Empirical Analysis of Architectural Primitives for NVRAM Consistency

Arun KP, Debadatta Mishra
Indian Institute of Technology Kanpur, India
{kparun,deba}@cse.iitk.ac.in

Biswabandan Panda
Indian Institute of Technology Bombay, India
biswa@cse.iitb.ac.in

***Abstract*—Non-volatile memory (NVM) provides persistent memory semantics with access latencies comparable to volatile DRAM. The persistent nature of NVM requires the application developers to design data consistency mechanisms for failure recovery, without which application may end up with inconsistent memory state after a power failure or a system crash. Most commonly employed methods use architectural support for cache line flushing and memory fencing to enforce ordering of writes to NVM.**

In this paper, we study the performance overhead of different hardware primitives used to achieve NVM consistency on Intel x86-64 and Arm64 systems using micro-benchmarks. Further, we also empirically analyze the impact of working set size and memory access characteristics (read-to-write ratio) of applications on different data consistency techniques. Logging based mechanisms (e.g., redo and undo logging), commonly used for NVM consistency, also use underlying architectural primitives like cache flushing. We comparatively study the overheads of redo and undo logging with different architectural primitives. The analysis presented in this paper can be useful to improve the software/hardware architecture, develop efficient applications and perform better capacity planning in NVM systems.

***Index Terms*—Non-volatile memory, persistent barriers, clflush, clwb, redo-undo logging**

I. Introduction

Byte addressable Non-Volatile Memory (NVM[1]) not only provides persistent memory semantics but also enables large memory capacity with access latencies comparable to volatile memory (DRAM). The persistent semantics offers an attractive alternative to meet data persistence requirements of applications' without relying heavily on off-the-chip persistent storage devices (e.g., HDDs and SSDs) and complex storage management middleware such as file systems [1]. The traditional application programmer interfaces are already going through an overhaul to leverage the benefits of NVM by storing the data directly in NVM, avoiding any serialization requirements [2]. However, the transition from volatile memory to non-volatile memory presents some unique challenges. One of the major research challenges is to design specialized techniques to provide *consistent* memory state for applications using NVM across system restarts. Considering the intricacies of processor-memory data path organization, traditional file system or database techniques (e.g., journaling and transactions) may require non-trivial adaptation. To provide *consistency* and *atomicity* guarantees for the applications using NVM, several techniques are proposed [3]–[6].

To leverage the full benefits of the NVM systems, applications can store and access data in the form of in-memory data structures residing in the NVM. The root cause of inconsistency arises in the event of a system crash where hardware caches and other volatile micro-architectural components lose in-flight memory modifications. There can be two major repercussions in the above scenario—*(i)* the system is left in a corrupt memory state, and *(ii)* the expected program execution state is different from the non-volatile memory state. One of the approaches used to address this inconsistency is to allow the application developer achieve a guaranteed temporal order for persisting the data into the persistent domain (i.e., NVM) [7]. With this approach, the application developer is guaranteed to see the updated memory content even after an abrupt system reboot caused due to software/hardware failure. *Persistent barrier* is one of the techniques to achieve NVM consistency in an efficient manner. The application programmer (or the NVM support library) can place persistent barriers at appropriate places to ensure data propagation to the NVM and achieve crash consistency. In this paper, *we analyze the performance implications of different architectural primitives* for NVM consistency. Other techniques like controlling the data cache behavior using page table attributes (e.g., PAT in x86) results in significant performance overheads (as we show in Section III) and are not considered in this paper.

The primitive architectural artifacts such as cache flush and memory fences are used to realize persistent barriers in NVM systems [7]. Logging based techniques (similar to file system journaling) such as redo or undo logging [5], [8]–[11] also depend heavily on the architectural implementation of persistent barriers. For example, the log write operations used for redo and undo logging techniques depend on the persistent barriers to achieve crash consistency. Persistent barriers also play a key role in the data structures, memory allocators and memory management schemes proposed specifically for NVM [12]–[19]. Most instruction set architectures (ISAs) provide a set of instructions (e.g., `clflush` and `mfence` in x86 ISA) along with required extension of the persistent domain (e.g., battery powered memory controller a.k.a. Intel ADR [20]) to propagate changes to NVM in a consistent manner. We empirically analyze the performance overhead of different data consistency methods provided by Intel x86-64 (`clflush`,

[1]NVM and NVRAM are used interchangeably throughout the paper

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00031

clflushopt, clwb) and Arm64 (civac, cvac) ISAs [21] [22]. Through this empirical study, we try to answer following questions,

1) What is the performance overhead of different consistency primitives for different ISAs (x86-64 and Arm64) with single and multi-threaded applications?

2) How does the memory footprint and access characteristics (read-to-write ratio) of different applications influence the performance overhead of different data consistency methods?

3) Transaction-based consistency mechanisms like redo and undo logging use the underlying architectural persistence barriers to achieve atomic update semantics. In this context, an interesting question is: What is the comparative performance overheads for redo and undo logging schemes with different data consistency methods?

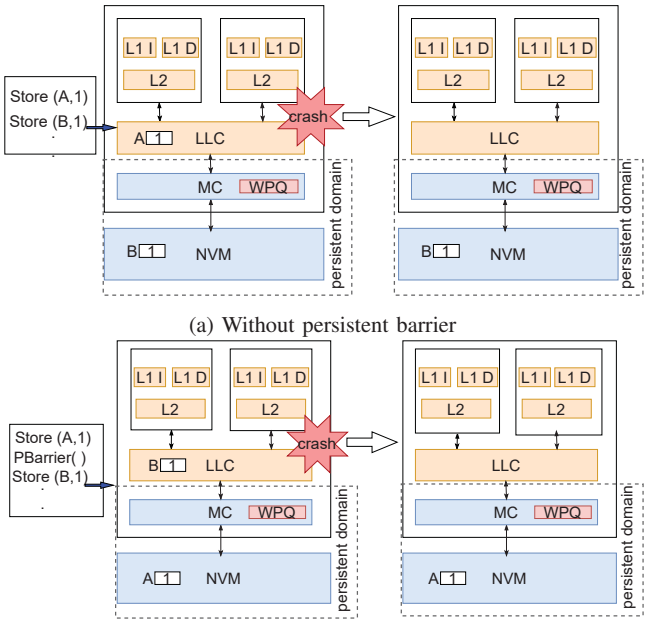
Answering the above questions can provide guidelines for application/middleware developers to choose the right technique and account for the resource overheads during capacity planning. Moreover, the analysis presented in this paper can provide directions to improve the efficiency of architectural primitives for NVM consistency. In this paper, we have used a set of workloads designed to operate on well known data structures and executed them on Gem5 simulation platform [23] to analyze the performance implication and provide justifications through different architecture layer metrics.

Some important observations from this empirical study are,

1) There is no *one-size-fits-all* approach to choosing consistency primitives, as the performance overhead of consistency primitives depend upon the nature of workload. For example, we observed that linear queue incurred the highest and cuckoo hashing the lowest performance overhead across different consistency primitives on x86-64 and Arm64 NVM systems. While advanced primitives like clwb (in x86-64) and cvac (in ARM) performed better than other techniques in most scenarios, there were workloads and setups for which the primitives did not offer any benefits, sometimes caused marginal performance degradation compared to other techniques like clflush and civac. Moreover, there are many instances where the fence operation becomes a significant bottleneck independent of the cache flush method.

2) The influence of memory footprint and memory access pattern on performance overhead of data consistency primitives are found to be mostly depended on the nature of workload. The performance overheads of cuckoo hashing (Table I) remained the lowest across all working set sizes while the maximum influence of memory access characteristics (read-to-write ratio) was observed for queue data structure. The results also show a decrease in relative performance overheads when the application working set does not fit in the cache hierarchy.

3) The choice of underlying architectural persistence barrier plays a key role in the performance of logging based NVM consistency (redo and undo). Redo logging with clwb performed better than clflush and clflushopt whereas clflush or clflushopt performs better than clwb for some cases with undo logging which suggests that clwb is not always the best while using undo logging. The performance of redo logging varied from x86-64 to Arm64 where redo showed performance overhead for most of the cases on Arm64 but performed better on x86-64.



(a) Without persistent barrier

(b) With persistent barrier

Fig. 1: Consistency issue with non-volatile memory and solution using persistent barrier (shown as PBarrier) techniques. [A, B - NVM Addresses, Store(A, 1) - writes 1 to NVM address A]

The contributions of this paper are as follows,

- We perform detailed experimental analysis of performance overheads with different NVM consistency mechanisms available in x86-64 and ARM64 architectures.
- We study the performance implications with different parameters like workload type, memory footprint and memory access characteristics along with causal insights.
- We show comparative analysis of redo and undo logging mechanisms on x86-64 and ARM64 systems with different logging requirements.

II. Background

In this section, we discuss the importance of data consistency in NVM and different architectural primitives based on cache flush instructions and memory barriers available on x86-64 and Arm64 ISAs. We also explain the working of undo and redo logging techniques to achieve failure atomicity in NVM systems.

A. Importance of Data Consistency in NVM

Modern systems have multiple levels of caches with a last-level cache (LLC) shared across cores (Figure 1). CPU caches offer significant performance improvements by leveraging both temporal and spatial locality of memory accesses and help to

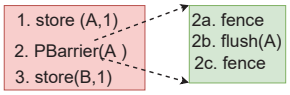


Fig. 2: Flush + Fence persistent barrier primitive

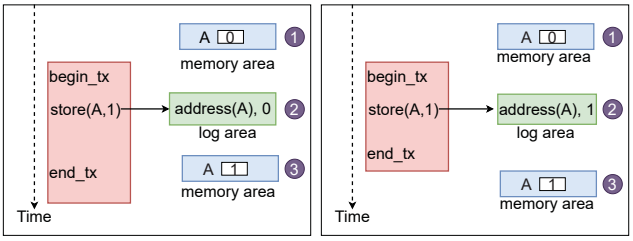
bridge the memory wall problem to a large extent. However, the volatile nature of caches pose new challenges when NVM is accessed using LOAD and STORE instructions as caches lose data in the event of a power failure or a crash. At the core, the challenge is the possible inconsistency between the execution state (program counter, register values etc.) and the memory state in the event of a system crash.

In Figure 1a, we show an example of inconsistency while using two store instructions to NVM addresses `A` and `B` without any persistent barrier. There can be two consistency issues if the system crashes before the cache line containing the updated value of `A` is written back to the non-volatile memory and the program counter (PC) is pointed to the address of the instruction following `Store B`. First, at the time of application restart, assuming the PC is restarted to an instruction after the store instructions, the application *will not* see the updated value of `A`. Second, if the cache line corresponding to `B` is written back before `A`, at the time of restart, the application will find that `B` is updated while `A` is not. This is a more serious issue as it makes any smart recovery process non-deterministic; the recovery process can not determine where to resume analyzing the memory content at the time of application restart. With a persistent barrier (Figure 1b), the cache line corresponding to store address (`A` in the example) is written back till the persistent domain such that the data is guaranteed to be in NVM in the event of a system crash. The persistent domain includes a specialized memory controller with persistent write queues (Write Pending Queue (WPQ) in Figure 1) to ensure data persistence once the write-back request is queued. Implementation of persistent barrier depends on the ISA as we discuss in the next subsection. Note that, one of the solutions to the consistency issues can be persistent caches [20] which is not in the scope of this paper.

B. Data Consistency Methods using Flush and Fence

Persistent barrier implemented by combining `flush` and `fence` instructions is shown in Figure 2. In both Intel x86-64 and Arm64, the persistent barrier is realized through different combination of instructions as we discuss in this subsection.

1) Intel x86-64: Intel x86-64 provides three cache flush instructions—`clflush`, `clflushopt` and `clwb`, with subtle differences in their implementation. `clflush` invalidates the cache line containing the linear address from all the levels of cache hierarchy and performs write back of the dirty cache lines to the memory, if required. `clflushopt` is an optimized variant of `clflush`, as it allows concurrency while flushing multiple cache lines [20]. On the other hand, `clwb` writes back the modified cache line and retains the cache line in a clean state. As a result, `clwb` is beneficial for the cases where future accesses to the data are expected. `clflush`, `clflushopt` and `clwb` are ordered by store-fence (`mfence` and `sfence`) instructions. The persistent barrier in Figure 2 can be optimized in x86-64 systems by removing `fence` at line 2a because of the following reasons. `clflush` instruction orders with respect to writes and other `clflush` instructions. Similarly, the `clflushopt` and `clwb` instructions order with respect to older writes to the cache-line being invalidated [21].



(a) Undo logging (b) Redo logging

Fig. 3: Working example of undo and redo logging approaches for NVM consistency. (1) shows initial value of A, (2) shows logged information [Address of A, value], and (3) shows modified value of A.

2) Arm64: Arm Cortex-A provides mechanisms to *invalidate* or *clean* a cache line. `civac` and `cvac` are two such operations provided by Arm64 to perform *invalidate* or *clean* on data caches. The `civac` performs both *clean* and *invalidate* of a virtual address to the point of coherency and `cvac` performs *clean* of a virtual address to point of coherency; more information about Point of Coherency (PoC) is given below.

Invalidation of a cache line clears the valid bit; *cleaning* writes the content of cache line to next level of cache or to main memory if the line is marked as dirty. *Cleaning* clears the dirty bit and makes the content of a cache line consistent with the next level of cache or memory. The *invalidate* or *clean* operation takes either the virtual address, or set and way of the cache-line. In case of virtual address, Arm64 allows invalidation and clean operations at two points—*(i)* Point of Coherency (PoC) and *(ii)* Point of Unification (PoU). PoC is the point at which all observers see the same copy of a memory location and PoU for a core is the point at which all operations of the core see the same copy of a memory location. Both `civac` and `cvac` operates at PoC [22].

For the implementation of persistent barrier (Figure 2) on Arm64, we used data memory barrier to the inner shareable domain (`DSB ISH`) as fence [22] along with `civac` and `cvac` operations.

C. NVM Consistency using Logging

Logging based NVM consistency techniques provide transaction semantics with atomicity guarantees [7]. Figure 3 shows the working of a failure-atomic update transaction (marked between `begin_tx` and `end_tx`) using log based mechanisms. The undo log mechanism stores the old value (in a log area in the NVM) prior to the modification and discards the log if operations in atomic block complete successfully (Figure 3a).

TABLE I: Micro-benchmarks used in the experiments

Benchmarks	Description
BST	Binary Search Tree
BST_P	Parallel Binary Search Tree
CUH	Cuckoo Hashing Table [26]
QUE	Linear Queue
RBT	Red-black Tree [27]

TABLE II: Working set sizes used for experiments

Type	Size Description	Inserts	Deletes	Searches*
Tiny	0.90 x L1-D Size	460	400	600
Small	0.90 x L2 Size	7370	4000	6000
Medium	0.90 x LLC Size	29490	4000	6000
Large	4 x LLC Size	131070	4000	6000

*Queue does not support search operations

A recovery mechanism uses the value in the undo log to revert back changes in case of a failure. Read operations inside an atomic block accesses the values directly from the corresponding memory addresses as the memory contains the updated values. In the example shown in Figure 3a, `store(A,1)` creates an undo log entry containing the address and the old value of `A`. Application is allowed to modify the value of `A` inside the atomic block after persisting the undo log entry (using a persistent barrier) as shown in Figure 3a.

With redo log technique, the updated values are stored in the log during the transaction and the application read requests to modified variables are served from the log area (Figure 3b). The log entries are written to the memory locations during the transaction commit which is known as synchronous redo logging. The application can also apply log entries to memory locations in an asynchronous manner after committing the atomic block. The recovery mechanism re-performs the operations in the redo log entries during recovery (after a failure). In the redo logging example shown in Figure 3b, `store(A,1)` creates a redo log entry containing the address and the updated value of `A`. Application updates the value of `A` during commit operation after persisting the redo log as shown in Figure 3b.

Transaction commit is faster with undo logging compared to redo logging since the data structure is modified in place for undo logging whereas redo logging requires applying the changes during commit. Failed transaction recovery is faster for redo since the data structure is not modified until transaction commits whereas undo requires applying the log entries during recovery. There is also a difference in the number of persistent barriers required for undo and redo logging techniques—undo requires a fence for each log write while redo requires only one fence for all log entries during the commit [24]. Undo logging performs better for workloads with more reads and it is more sensitive to read-write ratio as observed by Hu et. al [25]. The redo logging incurs additional overhead of redirecting reads to the log area for getting updated value.

III. Setup and Methodology

A. Benchmarks and Parameters

We have used micro-benchmarks using different well known data structures (Table I) to study the performance overhead of different data consistency methods.

BST benchmark performs insert, delete and search operations on a binary search tree. One of the data consistency methods (passed as a parameter to the benchmark) is used after each insert and delete operation to push changes into the persistence domain. BST_P is a parallel implementation of binary search tree using `POSIX` threads to parallelize search and update operations where a read-write lock is used for synchronization across the three operations.

CUH benchmark uses cuckoo hashing [26], a dictionary data structure with constant worst case lookup time. Cuckoo hashing uses two hash tables, with two hash functions `hash1` and `hash2`, respectively. Every key `x` is stored in the cell `hash1(x)` of the 1st table or the cell `hash2(x)` of the 2nd table. During insertion of any key `x`, if the cell `hash1(x)` is free, then key is inserted there. Otherwise, the previous occupant of the cell `hash1(x)` becomes "nestless" after `x` is inserted in that position. The nestless key is inserted into the 2nd table by following the same procedure until a free slot or "MaxLoop" count is reached. Reaching "MaxLoop" count results in resizing of the hash table. Note that, an insertion may cause multiple keys to become "nestless". We use different data consistency methods every time any cell in either of the the hash tables is updated.

QUE benchmark is based on linear queue with a head and a tail pointer providing enqueue and dequeue operations. Enqueue operation adds a new element at the rear end and updates the tail pointer. The enqueue operation ensures that both tail pointer and newly added item reach persistence domain by using one of the data consistency methods. Dequeue operation removes an item from the front and updates the head pointer while ensuring that the head pointer updation reach the persistent domain.

RBT benchmark performs different operations on a height balanced binary search tree (red-black tree) with *red-black properties*—(i) each node is either red or black, (ii) both children of a red node are black, (iii) path from a node to descendant leaves have same number of black nodes, and, (iv) root and leave nodes are black. An insert or delete operation may violate the red-black property, thus requiring change of color for multiple nodes or rotation operations to restore the red-black properties. RBT uses one of the data consistency methods when the RB tree is updated to ensure data persistence.

To study the performance implications with different cache working set size, we have used four variants— *tiny*, *small*, *medium* and *large* (Table II), as parameters for the experiments. Table II shows the number of insert, delete and search operations performed under each working set size variant; insert and delete operations maintain the working set size (mentioned under the size description) of the micro-benchmarks. For example, in case of *tiny* working set, size of micro-benchmarks always fits into L1 data cache (L1-D) while performing insert and delete operations.

TABLE III: Gem5 configuration (used for Section IV)

CPU	Out-of-order CPU
L1-D/I	32 KiB/core (8 way)
L2	512KiB/core (16 way)
L3	2 MiB/core shared (16 way)
MSHRs	16, 32, 32/core at L1-D, L2, L3
Cache data access latency	2, 9, 15 cycles at L1-D, L2, L3
Cache line size	64 B in L1, L2, L3
Replacement policy	LRU
L1 prefetcher	StridePrefetcher with degree=4
Memory controller	Nvmain* [28]
Memory	PCM with configuration based on [29]
Memory capacity	10 GB (20GB for section IV-B1)

*Gem5 NVM Interface [30] used for results in section IV-B1

TABLE IV: System Parameters (used for Section III-C)

CPU	Xeon(R) 3.20GHz
L1-D/I	32KB (8 way)
L2	1MB (16 way)
L3	8MB (11 way)
OS	Ubuntu 18.04.3 LTS
Kernel	4.19.13

B. Redo and Undo Logging

As discussed in Section II-C, undo logging techniques record old values and discard them on success whereas redo logging techniques record new values and applies them on success. To study the influence of different data consistency primitives with redo and undo logging schemes, we used a micro-benchmark that maintains the LRU order of data items in presence of different access patterns. The LRU micro-benchmark consists of a linked list of items where the recently accessed item is maintained at the head of the list. The implementation consists of a hash-table to speed up the access where an entry in the hash-table corresponds to an item in linked list. The hash-table entry indexes into the linked list and the item moves to the head position after the access. To ensures failure atomicity, undo or redo logging is implemented as a linked list of entries with each entry consisting of a <address, value> pair, where the $value$ is a pointer to a memory address of configurable size to support different data items. In our redo implementation, read access is implemented by looking up the item in the LRU data structure. Our study differs from Wan et al. [25] as we have compared the performance with different persistent barrier primitives while Wan et al. [25] used redo and undo logging with Intel PMDK framework.

We have configured Gem5 [23] with the configuration parameters shown in Table III for the experimental evaluation.

C. Why do we focus on flush based data consistency methods?

Even though cache line flushing is the most commonly used approach to ensure data consistency in NVM, other alternative mechanisms can be designed using cache bypassing, write-through caches or non-temporal stores [31]. To understand the behavior of these alternatives, we compared the performance overhead with uncacheable (UC) and write-through (WT) [21] memory against different cache line flush based data consistency methods. We used a set of micro-benchmarks (Table I), with LLC thrashing working set size, on an Intel x86-64 system with configuration mentioned in Table IV.

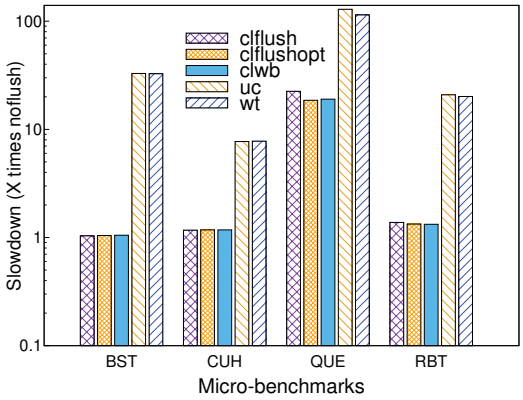


Fig. 4: Performance overhead of data consistency methods. Y-axis values are slowdown with respect to not using any data consistency method (lower the better).

We have used Page Attribute Table (PAT) of x86-64 systems to set the memory type as UC or WT. We augmented the `mmap` system call in the Linux kernel (v4.19.13) to introduce a special flag (`MAP_SENSITIVE`) to allow the user space to control the caching behavior. Depending on the value of the flag passed from the user space, any given virtual address range is mapped as UC or WT by configuring the page table entry with appropriate PAT value [21]. Figure 4 shows the performance overhead of different data consistency methods normalized to `noflush`. The performance overheads with UC and WT is significantly higher (between 5X - 31X) compared to the cache line flush based data consistency methods. The results clearly demonstrates the benefits of cache flushing based techniques and therefore, we focus on studying different cache line flushing based data consistency methods in this paper.

IV. Experimental Evaluation

A. Performance overhead of data consistency methods

In this subsection, we study the performance overhead of different data consistency methods with x86-64 and Arm64 systems (simulated using Gem5) using the micro-benchmarks mentioned in Table I. To analyze the performance implication of working set size with different levels of cache occupancy, we varied the working set size by configuring the benchmark parameters as mentioned in Table II.

1) Performance with x86-64: Figure 5 shows the performance slowdown of different micro-benchmarks with different data consistency methods. The slowdown is the ratio of completion time with different data consistency methods and `noflush` (i.e., not using any data consistency method). We can observe that `clflush` and `clflushopt` resulted in similar slowdown across all working set sizes with all workloads. This is primarily due to the inevitable requirement of ordering the flushes to ensure NVM consistency which negates

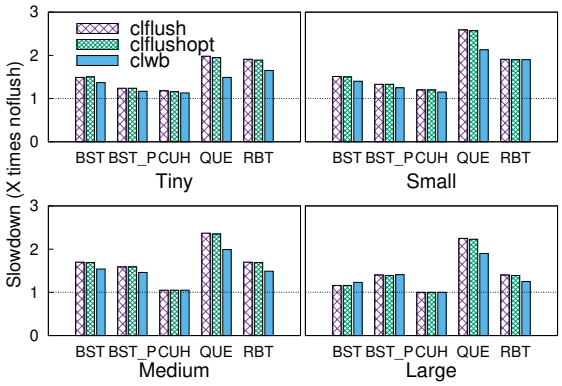


Fig. 5: Performance slowdown with different data consistency methods in x86-64 systems. Y-axis values are slowdown normalized to `noflush` (lower the better)

the optimizations (i.e., concurrency of flush operations) provided by `clflushopt` (Refer Section II for details). With `clwb`, the performance overheads are lower (by 1X-1.3X) compared to both `clflush` and `clflushopt` depending on the benchmark and working set size. The performance overheads of different flush methods vary between (1X-2.5X) compared to `noflush`, where QUE benchmark resulted in the highest performance overhead for all working set sizes and CUH has the lowest for all working set sizes.

For QUE micro-benchmark, with `clflush` and `clflushopt`, each insert results in two additional cache misses—(i) cache miss while updating the next pointer of the previously inserted element and (ii) cache miss while updating tail pointer. The delete operation also results in additional cache misses while updating the head pointer. By taking large working set size as an example, we can see the impact on the cache miss at L1-D for QUE with `clflush` and `clflushopt` compared to other benchmarks (Table V). QUE also resulted in highest MPKI[2] (based on demand misses for CPU data) at LLC for `clflush` and `clflushopt` as shown in Table V. It is interesting to note that, QUE with `clwb` resulted in significant slowdown (Figure 5) even though the MPKI at LLC was lower compared to other techniques (Table V). This can be explained by analyzing the delay in committing the reorder buffer (ROB) head. The total number of times ROB commit has to stall due to non-speculative instruction reaching head of the ROB is high for `clwb` and all other data consistency methods in comparison with `noflush` as shown in Table V. Therefore, the presence of *memory fences to order flushes* contributes significantly towards the overall performance overhead. The highest number of commit stalls at ROB head is for `clflush` and decreases by 33% to 50% with `clflushopt` or `clwb` for all micro-benchmarks except for BST_P. BST_P experiences high number of commit stalls at the ROB head even with `noflush` due to usage of locks.

[2]misses per kilo instruction

TABLE V: Cache miss and ROB stall behavior with large working set

Methods	BST	CUH	QUE
LLC MPKI			
noflush	1.74	1.68	5.23
clflush	1.79	1.68	20.42
clflushopt	1.79	1.67	19.27
clwb	1.69	1.66	0.69
L1-D miss rate			
noflush	6.23	0.65	2.63
clflush	5.97	0.63	7.89
clflushopt	5.96	0.63	7.55
clwb	6.03	0.63	3.11
#commit stalls at ROB			
noflush	6	34	6
clflush	798,423	3,348,775	536,286
clflushopt	532,284	2,232,528	270,146
clwb	532,284	2,232,528	270,146

The minimum performance overhead of CUH with different data consistency methods can be correlated with its similar miss rates at L1-D and similar MPKI values at LLC (Table V). The similar cache miss behavior can be attributed to the number of keys becoming "nestless" during inserts, which remains same across all techniques. The L1-D miss rate and LLC MPKI indicates that, CUH benchmark do not exhibit high temporal locality, and therefore, the performance impacts due to cache line flushing is negligible.

We observed a decrease in slowdown (with all flush methods) between medium and large working set sizes as shown in Figure 5. Increase in time taken for `noflush` can result in comparatively lower slowdown because we calculate the slowdown in a relative manner (w.r.t. the `noflush` performance). As the medium working set benchmark fits into the LLC while large working set does not, `noflush` resulted in better performance with medium working set by maximizing the benefits of temporal locality of memory accesses. Therefore, there is a decrease in the relative slowdown with large working set size compared to that of medium working set size. We can confirm the change in `noflush` performance under medium and large working set sizes by comparing the MPKI at LLC with `noflush` and `clflush` methods by taking BST benchmark as an example. The MPKI values at LLC for medium working set size were—0.22 for `noflush`, 1.04 for `clflush`, 1.04 for `clflushopt` and 0.22 for `clwb` (not shown in Table). The MPKI at LLC for `clflush` compared to `noflush` is 3.7X and 0.02X for medium and large working sets, respectively, resulting in higher relative performance overhead for medium working set.

2) Performance with Arm64: We characterized the performance of different benchmarks (Table I) with `civac` and `cvac` consistency primitives for ARM (Refer Section II for details). We observed that `cvac` performed better and resulted in lower slowdown compared to `civac` across all micro-benchmarks and working set sizes. We can functionally equate `cvac` with `clwb` as both of these mechanisms retain cache line in a clean state and shows same performance trend across different working set sizes. Like in the case of x86-64, QUE benchmark resulted in the highest performance overhead for

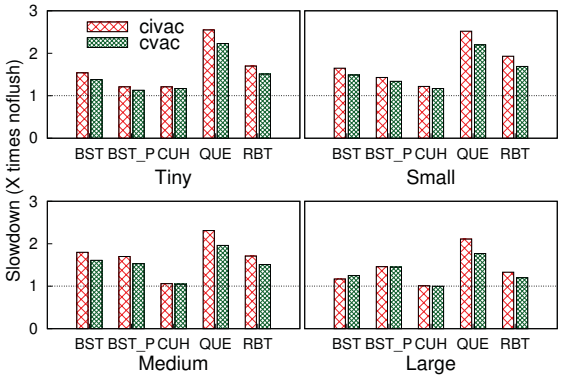


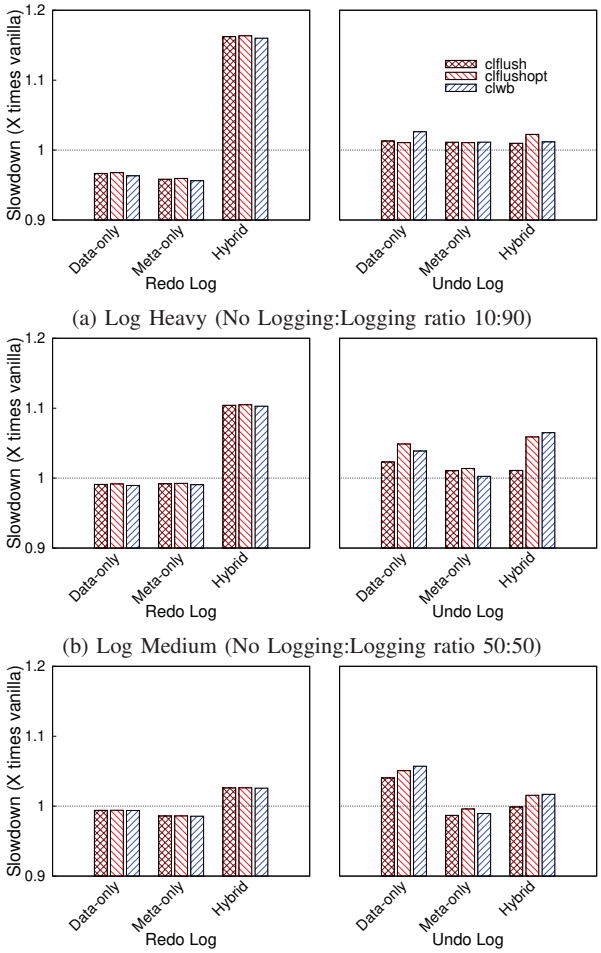
Fig. 6: Performance slowdown with different data consistency methods in Arm64 systems. Y-axis values are slowdown normalized to noflush (lower the better)

TABLE VI: Influence of read-to-write ratio on performance slowdown

Benchmark	read-light	read-balanced	read-heavy
civac			
BST	1.67	1.78	1.89
BST_P	1.54	1.64	1.97
QUE	2.47	2.34	2.22
RBT	1.66	1.71	1.82
cvac			
BST	1.53	1.63	1.83
BST_P	1.45	1.48	1.88
QUE	2.08	1.98	1.88
RBT	1.48	1.51	1.61

all working set sizes and CUH has the lowest for all working set sizes. We observed that the performance overhead of QUE is associated with the cost of ordering *clean* operations because MPKI at LLC for QUE is lower for `cvac` than for `noflush` and total number of times commit has to stall is higher for `cvac` than `noflush` due to usage of memory fence to order writes to NVM.

To study the impact of application memory access behavior, we performed an experiment where the ratio between read and write operations is used as a parameter. For this experiment, we used three variants for each benchmark—(i) *read-light* with read:write ratio as 10:90 (ii) *read-balanced* with read:write ratio as 50:50 and (iii) *read-heavy* with read:write ratio as 90:10. Note that, the write operations comprise of both insert and delete operations. All benchmarks with different working set sizes resulted in similar slowdown across the three variants (i.e., read-heavy, read-balance and read-light) except for the medium working set (shown in Table VI). With increase in percentage of read operations, the performance overhead increased for all benchmarks except for QUE. With *read-heavy*, BST_P resulted in 1.27X performance degradation compared to *read-light*. Interestingly, QUE resulted in comparatively less overheads with increasing number of read operations—with *read-heavy*, QUE resulted in ∽10% less overhead compared to *read-light*; the L1-D miss rate is reduced by ∽5% with *read-heavy* as compared with *read-light* for QUE.



(a) Log Heavy (No Logging:Logging ratio 10:90)

(b) Log Medium (No Logging:Logging ratio 50:50)

(c) Log Light (No Logging:Logging ratio 90:10)

Fig. 7: Influence of data consistency methods based on nature of update operations for different logging requirements in x86-64. Y-axis values are speedup w.r.t. vanilla case of no logging (lower the better).

B. Influence on Redo and Undo Logging

1) Performance with x86-64: We studied the performance overhead of redo and undo logging with different data consistency methods used to ensure consistency of the LRU benchmark (Section III-B). The characterization uses nature of modifications (Data-only, Meta-only, Hybrid) and logging requirements (Log Heavy, Log Medium, Log Light) as parameters for a transaction with total 1000 operations. In the LRU benchmark, *Data-only* corresponds to a *write* access operation to an object in the LRU list where both the LRU structure and object data are modified. *Meta-only* refers to a *read* operation to an object in the LRU list where the LRU structure is modified. *Hybrid* comprises of 75% of *Data-only* and 25% of *Meta-only* accesses. The logging requirement denotes the percentage of total operations requiring logging. For example, *Log-Heavy* requires logging for 90% of total operations performed in the LRU benchmark. All operations

TABLE VII: x86-64: L1-D cache miss and replacements with logging

Methods	Data-only	Meta-only	Hybrid
L1-D miss rate with logging			
Log Heavy			
vanilla	20.46	20.43	20.27
redo	18.44	18.52	17.76
undo	20.23	20.27	20.17
Log Medium			
vanilla	15.60	15.59	15.51
redo	14.65	14.81	14.58
undo	14.75	15.55	14.41
Log Light			
vanilla	11.27	11.33	11.26
redo	11.08	11.20	11.19
undo	10.59	11.28	11.16
% of change in L1-D replacements with redo			
Log Heavy	<0.1% ↓	<0.1% ↓	21.89% ↑
Log Medium	<0.4% ↑	<0.4% ↑	12.62% ↑
Log Light	<0.3% ↑	<0.3% ↑	3.18% ↑
↑ increase ↓ decrease w.r.t. vanilla			

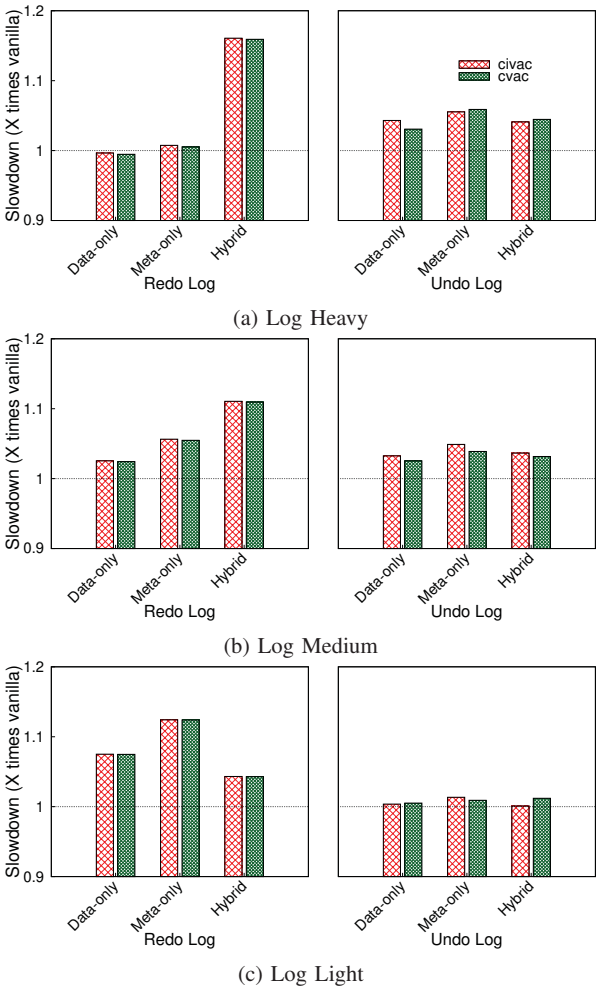
are performed under a single transaction demarcated between *tx_begin* and *tx_end*.

We observed that, using undo log caused slight performance slowdown for all modification categories and all types of logging requirements whereas, the redo log performed better for all modification categories except for Hybrid (Figure 7). With redo log, `clwb` performed marginally better than `clflush` and `clflushopt`. We also observed that, `clflush` or `clflushopt` performs better than `clwb` for some cases with undo log which suggests that `clwb` is not always the best. The benefit of using redo log can be explained by analyzing the reduction of L1-D cache miss rate (TableVII) for redo with `clwb`. We also noticed that L1-D cache write-backs are reduced (between 31% to 5%) in comparison with `vanilla` while using redo with `clwb` (because of log access locality).

One possible reason for comparatively higher performance slowdown for *Hybrid* with redo logging can be the L1-D cache pollution since L1-D replacements are increased with *Hybrid* as shown in Table VII for `clwb` with respect to `vanilla`. This cache pollution may be created by the hardware prefetcher at L1-D since we noticed a 37% increase in the number of prefetch requests issued for *Hybrid* compared to `vanilla`.

2) Performance with Arm64: We repeated the logging experiments (with same workload scenario as in Section IV-B1) for ARM using the Gem5 simulator. We observed that both redo and undo logging resulted in slowdown with all types of LRU access patterns and logging requirements as shown in Figure 8. Similar to the previous experiment, the slowdown shown in the figure is normalized to `vanilla`. `Cvac` performed better than `civac` with redo as `cvac` is functionally equivalent to `clwb`. Further, `cvac` also performed better with undo for most of the cases, especially with medium logging requirement.

The benefit of `cvac` with redo logging can be correlated with better cache performance since we observed lower L1-D miss rate while using redo with `cvac` in comparison to



(a) Log Heavy

(b) Log Medium

(c) Log Light

Fig. 8: Arm64: Influence of data consistency methods based on nature of update operations for different logging requirements. Y-axis values are slowdown w.r.t. vanilla case of no logging (lower the better).

`vanilla` as shown in Table VIII. Similar to x86-64 setup, the higher slowdown under *Hybrid* set of modifications with redo can be correlated with increase in L1-D replacements under redo compared to `vanilla`. We observed an increase in performance slowdown while using redo for meta-only modification with light logging requirements; this is because the completion time of LRU benchmark using redo with `cvac` for meta-only modifications increased by 1.18% (Log Medium), 3.4% (Log Light) as compared with data-only modifications, whereas `vanilla` case decreased by 1.16% (Log Medium), 1.11% (Log Light) as compared with data-only modifications.

C. Influence on co-running applications

Data consistency operations (flush+fence) may influence the performance of co-running applications due to the presence of shared resources such as LLC, internal buffers and interconnects. We studied the impact of data consistency methods

TABLE VIII: Arm64: L1-D miss rate with logging using cvac

Methods	Data-only	Meta-only	Hybrid
Log Heavy			
vanilla	21.41	21.47	21.39
redo	19.15	18.99	18.43
undo	21.16	21.22	21.12
Log Medium			
vanilla	16.64	16.68	16.61
redo	14.58	14.12	15.02
undo	16.57	16.58	16.54
Log Light			
vanilla	12.01	12.04	12.04
redo	10.34	9.69	11.42
undo	12.06	12.03	12.03

on the performance of co-running standard applications. We used a x86-64, 4-core GEM5 setup where core-0 executed SPEC CPU 2017 [32] and the remaining three cores executed a mix of different benchmarks selected from the set of micro-benchmarks (Table I). These micro-benchmarks were configured with large working set size. The performance of different SPEC benchmarks co-running with a mixture of micro-benchmarks was compared with different flush methods against the case when micro-benchmarks did not use any flush method. We observed that, among the SPEC benchmarks selected, except for 544.nab_r, there was no significant impact of the flush operations on the co-running application. There was a 5% slowdown for `nab` with `clwb` compared to the `noflush` scenario (because of LLC contention). We conclude that, the intermittent data consistency operations have negligible influence on the co-running applications because the persistent barrier operations do not significantly affect LLC and memory resource sharing aspects. However, it will be interesting to study a memory and cache congested scenario to analyze the extent of the performance implications which we leave as a future work.

Summary: Performance trends are similar across x86-64 and Arm64 ISAs. For applications exhibiting temporal locality, `clwb` (x86-64) and `cvac` (Arm64) primitives are comparatively better than others (`clflush`, `civac` etc.). We also observed memory barriers required for ordering contribute significantly to the overall performance overhead.

V. Related Work

Recent research in NVM systems proposed changes at different layers of the computing stack—system software, language runtimes, and application layer data structure design—to efficiently realize the advantages of NVM. Most of the proposed techniques use the underlying persistent primitives to address failure atomicity concerns in NVMs.

Application and runtime systems: Many existing works proposed changes to programming libraries and data structures to make them suitable for non-volatile memory by incorporating failure atomicity characteristics. These include memory allocators [15] [16], programming models [33] [20] and data structure modifications [12]–[14]. Schwalb et al. have proposed a user mode memory allocator for NVM with support from the underlying NVM aware file system [15]. Intel's PMDK [20] provides a set of libraries and tools for persistent memory programming. Similarly, NVthreads [33] and Atlas [9] provide programming models for multi-threaded NVM programs by inferring failure atomic section using synchronization points and tracking changes at OS page granularity. All of the above frameworks use architectural persistent barriers as the basic support technique for NVM consistency.

NoveLSM [13] uses mutable persistent memtables (stored on NVM) and provides optimistic parallel reads to multiple levels of LSM. NoveLSM uses an extended version of LevelDB to implement persistent memtables which in turn uses persistent barriers for crash consistency. Similarly, Zuo et al. [14] have proposed a write optimized hash indexing scheme for non-volatile memory using cache flushing.

Logging: Logging and versioning based techniques provide failure atomicity enabling a set of operations to complete or fail in an atomic manner. Venkataraman et al. [34] used versioning to enable failure recovery for single level store based consistent and durable data structures. Software based redo/undo logging mechanisms cause cache pollution because of additional memory operations. JUSTDO logging [8] minimized log size by each thread storing only the most recent store instruction executed within the failure atomic section. After failure, execution resumes at the last store instruction and executes the failed atomic section to completion. Most logging and versioning techniques use underlying architectural persistent barriers in their implementation. Wan Hu, et al. [25] compared the performance of redo and undo logging in persistent memory using PMDK. In this paper, we present a comprehensive comparison across different architectural alternatives with architectural insights and root-cause analysis.

Hardware and system software: Twizzler [2], an operating system designed for NVM, uses a data centric approach where memory objects reside in global object space and pointers are interpreted based on objects of residence as opposed to classical virtual addressing techniques. Kannan et al. [17] extended the OS virtual memory subsystem and exposed NVM as a memory node. In this scheme, each persistent object mapped to NVM pages is identified by an object identifier and logging is used to ensure consistency of OS data structures, objects and object meta-data.

All software based failure atomic solutions invariably incur the flush+fence overheads. Joshi et al. [5] have proposed hardware based undo logging by extending the cache controller to enforce write ordering. Zhao et al. [35] enabled in-place update to data structures without logging by ensuring order of writes to NVM using intelligent cache flush operations at the hardware level. Another requirement for recovery correctness after a failure is to reason about the order of writes to persistent memory. Steven et al. [36] proposed a memory persistence model to reason about the persist order after a failure. The authors have equated the order viewed by a recovery observer as a constraint on the persist order. The persist order can be relaxed in the same way as memory consistency model for better performance [37]. We believe, the comparative analysis

presented in this paper will be useful in future research striving for efficient data consistency mechanisms for NVM systems.

VI. Conclusion

Application state recovery in a consistent manner with NVM systems requires data consistency mechanisms supported by underlying architectural primitives like `flush` and `fence`. In this paper, we empirically analyzed the performance overhead of data consistency methods on x86-64 and Arm64. The study shows that the performance overhead of methods depends upon the nature of workload, proportion of read-to-write operations and working set size. The study also reaffirms that the usage of serializations operations such as `sfence` to enforce order of writes to NVM contributes significantly to the performance overhead for all data consistency methods. We observed that while `clwb` benefits applications with cache locality, it may not be always advantageous to use `clwb` over `clflushopt` for better performance. For example, `clwb` benefits redo log schemes while `clflush` or `clflushopt` performs better than `clwb` for some cases with undo log. The study concludes that, the selection of data consistency methods should be decided on a case-by-case basis depending on the workload characteristics.

References

[1] (2020) The intel® optane™ persistent memory website. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[2] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller, "Twizzler: a data-centric OS for non-volatile memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/bittman

[3] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: durable hardware transactional memory," in *2018 ACM/IEEE 45th ISCA*. IEEE, 2018, pp. 452–465.

[4] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of ASPLOS*, 2020, pp. 335–349.

[5] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on HPCA*. IEEE, 2017, pp. 361–372.

[6] (2020) The persistent memory development kit website. [Online]. Available: https://pmem.io/pmdk/

[7] (2020) Persistent memory programming website. [Online]. Available: https://pmem.io/book/

[8] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[9] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[10] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of ASPLOS*. ACM, 2019, pp. 441–454.

[11] T. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *51st Annual IEEE/ACM MICRO*. IEEE, 2018, pp. 507–519.

[12] R. Xiao, D. Feng, Y. Hu, F. Wang, X. Wei, X. Zou, and M. Lei, "Write-optimized and consistent skiplists for non-volatile memory," *IEEE Access*, 2021.

[13] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *USENIX Annual Technical Conference*, 2018, pp. 993–1005.

[14] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *USENIX OSDI*, 2018, pp. 461–476.

[15] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." *ADMS@ VLDB*, vol. 15, pp. 61–72, 2015.

[16] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–26, 2017.

[17] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the EuroSys*, 2016, pp. 1–16.

[18] M. Cai and H. Huang, "A survey of operating system support for persistent memory," *Frontiers of Computer Science*, vol. 15, no. 4, pp. 1–20, 2021.

[19] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: programming with multiple virtual address spaces," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 353–368, 2016.

[20] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[21] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, p. 5, 2011.

[22] (2020) Arm cortex-a series programmer's guide for armv8-a website. [Online]. Available: https://developer.arm.com/documentation/den0024/a/preface

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[24] A. Baldassin, J. a. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," vol. 54, no. 7, 2021.

[25] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.

[26] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[28] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[29] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Improving phase change memory performance with data content aware access," *arXiv preprint arXiv:2005.04753*, 2020.

[30] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[31] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[32] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42.

[33] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of EuroSys*, 2017, pp. 468–482.

[34] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[35] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *46th Annual IEEE/ACM MICRO*, 2013, pp. 421–432.

[36] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 265–276.

[37] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 652–665.