Intel PMDK Transactions: Specification, Validation and Concurrency*

Azalea Raad[1], Ori Lahav[2], John Wickerson[1], Piotr Balcer[3], and Brijesh Dongol[4]

[1] Imperial College London, London, UK
[2] Tel Aviv University, Tel Aviv, Israel
[3] Intel, Gdansk, Poland
[4] University of Surrey, Guildford, UK

Abstract. Software Transactional Memory (STM) is an extensively studied paradigm that provides an easy-to-use mechanism for thread safety and concurrency control. With the recent advent of byte-addressable persistent memory, a natural question to ask is whether STM systems can be adapted to support *failure atomicity*. In this paper, we answer this question by showing how STM can be easily integrated with Intel's Persistent Memory Development Kit (PMDK) transactional library (which we refer to as TXPMDK) to obtain STM systems that are both concurrent and persistent. We demonstrate this approach using known STM systems, TML and NOREC, which when combined with TXPMDK result in persistent STM systems, referred to as PMDK-TML and PMDK-NOREC, respectively. However, it turns out that existing correctness criteria are insufficient for specifying the behaviour of TXPMDK and our concurrent extensions. We therefore develop a new correctness criterion, *dynamic durable opacity*, that extends the previously defined notion of durable opacity with dynamic memory allocation. We provide a model of TXPMDK, then show that this model satisfies dynamic durable opacity. Moreover, dynamic durable opacity supports concurrent transactions, thus we also use it to show correctness of both PMDK-TML and PMDK-NOREC.

1 Introduction

Persistent memory technologies (aka non-volatile memory, NVM) such as Memory-Semantic SSD [53] and XL-FLASH [13], combine the durability of hard drives with the fast and fine-grained accesses of DRAMs, with the potential to radically change how we build fault-tolerant systems. However, NVM also raises fundamental questions about semantics and the applicability of standard programming models.

* Raad is funded by a UKRI fellowship MR/V024299/1, EPSRC grant EP/X037029/1 and VeTSS. Lahav is supported by the Israel Science Foundation (grant 814/22) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811). Wickerson is funded by EPSRC grant EP/R006865/1. Dongol is funded by EPSRC grants EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, EP/R025134/2 and VeTSS.

```
1  struct loc {
2    pmem::obj::p<int> value;
3    pmem::obj::persistent_ptr<loc> next; };
4
5  struct root { pmem::obj::persistent_ptr<loc> head = nullptr; };
6
7  void post_crash(...) {
8   auto pop = pmem::obj::pool<root>::open("file",...);
9   auto root = pop.root();
10  pmem::obj::transaction::run(pop, [&]{
11     auto xvalue = root->head->value;
12  }); }
13
14 int main(...) {
15   auto pop = pmem::obj::pool<root>::open("file",...);
16   auto root = pop.root();
17   pmem::obj::transaction::run(pop, [&]{
18     auto x = pmem::obj::make_persistent<loc>();
19     x->value = 42;
20     x->next = nullptr;
21     root->head = x;
22   }); }
```

Fig. 1: C++ snippet for allocating in persistent memory using TXPMDK [54]

Among the most widely used collections of libraries for persistent programming is Intel's Persistent Memory Development Kit (PMDK), which was first released in 2015 [30]. One important component of PMDK is its transactional library, which we refer to as TXPMDK, and which supports generic *failure-atomic* programming. A programmer can use TXPMDK to protect against full system crashes by starting a transaction, performing transactional reads and writes, then committing the transaction. If a crash occurs during a transaction, but before the commit, then upon recovery, any writes performed by the transaction will be rolled back. If a crash occurs during the commit, the transaction will either be rolled back or be committed successfully, depending on how much of the commit operation has been executed. If a crash occurs after committing, the effect of the transaction is guaranteed to persist.

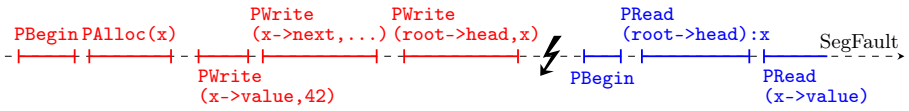
Most software transactional memory (STM) algorithms leave memory allocation implicit, since they are generally safe under standard allocation techniques (e.g. `malloc`). Memory that is allocated as part of a transaction can be deallocated if the transaction is aborted. However, in the context of persistency, memory allocation is more subtle since transactions may be interrupted by a crash.

For example, consider the program in Fig. 1. Persistent memory is allocated, accessed and maintained via *memory pools* [54] (files that are memory mapped into the process address space) of a certain type (e.g. of type `loc` in Fig. 1). Due to address space layout randomization (ASLR) in most operating systems, the location of the pool can differ between executions and across crashes. As such, every pool has a root object from which all other objects in the pool can be

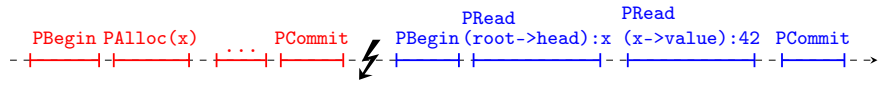
found. That is, to avoid memory leaks, all objects in the pool must be reachable from the root. An application locates the root object using a *pool object pointer* (POP) that is to be created with every program invocation (e.g. line 15). After locating the pool root (line 16), we use a TXPMDK transaction (lines 17–22) to allocate a persistent `loc` object `x` (line 18) with value 42 (line 19) and add it to the pool (line 21).

Consider the scenario where the execution of this transaction crashes. After recovery from the crash, we then execute `post_crash` (line 7). As before, we open the pool (line 8) and locate its root (line 9). We then use a TXPMDK transaction to read from the `loc` object allocated and added at the pool head prior to the crash (line 11). There are then three cases to consider: the crash may have occurred **(1)** before the transaction started the commit process, **(2)** after the transaction successfully committed, or **(3)** while the transaction was in the process of committing.

In case **(1)**, the execution of the two transactions can be depicted as follows, where the `PBegin` events capture commencing the transactions (lines 17 and 10), `PAlloc(x)` denotes the persistent allocation of `x` (line 18); `PWrite(x->value,42)` captures writing to `x` (line 19); and `PRead(root->head):x` denotes reading from `x->value` and returning the value `x` (first part of line 11). As the first transaction never reached the commit stage, its effects (i.e. allocating `x` and writing to it) should be invisible (i.e. rolled back), and thus the read of the second transaction effectively reads from unallocated memory, leading to an error such as a segmentation fault.



In case **(2)**, the execution of the transactions is as follows, where the `PCommit` events capture the end (successful commit) of the transactions (lines 22 and 12), the effects of the first transaction fully persist upon successful commit, and thus the read in the second transaction does not fault.



Finally, in case **(3)**, either of the two behaviours depicted above is possible (i.e. the second transaction may either cause a segmentation fault or read from `x`).

Efficient and correct memory allocation in a persistent memory setting is challenging ([54, Chapter 16] and [55]). In addition to the ASLR issue mentioned above, the allocator must guarantee failure atomicity of heap operations on several internal data structures managed by PMDK. Therefore, PMDK provides its own allocator that is designed specifically to work with TXPMDK.

We identify two key drawbacks of TXPMDK as follows. In this paper, we take steps towards addressing both of these drawbacks.

A) Lack of concurrency support. Unlike existing STM systems in the persistent setting [39,32] that provide *both failure atomicity* (ensuring that a transaction

either commits fully or not at all in case of a crash) *and isolation* (as defined by ACID properties, ensuring that the effects of incomplete transactions are invisible to concurrently executing transactions), TXPMDK only provides failure atomicity and does not offer isolation in concurrent settings. In particular, naïvely implemented applications with racy PMDK transactions lead to memory inconsistencies. This is against the spirit of STM: the primary function of STM systems is providing a concurrency control mechanism that ensures isolation. The current TXPMDK implementation provides two solutions: threads either execute *concurrent* transactions over *disjoint* parts of the memory [54, Chapter 7], or use user-defined *fine-grained locks* within a transaction to ensure memory isolation [54, Chapter 14]. However, both solutions are sub-optimal: the former enforces serial execution when transactions operate over the same part of the memory, and the latter expects too much of the user.

B) Lack of a suitable correctness criterion. There is no formal specification describing the desired behaviour of TXPMDK, and hence no rigorous description or correctness proof of its implementation. This undermines the utility of TXPMDK in safety-critical settings and makes it impossible to develop formally verified applications that use TXPMDK. Indeed, there is currently no correctness criterion for STM systems that provide dynamic memory allocation (a large category that includes all realistic implementations).

1.1 Concurrency for TXPMDK

Integrating concurrency with PMDK transactions is an important end goal for PMDK developers. The existing approach requires integration of locks with TXPMDK, which introduces overhead for programmers. Our paper shows that STM and PMDK can be easily combined, improving programmability. Many other works have aimed to develop failure-atomic and concurrent transactions (e.g. OneFile [52] and Romulus [16]), but none use off-the-shelf commercially available libraries. Moreover, these other works have not addressed correctness with the level of rigour that our paper does. In other work, popular key-value stores Memcached and Redis have been ported to use PMDK [36,37]; our work paves the way for concurrent version of these applications to be developed. Another example is the work of Chajed et al [11], who provide a simulation-based technique for *verifying* refinement of durable filesystems, where concurrency is handled by durable transactions.

We tackle the first drawback (A) mentioned above by developing, specifying, and validating two thread-safe versions of TXPMDK.

Contribution A: Making TXPMDK thread-safe. We combine TXPMDK with two off-the-shelf (thread-safe) STM systems, TML [17] and NOREC [18], to obtain two new implementations, PMDK-TML and PMDK-NOREC, that support *concurrent* failure-atomic transactions with dynamic memory allocation. In particular, we reuse the existing concurrency control mechanisms provided by these STM systems to ensure atomicity of write-backs, thus obtaining memory isolation even in a multi-threaded setting. We show that it is possible to integrate

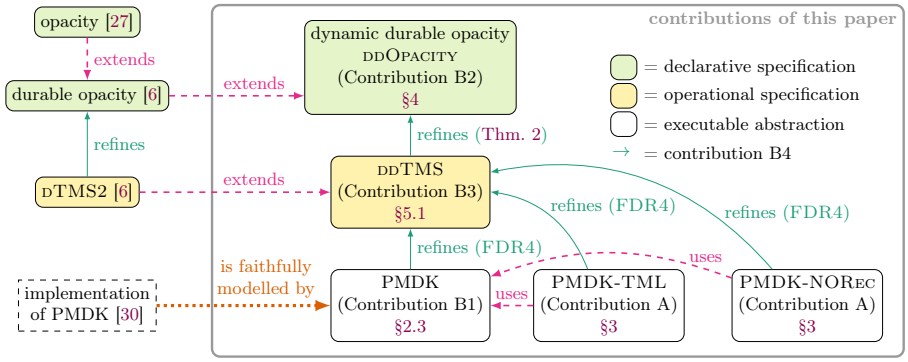


Fig. 2: The contributions of this paper and their relationships to prior work

these mechanisms with txPMDK to additionally achieve failure atomicity. Our approach is modular, with a clear separation of concerns between the isolation required due to concurrency and the atomicity required due to the possibility of system crashes. This shows that concurrency and failure atomicity are two orthogonal concerns, highlighting a pathway towards a mix-and-match approach to combining (concurrent) STM and failure-atomic transactions. Finally, in order to provide the same interface as PMDK, we extend both TML and NOrec with an explicit operation for memory allocation.

1.2 Specification and Validation

To tackle drawback (B) above, we make four contributions. Together, they provide the first formal (and rigorous) specification of txPMDK and validation of its implementation.

Contribution B1: A model of txPMDK. We provide a formal specification of txPMDK as an abstract transition system. Our formal specification models almost all key components of txPMDK (including its redo and undo logs, as well as the interaction of these components with system crashes), with the exception of memory deallocation within txPMDK transactions.

Contribution B2: A correctness criterion for transactions with dynamic allocation. Although the literature includes several correctness criterion for transactional memory (TM), none can adequately capture txPMDK in that they do not account for dynamic memory allocation. We develop a new correctness condition, *dynamic durable opacity* (denoted ddOpacity), by extending durable opacity [6] to account for dynamic allocation. ddOpacity supports not only sequential transactions such as txPMDK, but also concurrent ones. To demonstrate the suitability of ddOpacity for concurrent and persistent (durable) transactions, later we validate our two concurrent txPMDK implementations (PMDK-NOrec and PMDK-TML) against ddOpacity.

Contribution B3: An operational characterisation of our correctness criterion. Our aim is to show that txPMDK conforms to ddOpacity, or

more precisely, that our model of TXPMDK refines our model of DDOPACITY. To demonstrate this, we use a new intermediate model called DDTMS. While DDOPACITY is defined declaratively, DDTMS is defined operationally, which makes it conceptually closer to our model of the TXPMDK implementation. We prove that DDTMS is a sound model of DDOPACITY (i.e. every trace of DDTMS satisfies DDOPACITY).

Contribution B4: Validation of TXPMDK, PMDK-TML and PMDK-NOREC in FDR4. We mechanise our implementations (TXPMDK, PMDK-TML and PMDK-NOREC) and specification (DDTMS) using the CSP modelling language. We use the FDR4 model checker [26] to show the implementations are refinements of DDTMS over both the persistent SC (PSC) [31] and persistent TSO (Px86_{sim}) [50] memory models. For Px86_{sim}, we use an equivalent formulation called PTSO_{syn} developed by Khyzha and Lahav [31]. The proof itself is fully automatic, requiring no user input outside of the encodings of the models themselves. Additionally, we develop a sequential lower bound (DDTMS-Seq), derived from DDTMS, and show that this lower bound refines TXPMDK (and hence that TXPMDK is not vacuously strong). Our approach is based on an earlier technique for proving durable opacity [23], but incorporates much more sophisticated examples and memory models.

Outline. Fig. 2 gives an overview of the different components that we have developed in this paper and their relationships to each other and to prior work. We structure our paper by presenting the components of Fig. 2 roughly from the bottom up. In §2, we present the abstract TXPMDK model, and in §3 we describe its integration with STM to provide concurrency support via PMDK-TML and PMDK-NOREC. In §4 we present DDOPACITY, in §5 we present DDTMS, and in §6 we describe our FDR4 encodings and bounded proofs of refinement.

Additional Material. We provide our FDR4 development as supplementary material [47]. The proofs of all theorems are given in an extended version [46].

2 Intel PMDK transactions

We describe the abstract interface TXPMDK provides to clients (§2.1), our assumptions about the memory model over which TXPMDK is run (§2.2) and the operations of TXPMDK (§2.3). We present our PMDK abstraction in §2.3.

2.1 PMDK Interface

PMDK provides an extensive suite of libraries for simplifying persistent programming. The PMDK transactional library (TXPMDK) has been designed to support failure-atomicity by providing operations for tracking memory locations that are to be made persistent, as well allocating and accessing (reading and writing) persistent memory within an atomic block.

In Fig. 3 we present an example client code that uses TXPMDK. The code (due to [54, p. 131]) implements the `push` operation for a persistent linked-list queue.

```
1  struct queue_node {
2      pmem::obj::p<int> value;
3      pmem::obj::persistent_ptr<queue_node> next; };
4
5  struct queue { private:
6      pmem::obj::persistent_ptr<queue_node> head = nullptr;
7      pmem::obj::persistent_ptr<queue_node> tail = nullptr; };
8
9  void push(pmem::obj::pool_base &pmem_op, int value) {
10    pmem::obj::transaction::run(pmem_op, [&]{
11       auto node = pmem::obj::make_persistent<queue_node>();
12       node->value = value;
13       node->next = nullptr;
14       if (head == nullptr) {
15          head = tail = node;
16       } else {
17          tail->next = node;
18          tail = node; }
19     }); }
```

Fig. 3: C++ persistent `push` operation using TXPMDK ([54, p. 131])

The implementation wraps a typical (non-persistent) `push` operation within a transaction using a C++ lambda `[&]` expression (line 10). The transaction is invoked using `transaction::run`, which operates over the memory pool `pmem_op`. The node structure (lines 2 and 3), the queue structure (lines 6 and 7), and any new node declaration (line 11) are to be tracked by a PMDK transaction. Additionally, the `push` operation takes as input the *persistent memory object pool*, `pmem_op`, which is a memory pool on which the transaction is to be executed. This argument is needed because the application memory may map files from different file systems. On line 7 we use `make_persistent` to perform a transactional allocation on persistent memory that is linked to the object pool `pmem_op` (see [54] for details). The remainder of the operation (lines 12–18) corresponds to an implementation of a standard `push` operation with (transactional) reads and writes on the indicated locations. At line 19, the C++ lambda and the transaction is closed, signalling that the transaction should be committed.

If the system crashes while `push` is executing, but before line 19 is executed, then upon recovery, the entire `push` operation will be rolled back so that the effect of the incomplete operation is not observed, and the queue remains a valid linked list. After line 19, the corresponding transaction executes a commit operation. If the system crashes during commit, depending on how much of the commit operation has been executed, the `push` operation will either be rolled back, or committed successfully. Note that roll-back in all cases ensures that the allocation at line 11 is undone.

2.2 Memory Models

We consider the execution of our implementations over two different memory models: PSC and PTSO_{syn} [31]. Both models include a `flush x` instruction

to persist the contents of the given location `x` to memory. PTSO_{syn} aims for fidelity to the Intel x86 architecture. In a race-free setting (as is the case for single-threaded TXPMDK transactions) it is sound to use the simpler PSC model, though we conduct all of our experiments in both models.

PSC is a simple model that considers persistency effects and their interaction with sequential consistency. Writes are propagated directly to per-location persistence buffers, and are subsequently flushed to non-volatile memory, either due to a system action, or the execution of a `flush` instruction. A read from `x` first attempts to fetch its value from the persistence buffer and if this fails, fetches its value from non-volatile memory.

Under Intel-x86, the memory models are further complicated by the interaction between *total store ordering* (TSO) effects [40] and persistency. Due to the abstract nature of our models (see Fig. 4) it is sufficient for us to focus on the simpler Px86_{sim} model [50] since we do not use any of the advanced features [48,49,50]. We introduce a further simplification via PTSO_{syn} that is *observationally equivalent* to Px86_{sim} [31]. Unlike Px86_{sim}, which uses a single (global) persistence buffer, PTSO_{syn} uses per-location buffers simplifying the resulting FDR4 models (§6).

In PTSO_{syn}, writes are propagated from the store buffer in FIFO order to a per-location FIFO persistency buffer. Writes in the persistency buffer are later persisted to the non-volatile memory. A read from location `x` first attempts to fetch the latest write to `x` from the store buffer. If this fails (i.e. no writes to `x` exists in the store buffer), it attempts to fetch the latest write from the persistence buffer of `x`, and if this fails, it fetches the value of `x` from non-volatile memory.

2.3 PMDK Implementation

We present the pseudo-code of our TXPMDK abstraction in Fig. 4. We model all features of TXPMDK (including its redo and and undo logs as well as its recovery mechanism in case of a crash) except memory deallocation within a TXPMDK transaction. We use `mem` to model the memory, mapping each location (in `loc`) to a value-metadata pair. We model a value (in `val`) as an integers, and `metadata` as a boolean indicating whether the location is allocated. As we see below, the list of free (unallocated) locations, `freeList`, is calculated during recovery using `metadata`.

Each PMDK transaction maintains redo logs and an undo log. The redo logs record the locations allocated by the transaction so that if a crash occurs while committing, the allocated locations can be reallocated, allowing the transaction to commit upon recovery. Specifically, TXPMDK uses two *distinct* redo logs: `tRedo` and `pRedo`. Both are associated with fields `undoValid` (which is unset when the log is invalidated), `checksum` (used to indicate whether the log is valid), and `allocs` (which contains the set of locations allocated by the transaction). Note that TXPMDK explicitly sets and unsets `undoValid`, whereas `checksum` is calculated (e.g. at line 36) and may be invalidated by crashes corrupting a partially completed write. The undo log records the original (overwritten) value of each location written to by the transaction, and is consulted if the transaction is to be rolled back. We model it as a map from locations to values (of type `int`).

```
// Each location is persistent; there is no explicitly volatile memory.
mem : loc -> {
  val : int; // the contents of this location
  metadata : bool; } // false = not allocated, true = allocated
freeList : loc list // transient list of free locations

// Redo logs -- tRedo is transient; pRedo is persistent.
tRedo_t, pRedo_t : {undoValid:bool;  checksum:int;  allocs:loc set;}
undo_t : loc -> int  // undo log recording the original val of each loc
undoValid : bool  // undoValid global flag, initially true
```

```
PBegin_t ≜
   tRedo_t := (true, -1, {})
   pRedo_t := (true, -1, {})
   undo_t := {}
   undoValid_t := true

PAlloc_t ≜
   x_t := freeList.take
   tRedo_t.allocs :=
     tRedo_t.allocs ∪ {x_t}
   return x_t

PRead_t(x) ≜
   return mem[x].val

PWrite_t(x,v) ≜
   if x ∉ dom(undo_t) then
      w_t := mem[x].val
      undo_t := undo_t ∪ {x ↦ w_t}
      flush undo_t
   mem[x].val := v

PCommit_t ≜
   persist_writes_t
   tRedo_t.undoValid := false
   tRedo_t.checksum :=
         calc_checksum(tRedo_t)
   pRedo_t := tRedo_t
   flush pRedo_t
   apply_pRedo_t
   pRedo_t.checksum := -1
   flush pRedo_t.checksum
```

```
apply_pRedo_t ≜
   foreach x ∈ pRedo_t.allocs:
      mem[x].metadata := true
      flush mem[x].metadata
   if ¬pRedo_t.undoValid  then
      undoValid_t := false
      flush undoValid_t

persist_writes_t ≜
   foreach x ∈ dom(undo_t): flush x

roll_back_t ≜
   foreach (x ↦ v) ∈ undo_t:
      mem[x].val := v
   persist_writes_t

PAbort_t ≜
   roll_back_t
   undoValid_t := false
   flush undoValid_t
   foreach x ∈ tRedo_t.allocs:
      freeList.add(x)

PRecovery_t ≜
   if calc_checksum(pRedo_t)
      = pRedo_t.checksum
   then apply_pRedo_t
   if undoValid_t then
      roll_back_t
   foreach x ∈ dom(mem):
      if ¬mem[x].metadata then
         freeList.add(x)
```

Fig. 4: PMDK global variables and pseudo-code

A separate variable `undoValid` (distinct from `undoValid` in `tRedo` and `pRedo`) is used to determine whether this undo log is valid.

Each component in Fig. 4 have both a volatile and persistent copy, although some components, e.g. `tRedo` and `freeList`, are *transient*, i.e. their persistent

versions are never used. Likewise, the persistent redo log, `pRedo`, is only used in a persistent fashion and its volatile copy is never used.

We now describe the operations in Fig. 4. We assume the operations are executed by a transaction with id t. This id is not useful in the sequential setting in which TXPMDK is used; however, in our concurrent extension (§3) the transaction id is critical.

PBegin. The begin operation simply sets all local variables to their initial values.

PAlloc. Allocation chooses and removes a free location, say `x`, from the free list, adds `x` to the transient redo log (line 20) and returns `x`. Removing `x` from `freeList` ensures it is not allocated twice, while the transient redo log is used together with the persistent redo log to ensure allocated locations are properly reallocated upon a system crash.

When the transaction commits, the transient redo log is copied to the persistent one (line 37), and the effect of the persistent log is applied at line 39 via `apply_pRedo`. (Note that `apply_pRedo` is also called by `PRecovery` on line 68.) The behaviour of this call depends on how much of the in-flight transaction was executed before the crash leading to the recovery. If a crash occurred after the transaction executed (line 37) and the corresponding write persisted (either due to a system flush or the execution of line 38), then executing `apply_pRedo` via `PRecovery` has the same effect as the executing line 39, i.e. the effect of the redo log will be applied. This (persistently) sets the `metadata` field of each location in the redo log to indicate that it is allocated (lines 43–45), and then invalidates the undo log (lines 46–48) so that the transaction is not rolled back.

PRead. A read from `x` simply returns its in-memory value (line 24). Note that location `x` may not be allocated; TXPMDK delegates the responsibility of checking whether it is allocated to the client.

PWrite. A write to `x` first checks (line 27) if the current transaction has already written to `x` (via a previously executed `PWrite`). If not, it logs the current value by reading the in-memory value of `x` (line 28) and records it in the undo log (line 29). The updated undo log is then made persistent (line 30). Once the current value of `x` is backed up in the undo log (either by the current write or by the previous write to `x`), the value of `x` in memory is updated to the new value `v` (line 31). As with the read, location `x` may not have been allocated; TXPMDK delegates this check to the client.

PCommit. The main idea behind the commit operation is to ensure all writes are persisted, and that the persistent redo and undo logs are cleared in the correct order, as follows. **(1)** On line 34 all writes written by the transaction are persisted. **(2)** Next, the transient redo log is invalidated (line 35) and the checksum for the log is calculated (line 36). This updated transient log is then set to be the persistent redo log (line 37), which is then made persistent (line 38). Note that after executing line 38, we can be assured that the transaction has committed; if a crash occurs after this point, the recovery will *redo* and persist the allocation and the undo log will be cleared. **(3)** The operation then calls `apply_pRedo` at line 39, which makes the allocation persistent and clears the undo log. **(4)** Finally,

at line 40, the `pRedo` checksum is invalidated since `apply_pRedo` has already been executed. If a crash occurs after line 40 has been executed, then the recovery checks at line 67 and line 69 will fail, i.e. recovery will calculate the free list.

PAbort. A PMDK transaction is aborted by a `PRead`/`PWrite` that attempts to access (read/write) an unallocated location. When a transaction is aborted, all of its observable effects must be rolled back. First, the memory effects are rolled back (line 59), then the undo log is invalidated (line 60) and made persistent (line 61), preventing undo from being replayed in case a crash occurs. Finally, all of the locations allocated by the executing transaction are freed (lines 62–63). Note that if a crash occurs during an abort, the effect of the abort will be replayed. `PRecovery` reconstructs the free list at lines 71–73, which effectively replays the loop at lines 62–63 of `PAbort`. Additionally, if a crash occurs before the write at line 60 has persisted, then the effect of undoing the operation will be explicitly replayed by the roll-back executed by `PRecovery` since `undoValid` holds. If the crash occurs after the write at line 60 has persisted, then no roll-back is necessary.

PRecovery. The recovery operation is executed immediately after a crash, and before any other operation is executed. The recovery proceeds in three phases: **(1)** The checksum of the persistent redo log is recalculated (line 67) and if it matches the stored checksum (`pRedo.checksum`) the `apply_pRedo` operation is executed. As discussed, `apply_pRedo` sets and persists the metadata of each location in the redo log, and then invalidates the undo log. **(2)** The transaction is rolled back if `apply_pRedo` in step 1 fails; otherwise, no roll-back is performed. **(3)** The free list is reconstructed by inserting each location whose metadata is set to false into `freeList` (lines 71–73).

Correctness and Thread Safety. As discussed in §2.1, TXPMDK is designed to be failure-atomic. This means that correctness criteria such as opacity [27,2] and TMS1/TMS2 [20] (restricted to sequential transactions) are inadequate since they do not accommodate crashes and recovery. This points to conditions such as durable opacity [6], which extends opacity with a persistency model. However, durable opacity (restricted to sequential transactions) is also insufficient since it does not define correctness of allocations and assumes totally ordered histories. In §4 we develop a generalisation of durable opacity, called *dynamic durable opacity* (DDOPACITY) that addresses both of these issues. As with durable opacity, DDOPACITY defines correctness for concurrent transactions. We develop concurrent extensions of PMDK transactions in §3, which we show to be correct against (i.e. refinements of) DDOPACITY.

As discussed, PMDK transactions are not thread-safe; e.g. concurrent calls to `PRead` and `PWrite` on the same location create a data race causing `PRead` to return an undefined value (see the example in §1). We discuss techniques for mitigating against such races in §3. Nevertheless, some PMDK transactional operations are naturally thread-safe. In particular, `PAlloc` is designed to be thread-safe via an built-in *arena* mechanism: a memory pool split into disjoint arenas with each thread allocating from its own arena. Moreover, each thread uses locks for each arena to publish allocated memory to the shared pool [55].

Init. glb = 0

```
1  TxBegin_t ≜
2     do loc_t := glb
3     until even(loc_t)
4     PBegin_t
5
6  TxAlloc_t ≜
7     return PAlloc_t
8
9  TxWrite_t(x, v) ≜
10    if even(loc_t) then
11       if ¬cas(glb, loc_t, loc_t+1)
12       then PAbort_t; return abort
13       else loc_t++
14    PWrite_t(x,v)
```

```
15 TxRead_t(x) ≜
16   v_t := PRead_t(x)
17   if even(loc_t) then
18      if glb = loc_t then
19         return v_t
20      else PAbort_t; return abort
21   else return v_t
22
23 TxCommit_t ≜
24    PCommit_t
25    if odd(loc_t) then
26       glb := loc_t+1
27
28 Recovery ≜
29    foreach t ∈ TXID:
30       PRecovery_t
31    glb := 0
```

Fig. 5: Pseudo-code for PMDK-TML with our additions made w.r.t. TML highlighted red

3 Making PMDK Transactions Concurrent

We develop two algorithms that combine two existing STM systems with PMDK. The first algorithm (§3) is based on TML [17], which uses *pessimistic concurrency control* via an eager write-back scheme. Writing transactions effectively take a lock and perform the writes in place. The second algorithm (§3) is based on NOrec [18], which utilises *optimistic concurrency control* via a lazy write-back scheme. In particular, transactional writes are *collected* in a local write set and *written back* when the transaction commits.

It turns out that PMDK can be incorporated within both algorithms straightforwardly. This is a strength of our approach and points towards a generic technique for extending existing STM systems with failure atomicity. Given the challenges of persistent allocation, we reuse PMDK's allocation mechanisms to provide an explicit allocation mechanism in both our extensions [54].

PMDK-TML. We present the pseudo-code for PMDK-TML (combining TML and txPMDK) in Fig. 5, where we highlight the calls to txPMDK operations. These calls are the only changes we have made to the TML algorithm. TML is based on a single global counter, `glb`, whose value is read and stored within a local variable $\texttt{loc}_t$ when transaction t begins (`TxBegin`). There is an in-flight *writing* transaction iff `glb` is odd. TML is designed for read-heavy workloads, and thus allows multiple concurrent read-only transactions. A writing transaction causes all other concurrent transactions to abort.

PMDK-TML proposes a modular combination of PMDK with the TML algorithm by nesting a PMDK transaction inside a TML transaction; i.e. each transaction additionally starts a PMDK transaction. All reads and writes to

memory are replaced by TXPMDK read and write operations. Moreover, when a transaction aborts or commits, the operation calls a TXPMDK abort or commit, respectively. Finally, PMDK-TML includes allocation and recovery operations, which call TXPMDK allocation and recovery, respectively. The recovery operation additionally sets `glb` to 0.

A read-only transaction t may call $\texttt{PRead}_t$ at line 16 when another transaction t' is executing $\texttt{PWrite}_{t'}$ at line 14 on the same location. Since TXPMDK does not guarantee thread safety for these calls, the value returned by $\texttt{PRead}_t$ should not be passed back to the client. This is indeed what occurs. First, note that if transaction t is read-only, then $\texttt{loc}_t$ is even. Moreover, a read-only transaction only returns the value returned by $\texttt{PRead}_t$ (line 19) if no other transaction has acquired the lock since t executed $\texttt{TxBegin}_t$. In the scenario described above, t' must have incremented `glb` by successfully executing the CAS at line 11 as part of the first write operation executed by t', changing the value of `glb`. This means that t would abort since the test at line 18 would fail.

PMDK-NOREC. We present PMDK-NOREC (combining NOREC and PMDK) in Fig. 6, where we highlight the calls to TXPMDK. These calls are the only changes we have made to the NOREC algorithm. As with TML, NOREC is based on a single global counter, `glb`, whose value is read and stored within a transaction-local variable `loc` when a transaction begins (`TxBegin`). There is an in-flight writing transaction iff `glb` is odd. Unlike TML, NOREC performs lazy write-back, and hence utilises transaction-local read and write sets. A transaction only performs the write-back at commit time once it "acquires" the `glb` lock. Prior to write-back and read response, it ensures that the read sets are consistent using a per-location validate operation. We eschew details of the NOREC synchronisation mechanisms and refer the interested reader to the original paper [18].

The transformation from TXPMDK to PMDK-NOREC is similar to PMDK-TML. We ensure that a PMDK transaction is started when a PMDK-NOREC transaction begins, and that this PMDK transaction is either aborted or committed before the PMDK-NOREC transaction completes. We introduce `TxAlloc` and `Recovery` operations that are identical to PMDK-TML, and replace all calls to read and write from memory by `PRead` and `PWrite` operations, respectively.

As with PMDK-TML, a `PRead` executed by a transaction (at line 12, line 15 or line 31) may race with a `PWrite` (at line 43) executed by another transaction. However, since `PWrite` operations are only executed after a transaction takes the `glb` lock (at line 40), any transaction with a racy `PRead` is revalidated. If validation fails, the associated transaction is aborted.

4 A Declarative Correctness Criteria

We present a declarative correctness criteria for TM implementations. Unlike prior definitions such as (durable) opacity, TMS1/2 etc. that are defined in terms of histories of invocations and responses, we define dynamic durable opacity (DDOPACITY) in terms of execution graphs, as is standard model for weak memory setting. Our models are inspired by prior work on declarative specifications for

Init: `glb = 0`

```
1  TxBegin_t ≜
2     do loc_t := glb
3     until even(loc_t)
4     PBegin_t
5
6  TxAlloc_t ≜
7     return PAlloc_t
8
9  TxRead_t(x) ≜
10    if x ∈ dom(wrSet_t) then
11       return wrSet_t(x)
12    v_t := PRead_t(x)
13    while loc_t ≠ glb
14       loc_t := Validate
15       v_t := PRead_t(x)
16    rdSet_t := rdSet_t ∪ {x ↦ v_t}
17    return v_t
18
19 Recovery ≜
20    foreach t ∈ TXID:
21       PRecovery_t
22    glb := 0
```

```
23 TxWrite_t(x,v) ≜
24    wrSet_t := wrSet_t ∪ {x ↦ v}
25
26 Validate_t ≜
27    while true
28       time_t := glb
29       if odd(time_t) then goto 28
30       foreach x ↦ v ∈ rdSet_t:
31          if PRead_t(x) ≠ v
32          then PAbort_t; return abort
33       if time_t = glb
34       then return time_t
35
36 TxCommit_t ≜
37    if wrSet_t.isEmpty
38       then PCommit_t
39            return
40    while ¬cas(glb, loc_t, loc_t + 1)
41       loc_t := Validate_t
42    foreach x ↦ v ∈ wrSet_t:
43       PWrite_t(x, v)
44    PCommit_t
45    glb := loc_t + 2
46    return
```

Fig. 6: Pseudo-code for PMDK-NOREC, with our additions made w.r.t. NOREC highlighted red

transactional memory, which focussed on specifying relaxed transactions [22,14]. However, these prior works do not describe crashes or allocation.

Executions and Events. The traces of memory accesses generated by a program are commonly represented as a set of *executions*, where each execution G is a graph comprising: 1. a set of events (graph nodes); and 2. a number of relations on events (graph edges). Each event e corresponds to the execution of either a transactional event (e.g. marking the beginning of a transaction) or a memory access (read/write) within a transaction.

Definition 1 (Events). *An* event *is a tuple* $a = \langle n, \tau, t, l \rangle$*, where* $n \in \mathbb{N}$ *is an event identifier,* $\tau \in TID$ *is a* thread identifier*,* $t \in TXID$ *is a* transaction identifier *and* $l \in LAB$ *is an event* label*.*

A label may be `B` *to mark the beginning of a transaction;* `A` *to denote a transactional abort;* $(\mathtt{M}, x, 0)$ *to denote a memory allocation yielding* x *initialised with value* 0*;* $(\mathtt{R}, x, v)$ *to denote reading value* v *from location* x*;* $(\mathtt{W}, x, v)$ *to denote writing* v *to* x*;* `C` *to mark the beginning of the transactional commit process; or* `S` *to denote a successful commit.*

The functions `tid`, `tx` and `lab` respectively project the thread identifier, transaction identifier and the label of an event. The functions `loc`, $\mathtt{val_r}$ and $\mathtt{val_w}$

respectively project the location, the read value and the written value of a label, where applicable, and are lifted to events by defining e.g. $\mathtt{loc}(a) = \mathtt{loc}(\mathtt{lab}(a))$.

Notation. Given a relation r and a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$; $\mathsf{irreflexive}(r)$ for $\nexists a.\,(a,a) \in r$; and $\mathsf{acyclic}(\mathsf{r})$ for $\mathsf{irreflexive}(\mathsf{r}^+)$. We write $\mathsf{r}_1;\mathsf{r}_2$ for the relational composition of r_1 and r_2, i.e. $\{(a,b) \mid \exists c.\,(a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$. When A is a set of events, we write A_x for $\{a \in A \mid \mathtt{loc}(a) = x\}$, and r_x for $\mathsf{r}|_{A_x}$. Analogously, we write A_t for $\{a \in A \mid \mathtt{tx}(a) = t\}$. The *'same-transaction' relation*, $\mathsf{st} \subseteq E \times E$, is the equivalence relation $\mathsf{st} \triangleq \left\{(a,b) \in E \times E \,\middle|\, \mathtt{tx}(a) = \mathtt{tx}(b)\right\}$.

Definition 2. *An* execution*, $G \in \mathit{Exec}$, is a tuple $(E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$, where:*

- *E is a set of* events*. The set of* reads *in E is $R \triangleq \left\{e \in E \,\middle|\, \mathtt{lab}(e) = (\mathtt{R}, -, -)\right\}$. The sets of* allocations *(M),* writes *(W),* aborts *(A),* transactional begins *(B),* transactional commits *(C) and* commit successes *(S) are analogous.*
- *$\mathsf{po} \subseteq E \times E$ denotes the* 'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread.*
- *$\mathsf{clo} \subseteq E \times E$ denotes the* 'client-order' relation*, which is a strict partial order between transactions ($\mathsf{st};\mathsf{clo};\mathsf{st} \subseteq \mathsf{clo} \setminus \mathsf{st}$) that extends the program order between transactions ($\mathsf{po} \setminus \mathsf{st} \subseteq \mathsf{clo}$).*
- *$\mathsf{rf} \subseteq (M \cup W) \times R$ denotes the* 'reads-from' relation *between events of the same location with matching values; i.e. $(a,b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a) = \mathtt{loc}(b) \wedge \mathtt{val_w}(a) = \mathtt{val_r}(b)$. Moreover, rf is total and functional on its range.*
- *$\mathsf{mo} \subseteq E \times E$ is the* 'modification-order'*, defined as the disjoint union of relations $\{\mathsf{mo}_x\}_{x \in \mathit{Loc}}$, such that each mo_x is a strict total order on $M_x \cup W_x$.*

Given a relation $\mathsf{r} \subseteq E \times E$, we write $\mathsf{r_T}$ for lifting r to transaction classes: $\mathsf{r_T} \triangleq \mathsf{st};(\mathsf{r} \setminus \mathsf{st});\mathsf{st}$. For instance, when $(w,r) \in \mathsf{rf}$, w is a transaction t_1 event and r is a transaction t_2 event, then all events in t_1 are $\mathsf{rf_T}$-related to all events in t_2. We write $\mathsf{r_I}$ to restrict r to its *intra-transactional* edges (within a transaction): $\mathsf{r_I} \triangleq \mathsf{r} \cap \mathsf{st}$; and write $\mathsf{r_E}$ to restrict r to its *extra-transactional* edges (outside a transaction): $\mathsf{r_E} \triangleq \mathsf{r} \setminus \mathsf{st}$. Analogously, we write $\mathsf{r_i}$ to restrict r to its *intra-thread edges*: $\mathsf{r_i} \triangleq \left\{(a,b) \in \mathsf{r} \,\middle|\, \mathtt{tid}(a) = \mathtt{tid}(b)\right\}$; and write $\mathsf{r_e}$ to restrict r to its *extra-thread edges*: $\mathsf{r_e} \triangleq \mathsf{r} \setminus \mathsf{r_i}$.

In the context of an execution G (we use the "G." prefix to make this explicit), the *reads-before* relation is $\mathsf{rb} \triangleq (\mathsf{rf}^{-1};\mathsf{mo})$.

Lastly, we write Commit for the events of *committing* transactions, i.e. those that have reached the commit stage: $\mathsf{Commit} \triangleq \mathit{dom}(\mathsf{st};[C])$. We define the sets of *aborted* events, Abort, and *(commit)-successful* events, Succ, analogously. We define the set of *commit-pending* events as $\mathsf{CPend} \triangleq \mathsf{Commit} \setminus (\mathsf{Abort} \cup \mathsf{Succ})$, and the set of *pending* events as $\mathsf{Pend} \triangleq E \setminus (\mathsf{CPend} \cup \mathsf{Abort} \cup \mathsf{Succ})$.

Given an execution $G = (E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$, we write $G|_A$ for $(E \cap A, \mathsf{po}|_{E\cap A}, \mathsf{clo}|_{E\cap A}, \mathsf{rf}|_{E\cap A}, \mathsf{mo}|_{E\cap A})$. We further impose certain "well-formedness" conditions on executions, used to delimit transactions and restrict allocations. For example, we require that events of the same transaction are by the same thread and the

each t contains exactly one begin event. In particular, these conditions ensure that in the context of a well-formed execution G we have 1. $G.\mathsf{Succ} \subseteq G.\mathsf{Commit}$; 2. each t contains at most a single abort or success ($|G.E_t \cap (A \cup S)| \leq 1$) and thus $G.(\mathsf{Succ} \cap \mathsf{Abort}){=}\emptyset$; and 3. $G.E = G.(\mathsf{Pend} \uplus \mathsf{Abort} \uplus \mathsf{CPend} \uplus \mathsf{Succ})$, i.e. the sets $G.\mathsf{Pend}$, $G.\mathsf{Abort}$, $G.\mathsf{CPend}$ and $G.\mathsf{Succ}$ are pair-wise disjoint.

Execution Consistency. The definition of (well-formed) executions above puts very few constraints on the rf and mo relations. Such restrictions and thus the permitted behaviours of a transactional program are determined by defining the set of *consistent* executions, defined separately for each transactional consistency model. The existing literature includes several definitions of well-known consistency models, including *serialisability* (SER) [41], *snapshot isolation* (SI) [9,44] and *parallel snapshot isolation* (PSI) [10,43].

Serialisability (SER). The *serialisability* (SER) consistency model [41] is one of the most well-known transactional consistency models, as it provides strong guarantees that are intuitive to understand and reason about. Specifically, under SER, all concurrent transactions must appear to execute atomically one after another in a *total sequential order*. The existing declarative definitions of SER [9,10,50] are somewhat restrictive in that they only account for fully committed (complete) transactions, i.e. they do not support pending or aborted transactions. Under the assumption that all transactions are complete, an execution $(E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$ is deemed to be serialisable (i.e. SER-consistent) iff:

- $\mathsf{rf}_\mathsf{I} \cup \mathsf{mo}_\mathsf{I} \cup \mathsf{rb}_\mathsf{I} \subseteq \mathsf{po}$ (SER-INT)
- $\mathsf{clo} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$ is acyclic. (SER-EXT)

The SER-INT axiom enforces *intra-transactional* consistency, ensuring that e.g. a transaction observes its own writes by requiring $\mathsf{rf}_\mathsf{I} \subseteq \mathsf{po}$ (i.e. intra-transactional reads respect the program order). Analogously, the SER-EXT axiom guarantees *extra-transactional* consistency, ensuring the existence of a total sequential order in which all concurrent transactions appear to execute atomically one after another. This total order is obtained by an arbitrary extension of the (partial) 'happens-before' relation which captures synchronisation resulting from transactional orderings imposed by client order (clo) or *conflict* between transactions ($\mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$). Two transactions are conflicted if they both access (read or write) the same location x, and at least one of these accesses is a write. As such, the inclusion of $\mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$ enforces conflict-freedom of serialisable transactions. For instance, if transactions t_1 and t_2 both write to x via events w_1 and w_2 such that $(w_1, w_2) \in \mathsf{mo}$, then t_1 must commit before t_2, and thus the entire effect of t_1 must be visible to t_2.

Opacity. We do not stipulate that all transactions commit successfully and allow for both aborted and pending transactions. As such, we opt for the stronger notion of transactional correctness known as *opacity*. In what follows we describe our notion of opacity over executions (formalised in Def. 3), and later relate it to the existing notion of opacity over histories [27] and prove that our characterisation of opacity is *equivalent* to that of the existing one (see Thm. 1). Further intuitions are provided in the extended version of this paper [46].

Definition 3 (Opacity). *An execution* $G = (E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$ *is* opaque *iff:*

- $dom(\mathsf{rf}_\mathsf{T}) \subseteq \mathsf{Vis}$ *(VIS-RF)*
- $\mathsf{rf}_\mathsf{I} \cup \mathsf{mo}_\mathsf{I} \cup \mathsf{rb}_\mathsf{I} \subseteq \mathsf{po}$ *(INT)*
- $(\mathsf{clo} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup (\mathsf{rb}_\mathsf{T}; [\mathsf{Vis}]))$ *is acyclic* *(EXT)*

where $\mathsf{Vis} \triangleq \mathsf{Succ} \cup \mathsf{CPendRF}$ *with* $\mathsf{CPendRF} \triangleq dom([\mathsf{CPend}]; \mathsf{rf}_\mathsf{T})$.

The existing definition of opacity [27] does not account for memory allocation and assumes that all locations accessed (read/written) by a transaction are initialised with some value (typically 0). In our setting, we make no such assumption and extend the notion of opacity to *dynamic opacity* to account for memory allocation. More concretely, our goal is to ensure that accesses in *visible* transactions are *valid*, in that they are on locations that have been previously allocated in a visible transaction. We define an execution to be dynamically opaque (Def. 4) if its visible write accesses are valid, i.e. are mo-preceded by a visible allocation.

Definition 4 (Dynamic opacity). *An execution* G *is* dynamically opaque *iff it is opaque (Def. 3) and* $G.(W \cap \mathsf{Vis}) \subseteq rng\big([M \cap \mathsf{Vis}]; G.\mathsf{mo}\big)$.

We next use the above definitions to define (dynamic durable) opacity over execution *histories*. In the context of persistent memory where executions may crash (e.g. due to a power failure) and resume thereafter upon recovery, a history is a sequence of events (Def. 1) partitioned into different *eras* separated by *crash markers* (recording a crash occurrence), provided that the threads in each era are distinct, i.e. thread identifiers from previous eras are not reused after a crash.

Definition 5 (Histories). *A* history, $H \in \textsc{Hist}$, *is a pair* (E, to), *where* E *comprises events and* crash markers, $E \subseteq \textsc{Event} \cup \textsc{Crash}$ *with* $\textsc{Crash} \triangleq \big\{(n, \lightning) \,\big|\, n \in \mathbb{N}\big\}$, *and* to *is a total order on* E, *such that:*

- $(E, \mathsf{to}_\mathsf{i})$ *is well-formed; and*
- *events separated by crashes have distinct threads:* $([E]; \mathsf{to}; [\textsc{Crash}]; \mathsf{to}; [E]) \cap \mathsf{to}_\mathsf{i} = \emptyset$.

A history (E', pto) *is a* prefix *of history* (E, to) *iff* $E' \subseteq E$, $\mathsf{pto} = \mathsf{to}|_{E'}$ *and* $dom(\mathsf{to}; [E']) \subseteq E'$.

The *client order* induced by a history $H = (E, \mathsf{to})$, denoted by $\mathsf{clo}(H)$, is the partial order on TXID defined by $\mathsf{clo}(H) \triangleq [S \cup A]; \mathsf{to}_\mathsf{T}; [B]$. We define history opacity as a *prefix-closed* property (*cf.* [27]), designating a history H as opaque if *every prefix* (E, pto) of H induces an opaque execution. The notion of dynamic opacity over histories is defined analogously.

Definition 6. *A history* H *is* opaque *iff for each prefix* $H_p = (E, \mathsf{pto})$ *of* H, *there exist* rf, mo *such that* $(E, \mathsf{pto}_\mathsf{i}, \mathsf{clo}(H_p), \mathsf{rf}, \mathsf{mo})$ *is opaque (Def. 3).* H *is* dynamically opaque *iff for each prefix* $H_p{=}(E, \mathsf{pto})$ *of* H, *there exist* rf, mo *such that* $(E, \mathsf{pto}_\mathsf{i}, \mathsf{clo}(H_p), \mathsf{rf}, \mathsf{mo})$ *is dynamically opaque (Def. 4).*

We define *durable opacity* over histories: a history H is durably opaque iff the history obtained from H by removing crash markers is opaque. We define *dynamic, durable opacity* analogously.

Definition 7. *A history* (E, to) *is* durably opaque *iff* $(E \setminus \textsc{Crash}, \mathsf{to}|_{E \setminus \textsc{Crash}})$ *is opaque. A history* (E, to) *is* dynamically and durably opaque *iff the history* $(E \setminus \textsc{Crash}, \mathsf{to}|_{E \setminus \textsc{Crash}})$ *is dynamically opaque.*

Finally, we show that our definitions of history (durable) opacity are equivalent to the original definitions in the literature. (See [46] for the proof.)

Theorem 1. *History opacity as defined in Def. 6 is equivalent to the original notion of opacity [27]. History durable opacity as defined in Def. 7 is equivalent to the original notion of durable opacity [6].*

5 Operationally Proving Dynamic Durable Opacity

We develop an operational specification, ddTMS (§5.1), and prove it correct against ddOpacity (§5.2). In particular, we show that every history (i.e. observable trace) of ddTMS satisfies ddOpacity. As ddTMS is a concurrent operational specification, it serves as basis for validating the correctness of txPMDK as well as our concurrent extensions PMDK-TML and PMDK-NORec.

5.1 ddTMS: The dTMS2 Automaton Extended with Allocation

ddTMS is based on dTMS2, which is an operational specification that guarantees durable opacity [6]. dTMS2 in turn is based on TMS2 automaton [20], which is known to satisfy opacity [33]. Furthermore, the ddTMS commit operation includes the simplification described by Armstrong et al [1], omitting a validity check when committing read-only transactions. In what follows we present ddTMS as a transition system.

ddTMS state. Formally, the state of ddTMS is given by the variables in Fig. 7. dTMS2 keeps track of a sequence of memory stores, mems, one for each committed writing transaction since the last crash. This allows us to determine whether reads are consistent with previously committed write operations. Each committing transaction that contains at least one write adds a new memory version to the end of the memory sequence. As we shall see, mems tracks allocated locations since it maps every allocated location to a value different from $\bot$.

Each transaction t is associated with several variables: pc_t, $beginIdx_t$, $rdSet_t$, $wrSet_t$ and $alSet_t$. The pc_t denotes the program counter, ranging over a set of *program counter values* ensuring each transaction is well-formed and that each transactional operation takes effect between its invocation and response. The $beginIdx_t \in \mathbb{N}$ denotes the *begin index*, set to the index of the most recent memory version when the transaction begins. This is used to ensure the real-time ordering property between transactions. The $rdSet_t \in \textsc{Loc} \rightharpoonup \textsc{Val}$ is the *read set* and $wrSet_t \in \textsc{Loc} \rightharpoonup \textsc{Val}$ is the *write set*, recording the values read and written by

$$\mathsf{mems} \in \textsc{Mem} \triangleq \textsc{Seq}\,\langle \textsc{Loc} \to \textsc{Val}_\bot \rangle \qquad \textsc{Val}_\bot \triangleq \textsc{Val} \cup \{\bot\}, \text{where } \bot \notin \textsc{Val}$$
$$\mathsf{S} \in \textsc{State} \triangleq \textsc{TXId} \to \textsc{TState}$$
$$\mathsf{s} \in \textsc{TState} \triangleq \mathbb{N} \times (\textsc{Loc} \rightharpoonup \textsc{Val}) \times (\textsc{Loc} \rightharpoonup \textsc{Val}) \times \mathcal{P}\,(\textsc{Loc})$$
storing the local begin index, read set, write set and allocation set
$$\mathsf{PC} \in \textsc{PCMap} \triangleq \textsc{TXId} \to \textsc{PCVal}$$
$$\textsc{Invs} \triangleq \left\{ \begin{array}{l} \texttt{TxBegin}, \texttt{TxRead}(l), \texttt{TxWrite}(l, v), \\ \texttt{TxAlloc}, \texttt{TxCommit} \end{array} \middle| \; l \in \textsc{Loc}, v \in \textsc{Val} \right\}$$
$$\textsc{Resps} \triangleq \left\{ \begin{array}{l} \texttt{TxBegin}, \texttt{TxRead}(l, v), \texttt{TxWrite}(l, v), \\ \texttt{TxAlloc}(l), \texttt{TxCommit}, \texttt{Abort} \end{array} \middle| \; l \in \textsc{Loc}, v \in \textsc{Val} \right\}$$
$$\textsc{PCVal} \triangleq \{\text{init}, \text{ready}, \text{aborted}, \text{committed}, \text{fault}, \mathit{\Pi}(i), \mathit{\Delta}(\texttt{TxCommit}) \mid i \in \textsc{Invs}\}$$
$$\alpha \in \textsc{Action} \triangleq \{\mathit{inv}(i), \mathit{res}(r), \varepsilon, \lightning \mid i \in \textsc{Invs}, r \in \textsc{Resps}\}$$

Initially, $\mathsf{PC}_0 \triangleq \lambda t.\text{init}$ $\quad \mathsf{S}_0 \triangleq \lambda t.(0, \emptyset, \emptyset, \emptyset)$ $\quad \mathsf{mems}_0 \triangleq [\lambda x.\ \bot]$

Fig. 7: ddTMS state

the transaction during its execution, respectively. We use $S \rightharpoonup T$ to denote a partial function from S to T. Finally, $\mathit{alSet}_t \subseteq \textsc{Loc}$ denotes the *allocation set*, containing the set of locations allocated by the transaction t. We use $\mathsf{s.beginIdx}$, $\mathsf{s.rdSet}$, $\mathsf{s.wrSet}$ and $\mathsf{s.alSet}$ to refer to the begin index, read set, write set and allocation set of a state s, respectively.

The read set is used to determine whether the values read by the transaction are consistent with its version of memory (using $\mathsf{validIdx}$). The write set, on the other hand, is required because writes are modelled using *deferred update* semantics: writes are recorded in the transaction's write set and are not published to any shared state until the transaction commits.

ddTMS Global Transitions. ddTMS is specified by the transition system shown in Fig. 8, where the ddTMS global transitions are given at the top and the per-transaction transitions are given at the bottom. The global transitions may either take a per-transaction step (rule (S)), match a transaction fault (rule (F)), crash (rule (X)), or behave chaotically due to a fault (rule (C)).

Note that a *crash* transition models both a crash and a recovery. It sets the program counter of every live transaction to aborted, preventing them from performing any further actions after the crash. Since transaction identifiers are not reused, the program counters of completed transactions need not be modified. After restarting, it must not be possible for any new transaction to interact with stale memory states prior to the crash. Thus, we reset the memory sequence to be a singleton sequence containing the last memory state prior to the crash.

Following the design of txPMDK (and our concurrent extensions PMDK-TML and PMDK-NORec) we do not check for reads and writes to unallocated memory within the library and instead delegate such checks to the client. An execution of txPMDK (as well as PMDK-TML and PMDK-NORec) that accesses unallocated memory is assumed to be faulty. In particular, a read or write of unallocated memory induces a *fault* (rule (F)). Once a fault is triggered, the program counter of each transaction is set to "fault" and recovery is impossible.

$$\mathsf{validIdx}(n,\mathsf{s},\mathsf{mems}) \triangleq \mathsf{s.beginIdx} \le n < |\mathsf{mems}| \wedge \mathsf{s.rdSet} \subseteq \mathsf{mems}(n) \wedge \mathsf{s.alSet} \subseteq \{l \mid \mathsf{mems}(n)(l) = \bot\}$$

$$\frac{\mathsf{PC}(t), \mathsf{S}(t), \mathsf{mems} \xrightarrow{\alpha} \mathsf{pc}, \mathsf{s}, \mathsf{mems}' \quad \mathsf{pc} \neq \mathsf{fault}}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\alpha_t} \mathsf{PC}[t \mapsto \mathsf{pc}], \mathsf{S}[t \mapsto \mathsf{s}], \mathsf{mems}'} \text{ (S)} \qquad \frac{\mathsf{PC}(t), \mathsf{S}(t), \mathsf{mems} \xrightarrow{fault} \mathsf{fault}, \mathsf{s}, \mathsf{mems}' \quad \mathsf{PC}' = \lambda t.\mathsf{fault}}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{fault} \mathsf{PC}', \mathsf{S}[t \mapsto \mathsf{s}], \mathsf{mems}'} \text{ (F)}$$

$$\frac{\mathsf{PC}' = \lambda t.\ \texttt{if}\ \mathsf{PC}(t) \notin \{\mathrm{init}, \mathrm{committed}, \mathrm{fault}\}\ \texttt{then}\ \mathrm{aborted}\ \texttt{else}\ \mathsf{PC}(t)}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\lightning} \mathsf{PC}', \mathsf{S}, \langle last(\mathsf{mems})\rangle} \text{ (X)} \qquad \frac{\mathsf{PC} = \lambda t.\mathrm{fault}}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\alpha_t} \mathsf{PC}, \mathsf{S}, \mathsf{mems}} \text{ (C)}$$

$$\text{(IB)}\ \frac{\mathsf{pc} = \mathrm{init}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{inv(\texttt{TxBegin})} \Delta(\texttt{TxBegin}), \mathsf{s}', \mathsf{mems}} \qquad \text{(DB)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxBegin}) \quad \mathsf{s}' = \mathsf{s}[\mathsf{beginIdx} \mapsto |\mathsf{mems}| - 1]}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxBegin})} \mathrm{ready}, \mathsf{s}, \mathsf{mems}} \qquad \text{(IOp)}\ \frac{\mathsf{pc} = \mathrm{ready} \quad a \in \mathit{InvOps}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{inv(a)} \Delta(a), \mathsf{s}, \mathsf{mems}}$$

$$\text{(DR-E)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxRead}(l)) \quad l \notin \mathsf{s.alSet} \cup dom(\mathsf{s.wrSet}) \quad \mathsf{validIdx}(n, \mathsf{s}, \mathsf{mems}) \quad \mathsf{mems}(n)(l) = v \quad v \neq \bot \quad rs = \mathsf{s.rdSet} \oplus \{l \mapsto v\}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxRead}(l,v))} \mathrm{ready}, \mathsf{s}[\mathsf{rdSet} \mapsto rs], \mathsf{mems}}$$

$$\text{(FR)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxRead}(l)) \quad l \notin \mathsf{s.alSet} \cup dom(\mathsf{s.wrSet}) \quad \mathsf{validIdx}(n, \mathsf{s}, \mathsf{mems}) \quad \mathsf{mems}(n)(l) = \bot}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{fault} \mathrm{fault}, \mathsf{s}, \mathsf{mems}}$$

$$\text{(RA)}\ \frac{\mathsf{pc} \notin \left\{\begin{array}{l}\mathrm{init}, \mathrm{ready},\\ \mathrm{committed},\\ \mathrm{aborted}, \mathrm{fault}\end{array}\right\}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{Abort})} \mathrm{aborted}, \mathsf{s}, \mathsf{mems}}$$

$$\text{(DR-I)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxRead}(l)) \quad l \in dom(\mathsf{s.wrSet}) \quad \mathsf{s.wrSet}(l) = v}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxRead}(l,v))} \mathrm{ready}, \mathsf{s}, \mathsf{mems}} \qquad \text{(DR-A)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxRead}(l)) \quad l \notin dom(\mathsf{s.wrSet}) \quad l \in \mathsf{s.alSet}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxRead}(l,0))} \mathrm{ready}, \mathsf{s}, \mathsf{mems}}$$

$$\text{(DW)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxWrite}(l,v)) \quad l \in \mathsf{s.alSet} \vee last(\mathsf{mems})(l) \neq \bot \quad ws = \mathsf{s.wrSet} \oplus \{l \mapsto v\}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxWrite}(l,v))} \mathrm{ready}, \mathsf{s}[\mathsf{wrSet} \mapsto ws], \mathsf{mems}}$$

$$\text{(FW)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxWrite}(l,v)) \quad l \notin \mathsf{s.alSet} \quad last(\mathsf{mems})(l) = \bot}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{fault} \mathrm{fault}, \mathsf{s}, \mathsf{mems}} \qquad \text{(DA)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxAlloc}) \quad l \notin \mathsf{s.alSet} \quad as = \mathsf{s.alSet} \uplus \{l\}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxAlloc}(l))} \mathrm{ready}, \mathsf{s}[\mathsf{alSet} \mapsto as], \mathsf{mems}}$$

$$\text{(DC-RO)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxCommit}) \quad \mathsf{s.alSet} = \emptyset \quad dom(\mathsf{s.wrSet}) = \emptyset}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\varepsilon} \Pi(\texttt{TxCommit}), \mathsf{s}, \mathsf{mems}}$$

$$\text{(RC)}\ \frac{\mathsf{pc} = \Pi(\texttt{TxCommit})}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{res(\texttt{TxCommit})} \mathrm{committed}, \mathsf{s}, \mathsf{mems}} \qquad \text{(DC-W)}\ \frac{\mathsf{pc} = \Delta(\texttt{TxCommit}) \quad \mathsf{validIdx}(last(\mathsf{mems}), \mathsf{s}, \mathsf{mems}) \quad \mathsf{mems}' = \mathsf{mems} +\!\!+ ((last(\mathsf{mems}) \oplus \{l \mapsto 0 \mid l \in \mathsf{s.alSet}\}) \oplus \mathsf{s.wrSet})}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\varepsilon} \Pi(\texttt{TxCommit}), \mathsf{s}, \mathsf{mems}'}$$

Fig. 8: The DDTMS global transitions (above) with its per-transaction transitions (below), where
$\mathit{InvOps} \triangleq \{\texttt{TxWrite}(l,v), \texttt{TxRead}(l) \mid l \in \mathrm{Loc}, v \in \mathrm{Val}\} \cup \{\texttt{TxAlloc}, \texttt{TxCommit}\}$

From a faulty state the system behaves chaotically, i.e. it is possible to generate any history using rule (C).

ddTMS Per-Transaction Transitions. The system contains externally visible transitions for *invoking* an operation (rules IB and IOp), which set the program counters to $\Delta(a)$, where a is the operation being performed. This allows the histories of the system to contain operation invocations without corresponding matching responses.

For the begin, allocation, read and write operations, an invocation can be followed by a single transition (rules DB, DA, DR-E, DR-I, DR-E and DW) that performs the operation combined with the corresponding *response*. Following an invocation, the commit operation is split into internal do actions ((DC-RO) and (DC-W)) and an external response (rule RC). Finally, after a read/write invocation, the system may perform a *fault transition* for a read (rule FR) or a write (rule FW). The main change from dTMS2 is the inclusion of an allocation procedure. The design of ddTMS allows the executing transaction, t, to tentatively allocate a location l within its transaction-local allocation set, $alSet_t$. This allocation in ddTMS is optimistic – correctness of the allocation is only checked when t performs a read or commits.

Successful (non-faulty) read and write operations take allocations into account as follows. **(1)** A read operation of transaction t reads from a prior write (rule (DR-I)) or allocation (rule (DR-A)) performed by t itself. In this case, the operation may only proceed if the location l is either in the allocation or write set of t. The effect of the operation is to return the value of l in the write set (if it exists) or 0 if it only exists in the allocation set. **(2)** A read operation of transaction t reads from a write or allocation performed by another transaction (rule (DR-E)). Note that as with dTMS2 and TMS2, in ddTMS a read-only transaction may serialise with any memory index n after $beginIdx_t$. Moreover, within validIdx, in addition to ensuring that t's read set is consistent with the memory index n (second conjunct), we must also ensure that t's allocation set is consistent with memory index n (third conjunct) by ensuring that none of the locations in the allocation set have been allocated at memory index n. **(3)** A write of transaction t successfuly performs its operation (rule (DW)), which can only happen if the location l being written has been allocated, either by t itself (first disjunct), or by a prior transaction (second disjunct). A writing transaction must serialise after the last memory index in mems, thus the second disjunct checks allocation against the last memory index.

A successful (non-faulty) transaction is split into two cases: **(1)** t is a read-only transaction (rule (DC-RO)), where both $alSet_t$ and $wrSet_t$ are empty for t. In this case, the transaction simply commits. **(2)** t has performed an allocation or a write (rule (DC-W)). Here, we check that t is valid with respect to the last memory in mems using validIdx. The commit introduces a new memory into the memory sequence mems. The update also ensures that all pending allocations in $alSet_t$ take effect before applying the writes from t's write set.

5.2 Soundness of DDTMS

We state our main theorem relating DDTMS to DDOPACITY. As the models are inherently different, we need several definitions to transform DDTMS histories to those compatible with DDOPACITY.

An *execution* of a labelled transition system (LTS) is an alternating sequence of states and actions, i.e. a sequence of the form $s_0\, a_1\, s_2\, a_2 \ldots s_{n-1}\, a_n\, s_n$ such that for each $0 < i \leq n$, $s_{i-1} \xrightarrow{a_i} s_i$ and s_0 is an initial state of the LTS. Suppose σ is an execution of DDTMS. We let $AH_\sigma = a_1\, a_2 \ldots a_n$ be the *action history* corresponding to σ, and EH_σ be the *external history* of σ, which is AH_σ restricted to non-ϵ actions. Let FF_σ be the longest *fault-free prefix* of EH_σ. We generate the history (in the sense of Def. 5) corresponding to FF_σ as follows. First, we construct the *labelled history*, LH_σ of σ from FF_σ by removing all invocation actions (leaving only responses and crashes). Then, we replace each response $a_i = \alpha_t$ by the event $(i, t, t, L(\alpha))$, where $L(res(\texttt{TxBegin})) = \texttt{B}$, $L(res(\texttt{TxAlloc}(l))) = (\texttt{M}, l, 0)$, $L(res(\texttt{TxRead}(l, v))) = (\texttt{R}, l, v)$, $L(res(\texttt{TxWrite}(l, v))) = (\texttt{W}, l, v)$, $L(res(\texttt{Abort})) = \texttt{A}$, $L(inv(\texttt{TxCommit})) = \texttt{C}$, and $L(res(\texttt{TxCommit})) = \texttt{S}$. Similarly, we replace each crash action $a_i = \lightning$ by the pair $(i, \lightning)$. Note that in this construction, for simplicity, we conflate threads and transactions, but this restriction is straightforward to generalise. Finally, let the *ordered history* of σ, denoted OH_σ, be the total order corresponding to LH_σ.

Theorem 2. *For any execution σ of DDTMS, the ordered history OH_σ satisfies DDOPACITY.*

The definitions of (dynamic) durable opacity can lifted to the level of systems in the standard manner, providing a notion of correctness for implementations [28].

6 Modelling and Validating Correctness in FDR4

FDR4 [26] is a model checker for CSP [29] that has recently been used to verify linearisability [38], as well as opacity and durable opacity [23]. We similarly provide an FDR4 development, which allows proofs of refinement to be *automatically* checked up to certain bounds. This is in contrast to manual methods of proving correctness of concurrent objects [21,19], which require a significant amount of manual human input (though such manual proofs are unbounded).

An overview of our FDR4 development [47] is given in Fig. 9. We derive two specifications from DDTMS. The first is an FDR4 model of DDTMS itself, based on prior work [38,23], but contains the extensions described in §5.1. The second is DDTMS-Seq, which restricts DDTMS to a sequential crash-free specification. We use DDTMS-Seq to obtain (lower-bound) liveness-like guarantees, which strengthens traditional deadlock or divergence proofs of refinement. These lower-bound checks ensure our models contain at least the traces of DDTMS-Seq.

Fig. 10 summarises our experiments on the upper bound checks, where the times shown combine the compilation and model exploration times. Each row represents an experiment that bounds the number of transactions (#txns),

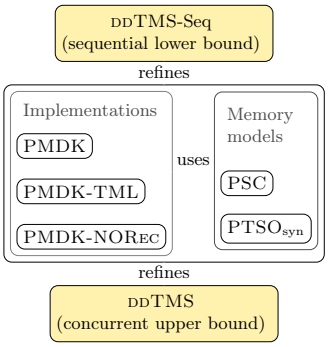


Fig. 9: Overview of FDR4 checks

Memory	#txns	#locs	#val	#buff	TX-PMDK	PMDK-TML	PMDK-NOREC
PSC	2	2	2	2	5.83s	5.90s	6.74s
PSC	2	3	2	2	201.03s	213.97s	271.35s
PSC	2	2	3	2	21.65s	23.47s	27.40s
PSC	2	2	2	3	5.83s	5.78s	6.60s
$PTSO_{syn}$	2	1	2	2	0.61s	3.96s	1.57s
$PTSO_{syn}$	2	2	2	2	6.67s	6.71s	7.73s
$PTSO_{syn}$	2	3	2	2	267.1s	268.91s	319.18s
$PTSO_{syn}$	2	2	3	2	24.10s	25.53s	29.24s
$PTSO_{syn}$	2	2	2	3	14.37s	14.19s	15.41s

Fig. 10: Summary of upper bounds checks (total time in seconds: compilation + model exploration). The time out (TO) is set to 1000 seconds of compilation time.

locations (#locs), values (#val) and the size of the persistency and store buffers (#buff). The times reported are for an Apple M1 device with 16GB of memory. The first row depicts a set of experiments where the implementations execute directly on NVM, without any buffers. As we discuss below, these tests are sufficient for checking lower bounds. The baseline for our checks sets the value of each parameter to two, and Fig. 10 allows us to see the cost of increasing each parameter. Note that all models time out when increasing the number of transactions to three, thus these times are not shown. Also note that for TXPMDK (which is single-threaded), the checks for PSC also cover $PTSO_{syn}$, since $PTSO_{syn}$ is equivalent to PSC in the absence of races [31]. Nevertheless, it is interesting to run the single-threaded experiments on the $PTSO_{syn}$ model to understand the impact of the memory model on the checks.

In our experiments we use FDR4's built-in *partial order reduction* features to make the upper bound checks feasible. This has a huge impact on the model checking speed; for instance, the check for PMDK-TML with two transactions, two locations, two values and buffer size of two reduces from over 6000 seconds (1 hour and 40 minutes) to under 7 seconds, which is almost a 1000-fold improvement! This speed-up makes it feasible to use FDR4 for rapid prototyping when developing programs that use TXPMDK, even for the relatively complex $PTSO_{syn}$ memory model.

7 Related Work

Crash Consistency. Several authors have defined notions of atomicity for *concurrent objects* that take persistency into account (see [4] for a survey.) None of these conditions are suitable as they define consistency for concurrent operations (of concurrent data structures) as opposed to transactional memory.

Approaches and semantics to *crash-consistent* transactions stretch back to the mid 1970s, which considered the problem in the database setting [24,34]. Since then, a myriad of definitions have been developed for particular applications

(e.g. distributed systems, file systems, etc.). For plain reads and writes, one of the first studies of persistency models focussed on NVM is by Pelley et al. [42]. Since then, several semantic models for real hardware (Intel and ARM) have been developed [50,49,31,12,48]. For transactional memory, there are only a few notions that combine a notion of crash consistency with ACID guarantees as required for concurrent durable transactions. Raad et al. [50] define a *persistent serializability* under relaxed memory, which does not handle aborted transactions. As we have already discussed, Bila et al. [6] define *durable opacity*, but this is defined in terms of (totally ordered) histories as opposed to partially ordered graphs. Neither persistent serialisability nor durable opacity handle allocation.

Validating the TXPMDK Implementation. Even without a clear consistency condition, a range of papers have explored correctness of the C/C++ implementation. Bozdogan et al. [8] built a sanitiser for persistent memory and used it to uncover memory-safety violations in TXPMDK. Fu et al. [25] have built a tool for testing persistent key-value stores and uncovered consistency bugs in the PMDK libraries. Liu et al. [36] have built a tool for detecting cross-failure races in persistent programs, and uncovered a bug in PMDK's libpmemobj library (see 'Bug 4' in their paper). They are at a different level of abstraction than ours since they focus on the code itself and do not provide any description of the design principles behind PMDK.

Raad et al. [45] and Bila et al. [7] have developed logics for reasoning about programs over the Px86-TSO model (which we recall is equivalent to PTSO_{syn}). However, these logics have thus far only been applied to small examples. Extending these logics to cover a proof by simulation and a full (manual) proof of correctness of PMDK, PMDK-TML and PMDK-NOREC would be a significant undertaking, but an interesting avenue for future work.

Transactional Memory (TM). Several works have studied the semantics of TM [15,22,44,43]. However, our works differ from those in that they do not account for persistency guarantees and crash consistency. However, while earlier works [44,43] merely *propose* a model for weak isolation (i.e. mixing transactional and non-transactional accesses), [15,22] formalise the weak isolation in various hardware and software TM platforms, albeit without validating their semantics.

Several approaches to crash consistency have recently been proposed. For a survey and comparison of techniques (in addition to transactions) see [3]. OneFile [52], Romulus [16], and Trinity and Quadra [51] together describe a set of algorithms that aim to improve the efficiency of TXPMDK by reducing the number of fence instructions. Liu et al. [35] present DudeTM, a persistent TM design that uses a shadow copy of NVM in DRAM, which is is shared amongst all transactions. Their approach comprises three key steps: Zardoshti et al. [56] present an alternative technique for making STMs persistent by instrumenting STM code with additional logging and flush instructions. However, none of these works have defined any formal correctness guarantees, and hence do not offer any proofs of correctness either. In particular, the role of allocation and its interaction with reads and writes is generally unclear.

As well as defining durable opacity, Bila et al. [6] develop a persistent version of the TML STM [17] by introducing explicit undo logging and flush instructions. They then prove this to be durably opaque via the DTMS2 specification. More recently, Bila et al. [5] have developed a technique for transforming both an STM and its corresponding opacity proof by delegating reads/writes to memory locations controlled by the TM to an abstract library that is later refined to use volatile and non-volatile memory. Neither of these works use TXPMDK, and are over a sequentially consistent memory model.

8 Conclusions and Future Work

Our main contribution is validating the correctness for TXPMDK via the development of declarative (DDOPACITY) and operational (DDTMS) consistency criteria. We provide an abstraction of TXPMDK and show that it satisfies DDTMS and hence DDOPACITY by extension. Additionally, we develop PMDK-TML and PMDK-NOREC as two concurrent extensions of TXPMDK that are based on existing STM designs, and show that these also satisfy DDTMS (and hence DDOPACITY). All of our models are validated under the PSC and PTSO_{syn} memory models using FDR4.

As with most accepted existing transactional models (be it with or without persistency), we assume *strong isolation*, where each non-transactional access behaves like a singleton transaction (a transaction with a single access). That is, even ignoring persistency, there are no accepted definitions or models for mixing non-transactional and transactional accesses, and all existing transactional models (including opacity and serialisability) assume strong isolation. Indeed, PMDK transactions are specifically designed to be used in a purely transactional setting and are not meant to be used in combination with non-transactional accesses; i.e. they would have undefined semantics otherwise. Consequently, as we do not consider mixing transactional code with non-transactional code, RMW (read-modify-write) instructions are irrelevant in our setting. Specifically, as non-transactional access are treated as singleton transactions, RMW instructions are not needed or relevant since they behave as transactions and their atomicity would be guaranteed by the transactional semantics.

One threat to validity of our work is that the model checking results are on a small number of transactions, locations, values, and buffer sizes (see Fig. 10). However, we have found that these sizes have been adequate for validating all of our examples, i.e., when errors are deliberately introduced, FDR validation fails and counter-examples are automatically generated. Currently, we do not know whether there is a small model theorem for durable opacity in general. This is a separate line of work and a general question that we believe is out of the scope of this paper. Specifically, our focus here is on making PMDK transactions concurrent, providing a clear specification for PMDK (and its concurrent variations) with dynamic allocation, and validating correctness of the results under a realistic memory model.

References

1. Armstrong, A., Dongol, B., Doherty, S.: Proving opacity via linearizability: A sound and complete method. In: Bouajjani, A., Silva, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 50–66. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_4
2. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: Safety of live transactions in transactional memory: TMS is necessary and sufficient. In: Kuhn, F. (ed.) DISC. Lecture Notes in Computer Science, vol. 8784, pp. 376–390. Springer (2014). https://doi.org/10.1007/978-3-662-45174-8_26
3. Baldassin, A., Barreto, J., Castro, D., Romano, P.: Persistent memory: A survey of programming support and implementations. ACM Comput. Surv. **54**(7), 152:1–152:37 (2022). https://doi.org/10.1145/3465402
4. Ben-David, N., Friedman, M., Wei, Y.: Survey of persistent memory correctness conditions. CoRR **abs/2208.11114** (2022). https://doi.org/10.48550/ARXIV.2208.11114
5. Bila, E., Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Modularising verification of durable opacity. Log. Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/LMCS-18(3:7)2022
6. Bila, E., Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Defining and verifying durable opacity: Correctness for persistent software transactional memory. In: Gotsman, A., Sokolova, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 12136, pp. 39–58. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_3
7. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based Owicki-Gries reasoning for persistent x86-TSO. In: Sergey, I. (ed.) ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 234–261. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_9
8. Bozdogan, K.K., Stavrakakis, D., Issa, S., Bhatotia, P.: SafePM: a sanitizer for persistent memory. In: Bromberg, Y., Kermarrec, A., Kozyrakis, C. (eds.) EuroSys. pp. 506–524. ACM (2022). https://doi.org/10.1145/3492321.3519574
9. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. pp. 55–64 (2016)
10. Cerone, A., Gotsman, A., Yang, H.: Transaction chopping for parallel snapshot isolation. In: DISC. vol. 9363, pp. 388–404 (2015)
11. Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M.F., Zeldovich, N.: GoJournal: a verified, concurrent, crash-safe journaling system. In: Brown, A.D., Lorch, J.R. (eds.) OSDI. pp. 423–439. USENIX Association (2021)
12. Cho, K., Lee, S., Raad, A., Kang, J.: Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In: Freund, S.N., Yahav, E. (eds.) PLDI. pp. 16–31. ACM (2021). https://doi.org/10.1145/3453483.3454027
13. Choe, J.: Review and things to know: Flash memory summit 2022. TechInsights (August 2022), https://www.techinsights.com/blog/review-and-things-know-flash-memory-summit-2022
14. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 211–225. ACM (2018). https://doi.org/10.1145/3192366.3192373

15. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, power, arm, and C++. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 211–225. ACM (2018). https://doi.org/10.1145/3192366.3192373
16. Correia, A., Felber, P., Ramalhete, P.: Romulus: Efficient algorithms for persistent transactional memory. In: Scheideler, C., Fineman, J.T. (eds.) Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018. pp. 271–282. ACM (2018). https://doi.org/10.1145/3210377.3210392
17. Dalessandro, L., Dice, D., Scott, M.L., Shavit, N., Spear, M.F.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M.R., Talia, D. (eds.) Euro-Par. Lecture Notes in Computer Science, vol. 6272, pp. 2–13. Springer (2010). https://doi.org/10.1007/978-3-642-15291-7_2
18. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining STM by abolishing ownership records. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPoPP. pp. 67–78. ACM (2010). https://doi.org/10.1145/1693453.1693464
19. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Mechanized proofs of opacity: a comparison of two techniques. Formal Aspects Comput. **30**(5), 597–625 (2018). https://doi.org/10.1007/s00165-017-0433-3
20. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects Comput. **25**(5), 769–799 (2013). https://doi.org/10.1007/s00165-012-0225-8
21. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015). https://doi.org/10.1145/2796550
22. Dongol, B., Jagadeesan, R., Riely, J.: Transactions in relaxed memory architectures. Proc. ACM Program. Lang. **2**(POPL) (Dec 2018). https://doi.org/10.1145/3158106
23. Dongol, B., Le-Papin, J.: Checking opacity and durable opacity with FDR. In: Calinescu, R., Pasareanu, C.S. (eds.) SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 222–242. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_13
24. Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976). https://doi.org/10.1145/360363.360369
25. Fu, X., Kim, W., Shreepathi, A.P., Ismail, M., Wadkar, S., Lee, D., Min, C.: Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In: van Renesse, R., Zeldovich, N. (eds.) SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021. pp. 100–115. ACM (2021). https://doi.org/10.1145/3477132.3483556
26. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
27. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010). https://doi.org/10.2200/S00253ED1V01Y201009DCT004
28. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
29. Hoare, C.A.R.: Communicating sequential processes (reprint). Commun. ACM **26**(1), 100–106 (1983). https://doi.org/10.1145/357980.358021
30. Intel: Persistent memory development kit, `libpmemobj` library (2022), https://pmem.io/pmdk/libpmemobj/

31. Khyzha, A., Lahav, O.: Taming x86-TSO persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434328
32. Krishnan, R.M., Kim, J., Mathew, A., Fu, X., Demeri, A., Min, C., Kannan, S.: Durable transactional memory can scale with timestone. In: ASPLOS. p. 335–349. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378483
33. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
34. Lien, Y.E., Weinberger, P.J.: Consistency, concurrency and crash recovery. In: Lowenthal, E.I., Dale, N.B. (eds.) ACM SIGMOD International Conference on Management of Data. pp. 9–14. ACM (1978). https://doi.org/10.1145/509252.509258
35. Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., Ren, J.: Dudetm: Building durable transactions with decoupling for persistent memory. In: Chen, Y., Temam, O., Carter, J. (eds.) Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. pp. 329–343. ACM (2017). https://doi.org/10.1145/3037697.3037714
36. Liu, S., Seemakhupt, K., Wei, Y., Wenisch, T.F., Kolli, A., Khan, S.M.: Cross-failure bug detection in persistent memory programs. In: Larus, J.R., Ceze, L., Strauss, K. (eds.) ASPLOS. pp. 1187–1202. ACM (2020). https://doi.org/10.1145/3373376.3378452
37. Liu, S., Wei, Y., Zhao, J., Kolli, A., Khan, S.M.: PMTest: A fast and flexible testing framework for persistent memory programs. In: Bahar, I., Herlihy, M., Witchel, E., Lebeck, A.R. (eds.) ASPLOS. pp. 411–425. ACM (2019). https://doi.org/10.1145/3297858.3304015
38. Lowe, G.: Analysing lock-free linearizable datatypes using CSP. In: Gibson-Robinson, T., Hopcroft, P.J., Lazic, R. (eds.) Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10160, pp. 162–184. Springer (2017). https://doi.org/10.1007/978-3-319-51046-0_9
39. Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., Swanson, S.: Atomic in-place updates for non-volatile main memories with Kamino-Tx. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 499–512. EuroSys '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064215
40. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27
41. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (oct 1979). https://doi.org/10.1145/322154.322158
42. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. IEEE Micro **35**(3), 125–131 (2015). https://doi.org/10.1109/MM.2015.46
43. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 940–967. Springer International Publishing, Cham (2018)
44. Raad, A., Lahav, O., Vafeiadis, V.: On the semantics of snapshot isolation. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 1–23. Springer International Publishing, Cham (2019)

45. Raad, A., Lahav, O., Vafeiadis, V.: Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA), 151:1–151:28 (2020). https://doi.org/10.1145/3428219
46. Raad, A., Lahav, O., Wickerson, J., Balcer, P., Dongol, B.: Intel PMDK transactions: Specification, validation and concurrency (Extended version) (2023), https://doi.org/10.48550/arXiv.2312.13828
47. Raad, A., Lahav, O., Wickerson, J., Balcer, P., Dongol, B.: Intel PMDK transactions: Specification, validation and concurrency (Artifact) (2024). https://doi.org/10.6084/m9.figshare.24988173.v1
48. Raad, A., Maranget, L., Vafeiadis, V.: Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498683
49. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1–11:31 (2020). https://doi.org/10.1145/3371079
50. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1–135:27 (2019). https://doi.org/10.1145/3360561
51. Ramalhete, P., Correia, A., Felber, P.: Efficient algorithms for persistent transactional memory. In: Lee, J., Petrank, E. (eds.) PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021. pp. 1–15. ACM (2021). https://doi.org/10.1145/3437801.3441586
52. Ramalhete, P., Correia, A., Felber, P., Cohen, N.: Onefile: A wait-free persistent transactional memory. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019. pp. 151–163. IEEE (2019). https://doi.org/10.1109/DSN.2019.00028
53. Samsung Electronics: Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit (2022), https://bit.ly/samsung_flash_memory_summit
54. Scargall, S.: Programming Persistent Memory: A Comprehensive Guide for Developers. APress (2020). https://doi.org/10.1007/978-1-4842-4932-1_8
55. Upadhyayula, U.: Introduction to persistent memory allocator and transactions (2020), https://www.intel.com/content/www/us/en/developer/videos/introduction-to-persistent-memory-allocator-and-transactions.html
56. Zardoshti, P., Zhou, T., Liu, Y., Spear, M.F.: Optimizing persistent memory transactions. In: 28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019. pp. 219–231. IEEE (2019). https://doi.org/10.1109/PACT.2019.00025