Mangosteen: Fast Transparent Durability for Linearizable Applications using NVM

Sergey Egorov, Gregory Chockler, and Brijesh Dongol, *University of Surrey, UK;*
Dan O'Keeffe, *Royal Holloway, University of London, UK;*
Sadegh Keshavarzi, *University of Surrey, UK*

https://www.usenix.org/conference/atc24/presentation/egorov

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Mangosteen: Fast Transparent Durability for Linearizable Applications using NVM

Sergey Egorov[1], Gregory Chockler[1], Brijesh Dongol[1], Dan O'Keeffe[2], and Sadegh Keshavarzi[1]

[1]University of Surrey, UK

[2]Royal Holloway, University of London, UK

Abstract

The advent of byte-addressable non-volatile memory (NVM) technologies has enabled the development of low-latency high-throughput durable applications, i.e., applications that are capable of recovering from full-system crashes. However, programming such applications is error-prone as efficiency gains often require fine-grained (programmer-controlled) management of low-level persistence instructions.

We propose Mangosteen, a high-level programming framework that allows developers to transform an existing linearizable in-memory application to a corresponding durably linearizable version using NVM. Our framework's API consists of a set of callback hooks that interpose on an application's request processing flow with minimal developer effort. Mangosteen executes client operations on DRAM and persists their effects using binary instrumentation and redo logging. Mangosteen's concurrency control facilitates batching of read-write requests to minimize the cost of persistence, while allowing read-only requests to execute concurrently. A novel intra-batch deduplication mechanism further reduces persistence overheads for common OLTP workloads. Our empirical evaluation results show that Mangosteen-enabled applications outperform state-of-the-art solutions across the entire spectrum of read-write ratios. In particular, the Mangosteen-based version of Redis demonstrates throughput gains of between 2×–5× in comparison to prior work.

1 Introduction

In-memory storage applications (e.g., Redis, Memcached) are popular alternatives to traditional disk-oriented databases for both analytical and transactional workloads [41]. The advent of *non-volatile memory* (NVM) hardware such as Intel Optane and CXL Memory-Semantic SSDs [39] has driven the development of high-performance durable (aka *failure atomic*) versions of such in-memory applications [29, 50]. NVM is byte-addressable and greatly reduces the cost of persistency. However, achieving the purported benefits of NVM for complex existing applications is challenging.

First, the latency and memory bandwidth of existing NVM hardware is not yet comparable to DRAM [49]. NVM reads measure 2x slower and writes 3x slower than even the slowest DRAM. As a result, NVM cannot be used directly as a drop-in replacement for DRAM in existing in-memory applications without incurring a substantial performance overhead.

Second, correct recovery after failures is challenging since program failures may occur at any point, potentially leaving the application in an inconsistent state. To ensure correctness, the system must be made *crash-consistent*, i.e., guaranteed to recover into a consistent state after a failure. Here, a range of correctness conditions have been defined such as persistent atomicity [18], (buffered) durable linearizability [24] and recoverable linearizability [2] (see [1] for a survey). These define correctness in terms of both *failure atomicity* (in the event of a crash, an operation either occurs in its entirety or not at all), and *consistency* (concurrent operations can be understood in terms of sequential executions of the operations).

Unfortunately, achieving both correctness and good performance with NVM is notoriously difficult. Existing NVM interfaces require programmers to explicitly use costly persistence instructions (such as fences) to achieve durability. As a result, they must tread a thin line between avoiding unnecessary persistence instructions to achieve good performance and ensuring the desired correctness properties.

To address this issue, recent research proposed higher level abstractions, such as persistent transactional memory (PTM) [12, 35, 43], persistent object libraries [9, 42], and failure-atomic sections (FASEs) [8, 19, 23], to facilitate durability. From a programming perspective, these approaches are still non-trivial to use because of source annotation effort, compiler modifications or explicit library calls that need to be added for each durable update [35]. Furthermore, they require programmers to reason which parts of their code need to be made transactionally durable. This is particularly difficult and error-prone for large legacy codebases with complex existing dependencies (e.g. libraries and memory allocators) [30–32, 47]. As an alternative, Zhang et al. [50] proposed a framework that exploits dynamic binary instrumentation

(DBI) [7] to make persistence support more transparent. However, their framework relies on undo logging which requires a high number of fence operations and does not exploit concurrency to achieve performance gains on multicore hardware.

In this paper, we present Mangosteen, a new framework for transforming existing in-memory applications to exploit NVM on modern multi-core hardware. Mangosteen provides a high-level API consisting of a small set of callback hooks to interpose on an application's request processing with minimal developer effort. The Mangosteen runtime executes client requests concurrently on a DRAM copy of the application state. It exploits DBI to transparently persist the effects of update requests in a persistent redo log, which are then applied asynchronously to a persistent copy of the application state on NVM. In comparison to existing high-level persistence frameworks, Mangosteen achieves state-of-the-art performance on a range of real-world workloads.

Mangosteen's design is carefully optimized for performance. Mangosteen's concurrency control allows read-only requests to execute in parallel and employs flat-combining [20] to batch execution of read-write requests. This facilitates a novel intra-batch deduplication mechanism to further reduce persistence overheads for common OLTP workloads. Overall, Mangosteen is able to persist update effects using just 2 persistent fences *per batch* on the critical path, and another 2 persistent fences per batch to make them durable asynchronously on the persistent copy of the application state.

Unlike prior approaches based on redo logging, Mangosteen's binary instrumentation only needs to intercept write instructions. Mangosteen also enables applications to use their existing memory allocators unmodified, and employs a novel split allocation scheme to avoid unnecessary overhead for updates to transient memory that do not need to be persisted for correct recovery.

For our empirical evaluation, we compared Mangosteen's performance against state-of-the-art persistence frameworks – Persimmon [50] and Romulus [12]. For that, we implemented two prototypes of Mangosteen-enabled applications: a persistent version of Redis and a persistent key-value store compatible with the LevelDB API. Our results demonstrate that both prototypes achieve throughput gains of between 2×–5× compared to the Persimmon-based version of Redis [50], and 2.8×–6.5× compared to RomulusDB, a LevelDB API implementation based on Romulus PTM [12].

Overview. The paper is organized as follows. In the next section we give background necessary to understand the rest of the paper. We then describe the high-level design of Mangosteen including its API and runtime architecture (§3). Next, we give details of Mangosteen's core algorithms and optimizations it employs to achieve good performance (§4). We present a sketch of the correctness argument in §5, and the performance analysis in §6. We present related work in §7 and finally, §8 concludes the paper.

2 Preliminaries

Memory model. The interaction between NVM and both language and hardware memory models are well studied [24, 26, 37]. Throughout this paper, to enable us to focus on the core design principles of Mangosteen, our high-level models assume *sequential consistency* [27] enhanced with persistency operations: pwb(x) (i.e., *persistent write back* of location x), pfence and psync to manage NVM updates [26].

We assume pwb(x) tags the given location x as a location to be flushed; tagged locations are only guaranteed to be persisted after the execution of psync. Additionally, we assume a *persist barrier* pfence [12, 24, 36]), which separates different epochs [34]. A pwb'd write followed by a pfence, further followed by a second write are persisted in order. The differences between these instructions are further illustrated in Example 1 in Appendix A.

We implement pwb and psync using the PMDK API [40], in particular by functions pmem_flush and pmem_drain, respectively. Although operation pfence appears in our pseudocode, it is currently not supported directly by any architecture, and hence, as in prior work (e.g., [12]), is implemented by the more heavyweight pmem_drain operation [12].

Correctness guarantees. Mangosteen supports *durable linearizability* [24] as its main correctness guarantee. As in standard linearizability [22], a durably linearizable concurrent execution is correct with respect to its (sequential) specification iff it can be linearized in a manner consistent with the order of non-overlapping requests to form a history of its specification. In addition, a durably linearizable object must ensure *failure atomicity*, i.e., completed operations are never lost (even after a crash), while incomplete operations may either be discarded completely or be kept in their entirety. With respect to liveness, Mangosteen provides *deadlock freedom* [21] (i.e., in the absence of system crashes, some operation that has been previously invoked is guaranteed to eventually return) under the assumption the original application is (at least) deadlock-free.

3 Mangosteen Design

Mangosteen functions as a generic wrapper for a linearizable application that resides in volatile memory (DRAM), allowing the application to handle concurrent requests in a fault tolerant (recoverable) manner. We next describe the Mangosteen framework's API and how it can be integrated with an existing application (§3.1). We then give an overview of the key architectural features of the Mangosteen runtime (§3.2).

3.1 Mangosteen API

The Mangosteen framework is implemented as a user-level library. Mangosteen's application programming interface is given in Listing 1. The API consists of 6 callback

functions through which the Mangosteen framework interacts with the application. A framework initialization function (`mangosteenInit`) takes as its arguments pointers to the implementations of each callback. Two of the callbacks (`initAppClient` and `destroyAppClient`) are executed once for every application client, while the remaining callbacks are executed once for each client request.

```
int initMangosteen(/*... Callback fn ptrs ...*/, int mode);

void* initAppClient(void *params);
void destroyAppClient(void *c);
void deserializeRequest(void *c, void *req);
bool isReadOnly(void *c);
void processRequest(void *c);
mangoResponse* getResponse(void *c);
```

Listing 1: Mangosteen API

The framework is flexible in that it supports both an *RPC mode* and a *local mode* (as indicated by an additional `mode` argument to `mangosteenInit`). RPC mode targets settings where clients interact with the application over a network. Local mode can be used for integration of persistent data structures into a standalone application.

To use Mangosteen in the RPC mode, the application developer must implement all callbacks in Listing 1 though for most applications this can be achieved using simple wrapper functions around existing application code. For the local mode, the applications are only required to implement `isReadOnly` and `processRequest`, and call `ClientCmd` (§3.2) directly to execute requests.

KV Store Example (RPC Mode). We next describe a basic in-memory key-value (KV) store to illustrate how a developer can integrate with the Mangosteen API. We first give the core functionality of an initial (non-persistent) version of the KV store (Listing 2). [1] Its state consists of an array `store` that is initialized in `initKVStore` (line 16). The store supports two operations, `getValueAtIndex` and `putValueAtIndex` (lines 20 and 25), which get and set respectively the value at a given index into the array. Both operations take, as an argument, a client context object `c` that identifies the client issuing the command (`c->id`), the command itself (`c->currCmd`) and a buffer to store the response (`c->response`). The `handleRequest` function (line 31) dispatches to the appropriate operation according to the type of the current command (`c->currCmd->type`). On return from `handleRequest`, the store's networking layer (not shown) replies to the client with the response contained in `c->resp`.

```
char *store;

typedef struct kvCmd {
    enum {PUT, GET} type;
    unsigned int index;
    char value[VAL_SZ];
} kvCmd;

typedef struct kvClient {
    int id;
    kvCmd* currCmd;
    char resp[RESP_SZ];
} kvClient;


void initKVStore() {
    store = malloc(STORE_SZ * VAL_SZ);
}

void getValueAtIndex(kvClient *c) {
    memcpy(c->resp,
      &store[c->currCmd->index * VAL_SZ], VAL_SZ);
}

void putValueAtIndex(kvClient *c) {
    memcpy(&store[c->currCmd->index * VAL_SZ],
      c->currCmd->value, VAL_SZ);
    memcpy(c->resp, "OK",2);
}

void handleRequest(kvClient *c) {
    if(c->currCmd->type == PUT)
        putValueAtIndex(c);
    else
        getValueAtIndex(c);
}
```

Listing 2: Basic In-Memory KV Store

Mangosteen Integration. Listings 3 and 4 illustrate how to use the Mangosteen API to create a *durable* version of the in-memory KV store. The application first initializes its state as usual using `initKVStore` (Listing 3). It then invokes `initMangosteen` to start the Mangosteen framework's runtime (see §3.2), passing as parameters function pointers to implementations of the Mangosteen API callbacks and configuring Mangosteen to use RPC mode.

Mangosteen invokes the first callback, `initAppClient` (Listing 4, line 1), whenever a new client is initialized, e.g. when a client first connects to the application in RPC mode. The KV store application uses this callback to initialize a new client context object (line 2). The structure of this object is application-specific and remains opaque to Mangosteen, but is passed as a parameter to subsequent application callbacks during request processing. The `params` argument contains a unique client identifier that the application can optionally record within the client object. When a client terminates, Mangosteen invokes a corresponding `destroyAppClient` callback to allow the application to clean up (line 7).

The `deserializeRequest` callback performs application-specific deserialization of an incoming request `req` (line 11). Mangosteen assumes the deserialized request is stored in an application-specific location in `opaqueCtx`. For our basic KV store, no complex deserialization is required and the location of the request buffer is recorded directly in `currCmd`.

The `isReadOnly` callback (line 15) allows the application to indicate to Mangosteen which requests (if any) do not modify the application's state. This enables Mangosteen to exploit read-read concurrency for improved performance. For the KV store, `getValueAtIndex` is considered read-only, whereas `putValueAtIndex` is read-write since it modifies `store`.

The `processRequest` callback (line 21) interposes on the application's core request processing logic. For the KV store example it is a simple wrapper around the `handleRequest` function.

[1] For simplicity we omit bounds-checking and error-handling and assume fixed-size values.

Mangosteen assumes that the results of request processing are recorded in `opaqueCtx`, e.g., in the KV store they are stored in a client `c`'s response buffer `c->resp`.

The last callback, `getResponse`, allows Mangosteen to extract the response (line 25). It returns a `mangoResponse` object consisting of a pointer to the start of the response (i.e. `c->resp` for the KV store) and the size of the response. The Mangosteen runtime then responds to the client.

```
1  int main() {
2      initKVStore();
3
4      initMangosteen(&initAppClient,
5                     &deserializeRequest,
6                     &isReadOnly,
7                     &processRequest,
8                     &destroyAppClient,
9                     &getResponse,
10                    MODE_RPC);
11     return 0;
12 }
```

Listing 3: Mangosteen Initialization

```
1  void *initAppClient(void *params) {
2      kvClient* c = malloc(sizeof(kvClient));
3      c->id = *(int*)params;
4      return c;
5  }
6
7  void destroyAppClient(void *opaqueCtx) {
8      free(opaqueCtx);
9  }
10
11 void deserializeRequest(void *opaqueCtx, void *req) {
12     ((kvClient*)opaqueCtx)->currCmd = (kvCmd*)req;
13 }
14
15 bool isReadOnly(void *opaqueCtx) {
16     if(((kvClient*)opaqueCtx)->currCmd->type == PUT)
17         return false;
18     return true;
19 }
20
21 void processRequest(void *opaqueCtx) {
22     handleRequest((kvClient*)opaqueCtx);
23 }
24
25 mangoResponse *getResponse(void *opaqueCtx) {
26     kvClient *c = (kvClient*)opaqueCtx;
27     mangoResponse *mResponse = malloc(sizeof(mangoResponse));
28     mResponse->response = c->resp;
29     mResponse->size = VAL_SZ;
30     return mResponse;
31 }
```

Listing 4: KV Store Implementation of Mangosteen Callbacks

3.2 Architecture Overview

An overview of the Mangosteen architecture is given in Fig. 1. It consists of two main components:

(1) the *frontend*, which stores the application state in DRAM and processes client requests concurrently, and

(2) the *backend*, which interfaces with the frontend using a persistent redo log and also persists the application state in NVM, enabling recovery.

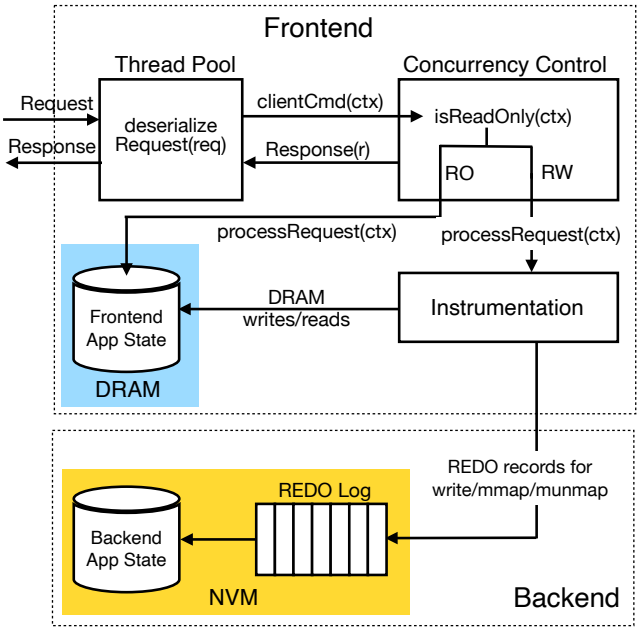


Figure 1: Architecture overview

Initialization. When the application starts it calls `mangosteenInit` to initialize the Mangosteen runtime. Mangosteen initialization follows the approach of [50]. The frontend executes within the application process. It first allocates the shared redo log in persistent memory, in addition to a persistent region table that the backend will use to record mapped regions in the application's address space. The frontend then checkpoints itself using CRIU [13], a Linux process checkpointing tool. This checkpoint serves as a base image of the application's address space during recovery. The frontend then forks to create the backend and blocks until the backend initialization completes.

To initialize itself, the backend maps relevant parts of its address space to persistent memory. To achieve this it creates a persistent memory region for all application memory regions listed in `/proc/self/maps` except the stack, read-only regions and the redo log. Each persistent memory region is a contiguous chunk of NVM backed by a file. The backend records the set of persistent memory regions in the persistent region table. Once backend initialization completes, its virtual address space memory mappings are identical to that of the frontend. The frontend then initializes a pool of worker threads (and in RPC mode an event loop for each thread) and begins processing client requests.

Request processing. When a client first establishes a connection with Mangosteen, it is assigned a thread from the *Thread Manager* pool, which allocates and initializes a thread-local Mangosteen client context and triggers the Mangosteen API's `initAppClient` callback (§3.1). Clients then submit requests to be executed, with each submitted request handled by the thread responsible for the client in RPC mode, and the client

thread itself in local mode.

Mangosteen deserializes requests in an application-specific manner using the `deserializeRequest` callback (see Fig. 1). It then passes the resulting client context object to the frontend's concurrency control mechanism (see `clientCmd(ctx)` in Fig. 1). Mangosteen executes read-only (RO) operations in parallel and *batches* execution of read-write (RW) operations using flat-combining [20] (§4.1). Before responding to the client, the frontend persists updates from RW operations in a persistent redo log (implemented as a fixed size ring-buffer) (§4.2). To capture updates in a transparent fashion, Mangosteen uses a DBI framework (DynamoRIO [5]) to intercept write instructions and `memcpy` operations. To minimize the cost of persistence, Mangosteen employs a novel intra-batch deduplication mechanism to filter redundant updates (§4.3). Once the updates for a RW command are persisted in the redo log, a response is returned to the client.

The Mangosteen backend asynchronously copies entries from the persistent redo log and applies the updates to an NVM copy of the DRAM application state. Once the changes to the NVM copy are persisted, the backend removes the corresponding entries from the redo log (§4.3).

Recovery. During recovery from a crash, Mangosteen first restores the CRIU checkpoint and then forks. Next, the backend iterates over the region table and remaps all persistent memory regions. The backend then processes all committed entries in the redo log and applies their updates (§4.3, Fig. 4). Once the backend is finished, the frontend iterates over the region table and recreates all memory mappings in DRAM. Finally, it copies data from the corresponding regions in persistent memory and resumes execution.

4 Optimizing Transparent Persistence

While transparency is valuable in the context of persistent memory, it cannot come at the cost of performance. In this section we describe in detail how Mangosteen's design ensure high performance, in particular its concurrency control scheme (§4.1), binary instrumentation (§4.2), persistence mechanisms (§4.3) and memory allocation (§4.4).

4.1 Mangosteen Concurrency Control

The Mangosteen concurrency control ensures durably linearizable [24] execution of commands supported by the underlying data store API. Its implementation aims to maximize concurrency while minimizing the cost of read-write commands persistence. To achieve this, we chose to support full read-only command parallelism while forgoing read-write vs read-only and read-write vs read-write parallelism for the sake of optimizing the persistent memory performance.

As in [12], the read-write commands are batched via flat-combining [20] and executed under an exclusive lock. However, unlike [12], we use binary instrumentation to convert the batched commands into a stream of elementary store instructions, which are then deduplicated and appended to a redo log via pwb. This reduces the overall amount of data that needs to be persisted per batch, and ensures that the persistent memory is always accessed sequentially on a critical path. Furthermore, as we discuss in §4.3, persisting a batch only requires 1 persistent fence (pfence) and 1 persistent sync (psync) instruction (as opposed to 3 pfences and 1 psync per transaction in [12]), amortizing the cost of synchronous persistence across multiple read-write commands.

Overall, our concurrency control implementation (Figure 2) follows the writer-preference lock (C-RW-WP) algorithm of [6], which we adapt to a multi-core setting and integrate

```
1  when notified ClientCmd(ctx)
2    if isReadOnly(ctx) then
3      while true do
4        rFlag[i] ← true;
5        if ¬wrFlag then
6          processRequest(ctx);
7          rFlag[i] ← false;
8          break;
9        else
10         rFlag[i] ← false;
11         while wrFlag do pause;
12   else
13     (ctx, status)[i] ← (ctx, READY);
14     while true do
15       if status[i] = STARTED then
16         while status[i] ≠ DONE do pause;
17       if status[i] = DONE then break;
18       if ¬wrFlag then
19         if CAS(wrFlag, false, true) then
20           readySet ← ∅;
21           repeat
22             forall {j | status[j] = READY}
                 do
23               status[j] ← STARTED;
24               readySet ← readySet ∪ {j};
25           until ∀k. ¬rFlag[k];
26           start_tx();
27           forall j ∈ readySet do
28             processRequest(ctx[j]);
29           end_tx();
30           forall j ∈ readySet do
               status[j] ← DONE;
31           wrFlag ← false;
32   trigger Response(getResponse(ctx));
```

Figure 2: Client request handler executed by thread i

with flat combining, as we explain below. Once a frontend thread i finishes receiving and deserializing a client request, it creates an opaque client request context *ctx*, and calls the client request handler ClientCmd with *ctx* as argument (line 2.1). The thread then calls isReadOnly (line 2.2) to determine the request type, and based on the outcome proceeds to execute either the read-only (lines 2.3–2.11) or the read-write (lines 2.13–2.31) request handling branch. Upon completion of either branch, the response is returned to the client in line 2.32. Below, we will refer to a thread executing in the read-only branch as a *reader* and a thread executing in the read-write branch as a *writer*.

Read-only branch. Once in the read-only branch, a reader i enters a while loop where it attempts to acquire a shared lock. To this end, it first sets its entry rFlag[i] in a shared memory array rFlag to `true` (line 2.4) to announce its intention to perform a read-only command. (To prevent false sharing among the readers, the entries in rFlag are cache-line aligned.) It then inspects wrFlag (line 2.5) to determine whether there is a concurrently executing writer. If not, the reader proceeds to execute the read-only request (line 2.6) using a main memory copy of the data store state, and sets rFlag[i] to `false` (line 2.7). It then breaks from the while loop and returns the response. Note that a writer will only able to acquire an exclusive lock once all concurrently executing readers have set their rFlag entries to `false` (see below).

If wrFlag is set, the reader first assigns `false` to rFlag[i] (line 2.10) to allow a writer that set wrFlag to complete the exclusive lock acquisition. It then proceeds to execute a busy-waiting loop in line 2.11 until wrFlag becomes `false`. The pause instruction suspends the thread for a short time duration to optimize busy waiting. Once wrFlag indicates a shared lock is available, the reader goes back to the beginning of the while loop to retry the shared lock acquisition.

Read-write branch. Upon entering the read-write branch, a writer i first advertises *ctx* as ready for processing by setting ctx[i] to *ctx* and status[i] to `READY` (line 2.13). (Both arrays are cache-line aligned to avoid false sharing among the writers.) It then enters a while loop where it attempts to acquire an exclusive lock by setting wrFlag to `true` via a compare-and-swap (CAS) instruction (line 2.19). To minimize the cache invalidation traffic caused by CAS, the writer first reads the current value of wrFlag, and only proceeds with CAS, if it finds wrFlag = `false` (line 2.18).

A writer, which is able to acquire an exclusive lock, becomes a *combiner*, and is responsible for processing all the read-write commands that have been advertized in ctx. To this end, it first executes the loop in lines 2.22–2.24 to ensure all shared locks previously acquired by the readers have been released. While in this loop, the combiner collects in *readySet* the indices of the status array entries corresponding to the read-write commands that are ready to be processed. It also flags these commands as `STARTED` thus causing the remaining (non-combiner) writers to abandon any further attempts to acquire the exclusive lock. These writers then proceed directly to line 2.16 where they await the completion of their requests by the combiner. This early notification mechanism helps to avoid unnecessary querying of wrFlag by non-combiners thus reducing the number of non-local shared memory accesses.

Once the combiner validates all shared locks have been relinquished, it calls start_tx to activate the instrumentation state machine (Figure 3). (To prevent start_tx from being instrumented, it is implemented as a DynamoRIO clean call [14].) The combiner then proceeds to execute the read-write commands occupying the entries ctx[j] for each $j \in$ *readySet* under instrumentation, which in particular, causes all subsequent write instructions to be aggregated and deduplicated by the instrumentation code (see §4.3). Once the combiner finishes processing the advertized commands, it issues a DynamoRIO clean call to end_tx, which disables instrumentation, and persists the accumulated write records to the redo log (see §4.3). The combiner then proceeds to flag the processed commands as `DONE` (line 2.30), which enables the writers awaiting their completion to respond to their clients (lines 2.17 and 2.32). Further details of our deduplication and persistence implementations can be found in §4.3.

4.2 Minimizing Combiner Instrumentation

Mangosteen relies on program instrumentation at the frontend to capture the effects of commands that modify the application's persistent state. While this simplifies the task of the persistent memory programmer, it is crucial the instrumentation is lightweight since it is on the critical path. To ensure this, Mangosteen exploits redo logging and the concurrency control mechanism described in the previous section to minimize the number of instrumented instructions.

Read-write commands. Write instructions in RW commands may modify the application's persistent state, and hence need to be instrumented.

Unlike prior work that combines DBI with undo logging [50], Mangosteen's redo logging does not need to immediately persist data for intercepted write instructions. Instead it only needs to add an address to a hashset, which can be implemented efficiently as inline instrumentation without needing an expensive DynamoRIO 'clean call' [14]. Furthermore, many large updates are implemented as calls to `memcpy` which can be intercepted by Mangosteen directly. Similar to prior work, Mangosteen assumes data on the stack is transient and filters any associated writes by checking if the destination address is below the stack pointer `rsp` (also see [50]).

For read instructions within RW commands, we observe that due to Mangosteen's flat-combining concurrency control mechanism, batches of RW commands are effectively executed sequentially by the same combiner thread. As a result, read instructions within a RW command cannot observe

conflicting updates from another concurrently executing RW command. Therefore, in contrast to prior approaches based on redo logging (e.g. [44]), it is unnecessary for Mangosteen to instrument read instructions within RW commands.

Read-only commands. In contrast to RW commands, for RO commands we assume the application's persistent state is never modified nor accessed at the same time as RW commands. This means that no instrumentation is required for read or write instructions in RO commands. We note that for RO commands front-end threads still execute under the control of our DBI framework, but we found the overhead to be minimal in the absence of any instrumentation.

4.3 Optimizing Persistence

Apart from synchronization and instrumentation overheads, a key performance concern for the Mangosteen frontend is the cost of storing updates in the persistent redo log. As shown in previous empirical work [48], persisting to NVM efficiently involves minimizing (i) the amount of data to persist and (ii) the number of fence instructions needed for correctness. We next describe how Mangosteen employs *write deduplication* and *batch redo logging* to meet these objectives.

Write deduplication. As discussed in the previous section, Mangosteen's frontend instrumentation (Figure 3) is responsible for capturing the effects of relevant write instructions and storing them in a persistent redo log. A straightforward implementation of frontend redo logging involves persisting an update for every relevant write instruction. However, this approach wastes persistent memory bandwidth when there are multiple updates to the same memory location, since the effects of earlier updates in the same batch of advertised commands are no longer relevant.

To minimize the amount of data that needs to be persisted to the redo log, Mangosteen deduplicates updates before persisting them. A challenge here is that write instructions that modify the same location in memory may have different granularities and alignments. Furthermore, the deduplication mechanism is on the critical path, so it must be carefully designed to minimize performance overheads.

To overcome these challenges, the Mangosteen deduplication algorithm (lines 6–12 in Figure 3) employs a thread-local hash table (H) to record the addresses modified by the write instructions within a batch of commands. The hash table is implemented as an array of 64-bit integers and uses open addressing with linear probing as a collision resolution scheme. The array's starting address is cache line aligned, and the entries are tightly packed without any internal padding. This allows a set of adjacent entries to be loaded together on each lookup thus optimizing the linear probing.

Each entry in the hash table represents the start of an aligned memory block of size `BLOCK_SIZE` bytes. This approach allows us to coalesce small non-overlapping writes within the same `BLOCK_SIZE` byte memory range into a single redo log entry, but may lead to write amplification if most writes are smaller than `BLOCK_SIZE` bytes. In our empirical evaluation (§6), we found that `BLOCK_SIZE` = 32 bytes gives

```
1  when called start_tx()
2      instr_status ← ACTIVE;
3      (head, tail) ← (rb_head, rb_tail);
4  when called write(addr, size)
5      pre: instr_status = ACTIVE
6      x1 ← ⌊addr/BLOCK_SIZE⌋;
7      x2 ← ⌊(addr + size)/BLOCK_SIZE⌋;
8      for a ← x1 to x2 by BLOCK_SIZE do
9          i ← a % length(H);
           // linear probing
10         while H[i] ≠ null ∧ H[i] ≠ a do
11             i ← (i + 1) % length(H);
12         H[i] ← a;
13 when called mmap(addr, size)
14     pre: instr_status = ACTIVE
15     rb_enque(⟨MMAP, addr, size⟩);
16     pfence();
17     p_rb_tail ← tail;
18     pwb(p_rb_tail);
19     pfence();
20 when called end_tx()
21     instr_status ← IDLE;
22     forall k = 0..length(H) − 1 do
23         if H[k] ≠ null then
24             memcpy(&blk, H[k], BLOCK_SIZE);
25             rb_enque(⟨REDO, H[k], blk⟩);
26             H[k] ← null;
27     rb_enque(⟨COMMIT⟩);
28     pfence();
29     p_rb_tail ← tail;
30     pwb(p_rb_tail);
31     psync();
32     (rb_head, rb_tail) ← (head, tail);
33 function rb_enque(R)
34     if (tail + 1) % ringBufSize ≠ head then
35         tail ← (tail + 1) % ringBufSize;
36         ringBuf[tail] ← R;
37         pwb(ringBuf[tail]);
38     else
39         while true do
40             head ← p_rb_head;
41             if (tail + 1) % ringBufSize = head then
                   pause else break;
```

Figure 3: Instrumentation State Machine

```
1  (b_head, b_tail) ← (p_rb_head, p_rb_tail);
2  while b_head ≠ b_tail do          /* Startup */
3  |  proc_redo_record(ringBuf[b_head]);
4  |  b_head ← (b_head + 1) % ringBufSize;
5  notify_ready(frontend);
6  while true do                     /* Main loop */
7  |  if b_head ≠ b_tail then
8  |  |  proc_redo_record(ringBuf[b_head]);
9  |  |  b_head ← (b_head + 1) % ringBufSize;
10 |  else
11 |  |  while true do
12 |  |  |  b_tail ← p_rb_tail;
13 |  |  |  if b_head = b_tail then pause else
   |  |  |    break;
14 function proc_redo_record(R)
15 |  if R = ⟨REDO, addr, blk⟩ then
16 |  |  memcpy(addr, blk, BLOCK_SIZE);
17 |  |  forall w ∈ [addr, addr + BLOCK_SIZE) do
   |  |    pwb(w);
18 |  else if R = ⟨MMAP, addr, size⟩ then
   |  |  // Map page-aligned pmem region
   |  |  // [addr, addr + size)
19 |  |  p_mmap(addr, size);
20 |  |  pfence();
21 |  else if R = ⟨COMMIT⟩ then
22 |  |  pfence();
23 |  |  p_rb_head ← b_head;
24 |  |  pwb(p_rb_head);
25 |  |  psync()
```

Figure 4: Backend thread

the best results for most of the workloads we considered.

Whenever an instruction to write a payload of size *size* to an address *addr* is intercepted by the instrumentation (line 3.4), it computes the addresses x_1 and x_2 of the first and the last BLOCK_SIZE-aligned blocks falling within the range $[addr, addr + size)$ (lines 3.6–7). It then inserts the addresses $x_1 + i \cdot \texttt{BLOCK\_SIZE}$ for all integers i such that $0 \le i \le (x_2 - x_1)/\texttt{BLOCK\_SIZE}$ into the hash table (lines 3.9–12) using linear probing to resolve collisions (lines 3.10–11)[2].

The hash table is statically allocated to fit into the CPU cache, and is not dynamically resized. Instead, whenever its load factor becomes too high for linear probing to work efficiently (60% in our implementation), the newly produced addresses are not stored in the hash table, but instead spilled into a separate overflow buffer. (This is not shown in the pseudocode in Figure 3 for clarity.)

[2]Note that since both BLOCK_SIZE and length(H) are typically powers of 2, the computations performed by the write handler can be implemented efficiently using bitwise operations.

The overflow buffer is implemented as a collection of dynamically allocated integer arrays each of which is cache line aligned and fits into the CPU cache. The addresses are added to the overflow buffer in the order of their generation, and are not deduplicated. Our experimental evaluation shows that most batches produced by the workloads we studied can be processed in full without using the overflow buffer thus maximizing the deduplication benefits.

Batched Redo Logging. The content of the persistent redo log is stored in a ring buffer ringBuf of size ringBufSize on NVM. New records are added to the redo log by the frontend and are removed (consumed) by the backend. The two end points are synchronized using persistent head (p_rb_head) and tail (p_rb_tail) indices stored on NVM. For efficiency, wherever possible, both frontend and backend use cached copies of these indices and defer their persistent updates until it is necessary for durability. Details of the log handling mechanism at both frontend and backend are discussed below.

Frontend redo log handling The logic of the redo log handling at the frontend is shown as a part of the instrumentation state machine pseudocode in Figure 3. The frontend caches the latest known values of p_rb_head and p_rb_tail in the shared variables rb_head and rb_tail, respectively. These variables are copied into thread-local variables `head` and `tail` by a combiner thread before it starts processing a new batch of read-write requests (line 3.3).

Once all the read-write requests in the current batch have been executed, the combiner calls end_tx (line 3.20) where it executes the following steps for each non-null address *A* stored in the deduplication hash table H (lines 3.22–26). First, it creates a REDO record *R* consisting of *A* and the block of data pointed by *A*, which it fetches directly from the main memory (line 3.24). It then executes the following logic to add *R* to the persistent log.

Since the log can be full, the addition requires extra care to synchronize with the backend (see function rb_enque in line 3.33). To this end, the combiner first attempts to use the cached copies of the head and tail indices to test whether the ring buffer entry at index $j = (\texttt{tail} + 1) \% \text{ringBufSize}$ is not occupied by `head` (line 3.34). If so, it copies *R* to that entry, persists it with pwb, and sets `tail` = *j* (lines 3.35–37). Note that `tail` does not need to be persisted at this point as no other thread, apart from the combiner, can use it to add records concurrently. If the entry at *j* is occupied, the combiner enters a busy-waiting loop (lines 3.39–41) where it loads p_rb_head into `head`, and checks whether it has advanced past *j*. If so, it breaks from the loop, persistently stores *R* at *j*, and assigns `tail` to *j* as above. Otherwise, it pauses for a short while, and then resumes the loop.

Once the combiner has finished processing the addresses in H, it proceeds to process those in the overflow buffer in the same fashion. (This is not shown in the pseudocode

for conciseness.) Once all addresses have been processed, it calls rb_enque to add and persist a COMMIT record in the log (line 3.27). It then calls pfence, copies the value of tail to p_rb_tail, and persists it with pwb (lines 3.28–30). This guarantees integrity of the ring buffer: i.e., *(i)* a COMMIT record always appears in the log immediately after all previously persisted REDO records, and *(ii)* the index stored in p_rb_tail always points to the entry immediately following the one occupied by a COMMIT record. The combiner then calls psync (line 3.31) to ensure all previously issued persistent stores have been made durable thus enabling the read-write requests in the current batch to return to their clients. Finally, it copies head and tail back to rb_head and rb_tail (line 3.32) to make them available for the next combiner.

Backend redo log handling The pseudocode of the redo log handling at the backend is shown in Figure 4. It is executed by a single dedicated thread in a separate OS process as explained in §3.2.

As it is the case for the frontend, the backend caches the persistent head and tail indices in thread-local variables b_head and b_tail, respectively (line 4.1) before it starts processing the log. It then proceeds to execute the following logic in an infinite loop.

First, if the ring buffer is not empty (i.e., b_head ≠ b_tail) (line 4.7), the backend reads the record R stored at head and inspects its type. If R is a REDO record with an address A and payload D, it copies D to the address A on NVM, and persists the words spanned by D with pwb (lines 4.16–17). If R is either mmap or munmap record (the latter is not shown in the pseudocode for conciseness), it either allocates or deallocates persistent memory as detailed in §4.4.

Finally, if R is a COMMIT record, the backend reads the record R stored at head and inspects its type. If R executes pfence, copies head to p_rb_head, and persists it with pwb (lines 4.22–25). This ensures that p_rb_head is consistent with the log entries that have been processed. The backend thread reads the record R stored at head and inspects its type. If R then calls psync to flush all previously issued persistent stores to NVM. Note that although in principle, this psync can be replaced with pfence, using psync allows the backend to keep up with the frontend in terms of its persistence granularity. This ensures that under optimal load, the ring buffer is emptied at roughly the same rate as new records are produced thus minimizing frontend blocking and the system recovery time.

Once a record has been processed the backend reads the record R stored at head and inspects its type. If R increments head modulo ringBufSize, and proceeds to handle the next record (lines 4.9).

If the ring buffer is empty, the backend reads the record R stored at head and inspects its type. If R enters a busy-waiting loop (lines 4.11) where it copies p_rb_tail into tail, and checks whether b_head is still equal b_tail. If so, it pauses for a short while, and resumes the loop. Otherwise, it breaks from the loop and handles the next record on the log as above.

Persistence cost. Let F and S denote the costs of pfence and psync respectively, and B be the average number of requests occupying a single flat combining batch. The total cost of persistence at the frontend is then $F+S$ per batch or $(F+S)/B$ per request on average. Likewise, the total cost of persistence at the backend is $F+S$ per batch or the average per-request of $(F+S)/B$. The above indicates that the backend is able to process requests at the same rate as the frontend, and therefore, unlike [50], is not a bottleneck.

4.4 Split Allocation

Memory allocators are a challenge when porting applications to persistent memory since any modifications to allocator metadata must also be persisted. However, for many applications the memory allocator has been chosen carefully by developers [38]. Forcing them to switch to a custom persistent allocator [3] violates Mangosteen's transparency goals. Furthermore, persisting every modification to an application's address space is inefficient, since in many cases the corresponding data is *transient* and does not need to be persisted to ensure correct recovery (i.e. durable linearizability). For example, in Redis the contents of incoming request buffers do not need to be persisted.

To overcome these challenges, we propose a novel *split allocation* scheme that allows applications to reuse their existing allocator while minimizing unnecessary persistence of transient data. During application initialization, the Mangosteen framework loads an additional instance of the application's memory allocator using dlmopen (e.g. jemalloc [25] in the case of Redis). We refer to the application's original allocator as the *persistent allocator* and the additional allocator as the *transient allocator*. After application initialization completes, we use binary instrumentation to intercept all memory allocator functions (e.g., malloc, free). Memory allocator function calls that occur within processRequest during execution of a RW command are directed to the persistent allocator. Other memory allocator operations, e.g. those that occur during calls to init/destroyAppClient, deserializeRequest or processRequest for RO commands, are considered transient and directed to the transient allocator.

The combiner is permitted to access and modify objects allocated using the transient allocator (e.g., the incoming request buffer). However, any modifications to transient objects that occur within processRequest for RW commands are intercepted by Mangosteen's binary instrumentation. This allows them to be filtered based on the virtual memory address ranges assigned to the transient allocator by the OS (e.g. in response to calls to mmap/munmap, see Fig. 3, line 13). Empirically, due to additional instrumentation overhead on the critical path we

found it more expensive to filter transient writes from RW commands at the front-end than to persist them in the redo log and perform filtering at the backend. To enable the back-end to filter transient writes, the Mangosteen front-end therefore adds redo log entries for any internal calls to `mmap/munmap` made by the transient allocator. Such calls occur infrequently in practice. We emphasize that only transient writes that occur within `processRequest` for RW commands are added to the redo log, with all other transient writes incurring no persistence overhead.

5 Correctness

Given the subtle interaction between request handler threads, we have developed a TLA+ model [28] of the concurrency control algorithm in Figure 2 to validate its safety and liveness properties. For safety, we have established the following invariant:

$$Mutex = \forall i, j \in TID.\ i \neq j \wedge \mathsf{pc}[i] \in WR \Rightarrow \mathsf{pc}[j] \notin WR \cup RD$$

where $WR = [2.26 \ldots 2.29]$ and $RD = \{2.6, 2.7\}$ correspond to program counter values for read-write and read-only request handling, respectively, and $\mathsf{pc}[i]$ is a program counter of thread i. We use 2.26 as notation for line 26 of Fig. 2. The *Mutex* invariant ensures mutual exclusion between the current combiner and the other threads: i.e., at all times, the current combiner is the only thread that can execute the code in lines 26–29 of Figure 2.

The following theorem (proved in Appendix B) establishes Mangosteen's main safety guarantee. It is very similar to the property guaranteed by libraries such as FLiT [4, 45], but the changes to the underlying implementation that Mangosteen requires are much less invasive (see §2).

Theorem 1. *Given an implementation* **I** *of an application, let* $M[\mathbf{I}]$ *denote the Mangosteen-enhanced version of* **I**. *Suppose* **I** *is an in-memory linearizable implementation of an object with a sequential specification* **S**. *Then* $M[\mathbf{I}]$ *is durably linearizable wrt* **S**.

Additionally, in the absence of crashes, we have proved the following property in TLA+.

$$Prog = \Box(\forall i \in TID.\ \mathsf{status}[i] = \texttt{READY} \Rightarrow \Diamond(\mathsf{pc}[i] = 2.31))$$

which asserts that no read-write operation is starved. The following theorem establishes Mangosteen's liveness guarantee.

Theorem 2. *If* **I** *is an application program that is deadlock-free, then* $M[\mathbf{I}]$ *is deadlock-free in the absence of crashes.*

Note that, for liveness, we restrict attention to crash-free histories because progress properties in the presence of crashes have not yet been well-defined, and there are many options for how to characterise progress after recovery. We therefore leave a full investigation of Mangosteen's progress guarantees as a topic for future work.

6 Evaluation

In this section, we evaluate Mangosteen's performance and show that it achieves high end-to-end throughput and low latency for different update ratios and request sizes and scales well across multiple cores.

6.1 Experiment Setup

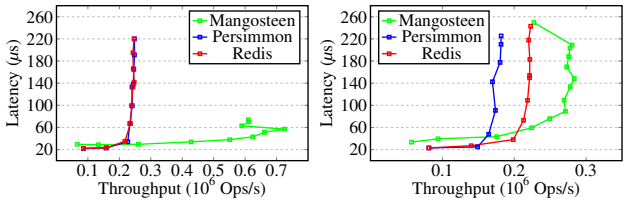
Our evaluation testbed uses a dual socket Intel Ice Lake server where each server CPU is an Intel Xeon Gold 6236 2.9 GHz with 16 physical cores per CPU for a total of 32 physical cores (64 hyperthreaded). The server is equipped with 128GB GB of DRAM and 1TB (8×128GB DIMMs) of Intel Optane memory. Our server runs Ubuntu 20.04 with Linux kernel version 5.15.0. For Redis, the machine used to generate the client workload has the same specification as the server. Both machines are equipped with 25 Gb/s Mellanox ConnectX-5 NICs and are connected using a Dell EMC 32×100GbE switch.

6.2 Redis Performance

We first evaluate a Mangosteen-enabled version of Redis using the YCSB benchmark [11]. In line with prior work [50] and the YCSB defaults we use a closed loop client without Redis pipelining, represent each record with a Redis hash and access fields using Redis' `HSET/HGET` commands. We load 13 million records (the YCSB default), each record has 10 fields (i.e. 130 million items in total) and each field has a 100B value. The client issues reads and updates at a fixed ratio and chooses which records to access according to a Zipfian distribution. For our YCSB experiments we disable turboboost and hyperthreading on the server. As a baseline we compare against Persimmon [50]. We also include vanilla in-memory Redis as an indicative, more challenging baseline than Redis with append-only-file (AOF) persistence on NVM which is approximately 3× slower. For Mangosteen and Persimmon we pin the backend to a specific core. We use Redis version 4.0.9 (as in [50]) and the Redis server uses jemalloc (as recommended for Linux).

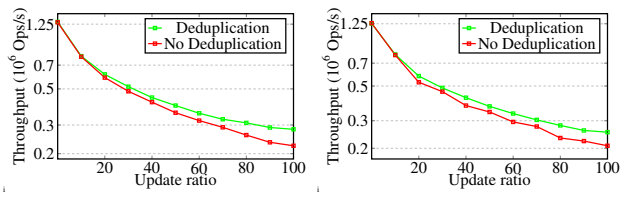
End-to-end performance. We start with end-to-end performance for typical read and write-intensive YCSB workloads with update ratios of 10% and 90% and a Zipfian constant of 0.99. We measure the latency and throughput of Redis, Mangosteen Redis and Persimmon Redis. Fig. 5 shows the results with data points for 2,4,8,16,...,88,96 clients.

For read-intensive workloads (Fig. 5a), Mangosteen's peak throughput is approximately 2.9× that of both Persimmon and in-memory Redis due to its ability to execute read-only requests concurrently. Mangosteen's latency also remains low until the number of clients equals the number of cores (32). Mangosteen's throughput continues to grow up to 48 clients as the cores on average remain underutilized, although latency



(a) 10% updates (b) 90% updates

Figure 5: Latency vs throughput for read (10%) and write (90%) intensive workloads



(a) Zipfian (b) Uniform

Figure 7: Deduplication vs no deduplication for Zipfian and Uniform workloads

increases gradually due to oversubscription. Beyond this point the server becomes saturated.

For write-intensive workloads (Fig. 5b), Mangosteen's peak throughput is still substantially higher than Persimmon and Redis (1.6× and 1.3× respectively), although the gap closes due to the reduction in reader-reader concurrency. However, Mangosteen's latency is slightly higher than Redis and Persimmon because of write batching.

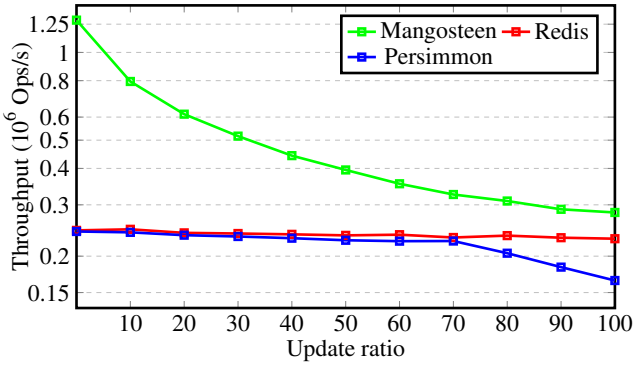


Figure 6: Peak throughput vs update ratio

Peak throughput vs update ratio. As Mangosteen only accesses NVM for read-write operations, its performance depends on the workload's read-only to read-write ratio. Figure 6 shows how peak throughput changes as we vary this ratio. Mangosteen outperforms Persimmon for all update ratios by between 2×–5×. Furthermore, Mangosteen's throughput degrades gracefully and does not become bottlenecked by its backend, demonstrating the effectiveness of its persistence optimizations (e.g. write deduplication). In contrast, for write-intensive workloads Persimmon's shadow execution with undo logging becomes the bottleneck at an update ratio of between 70–80%. In comparison to Redis, Mangosteen also achieves higher peak throughput even for an update-only workload because it is still able to parallelize some transient data processing (e.g. request parsing in `deserializeRequest`).

Deduplication effectiveness Deduplication masks the cost of persistent memory operations by coalescing write instructions that have adjacent, overlapping or duplicated address ranges. Since skewed workloads are more likely to have redundant updates, we evaluate the benefits of deduplication for both Zipfian and uniform variants of our YCSB workload (Fig. 7).

For both sets of experiments, the throughput gains from deduplication grow as the workload becomes more update intensive. For an update ratio of 100%, deduplication increases throughput by 26% for a skewed (Zipfian) workload, and 21% for a uniform workload. This indicates that deduplication is slightly more effective for skewed workloads, but is beneficial regardless of the workload skew. We speculate this is because a non-negligible fraction of the the memory regions accessed by different requests are independent of the request keys.

6.3 LevelDB Performance

We next evaluate Mangosteen using the LevelDB benchmark suite. Similar to [12], we use Mangosteen to implement a persistent key-value store compatible with the LevelDB API. Due to the nature of the LevelDB benchmark we configure Mangosteen in local mode such that no separate client machine is required and each thread generates its own workload. As a baseline we use RomulusDB [12], a persistent key-value store that implements the LevelDB interface and outperforms vanilla LevelDB for all benchmarks in the LevelDB benchmark suite. By default all LevelDB benchmarks use 16-byte keys and 100-byte values. Our LevelDB evaluation uses the same Intel Ice Lake server as our Redis benchmarks.

Update-only workloads. We first explore update workloads. The `fillSeq` benchmark measure the time for a thread to insert one million distinct key-value pairs in the database using sequential keys. `fillRandom` performs one million insertions of random keys per thread. `overwrite` is similar to `fillRandom` but starts with a pre-populated database. `fill100k` measures how long it takes to write 1000 large key-value pairs of 100 kB.

Fig. 8 shows the results. For small update workloads (`fillSeq`, `fillRandom`, `overwrite`) Mangosteen scales better than Romulus independent of the access pattern, achieving between 5.5×–6.5× lower latency per operation for 32 cores. The `fill100K` benchmark illustrates the importance of Mangosteen's `memcpy` instrumentation for large objects (Fig. 8d). With `memcpy` instrumentation enabled (`Mangosteen`), Mangosteen outperforms Romulus by

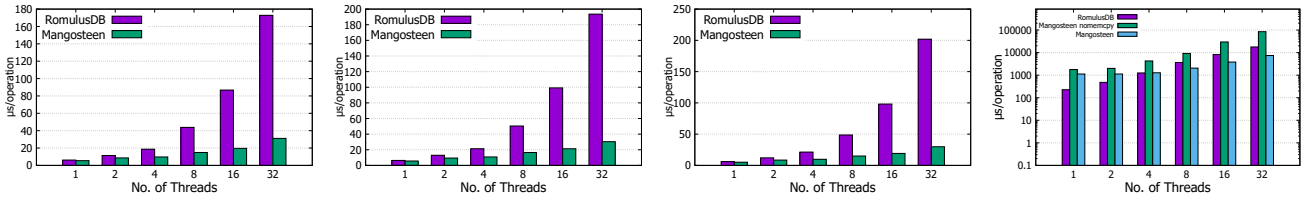


Figure 8: LevelDB Update Workloads

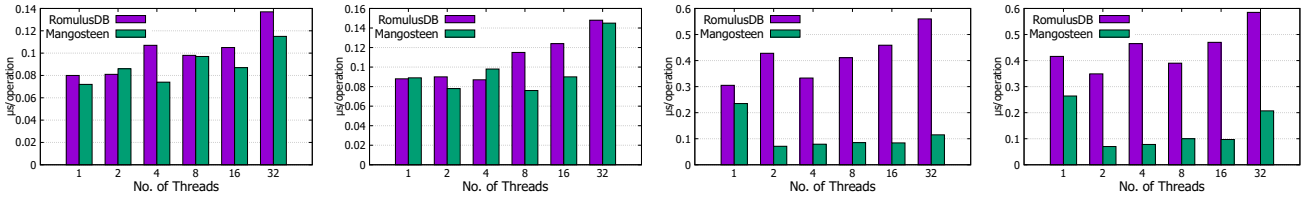


Figure 9: LevelDB Read Workloads

2.4× for 32 cores (note the log scale). Here for each value written Mangosteen requires only a single clean-call and a single sequential write of the associated memory range to the redo log (§4.2). In contrast, with `memcpy` instrumentation disabled (`Mangosteen nomemcpy`), Mangosteen's binary instrumentation must intercept each individual write instruction and performance plummets.

Read-only workloads. For read-only LevelDB workloads, `readSeq` and `readReverse` do a single read-only iteration over the database. `readSeq1KB` and `readReverse1k` are identical but use larger key-value pairs of size 1KB. Mangosteen has slightly better performance than RomulusDB for `readSeq` since Mangosteen reads from DRAM which is faster than NVM (Fig. 9). For `readReverse` the difference is less pronounced since more reads are cache resident, masking the overhead of reading from NVM for RomulusDB. The negative impact of reading from NVM is further highlighted by the `readSeq1KB` and `readReverse1KB` benchmarks, where Mangosteen outperforms Romulus by 5× and 2.8× for 32 cores.

6.4 Recovery

In Mangosteen, the cost of applying entries in the redo log during recovery is negligible since the redo log is typically short and re-execution of commands is not required (unlike [50]). Instead, the cost of recovery is dominated by the overhead of copying the recovered persistent memory regions from NVM to the front-end's DRAM copy. For our YCSB experiments with a 22GB state size, recovery using a single thread takes approximately 10 seconds. A basic implementation of multi-threaded recovery that transfers different persistent regions in parallel reduces this to approximately 3 seconds. We leave further optimization of this mechanism as a subject for future work.

7 Related Work

Romulus offers a *persistent transactional memory* (PTM) programming model that uses language interposition to instrument stores [12], maintaining two copies of the application state in persistent memory. Like Mangosteen, Romulus integrates flat combining with reader-write locks to manage concurrency, and shares the design goals of minimizing developer effort and reducing the number of fences required to persist updates. However, unlike Mangosteen (which uses redo logging), Romulus uses copy-on-write, and records the address and range of each write in an in-memory (non-persistent) array to supports deduplication. Redo logging enables Mangosteen's read commands to execute over DRAM, which offers higher read bandwidth than NVM, while Mangosteen's hash-set based deduplication is faster than recording ranges in an in-memory array for the workloads we evaluated. Finally, Romulus requires more extensive allocator modifications compared to Mangosteen.

Mnemosyne [44] is a PTM that combines redo logging with an existing STM called TinySTM [15]. Like Mangosteen's use of DBI, Mnemosyne employs compiler instrumentation to automatically identify write instructions within transactions that need to be persisted. However, Mnemosyne performs out-of-place logging of writes and so also needs to instrument read instructions, increasing overhead. Moreover, Mnemosyne employs eager conflict detection which allows individual transactions to abort (e.g. on failure to acquire a lock), whereas Mangosteen currently does not permit individual commands

to abort.

Atlas leverages FASEs to ensure crash consistency for lock-based programs [8]. As with Mangosteen, Atlas employs a *write-ahead log* and interposes on stores. However, unlike Mangosteen, Atlas interposes synchronization operations and employs undo logging. This means that log entries must be made persistent before performing an in-place update, which in turn implies a fence instruction per log entry.

JustDo logging leverages persistent CPU caches and FASEs in lock based programs with non-abortable transactions [23]. FASEs execute directly on NVM, and each FASE maintains a record of only the most recent write instruction. On recovery, execution simply resumes after the last write instruction within a FASE. In contrast, Mangosteen does not assume CPU caches are persistent.

NV-heaps uses undo logging to support transactions over persistent memory heaps [10]. It provides a more general transaction model than Mangosteen since individual transactions can abort. In addition, it provides additional safety features including garbage collection for persistent memory objects. In general, it targets a different programming model than Mangosteen (persistent memory objects). Its approach to undo logging potentially suffers from write amplification since complete objects are logged even when only a subset of fields are accessed in order to offset the cost of fences/epoch barriers needed for undo logging.

NV-Traverse [16], Mirror [17], and FliT [45] are generic techniques for transforming lock-free concurrent data structures to persistent data structures. NV-Traverse executes writes directly on NVM, requiring fences after each write operation, but minimizes the number of flushes and fences for reads. Mirror improves performance over NV-Traverse by keeping a separate copy of the data structure in DRAM such that reads need never access NVM. Finally, FliT performs optimized tracking of unpersisted writes using pwb's. However, all three libraries require significant changes to the original program whereby *every* read/write/CAS operation of the application program is replaced by a call to the respective library. This is unlike Mangosteen, which only requires instantiation of a simple API (see §3.1). Montage [46] proposes another system for making legacy applications durable, but drops durable linearizability in favor of a weaker condition: buffered durable linearizability [24].

Pronto [33] is an NVM-aware library for transforming sequential and lock-based concurrent data structures to persistent data structures. With respect to transparency, Mangosteen is general enough to support complex applications that employ a mixture of multiple persistent data structures and shared global variables, whereas Pronto targets individual encapsulated/object-oriented data structures (e.g. class instances). Unlike Mangosteen's update redo logging, Pronto uses command logging, allowing it to overlap command execution with persist operations using dedicated background logging threads. Depending on the workload, command logging can also consume less write bandwidth than update logging, although Mangosteen mitigates this using deduplication. On the other hand, command logging implies that Pronto cannot correctly support non-deterministic operations (e.g. external network calls that may timeout) as those may produce different results when re-executed upon recovery. Pronto also requires periodic snapshotting to bound the size of the command log. It currently only supports full snapshots, which does not scale to large data store sizes, but even incremental snapshots will suffer from write amplification for workloads with fine-grained writes assuming tracking is done at page granularity. Finally, to guard against crashes occurring whilst snapshotting is in progress, a new snapshot cannot overwrite the previously created one. This effectively doubles the NVM requirements of Pronto compared to Mangosteen.

8 Conclusions

We presented Mangosteen, an easy-to-use programming framework to enable linearizable in-memory applications to gain durability using NVM. Mangosteen allows read-only requests to execute in parallel while batching and deduplicating read-write requests to minimize the cost of persistence. It enables applications to use their existing memory allocators unmodified, and employs a novel split allocation scheme to avoid unnecessary overhead for updates to transient memory that do not need to be persisted for correct recovery. The performance evaluation demonstrates Mangosteen-enabled applications are able to achieve significant throughput improvements on realistic workloads compared to the state-of-the-art persistence frameworks.

9 Acknowledgements

We thank the anonymous reviewers for their insightful suggestions. We are grateful to Ymir Vigfusson for his invaluable feedback on the early versions of Mangosteen, and University of Surrey Institute of Advanced Studies for supporting his visit. This work was partially supported by the CHIST-ERA grant CHIST-ERA-22-SPiDDS-05 (REDONDA project) and the UK Engineering and Physical Sciences Research Council (EPSRC) (grant numbers EP/Y036417/1 and EP/Y036425/1). Dongol and Chockler are both supported by EPSRC grants EP/X037142/1 and EP/X015149/1, and Dongol is additionally supported by VeTSS and EPSRC grants EP/V038915/1 and EP/R025134/2.

References

[1] Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions, 2022.

[2] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694, oct 2016.

[4] Stefan Bodenmüller, John Derrick, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. A fully verified persistency library. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 26–47, Cham, 2024. Springer Nature Switzerland.

[5] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, page 133–144, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. *SIGPLAN Not.*, 48(8):157–166, feb 2013.

[7] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004, Boston Marriott Copley Place, Boston, MA, USA*, pages 15–28. USENIX, 2004.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.*, 49(10):433–452, oct 2014.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[13] CRIU. https://criu.org.

[14] DynamoRIO Clean Calls. dynamorio.org/API_BT.html#sec_clean_call.

[15] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 237–246, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 377–392, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1218–1232, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, page 400–407, USA, 2004. IEEE Computer Society.

[19] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[21] Maurice Herlihy and Nir Shavit. On the nature of progress. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 2011.

[22] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[23] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *SIGARCH Comput. Archit. News*, 44(2):427–442, mar 2016.

[24] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[25] jemalloc. http://jemalloc.net.

[26] Artem Khyzha and Ori Lahav. Taming x86-tso persistency. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[27] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[28] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[29] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[30] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 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*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, page 4, USA, 2017. USENIX Association.

[33] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 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, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

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

[35] Persistent Memory Development Kit. pmem.io/pmdk.

[36] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA):137:1–137:27, 2018.

[37] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL):11:1–11:31, 2020.

[38] redis/readme.md at 4.0. https://github.com/redis/redis/blob/4.0/README.md.

[39] Samsung Electronics. Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit, 2022.

[40] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. APress, 2020.

[41] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 44(2):35–40, 2015.

[42] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[43] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[44] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[45] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: A library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 309–321, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen, editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 73:1–73:11. ACM, 2021.

[47] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[50] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent state machines for recoverable in-memory storage systems with nvram. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

A Additional Background on Persistent Memory

Example 1. *To demonstrate the behavior of* pwb, psync *and* pfence *instructions, consider the programs below. Assume that x and y are both initially* 0*, and that a crash occurs* at some point *during the program's execution.*

```
1 Prog1          1 Prog2          1 Prog3          1 Prog4
2   x ← 1;       2   x ← 1;       2   x ← 1;       2   x ← 1;
3   pwb(x);      3   psync();     3   pwb(x);      3   pwb(x);
4   y ← 1;       4   y ← 1;       4   psync();     4   pfence();
                                  5   y ← 1;       5   y ← 1;
```

In both Prog1 and Prog2, after the crash, it is possible to have $x = 0$ *and* $y = 1$ *in NVM since the writes to x and y may not be persisted in the order that they are executed. In particular, in Prog1, the write to x is tagged by a* pwb *but not synced before the write to y, and in Prog2, the write to x is not tagged before the* psync() *is executed. This behaviour is impossible in both Prog3 and Prog4, i.e., regardless of when the crash occurs, if* $y = 1$ *in NVM, then* $x = 1$*. The difference between Prog3 and Prog4 is that in Prog3,* $x \leftarrow 1$ *is guaranteed to be persisted when the program reaches line 5, whereas in Prog4, this is not necessarily true.*

B Mangosteen Safety and Liveness Argument

Theorem 1. *Given an implementation* **I** *of an application, let* $M[\mathbf{I}]$ *denote the Mangosteen-enhanced version of* **I***. Suppose* **I** *is an in-memory linearizable implementation of an object with a sequential specification* **S***. Then* $M[\mathbf{I}]$ *is durably linearizable wrt* **S***.*

Proof (sketch). First, we note that every history of $M[\mathbf{I}]$ is of the form $\sigma_1 \cdot C_1 \cdot \sigma_2 \cdot C_2 \ldots$ where each σ_i is a crash-free *era* and C_i is a crash.

We now consider each era. The main synchronization between concurrent client requests is handled by the ClientCmd program in Fig. 2. Given the *Mutex* property for ClientCmd described above, a read-only operation that executes concurrently with a combiner operation is either linearized before or after the (batch of) read-write operations executed by the combiner. Similarly, in any trace involving the concurrent execution of two different combiner threads i and j, the batch (of read-write operations) executed by i either occurs before or after the batch executed by j. Finally, read-write operations that are part of the same batch are executed *sequentially* by the combiner thread. Thus every era σ_i is consistent with a history of the form: $RO_1 \cdot B_1 \cdot RO_2 \cdot B_2 \ldots$, where RO_i is a (possibly empty) sequence of read-only operations and B_i is a sequence of read-write operations.

Therefore, the only remaining properties to check are regarding failure atomicity, i.e., **(i)** when a combiner thread completes, all read-write operations executed by the combiner are persisted, and **(ii)** if a combiner is interrupted by a crash before it completes, either all writes performed by the combiner are rolled back (and the corresponding read-write operations are cancelled) or the writes are persisted on behalf of the combiner by the recovery operation that is executed immediately after the crash (so that the corresponding read--write operations are committed). It is straightforward to see that both of these hold via the mechanisms already described in §4.3. □

Theorem 2. *If* $\mathbf{I}$ *is an application program that is deadlock-free, then* $M[\mathbf{I}]$ *is deadlock-free in the absence of crashes.*

Proof (sketch). Every execution of $M[\mathbf{I}]$ is of the form $RO_1 \cdot RW_1 \cdot RO_2 \ldots$ or $RW_1 \cdot RO_1 \cdot RW_2 \ldots$ where each RO_i contains no concurrent read-write operations and RW_i contains some concurrent read-write operation. We show that each RO_i and RW_i segment is deadlock-free.

RO_i. Each RO_i segment is trivially deadlock-free since $\mathbf{I}$ is deadlock-free.

RW_i. By *Prog* from §5, each waiting writer is guaranteed to complete its operation. Informally, this holds because the combiner sweeps through the `status` array and executes the operation corresponding to each waiting writer sequentially. Since $\mathbf{I}$ is deadlock-free, the sequential execution of each operation of $\mathbf{I}$ must terminate. Hence, in $M[\mathbf{I}]$, each writing operation executed by the combiner terminates.

□

C Artifact Appendix

Abstract

The Mangosteen artifact is a collection of applications integrated with the Mangosteen framework.

Scope

The artifact enables the reader to run several example applications that have been integrated with Mangosteen and review the Mangosteen source code. If the reader has access to an experimental setup with persistent memory like that described in §6.1 it should also be possible to reproduce the performance evaluation of Redis integrated with Mangosteen in §6.2. In cases where the reader has access to real or emulated persistent memory, it should also be possible to test application recovery using the instructions provided in the README.txt file that accompanies the artifact.

Contents

The artifact contains Mangosteen source code as well as external dependencies that are required to correctly build and run Mangosteen. The artifact contains a detailed README.txt file that describes how to build Mangosteen on Ubuntu using the supplied build scripts.

The artifact contains the following application examples:

- Simple key/value store: Similar to the example in §3.1. It uses Mangosteen's local API.
- Client/server: A simple client/server application that accepts requests of the form add(byte_array) and adds the argument to an in-memory array (persisted via Mangosteen).
- Redis: The Redis database integrated with Mangosteen for persistence. This is our main case study in the paper.

Hosting

The artifact can be obtained from Zenodo using the unique DOI 10.5281/zenodo.11390432