of cache-line aligned addresses within the reserved memory region. The addresses in the list are aligned to the size of an LLC line (64 Bytes) and arranged at a random order. While executing the while() loop, wbbench writes a value to the cache line that is currently referred by a pointer (clp). Next, it calls get_nextline_from_list() to refer to the address of the next cache line in the list and store it in the pointer, which causes an LLC miss with a line eviction. Since the addresses in the list are arranged randomly, wbbench prevents memory parallelism caused by the PFs and the out-of-order mechanism. Wbbench measures the total elapsed time during the while() loop and calculates the average write latency.

Wbbench guarantees get_nextline_from_list() to always induce a write-back LLC miss by reserving the memory region that is sufficiently bigger than the LLC size. However, such memory references over a wide range of address space cause frequent TLB misses which lead to additional latencies. To prevent TLB misses, we enabled 1 GB transparent hugepages during experiments with wbbench. Because the memory region reserved by wbbench falls within one page owing to the hugepage support, wbbench can measure a pure write latency.

B. Validating Accuracy of Emulation with wbbench

We evaluated the accuracy of our emulation model using wbbench. We applied our prototype emulator to wbbench. If our prototype can emulate the NVM write latency accurately, the latency emulated by our prototype and the latency measured by wbbench become the same value or quite close. To ensure that every get_nextline_from_list() call induces a write-back LLC miss, we set the size of the memory region reserved by wbbench to 40 MB, which is twice as large as the LLC size of our environment (20 MB).

Table III shows the evaluation results. The results show that our prototype emulates the selected NVM write latencies with errors of 12.1% to 28.5%. These errors are occurred because our prototype over-estimates the number of write-back LLC misses of the target

TABLE III
WRITE LATENCY EMULATED BY OUR PROTOTYPE, AND WRITE LATENCY MEASURED BY WBBENCH.

Emulated latency	Measured latency	error
100 ns	112.1 ns	12.1 ns (12.1 %)
200 ns	244.6 ns	44.6 ns (22.3 %)
300 ns	376.4 ns	76.4 ns (25.5 %)
400 ns	510.0 ns	110.0 ns (27.5 %)
500 ns	642.5 ns	142.5 ns (28.5 %)

process. To investigate the reason, we compared the number of write-backs obtained from CBo counters and the number of LLC misses obtained from PMCs with the actual number of reading/writing to the memory modules obtained from the memory controller. As a result, we found that our prototype correctly measures the number of write-backs, while it under-estimates the number of LLC misses. We will try a more detailed investigation for our future work.

C. Applying to SPECCPU 2006

To show the effectiveness of our emulation model for estimating performance of future NVM devices, we applied our prototype to practical workloads. We executed benchmarks of SPECCPU 2006 and applied our prototype to them to emulate their behavior with NVM modules. We measured the execution time when our prototype delays the benchmark execution based on its write-back behavior and the assigned NVM write latency. We used one compute-intensive benchmark (hmmer) and two memory-intensive benchmarks (mcf and libquantum) in the experiment. We set the emulated NVM read latency to the same value as the DRAM read latency, while we set the emulated NVM write latency to 1X, 2X, 3X, 4X, and 5X of the DRAM read latency (90.7 ns). To clarify the effect of write-back behavior on processing speed, we also measured memory write throughputs during the benchmark execution. To measure the write throughput, we used an internal performance counter of the memory controller. The counter counts the total bytes written in the memory modules in a certain period of time. The write throughput was calculated by dividing the total written data size by the total execution time of the benchmark.

Fig. 5 shows the emulated execution time of each benchmark program, and Fig. 6 shows their average write throughputs. According to the results, the hmmer keeps its performance as the same as in the DRAM environment because the hmmer causes a small number of write-backs. On the other hand, the mcf, which is a write-intensive workload, leads to the increase of the execution time and the degrease of the write throughput due to the high NVM write latency. The results also show that the performance degradation of the libquantum is more significant, because the libquantum performs writing to the main memory more frequently than the mcf. These results indicate that our model can emulate the behavior of practical workloads running on NVM environments according to their memory access characteristics.

Moreover, to show the benefit of our write-back aware emulation model, we compared our model with write-back unaware emulation models such as Quartz. To evaluate the behavior of write-back unaware models using our prototype, we measured the execution time of libquantum when we set both emulated read/write latencies to the same value (e.g., 5X higher than DRAM). Fig. 7 shows the emulated execution time of libquantum at five types of read/write latency settings. As shown in the results, there is a significant performance difference between the case both reading and writing are slow and

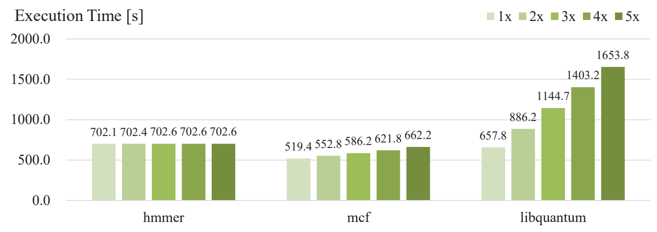

Fig. 5. Execution time of each benchmark when the emulated NVM write latencies are set to 1X, 2X, ..., 5X longer than the DRAM latency and the emulated NVM read latency is always set to the same as the DRAM latency.

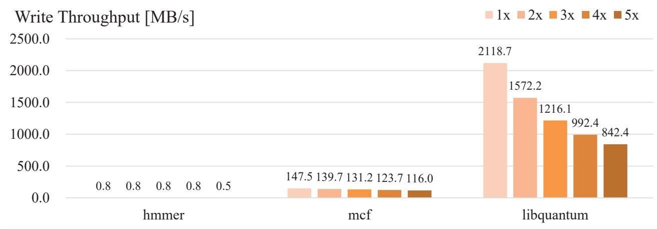

Fig. 6. Emulated write throughput of each benchmark. The experimental condition is the same as Fig. 5.

the case only writing is slow. This performance gap is difficult to analyze by write-back unaware approaches such as Quartz because they cannot set read/write latencies independently. In contrast, our write-back aware emulation model is effective for finding out performance impacts of the asymmetric read/write latencies. To exhibit the advantages of our model more clearly, we will improve our prototype to adapt it to interference between multiple threads and evaluate the accuracy of our model for multi-thread workloads.

V. Conclusion & Future Work

In this paper, we presented a software emulation model for the asymmetric read/write latencies of NVM using DRAM. Our model calculates the execution time of a process with NVM devices based on the number of LLC misses and the number of LLC lines written back to the memory modules. We implemented a prototype of our emulation model and evaluated it on a commercial Intel processor. The results of our preliminary evaluation show that the accuracy of our prototype is acceptable to grasp a rough trend of applications performance with NVM.

Our future work includes mitigating the errors of our prototype, evaluating the effectiveness for emulating actual NVM devices (e.g., PCM, STT-MRAM), and validating the accuracy for multi-threads workloads. Since the energy consumed by reading and writing NVM is different, we assume that our write-back aware emulation model is also effective for evaluating energy performance of NVM devices. we will clarify the effectiveness of our model for the energy asymmetry of NVM for our future work.

References

[1] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[2] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, Dec 2003.

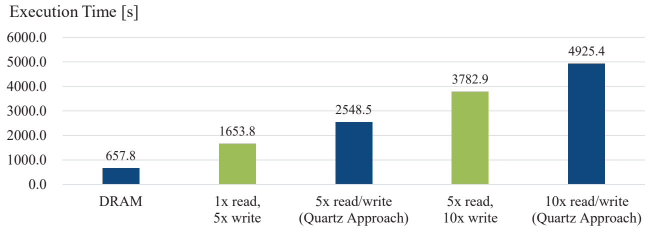

Fig. 7. Execution time of libquantum at each latency setting.

[3] Intel. (2017) Intel®optane^{TM}ssd dc p4800x series. Intel. [Online]. Available: http://www.intel.com/content/www/us/en/solid-state-drives/optane-solid-state-drives-dc-p4800x-series.html

[4] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[5] ITRS. (2013) International technology roadmap for semiconductors 2013 edition. The International Technology Roadmap for Semiconductors (ITRS). [Online]. Available: http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2013/2013PIDS.pdf

[6] M. Giardino, K. Doshi, and B. Ferri, "Soft2lm: Application guided heterogeneous memory management," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Aug 2016, pp. 1–10.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 15:1–15:15.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146.

[9] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug 2012, pp. 392–397.

[10] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Hmmsim: a simulator for hardware-software co-design of hybrid main memory," in *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, Aug 2015, pp. 1–6.

[11] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: ACM, 2015, pp. 37–49.

[12] Intel. Intel 64 and ia-32 architectures optimization reference manual. Intel Corporation. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[13] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[14] 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," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[15] Intel. Intel 64 and ia-32 architectures software developer's manual. Intel Corporation. [Online]. Available: http://www.intel.in/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[16] Intel. Intel xeon processor e5-2600 product family: Uncore guide. Intel Corporation. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf

[17] V. Viswanathan. Intel memory latency checker. Intel Corporation. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker