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

Sihang Liu[†], Korakit Seemakhupt[†], Gennady Pekhimenko[⋆], Aasheesh Kolli[‡*], and Samira Khan[†]
[†]University of Virginia [⋆]University of Toronto [‡]Penn State University [*]VMware Research

ABSTRACT

Non-volatile memory (NVM) technologies can manipulate persistent data directly in memory. Ensuring crash consistency of persistent data enforces that data updates reach all the way to NVM, which puts these write requests on the critical path. Recent literature sought to reduce this performance impact. However, prior works have not fully accounted for all the *backend memory operations (BMOs)* performed at the memory controller that are necessary to maintain persistent data in NVM. These BMOs include support for encryption, integrity protection, compression, deduplication, etc., necessary to provide security, endurance, and lifetime guarantees. These BMOs significantly increase the NVM write latency and exacerbate the performance degradation caused by the critical write requests. The *goal* of this work is to minimize the BMO overhead of write requests in an NVM system.

The central challenge is to figure out how to optimize these seemingly dependent and monolithic BMOs. Our key insight is to decompose each BMO into a series of *sub-operations* and then reduce their overall latency through two mechanisms: (i) *parallelize* sub-operations across BMOs and (ii) *pre-execute* sub-operations off the critical path as soon as their inputs are ready. We expose a generic software interface that can be used to issue pre-execution requests compatible with common crash-consistency programming models and various BMOs. Based on these ideas, we propose Janus[1] – a hardware-software co-design that parallelizes and pre-executes BMOs in an NVM system. We evaluate Janus in an NVM system that integrates encryption, integrity verification, and deduplication and issues pre-execution requests through the proposed software interface, either manually or using an automated compiler pass. Compared to a system that performs these operations serially, Janus achieves 2.35× and 2.00× speedup using manual and automated instrumentation, respectively.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; • **Hardware → Memory and dense storage**.

[1] Janus (/'dʒeɪnəs/) is a god in Roman mythology. He has two opposite faces: one looks to the future and the other to the past.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322206

KEYWORDS

Non-volatile Memory, Crash Consistency, Parallelization, Pre-execution

ACM Reference Format:
Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-Volatile Memory Systems. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322206

1 INTRODUCTION

Non-volatile memory (NVM) technologies, such as Intel and Micron's 3D XPoint [36], offer data persistence similar to storage devices (e.g., SSD) while also delivering performance close to that of DRAM, having the potential to revolutionize persistent data management. NVMs store persistent/recoverable 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). An assortment of research efforts have sought to optimize recoverable or crash-consistent software (e.g., databases [4, 5, 22, 32, 58, 98], file systems [19, 24, 44, 93, 101, 109], key-value stores [5, 59, 102]) for NVMs.

Crash-consistent software for NVMs exhibit a unique property – they place *writes to memory on the critical path* of program execution. For conventional software, 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. However, for crash-consistent software, the order of writes to memory is severely constrained to ensure data recoverability across failures [5, 42, 54, 62, 68]. Furthermore, crash-consistent software often has to guarantee the durability of data. For example, programmers executing a database transaction expect that data modified within a transaction becomes persistent when the transaction completes. Therefore, all the writes issued to persistent data within a transaction have to reach all the way to NVM (or more specifically, the persistent domain) before the transaction completes. The x86 and ARM ISA introduced new instructions [7, 33] that programmers can use to ensure that writes reach the persistent domain to provide durability guarantees required in crash-consistent software. However, these durability guarantees imply that writes to persistent data fall on the critical path of program execution.

Placing recoverable data on NVMs not only moves writes onto the critical path, it further degrades performance by increasing the latency of write operations. The latency increases due to additional constraints on maintaining persistent data in NVMs. For example, confidentiality and integrity of data in NVM must be maintained to provide security guarantees [13, 53, 70, 71, 90, 106, 107, 112]. Furthermore, most NVM technologies suffer from a limited bandwidth and wear out after a certain number of writes, necessitating deduplication, compression, and/or wear-leveling of NVM writes [20, 21, 50, 57, 95, 113]. 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 application performance. In this work, our *goal* is to minimize the latency overhead in write operations caused by these BMOs in NVM systems.

The key challenge here is in figuring out *how to optimize these seemingly dependent, monolithic operations*. When viewed as dependent, indivisible operations, common latency optimizations (e.g., parallelization) are precluded. For example, in a system with encryption and compression, performing compression before encryption is a reasonable approach, while performing them in parallel is not, as compression can change the address mapping of the compressed data which will then invalidate the encryption output that used the old address. Our key insight to optimize these BMOs is that when they are viewed as monolithic, indivisible operations, they have to be performed in series, however, *if each BMO is viewed as a series of sub-operations, there exist many opportunities to optimize individual sub-operations across BMOs*.

By viewing each BMO as a series of sub-operations, we can optimize them for latency using two mechanisms: (i) *parallelization* of sub-operations across BMOs and (ii) *pre-execution* of sub-operations without waiting until the NVM write reaches the memory controller. First, when BMOs are viewed as a series of sub-operations, there are many opportunities for parallelization as some sub-operations across BMOs do not have any dependencies among them and can be executed in parallel. For example, even though deduplication should happen before encrypting data, the first sub-operation of deduplication calculates and looks up the hash of the data value in the write request and can be executed in parallel with the first sub-operation of NVM encryption that uses the address of the write to generate a one-time pad (details in Section 3.1).

Second, while parallelization of the sub-operations helps speeding up the BMOs, we observed that significant performance gains are still left on the table. Our key observation is 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, undo-logging [16, 24, 35, 41] is frequently used in crash consistent NVM programs. 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 during the backup step. Therefore, the BMOs for the update can be pre-executed as soon as the data and address become known at the backup step. We categorize sub-operations as address-dependent, data-dependent, or both. They can be *pre-executed* as soon as the address and/or data is available. Pre-execution of these sub-operations *decouples* them from the original write and moves them off the critical path, delivering a significant performance gain.

Based on these two key ideas, we introduce Janus, a generic and extensible framework that parallelizes and pre-executes BMOs in NVM systems by decomposing them into smaller sub-operations. It provides a hardware implementation for parallelization and pre-execution, and exposes an interface to the software to communicate the address and data values of write requests before the write reaches the memory controller. However, several challenges need to be addressed both in the design of the hardware and the software interface of Janus. The challenges in the hardware design are as follows: First, the pre-executed results of the various sub-operations for the individual writes should not change the processor or memory state until the corresponding write operation happens. Second, the pre-execution should not be dependent on any *stale* processor or memory state to maintain correctness of the results. To address these challenges, we maintain an *intermediate result buffer (IRB)* in the memory controller to store the pre-executed results and isolate them from any other processor or memory state. We also track the address and data of the write operations in IRB to detect and invalidate any stale pre-execution results.

The challenges in the software interface design are the following: First, with NVMs still being in a nascent stage of adoption, the software interface must be generic and extensible to systems with different BMOs. We only expose the address and the input data in the interface, decoupling the interface from the BMOs implemented in the system. Second, the software interface needs to be easy-to-use and applicable to different NVM-based programs. We address this issue by providing a variety of functions that are suitable for different NVM programming models. We show that the frequently used crash-consistent software mechanisms, such as undo-logging, are particularly amenable to leveraging Janus's pre-execution interface and programmers can manually insert these pre-execution requests to gain significant performance improvement. However, we also provide a compiler pass to automatically instrument the source code to alleviate programmer's burden. We describe our design and our proposed solutions to these challenges in Section 4.

The contributions of this work are as follows:

- We show that it is possible to optimize the BMOs in NVM systems by decomposing these seemingly monolithic, dependent operations into a series of *sub-operations*.
- We propose a generic and extensible solution to optimize the sub-operations by categorizing their dependencies. First, we show that independent sub-operations across BMOs can be executed *in parallel*. Second, we show that the sub-operations can be *pre-executed* as soon as their inputs are available, which moves the latency of BMOs off the critical path of the writes.
- We propose Janus, the first system that parallelizes and pre-executes BMOs before the actual write takes place. Janus provides a generic interface that decouples different BMOs at the hardware from the software.
- We exhibit the effectiveness of Janus by evaluating an NVM system that integrates encryption, integrity verification, and deduplication as the BMOs in the hardware. Our experimental results show that Janus achieves on average a 2.35× speedup while executing a set of applications where pre-execution requests are inserted manually over a baseline system that performs the BMOs serially. In comparison, instrumenting programs by our automated compiler pass achieves on average a 2.00× speedup over the serialized baseline.

Type	Backend Operation	Description	Extra Latency on Writes
Security	Encryption [13, 53, 70, 71, 77, 85, 90, 105–107, 112]	Ensures data confidentiality. Counter-mode encryption is typically used in NVM.	40 ns [53, 85]
	Integrity Verification [27, 71, 76, 84–86, 90, 91, 100, 106]	Ensures the integrity of data preventing unauthorized modification. Typically, a Merkle Tree (a hash tree) is used to verify memory integrity.	360 ns (assume 9-layer Merkle Tree) [85]
	ORAM [26, 73–75, 81, 83, 96, 97]	Hides the memory access pattern by changing the location of data after every access.	~ 1000 ns [75]
Bandwidth	Deduplication [21, 50, 57, 95, 113]	Reduce write accesses that have duplicated data to reduce the write bandwidth.	91-321 ns [113]
	Compression [1–3, 11, 12, 20, 56, 66, 67, 82]	Reduce the size of memory accesses to save the bandwidth.	5-30 ns [66, 67]
Endurance	Error Correction [8, 80, 99]	Corrects memory error. Typical solutions include error-correcting code and pointers.	0.4-3 ns [63]
	Wear-leveling [49, 55, 70]	Spreads out writes requests to even out memory cell wear-out.	~ 1 ns [70]

Table 1: A description of the existing backend memory operations in NVM systems.

2 BACKGROUND AND MOTIVATION

The new non-volatile memories (NVMs) [36, 43, 47, 103] offer persistency similar to storage devices, and performance close to DRAMs. The persistent data in NVM device can be accessed through a byte-addressable load/store interface instead of a traditional file interface.

2.1 NVM Crash Consistency

Programs can directly manage persistent data in NVM without using the conventional file system indirection for better performance. The persistent data maintained by the program is expected to be recoverable in the event of a failure. We refer to this requirement as the crash consistency guarantee. However, it is not trivial to ensure crash consistency. Due to performance optimization techniques, such as caching, buffering and reordering in modern processors, the order a write reaches NVM can be different from the program order. The program cannot recover to a consistent state if it fails to enforce a correct order. The x86 ISA has provided low-level primitives (e.g., `clwb`, `sfence` [33]) to writeback data to NVM and enforce the ordering between writes.

There have been many other works that provide crash consistency guarantees for NVM systems, including software-based solutions such as redo/undo logging [9, 10, 15, 30, 35, 37, 41, 94, 104], checkpointing [25, 39] and shadow paging [19, 65], and hardware mechanisms [38, 42, 51, 62, 72, 110]. The aforementioned methods diverge, while the key concept is similar, that is to enforce data writeback to NVM (e.g., using a sequence of `clwb; sfence`) before carrying out the next step. Let's take the commonly used undo logging method as an example. An undo log transaction typically has three steps: (1) creating a backup of the old data, (2) updating in-place and (3) committing the transaction [30, 35, 41, 62]. The backup (step 1) needs to be written back to NVM before the actual in-place update (step 2) happens; the in-place update needs to be written back before committing the transaction (step 3). Enforcing data writeback and ordering provides crash consistency guarantee, but it leaves the write latency on the critical path.

2.2 Memory and Storage Support

Using NVM as both main-memory and a storage device at the same time requires us to enforce the properties necessary for both transient and persistent (recoverable) data at the same time. First, due to its non-volatility, data remains on NVM even after being powered off. Therefore, attackers with physical access to the NVM device can potentially access data on NVM. To ensure the confidentiality of data, recent works encrypt data on NVM [13, 53, 70, 71, 90, 106, 107, 112]. Attackers can also tamper with the data on NVM. To ensure the integrity of data, recent works also use integrity verification techniques [71, 90, 106]. Second, NVM has a limited lifetime [36, 47]. A practical NVM system needs to overcome the limitation in lifetime. Prior works have proposed wear-leveling [49, 55, 70] and error correction [8, 80, 99] techniques to mitigate the lifetime issue. Third, NVM has a limited write bandwidth compared to that of read [69, 89, 108]. A common way of overcoming the write bandwidth is to reduce the write traffic using compression [1–3, 11, 12, 14, 20, 29, 56, 66, 67, 82] or deduplication [21, 50, 57, 95, 113] techniques. We summarize the existing flavors of memory and storage support for NVM in Table 1. These memory and storage supports happen in the background, at the memory controller and are transparent to the processor, therefore, we collectively refer to them as *backend memory operations (BMOs)*. In conventional programs, reads are on the critical path of execution, therefore, these BMOs optimize for read latency. For example, counter-mode encryption [23] allows the decryption to happen when the data is being read from memory; the integrity verification [27, 71, 76, 84–86, 90, 100, 106] uses caching to reduce the number of verification steps. There have been works that integrate one or multiple of these backend operations, but the integration is BMO-specific [90, 106, 113]. However, systematically integrating the BMOs in NVM systems and optimizing them for latency has been largely unstudied.

2.3 Challenge: Performance Overhead

While the BMOs make NVMs more secure and robust, they add extra latency to writes, as they all require certain computation or cache lookup before actually performing the write access. To maintain the correctness, BMOs should follow certain dependencies among themselves. For example, a system with encryption, deduplication and integrity verification, these BMOs should happen in the order of deduplication, encryption, and integrity information update during a write access. Deduplication first tells whether the write is necessary or not. Then, the encryption engine encrypts the data if the write is not a duplicate. Finally, the integrity mechanism (e.g., Bonsai Merkle Tree [76]) updates the message authentication code (MAC) and hash tree to protect the encrypted data and counters. The ordering constraints serialize the latency from different BMOs, which is in the order of hundred nanoseconds.

Figure 1 demonstrates the latency breakdown of a write access. We assume a system with the Intel Asynchronous DRAM Refresh (ADR) technique [60] that ensures the write queues are in the persistence domain. Therefore, writes to NVM become persistent (or non-volatile) as soon as they are placed in the write queue in the memory controller, as the ADR technique can flush the write queue to NVM in case of a crash. Without any BMOs (Figure 1a), only the writeback from the cache hierarchy to the memory controller is exposed on the critical path, which typically takes around 15 ns in modern processors (e.g., Intel i7 processor [48]). The subsequent operations in the memory controller and the actual NVM device write operations do not contribute to the critical write latency. However, with BMOs (Figure 1b), both the writeback and the BMO latency are exposed on the critical path, as until BMOs are completed, the write cannot be placed in the write queue and hence cannot be considered persistent. As these BMOs add extra hundreds of nanoseconds of latency, the critical latency increases by more than 10 times.

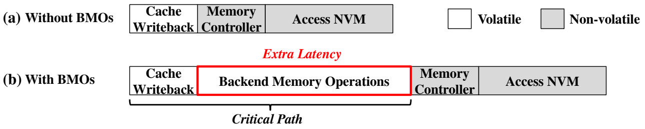


Figure 1: Write latency (a) without and (b) with BMOs.

3 OVERVIEW

In this section, we describe our key ideas in optimizing the BMOs and providing the crash consistency guarantee.

3.1 Key Ideas

The BMOs need to execute in series if we regard them as monolithic, indivisible operations. However, we observe that BMOs can be further decomposed into smaller *sub-operations*. We first demonstrate decomposing two commonly used BMOs in NVM systems: counter-mode encryption [53, 71, 77, 85, 105–107, 112] and deduplication [21, 50, 57, 95, 113]. Next, we take a two-pronged approach to minimizing their latency: (1) parallelize BMOs as much as possible, and (2) pre-execute BMOs to move their latency off the critical path.

Decomposing BMOs. The counter-mode encryption [23] is an efficient encryption scheme that indirectly encrypts data blocks using unique counters. Its hardware implementation typically encrypts a unique counter together with the address of the data block into a bitstream called one-time padding (OTP), and then it XORs this bitstream with the data block to complete the encryption. To accelerate the read access, the hardware mechanism buffers these counters in an on-chip counter cache so that decryption can begin without waiting for data to be fetched from NVM, reducing the read latency. During a write access, it performs three sub-operations : (E1) generate a new counter, (E2) generate the one-time padding: `OTP = En(counter|address)`, and (E3) encrypt data with an XOR operation: `EncData = OTP` ⊕ `Data`. As encryption begins only when both the data and address of the write access reaches the encryption engine, the whole latency is added to the write access.

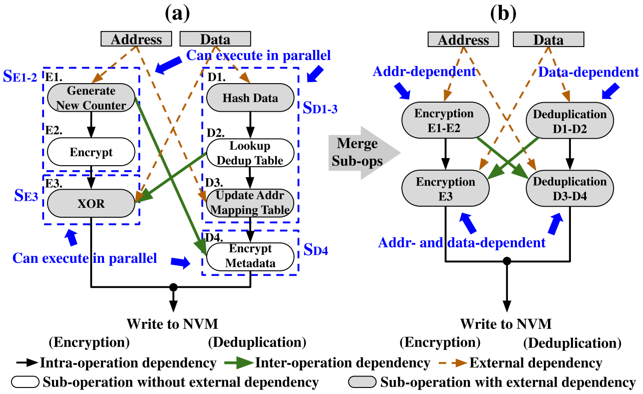


Figure 2: Optimize encryption and deduplication by: (a) parallelizing sub-operations, and (b) categorizing sub-operations by external dependency for pre-execution.

On the other hand, the deduplication mechanism detects whether writes contain a value that already exists in memory and cancels the write if a duplicated value is found. The hardware mechanism maintains a deduplication table that stores the hashes (fingerprints) of existing data blocks to detect duplicates, and an address mapping table to redirect the writes to the existing copy of data in memory. During a write access, a deduplication operation consists of four sub-operations: (D1) hash data, (D2) lookup the hash value in the deduplication table, (D3) update the address mapping table, and (D4) encrypt the new address mapping table entries and writeback to NVM. To integrate encryption and deduplication, We assume a scheme similar to DeWrite, where the counter and deduplication address mapping co-locate in the same metadata entry [113]. Next, we describe the parallelization and pre-execution of the decomposed BMOs.

Parallelization. We observe that there are two types of dependencies between the previously decomposed sub-operations: *intra-operation dependency* and *inter-operation dependency*. Intra-operation dependency describes the dependency between sub-operations within one BMO, while, inter-operation dependency describes the dependency between sub-operations between different BMOs. We demonstrate the dependencies as a dependency graph in Figure 2a. Two sets of sub-operations can happen in parallel as long as there is no incoming inter- or intra-operation dependency path from one set to another. Formally, let a node of sub-operation be Op, a set of Op be S, and a path from Op_1 to Op_2 be $Op_1 \rightsquigarrow Op_2$, S_1 and S_2 can execute in parallel, i.e., $S_1 \parallel S_2$, if and only if $\forall Op_1 \in S_1$ and $\forall Op_2 \in S_2, \nexists Op_1 \rightsquigarrow Op_2 \wedge \nexists Op_2 \rightsquigarrow Op_1$.

Next, we apply our theory to the example in Figure 2. We mark the intra- and inter-operation dependencies with black and green edges, respectively. The intra-operation dependencies in each BMOs follows the order of steps. And, there are two inter-operation dependencies: D4 depends on E1 as the address mapping co-locates with counter, and E3 depends on D2 as the memory controller cancels duplicated writes. According to these dependencies, we can circle out the sub-operations that are independent in each backend operations (blue boxes): S_{E1-2} and S_{D1-3} are independent, and S_{E3} and S_{D4} are independent. Therefore, they can be executed in parallel. By parallelizing groups of sub-operations, we reduce the serialization overhead. Figure 3a shows the execution timeline of an undo-logging transaction that consists of three steps (each with NVM writes): backup, update, and commit. In the serialized approach, deduplication and encryption operations are serialized for each step of the transaction. However, by parallelizing independent sub-operations, the execution latency of the three steps in an undo-logging transaction can be reduced, as shown in Figure 3b.

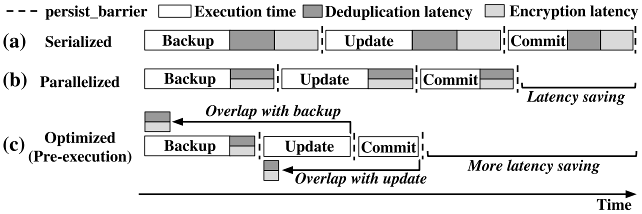


Figure 3: Timeline of an undo log with (a) serialized, (b) parallelized and (c) pre-executed BMOs.

Pre-execution. So far, we have exploited the parallelism between BMOs by decomposing them into sub-operations. We further observe that the BMOs process two types of external inputs: the data and address of a write. These external inputs are different from any intermediate inputs generated and used between different sub-operations of the same BMO. Accordingly, apart from the inter- and intra-operation dependencies introduced earlier, external inputs introduce a new dependency, *external dependency* (marked as yellow arrow), that takes into account the external input of each sub-operation. A sub-operation is dependent on an external input if there exists an external dependency edge from the input. We merge nodes without any external dependency (marked in white) with their preceding nodes with external dependency (marked in gray). Figure 2b shows the simplified graph after the merge operation. A set of merged sub-operations is externally dependent on an external input if there exists an external dependency edge from the input or a path that indirectly connects the input to one/some of its sub-operation node (via inter- and intra-operation dependency edges). Formally, let the set of merged sub-operation nodes be S, an input (address or data of a write) be In, then S has an external dependency to In if and only if $\exists Op \in S, In \rightsquigarrow Op$.

Based on the type of external input, we categorize sub-operations into three types: address-dependent, data-dependent, and address- and data-dependent. In the example of Figure 2b, E1-E2 are address-dependent, D1-D2 are data-dependent, and E3 and D3-D4 are both address- and data-dependent. The external dependency implies that as soon as the external inputs are available, the BMOs can start execution even before the actual write access reaches the memory controller. Next, we use a code example to explain how we can exploit the opportunity of pre-executing BMOs.

Example. Figure 4 shows an example of updating an array using an undo-logging transaction that follows three steps: backup the old data, perform the in-place update, and commit the update. In this example, the address and data for the in-place update are known before the backup step (at line 1). Similarly, the address and data for the commit (validate the in-place update) are known before the commit step (at line 5). Therefore, the pre-execution of the BMOs for the in-place update and the commit steps can be overlapped with the previous steps of the undo-logging transactions, moving them off the critical path. Figure 3c shows the timeline of this pre-execution. By pre-executing the BMOs that have already been parallelized, we can gain a significant speedup.

```
1 void arrayUpdate(int index, item_t new_val) {
2   // backup old value
3   backup(index);
4   // in-place update
5   update(index, new_val);
6   // commit undo-logging transaction
7   commit(index);
8 }
```

The *address* and *data* for update are known

The *address* and *data* for commit are known

Figure 4: An example of pre-executing BMOs in an undo-logging transaction.

3.2 Requirements

Pre-executing the BMOs before the actual write happens provides a significant benefit. However, the pre-execution should not affect the correctness of the normal execution. We summarize the requirements on the *hardware* support for pre-execution as follows:

1. Does not affect processor state. The pre-execution should not affect the processor or memory state, i.e., it should not change the data or metadata in memory, cache or register files.

2. Invalidates stale pre-execution results. The pre-execution should not be dependent on a stale processor or memory state. i.e., if the processor or memory state used in the pre-execution has been modified before the actual write access, the pre-execution result becomes invalid.

On the other hand, we need to provide an interface to let the software leverage the hardware support. We summarize the requirements on the *software* interface as follows:

3. Extensible interface. The software interface needs to be generic and extensible to systems with different BMOs, i.e., programs developed with the same interface should be compatible even though the BMOs change in the hardware.

4. Programmable. The software interface needs to be easy-to-use for different programming models that ensure crash consistency (e.g., undo /redo/shadow logging), and should abstract away the memory layout.

Section 4.3 presents our solution to meet the two requirements for the hardware support, and Section 4.4 presents our software support that meets the two requirements on the software interface.

4 JANUS

In this section, we first describe the high-level design of our proposed system and then provide the details of the hardware mechanism (Section 4.3) and software support (Section 4.4).

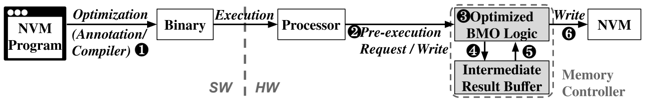

NVM Program
Optimization (Annotation/ Compiler) ❶
Binary
Execution
Processor
SW
HW
❷Pre-execution Request / Write
❸Optimized BMO Logic
❹
❺
Intermediate Result Buffer
Write ❻
NVM
Memory Controller

Figure 5: High-level of Janus (HW changes are shaded).

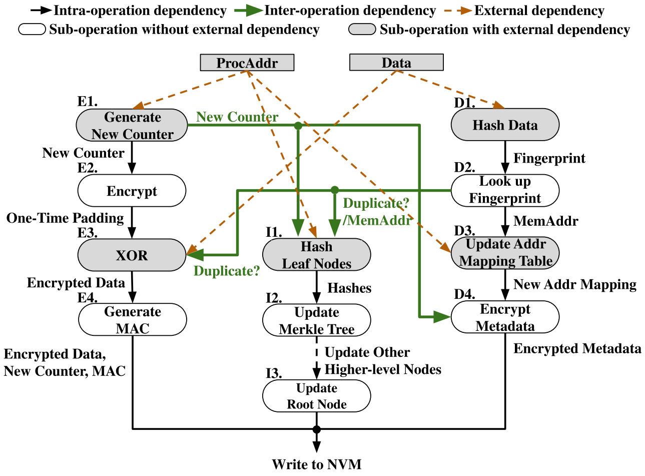
4.1 High-level of Janus

The *goal* of this work is to reduce the overhead of BMOs in write accesses using a software-hardware co-design. Figure 5 shows an overview of Janus. On the software side, programmers annotate the NVM programs using our software interface (step ❶). To further reduce the programming effort, we provide a compiler pass that automatically instruments the program. We present the use of Janus interface in Section 4.4 and the design of our compiler pass in Section 4.5. On the hardware side, the processor issues pre-execution requests to the memory controller during the execution of the annotated programs (step ❷). The processing of pre-execution requests consists of two parts. First, the *optimized BMOs logic* of Janus executes the sub-operations of the requests in parallel (step ❸). Then, it stores the temporal results in the *intermediate result buffer* (step ❹). When the actual writes arrive at the memory controller, they do not need to go through the BMOs, instead, they use the pre-executed results from the *intermediate result buffer* (step ❺) and complete the access to NVM (step ❻).

In the rest of this section, we first introduce the integration of three common BMOs in NVM systems. Then, we present Janus hardware details in Section 4.3, and the software interface in Section 4.4. Finally, we discuss the solutions to potential exceptions when integrating Janus in real systems in Section 4.6.

4.2 Backend Memory Operations

BMOs are integrated into memory and storage systems for different purposes, such as ensuring confidentiality and integrity, improving the lifespan, mitigating the write bandwidth limitation, etc. If we treat each of them as an entity, it seems difficult to execute them in parallel as the output of one operation flows into another. However, by breaking them down into smaller steps, we can leverage the underlying parallelism. There has been a myriad of BMOs, as shown in Table 1. To better demonstrate our idea, we take the two BMOs introduced in Section 3: encryption and deduplication, together with another popular BMO, integrity verification. Figure 6 presents the break down of the three BMOs.



(a) Encryption (b) Integrity Verification (c) Deduplication

Figure 6: The dependency graph of backend operations.

As we already described the operations in encryption and deduplication in Section 3, here we introduce the steps in an integrity verification technique. The Bonsai Merkle Tree [76] is an integrity verification scheme designed for memory encrypted under counter-mode. The leaf nodes of the tree are counters and the intermediate nodes are hashes of their child nodes. Therefore, the root hash is essentially the hash of all leaf nodes. Keeping the root hash in a secured non-volatile register ensures the integrity of the entire memory [76, 106]. Each data block is protected by a message authentication code (MAC) that consists of the encrypted data and its counter, i.e., `MAC = Hash(EncData, Counter)`. During a read accesses, the integrity verification mechanism compares the root hash computed from the counter read from memory with the existing root to verify the integrity. During a write access, this mechanism updates the integrity information in the following steps (Figure 6b): First, the integrity verification mechanism computes the hash of leaf nodes (step I1), and then it keeps computing higher-level intermediate nodes *all the way to the root* (step I2-I3). The intra-operation dependencies between these steps are indicated by black arrows. In this mechanism, the write access includes this long latency of hashing. For example, if we assume each intermediate node is the hash of eight lower-level nodes, then the height of the Merkle Tree is 9 in a system with only 4GB NVM, resulting in a 360 ns latency for each write.

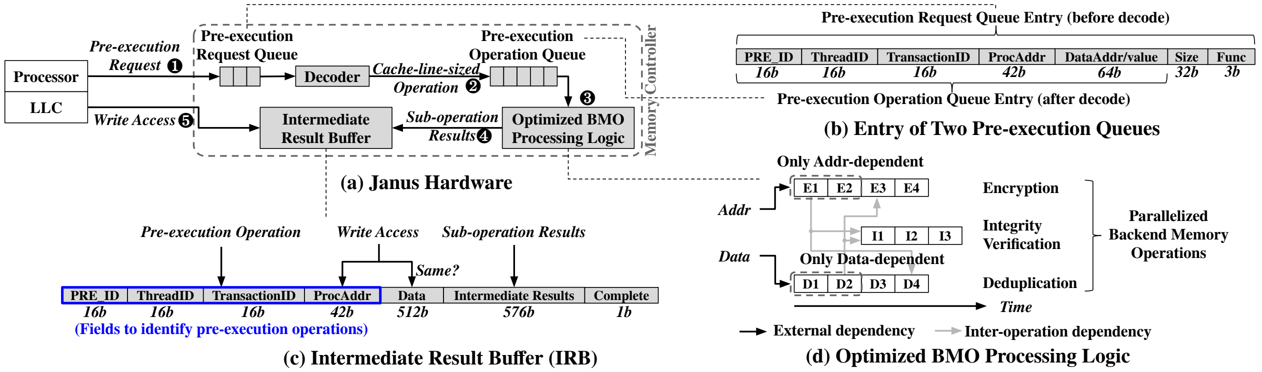
The integration of integrity verification with the other two BMOs introduces an extra step: the encryption operation needs to compute the MAC for Integrity verification before writing data back to NVM (step E4). Similar to the prior work, DeWrite [113], the Merkle Tree in our mechanism is built on the co-located address mapping and counter so that the metadata storage can be minimized. Therefore, the integration also introduces new inter-operation dependencies (green edges). The integrity verification support needs to take the latest counters or the remapping address (if duplicate) to update the Merkle Tree. Thus, step I1 depends on E1 and D2 (edge E1→I1 and D2→I1). To mitigate the extra latency on writes, we first apply the rule for *parallelization* (Section 3.1). Based on the intra-operation dependency edges, three sets of sub-operations E3-E4, I1-I3 and D3-D4 can execute in parallel as there is no edge between any pair of these sub-operation sets. Then, we apply the rule for *pre-execution*. We mark the nodes with external-dependency in gray. After merging the nodes without external-dependency (marked in white) to the ones with external-dependency, we find out that E1-E2 are address-dependent, D1-D2 are data-dependent, and the rest are both address- and data-dependent. These regions can be pre-executed once the dependencies are resolved. Next, we describe the hardware support that enables pre-execution.

4.3 Hardware Support

In this section, we describe the hardware support that meets the two requirements that we outlined in Section 3.2.

4.3.1 *Hardware Support for Pre-execution.*

Does not affect processor state. The pre-execution of BMOs should not change the processor or memory state. Therefore, Janus stores the temporary results in an *Intermediate Result Buffer (IRB)*. Figure 7c shows the fields in each IRB entry (and their sizes). IRB needs to support two basic functionalities: identify different pre-execution requests, and store and provide the pre-executed results. First, Janus uses a `PRE_ID` for each request in order to make sure the pre-execution requests are unique. Considering that different threads can be executing the same program and can have the same `PRE_ID`, each entry contains another field, `ThreadID` that distinguishes the requests from different threads. As recent works have proposed deferred commit [28, 41], a transaction may not



(a) Janus Hardware

(b) Entry of Two Pre-execution Queues

(c) Intermediate Result Buffer (IRB)

(d) Optimized BMO Processing Logic

Figure 7: Detailed hardware mechanism of Janus.

have all the updates written back to NVM before the transaction completes, causing more than one transactions with the same PRE_ID to co-exist. Each IRB entry further contains another field, TransactionID, that distinguishes pre-execution requests across different transactions. These three fields (PRE_ID, Thread_ID and Transaction_ID) are assigned by the software interface which we will introduce in Section 4.4. Using these fields, together with the physical address of the write (ProcAddr), Janus can uniquely identify pre-execution requests. Second, Janus needs to buffer pre-execution results for the actual write access when it arrives at the memory controller. The IntermediateResults field stores the intermediate results at cache line granularity. Considering the actual write access can arrive before the BMOs completes, a complete bit indicates whether all BMOs have completed or not.

Invalidates stale pre-execution results. The pre-execution becomes invalid if the memory or processor state it depends on has changed. Therefore, Janus invalidates the intermediate results from pre-execution by detecting any changes to the input memory or processor state. We summarize the potential cause of invalidation as the following two: (1) The input dependent data can be modified after the program issues the pre-execution request (e.g., due to cache line sharing, eviction or buggy program that modifies the input data for pre-execution). In order to detect any stale value used in pre-execution, Janus keeps a copy of the data value that has been used for pre-execution in the Data field of IRB entry. When the write access arrives, IRB compares the data from the write access with this copy. If they are the same, the write access can safely use the intermediate results and complete the write to NVM. Otherwise, data-dependent sub-operations have to be reprocessed using its new data. (2) Apart from the actual writes, pre-execution results buffered in the IRB may also depend on metadata structures employed by the BMOs. If these metadata structures are modified in such a way that they invalidate any prior pre-executed sub-operations, the pre-executed results must also be invalidated in the IRB. For example, pre-executing a deduplication sub-operation might identify that the current write (say to location B) is a duplicate of some prior write (say to location A). Therefore, the IRB stores the pre-execution result that the write to B is a duplicate of the value at A. However, before the pre-execution result is consumed, if an intervening write to location A changes the value of location A, then the pre-execution result in the IRB will be invalidated. In Janus, we extend BMOs to ensure that metadata changes trigger an IRB lookup and invalidates any stale pre-execution results.

4.3.2 *Hardware Integration.*

Figure 7a shows the detailed hardware mechanism. First, the processor sends the requests to a *Pre-execution Request Queue* that buffers the requests (step ❶). It supports two types of requests: (1) requests that start immediately, and (2) requests that are buffered in the queue and wait until the hardware receives an explicit start command. In the latter case, the requests with coalescing addresses will be merged within the queue for better efficiency (details in Section 4.4). Second, a *decoder* decodes the request from the *Pre-execution Request Queue* into cache-line-sized operations and sends them to a *Pre-execution Operation Queue* (step ❷). Therefore, the pre-execution operations after the decoder stage all have one-cache-line granularity. Note that systems that perform BMOs at larger granularities (e.g., 256B block for deduplication) can also be supported by modifying the decoder. Figure 7b shows the fields in both queue entries. Third, the *Pre-execution Operation Queue* sends the decoded operations to the *Optimized BMO Processing Logic* (step ❸), and at the same time, it creates a new IRB entry. Figure 7d shows the execution flow of the Optimized BMO Logic (correspond to the design in Figure 6), where independent sub-operations can be executed in parallel and can be pre-executed if their external dependency is resolved. Finally, the *Optimized Backend Operation Logic* writes the pre-execution results to the previously created *Intermediate Result Buffer* entry (step ❹), which keeps track of the pre-execution at a cache line granularity, i.e., each entry in the buffer keeps the pre-execution result of one cache line. When the actual write access arrives, it can lookup the intermediate results from the IRB using its ProcAddr (step ❺). Note that the IRB, the *Pre-execution Request Queue*, and the *Pre-execution Operation Queue* have a fixed number of entries. If the buffer/queue is full, it drops newer requests. We discuss the software interface for our hardware mechanism in Section 4.4, and discuss the system integration and exception handling in Section 4.6. We present the hardware overhead in Section 5.2.7.

Apart from the performance overhead, maintaining crash consistency is another issue as the BMOs have their own metadata. The *unreconstructable* metadata structures, the ones that cannot be rebuild using the data in NVM, need to be kept up to date in NVM when data gets updated. In the BMOs we have considered, there are three structures that cannot be reconstructed: counters for encryption, the deduplication address remapping table, and the root of the Merkle Tree. A recent work [53] has proposed counter-atomicity that atomically writes back both the encrypted data and its counter to NVM in an encrypted NVM system. In this work, we extend this

Type	Function	Description
Common	PRE_INIT(pre_obj* obj)	Initialize an pre_objwith a unique PRE_ID, the current ThreadID and TransactionID.
Immediate Execution	PRE_BOTH(pre_obj* obj, void* addr, void* data, int size)	Pre-execute all sub-operations.
	PRE_ADDR(pre_obj* obj, void* addr, int size)	Pre-execute address-dependent sub-operations.
	PRE_DATA(pre_obj* obj, void* data, int size)	Pre-execute data-dependent sub-operations.
	PRE_BOTH_VAL(pre_obj* obj, void* addr, int data_val)	Use an integer as the data. Pre-execute all sub-operations.
Deferred Execution	PRE_BOTH_BUF(pre_obj* obj, void* addr, void* data, int size)	Buffer pre-execution for all sub-operations.
	PRE_ADDR_BUF(pre_obj* obj, void* addr, int size)	Buffer pre-execution for address-dependent sub-operations.
	PRE_DATA_BUF(pre_obj* obj, void* data, int size)	Buffer pre-execution for data-dependent sub-operations.
	PRE_START_BUF(pre_obj* obj)	Start executing buffered pre-execution requests for pre_obj.

Table 2: Software interface of Janus for pre-execution.

atomicity to a more general metadata atomicity that writes back all unreconstructable metadata to NVM atomically, ensuring that the processor can still read correct data during recovery. Note that the root of the Merkle Tree is typically protected by a non-volatile register in the secured processor [76, 106]. Therefore, it does not require any metadata atomicity. In order to reduce the atomicity overhead, Janus also follows the selective method on atomicity proposed by prior work [53], where only the writes that can immediately affect the crash consistency status (e.g., write that commits a transaction) requires metadata atomicity.

4.4 Software Support for Optimization

Extensible. BMOs for NVM can vary in different systems. A program developed with a software interface should be compatible with systems using different BMOs, without a need for additional software modifications. Therefore, Janus only exposes the two fundamental external dependencies to the software: the address and data of the write access. Table 2 shows the software interface for pre-executing BMOs. Next, we explain how Janus provides an interface that can adapt to different NVM programming models.

Programmable. Janus provides a structure, pre_obj, that has its unique PRE_ID and keeps track of the current ThreadID and TransactionID. These three elements enables the hardware to distinguish different pre-execution requests. To perform pre-execution on a object stored in NVM, the programmer first needs to create a pre_obj and initialize it using PRE_INIT. Then, Janus provides two types of interfaces that enables programmers pre-execute the data structure that have either its address or data value available before the actual write to NVM. Functions are identified by the field Func in the *Pre-execution Request Queue* entry (Figure 7b).

The first type of function is for *immediate execution*. Janus provides three functions: PRE_BOTH, PRE_ADDR and PRE_DATA. Programmers can use them according to the availability of the dependent address or data. The input addresses are all virtual addresses from the program, and will be translated to processor-visible physical address (ProcAddr). Upon calling these functions, the pre-execution requests will be sent to the backend operation right away. Janus provides a special function, PRE_VAL, that takes a 64-bit integer value instead of the pointer to data. This function is designed to pre-execute transaction commit operations that typically set a valid bit or switches a pointer.

The second type is for *deferred execution*. Janus allows programs to *buffer* pre-execution requests using a class of functions that ends with the BUF suffix. These buffered requests can be executed together with the PRE_START_BUF function. Deferred execution provides more flexibility in scheduling the requests if the data structure to be pre-executed does not operate on a huge chunk of data, rather manipulates several elements in the structure separately. By buffering the requests for each element, the pre-execution buffer can merge the inputs before execution, leading to better efficiency.

Guideline for using the software interface. The hardware of Janus prevents misuse of the interface from causing any correctness issue. However, improperly placed Janus functions can lead to slowdown due to unused or discarded pre-execution. To effectively use the software interface, programmers need to be aware the following issues: (1) Between the pre-execution function call and the actual write operation, there should not be any update to the same location, or to the conflicting cache line. Although the underlying hardware mechanism can detect and fix such violations, the misuse can lead to a slowdown. (2) When using PRE_DATA alone, the data block must be cache-line-aligned (e.g., using alignment malloc). As the hardware tracks the pre_obj at cache line granularity, it is impossible to determine whether the data block shares its cache line with other data blocks without the address. Therefore, it is better to call pre-execution functions with PRE_ADDR or wait until both address and data become known if the programmer is not certain about the alignment. (3) As it takes a significant amount of time to execute the backend operations, it is better to place the pre-execution function calls sufficiently far away from the actual write. A simple and reasonable way to insert the pre-execution function call for a write request is to find the last update at that location and to insert that function right after that update.

Examples. Figure 8a shows an example using the immediate-execution interface. First, we observe that the value used in the update operation (val) is available right after the function call (assuming nodes are cache-line-aligned). Therefore, a PRE_DATA function can be placed at line 4 to pre-execute the data-dependent BMOs. Then, we observe that the program uses an undo log to back up the node before modification (line 11). Therefore, it is possible to issue a pre-execution request for the address-dependent BMOs by inserting a PRE_ADDR function at line 8. Using these two pre-execution requests, we move the latency from BMOs off the critical path of the actual write (line 11). Figure 8b shows an example of using the deferred-execution interface. The address and data for updates to field1 and field2 are already available after line 4. However, the two separate updates can be sharing a cache line (assuming the fields are not cache-line-aligned). The safe way to avoid invalidation of requests is to use the PRE_BUF function to buffer the pre-execution requests for each field and let them coalesce in the *Pre-execution Request Queue*. Then, placing a PRE_START_BUF function afterward will trigger the execution (line 10).

```
 1 void updateTree(int key, item_t val) {
 2  pre_t pre_obj;
 3  // assume val is cache-line-aligned
 4  PRE_DATA(&pre_obj, &val,
 5                    sizeof(item_t));
 6  // find tree node with key
 7  node* location = find(key);
 8  PRE_ADDR(&pre_obj, location,
 9                    sizeof(item_t));
10  // add old val to undo log
11  undo_log(location);
12  // update val
13  location->val = val;
14  // writeback updates
15  clwb(&location->val, sizeof(item_t));
16  sfence();
17  ...
18 }
```

(a)

```
 1 void updateTable(int id, item_t val1,
 2                        item_t val2) {
 3  // lookup entry location
 4  entry* location = tableLookup(id);
 5  pre_t pre_obj;
 6  PRE_BOTH_BUF(&pre_obj, &location->field1,
 7              &val1, sizeof(item_t));
 8  PRE_BOTH_BUF(&pre_obj, &location->field2,
 9              &val2, sizeof(item_t));
10  PRE_START_BUF(&pre_obj);
11  // backup old entry
12  undo_log(location);
13  // update fields
14  location->field1 = val1;
15  location->field2 = val2;
16  // writeback updates
17  clwb(location, sizeof(entry));
18  sfence();
19  ...
20 }
```

(b)

Figure 8: Two NVM transactions optimized by Janus.

4.5 Compiler Support

The software interface of Janus is easy-to-use, but it still requires a good understanding of the program. To alleviate programmer's effort, we provide a compiler pass that automatically instruments the program with Janus functions.

4.5.1 Compiler Design. We develop our compiler pass on LLVM 7.0.0 [46]. The compiler pass analyzes and instruments the intermediate representation (IR) of the source code in the following steps. (1) The first step is to locate the blocking writeback operations (e.g., a clwb() followed by an sfence()), as these operations are responsible for moving the write latency on the critical path. (2) The next step is to perform a dependency analysis on the writeback objects. Our compiler pass takes two different analysis approaches for the data and the address of these objects. For address, it tracks dependencies of the address generation IR instructions of the object; for data, it tracks the modification to the memory address of the object. (3) The final step is to inject Janus functions (PRE_DATA and PRE_ADDR). The compiler pass injects them as far away from the actual writeback as possible in order to provide a better performance benefit. The injection approach is different for address and data. For address, it hoists the previously tracked dependent IR instructions for address generation to the beginning of the function, and places a PRE_ADDR function after the address generation is complete; for data, it places a PRE_DATA function between the last two updates on the object. It inserts the function as close as possible to the *pre-last* update using the data value assigned by the last update. Note that for both data and address, if the writeback operation depends on a conditional statement (e.g., if/else), our pass conservatively inserts the pre-execution function under the same conditional statement to avoid introducing potentially useless pre-execution requests. We evaluate our compiler pass in Section 5.2.3 and compare it with our best-effort manual instrumentation.

4.5.2 Limitations. The compiler pass has the following limitations. First, it conservatively injects pre-execution functions within the same function as the writeback operation to guarantee correctness. Second, it can only inject pre-execution functions where both data and address dependencies can be resolved in compile time. For example, when a loop writes back an array of data, our pass cannot inject pre-execution for writebacks in the loop due to the lack of runtime information about the loop. Third, due to the lack of dynamic memory information, our compiler pass does not handle cache line sharing. We discuss future works that can mitigate these limitations in Section 6.

4.6 Real-World Considerations

This section described various scenarios that might arise while integrating Janus into real systems.

Unused pre-execution result. A buggy program can issue useless pre-execution requests without issuing a subsequent write access that uses the pre-executed result. Therefore, a useless pre-execution result can get stuck in the IRB. Janus takes a twofold approach to solve this problem: (1) Add an *age register* to each IRB entry, and discard an entry when the *age register* reaches its maximum lifetime. (2) Clear all entries belonging to a certain thread when that thread terminates.

Unused pre-execution request. A buggy program can also issue buffered pre-execution requests without starting their execution with a PRE_START_BUF function, causing congestion in the *Pre-execution Request Queue*. Janus solves this problem by using a fixed size FIFO for this queue. When the queue is full, it discards the *buffered* pre-execution requests at the top of the queue to make space for the new requests. Note that discarding pre-execution requests will never cause any correctness issue, but can result in missed opportunities to improve performance.

Memory swap. OS can swap memory to the disk and swap it back later. In this case, the physical address (ProcAddr) can be different. Our solution is to let the memory controller clear out all *Intermediate Result Buffer* entries that belong to the address range that will be swapped out.

5 EVALUATION

5.1 Methodology

Processor	Out-of-Order, 4GHz
L1 D/I cache	64KB/32KB per core, private, 8-way
L2 cache	2MB per core, shared, 8-way
Counter cache	512KB per core, shared, 16-way
Merkle Tree cache	512KB per core, shared, 16-way
Pre-exec. Request Queue	16 entries per core, shared
Pre-exec. Operation Queue	64 entries per core, shared
BMO Units	4 units per core (execute 4 BMOs in parallel), shared, perform at cache-line granularity
Intermediate Result Buffer	64 entries per core, shared
Memory	4GB PCM, 533MHz [40, 42, 53], $t_{RCD}/t_{CL}/t_{CWD}/t_{FAW}/t_{WTR}/t_{WR}$ = 48/15/13/50/7.5/300 ns [103]
Backend Operation Latency	AES-128 (Encryption): 40 ns [53, 85], SHA-1 (Integrity): 40 ns [85], MD5 (Deduplication): 321 ns [113]

Table 3: System configuration.

We model and evaluate an NVM system that has three BMOs: encryption, integrity verification and deduplication (introduced in Section 3.1 and 4.3) using the cycle-accurate Gem5 simulator [6].

The system configuration is shown in Table 3. The memory system is backed by Intel's ADR [60] support where all write accesses accepted by the write queue can drain to NVM in case of a failure. The encryption and deduplication mechanisms follow a recent work [113], where the encryption counter and the deduplication address mapping table share the same metadata entry to minimize the storage overhead, i.e., if data is duplicated, the metadata entry stores the address mapping, otherwise, it stores the counter. The Merkle Tree is built on the co-located counter or deduplication address mapping to protect the integrity of both. We use selective counter-atomicity [53] to ensure crash consistency of counter-mode encryption, and extend this support to the other unreconstructable metadata, including the address remapping table in the deduplication mechanism and the message authentication code (MAC) in the integrity verification mechanism. We store the root of the Merkle Tree in a non-volatile register in the secured processor, as proposed in previous works [76, 106]. Similar to the ratio in prior deduplication works [95, 113], our main results use a deduplication ratio of 0.5. We also present the performance in other deduplication ratios in Section 5.2.4. We evaluate and compare two system designs:

1. Serialized: Serialized backend operations.

2. Janus: Pre-execute the parallelized BMOs.

Our evaluation uses seven NVM-optimized transactional workloads (listed in Table 4), which are inspired by recent works [28, 42, 53], We evaluate the serialized design with the original program that only supports metadata atomicity. Then we manually instrument Janus primitives to evaluate Janus. We compare the manual and automated instrumentation through our compiler pass in Section 5.2.3.

Workload	Description
Array Sway	Swap random items in an array
Queue	Randomly en/dequeue items to/from a queue
Hash Table	Insert random values to a hash table
RB-Tree	Insert random values to a red-black tree
B-Tree	Insert random values to a b-tree
TATP	Update random records in the TATP benchmark [64]
TPCC	Add new orders from the TPCC benchmark [92]

Table 4: Evaluated workloads.

5.2 Results

This section presents the results of our evaluation that compares the performance of the two design points. The workloads are single-threaded unless explicitly mentioned.

5.2.1 Single- and Multi-core Performance. In this experiment, we test the single- and multi-core performance of our design. Figure 9 presents the speedup of Janus. Janus provides on average 2.35 ∼ 1.87× speedup in 1∼8-core systems, respectively, over the serialized baseline system. We observe three broad trends from our results. (1) The speedup from pre-execution decreases as the number of cores increases because the memory bus contention increases when there are more threads executing in parallel, leading to a higher queuing latency in the memory controller. As a result, the ratio of BMO overhead decreases and the benefit of pre-executing BMOs also decreases. (2) The gain from pre-execution depends on the characteristics of the workloads: The speedup in B-Tree, TATP and TPCC is higher than that in Hash Table and RB-Tree. The reason is Hash Table and RB-Tree first look up the update location and then perform the update at that location. As a result, the address-dependent pre-execution request has a smaller window to execute and many times cannot complete before the actual write arrives. (3) Parallelization delivers a lower speedup compared to pre-execution because parallelization only reduces BMO latency, while pre-execution moves it off the critical path.

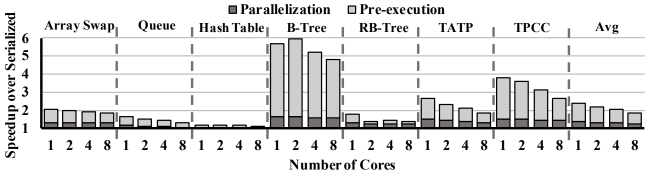


Figure 9: Speed up of Janus over the serialized design with different number of cores.

5.2.2 Comparison with Non-blocking Writeback. In this experiment, we evaluate an ideal case where the writeback requests do not block the execution. Therefore, the BMO latency is *not* on writes' critical path. We want to evaluate how much performance is lost when writes move on the critical path in crash consistent software and how much performance Janus can recover from that. Figure 10 shows the slowdown of the serialized baseline and Janus over the ideal case. We observe that the serialized baseline introduces almost 4.93× slowdown when the BMO latency falls on the critical path. Janus improves the performance by 2.35× by pre-execution and parallelization of the BMOs. However, it still incurs a 2.09× slowdown compared to the ideal scenario. There are two reasons behind the performance gap between Janus and the ideal case. First, not all BMOs can be pre-executed, as sometimes there is not enough gap between the point where the data and address are known and where the actual write happens. Second, not all pre-execution requests can complete before the actual write access arrives. We found that in our experiments, on average, only 45.13% BMOs have been completely pre-executed.

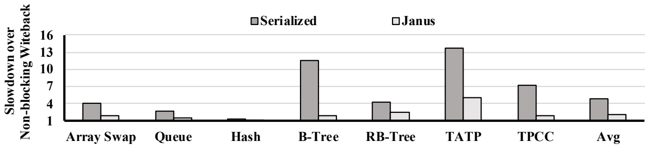


Figure 10: Comparison with the ideal case where BMO latency is not on critical path.

5.2.3 Automated vs. Manual Instrumentation. In this experiment, we evaluate the performance of the automated instrumentation of Janus functions using our compiler pass. Figure 11 shows the speedup of Janus with the *manual* and *automated* instrumentation over the serialized baseline. In most cases, the performance

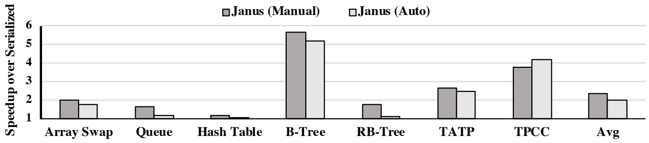


Figure 11: Speed up of Janus over the serialized design with *automated* and *manual* instrumentation.

difference is within 12%. We notice two special cases. (1) The automated solution does not provide a significant performance benefit in RB-Tree and Queue. The static compiler cannot handle loops and pointers (discussed in Section 4.5), which severely affects these two workloads. (2) The automated instrumentation is slightly faster in TPCC. We found that the instrumentation enabled other compiler optimizations on the program, such as hoisting the address generation. On average, the automated solution is only 13.3% slower than our best-effort manual instrumentation. We conclude that our compiler pass effectively finds opportunities for pre-execution and improves performance.

5.2.4 Different Deduplication Ratios and Algorithms. In this experiment, we test three deduplication ratios: 0.25, 0.5 and 0.75, and compare two different hashing algorithms: MD5 and CRC-32. The design using CRC-32 follows the method in [113], which has a lower overhead. Figure 12 shows the speedup of Janus in systems using the MD5 and CRC-32 hashing algorithm. We observe that the speedup of Janus is almost the same under different deduplication ratios with MD5. In contrast, a higher deduplication ratio improves the benefit with the lightweight CRC-32. As MD5 takes around 4× longer than CRC-32, the BMOs dominate the write overhead. Therefore, the performance gain with MD5 is not impacted by the deduplication ratio. Even with CRC-32, the increase in speedup is small because BMOs contribute to most of the overhead, despite the benefit of deduplication.

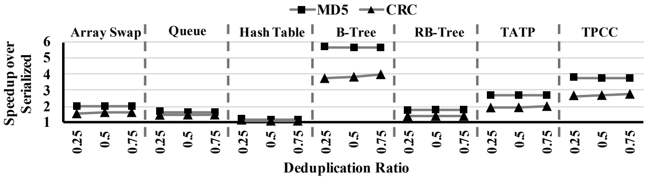


Figure 12: Speedup of Janus over the serialized design with variable deduplication ratios and different algorithms.

5.2.5 Variable Transaction Sizes. In this experiment, we vary the size of the data update in each transaction from 64B to 8KB. As TATP and TPCC are real-world workloads that cannot be easily scaled without changing their semantics, we scale the first five workloads in this experiment. Figure 13 shows the speedup of Janus (parallelized and pre-executed) over the serialized baseline. We observe that the speedup from pre-execution increase with the size of transaction in the beginning, then it starts decreasing at a certain point in all workloads. In comparison to that, the speedup from parallelization keeps increasing but at a slow rate. The reasons are as follows: (1) Pre-execution benefits from a larger transaction size. However, at some point, the units and buffers for BMOs become full. The benefit is maximum at that point and then starts declining after that. (2) On the contrary, the benefit from parallelization is not affected by the BMOs resources. Therefore, the more writes the processor executes, the higher the benefit. We conclude that the speedup from pre-execution can benefit the performance the most when the write intensity is within a certain limit.

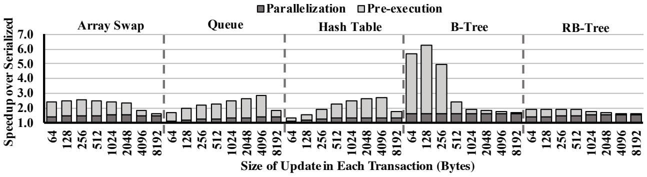


Figure 13: Speedup of Janus over the serialized design with different transaction sizes.

5.2.6 Variable Pre-execution Units and Buffer Size. The previous experiment has shown that the units and buffers for BMOs can become the bottleneck when processing large transactions. Therefore, in this experiment, we scale the number of units and buffers, while the size of transaction is fixed (8KB) for each of the five scalable workloads. We test the speedup of Janus over the serialized baseline with 1×, 2× and 4× of the default number of units and buffers (listed in Table 3). We also include a case with *unlimited* resources. Figure 14 shows that as the BMOs units and buffer size increases, the performance also increases. However, the speedup in most cases saturates when the BMOs units and buffers are no longer the performance bottleneck. B-Tree is an exception. It exhibits a high demand for pre-execution resources and can gain a significant benefit with unlimited resources.

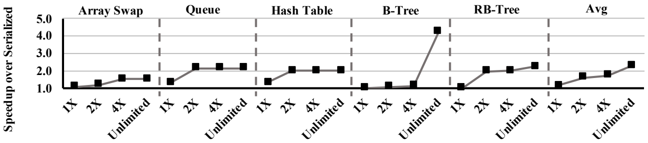


Figure 14: Speedup of Janus over the serialized design with variable number of BMO units and buffer entries.

5.2.7 Overhead Analysis. Table 3 in Section 5.1 lists the size of buffers and queues that support pre-execution. The size of each Pre-execution Request Queue entry and Pre-execution Operation Queue entry is 119 bits and 103 bits, respectively. The size of each IRB entry is 148B. In Janus, we have 16 Pre-execution Request Queue entries, 64 Pre-execution Operation Queue entries, and 64 IRB entries. Therefore, the total storage overhead from queues and buffers is 9.25KB, which is 0.51% of the LLC size. The 4-wide BMOs in our design take 300k gates (according to [78, 79]), which only incurs a 0.065mm^2 die area with 14nm technology.

6 FUTURE WORKS

More precise compiler instrumentation. The limitation of our compiler pass boils down to the unavailable dynamic information during the static compilation time. There are two directions to mitigate this limitation. (1) Improving the dependency analysis on pointers can allow safe but more aggressive pre-execution. Techniques such as SVF [87, 88] can be greatly useful. (2) Utilizing dynamic analysis techniques can provide runtime information and enable more optimization opportunities, such as pre-executing BMOs outside of its function or outside its loop.

Tools for misuse detection. Section 4.4 has described the guidelines on using Janus interface in order to gain the best performance. Future works can provide tools to detect misuse of the interface. There are three typical misuse scenarios: (1) *Modifications on pre-execution object.* Tools can detect whether the pre-executed address and/or data have been invalidated between the pre-execution function and the actual write. Address invalidation can be detected by monitoring memory de-allocation operations and data invalidations can be detected by monitoring assignments to the source of the data. (2) *Useless pre-execution functions*. Pre-execution on objects that do not affect the critical path is unnecessary. Tools can detect whether the pre-execution matches a subsequent blocking writeback. (3) *Insufficient pre-execution window*. The execution of BMOs takes a significant amount of time. The program should leave enough window between the pre-execution function and the actual write in order to maximize the benefit. A static tool can estimate the number of instructions in this window to determine whether the BMO latency can be perfectly overlapped; a dynamic tool can monitor the completion status of pre-execution functions and thereby adjust the instrumentation.

7 RELATED WORKS

Memory and Storage Supports for NVM. Prior works have proposed different optimizations for memory and storage support in NVM systems. Janus efficiently integrates these supports by the parallelization and pre-execution mechanisms, which is orthogonal to these prior works. For exmaple, i-NVMM [13], DEUCE [107] and SecPM [112] design efficient encryption schemes for NVM systems that guarantee the confidentiality of data. Qureshi et al. [70] propose Start-Gap wear-leveling for NVM systems that effectively spreads the writes evenly to the memory cells, improving its lifetime. Error-correcting pointers [80] provide an effective way of remapping worn-out NVM cells to an error-free location. Liu et al. [53] integrate counter-mode encryption into an NVM system and ensures crash consistency by proposing counter atomicity. The authors further reduce the overhead due to counter-atomic writes by selectively applying counter atomicity to the writes that immediately mutates the crash consistency status. Osiris [106] provides confidentiality and integrity guarantees for NVM systems with encryption and integrity verification mechanisms. The authors leverage the ECC bits to detect inconsistency between the encrypted data blocks and their counters. DeWrite [113] integrates both encryption and an efficient deduplication algorithm into an NVM system. By using a low-overhead hashing algorithm and executing encryption in parallel with hashing, this mechanism achieves better performance over the previous schemes.

NVM Crash Consistency. Providing the crash consistency guarantee is another important aspect in NVM systems. Prior works have proposed and implemented a variety of software and hardware solutions to maintain crash consistency. Hardware-based mechanisms include implementations of low-level primitives such as DPO [42] and HOPS [62], and high-level hardware transactions such as Kiln [110], ThyNVM [72], JUSTDO Logging [37] and ATOM [38]. Software-based solutions, such as NV-Heaps [16], Mnemosyne [94], REWIND [10], Intel's PMDK [35], LSNVMM [31], etc., abstract away the low-level crash consistency mechanism and provide a high-level software interface for programmers to manage their persistent data. There are also NVM-optimized file systems, such as Intel's PMFS [24], BPFS [19], NOVA [104], and SCMFS [101]. Janus can be integrated with these crash consistency mechanisms to improve performance. For example, NVM transactions can overlap BMOs latency with other transactional steps using our pre-execution technique.

Compilers and Tools for NVM. There have been works on compiler support and toolchains for NVM systems. Atlas [9], SFR [28] and iDO [52] provide compiler supports that automatically convert the program into failure-atomic regions based on multithreading synchronization primitives, and thereby guarantee crash consistency. The conversion approaches in these works follow the typical NVM transaction programming models. Therefore, it is possible to integrate Janus interface into these compiler techniques. Yat [45], Pmemcheck [34] and PMTest [54], provide tools to detect crash consistency bugs in NVM-based programs. These tools can be extended to detect the misuse of Janus software interface and the mistakes in enforcing metadata atomicity.

Pre-execution in Conventional Processors. There have been pre-execution techniques for conventional processors to reduce LLC misses. The run-ahead execution technique aims to exploit memory-level parallelism in a large instruction window [61]. Speculative precomputation techniques generate helper threads to generate new cache misses [17, 18, 111]. These techniques pre-execute at a coarse granularity of code blocks or instructions. In comparison, Janus exploits pre-execution *within* each write access by pre-executing its BMOs. Also, these prior works require invasive modification to the out-of-order core, while the hardware modifications of Janus are contained within the memory controller.

8 CONCLUSIONS

In this work, we show that backend memory operations (BMOs), e.g., encryption, integrity verification, deduplication etc., necessary for NVM systems can cause a significant performance degradation as they increase the latency of writes that lie on the critical path. To reduce these overheads, we seek to parallelize and pre-execute the BMOs in NVM systems. We propose Janus, a general and extensible software-hardware approach to mitigate the overhead of BMOs. Over a suite of crash-consistent NVM applications, we observe that Janus achieves 2.35× and 2.00× speedup over a serialized baseline NVM system by manual and automated instrumentation of Janus primitives. We hope that Janus will open up new research opportunities in optimizing backend memory operations and will be integrated into real NVM systems.

ACKNOWLEDGMENT

We thank the anonymous reviewers, Yizhou Wei and Vinson Young for their valuable feedback. This work is supported by NFS grants award 1822965 and 1566483, and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

REFERENCES

[1] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory expansion technology (MXT): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001.

[2] Alaa Alameldeen and David Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical Report 1500, Computer Sciences Dept., UW-Madison, 2004.

[3] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *ISCA*, 2004.

[4] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, 2017.

[5] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] David Brash. Armv8-A architecture evolution. https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution, 2016.

[8] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck. Codes for asymmetric limited-magnitude errors with application to multilevel flash memories. *IEEE Transactions on Information Theory*, 56(4), 2010.

[9] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[10] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[11] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, Aug 2010.

[12] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. HICAMP: Architectural support for efficient concurrency-safe shared structured data access. In *ASPLOS*, 2012.

[13] Siddhartha Chhabra and Yan Solihin. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *ISCA*, 2011.

[14] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. Compresso: Pragmatic main memory compression. In *MICRO*, 2018.

[15] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *SOSP*, 2013.

[16] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[17] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *MICRO*, 2001.

[18] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*, 2001.

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[20] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. NVM compression—hybrid flash-aware application level compression. In *INFLOW*, 2014.

[21] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *ATC*, 2010.

[22] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stanley B. Zdonik, and Subramanya Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In *VLDB*, 2014.

[23] Whitfield Diffie and Martin Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979.

[24] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, 2014.

[25] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. Phoenix: Memory speed HPC I/O with NVM. In *HiPC*, 2016.

[26] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [Nearly] free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, 2015.

[27] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *HPCA*, 2003.

[28] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *PLDI*, 2018.

[29] Seokin Hong, Prashant Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu Hyoun Kim, and Michael Healy. Attache: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *MICRO*, 2018.

[30] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *EuroSys*, 2017.

[31] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *ATC*, 2017.

[32] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[33] Intel Corporation. Intel architecture instruction set extensions programming reference (319433-034 may 2018). https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[34] Intel Corporation. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html.

[35] Intel Corporation. Persistent memory programming. https://pmem.io/.

[36] Intel Corporation. Revolutionary memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[37] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, 2016.

[38] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *HPCA*, 2017.

[39] Sudarsun Kannan, Gavrilovska, Karsten Schwan, and Dejan Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IPDPS*, 2013.

[40] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *ISCA*, 2017.

[41] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.

[42] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *MICRO*, 2016.

[43] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[44] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[45] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *ATC*, 2014.

[46] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[47] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[48] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[49] Qingan Li, Yanxiang He, Yong Chen, Chun Jason Xue, Nan Jiang, and Chao Xu. A wear-leveling-aware dynamic stack for PCM memory in embedded systems. In *DATE*, 2014.

[50] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *FAST*, 2016.

[51] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.

[52] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *MICRO*, 2018.

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

[54] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.

[55] Huizhang Luo, Qingfeng Zhuge, Liang Shi, Jian Li, and Edwin H.-M. Sha. Accurate age counter for wear leveling on non-volatile based main memory. *Design Automation for Embedded Systems*, 2013.

[56] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using transparent compression to improve ssd-based i/o caches. In *EuroSys*, 2010.

[57] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *FAST*, 2016.

[58] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *HotStorage*, 2017.

[59] Leonardo Marmol, Jorge Guerra, and Marcos K. Aguilera. Non-volatile memory through customized key-value stores. In *HotStorage*, 2016.

[60] David Mulnix. Intel Xeon Processor D product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview.

[61] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.

[62] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, 2017.

[63] Riaz Naseer and Jeff Draper. Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs. In *ESSCIRC*, 2008.

[64] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net/, 2011.

[65] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *HotStorage*, 2018.

[66] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *MICRO*, 2013.

[67] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*, 2012.

[68] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014.

[69] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA*, 2010.

[70] Moinuddin K. Qureshi, Michele Franchescini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO*, 2009.

[71] Joydeep Rakshit and Kartik Mohanram. ASSURE: Authentication scheme for secure energy efficient non-volatile memories. In *DAC*, 2017.

[72] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *MICRO*, 2015.

[73] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[74] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-RAM. In *HPEC*, 2013.

[75] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, 2013.

[76] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly. In *MICRO*, 2007.

[77] Brain Rogers, Yan Solihin, and Milos Prvulovic. Memory predecryption: Hiding the latency overhead of memory encryption. In *Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[78] Akashi Satoh and Tadanobu Inoue. ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS. *INTEGRATION, the VLSI journal*, 40(1):3–10, 2007.

[79] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-box optimization. In *Asiacrypt*, pages 239–254. Springer, 2001.

[80] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. Use ECP, not ECC, for hard failures in resistive memories. In *ISCA*, 2010.

[81] Ali Shafiee, Rajeev Balasubramonian, Mohit Tiwari, and Feifei Li. Secure DIMM: Moving ORAM primitives closer to memory. In *HPCA*, 2018.

[82] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. MemZip: Exploring unconventional benefits from memory compression. In *HPCA*, 2014.

[83] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *CCS*, 2013.

[84] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *ICS*, 2003.

[85] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO*, 2003.

[86] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *ISCA*, 2005.

[87] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC*, 2016.

[88] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, Feb 2014.

[89] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[90] Shivam Swami and Kartik Mohanram. Arsenal: Architecture for secure non-volatile memories. *IEEE Computer Architecture Letters*, 17(2):192–196, July 2018.

[91] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, 2000.

[92] Transaction Processing Performance Council (TPC)). TPC-C. http://www.tpc.org/tpcc/default.asp.

[93] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[94] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memeory. In *ASPLOS*, 2011.

[95] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue. NV-Dedup: High-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, 67(5):658–671, 2018.

[96] Rujia Wang, Youtao Zhang, and Jun Yang. Cooperative path-ORAM for effective memory bandwidth sharing in server settings. In *HPCA*, 2017.

[97] Rujia Wang, Youtao Zhang, and Jun Yang. D-ORAM: Path-ORAM delegation for low execution interference on cloud servers with untrusted memory. In *HPCA*, 2018.

[98] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. In *VLDB*, 2014.

[99] Wujie Wen, Yaojun Zhang, Mengjie Mao, and Yiran Chen. State-restrict MLC STT-RAM designs for high-reliable high-performance memory system. In *DAC*, 2014.

[100] D. Williams and Emin Gun Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *NCA*, 2004.

[101] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *SC*, 2011.

[102] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *ATC*, 2017.

[103] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, 2015.

[104] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

[105] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO*, 2003.

[106] Mao Ye, Clayton Hughes, and Amro Awad. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. In *MICRO*, 2018.

[107] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. DEUCE: Write-efficient encryption for non-volatile memories. In *ASPLOS*, 2015.

[108] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting pcm asymmetries. In *HPCA*, 2013.

[109] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.

[110] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.

[111] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *ISCA*, 2001.

[112] Pengfei Zuo and Yu Hua. SecPM: A secure and persistent memory system for non-volatile memory. In *HotStorage*, 2018.

[113] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *MICRO*, 2018.