IEICE TRANSACTIONS

on Information and Systems

VOL. E104-D NO. 5
MAY 2021

The usage of this PDF file must comply with the IEICE Provisions on Copyright.
The author(s) can distribute this PDF file for research and educational (nonprofit) purposes only.
Distribution by anyone other than the author(s) is prohibited.

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY

The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Non-Volatile Main Memory Emulator for Embedded Systems Employing Three NVMM Behaviour Models

Yu OMORI[†a)], ***Nonmember and*** **Keiji KIMURA**[†b)], ***Member***

SUMMARY Emerging byte-addressable non-volatile memory devices attract much attention. A non-volatile main memory (NVMM) built on them enables larger memory size and lower power consumption than a traditional DRAM main memory. To fully utilize an NVMM, both software and hardware must be cooperatively optimized. Simultaneously, even focusing on a memory module, its micro architecture is still being developed though real non-volatile memory modules, such as Intel Optane DC persistent memory (DCPMM), have been on the market. Looking at existing NVMM evaluation environments, software simulators can evaluate various micro architectures with their long simulation time. Emulators can evaluate the whole system fast with less flexibility in their configuration than simulators. Thus, an NVMM emulator that can realize flexible and fast system evaluation still has an important role to explore the optimal system. In this paper, we introduce an NVMM emulator for embedded systems and explore a direction of optimization techniques for NVMMs by using it. It is implemented on an SoC-FPGA board employing three NVMM behaviour models: coarse-grain, fine-grain and DCPMM-based. The coarse and fine models enable NVMM performance evaluations based on extensions of traditional DRAM behaviour. The DCPMM-based model emulates the behaviour of a real DCPMM. Whole evaluation environment is also provided including Linux kernel modifications and several runtime functions. We first validate the developed emulator with an existing NVMM emulator, a cycle-accurate NVMM simulator and a real DCPMM. Then, the program behavior differences among three models are evaluated with SPEC CPU programs. As a result, the fine-grain model reveals the program execution time is affected by the frequency of NVMM memory requests rather than the cache hit ratio. Comparing with the fine-grain model and the coarse-grain model under the condition of the former's longer total write latency than the latter's, the former shows lower execution time for four of fourteen programs than the latter because of the bank-level parallelism and the row-buffer access locality exploited by the former model.

key words: *NVMM, emulator, embedded system, behaviour model, Intel Optane DC persistent memory*

1. Introduction

Non-volatile main memory (NVMM) built with emerging byte-addressable non-volatile memory devices is expected to introduce a new trend in computer systems [1], [2]. NVMM can have larger capacity and lower power consumption than traditional DRAM-based main memory. It can also realize durable data structures by just storing the data to the NVMM instead of writing it to the file system through costly OS system calls. For these characteristics, NVMM will be popularized in embedded systems as well as server systems. However, it also introduces several drawbacks over traditional DRAM, such as relatively longer latency and narrower bandwidth than DRAM. Furthermore, its write operations usually cause a longer latency and larger energy consumption in the memory system than its read operations. Data durability will also come at the cost of expensive cache eviction and memory barrier operations [3].

Both software and hardware in a system must be cooperatively optimized for NVMM to sufficiently extract its performance and advantages because of different characteristics from traditional DRAM and flash devices. Nevertheless, only a few commercially NVMM modules are available and all of them are for rich servers. Intel Optane DC persistent memory (DCPMM) [4] is a representative NVMM released by Intel and Micron in April, 2019. It must be operated by memory controllers integrated in specific Xeon processors due to unique DDR-T protocol. The lack of embedded NVMM results in the use of simulators or emulators that were proposed in the existing studies.

Software-based simulators [5]–[9] enable cycle-by-cycle evaluations based on detailed models of memory systems at a micro architecture level. However, they require huge simulation time to evaluate large applications. In contrast, hardware-based emulators [10]–[14] built on real machines can execute applications at the speed of the base hardware and enable much faster evaluations than simulators. The base techniques of the emulators are injecting additional delays to memory requests. Quartz [10], TUNA v1 [11], and others [13], [14] inject delays based on the number of memory requests issued to a memory controller. Although these delay injection models can represent asymmetric delays between read and write operations, they are too coarse to capture the impact of bank parallelism and page locality in a memory module, which are important factors to reduce memory latency. TUNA v2.1 [12] introduced a new delay injection model that can capture them by injecting delays into primitive memory requests issued by a memory controller and evaluated some applications to reveal the impact of NVMM on application performance.

While existing NVMM emulators have made important contributions that enable to explore the possibilities of NVMMs, several issues still remain. First, the correctness and effectiveness of the delay injection method proposed in [12] are insufficiently shown. The model and implementation were not validated with golden models such as cycle accurate simulators. In addition, though the delay injection methods of [12] and [11] can have different impacts

Manuscript received April 21, 2020.
Manuscript revised November 24, 2020.
Manuscript publicized February 5, 2021.

†The authors are with Waseda University, Tokyo, 169–8555 Japan.
a) E-mail: oy@kasahara.cs.waseda.ac.jp
b) E-mail: keiji@waseda.jp

DOI: 10.1587/transinf.2020EDP7092

Copyright © 2021 The Institute of Electronics, Information and Communication Engineers

on performance in theory, it is not confirmed in experiments. Second, existing simulators and emulators do not consider DCPMM. While it is currently available only in rich servers, it will possibly be available in embedded systems in the future. Researchers have no way to evaluate it in embedded systems so far. Third, existing NVMM emulators emulate only one NVMM model and evaluation architecture is fixed. NVMM cells and architectures have been under research yet, thus emulators should be able to represent various NVMM models to explore optimization for NVMM. Fourth, most of papers mentioned above only focus on building evaluation environments. One of the important roles of NVMM simulators and emulators is to explore optimization techniques for NVMM, however, what factors impact on performance are not clearly shown.

In this paper, we propose an NVMM emulator for embedded systems built upon ARM multicore-based Zynq SoC board. We also explore factors that are important to reduce NVMM latency. This is an extension of the work originally published at NVMSA2019 [15]. Our emulator employs three behaviour models: coarse-grain, fine-grain and DCPMM-based. The coarse and fine-grain models represent expected NVMM behaviour by extending traditional DRAM-based main memories. The former model injects additional delays at the memory bus between the last level cache (LLC) and the memory controller. Similarly, the latter model injects delays at the memory controller. The DCPMM-based model is a new behaviour model based on a real DCPMM. It represents expected DCPMM behaviour and performance in embedded systems. These three behaviour models and implementations were validated with an existing NVMM emulator, a cycle-accurate NVMM simulator and a real DCPMM to confirm that they show the same behaviour and effectiveness as expected. Then, we reveal the impact of NVMM behaviour models on the performance of a system, especially focusing on the bank parallelism and the row-buffer access locality in a memory module, by using micro benchmarks and SPEC CPU 2017 benchmark programs.

The contributions of this paper are summarized as follows:

- We built an NVMM emulator employing three NVMM behaviour models: coarse-grain, fine-grain and DCPMM-based[†]. The coarse and fine models enable DRAM-based NVMM performance evaluations while the DCPMM model enables DCPMM-based NVMM performance evaluations on embedded systems.
- We also provided whole evaluation environment including Linux kernel modification, NVMM management library, and a kernel module for cache flush operations.
- We validated the behavior of proposed emulator models by comparing with an NVMM emulator, a cycle-accurate NVMM simulator, and a real DCPMM. Through validation, we also demonstrate the effectiveness of the fine-grain behaviour model.
- We revealed the impact of NVMM behaviour models, latency, and characteristics of memory requests on application performance.

The rest of this paper is organized as follows: Section 2 reviews related works on NVMM evaluation environment. Section 3 introduces three NVMM behavior models. Section 4 explains the implementation of the emulator by using models described in Sect. 3 and whole environment. Section 5 presents the validation results, then Sect. 6 discusses the experimental evaluation and its result. Finally, Sect. 7 concludes this paper.

2. Related Work

Software simulators and hardware emulators are current two major NVMM evaluation environments.

Gem5, NVMain, PCMSim, and HMMSim are examples of NVMM simulators [5]–[9]. They are implemented as software simulators that represent the micro architecture of target memory modules. While they enable cycle-accurate simulation with the flexible parameters and configuration settings, they require too much simulation time to evaluate system-wide performance for OS and compiler explorations.

TUNA [11], [12], Quartz [10], and others [13], [14] are examples of NVMM emulators. TUNA is built on an ARM-based SoC with FPGA chip. It originally employed a coarse-grain delay model such that the delay clock cycles were injected for the read and write operations given to the memory controller. Then, it introduced a fine-grain delay model in v2.1 [12]. It now injects delays for the primitive memory operations issued by the memory controller and thus, it offers a more realistic delay model. However, the impact of bank parallelism and row-buffer access locality that can be observed in a fine-grain delay model is still unclear. In this paper, we evaluate the programs in terms of these two points as well as the frequency of the memory requests that can lead to further software optimization techniques for OSs and compilers. Furthermore, we implemented an Intel Optane DC Persistent Memory [4] model in our emulator and evaluate it. Its performance model has not been implemented in the existing simulators and emulators so far.

3. NVMM Behaviour Models

To emulate NVMM performance with DRAM, additional latency must be injected into memory requests. According to the micro architecture of NVMM, several delay injection models can be assumed. In this section, we define *Behaviour Models* that represent possible NVMM architecture and behaviour. We introduce overview and behaviour of existing memories, then derive *Behaviour Models* from them.

[†]The emulator and related software are available at https://github.com/uyiromo/nvmtest.

3.1 Overview and Behaviour of Traditional DRAM-Based Main Memory

This section describes a behaviour of memory controllers and traditional DRAM-based main memory. The coarse-grain model (Sect. 3.2) and the fine-grain model (Sect. 3.3) are the extensions of it.

Figure 1 depicts the behaviour and architecture of DRAM-based main memory. A memory module consists of memory cells and row buffers. When CPU issues memory a request, it will be split up into some DDR commands in memory controllers. Three commands are mainly used (*ACT*, *R/W*, *PRE*) as follows.

1. *Activate (ACT)* opens one page and content of memory cells are read into row buffers.
2. A memory controller reads data from or writes data into the row buffers (*R/W*).
3. *Precharge (PRE)* writes back row buffers to memory cells and the page is closed.

According to DDR protocols [16], *ACT* and *PRE* are not always required. They are required only when a request misses row buffers. Row buffers are written back to memory cells by *PRE*, then a new row is loaded by *ACT*. Thus, the memory latency depends on row-buffer hit ratio.

The coarse-grain model (Sect. 3.2) and the fine-grain model (Sect. 3.3) are different in the granularity of delay injection. The former one injects delay into memory requests, on the other hand, the latter one injects delay into DDR commands.

3.2 Coarse-Grain Behavior Model

According to Sect. 3.1, the simplest NVMM model injects additional delay into ALL memory requests between the last level cache (LLC) and the memory controller ("0. READ/WRITE (from LLC)" in Fig. 1). While this simple model is widely used in existing works [10], [11], it can not represent the effects of row-buffer hit ratio. For instance, read requests will be delayed even if they hit row buffers.

In comparison to Sect. 3.3, this model is coarse because all memory requests are delayed without taking DDR commands into account. Thus, we define this model as *Coarse-Grain Behaviour Model*. This model represents NVMM that have no caches like row-buffer in memory modules, thus all memory requests are delayed.

3.3 Fine-Grain Behaviour Model

Contrary to the coarse-grain behaviour model, detailed NVMM model injects additional delay into DDR commands issued by a memory controller ("1. ACTIVATE (tRCD)" and "3. PRECHARGE (tRP)" in Fig. 1). In comparison to the coarse-grain one, this model delays memory requests only if they access memory cells and can represent the impact of row-buffer locality and bank parallelism. In addition, this model extends the DRAM-based behaviour as follows:

- Memory cells must be accessed only by *ACT* and *PRE*
- A memory controller in an SoC chip and a micro controller on a memory module are extended to manage the dirt of row buffers

NVMM holds data by physically stable way than DRAM and its memory cells are worn out especially by write operations. To reduce the latency and the number of writes, a memory controller should manage states of a row-buffer and write back data in the buffer only on the dirty case. Besides, NVMM does not require abundant write-backs because memory cells are not damaged by DRAM-like disruptive read operations.

In comparison to Sect. 3.2, this model is fine because it considers detailed DDR commands (*ACT*, *R/W*, *PRE*) and the effect of row-buffers. Thus, we define this model as *Fine-Grain Behaviour Model*. This model represents NVMM that has architecture similar to the traditional DRAM having the organization of banks, rows, and columns, thus memory requests are delayed only when they miss row-buffers.

3.4 Overview and Behaviour of Intel DC Persistent Memory

Figure 2 depicts the architecture of Intel DC Persistent

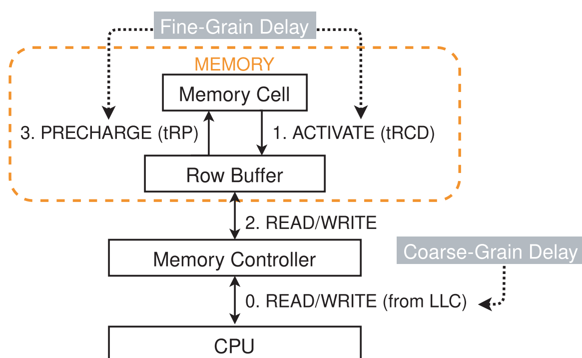


Fig. 1 Behaviour of DRAM-based main memory [15]

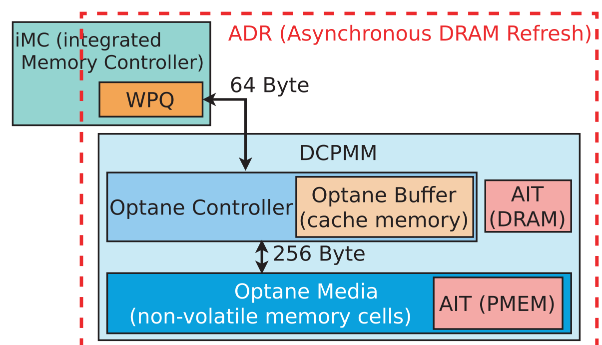


Fig. 2 Intel DC persistent memory architecture [4]

Memory (DCPMM) [4]. It requires specific Xeon processors due to the unique DDR-T protocol. DDR-T is defined as an extension of the DDR-4 protocol to realize asynchronous and out-of-order operations. DCPMM consists of the 3D Xpoint-based optane media [17] and the optane buffer. The address translation table (AIT) is provided for wear leveling. The optane controller controls them. The granularity of a communication is 64-bytes between the integrated memory controller (iMC) in a CPU chip and DCPMM. Similarly, it is 256-bytes between the optane controller and the optane media. We modeled its behaviour and provided environment to explore optimization techniques for DCPMM on embedded systems.

Figure 3 depicts the average access latency measured by a micro benchmark shown in Fig. 4 on a Xeon and DCPMM machine (Table 1) under the following conditions:

- An execution processor core was fixed by *taskset* command.
- A prefetcher was disabled.
- Non-temporal instructions and memory barriers are used to prevent the impact of CPU caches [18].
- We filled all allocated DCPMM with zeros to make page tables in advance, then invalidated all cache lines associated with them before measurement.

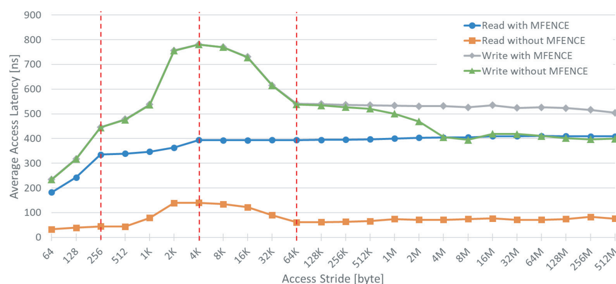


Fig. 3 Average Latency on Real DCPMM while changing *STRIDE*. "*with MFENCE*" means that all memory requests are ordered strictly by memory barrier instructions. "*without MFENCE*" uses memory barrier only before and after a loop.

```
base := return value of mmap()
start = clock();
for (offr = 0; offr < 124*GiB; offr += STRIDE) {
#if defined(READ)
    val = movntdqa(base+off);
#elif defined(WRITE)
    movntdq((base+off), val);
#endif
_mm_mfence(); or NO_FENCE
}
end = clock();
```

Fig. 4 Micro benchmark for latency measurement of DCPMM

Table 1 Configuration of DCPMM

CPU	Xeon Gold 5222 @3.80 GHz
DCPMM	DCPMM 128 GiB
Configuration	AppDirect not Interleaved device DAX (devdax)
Operating System	Ubuntu 18.04LTS

When looking over Fig. 3 from left to right, latency trends change at three points: 256 byte, 4K byte and 64K byte, respectively. The latencies for "Write with/without MFENCE" and "Read without MFENCE" increase until 4K byte, reach a maximum at 4K byte, decrease until 64K byte, and are constant until 512M byte. In contrast, "Read with MFENCE" shows constant latency from 4K byte. A DCPMM module used in this evaluation has eight data chips on it, each of which has 256-bytes buffer (totally 2KiB per module). Therefore, stride memory accesses up to 2KiB stride width can fully utilize those buffers. After that, doubling the stride width should cause heavy buffer access contentions resulting in the constant latency. However, as previously described, the latency decreases after 4KiB stride for three cases. Although the detail of the optane controller is unclear, interleaving accesses among eight data chips seems to be happened by utilizing some address translation when an exceeding of a certain amount of stride width in memory accesses is detected. This may be performed to avoid heavy memory access contentions to one data chip considering the endurance of a memory device. The latency differences between "with MFENCE" and "without MFENCE" are caused by the memory access parallelism, which is limited by "MFENCE" instructions.

In addition, DCPMM seems not to have bank parallelism. "Read without MFENCE" shows constant latency from 4K to 512M. ("Write"s are inappropriate as they are affected by write queues.) If a DCPMM has bank parallelism, latency of unordered read requests should decrease at some points.

3.5 DCPMM-Based Behaviour Model

In this section, we define *DCPMM-based Behaviour Model* based on the observations in Sect. 3.4 to emulate DCPMM on the emulator. This model represents NVMM whose architecture is similar to DCPMM (Fig. 2), thus memory requests are delayed like DCPMM.

Embedded systems are difficult to have rich memory controllers like DCPMM due to their cost limitation, hence the model in the emulator omits the impact of out-of-order execution and data-chip level interleaving. According to Fig. 3 and the abstraction above, this models the following trends:

- The read latency increases sharply when the address difference of two successive memory accesses is from 64 byte to 256 byte, and slowly from 256 byte to 4K byte (like "Read with MFENCE").
- The write latency increases slowly when the address difference of two successive memory accesses is from 64 byte to 256 byte, sharply from 256 byte to 4K byte (like "Write with MFENCE").
- Both read and write latencies are constant from 4K.
- The bank parallelism does not exist.

4. Implementation

4.1 Overview

The NVMM emulator in this paper is built on a Xilinx Zynq-7000 SoC ZC706 board with FPGA (Table 2). A ZC706 board has PS and PL sections. While the PS contains two CPU cores, an L2 cache as the Last Level Cache (LLC), and peripheral circuits, the PL has the FPGA. The ZC706 has two DRAM modules: one is connected to the PS, the other is connected to the PL. The DRAM connected to the PL is used for emulating an NVMM. To do so, the Memory Interface Generator (MIG) on the PL is used as the memory controller for the NVMM, as depicted in Fig. 5. The following steps have been implemented to provide the NVMM emulator environment:

1. Implementation of delay injection logic
2. Linux kernel modification for making the NVMM cacheable
3. Implementation of a kernel module to enable cache flush operations from user programs
4. Implementation of functions in C language for allocating and deallocating the NVMM region

4.2 Coarse-Grain Delay Injection

The coarse-grain delay injection is based on the coarse-grain behaviour model (Sect. 3.2). A delay injection module is inserted between the LLC and the MIG to inject the specified read/write delay clock cycles for memory requests. The specified clock cycles can be set by the user as required.

Table 2 Specification of baseline platform for emulator

FPGA	Xilinx Zynq-7000 SoC ZC706
Device	Zynq-7000 XC7Z045-2FFG900C SoC
CPU Core	Cortex-A9 Dual Core, 667 MHz
L1 Cache	I=32 KiB/core, D=32 KiB/core
L2 Cache	512 KiB/processor
PS DRAM	1 GiB, DDR3-1066, 16b×2 components
PL DRAM	1 GiB, DDR3-1600, 8b×8, SO-DIMM
PL Frequency	200 MHz
OS	GNU/Linux 4.14.0-xilinx-00081-g88cc987 [19] Ubuntu 16.04 LTS

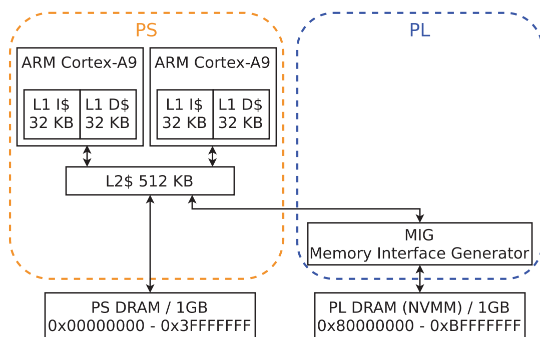


Fig. 5 Emulator overview [15]

4.3 Fine-Grain Delay Injection

The fine-grain delay injection is based on the fine-grain behaviour model (Sect. 3.3). We modified the RTL code of MIG to inject additional read/write delays for *ACT* and *PRE*. The MIG waits for *tRCD* nanoseconds after issuing ACT, and waits for *tRP* nanoseconds after issuing PRE, respectively. The modified MIG can insert additional latency into *tRCD* and *tRP* as the user required. The fine-grain delay injection does not delay successive memory requests if they hit row buffers.

4.4 DCPMM-Based Delay Injection

The DCPMM-based delay injection is based on the DCPMM-based behaviour model (Sect. 3.5). This model is implemented as an extension of the coarse-grain delay injection so that it injects additional latency only if physical addresses are multiple of 256 or 4,096. Different latencies on 256- and 4,096-byte boundaries realize different behaviour between 64-256 and 256-4K as depicted in Fig. 3. It can also consider the impact of access locality. For instance, when successive memory requests access physical addresses of 0x4000 and 0x40002, only the former is delayed. This implementation can ignore bank parallelism discussed in Sect. 3.5.

4.5 Kernel Modification

In our emulator, the system has a heterogeneous main memory consisting of DRAM and NVMM, and the user process must use NVMM explicitly through dedicated memory allocation APIs. In order to distinguish the NVMM region (PL DRAM in Fig. 3.1) from the DRAM explicitly, it must be excluded from the system RAM managed by Linux kernel. If the system RAM includes NVMM region, the kernel may allocate it to the system or user processes unintentionally.

In addition to NVMM exclusion from the system RAM, we must also consider that the Linux kernel provided by Xilinx [19] treats only system RAM as "cacheable" region. This cacheability management causes serious performance loss when a program uses NVMM outside the system RAM.

Therefore, we modify the kernel to allocate the NVMM as a cacheable region. The modified Linux kernel provides mmap system call to allocate the NVMM region outside the system RAM to user memory space. The region cacheability is determined in the mmap system call. In the modified kernel, cacheability of the NVMM region can be specified with "O_SYNC" flag. If "O_SYNC" is not specified in mmap(), the region will be allocated as "cacheable", otherwise "non-cacheable".

4.6 Cache Flush Operation

The NVMM can guarantee data persistency only when the

data arrives in it. If the CPU cache is enabled, the data will firstly be written into only the cache. Therefore, the data must be explicitly evicted from the CPU cache to the NVMM to ensure the data persistency. The ARM Cortex-A9 core (ARMv7-A ISA) on ZC706 has cache flush instructions for this purpose. However, they are privileged, thus an interface for them available from a user program must be provided.

We develop a kernel module that enables the user applications to issue CPU cache flush operations. The user can also specify the target address range to reduce the system call overhead. The flush instructions running in a loop evicts the data in the cache lines within the specified address region and they are performed in parallel by the hardware as much as possible. The memory barrier instructions are executed before and after the cache flush loop to ensure the data consistency.

4.7 NVMM Management Library

We develop a library for the memory allocation from the NVMM region whose interface is compatible with the standard C library functions, such as malloc, calloc, realloc, and free. The functions in the library are implemented by wrapping mmap/munmap system calls described in Sect. 4.5. The implemented functions are as follows.

```
void *NVMM_Malloc(size_t size)
void *NVMM_Calloc(size_t nmemb, size_t size)
void *NVMM_Realloc(void *ptr, size_t size)
void  NVMM_Free(void *ptr)
```

5. Validation of Accuracy

In this section, we demonstrate the correctness and reliability of our behaviour models and implementation by comparing them with golden models. Golden models in this paper are an existing NVMM Emulator [14] for the coarse-grain model, an NVMM simulator for the fine-grain model, and the real DCPMM module for the DCPMM-based model, respectively.

5.1 Validation of the Coarse-Grain Model

To validate the coarse-grain behaviour model, we compare it with the results in [14]. It implements the coarse-grain model and validates the implementation by confirming that the measured latency agrees with the expected latency, which is set to the emulator. We use the same validation methods.

The average latency is measured by using a micro benchmark shown in Fig. 6. The access strides (*STRIDE* in Fig. 6) are defined as 32 or 8192 to confirm that the coarse-grain model cannot consider row-buffer hit ratio. If the stride is 32, successive requests hit row-buffer and the average latency will be reduced because 32 is smaller than row-buffer size (8192).

```
base := return value of mmap()
start = clock();
for (offs = 0; offs < 1*GiB ; offs += STRIDE) {
#if defined(READ)
    val = *((volatile size_t) (base + offs));
#elif defined(WRITE)
    *((volatile size_t) (base + offs)) = 0;
#endif
}
end = clock();
ave = (end - start)/(1*GiB/STRIDE)
```

Fig. 6 Micro benchmark for measuring average latency

Table 3 Average latency while changing expected latency. “32” and “8192” are access strides in bytes.

Expected Latency [ns]	Measured Latency [ns]			
	READ		WRITE	
	32	8192	32	8192
200	211	211	215	217
400	411	416	414	422
600	612	615	616	623
800	812	815	817	817
1000	1012	1015	1019	1023

Table 3 shows the results. Each expected latency is injected into both read and write by the coarse-grain model. First, a small error ($\sim$ 23ns) is shown between each expected latency and the corresponding measured one. On the emulator, the expected latency is injected at the memory bus. The measured latency includes latency between a CPU core and the memory bus in addition to the expected one. These errors are derived from it because the differences between the expected and the measured latencies are almost constant while the expected latencies are changed. This result shows that the coarse-grain model is implemented on our emulator correctly. Second, in Table 3, the changes of the access strides have no impact on the measured latency. This result shows that the coarse-grain model is not affected by the row-buffer hit as described in Sect. 3.2.

Through this section, the coarse-grain model and its implementation was validated in comparison to [14]. The emulator shows its performance characteristics (no row-buffer locality) as expected.

5.2 Validation of the Fine-Grain Model

To validate the fine-grain behaviour model, we compare it with a cycle-accurate NVMM simulator combined with a multicore simulator. Besides, the fine-grain model is compared with the coarse-grain one to confirm the effects caused by its behavior model, such as row-buffer access locality and bank parallelism. We use gem5 [5] and NVMain2 [7] configured as Table 4. The ARMv7 CPU model in gem5 is modified to precisely simulate Cortex-A9 core according to [20]. NVMain2 is modified to change the latency for *ACT* and *PRE*.

First, we measured the average latency of the fine-grain

Table 4 The baseline simulator configuration

gem5	
Simulation Mode	Syscall Emulation
CPU Frequency	667 MHz
CPU Core	O3_ARM_v7a_3 x1
Cache Line Size	32 byte
L1 Cache	I=32 KiB, D=32 KiB
L2 Cache	512 KiB
NVMain2	
Frequency	166 MHz
Size	1 GiB
Command Queue	READ=32 entries, WRITE=32 entries
Page Policy	Relax Page

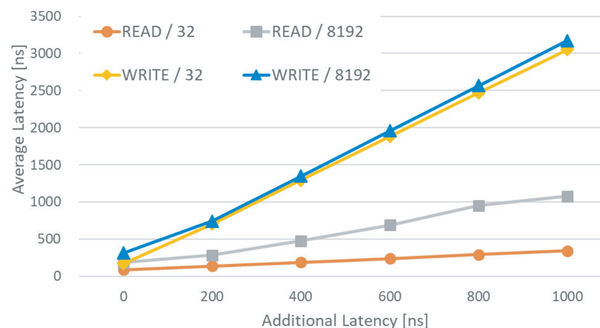
delay injection on the emulator and the simulator by using a micro benchmark shown in Fig. 6. Access strides (*STRIDE* in Fig. 6) are defined as 32 or 8,192 to confirm the effect of raw-buffer hit ratio as described in Sect. 5.1.

Figure 7 (a) and Fig. 7 (b) show the results of the proposed emulator and the simulator (gem5+NVMain2), respectively. “WRITE/32” in Fig. 7 (b) is calibrated based on the raw data shown in Fig. 7 (c). When the emulator and the simulator run the same benchmark for “WRITE/32”, the number of *ACT*/*PRE* differs. For instance, when 1,048,576 reads and writes are issued, the simulator issues 1,656,898 *ACT*/*PRE*, on the other hand, the emulator issues 2,050,429 *ACT*/*PRE*. This difference comes from detailed memory controller architectures, such as command queue, arbiter, connection between modules, and so on. To compare the simulator and the emulator appropriately, we calibrate the result according to the number of *ACT*/*PRE* because the fine-grain model injects additional latency into *ACT*/*PRE*. In the example above, each value in Fig. 7 (b) is calculated by multiplying each data in Fig. 7 (c) by (2,050,429/1,656,898).

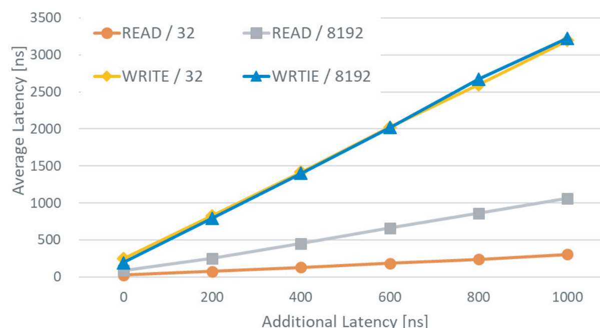
According to these graphs, while the additional latency has a small impact on “READ/32”, “WRITE/8192” is affected largely. Comparing the latency characteristics of the emulator with the simulator, each case has almost the same slope. Figure 7 (a) and Fig. 7 (b) show that ‘WRITE/32” and “WRITE/8192” have almost no difference while they differ in Fig. 7 (c). The architecture of the MIG IP on the emulator is conservative, hence this result is caused by the detailed memory controller architecture. These trends are allowable because these results are measured by much heavy write requests and such heavy write requests are basically not suitable for the NVMM.

Second, we measured the impact of bank parallelism on the coarse-grain, the fine-grain, and the simulator by using a micro benchmark shown in Fig. 8. Average latency was measured while changing *NBANK* that represents the number of banks to be accessed in parallel.

Figure 9 (a) and Fig. 9 (b) show the normalized average read and write latency, respectively. Comparison between “Coarse” and “Fine (Emulator)” reveals that only the latter can consider parallelism. For instance, the read latency and the write latency of “Fine (Emulator)” decrease to about 60% and 50% respectively while they are constant for



(a) Average latency on the proposed emulator



(b) Calibrated average latency on the simulator (gem5+NVMain2)

(c) Average latency on the simulator (Raw Data)

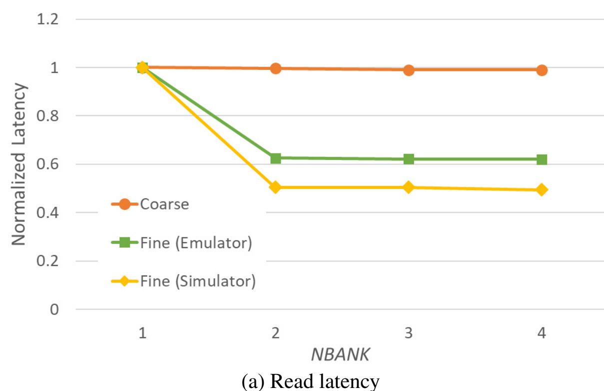
Fig. 7 Average latency while changing additional latency ((a): the Emulator, (b): the Simulator). 32 and 8192 are access strides. Additional latencies are injected by fine-grain delay injection.

“Coarse”. In addition, “Fine (Emulator)” and “Fine (Simulator)” show similar trends. In particular, the read latency decreases until $NBANK = 2$ then becomes almost constant, and the write latency decreases until $NBANK = 4$. The different trends among them are caused by their detailed architecture difference described above. These results demonstrate that the fine-grain model on the emulator and the simulator show similar behaviour as expected.

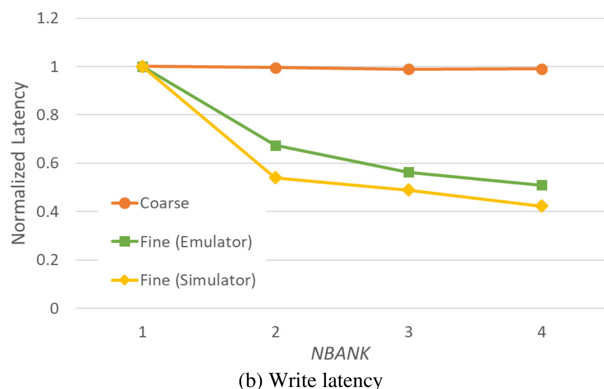
The results above demonstrate the correctness, reliability and effectiveness of the fine-grain model and implementation on our emulator. The two evaluations above show that the fine-grain delay injection is implemented correctly by comparing it with cycle-accurate NVMM simulators. In addition, we confirmed that only the fine-grain delay injection can capture the effect of row-buffer access locality

```
#define NROW   (16384)   // rows in one bank
#define NBANK  (8)
#define SZROW  (8*KiB)   // bytes in one row
#define SZBNK (128*MiB)  // bytes in one bank

base := return value of mmap()
start = clock();

// each row, each bank
for (st = 0; st < NROW*SZROW; st += SZROW){
  for (of = st; of < SZBNK*NBANK; of += SZBNK) {
    addr = base + of;
#if defined(READ)
    val = *((volatile unsigned long *)addr);
#elif defined(WRITE)
    *((volatile unsigned long *)addr) = 0L;
#endif
  }
}
end = clock();
```

Fig. 8 Micro benchmark for measuring bank parallelism



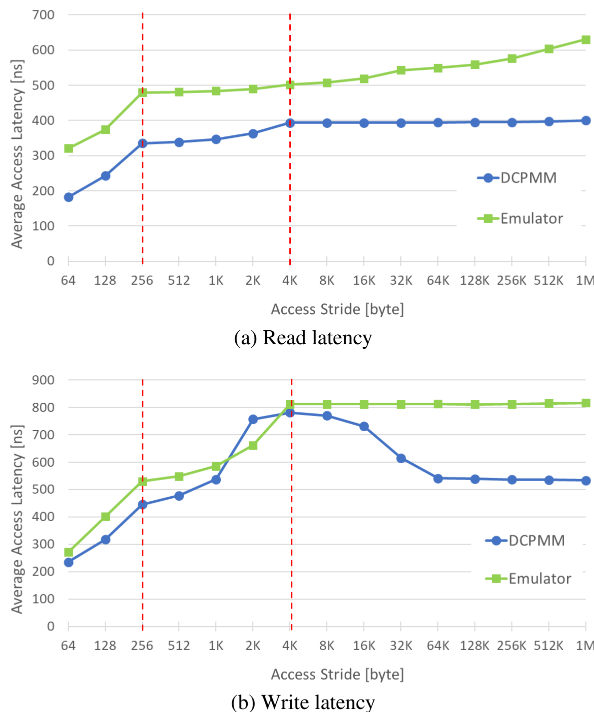
(a) Read latency



(b) Write latency

Fig. 9 Normalized average latency while changing *NBANK* ((a): Read, (b): Write). They are normalized against to the results when *NBANK* = 1. *Coarse* is the results of the coarse-grain delay injection. *Fine (Emulator)* and *Fine (Simulator)* are that of the fine-grain delay injection on the emulator and on the simulator, respectively. Additional 1,000ns are injected for both read and write.

and bank parallelism as expected. By comparing Table 3 and Fig. 7 (a), only the fine-grain model can consider the row-buffer access locality. Similarly, Fig. 9 (a) and Fig. 9 (b) show that only the fine-grain model can consider the bank parallelism.



(a) Read latency

(b) Write latency

Fig. 10 Average latency ((a): Read, (b): Write). *DCPMM* are the same as *with MFENCE* in Fig. 3. *Emulator* are the results of the emulator employing the DCPMM-based behaviour model (Sect. 3.5). In (a), additional 200ns and 225ns are injected at 256-byte and 4,096-byte boundaries, respectively. In (b), additional 500ns and 800ns are injected at 256-byte and 4,096-byte boundaries, respectively.

5.3 Validation of the DCPMM-Based Model

To validate the DCPMM-based behaviour model, we compare it with a real DCPMM module (Table 1). For DCPMM, "Read with MFENCE" and "Write with MFENCE" are used because of constraints in Sect. 3.5. For the average latency measurement, we prepare a micro benchmark shown in Fig. 6 that is an emulator's corresponding benchmark to Fig. 4. *STRIDE* was up to 1 MiB to get enough results. In this evaluation, CPU caches were disabled instead of using non-temporal instructions.

Figure 10 (a) and Fig. 10 (b) show the results. These results show that the emulator shows the same behaviour as DCPMM between 64-4K, then show constant latency for the case of the DCPMM-based model (Sect. 3.5). The increasing trend of the read latency of the emulator that is different from the real DCPMM (Fig. 10 (a)) is allowable. This result is caused by the continuous accesses by the micro benchmark. This kind of heavily memory access intensive applications are basically not suitable for the NVMM.

In Fig. 10 (b), "DCPMM" and "Emulator" show the same trends from 64 to 4K, and then show the different trends. The trend of "DCPMM" from 4K to 1M is probably caused by the advanced control of the optane controller. Such advanced micro controllers are not expected for an embedded system in terms of the cost.

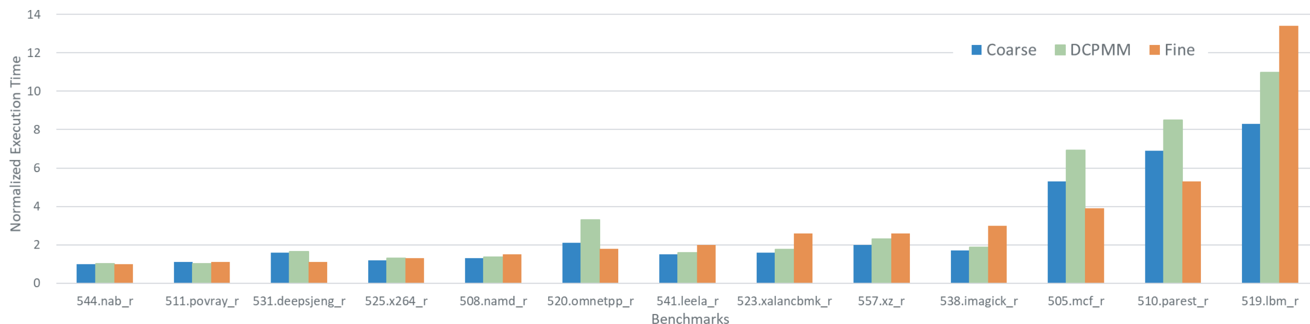


Fig. 11 Normalized execution time of SPEC CPU 2017 programs. *Coarse* and *Fine* are the result of coarse-grain and fine-grain delay injection when *ARL* = *AWL* = 1,000. *DCPMM* is the result of DCPMM-based delay injection. To compare *DCPMM* with other models, 2,000ns and 2,250ns are injected for read at 256 or 4K byte boundaries, 5,000ns and 8,000ns for write at 256 or 4K byte boundaries, and 1,000ns for other requests. All of them are normalized against to the execution time when no additional latency are injected.

Through this section, the DCPMM-based model and its implementation were validated in comparison to a real DCPMM module. Figure 10 (a) and Fig. 10 (b) show that the emulator can emulate DCPMM performance with allowable errors.

6. Experimental Evaluation with SPEC CPU Benchmark

This section describes the experimental evaluation of the NVMM emulator environment to explore a direction of optimization techniques. Two parameters, *ARL* and *AWL*, used throughout this section are defined as follows:

- *ARL*: Configured read latency in nanoseconds
  - coarse-grain: expected read latency
  - fine-grain: additional tRCD
- *AWL*: Configured write latency in nanoseconds
  - coarse-grain: expected write latency
  - fine-grain: additional tRP

6.1 Normalized Execution Time of SPEC 2017 Benchmark Programs

This section demonstrates how three models affect real application performance differently and what factors impact on performance by using the emulator. Three models are compared by using SPEC CPU 2017 benchmark [21]. Fourteen of 24 programs are chosen from SPEC CPU rate benchmark programs. They are written in C/C++ and can be successfully compiled and executed on the emulator. We replaced all malloc, calloc, realloc, and free functions with NVMM_Malloc, NVMM_Calloc, NVMM_Realloc, and NVMM_Free described in Sect. 4.7 to allocate heap objects on the NVMM.

For the coarse-grain and the fine-grain models, both of *ARL* and *AWL* are set to 1,000. For the DCPMM-based model, 1,000 nanoseconds are injected as the base latency, and the extra latency is injected for 256 or 4K byte boundaries. Our experiment and the software optimization manual by Intel [22] indicate that read requests to DCPMM become twice slower if they miss the optane buffer. In other words, they step over 256-byte boundaries. Thus, 2,000 nanoseconds (1,000 $\times$ 2) for read at 256-byte boundary, and then 2,000 $\times$ 1.25, 2,000 $\times$ 2.5, 2,000 $\times$ 4.0 nanoseconds are injected for other boundaries according to Fig. 10.

Figure 11 shows the evaluation result as the normalized execution time for each program to the execution time without any delay injection. These bars are sorted by normalized execution time of “Fine” in ascending order from left to right.

First, “Coarse” and “DCPMM” should be discussed. Though the “Fine” bars are sorted in ascending order, “Coarse” bars for 531.deepsjeng_r, 520.omnetpp_r, and 557.xz_r show longer execution time than their next to the right programs. “DCPMM” bars show the same trends as “Coarse” because the injection model of the DPCMM-based model is implemented based on the coarse-grain model. In the rest of this section, we focus on “Coarse” on behalf of them.

Looking at each bar, Fig. 11 shows that the delay models affect the latency differently depending on each program. For instance, the normalized execution time of 544.nab_r and 511.porvray_r are both almost 1.0 for both models. However, for 519.lbm_r, the normalized execution time of the coarse-grain model is 8.3 while that of the fine-grain model is 13.4; thus, the fine-grain model has a 1.61 times longer execution time than the coarse-grain one. In addition, for 531.deepsjeng_r, 520.pmnetpp_r, 505.mcf_r, and 510.parest_r, the coarse-grain model shows higher execution time than the fine-grain model, while the fine-grain model shows higher values for other programs.

For detailed investigation, memory access characteristics, such as the number of read/write requests to the NVMM, the number of ACT and PRE, and bank parallelism, are also measured. Memory requests between the LLC and the MIG are counted. Bank parallelism (*BANK_PARA*) is defined as follows:

Table 5 *BANK_PARA* and *ACT/REQ* for each program

Benchmark	*BANK_PARA*	*ACT/REQ*
544.nab_r	0.000	0.989
511.povray_r	0.000	0.844
531.deepsjeng_r	0.170	0.633
525.x264_r	0.070	0.964
508.namd_r	0.080	0.945
520.omnetpp_r	0.270	0.882
541.leela_r	0.000	0.913
557.xz_r	0.050	0.921
523.xalancbmk_r	0.000	0.970
538.imagick_r	0.000	0.987
505.mcf_r	0.280	0.809
510.parest_r	0.001	0.936
519.lbm_r	0.220	0.934

Table 6 Cache hit ratio and frequency of memory requests to NVMM for each program

Benchmark	Cache Hit Ratio [%]	Memory Requests [/s]
544.nab_r	99.998	2,615
511.povray_r	99.983	85,219
531.deepsjeng_r	99.784	623,954
525.x264_r	99.926	493,471
508.namd_r	99.858	669,040
520.omnetpp_r	97.968	1,561,328
541.leela_r	99.785	852,818
557.xz_r	99.596	1,824,788
523.xalancbmk_r	99.516	1,295,606
538.imagick_r	99.356	1,540,642
505.mcf_r	93.501	4,170,876
510.parest_r	95.384	5,967,728
519.lbm_r	88.551	11,812,742

1. If successive requests use different rows, add 1
2. Divide result of 1) by the total number of requests

Activate per requests (*ACT/REQ*) is defined by dividing the number of ACT by the total number of requests.

Table 5 shows *BANK_PARA* and *ACT/REQ* for each program. The programs are sorted similar to that shown in Fig. 11. This table shows that 531.deepsjeng_r has low *ACT/REQ* (0.633), showing high row-buffer access locality. It also shows that 520.omnetpp_r and 505.mcf_r have high *BANK_PARA* (0.270, 0.280), showing high bank parallelism. The values show why these programs show the fine-grain model attains a lower execution time in comparison with the coarse-grain model. They also prove that the fine-grain model can capture the effect of the row-buffer access locality and the bank parallelism as described in Sect. 5.2.

Although 510.parest_r is also an exception, its bank parallelism and row-buffer locality values are not good. In the coarse-grain injection, read and write requests can be processed in parallel, and either of read or write requests having a larger total number of requests can cause more impact on the total execution time than another. 510.parest_r has high read/write ratio (25.0), which is defined by dividing the number of read requests by write requests, to the NVMM. The significantly high read/write ratio for the coarse grain model spoils the parallelism of memory accesses resulting in a longer execution time than expected.

There still exists an important question: Which of the characteristics of an application mainly affect on the execution time? *BANK_PARA* and *ACT/REQ* shown in Table 5 are important factors. However, 505.mcf_r has high *BANK_PARA* (0.280) and low *ACT/REQ* (0.809), while the normalized execution time is longer than 538.imagick_r. To investigate this question, the cache hit ratio for the LLC and the frequency of memory requests to the NVMM are also measured. The frequency of memory requests is the number of memory requests per second. For this measurement, both *ARL* and *AWL* are set to 0.

Table 6 shows the measurement result for each program. As the normalized execution time of the fine-grain model gets longer from top to bottom, it is expected that the cache hit ratio will decrease and the memory requests frequency will increase. However, there are several exceptions, as shown by the underlined values in the table. One reason is attributed to the data location, because the cache hit ratio takes all memory requests into account not only to the heap area that is located on the NVMM but also to the whole memory area. Thus, the frequency of memory requests to the NVMM has more impact than the cache hit ratio for this evaluation.

Regarding the relationship between 505.mcf_r and 538.imagick_r, the former has twice the number of frequency accesses to the NVMM as the latter. This implies that the impact of the frequency of memory requests exceeds that of *BANK_PARA* and *ACT/REQ* for latency reduction. The same situation is found in 519.lbm_r and 510.parest_r.

As described previously, the fine-grain model has a higher execution time than the coarse-grain model for most programs (except 531.deepsjeng_r, 520.omnetpp_r, 505.mcf_r, and 510.parest_r). This is, of course, caused by the difference of the total write latency. However, *ACT/REQ* is another important factor. According to Table 5, the average *ACT/REQ* is about 0.90. This implies that most requests are processed with ACT and PRE together, resulting in the additional latency equaling $ARL + AWL$ (= 2,000 ns) in the fine-grain model.

6.2 Cache Flush Overhead

As described in Sect. 4.6, the data in the cache must be evicted to the NVMM to make it durable. This section demonstrates the impact of cache flush overhead and what factors impact on it. We insert cache flush instructions into each program in the SPEC CPU to make their main data structure durable. Four programs having the following characteristics are chosen: 508.namd_r has high data parallelism. 541.leela_r allocates a lot of small regions (20 Byte × 200,000). 557.xz_r allocates a large region and is an in-memory application. 519.lbm_r requires quite a high bandwidth. Table 7 presents the evaluation result of the overhead caused by the cache flush. In this table, an overhead of “zero” denotes the additional execution time caused by the cache flush operations when both *ARL* and *AWL* are set to 0. Similarly, an overhead of “coarse” and “fine” are the

Table 7 Cache flush overhead and flushed lines

Benchmark	Overhead [s]			Total Flushed Lines
	zero	coarse	fine	
508.namd_r	0.31	0.33	0.27	922,288
541.leela_r	0.30	0.35	0.28	248,525
557.xz_r	0.03	0.04	0.02	166,898
519.lbm_r	5.49	5.55	5.46	1,859,045

additional execution time when both *ARL* and *AWL* are set to 1,000 with the coarse-grain and the fine-grain injection models. "Total Flushed Lines" is the number of total cache lines flushed by the inserted flush instructions.

This table shows that "fine" is less than "coarse" and "coarse" is more than "zero". The former observation is due to high data locality. Memory requests caused by flushing the region have high row buffer access locality and additional latency is reduced. The latter observation shows that overhead is affected by *ARL* and *AWL*.

Regarding the amount of the overhead for each program, Table 7 indicates that it is mainly affected by the number of total flushed lines. However, 508.namd_r flushes about four times more lines than 541.leela_r and shows almost the same overhead, which is due to the granularity of flush operations. For 508.name_r, the large area is specified for each cache flush operation. Therefore, when the data is flushed, most part of it has been already evicted from the cache by line replacement, and resulting in the small number of the NVMM access. On the other hand, the small area is specified at a cache flush time for 541.leela_r, thus, when 541.leela_r flushes the data, most part of it is still in the cache and evicted by this flush operation. These cases indicate that the overhead caused by the explicit data eviction is affected by the cache flush granularity. However, it must be noticed that the data durability cannot be ensured until the end of a cache flush operation and the following memory barrier operation.

7. Conclusion

In this paper, we built an NVMM emulator environment on a Xilinx Zynq board having the ARM Cortex-A9 cores with FPGA. This emulator implemented three NVMM behaviour models: coarse-grain, fine-grain and DCPMM-based. The coarse-grain model is a coarse extension and fine-grain model is a detailed extension of the traditional DRAM behaviour. The DCPMM-based model implements abstract behaviour of the Intel Optane DC Persistent Memory (DCPMM). We also provided the cache flush software interface required for the persist operations, as well as the standard C library compatible NVMM allocation functions for this environment.

The above three models were validated with an NVMM emulator, a cycle-accurate simulator, and a real DCPMM. The comparison between the fine-grain and a simulator showed the effectiveness of the emulator implementation. In addition, the comparison between the fine-grain and the coarse-grain showed that only the former can capture the impact of the bank parallelism and the row-buffer access locality. The DCPMM-based implementation was compared with a real DCPMM and they showed similar behaviour.

The evaluation investigated the performance difference among three models by using the SPEC CPU 2017 benchmark. It also assessed the impact on the execution time due to the bank parallelism, the row-buffer access locality, and the frequency of the NVMM requests. The evaluation results with the SPEC benchmark demonstrate that the frequency of the NVMM requests has a higher impact on the execution time than the cache hit ratio for the total execution time. In addition, high bank parallelism and high row-buffer access locality can reduce the NVMM access latency. These three parameters should be considered when software optimization techniques for OSs and the compilers are explored.

Acknowledgements

The authors would like to thank Mr. Toshiya Otomo and Mr. Tomokazu Yoshida from Fixstars for valuable discussions. This work was partly executed under the cooperation of organization between Kioxia Corporation and Waseda University.

References

[1] M. Webb, "Overview of persistent memory," Flash Memory Summit 2018, 2018.
[2] B. Gervasi and J. Hinkle, "Overcoming system memory challenges with persistent memory and nvdimm-p," JEDEC Server Forum, 2017.
[3] S. Pelley, P.M. Chen, and T.F. Wenisch, "Memory persistency," Proc. 41st Annual International Symposium on Computer Architecture, ISCA '14, Piscataway, NJ, USA, pp.265–276, IEEE Press, 2014.
[4] L. Looi and J.J. Xu, "Intel® optane^{TM} data center persistent memory," Hot Chips (HC) 31, Aug. 2019.
[5] 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.
[6] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," 2012 IEEE Computer Society Annual Symposium on VLSI, pp.392–397, Aug. 2012.
[7] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," IEEE Computer Architecture Letters, vol.14, no.2, pp.140–143, July 2015.
[8] J. Wang and B. Wang, "Pcmsim: A hybrid memory system simulator for the cloud storage," 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD), pp.81–86, Aug. 2017.
[9] S. Bock, B.R. Childers, R. Melhem, and D. Mosse, "Hmmsim: a simulator for hardware-software co-design of hybrid main memory," 2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA), pp.1–6, Aug. 2015.
[10] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," Proc. 16th Annual Middleware Conference, Middleware '15, New York, NY, USA, pp.37–49, ACM, 2015.
[11] T. Lee, D. Kim, H. Park, S. Yoo, and S. Lee, "Fpga-based prototyping systems for emerging memory technologies," 2014 25nd IEEE International Symposium on Rapid System Prototyping, pp.115–120, Oct. 2014.

[12] T. Lee and S. Yoo, “An fpga-based platform for non volatile memory emulation,” 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp.1–4, Aug. 2017.
[13] A. Koshiba, T. Hirofuchi, S. Akiyama, R. Takano, and M. Namiki, “Towards write-back aware software emulator for non-volatile memory,” 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp.1–6, Aug. 2017.
[14] A. Koshiba, T. Hirofuchi, R. Takano, and M. Namiki, “A software-based NVM emulator supporting read/write asymmetric latencies,” IEICE Trans. Inf. & Syst., vol.102-D, no.12, pp.2377–2388, 2019.
[15] Y. Omori and K. Kimura, “Performance evaluation on nvmm emulator employing fine-grain delay injection,” 2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp.1–6, Aug. 2019.
[16] J.S.S.T. ASSOCIATION, “Ddr3 sdram standard (revision of jesd79-3e, july 2010),” July 2012.
[17] M. Webb, “Markets for 3d-xpoint,” Flash Memory Summit 2018, 2018.
[18] Intel, “Intel® 64 and ia-32 architectures software developer’s manual,” 2019.
[19] Xilinx, “Xilinx/linux-xlnx: The official linux kernel from xilinx,” https://github.com/Xilinx/linux-xlnx, Accessed on 2020-1.
[20] F.A. Endo, D. Couroussé, and H. Charles, “Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5,” 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), pp.266–273, 2014.
[21] spec.org, “Spec cpu(r) 2017,” https://www.spec.org/cpu2017/, Accessed on 2020-1.
[22] Intel, “Intel® 64 and ia-32 architectures optimization reference manual,” 2019.

Yu Omori received his B.E. and M.E. in Computer Science and Engineering from Waseda University in 2019 and 2020. He is now a Ph.D. student of Computer Science and Engineering of Waseda University. He is a member of IEEE Eta Kappa Nu Mu Tau Chapter. His research interest includes non-volatile main memory for embedded systems. His research interest includes non-volatile main memory for embedded systems.

Keiji Kimura received the Ph. D degrees in electrical engineering from Waseda University in 2001. He was an assistant professor in 2004, associate professor in 2005, and professor in 2012 at Waseda University. He is a director of Green Computing System Research Organization in Waseda from 2019. He is a recipient of 2014 MEXT (Ministry of Education, Culture, Sports, Science and Technology in Japan) award. His research interest includes multicore processor architecture and parallelizing compiler technologies. He is a member of IPSJ, ACM and IEEE. He has served on program committee of many conferences.