Janus: Optimizing Memory and Storage Support for Non-Volatile Memory System

Sihang Liu, University of Virginia
Korakit Seemakhupt, University of Virginia
Gennady Pekhimenko, University of Toronto
Aasheesh Kolli, Penn State University, VMware Research
Samira Khan, University of Virginia

1 Introduction

Non-volatile memory (NVM) technologies, such as Intel's Optane, offer data persistence similar to storage devices while delivering performance close to that of DRAM, having the potential to revolutionize persistent data management. NVMs store persistent data in memory and allow direct manipulation of persistent data with load and store instructions rather than relying on software intermediaries (e.g., file system). The developers of software, such as databases, file systems, key-value stores, etc., have been inspired to leverage this opportunity to manage persistent data at high efficiency. The directly managed persistent data are expected to be recoverable, even in the event of a failure (e.g., power outage or system crash), and therefore, we refer to these classes of software as crash consistent software.

1.1 Background

Crash consistent software for NVMs exhibit a unique property, that is they place *writes to memory on the critical path* of program execution. Conventionally, only reads to memory are on the critical path, while writes may be buffered, coalesced and reordered on the way to memory for better performance. In comparison, to guarantee crash consistency, the order of writes to memory is severely constrained to ensure data recoverability across failures [3, 6]. For example, a database transaction needs to have its updates persistent before it commits. Therefore, all writes issued to persistent data within a transaction have to reach all the way to NVM (or the persistent domain) before it commits. To prescribe the order in which writes become persistent, x86 and ARM ISAs have introduced new instructions to ensure the durability of persistent data. However, these durability guarantees imply that writes to persistent data may fall on the critical path of program execution.

Placing persistent data on NVM not only moves writes onto the critical path, but also introduces extra latency due to additional constraints on maintaining persistent data in NVM. First, due to the non-volatility, the confidentiality and integrity of data in NVM must be maintained using encryption and integrity verification mechanisms [1, 5, 7]. Second, most NVM technologies suffer from a limited bandwidth and lifetime, necessitating deduplication, compression, and/or wear-leveling of NVM writes [2, 4, 9]. All these encryption, integrity protection, deduplication, and compression operations, collectively referred to as *backend memory operations (BMOs)* henceforth, are performed at the memory controller and significantly increase the NVM write latency. Moreover, since writes fall on the critical path of the crash-consistent software, the increase in write latency significantly degrades the performance.

1.2 Motivation and Challenge

Figure 1 demonstrates the latency breakdown of a write access. We assume a system with Intel's Asynchronous DRAM Refresh (ADR) technique that ensures writes to NVM become persistent as soon as they are placed in the write queue in the memory controller. Without any BMOs (Figure 1a), only the writeback from the cache hierarchy to the memory controller falls on the critical path (typically 15 ns), whereas the subsequent operations in the memory controller and the actual NVM access latency do not contribute to the critical path. However, with BMOs (Figure 1b), both the writeback and the BMO latency are on the critical path, as the write cannot be placed in the write queue and hence cannot be considered persistent until BMOs are completed. Due to the extra hundred-nanosecond BMO latency, the critical latency is increased by more than 10 times. The *goal* of this work is to minimize the latency overhead in write operations caused by these BMOs in NVM systems.

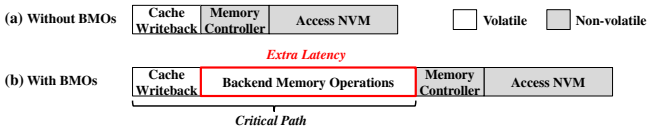

Figure 1. Write latency (a) without and (b) with BMOs.

When BMOs are viewed as dependent, indivisible operations, common latency optimizations (e.g., parallelization) are precluded. For example, in a system with encryption and deduplication, performing deduplication before encryption is a reasonable approach, while performing them in parallel is not — deduplication can change the address mapping of the data blocks and invalidate the encryption output that used the old address. Figure 2a shows the serialized BMO latency of encryption and deduplication that falls on the critical path of execution in an undo logging transaction. Therefore, the major *challenge* is in figuring out *how to optimize these seemingly dependent, monolithic operations.*

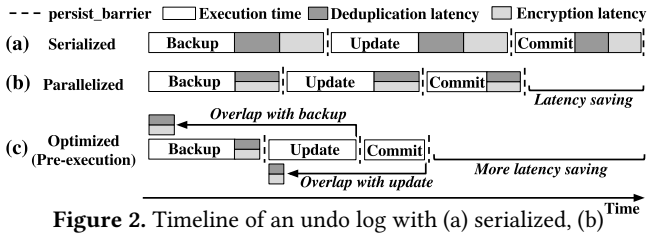

Figure 2. Timeline of an undo log with (a) serialized, (b) parallelized and (c) pre-executed BMOs.

2 Janus Mechanism

2.1 Key Ideas on Optimization

Our key insight to optimize these BMOs is to first decompose them into a series of *sub-operations*, and then explore opportunities to optimize the latency across BMOs in two ways: (1) parallelize BMOs as much as possible, and (2) pre-execute BMOs to move their latency off the critical path.

Decomposition. Figure 3a shows the breakdown of BMOs that consist of counter-mode encryption and deduplication. Encryption consists of three sub-operations: (E1) generate a new counter, (E2) generate a one-time padding (OTP) by encrypting an unique counter, and (E3) encrypt data by XORing data block with the OTP. Similarly, deduplication can be decomposed as: (D1) hash data, (D2) look up the hash value in the deduplication table, (D3) update the address mapping table, and (D4) encrypt the new address mapping table entries and write back to NVM. Next, we apply optimization approaches to the decomposed sub-operations

Parallelization. We observe that there are two types of dependencies among the previously decomposed sub-operations: The *intra-operation dependency* that describes the dependency between sub-operations within one BMO, and the *inter-operation dependency* that describes the dependency between sub-operations from different BMOs. Sub-operations can happen in parallel as long as there is no incoming inter- or intra-operation dependency path from

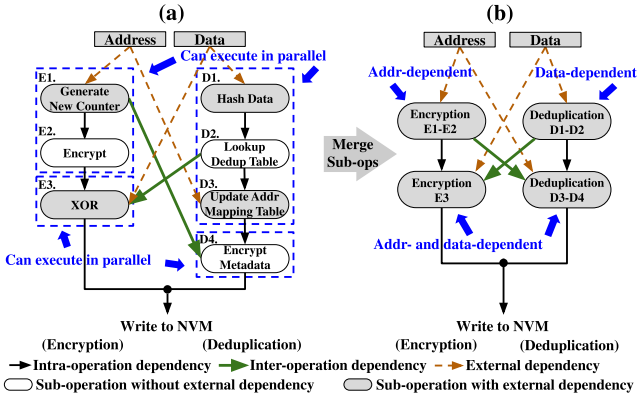

Figure 3. Optimize encryption and deduplication by: (a) parallelizing sub-operations, and (b) categorizing sub-operations based on the external dependency for pre-execution.

one set to another. Figure 3a shows the dependency graph among sub-operations in encryption and deduplication, and mark independent, parallelizable sub-operation sets with dashed lines. With sub-operations executed in parallel, the critical path is effectively reduced as the timeline in Figure 2b demonstrates.

Pre-execution. We further observe that the parallelized approach does not start any of the sub-operations until the write access reaches the memory controller, however, the inputs necessary for the sub-operations are available much earlier in practice. For example, the common crash consistency mechanism, an undo-logging transaction, creates a backup copy of the data before modifying it. Before the modification takes place, the address and data for modification are already known when creating the backup. Therefore, the BMOs for the update can be pre-executed as soon as the data and address are known. We categorize sub-operations as address-dependent, data-dependent, or both. Figure 3b groups the sub-operations according to their external dependencies. They can *pre-execute* as soon as the dependent address and/or data is available. Figure 2c demonstrates the performance improvement in an undo-logging transaction by pre-executing BMOs.

In conclusion, the decomposition of BMOs enables parallelized execution, and the pre-execution moves BMOs off the critical path.

2.2 Hardware-software Co-design

Based on the aforementioned key ideas, we propose Janus, a generic and extensible framework that parallelizes and pre-executes BMOs in NVM systems by decomposing them into smaller sub-operations. Janus enables parallelization and pre-execution in the hardware, and exposes an interface to the software for pre-execution. Figure 4 demonstrates the high-level description of Janus.

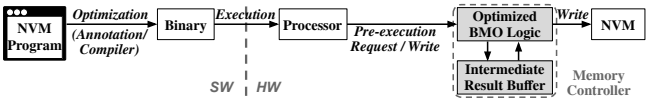

Figure 4. High-level view of Janus (HW changes are shaded).

Hardware Mechanism. The hardware of Janus ensures the correctness of pre-execution. First, Janus stores the pre-executed results in an isolated *intermediate result buffer (IRB)* in the memory controller. This way, pre-execution would not change the processor or the memory state until the actual write takes place. Second, Janus tracks the address and data of the write operations in IRB to detect and invalidate any stale pre-execution results.

Software Interface. Janus' software interface features hardware-transparency and programmability. First, Janus only exposes the external dependencies to the interface (the address and input data), such that it decouples the interface from the implementation of BMOs. Second, Janus provides a variety of functions for programmers to issue pre-execution requests. These functions are suitable for different NVM crash-consistent software mechanisms, such as undo-logging, shadow paging, etc. Janus further alleviates the programmer's burden by providing a compiler pass that automatically instruments the source code.

3 Janus Performance

We evaluate an NVM system with BMOs for encryption, integrity verification and deduplication. We compare the performance of Janus with a baseline that serializes all BMOs. Figure 5 shows the speedup of best-effort manual annotation and compiler-based automated instrumentation using the Janus software interface. We draw two conclusions from the results. First, compared to the baseline, manual and automated Janus instrumentation provides 2.35× and 2.09× speedup, respectively. The significant speedup shows Janus can effectively improve the performance. Second, the performance difference between the two methods is within 12%, as the compiler pass can leverage most opportunities for pre-execution.

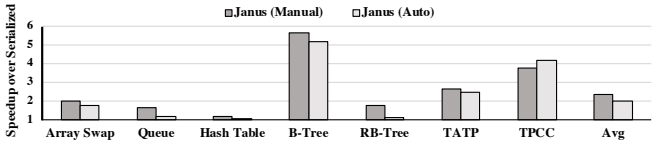

Figure 5. Speed up of Janus over the serialized design with automated and manual instrumentation..

4 Conclusions

Past research has been focused on optimizing the read latency by caching, speculation and reordering, as read latency is on the critical path in contrast to writes in conventional systems. Differently, in NVM systems, the write latency can be on the critical path due to programming patterns that guarantee crash consistency [8]. The write latency can further increase due to backend memory operations that are necessary to provide security, integrity, endurance, and bandwidth optimization in a practical NVM system. This work tackles the problem of the increased performance-critical write latency in NVM systems. This work has three major contributions: First, we take a holistic view of all integrated backend memory operations (BMOs) in NVM systems and show that it is possible to optimize them by decomposing these seemingly monolithic, dependent operations into a series of sub-operations. Second, on top of BMOs' decomposition, we propose a generic and extensible solution to optimize the sub-operations by parallelizing independent ones, and pre-execute sub-operations with resolved external dependencies. Third, we provide a software interface to leverage the pre-execution opportunities and enable programmer-transparent pre-execution with a compiler pass. By applying these optimization strategies, we exhibit substantial performance improvements. To the best of our knowledge, Janus is the first work that parallelizes and pre-executes BMOs before the actual write takes place.

References

[1] Chhabra et al. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *ISCA*, 2011.
[2] Debnath et al. ChunkStash: Speeding up inline storage deduplication using flash memory. In *ATC*, 2010.
[3] Kolli et al. Delegated persist ordering. In *MICRO*, 2016.
[4] Li et al. CacheDedup: In-line deduplication for flash caching. In *FAST*, 2016.
[5] Liu et al. Crash consistency in encrypted non-volatile main memory systems. In *HPCA*, 2018.
[6] Liu, et al. PMTest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.
[7] Moinuddin et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO*, 2009.
[8] Pelley et al. Memory persistency. In *ISCA*, 2014.
[9] Zuo et al. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *MICRO*, 2018.