Performance Evaluation on NVMM Emulator Employing Fine-Grain Delay Injection

Yu Omori
Waseda University
Tokyo, Japan
oy@kasahara.cs.waseda.ac.jp

Keiji Kimura
Waseda University
Tokyo, Japan
keiji@waseda.jp

***Abstract*—The emerging technology of byte-addressable non-volatile memory chips is expected to enable larger main memory and lower power consumption than the traditional DRAM. It also realizes durable data structure without ordinary file systems. However, while enumerating the advantages of non-volatile main memory (NVMM), its write-time expensive latency and higher energy consumption in comparision with a DRAM must be considered. These special characteristics of NVMM require new compiler techniques and OS support as well as new memory architectures. Several NVMM emulators built on real machines have been proposed to facilitate those software and hardware researches. Their designs were originally based on a simple coarse-grain delay model that injected additional clock cycles in the read and write requests sent to the memory controller. However, they could not utilize bank-level parallelism and row-buffer access locality, relied on by today's memory modules, to exploit their performance. Therefore, a fine-grain delay model was recently proposed where the delay is injected for the primitive memory operations issued by the memory controller. In this paper, we implement both the coarse-grain and the fine-grain delay models on an SoC-FPGA board along with the use of Linux kernel modifications and several runtime functions. Then, the program behavior differences between two models are evaluated with SPEC CPU programs. The fine-grain model reveals the program execution time is influenced by the frequency of NVMM memory requests rather than the cache hit ratio. Bank-level parallelism and row-buffer access locality also affect the memory access delay, and the fine-grain model shows lower execution time for four of fourteen programs than the coarse-grain even when the former has longer total write latency.**

I. 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. Larger memory capacity and lower power consumption in comparison with the traditional DRAM-based main memory are achieved because NVMM does not require the refresh operations. 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.

Although NVMM brings these attractive features to the system, 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 [1].

A new computer system employing NVMM must take these characteristics into account in terms of hardware, and especially software, because of the need for new OS and runtime system supports and new compiler optimization techniques.

While there is a strong demand for the exploration of new NVMM-related technologies, there are a few commercially available NVMM modules, resulting in a wide use of simulators and emulators by the previous studies. Software non-volatile memory simulators are already available, which can provide detailed models of memory modules and memory controllers at the micro architecture level [2], [3], [4], [5], [6]. Their detailed and cycle-accurate models are useful when a new micro architecture is investigated; however, they take too much time to investigate OS and compiler technologies.

In contrast, NVMM emulators built upon real machines enables rapid evaluations for the software exploration. The basic idea of their design is injecting an appropriate delay for read and write memory operations. They can basically execute at the speed of the base hardware, thus they can offer much faster experiment cycles than the software simulators.

TUNA [7], [8] is one such NVMM emulator. It is implemented on a Zynq Board with an ARM multicore-based SoC and FPGA chip. Its first version provided a coarse-grain delay model, where the clock cycles representing the delays for read and write operations were injected in front of the memory controller [7]. This model can represent the asymmetric delays between read and write operations. However, this model is too coarse to represent the actual latency in the memory cells, thus, it cannot evaluate the bank parallelism and the row-buffer access locality, which are two important factors for extracting the best performance from the memory modules. TUNA v2.1 introduced a fine-grain delay model, where the delay is injected for the primitive memory operations issued by the memory controller, hence removing the previous shortcomings [8]. Though bank parallelism and row-buffer access locality can be potentially evaluated by the new model, their impact on the NVMM-employed system during actual program execution is still unclear. Furthermore, cache eviction and memory barrier operations, which are required to ensure durable data structures on the NVMM, have not been investigated well on the emulators.

In this paper, we build an NVMM emulator employing both the fine-grain and the coarse-grain delay models. We also write memory-allocation functions in C language for the NVMM region, which are compatible with the ordinary C standard library functions, such as malloc, calloc, realloc, and free. Furthermore, we develop a kernel module to use a cache flush instruction, which is a privileged instruction in ARMv7-A ISA. After completing the build, first, we run a micro benchmark on the emulator to investigate the effect of the bank parallelism. Then, we execute programs from the SPEC CPU 2017 benchmark to evaluate the impact of the bank parallelism, row-buffer access locality, and frequency of memory requests. The overhead for the cache operations emulating the persistent operations are also evaluated by using SPEC programs.

The contributions of this paper can be summarized as follows:

- We develop an NVMM evaluation environment on an ARM-SoC and FPGA platform employing both the fine-grain and coarse-grain delay models. Additionally, we develop the memory allocation and cache eviction functions for that environment.
- We evaluate the impact of the bank parallelism, row-buffer access locality, frequency of memory requests, and cache eviction overhead on the emulator with SPEC CPU 2017 benchmark programs.

The rest of paper is organized as follows: Section II reviews related works on NVMM evaluation. Section III explains the implementation of the NVMM emulator environment. Section IV presents the experimental evaluation results. Finally, Section V concludes this paper.

II. Related Work

There are mainly two types of NVMM evaluation environments: simulators and emulators.

Gem5, NVMain, PCMSim, and HMMSim are examples of NVMM simulators [2], [3], [4], [5], [6]. 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 time to evaluate system-wide performance for OS and compiler explorations.

TUNA and Quartz are examples of NVMM emulators [7], [8], [9], [10]. 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 [8]. It now injects a delay for the primitive memory operations issued by the memory controller and thus, 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.

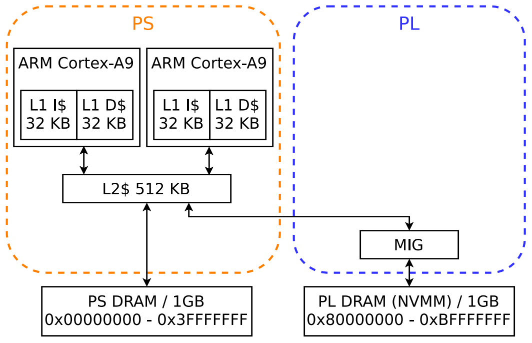

Fig. 1. Emulator Overview

TABLE I
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 KB/core, D=32 KB/core
L2 Cache	256 KB/processor
PS DRAM	1 GB, DDR3-1066, 16b×2 components
PL DRAM	1 GB, DDR3-1600, 8b×8, SO-DIMM
PL Frequency	200 MHz
OS	GNU/Linux 4.14.0-xilinx-00081-g88cc987 [11] Ubuntu 16.04 LTS

III. Implementation

A. Overview

The NVMM emulator in this paper is built on a Xilinx Zynq-7000 SoC ZC706 board with FPGA (Table I). A ZC706 board has two sections: PS and PL – the PS contains two CPU cores and peripheral circuits and 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 taken as NVMM, and the memory interface generator (MIG) on the PL is used as the memory controller for the NVMM, as depicted in Figure 1. 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

B. NVMM Behavior Model

The following behavior model is employed for the emulator in this paper:

- NVMM will be operated according to the existing main memory interface depicted in Figure 2 consisting of primitive requests, such as ACT, READ, WRITE, and PRE.

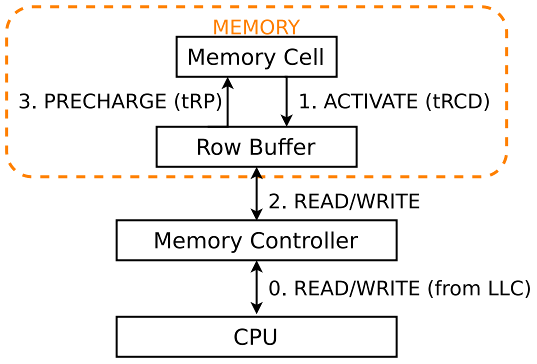

Fig. 2. NVMM behavior model

1) Open pages and read memory cells into row buffers (ACT)
2) Read/write to row buffer (RW)
3) Write back row buffers to memory cells and close pages (PRE)

- Memory cells are accessed only by ACT and PRE. Write command only writes data into row buffers and row buffers are written back to memory cells by PRE.
- Memory controllers are extended to manage the dirt of row buffers. Row buffers do not need to be written back if they are not dirty. In the extended model, only dirty row buffers are written back by PRE.

C. Coarse-Grain Delay Injection

The coarse-grain delay model is implemented by inserting a delay injection logic between the last level cache (LLC) and the MIG for the memory requests from the LLC ("0. READ-/WRITE (from LLC)" in Figure 2). It injects the specified read/write delay clock cycles for all memory requests from the LLC.

D. Fine-grain Delay Injection

The fine-grain delay injection model is implemented by modifying the RTL code of the MIG to inject the additional latency for ACT and PRE ("1. ACTIVATE (tRCD)" and "3. PRECHARGE (tRP)" in Figure 2).

The MIG waits for $tRCD$ ns after issuing ACT, and waits for tRP ns after issuing PRE, respectively. We modify the MIG to inject additional latency into $tRCD$ and tRP. Additional latencies can be set by the user as required. Successive memory requests are not delayed if they are issued to the same bank and row by introducing this fine-grain injection.

E. Kernel Modification

The mmap system call is employed to allocate NVMM region to the user memory space. The address space of PL-DRAM used as the NVMM region is allocated as "non-cacheable" by the Linux kernel provided by Xilinx [11]. Consequently, the NVMM is insufficient to be used by user applications instead of DRAM.

Therefore, we modify the kernel to allocate NVMM as a cacheable region. The region cacheability can be specified through the mmap system call. When the target address within the NVMM region is specified, if "O_SYNC" is not specified along with it, the region will be allocated as "cacheable", otherwise "non-cacheable".

F. Cache Flush Operatoin

NVMM can guarantee data persistency only when the data reach NVMM. If the CPU cache is enabled, the data will firstly be written into only the cache and not the main memory. Therefore, the data need to 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 rendering them unusable directly by the user programs.

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.

G. NVMM Management Library

We develop a library for the memory allocation of 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/mummap system calls described in Section III-E. 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)
```

IV. Experimental Evaluation

This section describes the experimental evaluation of the NVMM emulator environment. Two parameters, ARL and AWL, used throughout this section are defined as follows:

- ARL: Additional read latency in nanoseconds
  coarse-grain: additional read latency on memory bus
  fine-grain: additional tRCD
- AWL: Additional write latency in nanoseconds
  coarse-grain: additional write latency on memory bus
  fine-grain: additional tRP

A. Bank Parallelism

First, we evaluate the impact of the bank parallelism on NVMM. Today's memory modules rely on this parallelism to extract higher bandwidth because the successive memory requests can be processed in parallel if they are issued to different banks, thus hiding their access latency.

The following micro benchmark is used to measure the average NVMM access latency through the changing $NBANK$ values, which represents the number of banks to be accessed in parallel.

TABLE II
AVERAGE READ LATENCY

NBANK	Coarse-Grain [ns]	Fine-Grain [ns]
1	1071.0	634.0
2	1068.9	405.0
3	1061.6	399.5
4	1061.6	423.0

TABLE III
AVERAGE WRITE LATENCY

NBANK	Coarse-Grain [ns]	Fine-Grain [ns]
1	1072.6	3468.2
2	1067.3	1827.3
3	1060.9	1337.6
4	1061.8	1357.7

```
#define NROW   (16384)
#define NBANK  (8)
#define SZROW  (8*KiB)
#define SZBANK (128*MiB)

base := return value of mmap()
start = clock();
for (offr = 0; offr < NROW*SZROW; offr += SZROW) {
    for (off = offr; off < offr + NBANK*SZBANK;
         off += SZBANK) {
#if defined(READ)
    val = *((volatile unsigned long *)(base+off));
#elif defined(WRITE)
    *((volatile unsigned long *)(base+off)) = 0L;
#endif
    }
}
end = clock();
```

Table II and Table III show the measurement results of the average read/write latencies, respectively. Here, ARL and AWL are set to 1000.

These tables clearly reveal that the bank parallelism has a significant impact in the cases of the fine-grain model, while it has little impact for the cases of the coarse-grain model. For instance, comparing $NBANK = 4$ with $NBANK = 1$ in the fine-grain model, the average read latency is 33% lower (634.0 ns $\rightarrow$ 423.0 ns), and the average write latency is 61% lower (3468.2 ns $\rightarrow$ 1357.7 ns), respectively. On the other hand, the coarse-grain model shows almost the same latency (around 1066 ns) for both read and write operations even after changing the values of $NBANK$.

B. Normalized Execution Time of SPEC 2017 Benchmark Programs

Next, the fine-grain delay model is compared with the coarse-grain model by using SPEC CPU 2017 benchmark [12]. Fourteen programs that are written in C/C++ and can be successfully compiled and executed on the emulator are chosen from among 24 SPEC CPU rate benchmark programs. We replaced all malloc, calloc, realloc, and free functions with NVMM_Malloc, NVMM_Calloc, NVMM_Realloc, and NVMM_Free described in Section III-G to allocate heap objects on the NVMM. Figure 3 shows the evaluation result as the normalized execution time for each program. The normalized execution time is calculated by dividing the execution time when both ARL and AWL are set to 1000 by that when both ARL and AWL are set to 0. Each program is executed both on the coarse-grain and the fine-grain models. These bars are sorted by normalized execution time of the fine-grain model in ascending order from left to right. Note that the total write latency of the fine-grain model tends to be larger than that of the coarse-grain because the former model takes 1,000 ns of tRCD for an ACT and another 1,000 ns of tRP for a write-back at a PRE, while the latter takes only 1,000 ns.

This graph shows the additional latency due to NVMM, and 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 NVMM, the number of ACT and PRE, and bank parallelism, are also measured. Memory requests between LLC and MIG are counted. Bank parallelism ($BANK\_PARA$) is defined as follows:

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 IV shows $BANK\_PARA$ and ACT/REQ for each program. The programs are sorted similar to that shown in Figure 3. 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 shows the fine-grain model attains a lower execution time in comparison with the coarse-grain model, and also prove that the fine-grain model can capture the effect of row-buffer access locality and the bank parallelism.

Although 510.parest_r is also an exception, its bank parallelism and row-buffer locality values are low. In the coarse-grain injection, read and write requests can be processed in parallel, and any one of read or write requests having a larger total number of requests can cause more impact on the total execution time than the ones with lesser number of requests. 510.parest_r has high read/write ratio (25.0), which is defined by dividing the number of read requests by write requests, to NVMM. The significantly high read-write ratio for the coarse grain model spoils the delay processing parallelism and results in a longer execution time than expected.

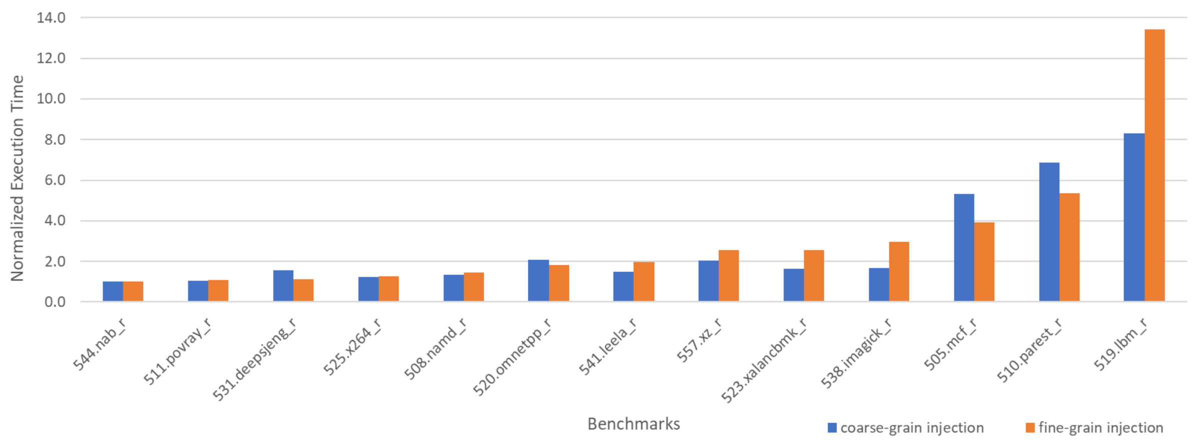

Fig. 3. Normalized Execution Time of SPEC CPU 2017 Programs

TABLE IV
$BANK\_PARA$ 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 V
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

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 IV 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 LLC and the frequency of memory requests to 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 V 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 of each program, because the cache hit ratio takes into account all memory requests 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 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, but ACT/REQ is another important factor. According to Table IV, 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 $ARL + AWL (= 2,000ns)$ in the fine-grain model.

C. Cache Flush Overhead

As described in Section III-F, the data in the cache must be evicted to NVMM to make it durable. We insert cache

TABLE VI
CACHE FLUSH OVERHEAD AND FLUSHED LINES

	Overhead [s]			
Benchmark	zero	coarse	fine	Total Flushed Lines
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

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 $\times$ 200,000). 557.xz_r allocates a large region and is an in-memory application. 519.lbm_r requires quite a high bandwidth. Table VI 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 additional execution time when both ARL and AWL are set to 1,000 with coarse-grain and 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 additional latency.

Regarding the amount of the overhead for each program, Table VI 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 the small number of 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.

V. 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 two types of delay injection models: coarse-grain and fine-grain. The fine-grain model can better capture the effects of the bank parallelism and the row-buffer access locality because it injects delay into the primitive memory requests issued by the memory controller. We also provided the cache flush software interface required for the persistent operations, as well as the standard C library compatible NVMM allocation functions for this environment.

The evaluation investigated the performance difference between two models by using a micro benchmark program and SPEC CPU 2017 benchmark. It also assessed the impact on the execution time due to bank parallelism, row-buffer access locality, and frequency of the NVMM requests. The evaluation results with 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.

VI. ACKNOWLEDGEMENTS

This work was partly executed under the cooperation of organization between Toshiba Memory Corporation and Waseda University.

REFERENCES

[1] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[3] 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.

[4] 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, Jul. 2015.

[5] J. Wang and B. Wang, "Pcmsim: A hybrid memory system simulator for the cloud storage," in *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, Aug. 2017, pp. 81–86.

[6] 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.

[7] T. Lee, D. Kim, H. Park, S. Yoo, and S. Lee, "Fpga-based prototyping systems for emerging memory technologies," in *2014 25nd IEEE International Symposium on Rapid System Prototyping*, Oct. 2014, pp. 115–120.

[8] T. Lee and S. Yoo, "An fpga-based platform for non volatile memory emulation," in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, Aug. 2017, pp. 1–4.

[9] 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. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814806

[10] A. Koshiba, T. Hirofuchi, S. Akiyama, R. Takano, and M. Namiki, "Towards write-back aware software emulator for non-volatile memory," in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, Aug. 2017, pp. 1–6.

[11] Xilinx. (2019) Xilinx/linux-xlnx: The official linux kernel from xilinx. [Online]. Available: https://github.com/Xilinx/linux-xlnx

[12] spec.org. (2019) Spec cpu(r) 2017. Standard Performance Evaluation Corporation. [Online]. Available: https://www.spec.org/cpu2017/