

# Pronto: Easy and Fast Persistence for Volatile Data Structures

Amirsaman Memaripour\*  
University of California, San Diego  
amemarip@eng.ucsd.edu

Joseph Izraelevitz  
University of Colorado Boulder  
joseph.izraelevitz@colorado.edu

Steven Swanson  
University of California, San Diego  
swanson@cs.ucsd.edu

## Abstract

Non-Volatile Main Memories (NVMMs) promise an opportunity for fast, persistent data structures. However, building these data structures is hard because their data must be consistent in the wake of a failure. Existing methods for building persistent data structures require either in-depth code changes to an existing data structure using an NVMM-aware library or rewriting the data structure from scratch. Unfortunately, both of these methods are labor-intensive and error-prone.

*Pronto* is a new NVMM library that reduces the programming effort required to add persistence to volatile data structures using *asynchronous semantic logging (ASL)*. ASL is generic enough to allow programmers to add persistence to the existing volatile data structure (e.g., C++ Standard Template Library containers) with very little programming effort. Furthermore, ASL moves most durability code off the critical path, and our evaluation shows *Pronto* data structures outperform highly-optimized NVMM data structures written with other libraries by a large margin.

**CCS Concepts.** • **Hardware** → **Emerging technologies**; • **Software and its engineering** → **Software libraries and repositories**; • **Information systems** → **Data structures**; • **Computer systems organization** → **Processors and memory architectures**.

**Keywords.** Non-volatile Memory, Persistent Memory, Persistent Objects, Data Structures, Storage Systems, Snapshots, Asynchronous Logging, Semantic Logging

---

\*The author is now at MongoDB, Inc.

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). *ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland*

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7102-5/20/03.

<https://doi.org/10.1145/3373376.3378456>

## ACM Reference Format:

Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3373376.3378456>

## 1 Introduction

Emerging non-volatile main memory (NVMM) technologies such as 3D XPoint [14, 23] offer higher density than DRAM with comparable latency and bandwidth, allowing computer architects to attach them to processors via the memory bus. Programs can then use load and store instructions to access persistent data directly. Bypassing the storage stack and directly accessing NVMM is essential for unleashing the performance benefits that NVMMs offer [48]. However, this strategy requires careful reasoning to ensure a consistent-state in NVMM in the wake of a crash — data in the caches will not survive [28, 36].

NVMMs appear to be an exceptional opportunity for building fast, persistent, data structures, and researchers have approached this problem in two ways. NVMM failure-atomicity libraries (e.g., [11, 51]) allow programmers to delineate *failure-atomic* updates to persistent data - writes within the update become persistent all at once. By identifying failure-atomic code regions and persistent writes, programmers can adapt an existing data structure to NVMM using these libraries [6, 12]. Alternatively, researchers have built custom data structures from scratch for NVMM (e.g., [43, 50]). Unfortunately, both of these design options are labor-intensive, require detailed program knowledge, and are a fertile source of subtle errors [54]. Furthermore, these options effectively ignore the wide range of useful, volatile data structures currently available (e.g., the C++ Standard Template Library or the Java Collection data structures).

In this work, we propose *Pronto*, a library that reduces the programming effort required to add persistence to off-the-shelf, volatile data structures, preserving the original operation of the data structure and, for concurrent data structures, their concurrency scheme. Furthermore,

Pronto minimizes the performance overhead of this transformation by moving almost all durability-related code off the critical path.

Pronto transforms the volatile data structure by changing every operation on the original data structure into a failure-atomic operation. Adding Pronto to an existing volatile data structure is simple. For sequential data structures, adding Pronto requires only adding a thin wrapper class around the data structure’s API and using the Pronto allocator. For concurrent data structures, adding Pronto also requires one additional line of code per API method.

Pronto uses a novel mechanism called *Asynchronous Semantic Logging* (ASL) to convert each operation on a volatile data structure into a failure-atomic operation. ASL records the arguments and execution order of each update operation performed on the data structure rather than recording the details (e.g., pointer updates) of how the data structure changed. For instance, ASL would record the insertion of an item into a binary tree rather than recording how the tree’s internal structure changed. ASL is analogous to operation logging in database systems [42], but addresses the specific needs of logging for persistent, in-memory data structures. ASL uses background threads, which run in parallel with the program (foreground) threads, to create and persist the logs off of the critical path. To recover from program or system failures, Pronto plays back semantic logs for a structure to reconstruct its most recent consistent state (read-only operations need not be logged at all in Pronto). To limit the cost of replaying semantic logs, Pronto creates periodic, persistent copies of the data structure on NVMM (i.e., periodic snapshots).

In this paper, we describe the Pronto system and demonstrate that many common, non-persistent data structure implementations (e.g., RocksDB’s MemTable and containers from the GNU C++ Standard Template Library) are readily amenable to a Pronto adaptation with minimal programming effort, and, furthermore, these new Pronto adaptations perform better than other failure-atomic variants.

This paper makes the following contributions:

- It introduces ASL, a new software mechanism that reduces the programming effort and performance overhead of adding failure-atomicity to volatile data structures.
- It explores the design decisions and correctness constraints of ASL in the context of NVMMs.
- It provides an implementation for Pronto and evaluates its performance.
- It demonstrates how to use Pronto to convert both sequential and concurrent volatile data structures into persistent ones with only a few lines of code.

The rest of this paper is organized as follows. Section 2 provides some background on NVMMs and motivates Pronto. We discuss the design and implementation of Pronto in Sections 3 and 4, respectively. Section 5 presents the evaluation results and puts Pronto’s performance in perspective. We discuss related work in Section 6 and conclude in Section 7.

## 2 Background and Motivation

Non-volatile memories promise to fill the gap between volatile memory and disks (both hard and solid-state) by offering byte-addressability, DRAM-like latency and bandwidth, and persistence [4, 47]. NVMMs based on battery- or flash-backed DRAM [40] have been available for many years, and cheaper main memory modules based on 3D XPoint [24, 39] have entered the market recently [23, 41]. These emerging NVMMs offer higher density, higher latency, and lower bandwidth [8, 35] than DRAM-based devices. Thus, we anticipate hybrid memory systems with both DRAM and NVMM.

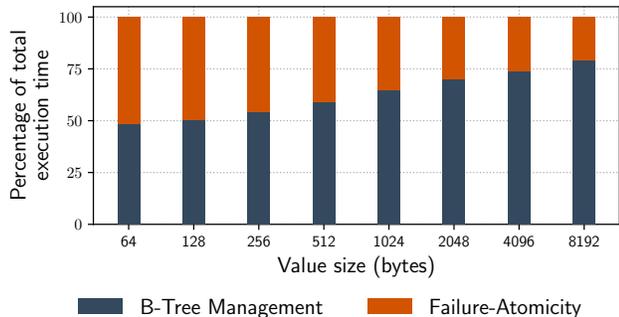
NVMMs are fast enough to sit on the processor’s memory bus [3], providing software with direct access to NVMM via load/store instructions. Since CPU caches are volatile, stores to non-volatile memory do not become durable until the cache writes back the affected data. Cache evictions are usually transparent to software, so programmers must use cache flush instructions to trigger write-backs and memory barriers to wait for the write-backs to complete [2, 22, 47].

Cache-flushes and memory barriers are necessary building blocks, but they do not suffice for providing the failure-atomicity that applications need to make use of NVMMs. Below, we describe the costs of providing failure-atomicity for programs with direct access to NVMM.

### 2.1 Programming Cost

Bypassing the filesystem to directly access NVMM via load/store instructions lets programmers fully exploit the performance benefits of NVMMs, but it introduces a series of challenges. Programs could lose part of an update to a persistent data structure during a system failure (e.g., power loss) because existing hardware does not support flushing multiple cache lines atomically: an ill-timed failure could cause permanent data inconsistency.

To avoid this issue, persistent data structures must be able to recover to a consistent state after a crash. NVMM transactional memory libraries [11, 36, 51] embody the most common approach to ensure failure-atomicity of updates to a persistent data structure: fine-grain logging of how the data structure changes. Unfortunately, annotating existing data structures with these libraries



**Figure 1.** Latency breakdown of inserts to PMEMKV

is labor-intensive and error-prone as programmers are required both to annotate every persistent data update and reason about failure-atomic update boundaries. As an example, we had to rewrite almost all of a volatile B+Tree to make it persistent using Intel’s popular Persistent Memory Development Kit (PMDK) library [25]. For more complicated data structures in use today (e.g., those in the C++ Standard Template Library), adding all these annotations without error would be an extremely invasive change to a code base that is already very complex and highly-optimized for volatile operation. Indeed, the difficulty of correctly adding annotations has spawned research into new debugging tools for finding these errors [21, 31, 34, 54].

## 2.2 Performance Cost

The cost of enforcing failure-atomic updates for NVMM data structures is large. Logging for failure-atomicity libraries adds overhead in the form of stores to transaction metadata, additional cache-flushes, and memory barriers [10, 56]. Moreover, the cost of fine-grain logging scales with the complexity of persistent data structures. Logging also limits the processor’s ability to reorder instructions [45], further hurting performance.

To explore this cost, we measured it in PMEMKV [26], a persistent key-value store that uses a B-Tree and stores its last level in NVMM [53]. PMEMKV uses the transaction facility in PMDK [25] to transactionally update the B-Tree.

We instrumented PMDK and PMEMKV to gather detailed latency numbers for inserting one million key-value pairs to PMEMKV using traces from YCSB [13]. Figure 1 reports the relative latency of managing the B-Tree data structure and ensuring its failure-atomicity (e.g., logging, persistent allocation, and transaction management) for value sizes ranging from 64 to 8192 bytes. Failure-atomicity increases the latency of insert operations by 26% to 106%.

Conventional NVMM transaction libraries put all the overhead of ensuring failure-atomicity (e.g., logging,

cache-flushes, and barriers) on the critical path, so applications bear the full cost. Pronto’s goal is to hide this overhead by moving it off the critical path.

Next, we modified PMEMKV to disable transaction management and logging. This modified version does not ensure the durability and consistency of updates to the NVMM-resident data, but still adopts PMDK’s persistent memory allocator for managing the last level of the B-Tree. Comparing the throughput of the modified and original versions of PMEMKV lets us estimate the performance boost that we can achieve by moving logging and transaction management off the critical path. We observe that the modified version (with no logging and transaction management) runs twice as fast.

## 3 Design

Pronto adds persistence to volatile data structures with minimal code changes and moves the cost of durability off the critical execution path. It accomplishes this by creating asynchronous semantic logs (ASLs) that allow for the reconstruction of the latest consistent state of the data structures during recovery from a failure. The semantic logs record every operation invoked on the object along with the operation’s arguments. This logging occurs *asynchronously* and in parallel with the actual operation.

In terms of the programming cost, ASLs are useful since they avoid the need to log fine-grained changes to the underlying data structure. With semantic logging, we only need to log the method call and its arguments — replaying operations after a failure is sufficient to recover the data structure’s state. Code changes, as a consequence, are minimal — for sequential data structures, we only need to intercept the public methods of the data structure and ensure that it uses Pronto’s allocator to allocate its internal structures. Adding ASLs to concurrent data structures is nearly as easy: it requires adding one additional line of code to each public method.

Our ASLs also reduce the performance cost of persistence by logging asynchronously, especially for slower NVMMs. By decoupling log creation from operation execution and performing logging in parallel, ASL can drastically reduce the performance cost of persistence. In fact, if the logging is quick enough, Pronto can almost completely hide the overhead of logging by moving it off the critical path.

Pronto is broadly applicable to most data structures. The only restriction is that the structures must meet two criteria that are common to most data structures. First, the data structure and its interface must be properly *encapsulated* so that modifications only occur through public methods and *deterministic* so that the externally-visible effect of those methods is only a function of the

current state of the data structure and the arguments to the method. In effect, this means that the methods cannot read or write global variables. Second, if the data structure is thread-safe (i.e., supports concurrent accesses), it must be *linearizable* [19, 38].

An update to the data structure is linearizable if the data structure’s synchronization mechanisms (e.g., locks) ensure that the effect of multiple (potentially parallel) updates is the same as those updates being applied one at a time in some order [19]. Linearizability is the common correctness condition for concurrent data structures, and most practical data structures meet this condition (e.g. [17, 32, 49]). For any linearizable data structure that uses locks to order updates that do not commute, Pronto provides failure-atomicity with no loss of concurrency.

These requirements are not onerous in practice, since they closely correspond to common data structure design practices. Most container libraries (e.g., the C++ STL) and many custom data structures (e.g., the core data structures of RocksDB [16] and Memcached [18]) meet them.

This section describes the design of Pronto. We begin with a description of the Pronto system and runtime. Next, we describe Pronto’s programming interface and elaborate on the durability and concurrency semantics that Pronto offers. Finally, we give examples of using Pronto for both sequential and concurrent data structures.

### 3.1 Pronto System Overview

The Pronto runtime maintains three entities for each persistent data structure it manages. An *asynchronous semantic log*, a volatile *online image* of the data structure in volatile memory, and a persistent *snapshot* of the data structure. This subsection describes Pronto’s runtime in terms of its ASL, memory management, and snapshot mechanisms.

**3.1.1 Asynchronous Semantic Logging.** Pronto’s semantic logs record the high-level updates that the data structure undergoes rather than the fine-grain changes to the memory that holds it. For example, Pronto only creates a single log record for inserting a new key-value pair to a B-Tree, unlike undo-logging that requires recording the fine-grain changes to the B-Tree’s structure that happen as part of the insert. Since recording the high-level operations is usually fast, ASL is generally more efficient than normal write-ahead logging.

For clarity, we describe ASL in terms of method invocations (or “updates,” read-only operations need not be logged) on container-style objects (e.g., linked lists, hash maps, and vectors), but ASL will work for any deterministic, linearizable (or sequential) data structure

with a well-defined set of operations that Pronto’s ASL can record.

For every operation that modifies the data structure, Pronto creates a *semantic log entry*, a persistent record that records the method invoked (e.g., an insert) and a copy of its arguments.

Besides an ASL and a persistent snapshot, Pronto maintains a volatile online image for each data structure. The online image reflects the current state of the data structure. In addition to logging operations, Pronto applies each operation to the volatile version and read-only operations run against it.

After a crash and upon restart, Pronto can recreate the volatile online image (i.e., recover the last consistent state of the data structure) by replaying the ASL. A snapshot mechanism described below keeps the cost of recovering the volatile online image manageable.

The key optimization that Pronto makes is to perform logging in an *ASL thread* that runs in parallel with the foreground update to the online image. If applying an update to the online image is slower than logging its arguments, Pronto can entirely hide the ASL’s latency.

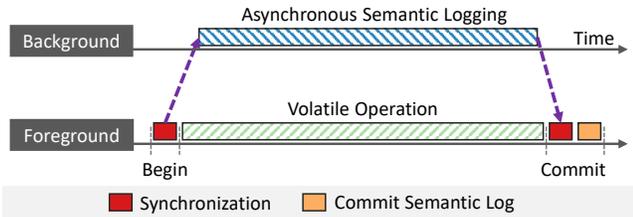
Under ASL, an operation is not complete until both the update to the volatile online image is finished and the semantic log entry is persistent. To enforce this requirement, the foreground thread must wait for the ASL thread to finish logging before any of the update’s effect becomes visible to other threads. In practice, this means synchronizing with the ASL thread before releasing any lock that protects the operation’s effects (changes) from being visible to other concurrent operations. This guarantees that the commit order of ASLs agrees with the execution order of updates to the data structure that do not commute (e.g.,  $\text{insert}(K_1, V_1)$  and  $\text{erase}(K_1)$ ).

Figure 2 illustrates the parallel execution of the foreground thread (bottom) and ASL thread (top). ASL operations are blue, DRAM updates are green, and synchronization is red. *Begin* marks the beginning of both logging and update execution. *Commit* marks completion of the operation. The small orange box in the foreground thread is the commit point for the ASL log entry when the entry becomes persistent.

Figure 3 compares ASL with undo-logging and redo-logging [36]. ASL allows executing the *Logging* code in parallel with the *Operation* and decreases the execution complexity of memory barriers and cache-line flushes in the critical path, thereby reducing the total overhead of adding persistence to volatile data structures.

### 3.1.2 Memory Management and Addressing.

Pronto provides a volatile memory allocator that manages a contiguous region of memory to hold the online, volatile image. Data structures must use the allocator for any internal objects (e.g., nodes in a linked list) and



**Figure 2.** Communications between the foreground and background execution paths to guarantee every committed semantic log represents a completed update operation.

applications must use the allocator for objects they pass to data structure methods via a pointer. This requirement ensures that the data structure and all memory reachable from it are fully contained within the memory region the allocator manages.

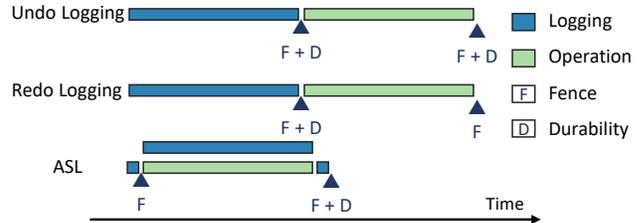
The online image of a data structure uses native pointers for addressing, so it is not relocatable (i.e., it must always reside at the same virtual address). This is not a fundamental limitation of Pronto or ASL, but it is necessary to support the easy conversion of volatile data structures into persistent data structures without compiler support. Previous work has shown how to ensure relocatability with a compiler [37]. Those techniques would apply to Pronto. We describe the allocator in detail in Section 4.2.

Pronto also manages NVMM space for semantic logs and snapshots. It allocates space by mapping NVMM files into the program’s address space. ASL uses the mapped NVMM space as a circular buffer and writes over old semantic log entries that precede the latest snapshot. Section 4.1 provides additional details.

**3.1.3 Snapshots.** Pronto provides a snapshot mechanism that works closely with its volatile memory allocator to take periodic snapshots of online images. Snapshots, which are durably stored on NVMM, reduce the ASL storage requirements and improve recovery time since Pronto only needs to store ASL entries since the last snapshot and replay those entries after a crash.

Snapshots contain a persistent copy of the (volatile) memory pages used by the volatile online images of the data structures along with a description of currently allocated memory (provided by Pronto’s allocator). Pronto always keeps the latest snapshot on NVMM to ensure fast recovery.

The application can change the frequency of snapshots to trade-off between snapshot overhead and recovery time. We describe the mechanics of taking a snapshot in Section 4.3 and measure its performance impact in Section 5.7.



**Figure 3.** Comparing the execution path of ASL against undo-logging and redo-logging. The operation represents a deterministic update, such as inserting a new node to a tree.

## 3.2 Using Pronto

Pronto offers a simple C++ interface for creating persistent data structures with ASL support. The interface provides access to Pronto’s volatile memory allocator, a mechanism to specify the boundaries of operations that the ASL will record, and a directory that allows accessing persistent data structures across restarts. Table 1 summarizes the interface.

Programmers can use Pronto to add persistence to both sequential (single-threaded) and concurrent (thread-safe) volatile data structures. This section provides an example of using Pronto for each case and elaborates on the requirements for using Pronto with concurrent data structures.

### 3.2.1 Adding Pronto to Sequential Data Structures.

Adding Pronto to a volatile single-threaded data structure is straight-forward. The programmer adds Pronto by creating a wrapper object for the volatile data structure, and the wrapper object inherits from `PersistentObject`. Extending the `PersistentObject` superclass provides a naming mechanism to enable programmers to access instances of the class across restarts using a unique name. Any instance of this new class is a persistent object, where the latest consistent state of its internal data structure survives failures and each public method executes as a failure-atomic operation.

The wrapper object contains an instance of the original data structure (i.e., the online image) and wrapper methods for every function in the data structure’s API. For any method that updates the wrapped data structure, the programmer inserts a special `op_begin()` at the top of the corresponding wrapper method and `op_commit()` at the end. The `op_begin()` method triggers semantic log entry creation and takes a copy of the input arguments, while the `op_commit()` method commits the operation. Note that Pronto only requires instrumenting public update (e.g., non-`const`) methods, while existing NVMM libraries (e.g., PMDK [25]) require tracking all

<code>PersistentObject(name)</code>	Every persistent object must inherit from this class. Pronto identifies objects by their unique name (provided to the constructor) and maintains a persistent directory for mapping names to references to objects.
<code>get_object&lt;T&gt;(name)</code>	Uses the persistent directory to return a reference to the persistent object of type <T> identified by <code>name</code> .
<code>op_begin(args)</code>	Marks the beginning of a failure-atomic operation, which accepts <code>args</code> as input, and initiates ASL.
<code>op_commit()</code>	Waits for the operation’s ASL to complete and then marks the semantic log entry as committed.
<code>palloc(size)</code> <code>prealloc(ptr, size)</code> <code>pfree(ptr)</code>	Programmers must replace <code>malloc()</code> , <code>realloc()</code> and <code>free()</code> with <code>palloc()</code> , <code>prealloc()</code> and <code>pfree()</code> for managing memory for their data structures (e.g., using GCC’s <code>-wrap</code> flag) to allow Pronto create periodic asynchronous snapshots.

**Table 1.** Pronto’s programming interface

```

template <class T>
class PVector : PersistentObject {
    // Alloc conforms with STL allocator
    // Alloc.allocate() calls palloc()
    // Alloc.deallocate() calls pfree()
    vector< T, Alloc<T> > *vVector;
public:
    PVector(string name):PersistentObject(name) {
        // alloc is an instance of Alloc<T>
        // *new* uses palloc() for allocation
        vVector = new vector< T, Alloc<T> >(alloc);
    }
    void push_back(const T& value) {
        op_begin(value);
        vVector->push_back(value);
        op_commit();
    }
    void pop_back() {
        op_begin();
        vVector->pop_back();
        op_commit();
    }
    size_t size() const {
        // no logging needed for read-only ops
        return vVector->size();
    }
};

```

**Figure 4.** Creating a template persistent vector using the STL’s vector container and Pronto.

writes to NVMM. Pronto uses a simple source preprocessor to provide every `op_begin()` with a pointer to the public method that calls into it, which enables mapping semantic logs to their matching public methods during recovery. This preprocessor also generates code to convert each semantic log entry to a corresponding method call and automate replaying semantic logs at recovery. Pronto assumes that the implementation of the data structure does not change before recovery.

Finally, the programmer must use Pronto’s memory allocator to manage memory for the wrapped data structure.

Figure 4 is an example of using Pronto’s APIs from Table 1 to create a persistent version of the `vector` container from the GNU C++ Standard Template Library (STL). We create a wrapper class (`PVector`) for the `std::vector` that extends `PersistentObject`. Since STL containers support user-specified allocators, we pass a reference to Pronto’s allocator to the constructor of the `std::vector`. Update methods of the STL vector are wrapped and surrounded by `op_begin()` and `op_commit()`. For the sake of simplicity, we only illustrate the implementation of the constructor, `push_back()` and `pop_back()` methods.

**3.2.2 Adding Pronto to Concurrent Data Structures.** Pronto supports a wide class of concurrent data structures that synchronize internally using locks. So long as they meet the standard correctness condition of *linearizability*, Pronto can make them resilient to power outages with simple code changes. In a linearizable (concurrent) data structure, each method appears to occur at some atomic instant in time between its invocation and return; putting the operations in this order gives us a *linearization order*, and the concurrent data structure must behave exactly like a sequential data structure executing the operations in this order [19, 38].

Converting a thread-safe data structure in Pronto follows the exact same requirements as a sequential data structure, save for the call to `op_commit()`, which, instead of being called in the wrapper object, is called within the wrapped data structure at a programmer identified point. For proper integration with Pronto, the order in which operations call `op_commit()` must be a valid linearization order. Put more simply, if two data structure operations cannot (semantically) commute (e.g., performing `insert(k1, v1)` and `erase(k1)` against a hash-map), then their calls to `op_commit()` must occur in program order.

In practice, this requirement can be trivially met by ensuring that the lock that protects the operation’s data structure modifications also protects the call to

```

template <class T>
class HashMap : PersistentObject {
    const unsigned Buckets = 32;
    unordered_map<T, T, hash<T>, equal_to<T>,
        Alloc<T>> *vMaps[Buckets];
    mutex locks[Buckets];
public:
    HashMap(string name):PersistentObject(name) {
        // initialize vMaps and per-bucket locks
    }
    void insert(const T& key, const T& value) {
        op_begin(key, value);
        unsigned b = hash<T>{}(key) % Buckets;
        locks[b].lock();
        vMaps[b]->insert(make_pair(key, value));
        op_commit();
        locks[b].unlock();
    }
};

```

**Figure 5.** Creating a persistent, concurrent hash-map using Pronto and C++ STL’s `unordered_map` container.

`op_commit()`. As a consequence, programmers can preserve their existing isolation for operations and avoid disruptive changes to the program to use a new synchronization interface.

If Pronto is properly integrated into a linearizable data structure according to the above requirements, it generates a *durably linearizable* data structure [28], in which the data structure’s operations not only appear to atomically occur in between their invocation and response, but also become persistent at the same instant. For blocking data structures that use locks to enforce linearizability, Pronto provides failure-atomicity with no loss of concurrency.

Figure 5 shows an example of using Pronto with a thread-safe, concurrent hash-map. Since STL containers are not thread-safe, we use locks to serialize accesses to each bucket of the hash-map. By committing semantic logs before releasing the per-bucket locks, we force semantic logs to commit in the order that the program performs non-commutable operations (e.g., `insert(K1, V1)` and `insert(K1, V2)`), but in either order for operations that commute (e.g., `insert(K1, V1)` and `insert(K2, V2)` when  $K_1 \neq K_2$ ).

**3.2.3 Requirements for Concurrent Data Structures.** The following equation formalizes the requirement for committing ASL entries for concurrent updates to a linearizable data structure.  $H_S$  and  $H_P$  denote sequential and parallel execution histories, respectively, and  $H_S \approx H_P$  denotes that  $H_S$  is a valid linearization order of  $H_P$ .  $op_1$  and  $op_2$  represent two atomic operations that occur in both  $H_S$  and  $H_P$ . The relations  $<_{H_S}$  and  $<_{commit}$  refer to the  $H_S$  order and the Pronto

commit order respectively.

$$if \ \forall_{H_S \approx H_P} \ op_1 <_{H_S} op_2 \ then \ op_1 <_{commit} op_2 \quad (1)$$

This requirement allows Pronto to reconstruct persistent objects after failures by replaying semantic logs sequentially according to their commit order – the commit order of semantic logs represents a valid sequential execution order of their corresponding failure-atomic operations.

## 4 Implementation

This section elaborates on the implementation of Pronto and revisits the most interesting technical challenges we addressed in building it by answering the following questions:

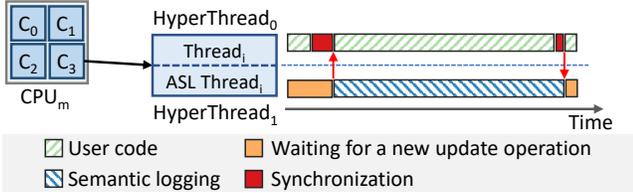
- How to minimize the programming effort of building persistent objects from volatile ones?
- How to implement ASL with minimum overhead on the critical execution path?
- How to identify modified memory pages to efficiently create periodic, asynchronous snapshots?
- How to store asynchronous, consistent snapshots of off-the-shelf volatile data structures with minor changes to the source code?
- How to use semantic logs and snapshots to reconstruct persistent objects after failures?

Pronto comprises a user-level C++ library and a simple source preprocessor. Below we describe how the library manages logs, allocates memory, takes snapshots, and recovers from failures. Then we describe the preprocessor.

### 4.1 Asynchronous Semantic Logging

To reduce the overhead of semantic logging on the critical path, Pronto creates a dedicated background ASL thread for every foreground thread. Foreground threads notify ASL threads upon starting a new failure-atomic operation by calling `op_begin()` and sync up with them to ensure the persistence of semantic logs before committing the log entry.

Pronto uses `pthread_create()` to create an ASL thread for every foreground thread, evenly distributes foreground threads over available physical cores, and co-locates foreground threads with their ASL threads. Sharing physical cores (i.e., running as hyperthreads) enable foreground and ASL threads to share L1 cache-lines and synchronize at low cost. Figure 6 shows the assignment of foreground and ASL threads to CPU cores and demonstrates the synchronization points between the two threads.



**Figure 6.** Pronto evenly distributes user threads over physical CPU cores and co-locates each one with its ASL thread.

Pronto’s implementation aims to minimize the overhead of ASL on the critical path and trades CPU and recovery time for faster execution of update operations. However, multiple user threads can share a single ASL thread for programs that are read-dominated or less sensitive to ASL overhead.

Pronto stores semantic logs in NVMM-resident files and creates a separate file for each persistent object. These files comprise a header and a body. The header includes the commit number of the last committed semantic log and relative pointers to the head and tail of the file’s body. Having a separate file for each object reduces the contention on the log’s header. The body stores semantic logs in a circular buffer.

Semantic log entries contain a pointer to the method they must replay during recovery, as well as a shallow copy of its input arguments. Making a copy is necessary. Otherwise, the application might change a value after the log entry is created, leading to a different result during recovery.

Pronto uses `DAX mmap()` to directly map the file to the program’s virtual address space, bypass the storage stack, and access the NVMM pages via load/store [33]. ASL threads use non-temporal store instructions followed by memory barriers to avoid cache pollution while appending semantic logs to the mapped pages, which also improves the performance of creating large semantic logs. Support for `DAX mmap()` is currently available through the ext-4, XFS, and NOVA [46, 52] file systems.

## 4.2 Memory Allocator

Pronto uses a custom memory allocator for the volatile online image of persistent objects to facilitate creating asynchronous snapshots. The allocator serves allocations from a contiguous volatile memory pool, which could reside on NVMM if the DRAM capacity is not sufficient, and maintains a bitmap for the pool to differentiate between used and unused regions. The bitmap granularity is 4 KB.

Pronto serves allocations by regions from an extensible volatile memory pool, which can expand by mapping huge-pages into the program’s address space. Pronto

uses huge-pages to reduce the number of page-table entries and thus, the overhead of creating asynchronous snapshots. The allocator always maps the volatile memory pool at the same virtual address to keep pointers valid throughout restarts and allow recovering objects from snapshots. Pronto maintains per-object allocators that serve allocation and free operations through per-core free-lists to reduce contention, allocation latency, and synchronization overhead. Free-lists sort memory regions based on their size and assign them into buckets to reduce lookup time. Each bucket holds a pointer to a doubly-linked list of unused memory regions [5, 15].

## 4.3 Periodic Snapshots

To create a persistent snapshot, Pronto must freeze the execution at a point of time where all persistent objects are in a consistent state (i.e., before or after running an update operation), and then copy the entire online image to NVMM. The process of creating snapshots comprises a synchronous and an asynchronous phase.

During the synchronous phase, Pronto freezes persistent objects in a consistent state by blocking new update operations and awaiting completion of those that are yet to be committed. It then streams the state of allocation tables, including the bitmap and free-lists, to NVMM and simultaneously marks the allocated volatile pages as read-only.

Next, Pronto unblocks new update operations and starts the asynchronous phase, where it saves the read-only volatile pages to NVMM. Pronto uses multiple threads to expedite the copying. The threads examine the allocated 2 MB volatile pages, identify its used 4 KB regions using the bitmap, stream the used regions to NVMM, and make each page writable as soon as the NVMM copy is durable. An update operation that attempts to write to a read-only volatile page will trigger a page-fault handler, which takes over copying the target page to NVMM before marking it writable and returning to the operation that caused the page-fault.

Pronto creates full snapshots for the sake of simplicity. To support incremental snapshots, it can keep volatile pages read-only until modified by an update operation, and only include writable (i.e., modified) pages in new snapshots.

For every persistent object, Pronto also records the identifier of its last committed operation and the tail offset of its semantic log at the time of creating the snapshot. It then recycles any log entry that precedes this tail offset for creating new semantic logs.

## 4.4 Recovery Management

After a crash, Pronto uses a combination of ASL and durable snapshots to restore persistent objects to their

state before the failure. It uses the most recent snapshot to restore the latest durable state of its memory pool.

Next, it replays semantic logs against their corresponding persistent objects in commit order. For every persistent object, Pronto only replays semantic log entries recorded after the latest snapshot. Once it replays all log entries, it passes control to user code.

Pronto uses multiple threads to recover persistent objects and assigns a subset of the persistent objects to each recovery thread. Pronto uses a valid linearization order, which is dictated by the commit order of update operations, to replay the semantic logs. Since the original execution of the program is deadlock-free and Pronto replays update operations in a valid linearization order, Pronto’s recovery is deadlock-free.

## 4.5 Preprocessor

Pronto’s preprocessor reduces the programming effort of using Pronto by automatically generating the code for translating method calls into matching semantic logs during execution and decoding semantic logs to matching method calls during recovery.

For every public method that updates the data structure, the preprocessor passes a pointer to the method as an extra argument to `op_begin()`. It then extends these data structures with a new function that creates semantic logs. These functions, which ASL uses at runtime, store all the input arguments provided to `op_begin()` as well as the pointer to the caller public method in a semantic log entry.

The preprocessor creates a member function for each persistent data structure to enable replaying semantic logs during recovery. This function translates semantic log entries of its data structure to the corresponding public method calls.

The preprocessor also overloads the `new` operator of persistent data structures (i.e., every class that extends `PersistentObject`) to allocate all memory the data structure uses with Pronto’s allocator.

## 5 Evaluation

In this section, we evaluate Pronto’s performance to provide answers to the following questions:

- What is the performance overhead of using Pronto to add persistence to volatile data structures?
- Can programmers use Pronto to build persistent data structures that outperform highly-optimized NVMM data structures?
- What is the performance benefit of using Pronto as the failure-atomicity mechanism for existing applications?

- How much is the speedup of replacing existing NVMM libraries with Pronto for persistent data structures?
- How much is the storage overhead of Pronto’s ASL and periodic snapshots?
- When is ASL most effective at hiding the persistence cost?
- What is the cost of creating asynchronous snapshots for data structures with either sequential or random memory access patterns?
- How does the size of data structures, the frequency of snapshots, and the number of threads impact the recovery time?

### 5.1 Testbed Setup

The evaluation platform has two Intel Cascade Lake-SP (engineering sample) processors with 12 physical cores and hyper-threading enabled that run at 2.2 GHz. The platform has 192 GB of DRAM and 1.5 TB ( $6 \times 256$  GB) of NVMM (Intel Optane DC 2666 MHz QS [23, 29]) on each socket. All benchmarks run on one processor, avoiding NUMA-related overheads in accessing NVMM. We use `ext4` to provide direct-access (DAX) to NVMM pages [33].

### 5.2 Persistence for Volatile Data Structures

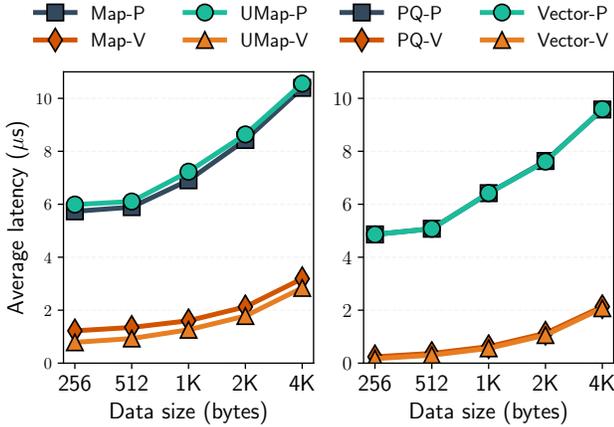
We measure the overhead of using Pronto to add persistence to both sequential (single-threaded) and concurrent (thread-safe) volatile data structures.

#### 5.2.1 Overhead for Sequential Data Structures.

Our first experiment uses four containers from the GNU C++ Standard Template Library (STL) to evaluate the overhead of integrating Pronto with volatile data structures. These containers are:

- *map*: a sorted map that stores key-value pairs in a red-black tree.
- *unordered\_map*: an unordered hash-table that stores key-value pairs.
- *vector*: a resizable array data structure.
- *priority\_queue*: an adapter for the vector container that creates a max-heap from the inserted elements.

Since STL containers provide deterministic update operations and support using user-defined allocators, we create persistent versions of each container by creating a wrapper class that extends Pronto and wraps calls to the container’s public methods, similar to the wrapper for STL’s `vector` in Figure 4. To measure the performance of `vector` and `priority_queue`, we insert 5 million elements to both versions of each container. We use traces from



**Figure 7.** Measuring the overhead of using Pronto to add failure-atomicity to the volatile benchmarks. The horizontal axis is the data size of insert operations (excluding the key for Map and Unordered Map benchmarks) in bytes and the vertical axis is the average latency in microseconds. V and P stand for Volatile and Persistent, respectively. UMap and PQ represent the Unordered Map and Priority Queue data structures, respectively.

YCSB [13] to evaluate map and unordered\_map containers. The traces comprise 5 million insert operations with 32-byte keys.

We measure the average latency of both volatile and persistent versions of the benchmarks to quantify the performance overhead of Pronto. Figure 7 shows how the average latency for the benchmarks change as we increase the size of data inserted into the STL containers. We create a snapshot for persistent benchmarks at least once every 15 seconds.

For small operations, such as inserting small values into the vector, Pronto imposes more overhead (up to 28 $\times$ ) as the synchronization between the user and the ASL thread is relatively more expensive, and the latency of the operation is significantly smaller than persisting the semantic log. The synchronization overhead is minimal for programs with more complex logic like the priority queue and the map. Moreover, ASL threads use non-temporal stores followed by memory fences to create semantic logs (i.e., copying pointers to operations and their input data to NVMM), which perform poorly for small writes and increase the relative overhead of ASL for small operations.

Therefore, the overhead of Pronto is significant for small operations (e.g., 28 $\times$  for inserting 256-byte values into STL’s vector) and lowest for programs with compute-intensive operations and large memory footprints (e.g., 3.2 $\times$  for adding key-value pairs with 4 KB values to STL’s Map).

**5.2.2 Concurrent Data Structures.** Our next experiment uses the persistent hash-map implementation from Figure 5, which adds locking to 32 instances of STL’s unordered\_map container to support concurrent operations, and compare its throughput against the volatile version of the hash-map to measure Pronto’s scalability. We use jemalloc [15] as the allocator for the volatile hash-map since thread-safe malloc uses an internal lock and serializes concurrent accesses. For the persistent hash-map, we create a snapshot at least once every 10 seconds. Figure 8 shows the average throughput for inserting 5 million key-value pairs with 1 KB values to the hash-map implementations – as we increase the number of threads (from 1 to 8), both volatile and Pronto versions of the concurrent hash-map show similar scalability.

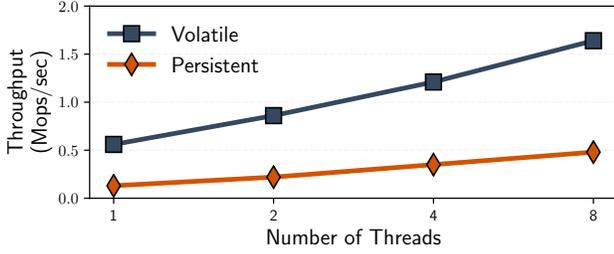
### 5.3 NVMM-Optimized Data Structures

Our next experiment compares the performance of Pronto against NVMM-optimized data structures. We use the YCSB traces from Section 5.2 to compare the performance of the failure-atomic versions of STL’s map and unordered\_map containers against PMEMKV [26], which is an NVMM-optimized key-value store. We configure PMEMKV v0.3x to use its *kvtree2* storage engine, which adopts undo-logging to implement failure-atomic updates. The persistent map and unordered\_map containers outperform PMEMKV and provide up to 3.83 $\times$  and 3.77 $\times$  lower latency, respectively. Figure 9 summarizes the results and reports the average latency of inserting key-value pairs in microseconds.

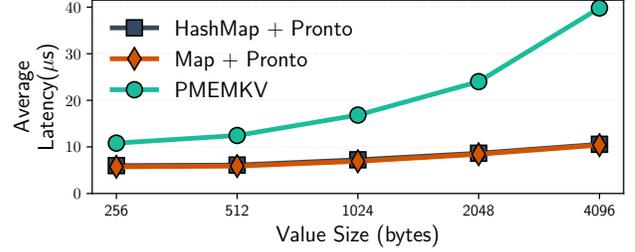
### 5.4 Optimizing Persistent Data Structures

To demonstrate the performance benefit of using Pronto to optimize existing persistent data structures, we modify RocksDB 5.17 [16], a persistent key-value store library, and replace its default failure-atomicity mechanism (redo-logging) with ASL. Using write-dominant (YCSB A with 50% reads and 50% writes) and read-dominant (YCSB B with 95% reads and 5% writes) traces from YCSB, we compare the performance of the modified version of RocksDB against its original version with *synchronous* and *asynchronous* writes. A synchronous-write does not return unless its redo-log is durable, while an asynchronous-write immediately returns once its redo-log reaches the filesystem’s page-cache. As a consequence, a failure may cause the last few asynchronous writes to be lost.

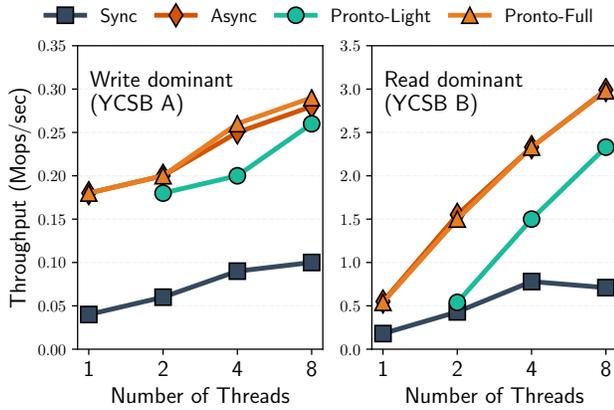
We warm-up the key-value stores by inserting 5 million key-value pairs (i.e., YCSB load phase) and then perform 5 million put/get operations based on the workload characteristics (YCSB A and YCSB B). We use 4 KB values for these experiments and configure Pronto in two modes: *Pronto-Full* that uses a dedicated ASL thread



**Figure 8.** Measuring the throughput of the volatile and persistent (Pronto) versions for the concurrent hash-map. Numbers show throughput in millions of 1 KB inserts per second.



**Figure 9.** Comparing the performance of PMEMKV against the persistent versions of STL’s map (Map + Pronto) and unordered\_map (HashMap + Pronto) containers.



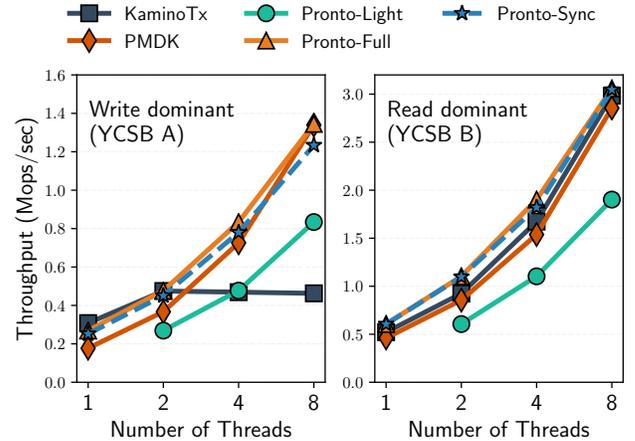
**Figure 10.** Comparing the performance of the NVMM-optimized version of RocksDB (i.e., Pronto-Full and Pronto-Light) against its original version with synchronous and asynchronous writes using read-dominant and write-dominant workloads from YCSB.

for each benchmark thread, and *Pronto-Light* that uses half the number of threads of the former version.

Figure 10 shows that both versions with Pronto (i.e., *Pronto-Light* and *Pronto-Full*) outperform RocksDB with synchronous writes with a wide margin. Furthermore, *Pronto-Full* matches the performance of asynchronous writes for both read-dominant and write-dominant workloads, despite giving stronger guarantees on failure.

### 5.5 Comparing ASL against Undo-Logging

We use the concurrent, persistent B+Tree implementation from Kamino-Tx [36] to compare the performance of Kamino-Tx and PMDK 1.5 [25], existing NVMM libraries that accomplish failure-atomic updates using undo-logging, against Pronto. We create a new version of the B+Tree by removing its failure-atomicity code and wrapping it by a Pronto object, thereby making it failure-atomic through Pronto. For the Pronto version of the B+Tree, we create a snapshot after performing 50%

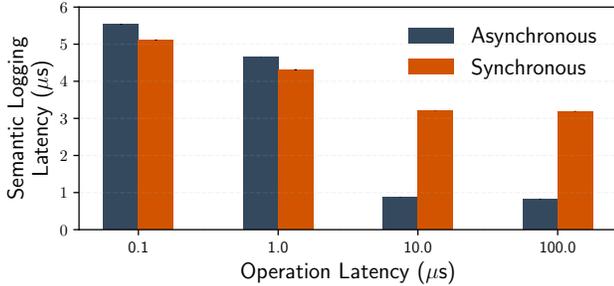


**Figure 11.** Comparing the performance of Pronto against PMDK [25], and KaminoTx using the B+Tree benchmark from KaminoTx [36]. We report the throughput for (read and write) operations with 1 KB values.

of the insert operations (around once every 5 seconds). The Kamino-Tx and PMDK versions only persist the last level of the B+Tree and reconstruct the internal nodes after restarts.

Figure 11 shows the average throughput of running the YCSB workloads from Section 5.4 against the Kamino-Tx, PMDK, and Pronto versions of the B+Tree. We use three Pronto modes for these experiments: *Pronto-Full* that creates an ASL thread for every benchmark thread, *Pronto-Light* that uses half the number of threads of *Pronto-Full*, and *Pronto-Sync* that creates semantic-logs synchronously.

In comparison to PMDK and Kamino-Tx, *Pronto-Full* provides higher performance for the write-dominant workload (YCSB A). Kamino-Tx does not scale when running YCSB A as it uses a single persist thread. For the write-dominant workload, *Pronto-Sync* closely matches the performance of PMDK and outperforms Kamino-Tx. *Pronto-Full* and *Pronto-Sync* offer slightly



**Figure 12.** Comparing the latency of creating asynchronous to synchronous semantic logs on the critical path. The latency of volatile operations varies from 100 ns to 100  $\mu\text{s}$ , and the size of semantic log entries is 1 KB.

higher throughput for YCSB B (the read-dominant workload).

### 5.6 Sensitivity Analysis

We use a microbenchmark to measure the sensitivity of ASL to the latency of the volatile operations. We vary the operation latency from 100 ns to 100  $\mu\text{s}$  and report the overhead of creating 1 KB asynchronous semantic logs on the critical path.

Figure 12 shows the results and compares the cost of ASL to synchronous semantic logging, where Pronto creates the 1 KB semantic logs on the critical path and before performing the volatile operations. We report average latencies of 5 million operations across five runs, and show the standard deviation atop each bar (the small, horizontal bars in black).

The experiments show that for sub-microsecond operations, ASL falls short in hiding the persistence overhead as the operation latency is a fraction of the cost of ASL. For other operations, ASL moves the entire cost of creating semantic logs to the background and only exposes a small fraction of semantic logging (i.e., committing entries and transferring the operation arguments to the ASL thread) to the critical path.

Note that the cost of persisting semantic logs and committing them decreases as we increase the latency of the volatile operations (i.e., the gap between consecutive writes to the same NVMM address). This behavior is due to how Intel Optane DC persistent memory handles back-to-back writes to the same address [29].

### 5.7 Overhead of Snapshots

Snapshot performance is critical for Pronto because it dictates the frequency at which programmers can create snapshots, and thus the trade-off between execution and recovery time. Here we use two micro-benchmarks to quantify the impact of Pronto’s snapshot mechanism on the average latency and the total execution time of programs.

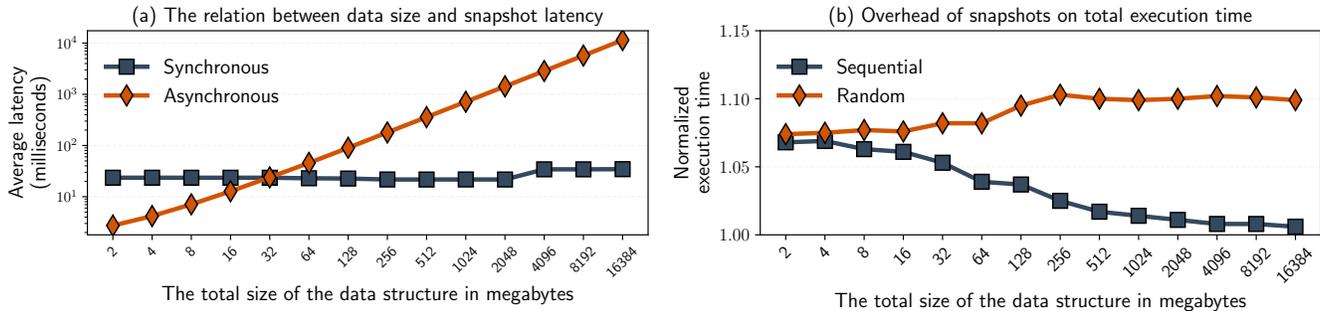
The first benchmark studies how the latency of the synchronous and asynchronous steps of creating snapshots change in response to increasing the workload size. Figure 13 (a) presents the outcome of this benchmark that varies the workload size (i.e., size of the persistent objects) from 2 MB to 16 GB and measures the latency of both synchronous and asynchronous paths of creating snapshots. The latency of the asynchronous path grows linearly with the workload size, as the size of memory regions that Pronto must persist on NVMM increases. However, the latency of the synchronous path only changes from 22 to 34 milliseconds. Thus, Pronto only stalls those update operations that run during the first few milliseconds of creating a new snapshot.

The other benchmark evaluates the impact of snapshots on the total execution time of programs that perform sequential or random 64-bit memory accesses (50% read and 50% write). We vary the workload size and run the benchmark with and without creating a snapshot to calculate normalized execution times. We vary the frequency of creating snapshots between 2 ms and 16 seconds based on the size of the data structure. Figure 13 (b) shows the normalized execution time for this benchmark. As the workload size increases, the impact of creating snapshots on the execution time converges to a constant: for programs with random memory access, the constant overhead is about 10%, while programs with sequential memory access only suffer from a 0.8% increase of the execution time. The overhead of Pronto’s snapshots is higher on the random-access benchmark because randomly accessing memory while creating an asynchronous snapshot escalates the chance of writing to read-only memory pages, which increases synchronous writes to NVMM as well as the impact of Pronto’s snapshots on the total execution time.

### 5.8 Recovery Time

We use a new benchmark, which uses Pronto to implement failure-atomic quick-sort, to measure the impact of data-structure size (i.e., size of the online image), number of threads, and snapshot frequency on the recovery time. The benchmark uses quick-sort to sort a large string array, comprising 1 KB strings. We vary the number of elements in the array from  $2^{20}$  (1 GB) to  $2^{25}$  (32 GB), the number of sort threads from 1 to 8, and the snapshot frequency from 2 to 32 seconds. Pronto uses 16 threads to load the snapshot and a single thread to replay semantic logs during recovery.

These experiments show that the primary determinant of recovery time for the failure-atomic quick-sort is the object size, as the snapshot frequency and the number of sort threads has no significant impact on the recovery time. Pronto recovers the 1 GB and 32 GB objects in less than 400 milliseconds and 7 seconds, respectively.



**Figure 13.** Measuring the impact of data size (i.e., total memory allocated by persistent objects) on the overhead of Pronto’s periodic snapshots.

Pronto Structure	Volatile Memory Online Image	NVMM	
		Semantic Logs	Snapshot
HashMap	4.48 GB	2.25 GB	4.38 GB
Map	4.23 GB	2.15 GB	4.13 GB
B+Tree	2.71 GB	0.70 GB	1.57 GB

**Table 2.** The storage cost of Pronto for HashMap and Map data structures from Figure 9 and the B+Tree from Figure 11. Snapshot cost only includes the space for the active snapshot.

Structure	Volatile Memory	NVMM
PMEMKV	5.98 GB	5.31 GB
B+Tree + PMDK	1.65 GB	1.16 GB
B+Tree + KaminoTx	1.66 GB	2.32 GB

**Table 3.** The storage cost of PMEMKV from Figure 9 and the PMDK and KaminoTX versions of the B+Tree from Figure 11.

## 5.9 Storage Cost

We use the benchmarks from Section 5.3 and Section 5.5 to compare the storage cost of Pronto against PMEMKV, PMDK, and KaminoTx. We measure the volatile and persistent memory footprints of Pronto data structures and report the cost in Table 2. Similarly, we report the storage cost of PMEMKV as well as PMDK and KaminoTx data structures in Table 3. All numbers are for single-threaded configurations with 1 KB values.

In contrast to PMEMKV, Pronto key-value stores (HashMap and Map) require 27% less volatile memory, while using 22% more persistent storage. For the B+Tree benchmark (Figure 11), Pronto’s volatile memory footprint is 61% higher than PMDK and KaminoTx. Pronto’s persistent memory requirement for the B+Tree is respectively 95% higher and 2% lower than PMDK and KaminoTx.

It is worth to note that Pronto does not need to store snapshots on NVMM as it creates snapshots asynchronously and off of the critical path. Thus, Pronto can

utilize SSDs for snapshot storage, which would significantly reduce its NVMM footprint (e.g., by up to 69% for the B+Tree).

## 6 Related Work

A large body of research with a focus on NVMM implications on computer architecture [44, 56], system software [52, 55], and programming support [11, 51] exists that addresses different challenges of integrating NVMMs with existing computer hardware and software. This work, in particular, focuses on reducing the overhead of adding failure-atomicity to volatile data structures in systems equipped with both volatile and non-volatile memories.

Researchers have built several persistent object libraries for NVMMs. NV-Heaps [11], Mnemosyne [51], and PMDK [25] provide libraries that allow programs directly and transactionally access NVMM. NVM Direct [7] achieves similar goals and adds compiler support. In contrast to Pronto, these systems require disruptive changes to existing programs and impose the overhead of transactional persistence on the critical path of execution.

Kamino-Tx removes the overhead of logging from the critical path and provides atomic in-place updates by maintaining two copies of persistent data [36]. It provides the same set of programming interfaces as PMDK and supports building highly available and reliable persistent data structures via replication. Compared to Pronto, it demands significant changes to existing programs; it also requires persisting transaction and allocation metadata in the critical path.

Atlas [9] automates enforcing failure-atomicity so long as persistent data is only modified inside critical sections, which are surrounded by acquisition and release of locks. NVthreads [20] provides similar failure-atomicity guarantees by using the page table protection bits to automatically track data modifications at the granularity of virtual memory pages and implement copy-on-write.

JUSTDO [27] extends on the idea of failure-atomic critical sections and utilizes persistent CPU caches to reduce the memory footprint of logs. In contrast, Pronto provides failure-atomic updates to data structures at the granularity of method calls, uses its allocator to track modified regions that it must persist on NVMM, and moves logging off the critical path without requiring hardware support.

Other work has focused on automatically creating persistent versions of volatile data structures. In [28], the authors explore a transform that takes a nonblocking, volatile data structure and creates a persistent version by transforming memory fences into cache-line flushes into NVMM. In contrast to this work, Pronto supports blocking data structures and also avoids extraneous cache-line flushes by moving most of the persistence instructions off the critical path.

Periodic checkpoints [1] and persistent virtual memory (pVM [30]) are other means of providing failure-atomicity to programs. However, they both require rigorous changes to the source code and enforce persistence synchronously.

## 7 Conclusion

We have described Pronto, a system that adds persistence to both sequential and concurrent volatile data structures and reduces the overhead of durability on the critical path of execution through asynchronous semantic logging. Pronto shrinks the performance gap between volatile and persistent data structures by trading recovery time for faster execution. It allows programmers to add failure-atomicity to existing code (e.g., GNU C++ STL containers) without requiring disruptive changes, while the resulting persistent containers provide comparable performance to the volatile versions. Furthermore, our persistent version of the STL's map container outperforms PMEMKV, a persistent key-value store highly optimized for NVMM, by up to 3.8 $\times$ .

## Acknowledgments

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We would like to thank Abhishek Bhattacharjee and the anonymous reviewers for their insightful feedback. We are also thankful to Intel Corporation for hardware access.

## References

- [1] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 439–451. <https://doi.org/10.1109/ISCA.2018.00044>
- [2] Andy Rudoff. 2016. Deprecating the PCOMMIT Instruction. Available at <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>.
- [3] Andy Rudoff. 2018. Persistent Memory Programming: The Current State and Future Direction. Available at [https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/03\\_PMSummit\\_18\\_Rudoff\\_Final\\_Post.pdf](https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/03_PMSummit_18_Rudoff_Final_Post.pdf).
- [4] Anirudh Badam. 2013. How Persistent Memory will Change Software Systems. *Computer* 46, 8 (August 2013), 45–51. <https://doi.org/10.1109/MC.2013.189>
- [5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications (*ASPLOS IX*). ACM, New York, NY, USA, 117–128. <https://doi.org/10.1145/378993.379232>
- [6] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2012. Implications of CPU Caching on Byte-Addressable Non-Volatile Memory Programming. *Hewlett-Packard, Tech. Rep. HPL-2012-236* (2012).
- [7] Bill Bridge. 2015. NVM Support for C Applications. Available at [http://www.snia.org/sites/default/files/BillBridgeNVM\\_Summit2015Slides.pdf](http://www.snia.org/sites/default/files/BillBridgeNVM_Summit2015Slides.pdf).
- [8] Chad Thibodeau, Arthur Sainio, Mark Carlson and Alex McDonald. 2017. Containers and Persistent Memory. Available at <https://www.snia.org/sites/default/files/CSI/Containers-and-Persistent-Memory-FInal.pdf>.
- [9] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency (*OOPSLA '14'*). ACM, New York, NY, USA, 433–452. <https://doi.org/10.1145/2660193.2660224>
- [10] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. <https://doi.org/10.14778/2752939.2752947>
- [11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories (*ASPLOS XVI*). ACM, New York, NY, USA, 105–118. <https://doi.org/10.1145/1950365.1950380>
- [12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory (*SOSP '09'*). ACM, New York, NY, USA, 133–146. <https://doi.org/10.1145/1629575.1629589>
- [13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB (*SoCC '10'*). ACM, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
- [14] Intel Corporation. 2015. Intel/Micron 3D-Xpoint Non-Volatile Main Memory. Available at <https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html>.
- [15] J Evans. 2016. Scalable Memory Allocation using jemalloc, 2011. (2016). <https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919>
- [16] Facebook. 2017. RocksDB. <http://rocksdb.org>.
- [17] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 371–384. <https://www.usenix.org/conference/nsdi13/technical->

- sessions/presentation/fan
- [18] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (Aug. 2004). <http://dl.acm.org/citation.cfm?id=1012889.1012894>
- [19] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. <https://doi.org/10.1145/78969.78972>
- [20] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications (*EuroSys '17*). ACM, New York, NY, USA, 468–482. <https://doi.org/10.1145/3064176.3064204>
- [21] Intel. 2015. An introduction to pmemcheck. Available at <http://pmem.io/2015/07/17/pmemcheck-basic.html>.
- [22] Intel Corporation. 2016. Enterprise and Cloud Storage Processor for the Digital Era. Available at <https://www.intel.sg/content/www/xa/en/storage/enterprise-cloud-storage-processor.html>.
- [23] Intel Corporation. 2019. Intel Optane DC Persistent Memory. Available at <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>.
- [24] Intel Corporation. 2019. Non-Volatile Memory. Available at <http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html>.
- [25] Intel Corporation. 2019. Persistent Memory Development Kit. Available at <http://pmem.io/pmdk/>.
- [26] Intel Corporation. 2019. PMemKV. Available at <https://github.com/pmem/pmemkv>.
- [27] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging (*ASPLOS '16*). ACM, New York, NY, USA, 427–442. <https://doi.org/10.1145/2872362.2872410>
- [28] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.
- [29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 <http://arxiv.org/abs/1903.05714>
- [30] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage (*EuroSys '16*). ACM, New York, NY, USA, Article 13, 16 pages. <https://doi.org/10.1145/2901318.2901325>
- [31] Kevin O'leary. 2016. How to Detect Persistent Memory Programming Errors using Intel Inspector. Available at <https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector>.
- [32] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing (*EuroSys '14*). ACM, New York, NY, USA, Article 27, 14 pages. <https://doi.org/10.1145/2592798.2592820>
- [33] Linux Kernel Organization. 2018. Direct Access for Files. Available at <https://www.kernel.org/doc/Documentation/filesystems/dax.txt>.
- [34] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. Pmtest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 411–425.
- [35] Mark Carlson. 2018. Persistent Memory: What Developers Need to Know. Available at <https://www.snia.org/sites/default/files/SDCEMEA/2018/Presentations/Persistent-Memory-for-Developers-SNIA-SDC-EMEA-2018.pdf>.
- [36] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx (*EuroSys '17*). ACM, New York, NY, USA, 499–512. <https://doi.org/10.1145/3064176.3064215>
- [37] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software (*ICCD '18*). 413–422. <https://doi.org/10.1109/ICCD.2018.00069>
- [38] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms (*PODC '96*). ACM, New York, NY, USA, 267–275. <https://doi.org/10.1145/248052.248106>
- [39] Micron Technology. 2019. Breakthrough Non-Volatile Memory Technology. Available at <https://www.micron.com/about/merging-technologies/3d-xpoint-technology>.
- [40] Micron Technology. 2019. NVDIMM. Available at <https://www.micron.com/products/dram-modules/nvdimm/>.
- [41] Mike Ferron-Jones. 2019. A New Breakthrough in Persistent Memory Gets Its First Public Demo. Available at <https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/>.
- [42] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. <https://doi.org/10.1145/128765.128770>
- [43] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16. <https://doi.org/10.4230/LIPIcs.DISC.2017.37>
- [44] M. A. Ogleari, E. L. Miller, and J. Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 336–349. <https://doi.org/10.1109/HPCA.2018.00037>
- [45] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency (*ISCA '14*). IEEE Press, Piscataway, NJ, USA, 265–276. <http://dl.acm.org/citation.cfm?id=2665671.2665712>
- [46] pmem.io. 2018. Using Persistent Memory Devices with the Linux Device Mapper. Available at [https://pmem.io/2018/05/15/using\\_persistent\\_memory\\_devices\\_with\\_the\\_linux\\_device\\_mapper.html](https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html).
- [47] Andy Rudoff. 2017. Persistent Memory Programming. *USENIX Association* 42, 2 (2017), 34–40.
- [48] Andy Rudoff. 2017. Persistent Memory: The Value to HPC and the Challenges (*MCHPC'17*). ACM, New York, NY, USA,

- 7–10. <https://doi.org/10.1145/3145617.3158213>
- [49] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories (*IMDM '15*). ACM, New York, NY, USA, Article 4, 8 pages. <https://doi.org/10.1145/2803140.2803144>
- [50] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory (*FAST'11*). USENIX Association, Berkeley, CA, USA, 1. <http://dl.acm.org/citation.cfm?id=1960475.1960480>
- [51] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory (*ASPLOS XVI*). ACM, New York, NY, USA, 91–104. <https://doi.org/10.1145/1950365.1950379>
- [52] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA, 323–338.
- [53] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems (*FAST'15*). USENIX Association, Berkeley, CA, USA, 167–181. <http://dl.acm.org/citation.cfm?id=2750482.2750495>
- [54] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA, 897–912.
- [55] Yiyang Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System (*ASPLOS '15*). ACM, New York, NY, USA, 3–18. <https://doi.org/10.1145/2694344.2694370>
- [56] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. 2013. Kiln: Closing the Performance Gap between Systems with and without Persistence Support. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 421–432.

## A Artifact Appendix

### A.1 Abstract

This artifact description provides the necessary information to build Pronto and run its performance and functionality tests. First, we give pointers to the source code and describe the hardware/software requirements for building and running the experiments. Next, we introduce the datasets used in evaluating Pronto, and then outline the necessary steps to run the experiments from Section 5. Finally, we explain how to read the evaluation results and introduce ways to configure the benchmarks (e.g., to reduce the evaluation time).

### A.2 Artifact check-list (meta-information)

- **Algorithm:** Asynchronous Semantic Logging (ASL) and Asynchronous Checkpointing.
- **Program:** Pronto’s library (including debugging tools and unit-tests), PMemKV v0.3x, RocksDB v5.17 (vanilla and Pronto versions), and STL wrappers for Pronto.
- **Compilation:** GNU C/C++ Compiler (version 7.4.0).
- **Data set:** Traces from YCSB [13].
- **Run-time environment:** See Section 5.1 for details.
- **Hardware:** You can reproduce the evaluation results by running the experiments on a machine equipped with NVMM (see Section 5.1 for details). Otherwise, you need a machine with at least 8 physical cores per socket and 100 GB of memory to run all experiments.
- **Execution:** See A.5 for details.
- **Metrics:** Performance (latency and throughput) of Pronto structures, recovery time, snapshot overhead, and sensitivity of ASL to the operation latency.
- **Output:** Performance of running YCSB traces against RocksDB, PMemKV, STL containers, and Pronto structures as well as the execution cost of the snapshots.
- **How much disk space required (approximately)?** Running the evaluations inside Docker requires 10 GB.
- **How much time is needed to prepare workflow (approximately)?** The experiments are ready to run in about 10 minutes (creating the Docker image is nonsupervised).
- **How much time is needed to complete experiments (approximately)?** About 7 hours to run all the experiments by running the Docker container.
- **Publicly available?** Code, datasets, unit-tests, tools, and benchmarks are publicly available. The only exception is the B+Tree in Figure 11, which is Microsoft proprietary and is not included in the archive.
- **Archived (provide DOI)?** 10.5281/zenodo.3605351.

### A.3 Description

**A.3.1 How delivered.** The artifacts are publicly available through Zenodo archival repository. You can access the code by using its DOI.

**A.3.2 Hardware dependencies.** We used two configurations for the development and final measurements of Pronto. In absence of access to real NVMM (e.g., Intel Optane DC), you can use the development configuration.

- **Evaluation setup:** We have evaluated Pronto’s performance using the testbed from Section 5.1. The evaluations, however, only require 8 physical cores per socket, 50+ gigabytes of NVMM, and 50+ gigabytes of DRAM. The benchmarks (almost always) use only one socket, so you do not need access to more than one CPU on multi-socket systems to run the benchmarks.
- **Development setup:** You need a machine with at least 8 physical cores per socket and 100 GB of memory to run all experiments. You will need to reserve 50 GB of memory to emulate NVMM (see <https://pmem.io/2016/02/22/pm-emulation.html> for instructions).

**A.3.3 Software dependencies.** We have tested Pronto on Ubuntu 18.04 and created a list of required dependencies for the test platform. Run `dependencies.sh` to build/install the necessary binaries for building Pronto and running the experiments. For other platforms, you need to manually install the following (versions are for the development platform):

- jemalloc v5.1.0
- PMDK v1.4.1
- Python v2.7.15
- NumPy v1.16.0
- CMake v3.10.2
- Autoconf v2.69
- libz-dev v1.3.3
- libdaxctl-dev v61.2
- libndctl-dev v61.2
- pkg-config v0.29.1
- uuid-dev v2.31.1
- numactl v2.0.11

To run the unit-tests, you also need to install *Google C++ Testing Framework*. You can find the source code and dependencies for the test framework in `googletest` and `gflags`.

**A.3.4 Data sets.** Pronto’s performance tests use traces from YCSB [13] (workloads A and B) to measure both latency and throughput of benchmark applications. All traces, as well as YCSB, are publicly available.

### A.4 Installation

Pronto uses `Make` for the compilation of the library and accepts multiple configurations (e.g., size of the semantic logs) via environment variables. The following commands configure the compilation environment and build the release version of Pronto using the GNU C/C++ compiler.

```
cd src
export DEBUG=1 # enables debug information

# updates the size of semantic-logs (gigabytes)
export LOG_SIZE=16

# disables core pinning for ASL threads
export DISABLE_HT_PINNING=1

# enables synchronous semantic logging
export PRONTO_SYNC=1

make # builds the static library
```

You can also use the commands below to run the unit-tests and verify the basic functionality of Pronto’s library. Make sure to reserve huge-pages for Pronto’s allocator and have an NVMM file-system mounted at `/mnt/ram`.

```
# mounts /dev/pmem1 as /mnt/ram (ext4-dax)
./init_ext4.sh 48 # partition size in GB

# reserves 1024 huge-pages for the allocator
echo 1024 | tee -a /proc/sys/vm/nr_hugepages

# builds and runs all unit-tests
cd unit && make && ./test
```

## A.5 Experiment workflow

There are two ways to run experiments. You can choose to run all the experiments from Section 5 in a Docker container, or customize and run individual experiments outside Docker.

**A.5.1 Running experiments in Docker.** Run the following commands to create a Docker image for Pronto and run all the experiments in a container. Creating the image and running the experiments take approximately 10 minutes and 7 hours, respectively. Note that the container uses `/dev/pmem1` as the NVMM device to run all the experiments and stores results under `/tmp`. You can update `init.sh` and `run.sh` or use Docker’s device mapping options to change this behavior.

```
# make sure to start with a clean repository
cd docker && ./arxiv.sh
docker build --tag=pronto .
docker run --privileged -v /tmp:/tmp pronto
```

**A.5.2 Running individual experiments.** You can also configure and run performance, recovery, and sensitivity experiments individually and outside Docker. Follow the instructions in the *README* file under the `benchmark` directory in the source repository for details.

## A.6 Evaluation and expected result

We divide benchmarks into three categories: performance, recovery, and sensitivity analysis. There are separate scripts to run benchmarks in each category, and they all print results in CSV format. Pronto’s Docker containers use the same categories and store experiment results in three files:

- `pronto-perf.csv`
- `pronto-recovery.csv`
- `pronto-sensitivity.csv`

**A.6.1 Performance.** Pronto’s performance benchmarks include STL, PMemKV, and RocksDB. For each benchmark, the script prints out its name, workload, number of threads, data size, iteration number, average latency, and average throughput. For instance, below is the output for running YCSB-A (one client thread) against RocksDB (sync mode), where the data size is 256 bytes, and the average latency and throughput across 5 runs are 6.5  $\mu$ s and 154 Kops/sec, respectively.

```
rocksdb , a-sync , 1 , 256 , 0 , 6336 , 157822
rocksdb , a-sync , 1 , 256 , 1 , 6455 , 154895
rocksdb , a-sync , 1 , 256 , 2 , 6326 , 158065
rocksdb , a-sync , 1 , 256 , 3 , 7127 , 140297
rocksdb , a-sync , 1 , 256 , 4 , 6206 , 161124
```

**A.6.2 Recovery.** There are three benchmarks:

1. The first benchmark measures the recovery time for different workload configurations (see Section 5.8 for details). Pronto’s Docker containers skip this benchmark by default, as it takes several hours to complete.
2. The second benchmark measures the overhead of snapshots on the execution time of programs. For each iteration, the benchmark reports the access pattern (random or sequential), size of the data structure (number of 2 MB pages), and execution time without/with periodic snapshots.
3. The last benchmark evaluates the cost of snapshots on the critical path. It varies the size of the data structure from 2 MB to 16 GB and reports the cost of creating a snapshot on and off the critical path in microseconds.

**A.6.3 Sensitivity analysis.** This benchmark varies the latency of volatile operations from 100 ns to 100  $\mu$ s and reports the cost of synchronous and asynchronous semantic logging. The example below shows synchronous semantic logging increases the execution time by 2,381 ns for a 1000 ns operation and a 1024 byte semantic-log. The last column is the standard deviation for the experiment across five runs.

```
1000 , 1024 , sync , 2381.81 , 1.19
```

## A.7 Experiment customization

Refer to the documentation under the `benchmark` directory in the code repository for details on configuring the benchmarks.

## A.8 Notes

The documentation (i.e., *README* files) that accompanies the source code contains additional information for using the code as well as further instructions on setting up and running the benchmarks.

## A.9 Methodology

Submission, reviewing and badging methodology:

- <http://cTuning.org/ae/submission-20190109.html>
- <http://cTuning.org/ae/reviewing-20190109.html>
- <https://www.acm.org/publications/policies/artifact-review-badging>