NVDIMM-C: A Byte-Addressable Non-Volatile Memory Module for Compatibility with Standard DDR Memory Interfaces

Changmin Lee*‡, Wonjae Shin*, Dae Jeong Kim*, Yongjun Yu*, Sung-Joon Kim*, Taekyeong Ko*, Deokho Seo*, Jongmin Park*, Kwanghee Lee*, Seongho Choi*, Namhyung Kim*, Vishak G*, Arun George*, Vishwas V*, Donghun Lee†, Kangwoo Choi†, Changbin Song†, Dohan Kim*, Insu Choi*, Ilgyu Jung*, Yong Ho Song*, and Jinman Han*
*Samsung Electronics, †SAP Labs Korea
Email: ‡c.m.lee@samsung.com

***Abstract*—Currently, there are two representative non-volatile dual in-line memory module (NVDIMM) interfaces: a proprietary Intel DDR-T and the JEDEC NVDIMM-P, which are not supported by existing platforms. Adoption of new platform is costly and measuring its efficiency of migrating to the new platform is much more complex. This study is an alternative way of them—finding a new memory device that can be supported by all existing systems. In this paper, we propose an NVDIMM architecture with several system-wide mechanisms to allow the synchronous DDR4 memory interfaces to support non-deterministic (asynchronous) timing. The proposed memory architecture is implemented as a real device prototype, and also evaluated using synthetic and real workloads on an x86-64 server system.**

***Keywords*-cache memory, nonvolatile memory, memory architecture, memory management, system software**

I. Introduction

Non-volatile memory (NVM) devices such as resistive random-access memory (ReRAM), spin-transfer torque magnetic random-access memory (STT-MRAM), phase-change random-access memory (PRAM), and ferroelectic random-access memory (FeRAM) are aggressively showing a lot of potential for storage-class memory (SCM). They provide competitive performance (in terms of latency [1]) and density with dynamic random-access memory (DRAM) and NAND flash memory, respectively. The persistent property also provides much opportunity and possibility to existing applications and infrastructures (e.g., reducing restart time of program or fast recovering from power failures). Furthermore, they are byte-addressable so that accessible through load/store instructions by CPUs. Thanks to those advantages, NVMs are targeting a wide range of areas, including in-memory database (IMDB), virtualization, cloud computing/services, high-performance computing (HPC), and storage cache.

Recently, Intel announced a persistent memory named DC persistent memory module (DCPMM) [2], which is compatible with the standard 288-pin double data rate 4 (DDR4) dual in-line memory module (DIMM) interface. DCPMM includes a media controller chip for 3D cross-point (3DX) media. The memory bus is directly connected to the media controller. An 8 Gb DDR4 DRAM component is also included in DCPMM to maintain address mappings between logical addresses of host requests and corresponding physical addresses of 3DX media. Since 3DX media is a PRAM-like device [3], which has lower write endurance and higher write latency than DRAM [4], [5], several background operations such as dynamic address translation, wear-leveling for endurance, and hiding write latency are necessarily required. Apart from the physical characteristics, those architectural workarounds make its latency non-deterministic.

DCPMM uses a proprietary DDR-T communication protocol that allows non-deterministic operations by re-using several DDR4 pins [6], [7]. For non-deterministic accesses, a dedicated DDR-T memory controller that implements a request/grant mechanism is integrated with the CPUs. Thus, DCPMM are not compatible with the current systems that are not having the DDR-T memory controllers. Most of NVM DIMMs, which have non-deterministic characteristics, will also require a new integrated memory controller (iMC) inside the CPUs, and significant modifications to the current system environments.

This work is inspired by the aforementioned aspect, i.e., *backward compatibility*, to build a non-volatile DIMM (NVDIMM) that supports not only existing systems but also the next-generation systems. In this paper, we present NVDIMM-C, an **NVDIMM** architecture for the **C**ompatibility with the standard DDR4 memory interfaces without any modification to the hardware and software environments. The NVDIMM-C architecture is currently based on high-density NAND flash devices (but, all kinds of NVM devices can be adopted) and high-speed DRAM as a cache of the NAND devices. The DRAM cache is the frontend of the proposed memory module; thus, NVDIMM-C acts as if it were a regular DRAM memory module. The host iMC and CPUs can directly access the proposed memory using the standard DDR4 commands and load/store instructions, respectively. Within NVDIMM-C, NVM controllers (NVMC) are also implemented to access the NAND devices. Interestingly, in our architecture, the DRAM cache can be accessed by both the host iMC and the NVDIMM-C internals (e.g., NVMCs), which means that bus conflicts on the memory channel can occur. To avoid the bus conflict, we introduce a novel mechanism that hides the accesses from the NVMCs under the refresh cycle

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00048

(tRFC), which is the time that guarantees no one can access the DRAM.

In this work, we implement a proof-of-concept (PoC) memory module to evaluate the proposed tRFC-based conflict avoidance mechanism on a real x86-64 server system. The experimental results show that the implemented PoC device achieves 1796 MB/s and 4615 MB/s for 4 KB random write accesses using a single thread and 16 threads, respectively, if the DRAM cache is fully hit. On misses of the DRAM cache, the 4 KB random write bandwidth is saturated to 99.7 MB/s using 4 threads. These results indicate that SCMs are difficult to be implemented on a NAND-based basis due to the low performance for small random accesses. We further investigate the minimum performance requirement of the back-end media that can serve as SCMs. We observe that the performance on the DRAM cache misses becomes 914 MB/s if the 4 KB access latency of the back-end media was 1.85 us or less, which can be covered by new memory technologies such as STT-MRAM and PRAM [1].

The contributions of this paper are summarized as follows:

- We introduce a new byte-addressable memory architecture that meets the objective of maintaining the backward compatibility with the existing systems.
- We describe a novel mechanism that allows multiple master chips (e.g., the host CPU and the device's media controller) to share the same memory bus without modifying the DDR4 interface and protocols.
- We present a detailed implementation of the proposed architecture including hardware and software stacks.
- We evaluate the proposed device on a real server system without any modification to the running environments. We find that the NVDIMM-C architecture can provide a balanced performance with low-latency NVM devices.

The rest of this paper is organized as follows. Section II briefly describes the DRAM refresh operation and the direct access feature in Linux. Section III describes our proposal to use a DRAM-as-frontend architecture for NVDIMMs. Section IV details the PoC device implemented in this work at the hardware and software levels, and Section V discusses several implications of the NVDIMM-C architecture. Section VI describes experimental methodology used in this study, and Section VII shows the experimental results for the PoC device. Related work is reviewed in Section VIII. Finally, we conclude this paper in Section IX.

II. Backgrounds

A. Direct Access Memory Management

Direct Access (DAX) is a mechanism for NVM devices and defined by Storage Networking Industry Association (SNIA). An NVM device that implements the DAX interface can be exposed as a byte-addressable device. Although the traditional mmap() approach allows the application to use pointer-based byte-addressable loads and stores, accesses to the memory-mapped file actually cause a 4KB page-sized block I/O through the traditional block and filesystem layers. However, the DAX mechanism directly uses mappings of the memory-mapping unit (MMU) to map a character device (device DAX, or devdax) or a file within the persistent memory-aware file system (filesystem DAX, or fsdax) to the virtual address space of user applications. Thus, an access to the devdax or fsdax region involves a page fault exception if the corresponding virtual-to-physical mapping is not residing in the MMU mappings (e.g., translation-lookaside buffer, or TLB).

The fault handler in the Linux kernel determines whether the faulting address belongs. If it belongs to the DAX address space, the default kernel fault handler passes the execution context into the page fault handler of DAX. A driver customized for its own device can also implement a fault handler by registering it in the kernel table. Whenever a page fault associated with that device occurs, the kernel finds and invokes the customized fault handler function.

B. DRAM Refresh and Programmability

The data cell of DRAM consists of a capacitor (typical value of cell capacity is more than 5fF [8]–[10]). Since the capacitor leaks its current over time, the data should be periodically restored by sense amplifier. When a row within DRAM is enabled (activated), the cell data is loaded to the sense amplifier through Bit lines. The sense amplifier develops the Bit lines to the supply voltage or 0V according to current levels of the cell data. After a certain period, the activated row is closed (precharged). A series of these operations is called refresh. Command/address buffer, data buffer, and internal I/O lines are disabled until the refresh is finished. Therefore, any request to DRAM cannot be valid during the refresh operation. This restricted time period is called the tRFC time.

The memory controller is responsible for issuing a refresh command periodically without a specific target row. Instead, an address counter within DRAM is used to calculate sequential target rows. The JEDEC specification recommends to issue 8K refresh commands within 64 milliseconds. The average refresh interval (tREFI) is 7.8us. For the purpose of thermal throttling, tREFI is variable. Since the leakage of cells is accelerated as the cell temperature increases, tREFI is adjusted to 3.9 us above 85°C. tRFC is also pre-defined according to the device (e.g., 260 ns for 4 Gb devices and 350 ns for 8 Gb devices). Both tRFC and tREFI parameters are dynamically adjusted by the iMC as well as they are programmable by the OS kernel. Most CPUs provide registers that configure functional logic in the processor. Note that these configuration registers are exposed as memory-mapped I/O and thus, accessible by the kernel space [11].

III. NVDIMM-C Architecture

A. Design Choice for the Compatibility

The integration of a variant of NVDIMM into the current DDR4-based systems requires significant changes to the entire software stack. For instance, the basic input/output system (BIOS) performs memory training at the boot time. Memory training is very complicated [12], [13] and it is responsible

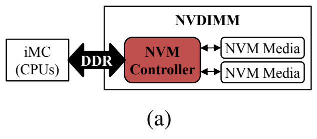
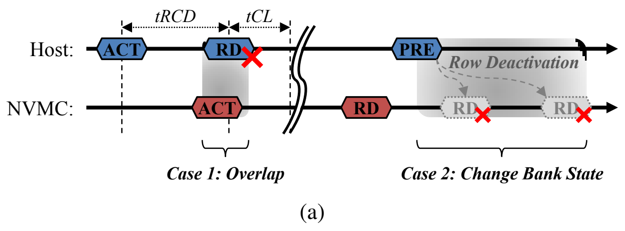


Fig. 1: Architectural design options of NVDIMM variants



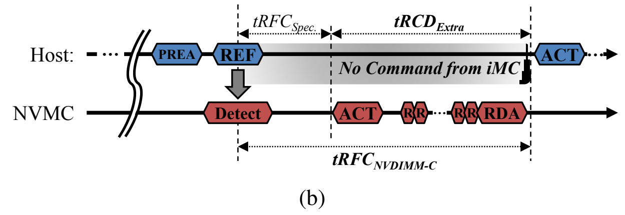


Fig. 2: DRAM bus usage examples: (a) two cases of bus collision and (b) commands serialized based on REFRESH

for determining information about memory specification, frequency, and signal timing of command/address (CA) and data (DQ) pins. Since this training process works with the iMC, which is dedicated to only DDR4, NVDIMM installed into the system has to follow the timing specification of DDR4 to successfully pass the training sequence.

Figure 1 shows two design options for NVDIMM architectures. First, NVMC can be placed at the frontend of the memory module as shown in Figure 1a; thus, it is directly connected to the DDR4 memory bus. Due to the timing constraints of DDR4, the NVMC has to act as a regular DRAM DIMM by responding to the host iMC commands according to the standard DDR4 specification. Loading the data from the back-end NVM media onto the DQ bus has to be performed within tRCD + tCL (e.g., 26.64 ns for DDR4-2400) after the NVMC receives an ACTIVATE command that is followed by a READ command from the iMC. Note that the Intel Xeon Skylake Scalable Processor exposes a 5b-wide configurable register for each DRAM timing parameter (i.e., tRCD or tCL) [11]; thus, the maximum *non-standard* value for each of the timing parameters is 5'b11111 (i.e., 51.615 ns for DDR4-2400). Currently, timing constraints of those levels (100 ns or less) are supported only by STT-MRAM, which has read and write speeds of tens of nanoseconds [1]. However, the maximum density of STT-MRAM for products in 2019 is 1 Gb [14], [15], which is still insufficient for SCM. Other NVM media such as PRAM (latency of hundreds of nanoseconds), ReRAM, and NAND (latency of tens of thousands of nanoseconds) are not applicable to the NVMC-as-frontend architecture due to the fact that reads and writes latencies are longer than those allowed by the iMC configurable timing registers. Obviously, this design option is available only if the latency of NVM media is competitive with DRAM or if both the iMC and NVMCs support an asynchronous handshake protocol (e.g., JEDEC NVDIMM-P and Intel DDR-T).

In this work, we choose to use DRAM as the frontend of the memory module as shown in Figure 1b. The CA and DQ buses are directly connected to the front-end DRAM. The entire NVDIMM is exposed to the system as a regular DRAM. Since the front-end DRAM is fully synchronous with the host iMC, there is no timing violation at the boot-up procedure while maintaining the existing environment including BIOS, timing parameters, and OS design. However, the back-end NVMC and NVM media are completely hidden so that operations and timings of the NVMC and NVM media are independent from the synchronous domain. Thus, any kind of NVM technologies can be used as the backend media regardless of the timing constraints of the host iMC. In this study, we use NAND flash as the NVM media due to its high density and low cost characteristics in addition to its well-established memory technology. With the DRAM-as-frontend architecture, the back-end NAND media is only accessible by the NVMC. The NVMC is similar to a conventional NAND controller with features of wear-leveling, garbage collection, and bad-block management and it also performs the primitive NAND operations (e.g., Read, Program, and Erase commands) with error correction code (ECC) at the granularity of 4KB.

The host iMC cannot access the NVM media since the NVMC and NVM media are physically separated from the DDR4 memory bus. Intuitively, the DRAM-as-frontend architecture can use the DRAM as a cache or buffer of the NVM media. When an application reads a memory address within the DRAM cache space, the data could be loaded into the processor's last-level cache (LLC) either immediately (i.e., DRAM cache hit) or in after the NVMC loads its associated data from the NVM media into the DRAM cache (i.e., DRAM cache miss). Those managements of the DRAM cache are performed by software (Section IV-B).

B. Serializing DDR Commands on Shared Bus

Depending on whether the accesses to the DRAM cache are hit or miss, the NVMC may access the DRAM cache. Thus, the NVMC must include a DDR4 controller in addition to NAND controllers. The DDR4 controller of the NVMC is configured to have the same DDR4 timing parameters with the host system. In addition, all the physical pins of the DRAM cache are routed and connected with the host iMC and the NVMC; that is, the memory channel of the DRAM is shared between them.

In computer design, such a case where more than one device accesses on a single shared channel at the same time mostly results in bus contention. The concept of the DRAM cache shared between the two bus masters (i.e., the iMC and the NVMC) is the most serious challenge in this work—

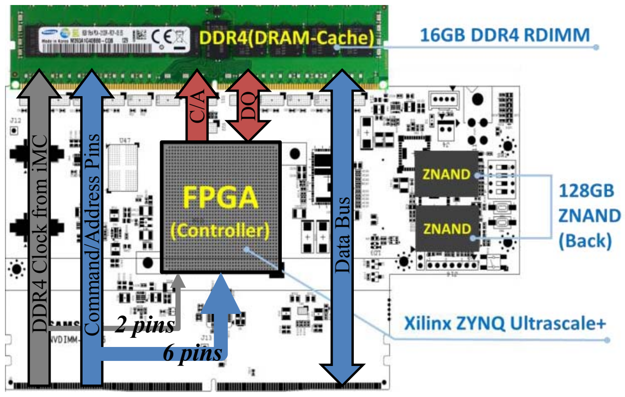


Fig. 3: Layout of the NVDIMM-C board and signal paths

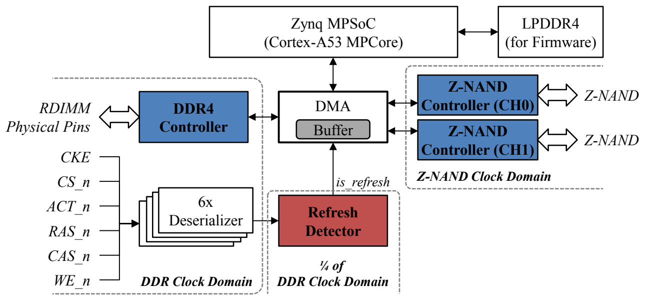


Fig. 4: Block diagram of NVMC RTL design

an arbitration scheme for preventing/avoiding contention can hardly be applicable to this architecture. The first reason is that we cannot predict memory access patterns of the host iMC since they depend on the dynamic behavior of workloads. Second, an arbitration protocol that buffers DDR commands issued by the iMC to delay their timings for several nanoseconds is also inappropriate because memory operations for the DDR commands must be performed in deterministic latencies. Moreover, there is no feedback/acknowledgement signal in the standard DDR4 interface[1]. In our architecture, an arbitration mechanism should be carefully designed to address the bus contention problem.

Figure 2a shows several cases where bus contention arises. In the case 1 (C1), the NVMC issues an Activate command for a DRAM row that is different from the row previously activated by the host iMC. At that time, the iMC may issue a Read command to the shared memory bus, thereby causing the conflict of the memory commands. This command conflict can possibly occur anytime even if the NVMC can monitor the iMC behavior through the CA bus. For C2, suppose that the two masters are accessing the same row of the DRAM. The NVMC tries to perform burst reads on the opened row. However, the iMC can possibly close (deactivate) that row by issuing Precharge and thus, the subsequent Read command becomes invalid, and also may cause an unexpected state or a critical memory error.

In this paper, we propose a collision-free mechanism for the DRAM-as-frontend NVDIMM architecture. Inspired by the fact that the iMC cannot issue any command during the DRAM refresh cycle (tRFC) time, the proposed conflict avoidance scheme exploits tRFC to allow the DRAM accesses from the NVMC. To do this, there are two requirements in hardware. First, the NVMC needs to monitor the shared CA bus so as to detect the Refresh commands issued by the iMC. Lastly, the tRFC time must be adjusted through BIOS or the iMC registers so that the NVMC can access the DRAM after the DRAM devices actually complete the refresh operations (the standard tRFC is 350 ns for an 8 Gb DDR4 device). As shown in Figure 2b, the NVMC waits until the Refresh command is issued by the host iMC. When detected, the NVMC first waits for the default tRFC time before accessing the DRAM during the extra tRFC time.

Note that the standard DDR4 specification does not support per-bank refresh that refreshes DRAM cells at the bank granularity; thus, DDR4 memory controllers are designed to precharge all opened banks (PREA as shown in Figure 2b) before issuing a REFRESH command. This requirement ensures that all banks of the DRAM cache are deactivated/closed before the extra tRFC time, and enables the NVMC to access all the banks during that time. That is, the NVMC can issue read and write requests to any part of the DRAM cache.

IV. Proof-of-Concept Device

A. *Hardware and RTL Design*

In this work, we implement the proposed DRAM-as-frontend architecture as a proof-of-concept (PoC) device. Figure 3 shows the printed circuit board (PCB) layout of our PoC device. It is designed as a 288-pin DDR4 DIMM and can be directly installed in the motherboard. The PoC device also integrates many integrated circuits including a Xilinx Zynq Ultrascale+ MPSoC [16], two 64GB Z-NAND devices (low-latency single-level cell NAND) [17], power management IC (PMIC) with battery-backed power source (on power failure), and I/O peripherals. Interestingly, a DDR4 DIMM socket is attached at the top of the board to allow a commodity unbuffered or registered DIMM (UDIMM or RDIMM) to be populated. The populated DIMM is easily replaceable, and will be used as the DRAM cache. All the DDR4 pins connected with the iMC are forwarded to the pins of the DRAM cache. Therefore, the iMC can access the DRAM cache normally.

Some of the DDR signals received from the host iMC are shared between the DRAM cache and the FPGA. For instance, additional wires for six of the CA signals are also routed and distributed into the FPGA package pins (i.e., pins CKE, CS_n, ACT_n, RAS_n, CAS_n, and WE_n), thereby allowing an register-transfer level (RTL) module in the FPGA to detect the Refresh commands by monitoring the CA bus. That RTL module makes a key contribution to address the bus contention problem (Section III-B). As shown in Figure 4, each of the CA

[1]The only exception is the ALERT_n pin, which is already dedicated to error reporting.

signals and the DDR4 differential clock, generated by the host, are input of the 1:8 deserializer that parallelizes the incoming signals by eight bits. This kind of serial-to-parallel converter is commonly used in high-speed communications to compensate the limited data rate of input/output (e.g., prefetch architecture in DDR DRAMs [18]). Assuming that the CA signals operate at DDR, the data of each CA signal is captured every four clock cycles so that the output of the deserializer is eight-bit wide. The refresh detector receives six 8-bit data per clock from the deserializers, and determine whether those parallel data includes the state of Refresh. Note that the refresh command can be defined by the state of the CA signals when pins CKE, ACT_n, and WE_n are H (High) while the other pins are L (Low). Those CA pins are sufficient for detecting Refresh since the CA states of all DDR4 commands are mutually exclusive. The variants of refresh commands such as self-refresh entry (SRE) and self-refresh exit (SRX) are defined by different states from the normal Refresh.

In this work, we use three Cortex-A53 cores of the Zynq MPSoC to run the firmware. One core is for the flash translation layer (FTL) that manages the two channel Z-NAND devices. Another core is responsible for orchestrating RTL modules at runtime by determining their execution states and/or configuring their internal registers. For example, when the refresh detector asserts the is_refresh port to H, that RTL management core may issue requests for data transfer from/to the DDR4 controller onto the direct memory access (DMA). Then, the DRAM will perform the requested transactions during the additional tRFC time, right after waiting for the default tRFC time. In this study, we configure the tRFC time to a thousand of device clock cycles (e.g., 1.25us at DDR4-1600) including the JEDEC requirement of 350ns and the subsequent additional tRFC time of 900ns. During the extra tRFC time, the DMA and DDR4 controllers, which are implemented in the FPGA, can perform up to 4KB data transfer from/to the DRAM cache. Note that in this paper we refer interchangeably to the DDR4 controller and the NVMC.

The JEDEC requirement defines that the average refresh interval (tREFI) in a normal state is 7.8us regardless of the device capacity. In our architecture, the accesses to the DRAM cache from the FPGA can be triggered only by the REFRESH commands. Thus, the DRAM latency and bandwidth on the FPGA are significantly limited by the tRFC and the tREFI. Section VII-D describes the impact of the refresh parameters on the performance.

B. Memory Mapping of NVM Space

In the Linux kernel, it is possible to mark a specific range of memory as reserved by using the kernel parameter memmap=nn$ss, where memory from ss to ss+nn-1 is excluded from normal usage [19]. Reserved memory regions are designed for the special usage by device drivers. Such memory regions can be allocated within the system address space either statically or dynamically.

In this work, we use the memmap parameter to mark the 16GB DRAM address space as a reserved region so that there are no accesses to the DRAM from applications and the OS. We also implement a device driver for NVDIMM-C (called nvdc). The nvdc driver allocates a block device of 128GB, as much the same capacity with the Z-NAND media, to the /dev directory (e.g., /dev/nvdcX where X is a device number). In addition, it implements a block device operation named device_access [20] for supporting fsdax. When an application accesses a block on our device nvdcX, the kernel layer of the DAX-aware filesystem (e.g., ext4 or xfs) calls the device_access function to retrieve a virtual address of that block.

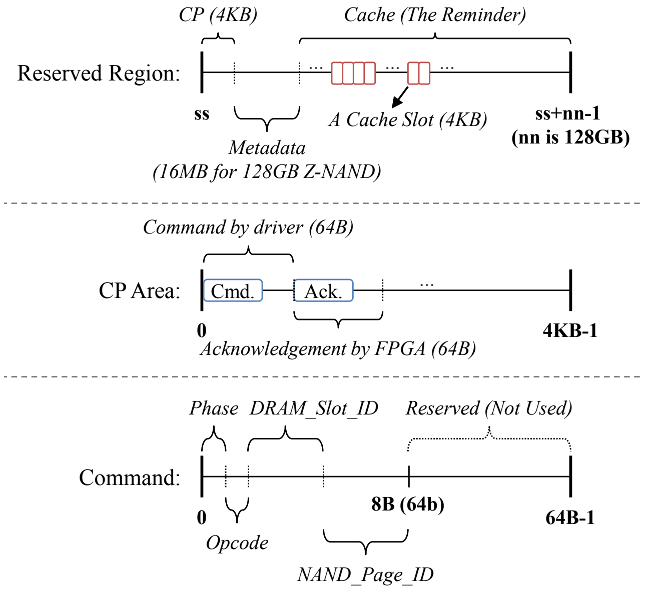


Fig. 5: Layout of the reserved memory region

To obtain this virtual address, the nvdc driver converts the block device sector (aligned to 512 bytes) number to the NAND page (4KB-aligned) number by assuming a direct mapping. Since the actual data corresponding to the converted NAND page number is essentially stored in the Z-NAND media, the nvdc driver initially performs a *cachefill* operation in order to load the data from Z-NAND into a free physical page (or a free slot) within the reserved DRAM region; that is, the DRAM address space is used as a cache. When the data is successfully loaded onto the free slot, the nvdc driver returns both the page frame number (PFN) of the free slot and its virtual address to the caller, i.e., the DAX-aware filesystem. With this virtual address, the application can normally access the block by using load/store instructions.

However, if there is no free slot in the DRAM cache, the nvdc driver finds a victim slot to be evicted from the DRAM cache if data of the selected victim slot is dirty. Our driver requests a *writeback* operation to the NVMC in order to store the corresponding physical page into the Z-NAND media. After the writeback task completes, the victim slot now becomes a free slot for the subsequent cachefill operation.

In this work, the selection of a victim cache slot is performed in a least-recently cached (LRC) policy. LRC information is managed during the lifetime of cache slots. When a physical page is *cached* in the DRAM cache, the nvdc driver stores the pointer to the associated PTE in a first-in, first-out (FIFO) manner. Thus, whenever eviction is needed (due to the DRAM cache in full), the first entry of the FIFO queue is selected as a victim. This LRC policy is possibly not optimal for various workloads from the performance perspective—caching/eviction of the same physical page may occur repeatedly—but, it is simple to implement. Note that the DRAM cache is managed as a fully associative cache with the cacheline granularity of 4KB so that data from the Z-NAND media can be stored in any cache slot.

C. Communication with NVM Controller

The reserved region is categorized into three types as shown in Figure 5. The first physical page of the reserved memory is used as a communication protocol (CP) area for communicating with the NVMC. The nvdc driver updates the CP area whenever it wants to request a command to the NVMC (in this work, multi-command is not supported). For instance, a command is 64b-wide data and stored in a single cacheline. Each command includes four bit-fields: *Phase*, *Opcode*, *DRAM_Slot_ID*, and *NAND_Page_ID*. The Phase field indicates whether the current CP data is new or not. The Opcode field indicates which operation to be performed (e.g., cachefill or writeback). The rest two fields respectively indicate the physical page number of the DRAM and the NAND page number that are associated with the current Opcode value.

Meanwhile, the DDR4 controller inside the FPGA always polls the CP area every tRFC time, and determines whether a new phase is written in the Phase field. If a new phase is detected, our PoC device performs the operation specified in the Opcode field. For cachefill operations, the FTL translates the NAND_Page_ID value to the NAND physical addresses, and issues the Read commands to the NVMC. When reading the 4KB data completes, the DDR4 controller writes it onto the physical address of the DRAM cache by referring to the DRAM_Slot_ID field. Note that those writes are only performed in the following tRFC time to avoid bus contention on the shared bus, as previously discussed in Section III-B.

For writeback operations, the DDR4 controller waits for the Refresh command in order to read the victim page of the DRAM cache, which is specified by the DRAM_Slot_ID field. After the refresh command is detected by the refresh detector, the 4KB data can be fetched from the DRAM, and then stored into the Z-NAND media. In addition, after issuing a command, the nvdc driver polls the acknowledgment region in the CP area, which stores a message from the FPGA. Whenever a received command is finished, the FPGA replies to the nvdc driver's request by updating its status onto the acknowledgment region.

As previously discussed, the nvdc maintains a mapping between filesystem block numbers and virtual addresses of the reserved region. To do this, a mapping between the NAND

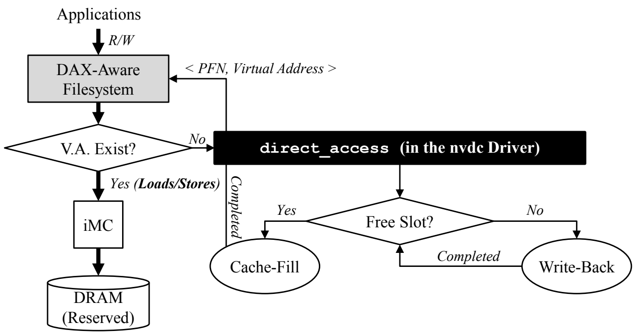


Fig. 6: Flow diagram of the NVDIMM-C device driver

page numbers and the virtual addresses of the DRAM cache slots. All these mappings are stored in the metadata area, shown in Figure 5, and managed at runtime.

V. Architectural Characteristics

A. Runtime Performance Characteristic

The performance of our device (in terms of latency and bandwidth) is nearly the same as the DRAM performance only if the physical page corresponding to the blocks that are being accessed by the application is already cached in the DRAM. As shown in Figure 6, since the virtual addresses of the physical page is maintained by the DAX-aware filesystem layer, i.e., the PTE is up-to-date and valid, the iMC can immediately access the DRAM cache by following the existing paging mechanism, which consists of translation look-aside buffers (TLBs) and a page walk (if needed).

However, if a physical page of the blocks is missing in the DRAM cache, there are two cases that show different performance from each other. When a free slot is available, the nvdc driver performs a cachefill operation before it returns the PFN and the virtual address of the newly allocated slot to the filesystem. Note that three tREFI windows are required to finish the cachefill operation because a refresh command is needed for each step; (1) polling the CP area to retrieve the new command data; (2) writing the NAND data into the DRAM cache; and (3) updating the completion status into the CP area. As a result, the cachefill operation takes at least 23.4us (3x tREFI of 7.8us). This estimation is the minimum theoretical latency of our device to load the missing physical page into the DRAM cache, while the read latency of Z-NAND is completely excluded. When there is no free slot in the DRAM cache (i.e., the DRAM cache is full), a writeback operation is additionally required before performing the cachefill operation. In this case, the minimum latency is now doubled to a total of 46.8us (6x tREFI).

In addition, the proposed tRFC-based bus utilization mechanism limits the DRAM bandwidth for the FPGA. Up to 4KB DRAM data can be accessed by the FPGA during the tRFC time, so the DRAM performance can be achieved up to 500.8MB/s with the default 7.8us tREFI. With tREFI at the double frequency (i.e., 3.8us), the maximum bandwidth is doubled to 1001.6MB/s. Increasing the tRFC time will improve the FPGA performance at the cost of DRAM performance for the CPU side. On the Intel Skylake platforms, the tRFC time is configurable for each memory channel. Only the DRAM populated in the same channel with NVDIMM-C will be negatively affected by the increased tRFC time. The DRAM performance for other memory channels will not experience performance degradation. Section VII-D describes performance tradeoffs between the iMC and the FPGA in detail.

Note that the Intel DCPMM also shares a memory channel and therefore, DCPMM performance negatively impacts on the performance of the DRAM in the same channel. All kinds of NVDIMMs will have a similar performance tradeoff since they share the memory channels with DRAMs.

B. Cache Coherence Maintenance

Typically, the contents of the caches inside the CPUs are coherent with the main memory. With the tRFC-based architecture, however, the cache consistency may not be maintained properly because the data transfer performed by the FPGA during the tRFC time is invisible to the cache and uncore hardware (e.g., iMC, interconnects, etc.).

The cache incoherence happens when the FPGA updates the contents of the DRAM cache (e.g., cachefill) that has been already cached by the CPU cache subsystem. The data in the CPU cache will be an older one. Later, this old data could be flushed, and written back into the DRAM cache, thereby overwriting the new data updated by the FPGA. The problem also occurs when the CPU changes its cache data. The new value in the CPU cache is now different from the value in the DRAM cache until the CPU cache data is flushed. If the nvdc driver performs the writeback operation for the cache slot associated with the new cacheline, the stale data will be read and stored into the Z-NAND media by the FPGA. In addition, a similar problem occurs when polling and updating the CP area. Both the nvdc driver and the FPGA must carefully consider the cache incoherence problem.

The nvdc driver explicitly flushes the CPU cache (e.g., clflush) and performs the memory fence instruction (e.g., sfence) before requesting the writeback operations to the CP area. These instructions force the ordering of writes so that the FPGA can read the up-to-date memory data in the subsequent tRFC time. For the cachefill operations, the nvdc driver invalidates the corresponding cachelines of the CPU cache after completing the cachefill operations. Thus, the contents of the CPU cache are explicitly coherent with respect to the DRAM cache without any data corruption. Addressing those inconsistency problems are similar to implementing DMA support codes [21], [22] regarding accesses to DMA buffers from CPUs and I/O devices.

C. Persistency Support

The systems that support persistent memory typically define the persistence domain; for instance, the Intel platforms ensure that any stores in the write pending queue (WPQ) inside the memory controller are flushed to NVM media on power failure [23]. Therefore, executing the cache flush instructions will be enough to make data persistent.

With the DRAM-as-frontend architecture, the additional memory layer (i.e., the DRAM cache) between the iMC and the NVM media limits our device to inheriting the persistence domain on power failure. On power loss, the firmware running on the FPGA reads the DRAM-to-NAND mappings stored in the 16MB metadata area of the DRAM (Section IV-C) while ignoring the tRFC-based serialization rule (Section III-B). Therefore, the valid physical pages inside the DRAM cache can be stored into the persistent Z-NAND media. Unfortunately, this process associated with the DRAM cache and the platform-level operation run in parallel; thus, several writes within the WPQ may not be flushed into the DRAM cache until the FPGA reads the corresponding physical pages. The precise persistence domain with our device will be scaled down to the DRAM cache, while the existing WPQ resource possibly becomes a weak persistence domain. To fully inherit the platform's persistence domain, further investigation such as detecting asynchronous DRAM refresh (ADR) or using flush hint addresses of the NVDIMM firmware interface table (NFIT) [24] is needed in the future. Note that the nvdc driver does not implement devdax, so direct manipulation of persistency from user applications is currently not supported.

TABLE I: Test system configuration

Hardware	Description
CPU	1 x Intel Xeon Platinum 8168 (Skylake-SP)
Platform	Intel Server Board S2600WF
Main Memory	2 x 128 GB DDR4 RDIMMs @1600 Mbps (tRFC: default, i.e., 350 ns)
Baseline (/dev/pmem0)	1 x 128 GB DDR4 RDIMM @1600 Mbps (tRFC: 1250 ns / mounted as XFS-dax)
NVDIMM-C (/dev/nvdc0)	1 x 128 GB NVDIMM-C @1600 Mbps with 16 GB DRAM and 2 x 64 GB Z-NANDs (tRFC: 1250 ns / mounted as XFS-dax)
Storage	PM863 1.92 TB SATA SSD (mounted as XFS) (Sequential read/write max speed: 520/475 MB/s)
OS	SUSE Linux Enterprise Server 12 SP3 (Linux kernel version 4.4.73)

VI. Experimental Methodology

For evaluation, we used the test system that consists of the 24-core Intel Xeon Platinum 8168 processor, two 128 GB RDIMMs, and the 128 GB NVDIMM-C PoC memory module. The address space of the NVDIMM-C device was reserved by the memmap parameter and thus the two RDIMMs are only served as the main memory. Due to the vertical length of the PoC module (i.e., longer than that of a regular RDIMM), the maximum frequency supported by the system is limited to 1600 Mbps. Although the Z-NAND capacity is a total of 128 GB, the FTL firmware implemented in this work uses 120 GB only. As a result, the block device exposed by the nvdc driver to the OS will have the same capacity of 120 GB. All experiments were performed on the SUSE Linux Enterprise Server 12.3. Table I summarizes the configuration of the testing system.

TABLE II: Used benchmarks and performance metrics

Benchmark	Used Metrics
FIO v3.10 [25]	Latency, bandwidth
TPC-H [26] on SAP HANA IMDB [27]	Query transaction time
In-House Mixed-Load IMDB	Number of allowed concurrent users, query validation

For workloads, we selected the flexible I/O tester (FIO) [25] as microbenchmark to measure primitive performance results of NVDIMM-C such as latency and bandwidth. For enterprise-level scenarios, we used the TPC-H benchmark [26] with the Scale Factor (SF) of 100, which represents a test database of 100 GB. The TPC-H transactions were performed on SAP HANA IMDB engine [27], which is the most popular in-memory platform that integrates software and hardware technologies to make a significant value of Big Data. HANA has already adopted NVM supports to store the main database data in persistent media. In addition, for mixed-load IMDB workloads, we used the SAP in-house benchmark that measures the number of concurrent users that can work simultaneously or concurrently in the system. This benchmark is also useful for validating data integrity and consistency during database transactions. All the benchmarks used in this paper are summarized in Table II.

In this paper, we compare the results of our device with the emulated NVDIMM [28], which is integrated in the Linux kernel v4.2 or later for the testing purpose (it can be seen as /dev/pmem0 in the OS). The NVDIMM emulation device uses the DRAMs as the back-end media (like a ramdisk); thus, it actually does not guarantee the persistency property. For the emulation environment, we used an additional 128 GB RDIMM by replacing the NVDIMM-C module. The capacity of the main memory was still maintained at the same amount as the NVDIMM-C configuration. From now on, we denote the emulated NVDIMM device as Baseline. All the NVDIMMs (NVDIMM-C and the baseline) are mounted as the XFS filesystem with the DAX option (e.g., mount -o dax).

VII. Experimental Results

A. Validation of Refresh Detection Accuracy

The accuracy of the refresh detection is significant in our architecture. As discussed in Section III-B, the FPGA only accesses the DRAM cache during the tRFC time to avoid the bus contention. If detecting the refresh commands is not performed accurately, a case where even if REFRESH is not issued by the host iMC, its detection is returned as true can occur. In those cases, DDR commands issued by the iMC will be conflicted with the NVMC accesses, thereby causing a system failure, which is critical from the reliability perspective.

In this study, we cannot quantify the accuracy of the refresh detector despite of many efforts to improve the refresh detector's performance (e.g., optimizing electrical characteristics for creating consistent impedance, using terminations, reducing cross talk). Instead, we evaluate its accuracy by aging tests. We ran the STREAM [29] benchmark intensively on all the CPU cores for the DRAM cache area. The STREAM benchmark was modified to compare the results with the reference data every iteration. The refresh detector is always enabled such that the FPGA accesses behind the tRFC time happen every REFRESH command. By doing so, we observed that the result comparison did not report any inconsistency and no system fault like memory errors occurred. We believe that the feasibility and practicalities of the tRFC-based command serialization are faithfully validated.

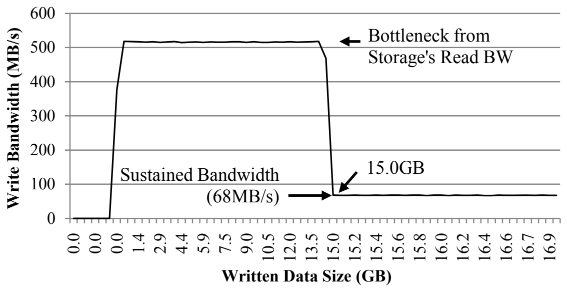


Fig. 7: Throughput of our device for a file copy workload

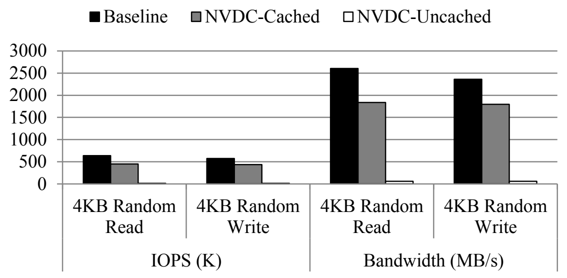


Fig. 8: 4KB random read/write performance

B. Performance of FPGA-Based PoC Device

1) Simple File Copy Workload: Figure 7 shows the sequential write bandwidth of the PoC device. In this experiment, we copied a 20 GB file from the SSD storage to our block device (i.e., a mounted directory of /dev/nvdc0), and measured the real-time bandwidth. When using the 16 GB RDIMM as the DRAM cache, the nvdc driver internally allocates 15 GB for cache slots. Thanks to those free cache slots, the PoC device achieves the peak sequential write bandwidth of 518 MB/s. From now on, we denote the performance in those (cache-hit) cases as the Cached performance. Note that the Cached performance reported in this section is not the best case, but is limited by the sequential read throughput of the solid-state drive (SSD) storage. The write bandwidth of our device is sustained at 68 MB/s, after the size of the written data reaches the DRAM cache boundary where there is no free slot. For those cases, a pair of writeback and cachefill operations is necessary for every 4 KB writes to the device (as previously discussed in Section V-A). From now on, we denote the performance in those cases as the Uncached performance.

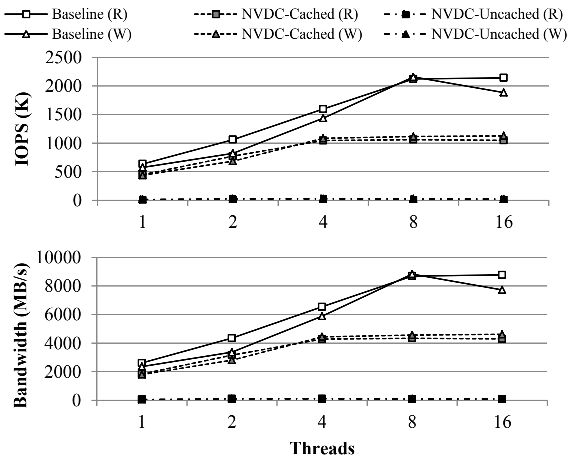


Fig. 9: Performance for 4KB random reads (R) or writes (W) with varying processor thread count

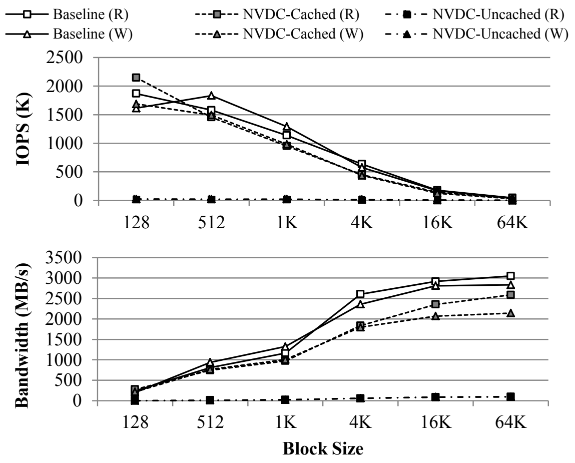


Fig. 10: Performance for 4KB random reads (R) or writes (W) with varying access granularity

2) Synthetic Workloads: In this section, we measure the performance of our device under several conditions (random reads and random writes) as compared to the baseline performance. The performance comparisons are performed using FIO by fixing the access granularity to 4 KB and both the iodepth and thread count are set to one. The ioengine for FIO is also fixed to libpmem to support the DAX-aware filesystem (i.e., bypassing the page cache). As shown in Figure 8, for the case of 4 KB random reads, the baseline provides 646 K input/output operations per second (IOPS) and 2606 MB/s. For 4 KB random writes, the average performance of 576 KIOPS and 2360 MB/s is achieved. We consider those baseline performances as the upper bound of our device performance since the baseline only uses the DRAM as the back-end media.

When the I/O operations are performed within the 16 GB DRAM cache boundary (NVDC-Cached), the NVDIMM-C device provides an average of 448 KIOPS and 1835 MB/s for 4 KB random reads. For 4 KB random writes, we observe 438 KIOPS and 1796 MB/s. Those results are 70-76% of the baseline performance, which corresponds to 24-30% overheads of the nvdc driver due to the explicit cache coherence (clflush and sfence) and page mapping managements. For the case of the Uncached scenarios, where the NVM accesses are involved, our device shows 13 KIOPS and 57.3 MB/s for the reads, and 14.2 KIOPS and 58.3 MB/s for the writes, which would be insufficient for SCMs.

In details, the average read bandwidth of 57.3 MB/s corresponds to 4 KB reads per 69.8 us. As previously discussed in Section V-A, this indicates that consecutive writeback and cachefill operations, which are caused by 4 KB reads to the Uncached page, take 69.8 us (i.e., 8.9x tREFI windows). Considering that either a writeback or cachefill operation requires 3x tREFI (23.4 us) by assuming that the access speed to the NVM media is near zero, the FPGA side of our device is 2.9x slower than the theoretically achievable maximum performance.

There are two ways to mitigate those poor performance results shown in the Uncached scenario; (1) using a much low-latency material such as STT-MRAM or PRAM as the back-end NVM media, and (2) issuing multiple commands to the CP area to allow the FPGA side to process multiple writeback/cachefill operations concurrently. In this work, we leave the exploration of such optimizations to future work.

3) Thread Count and IO Depth: In this section, different thread counts are compared for the random access patterns. We set iodepth to the same value as the thread count. As the thread count increases, we can find the point where the memory performance becomes saturated. Figure 9 shows the performance results with varying the thread count from one to 16. The ramdisk-like baseline scales with thread count, achieving the maximum performance of 2123 KIOPS and 8694 MB/s at eight threads. The NVDC-Cached case also stops scaling at eight threads (a peak of 1060 KIOPS and 4341 MB/s for reads), but it scales at a lower rate than the baseline. For writes, NVDC-Cached provides 1127 KIOPS and 4615 MB/s at 16 threads. For the case of Uncached, our PoC device is saturated at four threads, achieving the 4 KB read performance of 24.3 KIOPS and 99.7 MB/s on average.

4) Block Access Granularity: Figure 10 shows the I/O and bandwidth results with varying the granularity of device accesses. In these experiments, we fix the thread count and queue depths to one. Note that investigating performance with different access sizes is meaningful for application developers to fully utilize the memory device bandwidth. In the case of Cached, the FPGA-based NVDIMM-C device achieves 2147 KIOPS for the 128 bytes random reads, which is a speedup of 1.15x as compared to the baseline. Our device can scale to the performance of 3050 MB/s for the 64 KB access granularity using one thread. Using eight threads (omitted in the figure), the nvdc provides up to 10.9 MIOPS for the small accesses (128 bytes). These state that if the memory footprint is in a range of the DRAM cache size, the NVDIMM-C ar-

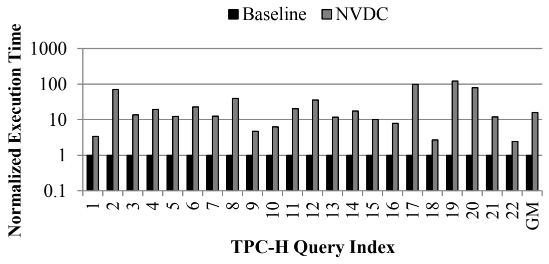


Fig. 11: Execution time of TPC-H queries running on HANA

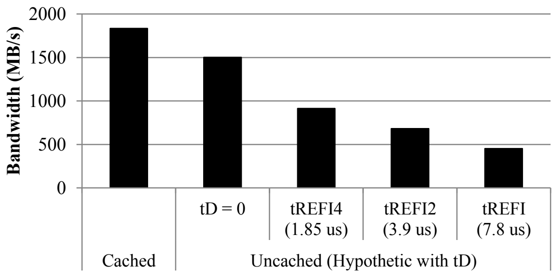


Fig. 12: Uncached performance with different tREFIs

chitecture will provide benefits to latency-sensitive workloads. In addition, there is a large bandwidth gap between 1 KB and 4 KB block sizes, illustrating that 4 KB or larger access granularities are preferred for throughput-oriented workloads in the NVDIMM-C architecture. This is because the nvdc driver always manages the mappings of Z-NAND and DRAM pages at the granularity of 4 KB.

5) SAP HANA IMDB Engine: In this section, we run the TPC-H benchmark with the HANA IMDB. Figure 11 shows the performance data for 22 TPC-H queries. For each query, the performance of our device is normalized to that of the baseline. The query 1 (Q1) performs a sequential table scan, so with increase in bandwidth of the device this query can become compute-bound. For Q1, the proposed device achieves 3.3x slower than the baseline. For Q20, which results in many small accesses [30], the execution time for our PoC device is 78x larger than that of the baseline. This performance gap between the baseline and our device are greater than the one seen in Section VII-B2. This large performance difference is due to inefficient use of the DRAM cache. As previously discussed in Section IV-B, we currently use the LRC cache slot replacement policy, which may cause frequent eviction of pages. According to our in-house simulation, for the TPC-H workloads, we observed that if a least-recently used (LRU) cache replacement policy is used, the DRAM cache hit rate of 78.7-99.3% can be achieved as the DRAM cache size is increased from 1 GB to 16 GB.

In addition, we also run a mixed-load benchmark, which is an in-house benchmark from SAP, to evaluate the data integrity of the memory device. This benchmark always performs data validation whenever a series of transactions are completed. In this experiment, we observed that five hundreds of user workload can be executed concurrently on our device without any data corruption.

C. Discussion: Mitigating Uncached Performance Slowdown

The results presented in previous sections show poor Uncached performance from a productization viewpoint. Note that this work has the purpose in validating the feasibility and applicability of exploiting the refresh commands to find a way for data transfer between the host and the DIMMs. The main reasons of the unsatisfactory performance are summarized as follows. The FPGA-based device is implemented to determine the finite state machine (FSM) of the RTL modules by the software running on the Cortex-A53 cores.

For example, the DDR4 controller is controlled by several software routines that are implemented in C/C++. Decoding the command in the CP area for the FPGA side is also performed by the software. Moreover, the device driver and the FPGA currently communicate with each other by using a single command only (i.e., queue depth of one) within the 4 KB CP area, even though each command needs 64-bit wide (so, it does not use the remaining memory space of 4 KB). The RTLs for the NVMC and the physical layer of the Z-NAND devices currently operate at a low clock frequency of 50 MHz, which is a tenfold of the maximum operating frequency supported by the Z-NAND devices. Those make data movements and FSM transitions so laggy.

In the future, those unoptimized factors would be mitigated by an application-specific integrated circuit (ASIC) implementation. (1) Eliminating the CPU-controlled data paths, (2) enabling multiple commands at a time for the CP region, (3) increasing the total amount of data transferred during tRFC up to 8 KB, (4) merging a writeback and cachefill operations that are independent from each other into a single command, so they are processed by the device in parallel, and (5) increasing the Z-NAND operation frequency (or using other kinds of NVM such as STT-MRAM or PRAM), will significantly improve the device performance. Further optimization on page mapping managements [31]–[34], eviction search algorithms [35], [36], or prefetch-based NVM accesses [37] could be also adopted for high performance.

D. Performance of Hypothetical NVDIMM-C Device

1) Impact of tREFI on NVDIMM-C Performance: As previously discussed, the Uncached performance of the proposed tRFC-based scheme depends on the frequency of the refresh commands. As the refresh rate increases, the FPGA has more opportunities to access the DRAM cache. To present the impact of the tREFI parameter on the Uncached performance, we now assume a hypothetical NVDIMM-C device that replaces the NVM access with a programmable time delay (denoted as *tD*); thus, the FPGA does nothing. For instance, when *tD* is set to zero, the Uncached performance for the hypothetical device will be similar to the Cached performance. To do this, we modified the nvdc driver to bypass the communication with the FPGA. Instead of writeback/cachefill

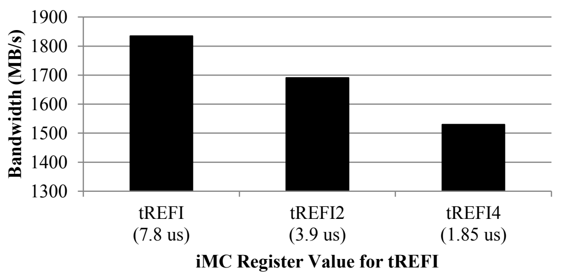


Fig. 13: Impact of tREFI on the host-side DRAM performance

the 4 KB page data to/from the NVM media, the modified nvdc driver will wait three times for the programmable delay because a single Uncached access requires three refresh windows when using the CP command depth of 1. In this experiment, we use different *tD* values of 7.8, 3.9, or 1.85, each of which represents the normal refresh rate (tREFI), the doubled refresh rate (tREFI2), or the quadrupled refresh rate (tREFI4), respectively. Note that, referring to the data shown in Section VII-B2, the FPGA-based PoC device currently requires 8.9x tREFI time for the 4 KB Uncached access.

Figure 12 shows the Uncached performance for the hypothetical device with the different *tD* values. For comparisons, the Cached performance of 1835 MB/s, which is the data presented in Section VII-B2, is also shown. When using *tD* = 0, the 4 KB random read performance of 1503 MB/s occurs, which is 82% of the Cached performance. Since in this experiment the explicit cache coherence management is excluded from the nvdc driver, the performance degradation of 18% can be considered as the software overhead of the page mapping management. For the case where the 4 KB access latency of the NVM media is less than 7.8 us, the Uncached bandwidth is 451 MB/s. If the NVM media could have the access latency of 3.9 us or below, the Uncached performance becomes 681 MB/s. For the case of tREFI4, the 4 KB random read performance will be 914 MB/s. Note that we still assume that the number of concurrent commands allowed within the CP area is only 1. We believe that allowing concurrent commands in the CP area will significantly increase the performance. Further investigation into the impact of the concurrent commands is needed.

2) Impact on Host-Side DRAM Performance: Unlike the relationship between tREFI and the FPGA-side DRAM performance, using the refresh interval at a finer granularity negatively impacts on the DRAM performance for the CPU-side accesses since a larger portion of the DRAM bandwidth is used for the refresh commands. We assume that the DRAM performance for the CPU accesses is similar to the Cached performance since the Cached pages can be directly accessed by load/store instructions. Figure 13 shows the Cached bandwidth for the 4 KB random reads when using the thread count of 1. For the normal refresh rate, we achieve an average of 1835 MB/s, as also shown in Figure 8. When the refresh rate is doubled to tREFI2, the DRAM performance decreases only by 8% (1691 MB/s) compared to the normal refresh rate. The extreme refresh rate, i.e., tREFI4, shows 1530 MB/s (a 17% reduction over the normal rate tREFI of 7.8 us). Referring to the data presented in Figure 12, tREFI4 will provide 914 MB/s for the FPGA-side performance while maintaining 83% of the host-side DRAM bandwidth. Using 16 threads, the peak of the Cached performance for 4 KB reads is 3690 MB/s even at the tREFI4 refresh rate. This tradeoff shows a balanced performance for the purpose of storage-class memory.

VIII. Related Work

The implications of NVM devices have been explored from various points of view [38], [39]. Thanks to its high-density property, prior work proposed to use NVM devices as a replacement for static random-access memory (SRAM) caches [40]–[42] or for the main memory [43]–[47]. Several approaches presented a hybrid architecture that exploits NVMs to build high-performance local storage [48]–[50] or network storage [51]. In addition, a high-density DRAM-only solution [52] and pushing a processing-in-memory feature into NVM devices [53] were also discovered.

To adopt NVM technologies for such use cases, modifications to the hardware, OS, and applications—maintaining different memory consistency models from the DRAM-based architecture [54], [55], recovering from system crashes [56], [57], exposing application programming interfaces (API)s to manage the persistency with architectural supports [58]–[63] or algorithmic optimizations [64]–[66], or providing a fair load balancing between DRAM and NVM commands issued by the memory controller [67]—have been carefully considered. Furthermore, prior work proposed the concept of hybrid main memory consisting of DRAMs and NVMs. To address long latency and low endurance of NVMs, memory architectures that uses DRAM as a cache of NVM to store frequently accessed data were introduced [68]–[72]. However, all these studies require a dedicated NVM memory controller, which is completely different from our proposal. Our study presented in this paper does not need NVM controllers inside the CPUs.

In addition, a mobile storage of the low-power DDR3 (LPDDR3) memory interface was proposed [73]. That storage device acts as a DRAM device with a small SRAM buffer that is shared between the host CPUs and the storage controller. This storage architecture is similar to our proposal, but different from the fact that the arbitration for accessing the shared buffer is performed by giving higher priority to the CPU accesses, instead of using the tRFC latency, so the accesses from the storage controller can be preempted anytime.

From industry, different types of the JEDEC NVDIMM standard were proposed: NVDIMM-N, NVDIMM-F, and NVDIMM-P [74]. NVDIMM-N is basically the same as a conventional DIMM except NAND flash devices for persistent backup, and thus the CPU can access it with load/store instructions. At a power failure, the data within the DRAM devices is completely copied to the NAND devices. Its power is supplied by super-capacitors. The data restoration from NAND is performed at the boot time. NVDIMM-F consists

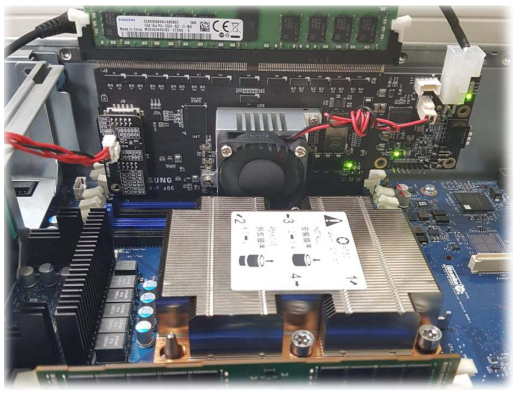


Fig. 14: The FPGA-based PoC device running on the system

of NAND devices and NAND controllers without DRAM components, thereby providing much larger DIMM capacity, but performance such as access latency and bandwidth is significantly limited. In addition, it is not a byte-addressable device (supports block access only). NVDIMM-P allows a combination of commercially successful two devices, i.e., DRAM and NAND devices. NAND devices can be replaced by any kind of NVM devices such as (but not limited to) PRAM, MRAM, and ReRAM. This hybrid memory device needs a dedicated memory controller that supports non-deterministic memory access, unlike commodity DDR DRAMs.

Netlist acquired a patent describing a mechanism that allows data movement from DRAM to multiple masters within a DIMM [75]. This mechanism requires a multi-ranked DIMM so that one rank is used for data and another rank is unused. To access the data rank by the DIMM controller, the device driver issues dummy writes to the unused rank while the DIMM controller simultaneously accesses the data rank. Thus, the actual DRAM capacity would be half of the total size. However, our device can fully utilize the entire DRAM area and employs the tRFC time for the data transfer. This dummy-access technique is also applicable to our architecture.

Intel DCPMM provides two operating modes. When configured for App Direct Mode, the applications and operating system are aware of the persistent memory. To access DCPMM, several dedicated software stacks must be involved. When configured for Memory Mode, the main memory is used as a cache of DCPMM. Cache managements are performed by the hardware, i.e., the DDR4 and DDR-T memory controllers, transparently to the entire software stack. The NVDIMM-C architecture proposed in this paper is similar to the Memory Mode within a single device. In addition, the proposed NVDIMM-C architecture is fully supported by existing DDR4 memory controllers.

IX. Conclusion

In this paper, we proposed NVDIMM-C, the DRAM-as-frontend memory architecture, to meet the goal of maintaining the backward compatibility with the existing DDR4-based systems. We presented the way of exploiting the tRFC time to arbitrate the concurrent accesses to the shared memory channel, thereby providing non-deterministic timing operations on the synchronous interface. The hardware and software requirements to support data movements on the shared bus were also described in detail. Furthermore, we showed the feasibility of the proposed concept of NVDIMM-C by building the FPGA-based implementation (as shown in Figure 14), and evaluated it using synthetic and real workloads on the real x86-64 server platform. The experimental results illustrate that the NVDIMM-C architecture should provide a balanced performance when using NVM devices that have a latency of less than 1.85 us for 4 KB random accesses. More importantly, we believe that the proposed concept to hide data transfer under the refresh commands is very useful and offers a powerful opportunity to new memory technologies that necessarily require direct communications between the CPUs and memory devices, e.g., processing-in-memory devices or DIMM-type accelerators.

Acknowledgment

The authors would like to thank all the anonymous reviewers for their valuable feedbacks. This work is performed in part by the SAP Co-Innovation Project and the Samsung-SAP Research Center (SSRC). We thank to Minseon Ahn and Jungmin Kim at SAP for insightful discussions and their dedicated supports. Corresponding author is Changmin Lee at Samsung Electronics. Note that the claims, directions, and statements noted in this paper are not representative of, and are irrelevant to, the companies' policies and directions.

References

[1] S. Bertolazzi, "Mram technology and market trends," in *Flash Memory Summit*, 2019.
[2] I. Cutress and B. Tallis, "Intel launches optane dimms up to 512gb: Apache pass is here!" 2018.
[3] J. Choe, "Xpoint memory comparison: Process & architecture," 2017.
[4] B. Gervasi, "Memory class storage," in *Flash Memory Summit*, 2018.
[5] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, 2010.
[6] Samsung, "Ddr4 sdram specification."
[7] B. Beeler, "Intel optane dc persistent memory module (pmm)," 2019.
[8] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, Jul. 2010.
[9] M. A. Siddiqi, *Dynamic RAM: Technology Advancements*, 1st ed. CRC Press, Dec. 2017.
[10] J. Choe, "Samsung 18 nm dram cell integration: Qpt and higher uniformed capacitor high-k dielectrics," 2017.
[11] Intel, "Intel® xeon® processor scalable family datasheet," 2019.
[12] S. Sethuraman, A. Lingambudi, K. Wright, A. Saurabh, K. Kim, and D. Becker, "Vref optimization in ddr4 rdimms for improved timing margins," in *EDAPS*, 2014.
[13] M. A. Islam, M. Y. Arafath, and M. J. Hasan, "Design of ddr4 sdram controller," in *ICECE*, 2014.
[14] S. Aggarwal, H. Almasi, M. DeHerrera, B. Hughes, S. Ikegawa, J. Janesky, H. K. Lee, H. Lu, F. Mancoff, K. Nagel, G. Shimon, J. Sun, T. Andre, and S. Alam, "Demonstration of a reliable 1 gb standalone spin-transfer torque mram for industrial applications," in *IEDM*, 2019.
[15] K. Lee, J. H. Bak, Y. J. Kim, C. K. Kim, A. Antonyan, D. H. Chang, S. H. Hwang, G. W. Lee, N. Y. Ji, W. J. Kim, J. H. Lee, B. J. Bae, J. H. Park, I. H. Kim, B. Y. Seo, S. H. Han, Y. Ji, H. T. Jung, S. O. Park, O. I. Kwon, J. W. Kye, Y. D. Kim, S. W. Pae, Y. J. Song, G. T. Jeong, K. H. Hwang, G. H. Koh, H. K. Kang, and E. S. Jung, "1gbit high density embedded stt-mram in 28nm fdsoi technology," in *IEDM*, 2019.

[16] Xilinx, "Zynq ultrascale+ mpsoc data sheet: Overview," 2018.

[17] Samsung, "Ultra-low latency with samsung z-nand ssd," 2017.

[18] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[19] "The kernel's command-line parameters." [Online]. Available: https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html

[20] "Direct access for files." [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[21] D. S. Miller, R. Henderson, and J. Jelinek. Dynamic dma mapping guide.

[22] L. Pinchart. Mastering the dma and iommu apis.

[23] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, 2017.

[24] Intel, "Nvdimm block window driver writer's guide," 2016.

[25] "Flexible i/o tester." [Online]. Available: https://github.com/axboe/fio

[26] "Tpc-h." [Online]. Available: http://www.tpc.org/tpch/

[27] "Sap hana." [Online]. Available: https://www.sap.com/products/hana.html

[28] "How to emulate persistent memory." [Online]. Available: https://pmem.io/2016/02/22/pm-emulation.html

[29] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comput. Soc. Technical Committee on Comput. Archit. (TCCA) Newsletter*, 1995.

[30] M. A. Kandaswamy and R. L. Knighten, "I/o phase characterization of tpc-h query operations," in *IPDS*, 2000.

[31] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *ICS*, 2011.

[32] S. Lee, H. Bahn, and S. H. Noh, "Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," *IEEE Trans. Comput.*, 2014.

[33] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *ASPLOS*, 2017.

[34] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, "Supporting superpages and lightweight page migration in hybrid memory systems," *ACM Trans. Archit. Code Optim.*, 2019.

[35] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Comput. Archit. Lett.*, 2012.

[36] P. Vila, B. Kopf, and J. Morales, "Theory and practice of finding eviction sets," in *SP*, 2019.

[37] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *MICRO*, 2018.

[38] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, 2016.

[39] G. O. Puglia, A. F. Zorzo, C. A. F. De Rose, T. Perez, and D. Milojicic, "Non-volatile memory file systems: A survey," *IEEE Access*, 2019.

[40] L. Ramos and R. Bianchini, "Exploiting phase-change memory in cooperative caches," in *SBAC-PAD*, 2012.

[41] J. Zhan, O. Kayıran, G. H. Loh, C. R. Das, and Y. Xie, "Oscar: Orchestrating stt-ram cache traffic for heterogeneous cpu-gpu architectures," in *MICRO*, 2016.

[42] K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. Young, and H. Wang, "Density tradeoffs of non-volatile memory as a replacement for sram based last level cache," in *ISCA*, 2018.

[43] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *SIGARCH Comput. Archit. News*, 2009.

[44] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *SIGPLAN Not.*, 2011.

[45] J.-Y. Jung and S. Cho, "Memorage: Emerging persistent ram based malleable main memory and storage architecture," in *ICS*, 2013.

[46] R. Chen, Z. Shao, and T. Li, "Bridging the i/o performance gap for big data workloads: A new nvdimm-based approach," in *MICRO*, 2016.

[47] H. Song and S. H. Noh, "Towards transparent and seamless storage-as-you-go with persistent memory," in *HotStorage*, 2018.

[48] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO*, 2010.

[49] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, and Y. Guan, "Unified non-volatile memory and nand flash memory architecture in smartphones," in *ASP-DAC*, 2015.

[50] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *ASPLOS*, 2019.

[51] M. Alian and N. S. Kim, "Netdimm: Low-latency near-memory network interface architecture," in *MICRO*, 2019.

[52] S. Lee, K. Lee, M. Sung, M. Alian, C. Kim, W. Cho, R. Oh, S. O, J. H. Ahn, and N. S. Kim, "3d-xpath: High-density managed dram architecture with cost-effective alternative paths for memory transactions," in *PACT*, 2018.

[53] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016.

[54] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "Nvm duet: Unified working memory and persistent store architecture," in *ASPLOS*, 2014.

[55] S. Zhang, M. Vijayaraghavan, A. Wright, M. Alipour, and Arvind, "Constructing a weak memory model," in *ISCA*, 2018.

[56] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015.

[57] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *HPCA*, 2018.

[58] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGPLAN Not.*, 2011.

[59] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*, 2014.

[60] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *MICRO*, 2016.

[61] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *HPCA*, 2016.

[62] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *ISCA*, 2017.

[63] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *ISCA*, 2018.

[64] S. Kannan, A. Gavrilovska, and K. Schwan, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *HPCA*, 2014.

[65] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys*, 2014.

[66] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *ASPLOS*, 2019.

[67] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for persistent memory systems," in *MICRO*, 2014.

[68] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *SIGARCH Comput. Archit. News*, 2009.

[69] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *DAC*, 2009.

[70] D. Xue, C. Li, L. Huang, C. Wu, and T. Li, "Adaptive memory fusion: Towards transparent, agile integration of persistent memory," in *HPCA*, 2018.

[71] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "A memory controller with row buffer locality awareness for hybrid memory systems," *CoRR*, 2018.

[72] F. Ware, J. Bueno, L. Gopalakrishnan, B. Haukness, C. Haywood, T. Juan, E. Linstadt, S. A. McKee, S. C. Woo, K. L. Wright, C. Hampel, and G. Bronner, "Architecting a hardware-managed hybrid dimm optimized for cost/performance," in *MEMSYS*, 2018.

[73] S. Seo, Y. Cho, Y. Yoo, O. Bae, J. Park, H. Nam, S. Lee, Y. Lee, S. Chae, M. Kwon, J. Choi, S. Cho, J. Jeong, and D. Chang, "Design and implementation of a mobile storage leveraging the dram interface," in *HPCA*, 2016.

[74] B. Gervasi and J. Hinkle, "Overcoming system memory challenges with persistent memory and nvdimm-p," in *JEDEC Server Forum*, 2017.

[75] H. Lee and S. Wang, "Direct data move between dram and storage on a memory module," Patent US20 180 113 614A1, 2018.