nvramdisk: A Transactional Block Device Driver for Non-Volatile RAM

Jaemin Jung and Youjip Won

Abstract—in this work, we developed nvramdisk, a transactional block device driver for byte-addressable NVRAM. nvramdisk effectively addresses the key technical challenges in using a section of NVRAM as a transactional persistent block device. nvramdisk adopts (i) shadow block, (ii) mapping table journaling, and (iii) type-dependent ordering guarantee to provide atomicity, consistency, integrity and durability in write operations on nvramdisk imposed block device. We fully implemented nvramdisk device driver on Linux OS and port it on the desktop computer as well as Android smartphones. In memcachedb, locating the database table in nvramdisk brings ×1.9 insertions/sec and updates/sec performance gain against locating the database table in a high-end SSD (FusionIO ioDrive2). SQLite performance increases by ×2.9, from 743 ins/sec to 2184 ins/sec, in smartphone (Samsung Galaxy S4) and ×15, from 730 ins/sec to 12390 ins/sec in PC. nvramdisk yields 26% higher random write performance against Persistent Memory Block Driver. The overhead of supporting transaction accompanies 6% performance penalty in memcachedb operations.

Index Terms—Block Device, NVRAM, transaction, Ordering Guarantee, Journaling

1 Introduction

Byte-addressable NVRAM devices, e.g., STT-MRAM, PCRAM, FRAM, etc., are expected to be commercially available in the foreseeable future. While NVRAM can effectively remove various complicated software techniques, e.g., virtual memory, swap, filesystem journaling, etc., which have been used to bridge the deep chasm between memory and storage, its proper exploitation requires complete overhaul of legacy software stack and introduction of new software layers, e.g., persistent heap [1], [2], [3], [4] and byte-addressable filesystem [5], [6].

Ramdisk is widely used as a way to improve the performance of a system. Ramdisk leaves much to be desired to effectively exploit the performance of DRAM due to metadata redundancy and memory copy overhead between buffer cache and ramdisk. Despite its inefficiency, ramdisk is a very popular means for system administrators to speed up the system, e.g., local cache for web browser [7], temporary storage for decrypted data, tiered storage for frequently accessed database table [8], etc. Ramdisk owes much of its wide and popular deployment to its compatibility; Ramdisk does not require any changes in the application code. Applications can use it as a very fast storage device. Due to its volatility, ramdisk harbors only non-critical data. Recently, there have been a few interesting solutions which addresses the volatility of the legacy ramdisk, e.g., NV-DIMM [9], teraDIMM [10], and SSDAlloc [11]. These solutions combine the DRAM and NAND Flash to provide fail-safe non-volatile memory device. NV-DIMM and teraDIMM provide DIMM interface to use NAND flash memory as byte-addressable persistent memory. Different from NV-DIMM and teraDIMM, SSDAlloc is a software technique which uses an SSD as an extension to memory. One of the biggest selling points of these technologies is that the user can impose ramdisk on it and can use it as a fast block device with non-volatility.

While computer hardware has gone through significant technological advancement during the past several decades, e.g., from tape drive to hard disk, from hard disk to flash based SSD, from single core to multiple core CPU, etc., the software interface, e.g., POSIX standard, has moved at the speed of a glacier [12]. Recent proposals for integrating NVRAM device into the existing software stack [1], [2], [3], [4] promise to fully exploit the physical characteristics of NVRAM. However, these approaches require the application software to use the newly introduced interfaces. Considering slow speed at which the industry adopts new interfaces in developing new software and in migrating the existing software over a new set of interfaces, we believe that there is a significant demand for easy to use and backward compatible NVRAM interface for the quick and wide adoption of NVRAM device [13], [14].

In this work, we developed nvramdisk, a block device driver for byte-addressable NVRAM. nvramdisk plays an identical role as ramdisk driver [15] except that it provides transaction support to persistently maintain the data. When the legacy ramdisk is used for NVRAM device, unexpected system crash, e.g., sudden power loss, may leave the remnants of incomplete update in NVRAM, e.g., pointer (word) or data block. This can lead to the loss of the entire filesystem even when the system restarts. We identified three technical issues on realizing transactional block device driver on the byte-addressable NVRAM, and provided effective solutions for each of these issues: atomicity of write operation(block), atomicity of store instruction (word) and the ordering guarantee. We developed shadow block, mapping table journaling, and type-dependent ordering guarantee, with which nvramdisk successfully supports the atomicity, consistency, integrity, and durability of block write operations. The salient achievement of nvramdisk is that it
addresses all these issues without any hardware assistance and with minimal overhead.

We fully implemented the nvramdisk on Linux OS with two different CPUs (Intel x86 and ARM). For macro-benchmark, we examined the performance of SQLite and memcachedb under nvramdisk, to account for IO workloads in a smartphone and a server, respectively. For SQLite, we port nvramdisk to Samsung Galaxy S3 and S4. By locating the WAL file of SQLite on nvramdisk, we were able to achieve impressive performance gain against the default setting where the WAL file resides at NAND Flash. The SQLite performance increased by ×15, from 730 ins/sec to 12390 ins/sec, and ×2.9, from 743 ins/sec to 2184 ins/sec, in PC and Galaxy S4, respectively. The subtle change in the location of database WAL file brings profound improvement on the IO performance of the storage. In memcachedb, locating the database file of memcachedb on nvramdisk brings ×1.9 performance gain against locating it at FusionIO. For a 4 KByte random write, nvramdisk exhibited 27% higher performance than Persistent Memory Block Driver [16]. Compared to ramdisk, nvramdisk yields 6% reduction in memcachedb performance. We carefully argue that the benefits of realizing transactional capability justifies the 6% performance overhead.

2 Related Work

To address the endurance issue of NVRAM technologies like PCM, there have been a fair number of proposals, e.g., row-shifting/segment-swapping [17], hot/cold swapping [18], and data inversion [19]. These techniques deal with the endurance issue of NVRAM devices in hardware layer.

The techniques for using a memory as block device have been around for a couple of decades. The most widely used one is ramdisk [15]. In ramdisk, both in-core and on-disk metadata reside in main memory which causes significant synchronization overhead. Ramfs [20] and tmpfs [21] are filesystems developed for memory address space. Pramfs [22] is designed to use persistent RAM for filesystem with byte granularity access. Pramfs does not support transactional write(), and therefore it is possible that the data block remains partially written in case of power failure.

A number of works propose to use NVRAM to boost up the filesystem performance. FRASH [23], PFSS [24], and MiNVFS [25] store filesystem metadata in NVRAM and file data in NAND Flash memory to improve the mount latency. UB [26] improves the filesystem performance via locating the filesystem journal at NVRAM. UB uses the EXT4 journal buffer in the page cache as the EXT4 journal region, eliminating the journal commit overhead of the filesystem. Delta journaling propose to place EXT4 journal at NVRAM to reduce journaling overhead [27].

Recently, a fair number of works have been proposed to exploit byte-addressable NVRAM in the existing computing systems [2], [3], [4], [5], [6], [28], [29]. They can be categorized into byte-addressable filesystem [5], [6], persistent heap [2], [3], [4], fast storage device [29], or whole persistent memory based system [28]. One of the key technical ingredients in these proposals is how to support transactional writes, i.e., how to clean-up the remnants of write operations in case of power failure. BPFS [5] is byte-addressable filesystem and requires hardware support, e.g., supercapacitor for word level atomicity and epoch barrier for ordering guarantee.

A number of works have been proposed to use flash memory as main memory extension to provide non-volatility for ramdisk abstraction [9], [10], [11]. These solutions are orders of magnitude slower than DRAM based ramdisk and still are not economically viable.

NV-Heaps [2], Mnemosyne [3], and SoftPM [4] propose to exploit NVRAM as persistent heap. They designate a special segment in the process address space as persistent heap to harbor persistent object and this segment is mapped into physical NVRAM. Applications need to use new set of interfaces to create/delete the persistent memory region and to ensure consistency against system crashes. Bailey et al. [30] provide a nice survey on the design considerations, e.g., address space, process model, data corruption, data portability, and security, when adopting NVRAM into existing computer systems.

For the correctness of the system against power crash, the order in which the CPU issues store instructions and the order in which they reach the NVRAM should match. WSP [28] suggested using capacitor for ordering guarantee. Bhandari et.al. [31] studied various software based ordering guarantee mechanisms and concluded write-through yield the best performance. The Rio system provides a technique to survive the system crash and reboot via treating the part of main memory as a protected region and via using this region to harbor crash recovery information [32]. However, it assumes that systems have an uninterrupted power supply. Persistent Memory Block Driver [16] provides several operations for ordering guarantee and can introduce arbitrary delay in memory operations. However, PMBD does not provide transactional guarantee. Using a journaling filesystem on top of PMBD can be a partial remedy to survive the information loss against unexpected system crash. However, the ext4 journaling overhead is significant [27], [33]. Supporting atomicity in a block layer can eliminate the double write [34] overhead in differential logging [35].

3 Problem Assessment

Writing a 4 KByte block in memory consists of multiple store instructions. Transactional block device is a block device that performs a write operation as a transaction, i.e. with atomicity, consistency, isolation, and durability being guaranteed. It persistently manages the data and is robust against unexpected power failure. Transaction property of a write is required for upper layer software, e.g., filesystem and DBMS, either to function correctly, or to simplify the algorithm for transactional guarantee. A few works [35], [37] showed that if a write atomicity on the block device abstraction is guaranteed, one can achieve a high level of data integrity with fewer writes. Supporting consistency and isolation is trivial. Since a write operation is a sequence of storage instructions starting from a given virtual address, two concurrent write operations can proceed independently or one of the two operations should be blocked until the other operation is completed. Key technical ingredient of
nvramdisk is to guarantee the *atomicity* and *durability* of write operations.

We assume that DRAM and NVRAM form a homogeneous physical address space and that DRAM and NVRAM can have different clock speeds [38], [39]. Both NVRAM and DRAM are directly attached to CPU. The nvramdisk should be able to dynamically expand and shrink as current ramdisk implementation [15]. Fig. 1 illustrates the architecture layout of memory subsystem consisting of DRAM and NVRAM in x86.

As nvramdisk use the same memory model of ramdisk, nvramdisk only concerns about load/store operations, excluding DMA (Direct Memory Access) operations. A few works analyze the trade-offs between polling-based IO and interrupt driven IO and show that the context switch overhead can be excessive in fast block devices, e.g., PCIe based high-end SSD [40], [41], [42].

Supporting the atomicity of write boils down to recovering partially written blocks. Since writing a block, e.g., 4 KByte, will be performed via multiple store instructions, unexpected power failure may leave a memory page partially written. For atomicity, nvramdisk needs to retain old copies of the data block. nvramdisk needs to be harnessed with metadata to keep track of the most recent copy of the block and the mechanism to protect the metadata against the crash. There exist various crash recovery techniques in filesystems, e.g., journaling [43], versioning [44], shadow paging [45], copy-on-write [46], etc. Most of these approaches are designed to recover concurrent transactions. In nvramdisk, the number of concurrent block writes is limited by the number of “running” threads. Therefore, the number of concurrent block writes is inherently limited to the number of CPU cores and there is no dependency among the accesses. The recovery scheme needs to be effectively tailored for nvramdisk. What lies at the core of the problem is making the metadata crash safe. Each field of the metadata is usually a word. An assembly instruction is atomic only against race condition, but not against unexpected power loss. In NVRAM device, unexpected power loss may result in partially written word in memory. We need a mechanism to detect whether a given metadata field contains a legitimate value and to restore the metadata to leave the system at the consistent state. Battery backed DRAM [32], [47] has been proposed to guarantee the atomicity in *store*, but it is subject to periodic maintenance which can be avoided by using NVRAM. Supercapacitors [5], [28] can be a better solution. Although supercapacitors require less maintenance attention than batteries, they are still not free from breakdown, defect, and aging [28]. One possible solution for atomic *store* is to develop dedicated memory controller for NVRAM device.

Durability support becomes non-trivial since the order at which CPU issues the store instructions and the order at which the writes are actually reflected on memory may differ. There can be several factors that cause the two orders to differ: cache replacement policy, out-of-order execution, etc. Moreover, modern memory subsystem architecture shuffles the write order to improve the locality and parallelism [5], [48]. A subset of the written block may be at the volatile layer of the memory hierarchy, e.g., CPU cache, while the rest of the block has already reached the non-volatile layer. Without ordering guarantee, sudden power loss may leave the system in an undefined state violating the causal dependency between the memory variables. For durability support, ordering guarantee is a necessary condition. Existing works use methods such as epoch-barrier [5], explicit cache flushing [6], etc., to preserve the order of the operations. “Write-through” mechanism can be a simple remedy but the performance penalty may not be acceptable. It is important that the right ordering guarantee mechanism, which properly incorporate the memory access characteristics, is used.

In this work, we dedicate our efforts to developing a set of solutions for transactional *write* for block device for NVRAM.

### 4 NVRAMDISK

#### 4.1 Design

![nvramdisk in Software Stack](image)

nvramdisk aims at providing block device layer that provides backward-compatible block I/O interface for NVRAM (Fig. 2). To ensure atomicity and durability, nvramdisk adopts shadow block mechanism, metadata journaling, and type-dependent ordering.
nvramdisk consists of a set of physical pages on NVRAM device. These pages do not have to be contiguous. There are two types of regions in nvramdisk: metadata and data. This paper is overloaded with the term “nvramdisk”. It can denote either device driver or storage partition.

For atomicity, nvramdisk adopts shadow block which updates blocks out-of-place so that old blocks are recovered in case of power failure. There exists a mapping table to keep track of the location of a given logical block. The mapping table translates the logical block address to the virtual address (virtual page number). For the consistency of the mapping table updates, nvramdisk journals the history of mapping table updates. When a block is being written, the subsequent write requests to the same block are blocked. The number of shadow blocks in the current nvramdisk is set to the number of hardware threads in the CPU. Mapping table and journal reside at metadata region.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2428708, IEEE Transactions on Computers
the respective mapping table entry is updated to refer to the new location. To guarantee the integrity of the mapping table update, nvramdisk journals the mapping table update and provides the consensus mechanism to determine the correct value for mapping table entry in case of power failure. To ensure the durability, we adopt type-dependent ordering guarantee programming techniques.

### 5.2 Writing a Block

The entire process of transactional block write is a combination of out-of-place updates for a data block and write-ahead logging for mapping table updates.

The process of transactional block write consists of five phases: allocate, update, journaling, checkpoint, and clean-up (Table 1). In the allocation phase, nvramdisk allocates a shadow block and a journal record to accommodate incoming write. In the update phase, the data is written to the allocated shadow block. Third, in the journaling phase, nvramdisk journals mapping table entry, which will be explained in more detail in section 5.3. Fourth, in the checkpoint phase, nvramdisk checkpoints the mapping table record to mapping table. Fifth, in the clean-up phase, nvramdisk adds the obsolete block to the shadow block pool and resets the journal record.

There should be a sufficient number of shadow blocks so that the system performance is not limited by the availability of the shadow blocks. The number of concurrent write operations to nvramdisk can be as many as the number of CPU cores.

Minimizing the overhead of shadow block allocation is one of the key ingredients in nvramdisk performance. Obtaining a page from the kernel page allocator based on the buddy algorithm carries too much overhead. We “pre-allocate” a certain number of NVRAM pages as shadow blocks instead of allocating page from the kernel page allocator on demand. In addition, nvramdisk replenishes a shadow block with an old data block which becomes obsolete after out-of-place update.

### 5.3 Journaling the Mapping Table

In a shadow block, the mapping is updated to refer to the new virtual memory location of the given logical block. Since mapping table resides at NVRAM, it is also subject to the system crash.

We devised a sophisticated journaling mechanism for mapping table updates to make the mapping table updates power-crash safe. A new virtual memory location of a given logical block address (virtual page number) and the logical block address are journaled. This occurs at the journaling phase in Table 1. The mapping table should be recovered even though the system crash leaves some fields of the journal entry incompletely updated, e.g., only a few bytes of 8 byte (unsigned long) field is written. The crux of this problem is determining the correctness of a given journal field.

In the journal, we maintain two replicas of logical block number. Only when these two values match, the logical block number is valid. In journal, the virtual page number should be recorded before the logical block number. This causal order in which the journal fields are updated is carefully crafted so that we replicate only logical block number, not the virtual page number.

A journal entry consists of the two replicas of a logical block address and the respective virtual address (virtual page number) which represents the new location of a given logical block on the virtual memory, $<\text{VPN, } L_1, L_2>$ (Fig. 5). In case of system crash, recovery module examines all two LBAs in the journal records to identify the right value for recovery. In each write, a journal record and a shadow block are allocated as a pair. The number of journal records equals the number of shadow blocks.

The proposed journaling scheme provides ordering guarantee. The journaling module first records the new location of a given logical block to VPN field and then creates two replicas of the logical block number. When the journal update completes, the mapping table entry is updated with new virtual page number for the given logical block. Journal entry is reset to NULL when a transaction completes. Journal reset plays a critical role. When OS detects the non-NULL value in the journal, it determines that the system has not shut down properly and invokes the recovery procedure.

### 5.4 Correctness and Crash Recovery

The state of the write operation is determined by five attributes: four attributes correspond to the four fields in a journal entry and one attribute corresponds to the respective value of the mapping table entry.

Given that the ordering guarantee is preserved, the process of updating the mapping table can be shown in 12 steps.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2428708, IEEE Transactions on Computers

<table>
<thead>
<tr>
<th>State</th>
<th>Journal Entry</th>
<th>Table Phase</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>φ null null</td>
<td>vpnnull</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>x null null</td>
<td>vpnnull</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>3</td>
</tr>
<tr>
<td>S3</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>4</td>
</tr>
<tr>
<td>S4</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>5</td>
</tr>
<tr>
<td>S5</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>6</td>
</tr>
<tr>
<td>S6</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>7</td>
</tr>
<tr>
<td>S7</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>8</td>
</tr>
<tr>
<td>S8</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>9</td>
</tr>
<tr>
<td>S9</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>10</td>
</tr>
<tr>
<td>S10</td>
<td>vpnnull</td>
<td>vpnnull</td>
<td>11</td>
</tr>
<tr>
<td>S11</td>
<td>null x</td>
<td>vpnnull</td>
<td>12</td>
</tr>
</tbody>
</table>

TABLE 2: States of nvramdisk (φ = Don’t care)

![State Diagram of nvramdisk Write](image)

Fig. 6: State Diagram of nvramdisk Write

each of which corresponds to a different state. Table 2 shows the 12 steps in mapping table journaling. When the system crashes while updating a certain field, value of the field is marked ‘X’. Initially, all lba fields, L1 and L2, are NULL. OS does not need VPN field if L1 and L2 are NULL. The initial value of VPN field is Don’t Care (φ).

At the start of transactional block write, nvramdisk allocates a shadow block and a journal record in the allocation phase (Table 1). Then, nvramdisk performs block write to the shadow block allocated. Once this is done, nvramdisk starts journaling. The initial state of journaling is S0, where all lba fields are NULL, <<φ, NULL, NULL, vpnnull >>. nvramdisk journals the virtual page number of the shadow block where a new block is written, vpnnull. If nvramdisk fails to journal the shadow block index in the middle of an update, the VPN field will contain corrupt value (S1). If it succeeds, the journal entry holds shadow block index in its VPN field (S2). Then, nvramdisk makes two replicas in the journal record. From S2, the state changes to S3 if the first replica of logical block number is recorded successfully. If it fails, it goes to S3. For the second replica, the same rule is applied. S4 denotes the state at which two replicas are successfully recorded. After recording two replicas, nvramdisk checkpoints the journal; updates the mapping table entry to refer to the new location for a given logical block (S5). After checkpoint, the process is complete. The journal entry is reset via writing NULL to two fields for logical block number. When all LBA fields are reset to NULL successfully, the system goes to the initial state, S0.

Let us explain the recovery procedure. In case of system failure, nvramdisk performs either undo or redo based on the state of the system. When the system is found to be in one of S0, ... , S5, recovery procedure simply performs “undo”, i.e., resets all lba fields to NULL. The system state reverts to S0 and the mapping table remains intact. When the system is in one of S6, ... , S11, the recovery procedure performs “redo”, i.e., the system updates the mapping table using the journal and performs clean-up. This leads the system to a new initial state, S0'. Fig. 6 illustrates the state transition diagram. There are three types of states: initial state, normal state, and fault state. The initial state is denoted by double circle. The normal states are the ones with single circle with white background. It denotes valid states. The fault states are the circles with black background. The solid lines denote normal state transitions. The dotted lines denote state transitions caused by system crash. The dashed-dotted lines denote state transitions caused by recovery procedure.

6 TYPE-DEPENDENT ORDERING GUARANTEE

6.1 Synopsis: Guaranteeing the Order

Despite the non-volatility of NVRAM device, it is possible that power failure causes data loss in the volatile layer of the memory hierarchy, e.g., TLB, CPU cache, etc. It is critical that the contents in the volatile layer are properly synchronized with the NVRAM in case of power failure. Bhandari et al. [31] showed write-through mechanism yields the best performance for ordering guarantee. However, Their work focuses on guaranteeing the order among all pairs of store instructions. In nvramdisk, shadow block mechanism along with mapping table journaling guarantees the atomicity of the block write operation. It is not required to guarantee the order among the store instructions for writing a data block.

We examined the overheads of the various ordering guarantee methods and propose to apply different ordering guarantee schemes subject to the access characteristics of the respective NVRAM regions; data block, mapping table, and journal.

There are a number of factors that determine the order in which “memory writes” are physically reflected in memory: (i) cache policy (write-back, write-through, write combining vs. uncachable), (ii) type of store instruction (movq vs. movntq), and (iii) use of explicit cache line flush (clflush) and memory barrier (mfence and sfence). Let us briefly explain each of these attributes. There are three caching policies: write-back, write-through, and uncachable. In write-back (WB) policy, cache contents are occasionally written to memory via a certain eviction policy, e.g., LRU. The unit of eviction is cache line. In write-through policy (WT), when a cache line is written, the respective contents are also written in memory. Unlike in write-back method, only the updated words are written to memory in write-through mode. In uncachable policy, cache is not used. There are two types of write instructions: temporal instructions, e.g., movq and non-temporal instructions, e.g., movntq. movq writes to cache and movntq writes directly to memory. We can preserve the ordering via explicitly flushing the cache contents or via using barrier instruction. Clflush is an instruction which flushes a given cache line. There are two types of barrier instructions: mfence and sfence. All load and store instructions that precede the mfence instruction in program order become globally visible before
all load/store instructions that appear after the mfence instruction in the program order. sfence serializes only store instructions.

Combining these options, we define five methods for ordering guarantee: M_{CF}, M_{NT}, M_{WT}, M_{UC}, and M_{WC} (Table 3). M_{CF} uses explicit cache flush and memory barrier. M_{WT} uses non-temporal command with barrier. M_{UC} sets the caching policy to uncacheable. M_{WC} uses explicit memory barrier to flush the contents on write-combining buffer into NVRAM.

### 6.2 Write Disturbance in NVRAM

Under unexpected power-loss, explicit cache line flush can corrupt memory addresses other than what is being updated. We call this Write Disturbance in NVRAM. Write disturbance denotes the situation where a memory update operation dismantles the contents of the neighboring memory locations. Write disturbance is widely observed in Flash memory [49]. When we use explicit cache flush with barrier synchronization for ordering guarantee, memory is updated in cache line unit. In the other four methods in Table 3, the words that are not updated are masked off and only the updated words are written to memory. The cacheline update consists of a number of memory IOs whose sizes are governed by the memory bus bandwidth. The cacheline can be distributed across multiple banks [48]. It is not possible in practice that all words that form a cacheline are updated at the same time. A sudden power failure may corrupt any words in the cache line in a non-deterministic manner. This bears rather important implications in implementing nvramdisk. When synchronizing the mapping table entry by clflush, power loss can corrupt a word that is not being updated but belong to the same cache lines as the word being updated.

### 6.3 Type-Dependent Ordering Guarantee

The objects in different regions which correspond mapping table, journal, and data blocks, bear different access characteristics. Data region is read and written in 4KByte units. Mapping table is frequently read and written in 8 Byte (word) granularity. Journal entry is write-only and is updated in 8 Byte (word) granularity. We propose to use different ordering guarantee scheme in Table 3 for different region.

Let us examine the overhead of two ordering guarantee schemes M_{NT} and M_{CF} in Table 3 in detail. Let us start with M_{NT}. For each 8 Byte word, we need one movntq and one mfence to synchronize. However, for 4 KByte block write, we do not need to put barrier for every store instruction. This is because we do not need ordering guarantee among the store instructions for the same data block. For 4 KByte write, we need 512 movntq’s but only one barrier mfence. M_{CF} yields different result. For 8 Byte word, we need one movq, clflush, and mfence for synchronization. For a 4KByte block, 512 movq’s and 64 clflush. As in the case of using non-temporal instruction, M_{NT}, we need one barrier in writing block.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cache Policy</th>
<th>Store Method</th>
<th>Flush</th>
<th>Barrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{CF}</td>
<td>WB</td>
<td>movq</td>
<td>clflush</td>
<td>mfence</td>
</tr>
<tr>
<td>M_{NT}</td>
<td>WB</td>
<td>movntq</td>
<td>none</td>
<td>mfence</td>
</tr>
<tr>
<td>M_{WT}</td>
<td>WT</td>
<td>movq</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>M_{UC}</td>
<td>UC</td>
<td>movq</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>M_{WC}</td>
<td>WC</td>
<td>movq</td>
<td>none</td>
<td>mfence</td>
</tr>
</tbody>
</table>

### TABLE 3: Ordering Guarantee Programming Methods

<table>
<thead>
<tr>
<th>x86 Inst.</th>
<th>movq</th>
<th>movntq</th>
<th>mfence</th>
<th>clflush</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>0.33</td>
<td>1</td>
<td>6 / 5</td>
<td>90</td>
</tr>
<tr>
<td>Max</td>
<td>3 / 300</td>
<td>33 / 9</td>
<td>240</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 4: Cycles Per Instruction in Intel x86 Processors [50]

Each of movq, movntq, clflush and mfence instruction has different latency and throughput. Based on the minimum and maximum CPI for each of these instructions (Table 4) [50], we can compute the minimum and maximum ordering guarantee overhead for 8 Byte and 4 KByte writes under M_{CF} and M_{NT} policies. Table 4 summarizes the result.

For block writes, M_{NT} exhibits better worst case performance than M_{CF} does. Meanwhile, M_{CF} yields the better worst case performance than M_{NT} does. Writing 4 KByte most likely yields the best case performance since the consecutive movntq or clflush instructions can exploit parallelism. For word writes, the latencies are expected to be close to the worst case latency because each instruction is executed once. According to this calculation, M_{CF} and M_{NT} may be the right ordering guarantee schemes for word updates and for block update, respectively. This observation clearly suggests the need for type-dependent ordering guarantee which applies different ordering guarantee schemes to individual objects in the nvramdisk subject to their access characteristics.

ARM architecture does not support non-temporal instructions. Current ARM port of nvramdisk (Samsung Galaxy S3 and S4) adopts the cache flush and memory barrier scheme for ordering guarantee in all regions of nvramdisk.

### 7 Evaluation

#### 7.1 Experiment Setup

Our performance study consists of three subjects: (i) performance of different ordering guarantee schemes, (ii) IO performance and (iii) application performance. We compared the IO performance of nvramdisk against three SSDs
7.2 Type-Dependent Ordering Guarantee

Fig. 7: Access Latency for Ordering Guarantee Methods (async: without ordering guarantee, M\textsubscript{NT}: non-temporal, M\textsubscript{WC}: write-combining, M\textsubscript{CF}: cache flush, M\textsubscript{WT}: write-through, and M\textsubscript{UC}: uncachable)

![Graph showing access latency for different ordering guarantee methods.](image)

In this experiment, we first examined the performance of ordering guarantee in raw memory access and then based on the observation obtained, we examined the performance of ordering guarantee in block device access. We examined the latencies of read (load), write (store) and read/write (load/store) operations, under five ordering guarantee methods. We used two IO sizes: 8Byte (word) and 4KByte (block). 4KByte read/write consists of multiple load/store instructions. The read/write workload consists of 50% read and 50% write. We used mfence for memory barrier. Fig. 7 illustrates the result. For 8 Byte access, write-through, M\textsubscript{WT}, yields the best performance in all three workloads (Fig. 7a). Accesses to mapping table is a mixture of read and write operations and therefore write-through, M\textsubscript{WT} is the optimal method. Accesses to journal are mostly write-only. For journal updates, both write-through, M\textsubscript{WT}, and uncachable, M\textsubscript{UC} yield the best performance. For 4 KByte accesses, non-temporal command with barrier, M\textsubscript{NT}, yields the best result (Fig. 7b).

With this observation, we developed four type-dependent ordering guarantee schemes for nvramdisk and applied different schemes to data block, mapping table, and journal updates (Table 6).

```
<table>
<thead>
<tr>
<th>Write Policy</th>
<th>Data block</th>
<th>Mapping table</th>
<th>Journal</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsubscript{P1}</td>
<td>M\textsubscript{NT}</td>
<td>M\textsubscript{WT}</td>
<td>M\textsubscript{WC}</td>
</tr>
<tr>
<td>\textsubscript{P2}</td>
<td>M\textsubscript{NT}</td>
<td>M\textsubscript{WT}</td>
<td>M\textsubscript{WT}</td>
</tr>
<tr>
<td>\textsubscript{P3}</td>
<td>M\textsubscript{NT}</td>
<td>M\textsubscript{WT}</td>
<td>M\textsubscript{WT}</td>
</tr>
<tr>
<td>\textsubscript{P4}</td>
<td>M\textsubscript{WT}</td>
<td>M\textsubscript{WT}</td>
<td>M\textsubscript{WT}</td>
</tr>
<tr>
<td>\textsubscript{P5}</td>
<td>M\textsubscript{CF}</td>
<td>M\textsubscript{CF}</td>
<td>M\textsubscript{CF}</td>
</tr>
</tbody>
</table>
```

Table 6: Type-dependent ordering policies (NT: non-temporal, WT: write-through, UC: uncachable)

![Graph showing bandwidth vs record size for different policies.](image)

Fig. 8: Performance of Type-dependent Ordering Guarantee Schemes, \textsubscript{P1}: write-through only, \textsubscript{P6}: cache flush only

In summary, for data block, mapping table, and journal region, use of non-temporal command (M\textsubscript{NT}), write-through (M\textsubscript{WT}), and use of non-temporal command (M\textsubscript{NT}), yielded the best performance, respectively. In subsequent experiment, this combination of ordering guarantee scheme is used.

7.3 Random Write

The overhead of nvramdisk is caused by ordering guarantee, mapping table updates and mapping table journaling,
which only happen with write operations.

We dedicate our efforts to examining the random write performance of nvramdisk. We compared the performance of nvramdisk with three SSD's, 840PRO (SATA3, Samsung), Revo Drive3 (PCIe, OCZ), and ioDrive2 (PCIe, FusionIO) and two memory based block devices, PMBD [16] and ramdisk. We generated 4 KByte random writes on these devices and measured IOPS and latency, using iometer [52]. We set the ordering guarantee options of PMBD as non-temporal store, write barrier, and write-back cache options which are similar to the one used by nvramdisk.

nvramdisk exhibits ×2.5 random IO performance improvement compared to the fastest SSD (FusionIO ioDrive2). nvramdisk exhibits 27% higher IOPS than PMBD. In IOPS and IO latency of 4 KByte random write, nvramdisk is on a par with ramdisk.

We examined the overhead of ordering guarantee in nvramdisk. We measured the random write performance of nvramdisk and ramdisk varying the record sizes from 4 KByte to 256 KByte. We measured the nvramdisk performance with and without ordering guarantee feature to physically examine its performance overhead. The nvramdisk without the ordering guarantee feature is labeled NVRAMDISK-async.

nvramdisk and ramdisk exhibit identical performance in small IO (4KB). However, the performance difference becomes more significant as the IO size increases. For 256KB IO, nvramdisk exhibits 37% lower performance against ramdisk (Fig. 10).

In small IO, the overhead of ordering guarantee is not visible. Removing the ordering guarantee does not bring any visible performance gain. However, when IO size is large, e.g., 64 KByte, the ordering guarantee overhead becomes significant. When ordering guarantee is removed, the performance of nvramdisk becomes on a par with the performance of ramdisk. With ordering guarantee, the performance of nvramdisk decreases by 32%. Given that the dominant use of ramdisk is to harbor database table for structured and unstructured data, small random write performance will be the most important measure of the effectiveness of nvramdisk. Based on our observation in this experiment, we do not expect a transactional block device to incur any significant performance overhead compared to its ramdisk counterpart. The next two sections are dedicated to studying the performance of nvramdisk in real applications.

7.4 Application 1: memcached

Memcached is distributed key-value store which usually uses fast storage device, e.g., an SSD, to maintain its data persistently [53]. Memcached uses Berkeley DB [54] as its key-value engine. We examined the performance of memcached operation under four block devices: nvramdisk, PMBD, ramdisk and ioDrive2. We included ramdisk in this experiment to examine the overhead of supporting the transaction property in nvramdisk. The size of key and value are 16 byte and 512 byte, respectively. We configured memcached in transactional mode with 64MB cache. Both the memcached server and the client were on the same machine to mitigate the network overhead.

Fig. 11 shows the performance of insertion, deletion, and update for ioDrive2, ramdisk, and nvramdisk. nvramdisk exhibits up to 89% and 26% performance gain against ioDrive2 and PMBD, respectively. nvramdisk exhibits as much as 6% lower performance than ramdisk. We carefully argue that 6% overhead to provide transaction property is justifiable.

In Fig. 10, we have observed that the performance of nvramdisk can decrease by as low as 37% against ramdisk's performance for large IO, e.g., 128KByte. Memcached experiment shows that nvramdisk is very unlikely to experience this overhead in the real settings since IO size is normally less than a few page.

7.5 Application 2: SQLite

SQLite [55] is serverless DBMS which is widely used to maintain small sets of records especially in mobile plat-
forms, e.g. Android, iOS, Mozilla, Tizen as well as PC, e.g. web browser. A number of efforts have recently been proposed to reduce the SQLite driven IO traffic [33], [56], [57]. SQLite maintains separate roll-back journal file for crash recovery. This roll-back journal file resides at eMMC, the NAND Flash device for mobile devices. We improve the SQLite performance by locating the roll-back journal file at nvramdisk. We modified the SQLite to locate the rollback journal file (or WAL file) on the nvramdisk. We port nvramdisk driver to two smartphones. Galaxy S3 (Android 4.0, Linux 3.0.31) and Galaxy S4 (Android 4.3, Linux 3.4.5).

We examined the performance of SQLite under different journal modes. Fig. 12 illustrates the result of the experiment in Smartphone platforms. For WAL mode, which is the default journal mode in recent Android OS, SQLite performance (insert operation) increases by \(\times 2.9\) from 743 ins/sec to 2184 ins/sec (Galaxy S4) and \(\times 2.6\) from 235 to 611 (Galaxy S3), respectively. The SQLite performance gain in PC is even starker. Via locating WAL file in nvramdisk, SQLite performance increases by \(\times 15\) (Fig. 13). Compared with ramdisk, the nvramdisk imposes only about 1% of insertions/sec performance for WAL mode.

We examined the block accesses to identify the source of improvement. In WAL mode, the updated database page is appended to WAL file. When the number of outstanding pages exceeds a certain threshold or when the application closes, the page entries in the WAL file are checkpointed to the database. If WAL file resides at nvramdisk, most of the write will go to nvramdisk which is orders of magnitude faster than Flash device. Fig. 14 shows the block traces for performing a sequence of insert operations when WAL resides at the SSD and at nvramdisk, respectively (PC platform). In Fig. 14a and Fig. 14b, the time to checkpoint the WAL entries to the SQLite database is almost identical since the checkpoints are performed on DB file in the SSD. However, the time for inserting a page entry to WAL file dramatically decreases when locating a WAL file to an nvramdisk instead of an SSD. When WAL resides at the SSD, the interval between the successive checkpoint is 663 msec (Fig. 14a). When WAL file resides at nvramdisk, the interval between the successive checkpoint reduces to 35 msec (14b), resulting in \(\times 15\) performance improvement.

### 7.6 Effect of NVRAM Latency

There exists a variety of different NVRAM technologies, e.g. STT-MRAM, PC-RAM, FRAM and, RE-RAM [58]. Each of these devices has different access latencies; write latency of PCRAM can be as much as \(\times 10\) slower than DRAM whereas write latency of STT-MRAM can be as fast as DRAM. The access latency also varies subject to the scale of the manufacturing process. We examined the effect of NVRAM latency on the application performance. We measured the performance of memcachedb and SQLite while varying latencies of NVRAM device. We used busy loop to introduce delay in PMBD [16]. We vary the NVRAM device latency from 1.0 to 2.0 compared to the DRAM latency. 1.0 and 2.0 means that the latency of NVRAM is the same as DRAM and twice longer than DRAM, respectively.

Fig. 15 illustrates the application performance normalized against the case when the latencies of NVRAM and
DRAM are the same. Both of these applications are IO intensive. As expected, the performance proportionally decreases with the NVRAM latency. The performance of SQLite is more sensitive to NVRAM latency than the performance of Memcached. Memcached performance decreases to 45% when the latency of NVRAM becomes twice as large as that of DRAM. In Galaxy S4, the SQLite performance decreases to 50% when NVRAM latency is twice larger than DRAM latency. If NVRAM latency is three times larger than DRAM's, there is no performance gain in locating the WAL file in nvramdisk.

### 7.7 Static vs. Dynamic Allocation of Shadow Block

Finally, we examine the effectiveness of our static pre-allocation scheme of shadow block. In nvramdisk, we over-provision a fixed number of pages to accommodate incoming write instead of relying on page allocator provided by Linux kernel. This is to avoid the various overheads of Kernel page allocator including buddy algorithm. We compared the performance of static pre-allocation scheme and allocator of Linux Kernel. We issued “write()” request (O_DIRECT) to nvramdisk with varying IO sizes (4, 16, 64, 256, and 1024 KB) and examined the performance under two different shadow block allocation schemes. The fixed pre-allocation scheme exhibits as high as 25% better performance than kernel page allocator. Fig. 16 illustrates the result.

### 8 CONCLUSION

In this work, we developed nvramdisk, a transactional block device driver for NVRAM, which consists of three key technical ingredients: shadow block, mapping table journaling, and type-dependent ordering guarantee. With these features, write operations on nvramdisk are guaranteed to be atomic. The most salient feature of nvramdisk is its backward compatibility. It is just a very fast block device with DRAM-like latency and persistency. Application do not require any modifications to work on top of it. With nvramdisk, performances of Memcachedb and SQLite increase by ×1.9 and by ×2.9, respectively. In particular, the performance improvement obtained by locating the SQLite WAL file on nvramdisk is significant since it effectively relieves the burden of heavy fsync(). Despite its non-trivial mechanism of supporting transaction property, the overhead of nvramdisk is insignificant in real situations. The transactional capability comes with as much as 6% performance degradation in key-value store operation (memcachedb) against legacy ramdisk. nvramdisk opens up a new opportunity for modern applications to exploit the salient nature of NVRAM in an extremely versatile manner.

### ACKNOWLEDGMENTS

This work is supported in part by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device). The authors would like to give special thanks to Myungsik Kim for his help, support, and expert opinions.

### REFERENCES

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2428708, IEEE Transactions on Computers

Jaemin Jung received the BS and MS degrees in electrical and computer engineering from the Hanyang University, in 2007 and 2009, respectively. He is currently working toward the PhD degree in the Department of Computer and Software, Hanyang University, Seoul, Korea. His research interests include operating systems, file and storage subsystems, and byte-addressable persistent memory.
Youjip Won received the BS and MS degrees in computer science from the Seoul National University, in 1990 and 1992, respectively. He received the PhD degree in computer science from the University of Minnesota, Minneapolis, in 1997. After receiving the PhD degree, he joined Intel as a server performance analyst. Since 1999, he has been with the Department of Computer and Software, Hanyang University, Seoul, Korea, as a professor. His research interests include operating systems, file and storage subsystems, multimedia networking, and network traffic modeling and analysis.