Persistent Memory Transactions*

Virendra J. Marathe[1] *Achin Mishra*[2] *Amee Trivedi*[3] *Yihe Huang*[4] *Faisal Zaghloul*[5]
Sanidhya Kashyap[6] *Margo Seltzer*[1,4] *Tim Harris*[1] *Steve Byan*[1] *Bill Bridge*[7] *Dave Dice*[1]

[1]*Oracle Labs* [2]*IBM* [3]*University of Massachusetts, Amherst* [4]*Harvard University*
[5]*Yale University* [6]*Georgia Institute of Technology* [7]*Oracle Corporation*

Abstract

This paper presents a comprehensive analysis of performance trade offs between implementation choices for transaction runtime systems on persistent memory. We compare three implementations of transaction runtimes: *undo logging*, *redo logging*, and *copy-on-write*. We also present a memory allocator that plugs into these runtimes. Our microbenchmark based evaluation focuses on understanding the interplay between various factors that contribute to performance differences between the three runtimes – read/write access patterns of workloads, size of the *persistence domain* (portion of the memory hierarchy where the data is effectively persistent), cache locality, and transaction runtime bookkeeping overheads. No single runtime emerges as a clear winner. We confirm our analysis in more realistic settings of three "real world" applications we developed with our transactional API: (i) a key-value store we implemented from scratch, (ii) a SQLite port, and (iii) a persistified version of memcached, a popular key-value store. These findings are not only consistent with our microbenchmark analysis, but also provide additional interesting insights into other factors (e.g. effects of multithreading and synchronization) that affect application performance.

1 Introduction

Byte-addressable persistent memory technologies (e.g. *spin-transfer torque MRAM (STT-MRAM)* [12, 13], *memristors* [30]), that approach the performance of DRAM (100-1000x faster than state-of-the-art NAND flash) are coming, as evidenced by Intel and Micron Technologies' recent announcement of their 3D XPoint persistent memory technology [1]. While the simple load/store based interface of these technologies is appealing, it introduces

```
void *p; // pointer to persistent memory
...
// obj and clone(obj) are persistent
p = clone(obj);
```

Figure 1: Example illustrating complexities of programming with just the hardware instructions for persisting data. p is a pointer hosted in persistent memory. clone clones its argument object (obj). The programmer must persist the clone before p's assignment, otherwise an untimely failure could result in a state where the clone is not persisted but p's new value is persisted.

new challenges; a simple store does not immediately persist data, because processor state and various layers or the memory hierarchy (e.g., store buffers, caches) are expected to remain *nonpersistent* for the foreseeable future. Prior research [9, 17, 25] and processor vendors, such as Intel, have proposed new hardware instructions [14] to flush or write cache lines back to lower layers in the memory hierarchy and new forms of *persist barrier* instructions that can be used to order persistence of stores. However, even with these new instructions, correctly writing programs to use them remains a daunting task. Figure 1 illustrates this challenge – the programmer must carefully reason about the order in which updates to various pieces of the application's persistent data structures are persisted. Omission of even a single flush, write back, or persist barrier instruction can result in persistent data inconsistencies in the face of failures.

These programming challenges have resulted in investigations of transactional mechanisms to access and manipulate data on persistent memory [5, 6, 8, 10, 18, 26, 33]. However, we observe that research on transaction runtime implementations in this new context is in its early stages. In particular, researchers and practitioners appear to have favored or rejected an implementation strategy based on intuition (e.g. redo logging is a bad idea since redo log lookups for read-after-write scenarios are expensive [5, 6, 8, 18, 26]) or inadequate evaluation (e.g.

*Work done when all authors were affiliated with Oracle. Contact: virendra.marathe@oracle.com

evaluation only on write-heavy workloads [10]). No one to our knowledge has endeavored to do a comprehensive analysis of performance trade offs between these implementations over a wide swath of workload, enclosing system, and transaction runtime choice parameters. This paper presents a holistic approach toward understanding these performance trade offs.

We consider the implications of *persistence domains* [31] on persist barrier overheads (see § 2). Briefly, a persistence domain is the portion of the memory hierarchy that is considered to be "effectively persistent" – the underlying hardware/software system ensures that data that reaches its persistence domain is written to the persistent media before the system is shut down, either planned or due to failures. We introduce a new taxonomy of persistence domain choices enabled by different hardware systems.

We present our three transaction runtimes based on *undo logging*, *redo logging* and *copy-on-write* implementations of transactional writes (see § 3). We also present our memory management algorithm that plugs into all three runtimes (see § 4). All our runtimes, including the memory manager, have been optimized to reduce the number of persist barriers required to commit a transaction.

Our microbenchmarking (see § 5), performed on Intel's Software Emulation Platform [27, 34], comprehensively sweeps through read-write mix ratios within transactions and shows how performance trends in the transaction runtimes change as the read-write mix within transactions changes, and over a wide range of persist barrier latencies. Our analysis reveals the significant influence of a combination of factors – read/write mix, transaction runtime specific bookkeeping overheads, persist barrier latencies, and cache locality – which determines the performance of a runtime. Because of the interplay of these factors, no single runtime's performance dominates the rest in all settings. We find similar performance trade offs in three "real world" workloads: (i) a key-value store we developed from scratch, (ii) a port of SQLite, and (iii) a port of memcached. The benchmarks provide insights in additional factors that influence performance (e.g. effects of multithreading and synchronization, overheads in other parts of the application).

2 Persistence Domain

While data hosted in persistent memory DIMMs is expected to survive power failures, the rest of the memory hierarchy (e.g. processor caches, memory controller buffers, etc.) is fundamentally not persistent. However, system solutions do exist that make various parts of the memory hierarchy "effectively persistent". For instance, in battery backed systems [7, 16, 23, 24], where the processor

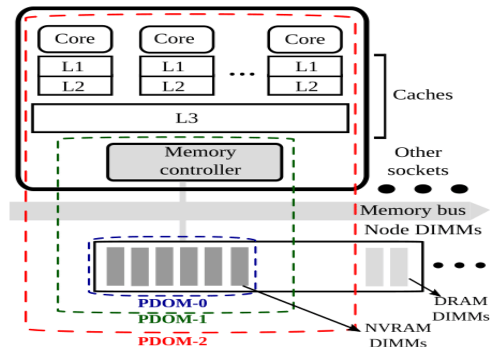

Figure 2: Persistence domains of a near future processor socket that hosts persistent memory DIMMs.

Operations	Persistence domains		
	PDOM-0	PDOM-1	PDOM-2
Writes	store clwb/clflush-opt	store clwb/clflush-opt	store
Ordering persists	sfence pcommit sfence	sfence	nop

Table 1: Persistent memory primitives needed on Intel's upcoming processors [14] for different persistence domains.

state and caches can be flushed out to persistent memory DIMMs on power failure, the whole memory hierarchy effectively becomes persistent. Another example is the *asynchronous DRAM refresh (ADR)* feature provided by modern processors, where the memory controller buffers are flushed out to memory DIMMs on power failure [15]. With the ADR feature, the memory controller buffers can be considered effectively persistent since the data is guaranteed, discounting ADR hardware failures, to persist. There may be other ways to slice the memory hierarchy in persistent and nonpersistent parts; however, we focus on 3 specific partitioning strategies that we believe will capture most future system configurations.

A *persistence domain* [31] as the portion of memory hierarchy where data is effectively persistent. As shown in Figure 2, we classify persistence domains in three categories: (i) PDOM-0, which contains only the persistent memory DIMMs. (ii) PDOM-1, which includes PDOM-0 and memory controller buffers. Modern processors with ADR capabilities and persistent memory DIMMs effectively support PDOM-1. (iii) PDOM-2, which includes the entire memory hierarchy as well as processor state, such as store buffers, containing persistent data. Battery backed systems support PDOM-2.

The persistence domain affects the instruction sequence needed to persist updates. Table 1 depicts the instructions needed to persist these updates on (near future) Intel processors with persistent memory [14]. There are two phases to the persistent update process: (i) The actual write (i.e., store) and (ii) the persist barrier. PDOM-0 and PDOM-1 require a flush instruction in addition to the store to move data into the persistence domain. Both

the clwb and clflush-opt trigger asynchronous cache-line sized writes to the memory controller; they differ in that clflush-opt invalidates the cache line while clwb does not. In principle, the flush instructions can be delayed, and almost certainly should be for multiple store instructions to the same cache line. In practice, as they are asynchronous, starting the writeback sooner speeds up the persist barriers in the second phase of this process. In PDOM-2, flush instructions are not needed, since store buffers and caches are part of the persistence domain.

In PDOM-0, the persist barrier needs to ensure that all flushes have completed (the first sfence), and then force any updates in the memory controller to be written to the DIMMs (pcommit). As the pcommit is asynchronous, persistence requires the second sfence to indicate when the pcommit has completed. In PDOM-1, the persist barrier need only ensure that prior flushes have completed, since the memory controller now resides inside the persistence domain. PDOM-2 requires no further action as data is persisted as soon as it has been stored. Intel has recently deprecated the pcommit instruction [15]. However, we include it in our discussion as a concrete example of a PDOM-0 persistence domain. Note that clwb, clflush-opt, and pcommit have store semantics in terms of memory ordering, and applications must take care to avoid problematic reordering of loads with these instructions, using sfence or other instructions with fence semantics.

3 Persistent Transactions

Similar to prior works [5, 6, 8, 33], our programming model is based on the abstractions of persistent regions, persistent data types, and transactions. A persistent region is a contiguous portion of the application's address space populated by a memory mapped file that is hosted in persistent memory. The region can be accessed with the load/store interface, hosts a heap, and a user instantiated root pointer. The heap provides pm_alloc and pm_free functions, callable only from transactions.

Our work focuses on supporting the needs of skilled system software programmers, who require programming support for just *failure atomicity* – across failure boundaries, either all updates of transactions persist or none of them do. These programmers are adept at manually using synchronization techniques to avoid data races in concurrent settings. Hence, while various semantic models for persistent memory transactions have been explored – full ACID transactions in the spirit of transactional memory [8, 33], failure-atomic critical sections [6], and failure-atomic transactions [5, 10, 18, 26] – we focus on failure atomic transactions. Our programming model, implemented as a C library, supports transaction begin and commit operations, as well as transactional accessor macros.

```
struct foo {
    int cnt;
};
// pm_foo, the persistent version of type foo
DEFINE_PM_TYPE(foo);
// x points to an instance of pm_foo
pm_foo *x;
// failure-atomic transaction for x->cnt++;
pm_txn_t txn;
do {
    TXN_BEGIN(txn);
    int counter; // temporary
    TXN_READ(txn, x, cnt, &counter);
    counter++;
    TXN_WRITE(txn, x, cnt, &counter);
    status = TXN_COMMIT(txn);
} while (status != TXN_COMMITTED);
```

Figure 3: Example of a simple transaction that increments a counter in a persistent object.

We also provide macros to define persistent types, which act as wrappers around traditional data types. Figure 3 provides a brief example illustrating our basic API. We also provide common memory buffer operators such as memcpy, memcmp, and memset

Our transactional accessors introduce runtime overheads. For programmers that want to elide these overheads without compromising correctness of their applications, we provide the PM_UNWRAP macro that returns a pointer to the data type instance wrapped by a persistent type instance. This tool can be useful in special circumstances including accessing objects in read-only mode, and initializing newly allocated objects.

The primary focus of our work is on understanding the performance trade offs between different implementations of persistent memory transactions. To that end we have developed three different transaction runtime systems: (i) undo logging, (ii) redo logging, and (iii) copy-on-write (COW). All the runtimes store transaction metadata in a persistent data structure called the *transaction descriptor*, which is assigned to a thread as part of TXN_BEGIN. A descriptor is always in one of four states: IDLE, RUNNING, ABORTED, or COMMITTED. A descriptor that is not in use is in the IDLE state. TXN_BEGIN transitions the descriptor into the RUNNING state. A transaction commits by entering the COMMITTED state and aborts by entering the ABORTED state. After the runtime cleans up a descriptor's internal state and buffers, the descriptor returns to the IDLE state. During its execution, a transaction may read, write, allocate, and deallocate persistent objects using our API.

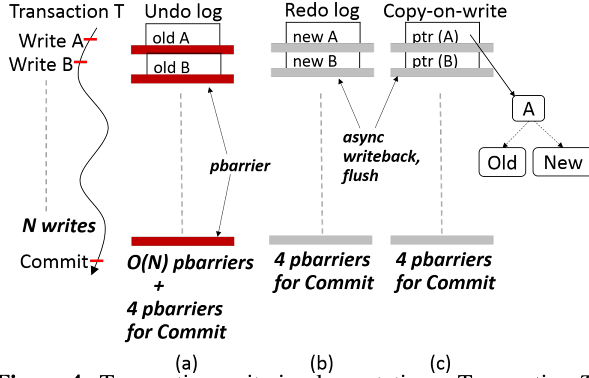

Figure 4: Transaction write implementations. Transaction T applies N distinct writes. pbarrier is a potentially high latency persist barrier, async wb/flush are asynchronous cache line write-back or flush instructions respectively.

3.1 Undo Log based Transactions

Figure 4(a) shows transaction T's undo logging activities. The undo log is implemented as a simple chunked list. Transaction T writes w, using TXN_WRITE producing a log record containing the original value of w. This log record must be persisted before w is modified. A typical implementation of the undo log append requires two persist barriers – one to persist the new log record and one to change the log's tail pointer. Although correct, this approach leads to $2N$ persist barriers for N appends, which results in high overheads.

Our implementation requires only a single persist barrier per record. Instead of relying on a correct tail pointer during recovery, we infer the tail of the log. We assign each transaction a monotonically increasing persistent 64-bit version number. Each undo record contains a transaction version number, a 64-bit checksum, and a 64-bit *prolog* sentinel value that appears at the beginning of the undo record. So, we write the prolog sentinel value, the transaction's version number, the record contents, and then compute and write the checksum. Then we issue a single persist barrier. If a failure occurs before or during execution of the persist barrier, and only part of the undo record has been persisted, we will detect a checksum mismatch during recovery. We also maintain the log tail pointer, but update it *after* the persist barrier, so the tail update is guaranteed to persist on or before the next record write and persist. Recovery can rely on the following invariant: the tail pointer will be at most one record behind the actual end of log. So log recovery requires only that we examine the record after the current end of log and determine if there is a valid log record present.

One of the most compelling benefits of undo logging is that transactional reads are implemented as uninstrumented loads [8, 5, 18, 26, 33].

An undo log transaction commits in four steps: (i) First it ensures that all transactional writes are persisted, which requires one persist barrier. (ii) Then it logically commits the transaction by appending the commit record to the transaction's undo log. (It also switches the transaction's state to COMMITTED, but that does not have to persist.) Steps (iii) and (iv) are largely related to transactional metadata cleanup, which requires persistence only if the transaction allocated or deallocated persistent memory (see § 4). (iii) Persist the allocation/deallocation calls' effects and cleans up the transaction's metadata. (iv) Mark the transaction IDLE; this state change needs to be persisted only if the transaction did allocations/deallocations.

3.2 Redo Log based Transactions

Figure 4(b) shows transaction T's redo logging activities. Like the undo log, the redo log is implemented as a simple chunked list. Transaction T writes w, using TXN_WRITE producing a log record containing the new value of w. The record need not persist at the time of the write; if a failure occurs, the entire redo log can be discarded. However, an implementation, like ours, may proactively schedule a low latency asynchronous writeback/flush of the record.

The challenge for redo logging schemes is handling read-after-write accesses. As the new value appears only in the log, a subsequent read must consult the log for the latest value. A naive implementation could walk down the whole log looking for the latest value. Furthermore, the redo log lookup could be done for *every* subsequent read done by the transaction. The resulting overhead can be significant. We apply two optimizations to overcome these overheads.

First, we add a 64-bit bitmap to each persistent object's header to indicate current writers of that object. A writer first sets its corresponding bit during the write (the bit is cleared after the transaction completes). A read checks the object's writers bitmap to determine if a redo log lookup is necessary, and does on if so. If a lookup is not necessary, the read becomes an uninstrumented load. Each transaction maps to a unique bit in the object's writers bitmap. Up to 64 transactions can concurrently "own" a bit in the writers bitmap. This can be easily extended to larger bitmaps, but at present we force additional transactions to consult the redo log for all their reads.

Second, we avoid scanning the entire log by maintaining a per-transaction hash table indexed by persistent object base addresses. The hash table record points to the latest redo log record for that object. Each such record also contains a pointer to the previous redo log record for that object, if one exists. We effectively superimpose a linked stack of records for each object within the redo log. This avoids unnecessary traversal of unrelated log records

during a redo log lookup.

Committing a transaction requires persisting the redo log. After the persist completes, the transaction logically commits by updating its state to COMMITTED, and then persists the new state with a second persist barrier. After the logical commit, the runtime applies the redo log to each modified object and issues a third persist barrier. Finally, we mark the transaction IDLE, and persist it. In total, the redo logging implementation requires four persist barriers for commit, but none on abort.

3.3 Copy-on-Write based Transactions

Our copy-on-write (COW) implementation introduces an extra level of indirection between a persistent type instance (the wrapper) and the real data type instance (payload) it encloses. As shown in Figure 4(c), the persistent type contains pointers to *old* and *new* versions of the enclosed type's instances. Before modifying an object, a transaction creates a new copy of the payload. We provide a special TXN_OPEN API that applications can use to obtain read-only or read-write access to a persistent object:

```
TXN_OPEN(txn, obj, mode, copy_ctor);
```

where mode is either read-only or read-write, and copy_ctor is the copy constructor. The copy constructor can be used to clone specialized objects (e.g. linked structures, self-relative pointers, etc.). A NULL copy constructor will default to using memcpy.

Each transaction descriptor maintains a *write set* containing the list of objects the transaction has written. Objects are added to the write set in TXN_OPEN invocations with read-write mode. Object wrappers also contain the writing transaction's ID (assuming at most one writer per persistent object), which is used to direct transactional reads to the appropriate payload copy.

Payload copies, as well as writes to their wrappers, need not be persisted during the writer's transaction. The transaction's write set and the objects it writes to are persisted using a single persist barrier at the beginning of the commit operation. Then, the runtime updates the transaction's state to COMMITTED and persists it.

The post-commit cleanup requires four steps: (i) make the modified (new) object payload the real (old) payload, (ii) reset new to NULL, (iii) discard (deallocate) the old payload, and (iv) clear the writer's ID from the wrapper. This process is susceptible to memory leaks: a failure between steps (i) and (iii) can result in the reference to the old payload being lost. We avoid this leak by adding another field in the wrapper, called old_backup, that is set to point to the old payload in TXN_OPEN. This update is persisted during the first persist barrier in the commit operation. old_backup is used to deallocate the old payload. Next, the transaction's allocations/deallocations are all persisted. The third persist barrier is issued after all this cleanup. Then, the transaction updates its state to IDLE and persists it using a fourth persist barrier. This ensures that no further cleanup is needed. Finally, we clear the transaction's ID from all the objects to which it wrote. If a transaction aborts, only the last two clean up related persist barriers are needed for correct rollback.

4 Persistent Memory Management

Memory management is a foundational tier in any software stack. We anticipate that applications using transactions to access persistent data will routinely allocate and deallocate persistent objects within these transactions. Most previous work on persistent memory management focuses either on wear-leveling [22] or techniques for correct allocation that tolerates failures [5, 8, 27], disregarding the overhead due to persist barriers. Volos et al. [33] present an algorithm that effectively eliminates persist barriers for memory allocation/deallocation calls within a transaction. But that works only in their redo logging transactions. Our algorithm is similar in nature, but works with all of our transaction runtimes.

We build our algorithm on previous approaches that separate the allocator's metadata in persistent and nonpersistent halves [27, 33]. Our allocator is modeled after the Hoard allocator [4], where the heap is divided in shared and thread-private superblocks. Each superblock, hosted in persistent memory, contains a persistent bitmap indicating allocation status of corresponding blocks, and nonpersistent metadata (free and used lists) hosted in DRAM. A superblock is protected by a nonpersistent lock.

Each transaction maintains a persistent private *allocation log* that consists of all the allocation/deallocation requests made by the transaction. In a pm_alloc call, the nonpersistent metadata of the superblock is updated by the transaction and a corresponding record is appended to its allocation log. pm_free simply appends an entry to the allocation log.

The first persist barrier in a transaction's commit operation persists the allocation log as well. Once the transaction persists its COMMITTED state, operations in the allocation log are reflected in the persistent metadata (bits are flipped using compare-and-swap instructions to avoid data races, and then the cache lines are written back or flushed). The post-commit cleanup phase's first persist barrier persists these flipped bits, and the last persist barrier marks the transaction as IDLE. Note that pm_free calls' nonpersistent heap metadata (free and used lists of a superblock) is updated *after* the cleanup persists.

5 Empirical Evaluation

Our performance evaluation comprises two parts: (i) microbenchmarking, where we sweep through a comprehensive range of read/write mixes within transactions to identify performance patterns of the transaction runtimes over changing read/write proportions under varied assumptions about persist barrier latencies; and (ii) evaluation of three "real-world" applications – a new persistent key-value store we developed, a port of SQLite [29] that uses our transactions to persist the database, and a persistent version of memcached [20] – that confirms, and adds to, our findings reported in the micrbenchmarking part.

We conducted all our experiments on Intel's Software Emulation Platform [27, 34]. This emulator hosts a dual socket 16-core processor, with 512GB of DRAM. 384GB of that DRAM is configured as "persistent memory" and 128GB acts as regular memory. Persistent memory is accessible to applications via mmapping files hosted in the PMFS instance [27] installed in the emulator.

The emulator emulates the clflush-opt and pcommit instructions. The clflush-opt is implemented using the clflush instruction. Since clflush-opt evicts the target cache line, we expect it to lead to significant increase in cache miss rates, thereby degrading application performance; 2-10X in our microbenchmarking, which we do not report in detailed due to space restrictions. We therefore focus our evaluation on the clwb instruction, which does not evict the target cache line, and is likely to be the instruction of choice for applications on Intel platforms. We emulate clwb behavior with a nop (it is not supported in the emulator); the actual latency of persisting the data is incurred by the subsequent persist barrier, which we emulate by using the emulator's pcommit instruction. We note that Intel recently announced deprecation of the pcommit instruction [15] from future Intel processors; the persist barrier in that case is simply an sfence instruction (see Table 1). However, in the emulator, the pcommit instruction simply stalls the calling thread for a configurable amount of time [27, 34], which lets us experiment with different latencies of the real persist barrier – it will be difficult to know the real latency of a persist barrier until real hardware becomes available, so any evaluation needs to target a broad range of latencies, which we do in our experiments. Latency of loads from persistent memory is a configurable parameter as well. store latency in the emulator is the same as DRAM store latency.

We conducted experiments over a wide range of latency parameters for persist barriers (0 – 1000 nanoseconds) and loads (100 – 500 nanoseconds). We report results for a load latency of 300 nanoseconds, and 3 different persist barrier latencies – 0, 100 and 500 nanoseconds, labeled as PDOM-2, PDOM-1, and PDOM-0 respectively to map them to the different persistence domains from our taxonomy. These latencies represent the overall performance trends we observed over the broader range of latencies.

5.1 Transaction Latency

Our latency microbenchmarking focuses on understanding performance of the transaction runtimes under different read/write loads. The synthetic Array microbenchmark developed by the SoftWrAP work [10] suffices this purpose. Array contains a 2-dimensional array of 64-bit integers hosted in persistent memory. The first dimension contains 10 million slots (each is an array of integers); the second dimension's (slot's) size is configurable, we vary it from 1 (8 bytes) to 64 (512 bytes) entries. Array continuously runs transactions, each of which randomly accesses a contiguous set of 20 slots. The slot sizes cover a broad range of access granularities. Each slot access can be a simple read of all the integers in the slot, or a read-write that increments all integers in the slot. We vary the number of slots accessed in read-only or read-write mode for different test runs. In addition, we implemented two versions of slot writes: (i) a "one shot" update version, called Array, where the transaction copies the slot integers in a private (nonpersistent) buffer, increments the integers in that buffer and then writes it back to the persistent slot; and (ii) a "read-after-write intensive" version, called Array-RAW, where each integer in the slot is individually incremented "in-place". The second version helps us understand overheads related to read-after-write accesses in the redo logging transaction runtime. We report results as the mean of three 10-second test runs preceded by a 10-second warmup phase (we observed less than 5% deviation from the mean in all results). Array microbenchmark is single threaded and is sufficient for latency measurements.

Figure 5 shows latency results of our experiments for slot sizes of 4 and 64 (they capture the performance patterns of the configurations with other slot sizes we tested), and over the three different persist barrier latencies discussed above. Latency graphs for Array appear in Figure 5(a)–(f). The first takeaway of these is that COW performs worst across the board. In fact the margin grows from 2X to 50X compared to the best performing alternative when we move from 4-integer slots to 64-integer slots. The overheads of COW are largely related to worse cache locality (proportionally high cache miss rates) compared to the undo and redo logging alternatives – (i) the extra level of indirection, and (ii) the constant cloning of objects as they are updated. This gets worse with increasing granularity of objects.

As expected, undo logging performance degrades with

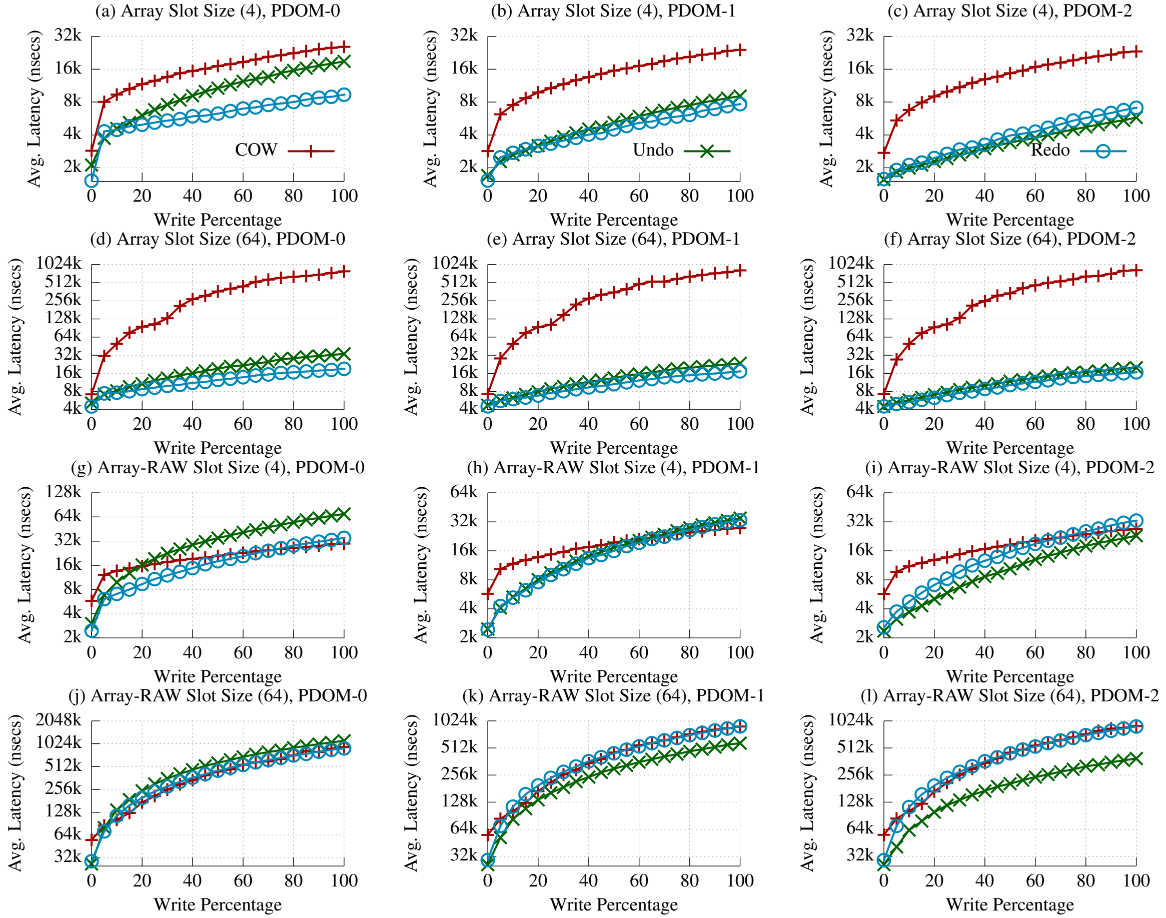

Figure 5: Average latency of transactions with increasing write percentage. Y-axes are in log scale. Each transaction accesses 20 slots. X-axes start with read-only access runs; we progressively increase read-write accesses as we move right.

increasing persist barrier latencies. This impact is most noticeable for PDOM-0, where the persist barrier overheads are high – the barrier overhead in undo logging tends to increase the latency gap (up to 2X) with redo logging as the percentage of writes per transaction increases (Figure 5(a),(d)). The same behavior manifests in PDOM-1 configurations (Figure 5(b),(e)), albeit at a lower scale (up to 20% higher latency than redo logging), since the persist barrier latencies are lower.

For PDOM-2 however, the persist barrier is a nop. This shows inFigure 5(c), where the slot size is 4. Undo logging either performs as well, or better than redo logging. On further observation, we realized that redo logging actually performs increasingly worse than undo logging as the write percentage grows (up to 25% worse at 100% writes). Furthermore, at 0% writes, redo and undo logging are comparable in performance. This implies that the source of overheads is in the writes done by redo logging transactions. We determined that the operations related to maintaining the lookup structure for accelerating read-after-write lookups was the source of these overheads. Figure 5(f) presents a contrasting result, where, inspite of zero persist barrier latency, undo logging performs increasingly worse (up to 15%) than redo logging as the write percentage per transaction grows. We determined that the overheads in undo logging were related to the checksum computations we needed to avoid an extra persist barrier per undo log append. Eliminating the checksum brought undo and redo logging performance at parity in these test runs. This indicates an interesting trade off in performance of undo logging implementations, where it is best to use checksums for transactions that do fine grain writes, whereas it may be best to use 2 persist barriers per undo log append for transactions that make coarse grained updates.

Notice that for read-only test runs (leftmost points in these graphs), redo logging is either at parity with undo logging, or slightly better (Figure 5 (a)). Our persistent

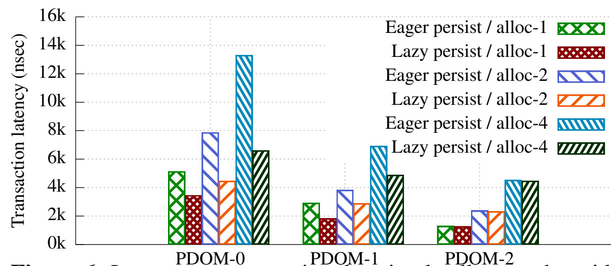

Figure 6: Latency per transaction that simply allocates the said number of blocks (Alloc-X), each of a random size from 1 to 512 bytes.

object header checks to detect read-after-write scenarios appear to have no effect on redo logs performance. In the absence of real read-after-write scenarios, all the overhead in redo logging appears to be related to writes that do the bookkeeping needed for fast read-after-write lookups.

Figure 5(g)–(l) show performance of the transaction runtimes on Array-RAW as the percentage of read-after-write instances increases. While the performance of COW transactions remains more or less identical to their performance on Array, both redo and undo logging transactions perform relatively worse. This is directly attributable to the proportional amount of churn happening on the redo/undo logs (one log record per integer increment). For PDOM-0 test runs, the high persist barrier latency combined with amplified number of persist barriers (4X or 64X) in undo logging, leads to worst performance (even in comparison with COW transactions) beyond a modest percentage of writes. Redo logging, on the other hand, incurs overheads related to read-after-write lookups (through the transaction's lookup table and per-object update lists), which are more modest than the persist barrier overheads, but are nonethelees high enough to force redo logging perform as badly as COW transactions for a modest percentage of writes. At PDOM-1 persist barrier latencies however, these lookups turn out to be relatively more expensive, because of which undo logging performs comparably or better than redo logging. This difference increases furthermore for PDOM-2 where the persist barrier is a nop. These results affirm the overall intuition of read-after-write lookup overheads in redo logging in prior work [5, 6, 8, 26]. The critical question of how often do such instances arise in real world applications is something we address later in our evaluation.

5.2 Memory allocation performance.

Figure 6 shows memory allocation latency, comparing the Eager Persist approach that uses persist barriers per allocation/deallocation call, to our Lazy Persist approach that avoids persist barriers altogether during allocation/deallocation calls. There is no performance difference in PDOM-2, because the persist barrier is a nop. However, for PDOM-1, the optimization produces a 20–30% latency improvement. In PDOM-0, the improvement grows to 30–100%, because the persist barrier latency is much higher.

5.3 Persistent Key-Value Store

We implemented a persistent key-value (K-V) store from scratch using our transactional interface. The implementation served as a vehicle to test the programmability limits of our transaction runtimes. Our K-V store's central data structure is a concurrent, closed addressed, hash table that uses chaining to resolve hash collisions. The K-V store supports string-type keys and values, and provides a simple get/put interface. Clients connect to the K-V store via UNIX domain socket connections. The K-V store spawns a thread for each connected client (we plan to extend our implementation to let server threads handle multiple clients concurrently).

We started from an implementation that makes use of our transactional API to perform all persistent data accesses. We introduced persistent types for all the persistent data structures. This introduces "wrapper" objects for all the persistent objects hosted in our K-V store. The wrapper objects introduce a level of indirection and hence overhead. We also implemented a hand-optimized version of the K-V store that avoids use of persistent wrapper types altogether. Our optimized version also aggressively bypasses the transactional accessors wherever possible – e.g. for read accesses, we can bypass the TXN_READ accessors and fetch the data directly from the target address in cases where the transaction has not yet written to the address. These optimizations can be trivially supported with the undo and redo log runtimes. COW, however, appears to have a fundamental limitation – it relies on the persistent wrappers to perform the copy-on-write. As a result, it is not possible to build such a version of our K-V store using COW transactions. This is a significant limitation of the COW transaction interface.

Figure 7 shows performance of our various K-V store versions for client (hence worker) thread count ranging from 1 to 8. All client threads are bound to 1 processor socket of the emulator, while the worker threads are bound to the other socket. Due to space restrictions, we report just the PDOM-2 numbers. COW experiences cache locality overheads and performs worse than redo and undo logging, which perform comparably. We observed 12% and 14% consecutive drops in performance for both redo and undo logging when we ran the same experiments with PDOM-1 and PDOM-0 configurations. COW also experiences similar performance drops. Our optimizations of bypassing accessors, persistent wrappers, and locality-friendly data placement deliver performance

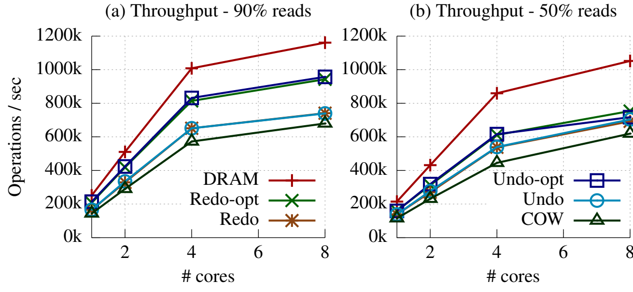

Figure 7: Persistent K-V store throughput. Suffixes "-opt" denote optimized versions. Performance is measured assuming PDOM-2. Each utilized core serves one connected client, who issues 1 million operations according to the specified read-write ratio. The keys are randomly drawn from a space of 238,328 keys, and the K-V store is pre-populated by 1 million random puts. Averages of 3 consecutive runs are reported. We omit error bars because of low variabilities in these benchmarks.

improvements of as much as 27% under read-dominated workloads. This difference comes down to 8% for redo logging and 2% for undo logging under the 50% reads workload.

Overall, we observe negligible difference between redo and undo logging versions (both base and optimized versions). The common case hash table accesses (gets and puts) are extremely short transactions, accessing a few cache lines, and updating even fewer (1–2) cache lines transactionally (e.g. linking or unlinking a node from a hash table bucket). Additionally, they do not contain any read-after-write accesses. Furthermore, the workers receive requests from clients over a TCP connection, which itself dominates the latency of client operations. This represents a possibly significant class of real world workloads, where differences in these lower level abstraction implementations may not matter to overall performance of the application.

5.4 SQLite

SQLite [29] is a popular light-weight relational database. It hosts the entire database in a single file, with another file for logging (rollback or write-ahead log) that is used to ensure ACID semantics for its transactions. SQLite can also be configured to an "in-memory" mode where the storage tier is completely removed. The database does not provide durability guarantees in that configuration. We have extended this configuration to use our transactional API for persistence.

SQLite stages all changes to the database at a page granularity in a *page cache*, and writes them out to the database file at the commit phase. For in-memory databases, the page cache is still populated, but it does not get persisted during the commit. We built a persistent version of in-memory SQLite by creating a "region file"

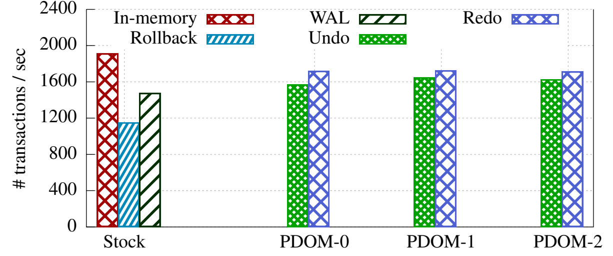

Figure 8: TPC-C, as implemented by PyTpcc [32], results on SQLite. For each persistence domain, we present both undo and redo logging performance. In-memory databases and stock databases – default unmodified SQLite with files hosted in PMFS – are presented for comparison. In our transactional version, all transactions are write-only (no reads).

based on our persistent region abstraction that is written to, and persisted, during the commit phase using our transactional API. The dirty list is essentially applied in a single transaction, thus making the transaction's effects durable in a failure atomic way. Our transaction effectively plays the role of rollback or write-ahead logs in the stock SQLite configuration.

Since the region file is a single large object, COW transactions would entail prohibitively high overheads, which is why we did not implement SQLite with COW transactions. We however tested our SQLite port with both undo and redo log transactions, and compared them with the in-memory configuration, and default SQLite databases whose files – both the database file, and journal file – were hosted in PMFS (we tested memory-mapped files, which is a feature supported by SQLite, but the results were identical to the default SQLite configurations). Our modifications and tests were carried out on SQLite 3.13.0.

On all mentioned configurations, we ran the TPC-C [32] benchmark, as implemented by PyTpcc [32]. Figure 8 shows the results. For each configuration, we took the average of 3 runs. As expected, the in-memory version has highest throughput All our redo logging transactions have comparable throughput, just 10% under the in-memory version. Undo logging transactions in PDOM-1 and PDOM-2 configurations have 2-3% lower throughput than the redo logging ones; all this overhead is attributable to the checksum overhead incurred for the page-granular transaction writes. Furthermore, the PDOM-0 throughput of undo logging is 6% lower than the throughput of redo logging; this is where the 500 nanosecond persist barrier latency shows up, approximately half of that overhead is because of the persist barriers. This performance nicely tracks the performance we observed in the Array microbenchmark with slot size of 64. We expect such performance patterns to emerge in applications that do lots of coarse grain writes. The unmodified SQLite with a rollback journal on PMFS are about 36% to 50% slower

than the in-memory database.

5.5 Memcached

We have used our transactional API to "persistify" memcached [20], a widely used, high performance, in-memory key-value store. The motivation for building a persistent version of memcached is to accelerate the restart-warmup cycle after a failure or shutdown, which can take several hours [11] in some instances because memcached is nonpersistent. A persistent memcached can significantly accelerate the warmup time. To that end, the cache's state must be correctly persistified across failure events such as power failures, kernel crashes, etc. Failure atomic updates are pivotal for this purpose.

We originally started the effort with the goal to simply port memcached's central data structure, a concurrent, growable, closed addressed hash table. However, we quickly realized that we needed to modify other parts of memcached (LRU cache management, slab allocator, lazy memory reclamation, etc.) to persist memcached's entire internal state for warm restart. As a result, this simple effort evolved into a major port of memcached to our transactional API. Transactions ended up encompassing fairly complex code paths that led to some interesting scenarios such as: tiny critical sections within transactions (supported using special capabilities such as *deferred operation execution* and *deferred lock release*, provided in our library), semantically nonpersistent data located within persistent data objects (supported using persistent generation numbers), etc.

Additionally, memcached itself is a copy-on-write based system, and uses its own slab allocator for memory management. This poses a significant problem in using our COW-based transaction runtime to perform updates since our runtime uses its own memory allocator for copy-on-writes. Furthermore, each key-value pair contains groups of fields that are protected by different locks and can be modified concurrently by multiple threads. Our COW-based persistent objects can be modified by just one thread at a time. Overall, porting memcached to our COW-based transaction runtime seemed like a significant enough restructuring of memcached that we decided to not do it. This is another example of programmability challenges for COW-based transaction runtimes. We report results of our undo and redo log based versions.

We evaluated memcached using the mutilate workload [3], fixing the number of client threads to 8. We varied the number of memcached worker threads from 1 to 8. Figure 9 shows our persistent memcached's performance with 90/10% and 50/50% get/put ratios. First, note that, for all thread counts, at 10% puts, the best performing runtime Undo/PDOM-2 has about 10–30% lower throughput than the original memcached, whereas the same runtime has about 45–60% lower throughput than the original memcached for 50% puts. This largely highlights the instrumentation and bookkeeping overheads of our transaction runtimes for transactional reads and writes.

Second, note that undo logging performs better than redo logging by 1-10% in the 10% put tests for the PDOM-2 configuration. This difference is consistently closer to 10% for the 50% put tests. We gathered stastical data on read/write access patterns of memcached's transactions. What we found was quite interesting. These transactions, particularly the put transactions, appear to have far more read-after-write instances (25-30% of all reads done by transactions) than we expected.

The get transactions are short, averaging 25 reads, 3 instances of which are read-after-writes, and 2 writes (to manage the LRU cache). The put operations break down into two distinct transactions: (i) a transaction that allocates and initializes a new key-value pair object, and (ii) a transaction to insert the new key-value pair, and remove an old pair matching the key if one exists. Both these transactions contain 80–90 read accesses, approximately 25–30% of which are read-after-writes. Furthermore, the transactions also perform 24 writes on average. A closer look at the source code indicates that almost all read-after-write instances manifest when a transaction first writes to one part of an object and then reads another part of it (e.g. increments of multiple counters in an object). It appears that memcached better matches our Array-RAW microbenchmark's profile than Array's profile; the performance also seems to nicely track that of Array-RAW (at about 25–30% read-after-write instances; see Figure 5). Similar to Array-RAW, the performance advantage undo logging has diminishes as the persist barrier latency increases.

We also observe a difference in scalability patterns. While scalability for the PDOM-2 configuration is comparable between the redo and undo logging implementations, undo logging scales worse than redo logging for PDOM-1 and PDOM-0 configurations. The explanation appears in the corresponding latency bar chart in Figure 9(c), where we clearly see the latency of put operations go up significantly for undo logging (over 200 microseconds). Our transactions end up inflating some of the critical sections of memcached (e.g. LRU cache management, slab allocator management). The higher latency of puts leads to greater lock hold intervals, which in turn hinders scalability. With 50% puts, for PDOM-0, the higher write latency (see Figure 9(d)) leads to significant slowdown in undo logging transactions at all thread counts.

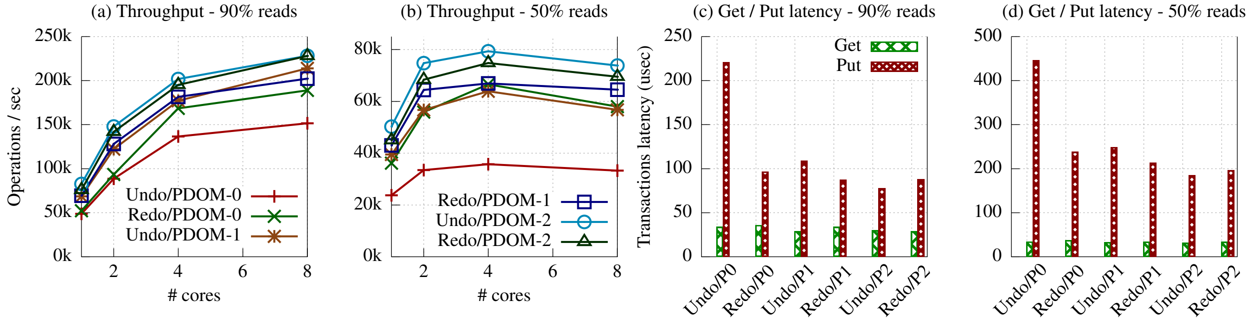

Figure 9: memcached scalability and latency results with 90/10% and 50/50% get/put ratios. Latency bar charts are for the 8 thread runs. (P0 = PDOM-0, P1 = PDOM-1, P2 = PDOM-2.)

6 Related Work

While most early work on persistent memory transaction runtimes ignores persist barrier overheads [5, 6, 8, 26], a growing number of efforts [10, 18, 19, 33] is addressing the problem in different ways.

Volos et al. [33] implement redo logging transactions in their Mnemosyne runtime. Giles et al. [10] go further with a "lazy" cleanup proposal that moves redo log application and cleanup to a background thread. They additionally add a DRAM-based aliasing mechanism to cache persistent objects in the faster DRAM. We experimented with this scheme in our framework and found that the alias table lookup induced hardware cache misses offset any gains provided by the faster DRAM cache. Lu et al. [19] optimize out Mnemosyne transactions' last persist barrier, and propose a full processor cache flush as a technique to checkpoint committed transactions' results. Kolli et al. [18] propose an undo logging based transaction runtime that introduces just four persist barriers per transaction, assuming that transactions know the data they need to modify in advance.

In contrast to these works that focus on one specific transaction runtime implementation, we perform a far more comprehensive analysis of the design space. Our analysis not only considers workload characteristics but also performance implications of persistence domains. In addition, we also report implications of cache locality.

Moraru et al. [22] present a memory allocator optimized for wear leveling, which can plug easily into our allocation log technique. Volos et al. [33] presented an allocator that uses the transaction's redo log to track persistent memory allocations. This is similar to our memory allocator. However we splice out the allocation records into a separate allocation log that lets us use it in undo and COW transaction runtimes.

Work similar to ours has emerged in the in-memory database setting [2], where the authors compare database transactions based on "in-place" updates (similar to our undo/redo logging runtimes), copy-on-write, and write-ahead logging [21] implementations. While their results align with ours – in-place updates tend to dominate over the other two approaches – the two settings are significantly different. Their runtimes are designed to optimize database processing, with hand optimized implementations of core database data structures, whereas ours are much lower level runtimes developed to track individual loads and stores to arbitrary data structures hosted in persistent memory.

Different memory persistency models have been proposed in academia over the past few years [9, 17, 25, 28]. However, there seems to be a convergence emerging on the thread-local *epoch persistency* model [9, 25] with Intel's recent deprecation of the pcommit instruction from future Intel processors. Our work applies to all these persistency models.

7 Conclusion

We presented a new taxonomy of persistence domains based on our observations of support for persistent memory in past and future systems. We also presented three transaction runtime systems based on undo logging, redo logging and copy-on-write implementations of transactional writes. Our runtimes are designed with the goal to reduce persist barriers needed in a transaction. Our allocator is also designed with the same goal and plugs into all our transaction runtimes. Our comprehensive microbenchmark-based evaluation combs through the read/write mix spectrum as well as persistence domain choices showing that there exists a complex interplay between several factors that contribute to performance of transaction runtimes – workload read/write access patterns, persistence domains, cache locality, and transaction runtime bookkeeping overheads. Our "real world" workload evaluations nicely conform to our microbenchmark analysis, and provide insights into influence of additional complexities including networking overheads (our K-V store), and synchronization in multi-threaded applications (memcached). While COW transactions appear to be a nonstarter, the choice between redo and undo logging based runtimes is non-trivial, and needs to be informed

by the various parameters pertinent to the workload and the enclosing system's support for persistence.

References

[1] 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[2] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 707–722, 2015.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[5] B. Bridge. Nvm-direct library. *https://github.com/oracle/nvm-direct*, 2015.

[6] D. R. Chakrabarti, H. Boehm, and K. Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 433–452, 2014.

[7] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 133–146, 2009.

[10] E. Giles, K. Doshi, and P. J. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–14, 2015.

[11] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener. Fast database restarts at facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 541–549, 2014.

[12] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting*, pages 459–462, 2005.

[13] Y. Huai. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[14] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf, 2015.

[15] NVDIMM Block Window Driver Writer's Guide. *http://pmem.io/documents/NVDIMM_DriverWritersGuide-July-2016.pdf*, 2016.

[16] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–442, 2016.

[17] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, 2015.

[18] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-Performance Transactions for Persistent Memories. In *21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[19] Y. Lu, J. Shu, and L. Sun. Blurred Persistence: Efficient Transactions in Persistent Memory. *ACM Transactions on Storage*, 12(1):3:1–3:29, 2016.

[20] Memcached – a distributed memory object caching system. *https://memcached.org/*.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[22] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pages 1:1–1:17, 2013.

[23] D. Narayanan and O. Hodson. Whole System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[24] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. M. III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015*, pages 689–694, 2015.

[25] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 265–276, 2014.

[26] pmem.io: Persistent Memory Programming. *http://pmem.io/*, 2015.

[27] D. S. Rao, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, page 15, 2014.

[28] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685, 2015.

[29] SQLite. *https://www.sqlite.org/*.

[30] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing Memristor found. *Nature*, 453:80–83, 2008.

[31] The SNIA NVM Programming Technical Working Group. NVM Programming Model (Version 1.0.0 Revision 10), Working Draft. http://snia.org/sites/default/files/NVMProgrammingModel_v1r10DRA 2013.

[32] TPC-C, *https://www.tpc.org/tpcc*.

[33] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.

[34] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–10, 2015.