הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

RELAX: Recovering Lazily From Failed Execution With Persistent Memory

Almog Zur

RELAX: Recovering Lazily From Failed Execution With Persistent Memory

Research Thesis in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Almog Zur

Submitted to the Senate of the Technion — Israel Institute of Technology
Adar, 5783, Haifa, March, 2023

This Research Thesis was Done under the Supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

The author of this thesis states that the research, including the collection, processing and presentation of data, addressing and comparing to previous research, etc., was done entirely in an honest way, as expected from scientific research that is conducted according to the ethical standards of the academic world. Also, reporting the research and its results in this thesis was done in an honest and complete manner, according to the same standards.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

List of Tables

1 Abstract

Recent non-volatile main memory technology (such as Intel's Optane) gave rise to an abundance of research on building persistent data structures, whose content can be recovered after a system crash. Such data structures employ infrequent (yet, costly) flush instructions to write back crucial data from the cache to the non-volatile memory. The main focus of previous work has been to minimize the overhead incurred by maintaining a consistent state in the non-volatile memory. In this thesis we focus on the next important goal in this domain: shortening recovery time. We start with the Mirror construction, which is one of the most effective general construction for efficient persistent lock-free data structures available today, and we extend it with a lazy recovery procedure. The resulting system has almost zero recovery time, with an overhead that quickly descends following a crash event. We implemented the proposed methodology on a hash table, a skip list, a binary tree, and a linked list. The evaluation shows that following a crash, even large data sets with millions of nodes become responsive immediately. The overhead of lazy recovery slows the structure for a couple of seconds until it regains maximal performance. We show that lazy recovery is especially efficient for hierarchical data structures such as trees and imbalanced key distributions such as zipfian.

Abbreviations

NVMM	Non Volatile Main Memory
DRAM	Dynamic Random Access Memory
CAS	Compare And Swap
DWCAS	Double Width Compare And Swap
SMR	Safe Memory Reclamation

Notations and Units

2 Introduction

Durable data structures form building blocks for constructing reliable systems, but their durability usually comes with a performance overhead. To improve efficiency, many storage systems employ in-memory data structures, which alleviate the secondary storage performance bottleneck. With the advent of non-volatile main memories (NVMM), which offer durable and byte-addressable access, at speeds comparable to main memory (DRAM), a new era of *durable* in-memory data structures has emerged [15, 19–21, 26, 42, 54]. Such durable data structures obtain performance that is much closer to the speed of in-memory volatile data structures than to traditional storage systems, allowing durability at a much lower cost.

Designing durable data structures is non-trivial, because the cache is still volatile in current systems, and their content is lost upon a crash. During execution, the hardware regularly evicts cache lines in an unpredictable order which may leave the NVMM in an arbitrarily inconsistent state; some updates may already be written-back and others may still be volatile and can potentially be lost in case of a system failure. Special write-back instructions are provided by the hardware, enabling the programmer to control the order of data movement from the cache to the memory, but these instructions bear a high cost. Designing algorithms with a minimal number of flush executions turns out to be difficult and error-prone. This made general constructions, which allow the user to easily transform a data structure into a durable data structure attractive.

To transform volatile data structures into durable data structures new techniques were developed. One such technique is logging the actions of the data structure, allowing for the recovery of lost actions. Another technique is fine-grained durability control, minimizing the use of the aforementioned write-back instructions as much as possible while still guaranteeing correctness. Furthermore, the use of NVMM and DRAM simultaneously as done in Mirror [21] further reduces the overhead by frequently accessing DRAM, which is currently still faster than NVMM. This allows Mirror, a general construction, to compete with hand-crafted persistent data structures.

While there has been significant progress in making durable data structures efficient, shortening the length of the recovery phase, i.e., the time interval after a crash in which the data structure cannot execute operations, has not received much attention. In fact, a trade-off has implicitly been placed between obtaining high-performance and achieving a fast recovery after a crash. Three avenues are typically used to build durable data structures: ad-hoc constructions, durable transactions, and general transformations. An ad-hoc durable data structure is built for a specific data structure, such as a linked-list or a skip-list. Significant wisdom is exercised to obtain an efficient structure with a minimal number of

persist instructions. Zuriel et al.'s construction [58] of set data structures is arguably the most efficient today, but it suffers from a long recovery, whereas the slower durable data structures of David et al. [15] obtains faster recovery time. As ad-hoc data structures are hard to get right for non-experts, programmers sometimes turn to transactions or general transformations. Durable transactions are known to yield low performance, but provide moderate recovery time. The recovery time usually does not depend linearly on the size of the data structure, but only on the size of a log of recorded modifications, which is typically of moderate size. Finally, general transformations, which are the focus of this thesis present a similar trade-off. General transformations take a non-durable concurrent data structure and generate a similar data structure (with the same operations) that is durable. The simplest and least efficient general transformation of Izraelevitz et al. [33] generates durably linearizable data structures by inserting flush and fence instructions at every load and update. This yields data structures with low performance, but very short recovery time. Advanced transformations like NVTraverse [19] and TIPS [37] incur a long recovery in which one needs to traverse the data structure or a substantial log before execution can resume. A state-of-the-art general transformation that generates highly efficient durable data structures is the Mirror transformation [21], yet data structures generated by Mirror require a long recovery time, which is linear in the size of the data structure.

A failure accompanied by a long recovery time, may render data inaccessible for a long period, defeating the purpose of storing the data in durable storage in the first place. Mirror [21] obtains high performance by storing a duplicate of the data structure on the DRAM, where loads are executed at high speed. Unfortunately, Mirror needs to copy the entire data structure from NVMM to DRAM during recovery, causing a long recovery, and inaccessible data following a crash.

In this thesis, we present RELAX (REcovering LAzily from failed eXecutions). RELAX extends the Mirror transformation to generate data structures that provide the best of both worlds. On one hand, the generated data structures enjoy performance that is almost equivalent to the one obtained by the original Mirror transformation. On the other hand, the generated data structures are able to execute almost immediately after a crash. This is achieved by allowing recovery to run lazily, concurrently with standard data structure operations. Lazy recovery allows immediate access to the data post-crash, at the cost of lengthening the recovery process. During RELAX's recovery, while the necessary information is being concurrently copied from NVMM to DRAM, the data structure performs at a reduced performance. Once all nodes are recovered, the data structure returns to full speed.

Mirror provides functionality by extending the `std::atomic` type of the standard C++ library with the `patomic` type. The simplest version of RELAX offers a fully general version of the `patomic` type, similarly to Mirror. However, this simple general method incurs a non-trivial overhead on normal (non-crashing) execution, as shown in the evaluation. Therefore, we study optimizations that fit standard operating systems and widely used data structures that speed the generated data structures and obtain the desired performance. For example, when generating a recursive data structure (such as a skip-list or a binary search tree) on standard operating systems (such as Linux or Windows), RELAX can reduce recovery time to a few milliseconds with performance almost equal to Mirror with an overhead of $0-5\%$. All versions of RELAX are presented in section 6.2

Like Mirror, RELAX can be applied to all lock-free data structures to create durable ones. We believe that similar techniques can be applied to other transformations, but much care must be put into performance and correctness. Extending the recovery design to be lazy and concurrent with program operations is non-trivial. Since recovery operates concurrently with the data structure operations, in a lock-free manner, extra care is needed to ensure both high performance and correctness. This is true especially for Mirror, where two consistent copies of the data structure are maintained, one in DRAM and one in NVMM,

as in Mirror.

Contribution And Organization

The main contribution in this thesis is in obtaining the best of both worlds: RELAX generates durable data structures with both low durability overhead, and a short recovery time. The technical challenge is due to concurrency: concurrent lazy recovery and concurrent executing threads need to execute concurrently correctly, while aiming at high performance. Finally, we provide multiple solutions according to the various guarantees available with existing operating systems, and according to the shape of the target data structures.

The rest of this thesis is organized as follows. Chapter 3 discusses the setting, presents correctness definitions. Chapter 4 presents an overview of the Mirror algorithm, which is integral to an understanding of RELAX. Chapter 5 discusses related work. Chapter 6 provides a detailed description of the RELAX construction and its correctness is argued in Chapter 7. The experimental evaluation for different data structures is presented in Chapter 8. Finally, Chapter 9 concludes.

3 Preliminaries and System Model

In this chapter, we define our assumptions about the underlying hardware and discuss related assumptions.

3.1 Persistent Memory

We consider a system of n processes $p_1, p_2, ..., p_n$ acting asynchronously with a shared memory system. This memory system is composed of three parts: (a) a volatile DRAM (b) a non-volatile main memory (NVMM) which is somewhat slower than the DRAM and (c) a volatile cache which is much faster than the DRAM, and stores recent accesses from the DRAM and NVMM.

Each process may access the DRAM or NVMM in byte granularity, which first goes through the cache. A write may stay in the cache for an unknown period of time, before being implicitly written back to the DRAM or NVMM. A process may also decide to explicitly force the cache to transfer a particular write to the DRAM or NVMM. After a crash, only the memory in the NVMM remains valid, while both the DRAM and caches are lost. For the sake of generality we assume the content of the DRAM after a crash is arbitrary.

Like Mirror, we use the Px86 persistency model [48] with the full-system crash model by Izraelevitz [33]. The persistency model does not impact the proof of RELAX's extensions, because RELAX's extensions operate between crashes without any persistence instructions, as any action of RELAX need not persist after a crash.

3.2 Explicit Write-Backs

A process can force an explicit write-back using a combination of two instructions. The first is *flush* which instructs the cache to evict a particular address to memory. This instruction however, is non blocking in its most efficient implementation, which means that a write followed by a *flush* might not be immediately persisted. To address this issue a *fence* instruction needs to be executed. The fence instruction guarantees write and flush operations cannot be reordered beyond the fence, i.e., no further instructions can become visible to other threads before write-back to non-volatile memory is completed. The specific instructions for each processor can be found in the respective manual (Intel [29,30], AMD [1], ARM [2]), see further details in [49, 50].

3.3 Durable Linearizability

We consider programs that execute data structure operations. Every operation's invocation and response are considered events which are related to the calling process. An *invocation* happens just before the first instruction is executed, and a *response* occurs just after the final instruction of the operation is completed. A crash is another event, that is not associated with any particular process, and resets all volatile memory. During a crash event the persistent memory is not affected. A *history* is defined as a finite series of events. A *linearizable* history is one where any operation can be considered to take effect instantly at a single point in time, between its invocation and response while satisfying the specifications of the sequential data structure [28].

Izraelevitz et al. extended linearizability to system-wide crash events [33]. A history is *durably linearizable* [33] if, after removing all crash events from the history, it remains linearizable. This means that any operation that completed before a crash must be visible after the crash along with some of the operations still in execution during the crash. In addition, if an operation survives a crash, then all operations it depends on must also survive the crash. To satisfy durable linearizability, the execution model may allow a recovery operation, which is called right after a crash. Note that a crash may also happen during recovery time (of a previous crash).

RELAX applies to *lock-free* data structures, which are a good fit for persistence. A data structure is lock-free if one of its threads must make progress, even in worst-case scheduling scenarios. In particular, threads must be able to make progress even if one or more threads stop responding. Since a thread may stop executing at any time, and execution cannot be disrupted, a lock-free data structure provides a consistent data structure state throughout its execution, which matches persistence very well. Whenever a crash occurs, a lock-free data structure must be in a consistent state, and so it can be recovered, if its state is well-represented in NVMM. Persistent lock-free data structures can be found in various papers, e.g., [15, 19–21, 33, 58].

4 Mirror Overview

Our lazy recovery algorithm extends Mirror [21]. For the sake of completeness, this chapter provides a brief overview of the Mirror construction.

The *Mirror library* is a general construction that provides durability automatically. It is easy to use, and it eliminates the need of a programmer to comprehend the complexities that arise when designing algorithms for non-volatile main memory.

The Mirror library extends the std::atomic library [5], adding support for persistence on non-volatile main memory, by overloading existing operations, e.g., compare_exchange_strong, load, store, etc. To use the library, only the following is required. Every field within a persistent object simply needs to be converted to `patomic`. The allocator wrapper needs to be used in every allocation, and a tracing operation must be provided. The purpose of the tracing mechanism is to be able to trace all the reachable data from the persistent roots after a crash and persist only reachable data. Mirror supports up to 8-byte fields, though support can be extended to larger fields using a layer of indirection, making the persistent field point to the larger field and changing fields by switching pointers.

When a crash occurs, a recovery operation is invoked, which traces all nodes in the data structure and copies each one to the DRAM, to allow further operation execution on the data structure. The recovery process is lengthy, and it is assumed that crashes occur infrequently. Still, we would like to be able to return to normal operation much earlier, even if crashes are infrequent, as recovering from a crash can take a long time. RELAX, proposed in this paper, employs lazy recovery to achieve this desired goal.

In the underlying construction, each persistent object has two replicas: volatile and persistent. Persistence is achieved by the persistent replica that resides on NVMM and is always flushed for persistence. The volatile replica resides in DRAM. By placing it in the DRAM, reading this replica becomes more efficient. Indeed, during normal execution, Mirror reads all data from the volatile replica. To ensure consistency between replicas, writes are carefully executed on both the volatile and the persistent replicas. The Mirror library maintains consistency between the two replicas while concurrent writes and reads occur, by implementing the `patomic` type with two fields: an `std::atomic` value that contains the actual value, and a corresponding `std::atomic` sequence number. Linearizability [28] is guaranteed by a careful write procedure, that makes sure the volatile replica update is delayed by at most one value behind the non-volatile update. The sequence number is used to signify how many updates occurred on the related field. Each time a value is updated, the associated sequence number is increased monotonically, and `ABA` problems are avoided.

To support both replicas, Mirror's allocator implements two operations. The first, called `init`, mmaps a persistent and a volatile memory region. The second, called `alloc`, which

is an allocator wrapper, guarantees the allocation of an object on both regions.

Next we will explain Mirror's writing operation, which delicately updates both replicas to preserve durable linearizability. The write is first done on the NVMM, the persistent replica, and then on the DRAM, which is the volatile replica. Every write is executed using the CAS operation. As *store* and *fetch_add* may never fail, they keep calling the *CAS* operation until they succeed. The pseudo-code of the CAS is taken from the original paper [21] and presented in Figure 4.1. Let v be a variable located on both replicas. First, it needs to verify that the value in the persistent replica is equal to the value in the volatile replica, both in terms of the sequence number and the value itself (lines 5-16). Since reading both the value and the sequence number is not atomic, the sequence number is read twice to guarantee that the value is indeed related to this sequence number. After verifying that both the volatile and persistent values are equal (after line 29), a new value can be written to the persistent replica. The new value will contain the new value itself and a sequence number that is increased by 1 compared to the last written sequence number. Every write is executed with the DWCAS instruction, *double-width-compare-and-swap*, which swaps two adjacent locations atomically in line 40. Upon success, the same value is written to the volatile replica in line 44, and the write is finished. Upon failure, another thread has attempted to write the value v to the volatile memory. In any case, the operation can be completed.

In case of a failure in writing the value to the persistent memory, the thread that failed in writing v helps the thread that succeeded by writing the successful value in the volatile replica in line 47 and returns false.

The last case is where the value read in the volatile replica is different than that read from the persistent replica. This case means that there is another ongoing concurrent operation, and the current thread needs to help it (line 19).

Algorithm 4.1: Patomic Compare_exchange_strong Implementation

```
template < typename T >
bool patomic<T>::compare_exchange_strong (T& expected, T newVal) {
  patomic<T>* rep_p_addr = REP_V_2_REP_P(this);
  while (true) {
    rep_p_seq = rep_p_addr->seq; // Read rep_p
    rep_p_val = rep_p_addr->val;
    rep_p_seq_again = rep_p_addr->seq;

    rep_v_seq = this->seq;       // Read rep_v
    rep_v_val = this->val;
    rep_v_seq_again = this->seq;

    // Restart if seq and val inconsistent
    if (rep_p_seq_again!=rep_p_seq ||
        rep_v_seq_again!=rep_v_seq)
      continue;

    // Help to complete another ongoing write
    if (rep_p_seq == rep_v_seq+1) {
      FLUSH(rep_p_addr);
      FENCE();
      before = {rep_v_val, rep_v_seq};
      after = {rep_p_val, rep_p_seq};
      DWCAS(this, before, after);
      continue;
    }

    // Make sure we have the same versions
    if (rep_p_seq != rep_v_seq) continue;

    // If value on rep_p is not expected, fail
    if (rep_p_val != expected) {
      expected = rep_p_val;
      return false;
    }

    // Update rep_p
    before = {rep_p_val, rep_p_seq};
    after = {newVal, rep_p_seq+1};
    bool res = DWCAS(rep_p_addr, before, after);
    FLUSH(rep_p_addr);
    FENCE();
    if (res) {
      DWCAS(this, before, after);
    } else {
      if (before.val == expected) continue;
      DWCAS(this, {rep_v_val, rep_v_seq}, before);
    }
    return res;
  }
}
```

5 Related Work

Many existing durable constructions can be roughly divided into data-structure specific implementations or general constructions. General constructions are normally less efficient but easier to integrate and can be used with a wider variety of algorithms.

The general constructions usually use a logging mechanism to be able to fully recover from a system failure. To provide crash consistency across system failures, general constructions for persistent applications are usually built upon transactional interfaces. These interfaces rely on having two versions of the data, usually by maintaining some sort of a logging mechanism, e.g., journaling, undo/redo/shadow logging, etc. [3, 7–10, 12, 14, 22–25, 31, 32, 35–37, 40–42, 44, 51, 55]. Logs are used for recovering to a consistent state, as they are usually persisted before the actual change is made to the data itself. A large effort has been invested to improve logging techniques over the years.

Haria et al. [26] provide a library of C++ data structures which relies on shadow paging. Only one fence is used per operation, but not guaranteeing durable linearizability [33], as it allows for a stale version of the data structure to reappear after a crash, unlike in RELAX. During recovery, they rely on garbage collection to clean up allocated memory from an incomplete FASE, based on reachability analysis. This recovery method, as mentioned in Section 6.5, requires a long downtime after a crash.

Pronto [42] adds durability to volatile data structures by using Asynchronous Semantic Logging (ASL) to convert each operation on a volatile data structure into a failure-atomic operation. It creates periodic, persistent snapshots of the data structure on NVMM. To recover, Pronto replays semantic log entries recorded after the latest snapshot. During evaluation Pronto did not recover immediately unlike RELAX. It displayed a recovery time of 7 seconds for 32GB of data.

Xu et al. [57] presents Clobber, which logs the minimal set of writes that guarantees a consistent state after recovery. Upon a system crash, the recovery restores all the overwritten inputs, and re-executes the transaction until it completes, for every thread. Clobber, however, provides ACID semantics, which are weaker than durable linearizability [33], and have only one copy in the non-volatile main memory, providing slower reads than RELAX.

FlatStore, designed by Chen et al. [6] proposes a persistent key-value storage engine. They decouple the role of a key-value store into a persistent log structure and a volatile index for fast indexing. After reboot, FlatStore loads the volatile index to DRAM, which might require a significant time, same as in Mirror [21]. In case of a crash, FlatStore sequentially scans its log to recover all the volatile data structures.

PMThreads [56] does not use a log, but uses two copies of the data in order to provide transparent failure-atomicity for lock-based parallel programs. The application writes to a

volatile buffer, which is eventually written to the active copy in the persistent memory. As these updates happen periodically, it only guarantees buffered durable linearizability [33], which is weaker condition than the one that RELAX provides.

Using a log, however, is very expensive due to the redundant writes one need to execute to the persistent memory. Updating a copy, on the contrary, might reduce some of these writes, but it might also degrade the performance as the updates to the active copy are eventually serialized.

Another trend of general constructions eliminates the need to execute extra writes relying on lock-freedom. Updates to a lock-free data-structure always leave it in a consistent state.

Izraelevitz et al. [33] presented a general technique that provides durable linearizability for any lock-free data structure. The construction requires inserting a flush and a fence for every shared read and write, which is not practical in terms of performance, as demonstrated in chapter 8. Recovering however, requires minimal time.

Zuriel et al. [58] eliminated the need to persist every shared read and write by not persisting any pointers, for set data structures, which improved performance tremendously. Recovering after a crash, however, requires traversing the entire heap and looking for valid nodes in the persistent memory, which might be very slow.

Friedman et al. [19] introduced NVTraverse, which is another general technique for constructing durable lock-free data structures. It provides an automatic way to insert flushes and fences for data structures that have a special traversal form. After a crash, it requires traversing the entire data structure and trimming all the virtually deleted nodes before new threads start to operate, unlike RELAX.

AutoPersist [52] uses another approach. It transparently and dynamically ensures that all reachable data from predefined durable roots will eventually reside in the NVMM. On recovery, it relies on tracing all the reachable object by the garbage collector, and cannot start working immediately, as RELAX can.

TIPS [37] uses logging to provide durable linearizability for any key-value store scheme. In addition, the DRAM contains a hashtable cache that accelerates reads. When recovering from a crash TIPS must process the entire log. During evaluation a recovery time of 9 seconds was measured, compared to RELAX's milliseconds.

Log-Free Concurrent Data Structures [15] provides hand-tuned techniques for constructing an efficient durable lock-free data structure. Durable linearizability is achieved using the link-and-persist technique. Updates are accumulated and persisted together using a link-cache, and a sync must be called in every operation in order to keep it durably linearizable. While David et al. [15] provide techniques for building durable data structures, those techniques require hand-tuning from an expert user. Complexity-wise, their recovery time is linear at the amount of active memory areas, or the size of the data structure, compared to RELAX's recovery which is $O(1)$ for all data structures.

RELAX's challenges resemble challenges of concurrent garbage collection algorithms that attempt to move an object while other threads are attempting to modify it (e.g., [17, 38, 46, 47, 53]). However, challenges for concurrent relocating garbage collectors are more complex because RELAX can copy each field before the program accesses it, and it just needs to make sure that all concurrent threads comply. In contrast, garbage collection must allow concurrent modifications of objects while the objects are being moved.

RELAX's challenge of ensuring data availability before access is reminiscent of a paging mechanism needing to page in data before it is accessed. While page-level recovery has the advantage of sequential access, it also has two main drawbacks. From a performance perspective, copying an entire page implies also copying un-allocated data unnecessarily, which may reduce performance. From a correctness perspective, RELAX coordinates concurrent recovery by multiple threads using the same version field that Mirror uses. Synchronizing data structure operations against copying is a further challenge for a page with no semantic

fields and correctness is more involved. Without full blocking, a field could be recovered and further modified by one thread, before a second delayed thread overrides the new values with older values from recovery time.

6 RELAX

6.1 Overview

In this chapter, we present the RELAX transformation. RELAX extends Mirror [21] with a responsive recovery algorithm, so that data structures created with RELAX can immediately start execution after a crash.

RELAX is composed of two parts. It inherits the patomic type from Mirror, with updates discussed in the next few sections. Following a crash, Mirror copies the entire data structure from non-volatile to volatile space. Thereafter, the data structure is in a consistent state and regular execution can proceed. In order to provide swift recovery, RELAX starts executing with most of the data structure stored only in the NVMM. While executing, RELAX tracks which data has been recovered. Recovered data is accessed in DRAM, whereas unrecovered data is first copied from NVMM to DRAM before being accessed. A tracking mechanism is used to track which fields have been recovered to the DRAM. The process can be seen in figure 6.1.

We start in section 6.2 by specifying how a programmer can take a lock-free data structure and use RELAX to make it persistent. Next, we present the basic algorithm for recovering a single `patomic` field in Section 6.3. We then discuss how to ensure each `patomic` is lazily recovered transparently and efficiently before it is first accessed after a crash in Section 6.4. We discuss the memory management system we used in Section 6.5 and describe a way to convert it into a persistent safe memory reclamation scheme in Section 6.5.1. We argue about correctness in Chapter 7.

6.2 RELAX's interface

The Mirror library requires the programmer to convert all fields they wish to persist into `patomic`. Mirror also requires the programmer to provide a traversal function that traverses all the data structure nodes that are reachable from the roots. This traversal function is used in the recovery process to copy all reachable data from NVMM to DRAM. RELAX does not need to traverse the data structure during recovery, and therefore, it eliminates the need to provide a traversal function.

Mirror provides the `patomic` type which extends the standard C++ library's `std::atomic` type. RELAX provides a similar type denoted `BitsetPatomic` which works for the general case (similarly to Mirror) with no extra assumptions. However, the most general case bears a noticeable overhead. While the recovery process allows the data structure to respond almost immediately, the normal execution (when no failures occur) becomes

Figure 6.1: RELAX System overview

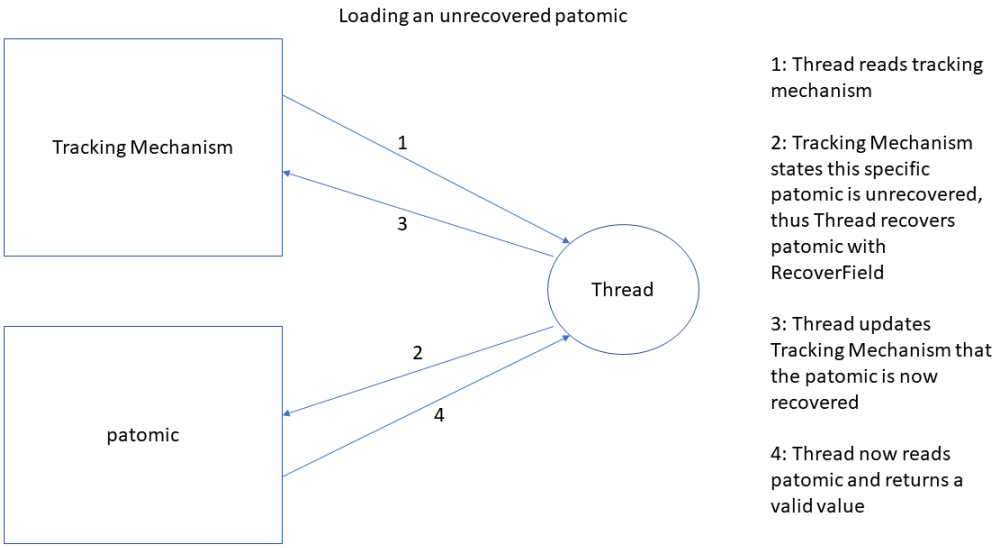


Figure 6.1 displays the process of loading an unrecovered patomic field. When a patomic is already recovered, steps 2 and 3 can be skipped and the value can be directly returned.

slower as can be seen in the evaluation in chapter 8. During this section we will start the description with the general case and no assumptions made. Later we also study two practical assumptions that enable performance optimizations. We study each of the assumptions separately, and then also the case where both assumptions hold.

The first assumption is that the data structure is recursive. Namely, it consists of nodes, connected using pointers. We also assume that the programmer is willing, when declaring persistent `patomic` fields, to distinguish between pointers and data (non-pointers) fields. This assumption on the data structure and on being able to identify persistent pointer fields allows effective optimizations that will be described below.

The second assumption that allows optimizing performance is that the operating system zeros a memory space when it is `mmap`'ed. This assumption is common in practice due to security concerns, and in particular, it holds for Linux and Windows [39, 43]. The initialization assumption implies that after a crash, the region of volatile memory to which we copy the data structure is initiated with zeros. Its content is not arbitrary. Using this assumption simplifies the algorithm and allows optimizations unavailable for the general case.

6.3 Recovering a `patomic` Field

RELAX uses the same `patomic` API as Mirror. The `patomic` type supports all C++'s atomic operations, but RELAX also extends this type with a (private) `recoverField` method. After a crash, RELAX internally makes sure that the `recoverField` method is called on each individual `patomic` before it is accessed. The goal of the `recoverField` method is to make sure that the `patomic`'s NVMM version has been copied to DRAM after the crash.

In the original Mirror setting, all fields are copied from NVMM to DRAM after a crash (before starting to execute operations) and therefore all fields are ready for access. Unlike Mirror, RELAX starts operation immediately after a crash, allowing recovery to

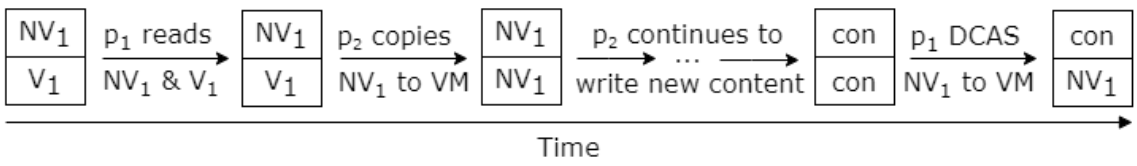
run concurrently with data structure operations. This means that data needs to be copied concurrently with the execution and there are multiple threads trying to complete operations and copy data from the NVMM to DRAM at the same time. Therefore, recovery needs a more involved concurrent algorithm, which we present below. In Section 6.3.1 we describe how to recover a field when recovery is needed, and in Section 6.4 we describe how to decide whether a field needs to be recovered (i.e., has not been previously recovered).

6.3.1 The Algorithm

We note first that RELAX cannot use a simple `memcpy` to copy data from NVMM to DRAM. As concurrent operations are executing, one thread can run `memcpy` to copy the NVMM value to DRAM, and then update the field value in both NVMM and DRAM. But a concurrent thread, running a bit slower, may overwrite the new value with the older value from NVMM due to its own slower `memcpy` execution, leading to an incorrect execution.

A seemingly simple solution to the above problem could be to use a DWCAS [30] in the copying process and relying on the sequence number (included in each `patomic` field) to prevent a delayed DWCAS attempt on a field that has already been copied and modified after the crash. But this solution is not linearizable (not correct) if the content of the DRAM after a crash is arbitrary. In particular, it is possible that after a crash, the value and sequence number of a field on the DRAM coincidentally are set to some value and sequence number that confuse a slow recovering thread into rolling-back DRAM values. An example of such a scenario is depicted in Figure 6.2. The upper square represents the content in the NVMM and the lower square represents the content in the DRAM. A process, p_1, attempts to recover a field, and in preparation to copy, it reads the content NV_1 from the NVMM and V_1 from the DRAM. the process p_1 stops right before executing a `DWCAS` to rewrite the DRAM field. At that time, another process p_2 recovers the same field and completes, copying NV_1 to DRAM. Then, p_2 continues execution, writing new content to both the NVMM and DRAM. At some point, p_2 writes the content *con* to both NVMM and DRAM and by coincidence *con* equals V_1. Finally, p_1 wakes up and wants to write the content it read, NV_1, to the DRAM. It uses DWCAS, which sees $con = V_1$ in the DRAM, allowing DWCAS success, overwriting DRAM with NV_1. As the original V_1 is arbitrary, it can exactly match both value and version number that confuse the erroneous DWCAS. Subsequently, any process that tries to read from the `patomic` field will see NV_1 instead of p_2 's *con*. This scenario shows that correctness is not obtained because either the semantics of the data structure are violated by introducing a new value with no corresponding operation, or the order of operations of the same thread are reversed, violating linearizability.

Figure 6.2: A problematic scenario



The main problem in the above example is that the DRAM content of the field following the crash matched an actual subsequent field content in the execution. Recall that the content of each `patomic` field consists of (val, seq) pairs where *val* is the value written by the application and *seq* is a separate number signifying how many times the field value was modified. To eliminate the above problem, we use *seq* to prevent such scenarios. To this

end, we make sure that the sequence numbers in the DRAM field content after a crash will not appear in a subsequent NVMM content within a very long time after the crash. This would ensure V_1, that includes the sequence number will not equal *con*. To avoid unnecessary updates, we only update the NVMM sequence ($NVMM.seq$) if $NVMM.seq$ could reach $DRAM.seq$ in at most 2^{63} updates. If this can occur, we modify $NVMM.seq \leftarrow DRAM.seq + 1$, which guarantees it would take at least 2^{63} updates to reach either the original DRAM sequence or the original NVMM sequence. This guarantees that sequence numbers do not repeat, and so contents do not repeat ensuring `recoverField` will not write old data to the DRAM.

The RELAX `recoverField` algorithm is presented in Algorithm 6.2. The algorithm starts (lines $1-2$) by reading the value and sequence number from the DRAM and then from the NVMM. If both values and sequence numbers match, the data is in a stable state and no recovery is needed. If not, in Line 4 we check whether the DRAM starts with a sequence number that might be reached from the NVMM sequence number in less than 2^{63} updates. As previously discussed, and illustrated in Figure 6.2, this could lead to overwriting valid modifications. In Line 4.1, if there is a potential sequence number collision we modify the NVMM's sequence number to be higher than the DRAM's sequence number, specifically higher by 1, which is $S_D + 1$ in our case. If $S_D = S_{NV}$ then the above problem will not occur and we do not have to take action, thus we can directly continue to Line 5. Once we ensure that the DRAM sequence value does not reach the NVMM sequence due to its arbitrary initial content, we modify the DRAM field to equal the content of the NVMM in Line 5, concluding recovery and allowing threads to rely on the DRAM's content when executing the data structure's operations.

Algorithm 6.2: RecoverField

```
RecoverField (address):
1 : read (V_D, S_D) from DRAM(address)
2 : read (V_NV, S_NV) from NVMM(address)
3 : if (V_NV, S_NV) = (V_D, S_D) return
4 : if 0 < (S_D − S_NV) mod 2^64 ≤ 2^63
  4.1 : if CAS(NVMM(address).seq , S_NV , S_D + 1) fail, goto 1
  4.2 : S_NV = S_D + 1
5 : DWCAS(DRAM(address) , (V_D, S_D) , (V_NV, S_NV))
    if DWCAS fails, goto 1
```

6.4 Tracking Recovery

In this section we describe how to tell whether a `patomic` field has already been restored since the last crash, or whether one needs to execute `recoverField` before accessing it. We could avoid tracking field recovery by calling `recoverField` before any load and store of a `patomic` field, but such a solution degrades performance significantly.

A natural solution to tracking field recovery is to use a flag for each `patomic` field signifying whether it was already recovered. We start with a simple solution that does not perform optimally, and then discuss assumptions that allow performance optimizations. To distinguish the different versions of the algorithm, we name `patomic` differently for each version.

6.4.1 Using a Markbit Separate Table.

The simplest solution for storing the recovery flags, is to use an external markbit table, with a single bit per potential field address, signifying whether a recovery was already applied to this field. In this simple case we replace the name `patomic` with `BitsetPatomic`. Given an address of a `BitsetPatomic` field, we query its associated mark bit in the markBit table, which represents the recovery flag. We can then call `recoverField` if the associated flag in the markbit table is unmarked. After calling `recoverField` we mark the markbit table to avoid redundant repeated calls to `recoverField`.

A problem with this approach is that the markbit table must be initialized before the first usage, which is costly, and defies the attempt to make a quick recovery. A straightforward initialization of an array takes $O(n)$ time[1], and must occur before execution can begin. To resolve this problem, we employ an array with $O(1)$ initialization time, for which a concurrent lock-free variant was recently proposed by Jayanti and Shun [34]. However, this method uses auxiliary arrays to initialize the array when accessing an individual cell. These auxiliary arrays contain in total two indices for each word in the main array. As a result, this method adds a space overhead, as each 64 bits in the markBit table, associated with 64 `patomic` variables, require at least 16 bytes. Which translates to 1 byte in a markbit table for every 64 bytes of data. This method offers an $O(1)$ initialization, but trades off memory overhead. Locality is also harmed, due to the need to access an external array during a load, which harms performance. In what follows we attempt to improve over the simple markBit table (that employs the $O(1)$ initialization scheme) by looking at realistic assumptions that can be made with standard data structures and systems.

6.4.2 Recursive Data Structures.

Our second method works with recursive data structures, which are widely used in practice. Recursive data structures consist of nodes connected by pointers (e.g, linked lists, trees, skip lists, but not arrays). A recursive data structure has (a small number of) root pointers from which all other nodes can be reached. Given that the data structure is recursive, we can improve the maintenance of the recovery flag (the mark bit). We place the bit that signifies whether a node has been recovered (copied from NVMM to DRAM) in the pointer to the node. Assume first that there is a single pointer pointing to each node (such as in a linked-list or a tree, but not a skip list). In this subsection we present an algorithm that works for this case, and we deal with multiple incoming pointers later in Subsection 6.4.3 below. The idea is that a pointer to an object will tell whether the referent is already recovered or not. A traversal of the data structure can make sure that all traversed nodes are recovered by examining the pointers on the way and recovering any node that is not yet recovered, before accessing it. For this method we denote persistent pointers with the type *MarkablePatomicPointer* and other non-pointer atomic fields use the type *patomic*.

This algorithm requires that the programmer provides a simple `recoverNode` method. This method receives a pointer to a node, and it calls `recoverField` for each field in the node. This method can be automatically generated when the programming language provides a reflection operation that can generate a list of fields for a given object, and it is trivial for the programmer to write such a method.

To place a flag on a pointer, we note that objects are often word-aligned (8-bytes-aligned), which leaves the three least-significant bits (LSBs) of each pointer permanently zeroed. We use the third LSB to mark whether the referenced object has been recovered. After a crash we recover the roots of the data structure by copying them to DRAM and setting their mark bit to signify that the objects they reference have not been recovered.

[1]Initialization time is O(n) when n is the size of the NVMM memory space, which is higher – and potentially much higher – than the data structure size.

From then on, the program can only reach an object through recovered pointers, which are marked if pointing to unrecovered objects. Thus when dereferencing a marked reference (via the `load` method of the MarkablePatomicPointer field) the `load` method will recover the referenced node (via the `recoverNode` method) and reset the mark bit of the reference itself. Thus the `recoverNode` will be called before accessing any unrecovered nodes.

Lock-free data structures (which we make persistent by the RELAX transformation) often make use of LSBs of a pointer to mark objects for deletion (e.g., [18,27,45]). But the algorithms we are aware of only use the first two LSBs for this purpose, and this is why we propose to use the third LSB for marking the recovery of a referent. The advantage of keeping the recovery bit in the pointer to an object is that we often have to read this pointer to access the object, and so the pointer is still in the cache when we check whether the referenced object has been recovered. This reduces the excessive number of cache misses and the space overhead that the general bitmap solution imposes.

To implement this approach, we extend the patomic<T*> class of the Mirror construction to MarkablePatomicPointer<T> which inherits from patomic<T*>. Pointer fields in the data structure should be declared as MarkablePatomicPointer, whereas non-pointer fields should be declared as patomic. MarkablePatomicPointer<T> is a `patomic` of a pointer and has all of its methods, with two methods modified: `recoverField` and `load`. The `load` method is changed to check for the mark bit of the pointer, and if marked, the `load` uses `recoverNode` to recover the referenced node and unmark the MarkablePatomicPointer pointer before continuing to execute with it. The code for `recoverField` is presented in Algorithm 6.3. This is similar to the `recoverField` code for `patomic`, but with two important changes. First, when copying the address from the NVMM replica to its corresponding DRAM replica as is, the DRAM replica of the pointer is marked as pointing to unrecovered data by setting the third LSB. Null pointers are not marked, since their referent is vacuously recovered. This can be seen in Line 4, where we write $(mark(V_{NV}), S_{NV})$ to the DRAM. Second, Line 3 from Algorithm 6.2 is removed, since now we expect V_D and V_{NV} to be different.

The code for `load` appears in Algorithm 6.4. When loading a pointer from DRAM, we check if it is marked. If the pointer is marked, we recover the referenced object using the object's `recoverNode` method (supplied by the programmer), which calls `recoverField` for every field within the object. We then unmark the pointer in DRAM and return it. Since unmarking is local to the DRAM, we do not change the sequence number.

Algorithm 6.3: RecoverField For MarkablePatomicPointer

RecoverField (address):
1 : read (V_D, S_D) from DRAM
2 : read (V_{NV}, S_{NV}) from NVMM
3 : if $0 < (S_D - S_{NV}) \mod 2^{64} \leq 2^{63}$
  3.1 : if CAS($NVMM(address).seq$, S_{NV}, $S_D + 1$) fail, goto 1
  3.2 : $S_{NV} = S_D + 1$
4 : DWCAS($DRAM(address)$, (V_D, S_D) , $(mark(V_{NV}), S_{NV})$)
  if DWCAS fails, goto 1

6.4.3 Recursive Data Structures with Multiple Pointers Pointing to a Node

The algorithm described in Subsection 6.4.2 above, that recovers nodes along a traversal, does not work smoothly with data structures in which a node has more than one reference (e.g, a skip list). An already recovered node may have one pointer that signifies the recovery, i.e., the pointer that was traversed and that caused the recovery, yet this node may have an

Algorithm 6.4: MarkablePatomicPointer Load Algorithm

Load (address):
1 : read (V) from DRAM
2 : if marked(V)
 2.1 : read (S) from DRAM
 2.2 : unmark(V)->recoverNode()
 2.3 : DWCAS($DRAM(address)$, (V, S) , (unmark(V), S))
 if DWCAS fails, goto 1
 2.4 : V = unmark(V)
3 : return V

additional pointer that is still marked, erroneously signifying that the referent has not been recovered. Such nodes with multiple parents can cause a repeated cascading recovery, with a destructive effect on performance. To understand how this can happen, consider a node n which is referenced by two different pointers, $p1$ and $p2$. Initially, both $p1$ and $p2$ are marked, and n is not recovered. When n is first accessed, say through $p1$, n is recovered and $p1$ is unmarked. Subsequently, many descendants of n may be recovered. However, when a thread accesses n through $p2$, it observes a marked pointer, so n has to be recovered, via the `recoverNode` method, which marks all the pointers in the object to erroneously signifies that also the descendants are unrecovered. Thus any node that was previously reached from n will be recovered again proceeding in a cascading manner, even if recovery was already executed. For this reason, using pointers to keep the recovered status of the referenced objects is not applicable for data structures with more than a single reference per node.

We overcome the repeated recovery problem by employing both the marking of pointers (i.e., using MarkablePatomicPointer for persistent pointers) as well as an external markbit table as in Section 6.4.1 that signifies for each node (rather than field in the previous section) whether it has already been recovered. When loading, if the pointer is unmarked (which it usually is, except for the first access), then the load can proceed with no overhead. If the pointer is marked, then we read the markbit table to verify that the object has not been recovered from a different pointer, and if it is indeed not recovered, then we invoke `recoverNode` before accessing it and clear its mark bit in the markBit table. Whether the node needed recovery or was already recovered, we remove the mark from the pointer. This use of both pointer marking and external marking ensures that a node will not be recovered for each pointer, and it eliminates the cascading issue. The performance issue due to the external markBit table becomes minimal since it is only examined when reading a marked pointer, and a pointer is marked only on the first time it is read after a crash. The other downside of using an external markbit table still remains, as the memory cost of storing the bits is significant. Nevertheless, note that the space overhead is slightly reduced because we keep a bit per object and not a bit per field (depending on objects alignment, typically by a factor of 4).

6.4.4 Adding a Zeroing Property of Standard Operating Systems.

We now consider a second assumption. Suppose that we have a recursive data structure with potentially multiple pointers referencing it. But suppose that we can further assume (as mentioned in the end of Section 6.2) that the `mmap` system call resets the allocated memory to 0. This is usually done due to security concerns, e.g., in Linux and Windows [39, 43]. Since the DRAM space is allocated by `mmap` following a crash, this volatile space is known to be zeroed before we recover objects. Thus, objects that are not zeroed must have been recovered. This can be used instead of an auxiliary markBit table to save space overhead

and improve performance. Specifically, pointers do not have to be re-marked, avoiding the cascade effect. Note that each persistent field in Mirror has a sequence, which can be kept non-zero to easily tell between fields that have been recovered and fields that have not.

6.4.5 Using the Zeroing Property in the General Case

The zeroing assumption for the DRAM space can be more generally be used to improve the general solution of Section 6.4.1. To this end, we can use the most significant bit (MSB) of the sequence number to signify whether the value of this field has been recovered. Since the DRAM is initialized to 0, the sequence number read from an unrecovered addresses is also 0. Thus the MSB will be 0. If we maintain the MSB of the sequence number to be 1 (by using only the 63 least significant bits of the sequence) we can always know whether an address has been recovered by checking only the DRAM copy of that address.

6.4.6 Summary

In summary, an external markbit table can allow us to track the status of each `patomic` but comes with a memory overhead and a high performance cost due to the loss of locality. This is how the `BitsetPatomic` tracks recovery. If the data structure is recursive with a single incoming pointer per persistent node, then we can use a mark bit in the pointer to the node. This employs the `MarkablePatomicPointer` type. If a node can have multiple incoming pointers, then we can use `MarkablePatomicPointer`, but must add an external bitset table to signify whether a node has been recovered. This approach mixes the markable pointers and bitset approaches, and thus we named it the `HybridPatomicPointer`. If we can further assume that the DRAM is zeroed then the bitset becomes unnecessary, which translates to just mark the pointers. That is the `ZMarkablePatomicPointer` approach. Finally, If we can assume the DRAM is initialized to zero but the data structure is not recursive, then we propose to put the mark bit in the sequence number, which we name `PseudoBitsetPatomic`.

6.5 Memory Allocation And Management

The Mirror transformation, which we extend, uses a memory management solution that foils a quick recovery. Mirror `mmap`s memory in the NVMM, and uses a safe memory reclamation (SMR) scheme called `ssmem` [16] as a memory manager to allocate memory in the DRAM. As ssmem operates in DRAM, all the memory manager's metadata – used to track allocation – are erased during a crash and must be rebuilt during recovery.

When a crash occurs, Mirror traces all live data in the non-volatile memory and re-allocates all nodes in the DRAM using `ssmem`. This already takes time linear in the data structure size. Moreover, Mirror needs all objects in DRAM to be mirrored in NVMM. Namely, to be copied in the same order, so that address translation between the two mirrored spaces can be executed with a single addition. Therefore, after traversing the data structure objects in the NVMM and re-allocating each one in DRAM, Mirror copies the data structure space of the DRAM to a newly mapped area in NVMM, which is again, linear in the data structure size. That completes the recovery process and execution can resume. This solution is valid, but costly. Also, it incurs a large initial cost and therefore cannot work with lazy recovery.

To allow a quick recovery, we let the `ssmem` allocator allocate memory (and its metadata) in the NVMM. To ensure that the modified memory manager does not provide unfair advantage to RELAX, we also integrated this memory management scheme into the Mirror library, ensuring that the comparison is not biased by the memory manager. This change improves the performance of Mirror for most data structures (specifically, the skip list, the

binary search tree, and the linked-list) and degrades performance some some (specifically, the hash and the array) as shown in the evaluation.

We remark that the solution we adopt does not fully solve the persistent safe memory reclamation problem. What we use is a patch for measuring RELAX. A full solution is an open problem, and a major project that is orthogonal to the efforts in this paper. Lock-free algorithms require that the underlying memory manager is a *safe memory reclamation scheme* (SMR), i.e, a memory manager that can handle concurrency, which requires higher complication to make concurrent reclamation safe. An efficient durable safe memory reclamation remains a challenge for future work. To make the above solution complete, one needs to go through the (substantial sized) code of `ssmem`, and determined where to install flushes and fences. This requires a deep understanding of the `ssmem` code and a deep understanding of durable linearizability. We believe the current paper (among others) can motivate SMR researchers to work on adequate persistent SMRs. We also remark that there exist some initial solutions for persistent SMRs that build on a substantial garbage collection during recovery [4]. These solutions, like Mirror, require a long downtime after a crash (to execute the garbage collection) and therefore do not apply to this work. In the following subsection we propose a design that could be used for a project on persistent safe memory reclamation.

6.5.1 A Persistent SMR

Let us shortly present a design idea for a possible persistent SMR, based on the `ssmem` volatile SMR. The `ssmem` library employs arrays of retired objects called free sets. Each thread retires objects and records their address in the free set array. Once the free-set array is completed, it is added to a local list of free sets, and a vector clock is added, describing the time the free set was completed. When the list size increases, free sets can be moved to a global free set list accessed by all threads.

When allocating, if the current vector clock is strictly larger than the vector clock of a free set then all threads have advanced since the objects in the free set were retired and it is safe to reallocate them. We propose to flush each free set array when it is completed. This can be done by flushing each cache line in the array, and then blocking on a fence instruction to wait for the flushing to complete. In addition, we propose to use a durable list (or queue) [20,58] to hold pointers to the competed free sets. This persists all completed free sets and allows using them after a crash. The allocator uses a free set to allocate. We do not propose to add persisting instructions for that and we assume that the allocator free set must be discarded as it is not clear which objects exactly were allocated before the crash. This free set can be considered as lost memory. In addition, it is possible that the free set currently being filled up by each thread is lost after a crash, because it is not persisted prior to being completed. All other objects handled by the memory reclamation scheme were persisted before the crash, since free sets are persisted when completed and when added or removed from the free set list.

The above design lets each thread lose at most two free sets during each crash. This could be harmless if crashes are seldom. But to solve this continuous memory leak, a concurrent garbage collection can be executed after a certain number of crashes, which could depend on the size of the free set, the number of threads, and the amount of memory available. The main additional cost of this scheme is small. It requires persisting the free set and persisting additions and removals from the free set list. When we measured this cost in regular execution of our skiplist workload we got a statistically insignificant difference.

6.6 Recovering the Persistent Roots

The roots to the persistent RELAX data structures are stored in the beginning of the NVMM space. We assume that once stored, the roots are immutable. Since a root can point to a sentinel node, which can be modified, this assumption does not lose generality. After a crash, the only blocking step that needs to be executed before the program resumes, is the recovery of the data structures roots, which happens almost instantaneously. Thereafter, application threads can start executing.

7 Correctness Argument

Theorem 7.1. *Given a lock-free linearizable data structure, RELAX generates a durably linearizable data structure with operations whose linearizability guarantees the same semantics as the original data structure.*

The proof of the main theorem follows from a technical claim and a correctness argument that follows. The technical claim in Claim 1 below asserts that the tracking mechanism works well.

Claim 1. Whenever a data structure generated by RELAX invokes the tracking mechanism to check whether a field (or a node) has been recovered since the most recent crash, the tracking mechanism returns a positive result only if some thread returned from the method `recoverField`) or `recoverNode` (on this field (node) since the most recent crash.

When a data structure generated by RELAX checks whether a field (or a node) has already been recovered, it will not receive a false positive. The core of the proof is the fact that in all tracking methods a field is marked as recovered only after `recoverField` has returned. The rest of the proof for this claim needs to go on a case by case analysis over the different versions and to verify that the tracking method does not initially mark fields. This is a tedious proof that follows in a straightforwards manner from the various tracking methods. Since this paper is not focused around formal claims, we do not provide the details of this proof. We focus on the more challenging correctness arguments below.

It remains to show the correctness of Theorem 7.1 given Claim 1. We first note that any access to any field by a data structure generated by RELAX happens only after `recoverField` has completed recovering the field. This happens because loading (or modifying) a field first checks whether the field has been recovered, and if not, it invokes `recoverField` on this field prior to accessing it. Therefore, in what follows we assume that a field is accessed only after its recovery is complete. The main thing to verify is that the execution of `recoverField`, potentially concurrently (or sequentially) by several threads on the same field, while this field is also potentially accessed by additional concurrent threads satisfies correctness.

Since RELAX extends Mirror, we proceed by reducing the correctness of RELAX to the correctness of Mirror concentrating on RELAX extensions. Mirror's correctness relies on the following two invariants. First, the NVMM sequence number of a specific `patomic` variable is always equal or greater by one than its DRAM replica's sequence number. In the latter case, there is at least one ongoing operation that modifies the field, and the NVMM's replica is only one step ahead of the DRAM for this field. Second, the NVMM's replica can

only be updated if the sequence numbers of a given `patomic` variable for both replicas are equal, meaning that the DRAM is up-to-date.

To argue the correctness of data structures generated by RELAX, we start by showing that the concurrent execution of `recoverField` does not violate Mirror's invariants. For the general system (that does not assume zeroing of the DRAM following a crash, as in modern operating systems) we need to assume that one thread is not "stuck" for an extremely long time. By extremely long time, we mean that it is not stuck for a time that allows other threads to execute 2^{63} modifications on the same field. (Something equivalent to thousands of years on current architectures.) Namely, we show that the following claim holds.

Claim 2. Consider a `patomic` variable P on which at least one thread finished executing `recoverField`, returning from Line 3 or successfully completing Line 5 of Algorithm 6.2. Assume that no thread is halted during its recovery execution while other threads execute 2^{63} modifications of this field. Then, Mirror's first and second invariants hold.

Mirror's first invariant :
either $NVMM(P).seq = DRAM(P).seq$ or $NVMM(P).seq = DRAM(P).seq + 1$.

Mirror's second invariant :
$NVMM(P)$ is modified only if $NVMM(P).seq = DRAM(P).seq$

Let us explain how `recoverField` of Algorithm 6.2 fulfills these claims. We first argue that when `recoverField` finishes for the first time, it must hold that at some point previously after the crash $NVMM(P) = DRAM(P)$. From claim 1 and the fact any access to P must first go through the tracking mechanism or execute `recoverField` we can conclude that for the data structure to access P there must be some call to `recoverField` that returned. Before this call to `recoverField` (which finished for the first time) finishes, no other call to `recoverField` can return. Thus only other calls to `recoverField` may modify P.

Assume that this first completion of `recoverField` returned from Line 3. If during the execution of this call to `recoverField` there were no modifications to P then $DRAM(P) = (V_D, S_D) = (V_{NV}, S_{NV}) = NVMM(P)$. Otherwise, since as mentioned only calls to `recoverField` can change P and no other call to `recoverField` completed Line 5, then the only possible modification is Line 4, which changes $NVMM(P)$. Since `recoverField` first reads $DRAM(P)$ and it cannot change, and because $(V_{NV}, S_{NV}) = (V_D, S_D)$, when $NVMM(P)$ was read it holds that $NVMM(P) = DRAM(P)$. The second possibility is that the first completion of `recoverField` returned after completing Line 5. If no other thread modified $NVMM(P)$ while the execution was running lines 2-5 then it holds from the $DWCAS$ that $NVMM(P) = DRAM(P)$. Otherwise, Like in the previous case, the only possible modification during the execution of this call of `recoverField` is from Line 4.1. In particular this means that $DRAM(P)$ is unmodified and therefore for this execution it holds that $DRAM(P) = (V_D, S_D)$. Since a modification occurred then some other execution of `recoverField` must have executed line 4.1. Let us look at the execution of `recoverField` that first completed Line 4.1. When it completed Line 4.1 that was the first modification to P since the crash. After this modification any call to `recoverField` will either read the old value of $NVMM(P)$, try to execute Line 4.1 and fail the CAS because of stale data, or it will read the new value which would cause the call to skip lines 4.1-4.2 since now $NVMM(P).seq = S_D + 1$. This means that until $DRAM(P)$ is changed there cannot be other successful executions of Line 4.1. We will denote the execution that first succeeded in Line 4.1 as β and the execution that first succeeded in Line 5 as α. Since β executed its modification between lines 2-5 of α, α must have read the old value of $NVMM(P)$, thus it has the same values of V_D, S_D, V_{NV}, S_{NV} as β. Thus it must have also tried executing Line 4.1. It did not fail, otherwise it would have returned to Line 1. Only one execution of `recoverField` completed Line 4.1 successfully so we can conclude that α and β are the same execution. This means that no other execution of `recoverField` could modify P.

This means that $(V_{NV}, S_{NV}) = NVMM(P)$ and so after completing Line 5 it holds that $NVMM(P) = DRAM(P)$.

Any execution of Mirror's CAS implementation (which RELAX does not change) by a generated data structure increments $NVMM(P).seq$ only if it equals $DRAM(P).seq$ and it increments $DRAM(P).seq$ only if $NVMM(P).seq = DRAM(P).seq + 1$. Thus, modifications of P by the data structure maintain Mirror's invariants. It remains to show that modifications of P by other concurrent `recoverField` operations do not violate Mirror's invariants.

Consider the first field modification that violates Mirror's first invariant, mentioned in claim 2. We will later discuss the second invariant. A thread executes a field modification only after a field recovery has completed. Thus, consider the case where some thread completed the recovery of the field P and other threads are still executing `recoverField`. Our goal is to show that all modifications of any executing `recoverField` will not violate the invariant. Let us first consider the modification of Line 4.1. Since at least one thread finished executing one `recoverField`, then at the point where that thread completed `recoverField`, the sequence was equal in NVMM and DRAM. In this case, for the modification by Line 4.1 to succeed thereafter, it is required that $0 < (S_D - S_{NV}) \mod (2^{64}) < 2^{63}$. Since until this execution the invariant holds it must be that $NVMM(P).seq \in DRAM(P).seq, DRAM(P).seq + 1$. Since S_{NV} is read after S_D, it would take 2^{63} increments of $NVMM(P)$ between Line 1 and 2 for this condition to be fulfilled. This contradicts the assumption of claim 2. Alternatively, it is possible the thread read the original post-crash values of P, and then halted for some time before trying to perform Line 4.1. In this case it would take at least 2^{63} increments of $NVMM(P).seq$ for it to reach its original value, which again contradicts the assumption of claim 2. Therefore, even after modifications of P by the data structure are allowed, a successful modification in Line 4.1 of Algorithm 6.2 can happen only by the thread that manages to execute this line first after the crash. All other threads will either fail executing the CAS in Line 4.1 and return to Line 1 or continue directly to Line 5.

We now consider whether a `recoverField` execution can succeed in its modification in Line 5 (after another thread has completed execution of `recoverField`). In Line 5, the DRAM values are modified to reflect the thread's view of the NVMM values of this field. One benign case is that both values read by this thread are the result of modifications the data structure operations executed after recovery of this field was complete. In this case, like in Mirror, there is no harm in updating the DRAM to match the NVMM. Since the invariant is first broken by this call to `recoverField`, the invariant holds when the DRAM and NVMM are read in Lines 1-2. Because a DWCAS is used, Line 5 can succeed only if $DRAM(P) = (V_D, S_D)$. In addition, (V_{NV}, S_{NV}) is read after (V_D, S_D) and $NVMM(P)$ can only be newer than (V_{NV}, S_{NV}). The above gives us the following:

1. $S_{NV} \in \{S_D, (S_D + 1)\}$
2. $NVMM(P).seq \in \{S_D, (S_D + 1)\}$
3. $NVMM(P).seq \geq S_{NV}$

(1) is true because of the invariant during the execution of Line 2. (2) is true because of the invariant during the execution of Line 5 and the fact that $DRAM(P) = (V_D, S_D)$. (3) is true because $NVMM(P)$ must contain a newer value than S_{NV} Combined we get that $NVMM(P).seq \in \{S_{NV}, (S_{NV} + 1)\}$. Thus invariant 1 holds after Line 5 where $DRAM(P) = (V_{NV}, S_{NV})$

The only time Mirror's first invariant may be invalidated is if the process successfully updated the DRAM after reading the original post-crash value of $DRAM(P)$. Since recovery has already succeeded once, and initialized $DRAM(P) = NVMM(P)$, the fact this value

repeated means it again became (V_D, S_D) through the execution of the data structure. But since Mirror only increments sequence numbers by 1 at each operation, that would mean at least 2^{63} operations on P were executed since the success of recovery. This is because either $NVMM(P)$ was updated during Line 4.1 and the sequence after recovery was $S_D + 1$ or it did not fulfill the condition of Line 4 and then for the original $NVMM(P).seq$ to reach S_D would require at least 2^{63} steps. Since our execution of `recoverField` which first violates the invariant must have started execution before recovery, then this violates our assumption that no process halts for longer than 2^{63} operations on P.

As for second invariant from claim 2, we proved above that as long as the two invariants hold, Line 4.1 of `recoverField` cannot be executed after recovery. Since `recoverField` cannot modify $NVMM(P)$ after recovery, and the data structure maintains Mirror's second invariant, as long as Mirror's first invariant is maintained it is impossible that Mirror's second invariant is invalidated.

Therefore, the first call to `recoverField` will synchronize the NVMM and DRAM. Any subsequent calls will be harmless, and at most can assist the normal execution of the data structure in updating the DRAM to replicate the NVMM.

8 Evaluation

8.1 Experimental Setup

We evaluated our code on an Intel machine possessing two Xeon Gold 6234 processors, each with 8 cores, running at 3.3GHz. The machine has 366 GB of DRAM and 1.5 TB of NVMM (Intel OptaneTM DC memory), organized as 12 × 128 GB DIMMS. There is an L1 cache for each core with 32 KB, an L2 cache with 1 MB, and an L3 cache that is shared between all cores in the processor and can contain 25 MB. The operating system is Ubuntu 18.04.6, and code was written in C++ and compiled using g++ (GCC) version 9.4.0. We used the App-Direct Mode configuration. For persisting objects, we used the `clwb` and `sfence` instructions for flush and fence, respectively. The `clwb(address)` instruction was used to allow different write-backs to occur in parallel.

We measured performance results using the YCSB workload [11], where key-value pairs are 4 B each. Nodes are cache-aligned to 128 B. Each reported result is the average of 5 repetitions, lasting 10 seconds. Unless stated otherwise, we used a uniform random key distribution from the range of $[0, r-1]$ for varying range size. Every data structure was initialized with $r/2$ keys and then measured with a varying ratio of reads vs. writes. Inserts and removals were drawn with the same probability.

We evaluated in three different ways: varying number of threads, varying ratio of writes vs. reads and varying keys range. When not varying, the default setting was 8 threads, a ratio of 20% writes, and $r = 16M$ for all structures but the linked list, which had $r = 256$. In a different experiment, meant to show the process of recovery, we estimated the throughput of the program with respect to time, by logging a timestamp every $50,000$ operations and counting the number of timestamps every 2 seconds. Our evaluation, similarly to Mirror included a sorted linked-list [27], A hash-table with a Harris et al.'s [27] sorted linked-list in every bucket, a lock-free BST by Aravind et al. [45], a lock-free Skip-List [18] and a lock-free array used as a constrained set where the range of keys is known in advance. Large linked-lists are inefficient and seldom seen in real applications, so we test the throughput of a small list, compared to the rest of the data structures.

We tested four versions of RELAX. First is where recovery is tracked using an external markbit table (§ 6.4.1), denoted `Bit Table`. Second is where the recovery status is stored in the sequence number (§ 6.4.5), denoted `Bit In Sequence`. Third uses mark pointers to track recovery (§ 6.4.4), denoted `Bit In Pointer`. Finally, the combination of marked pointers with a markbit table (§ 6.4.3) is denoted `Combination`.

Since we wanted to test all of the data structures with all of the versions, in parts of data structures that do not satisfy our recursive assumption (array and the hash table array)

we used the appropriate tracking system, using `Bit Table` when the initialization of the memory space is unknown and `Bit In Sequence` when the memory space is initialized to be 0

8.2 Results

8.2.1 Scalability

In Figure 8.1 we report the average throughput of the program, running with a varying number of threads. To better demonstrate the differences, Figure 8.2 represents the normalized results relative to Mirror. When running more than 8 threads the program executes on two separate processors, crossing the NUMA boundary and increasing memory latency, resulting in a loss of performance.

We can see in the graphs that both Bit In Pointer and `Combination` have very low overhead. For the List data structure, the overhead is less that 2.65%. For Skiplist, the overhead is up to 2.83% and for the Bst, it is up to 3.6%. For Hash and Array, the measurements are quite noisy, as apparent in the large confidence intervals on the graph. Still, we can see that the confidence intervals of Bit In Pointer, Combination, and Mirror overlap. The Bit In Sequence approach has higher overhead, but still below 12.72% for List, below 3.89% for Skiplist, and below 6.03% for Bst. For Hash and Array, again the confidence intervals overlap, except for 14 and 16 threads, where the maximum overhead is 4.46%.

Unsurprisingly, `Bit Table` has the highest overhead, reaching 60.88% for a list with a single thread. Still, for Skiplist the overhead is always less than 13.89%, and for Bst, Hash, and Array, always less than 30.81%, even in this setting where we make no assumption about the data structure.

While the overhead of `Bit Table` seems high, it is fair to compare it to existing general constructions offering near-instant recovery in the literature. As discussed in the introduction (§ 2, the only such general construction is Izraelevitz et al. [33], which is also depicted in the graph. As can be seen, `Bit Table` is 2x - 25x faster, and with better scalability, compared to Izraelevitz. Due to this huge difference, we do not discuss this further in the rest of the section.

8.2.2 Varying Size

Next, Figure 8.3 presents measurements of average throughput with varying average sizes of the data structure. We see that the overhead tends to reduce as the size of the data structure increases. This is most visible in the Skiplist, where at 8 K nodes the Bit In Pointer Skiplist operates 9.86% worse than Mirror, and at 8 M nodes the difference diminishes to 2.83%. Large data structures cause more cache misses, making the cost of recovery checks smaller in proportion. We note that lazy recovery is most crucial for large data structures, where the cost of recovery can introduce a long time during which the system is unresponsive; for such use cases, the overhead of RELAX is quite small.

8.2.3 Varying Updates Ratio

In Figure 8.4 we measured the average throughput with different ratios between lookups and updates to the data structure. We see that the overhead decreases when the update ratio increases. Updates are generally more costly, since they need to modify both the NVMM and the DRAM, while reads access the DRAM only. Therefore, the relative overhead of RELAX tend to reduce when the update percentage is higher. For example, in the Bst measurement, the overhead of Bit In Pointer reduces from 5.92% for 100% reads to 0.47% for 100% writes.

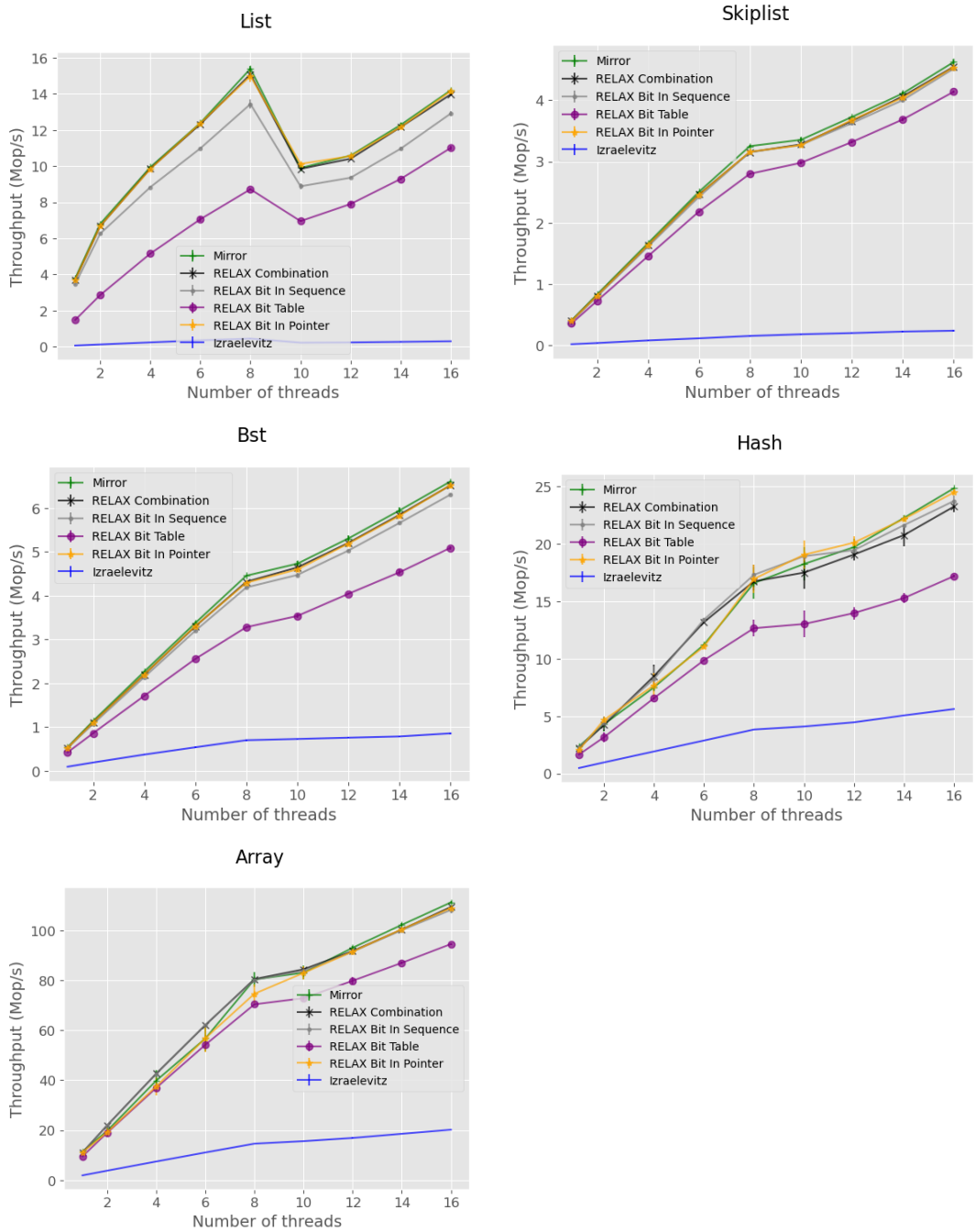


Figure 8.1: Varying number of threads - Average throughput at 80% reads. Average size for list is 128 keys, for all other data structures it is 8 M keys. Error bars represent 95% confidence intervals.

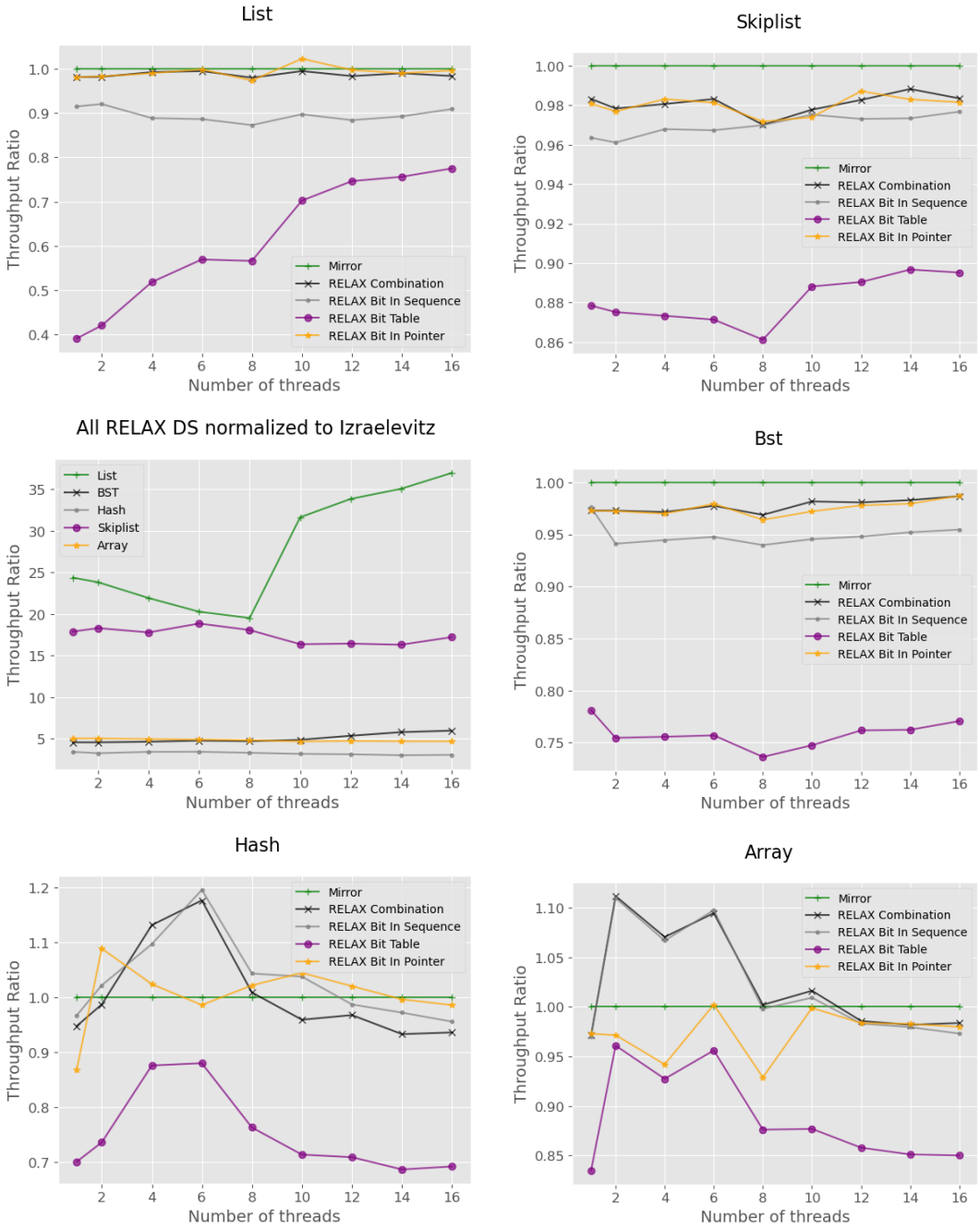


Figure 8.2: Varying number of threads normalized - Average throughput normalized according to Mirror. 80% reads. Average size for list is 128 keys, for all other data structures it is 8 M keys. Higher normalized throughput means RELAX performs better relative to Mirror. In addition, performance of RELAX compared to Izraelevitz. Displays RELAX's average throughput for all data structures normalized to Izraelevitz.

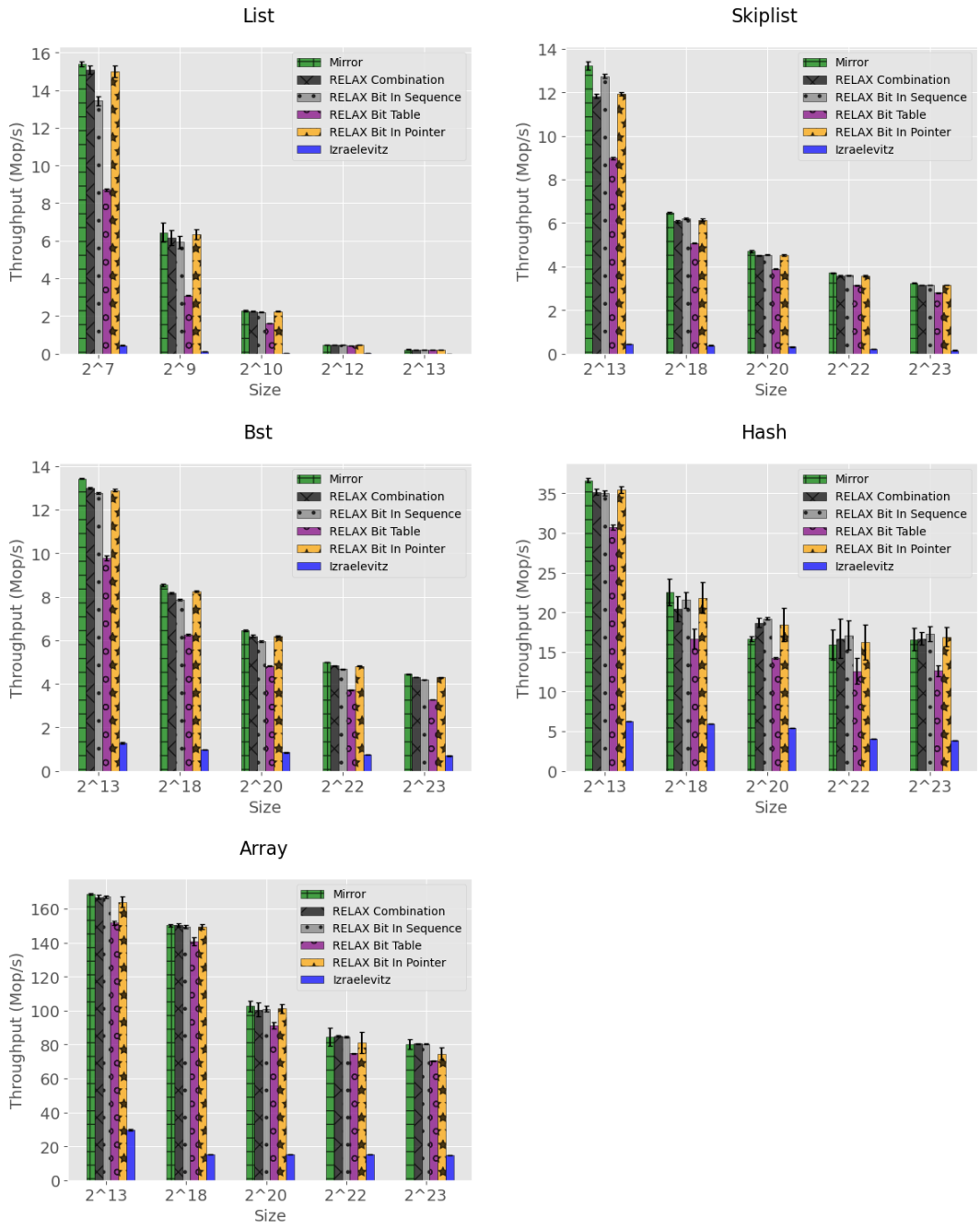


Figure 8.3: Varying size - Average throughput, at 80% reads and 8 threads. Error bars represent 95% confidence intervals.

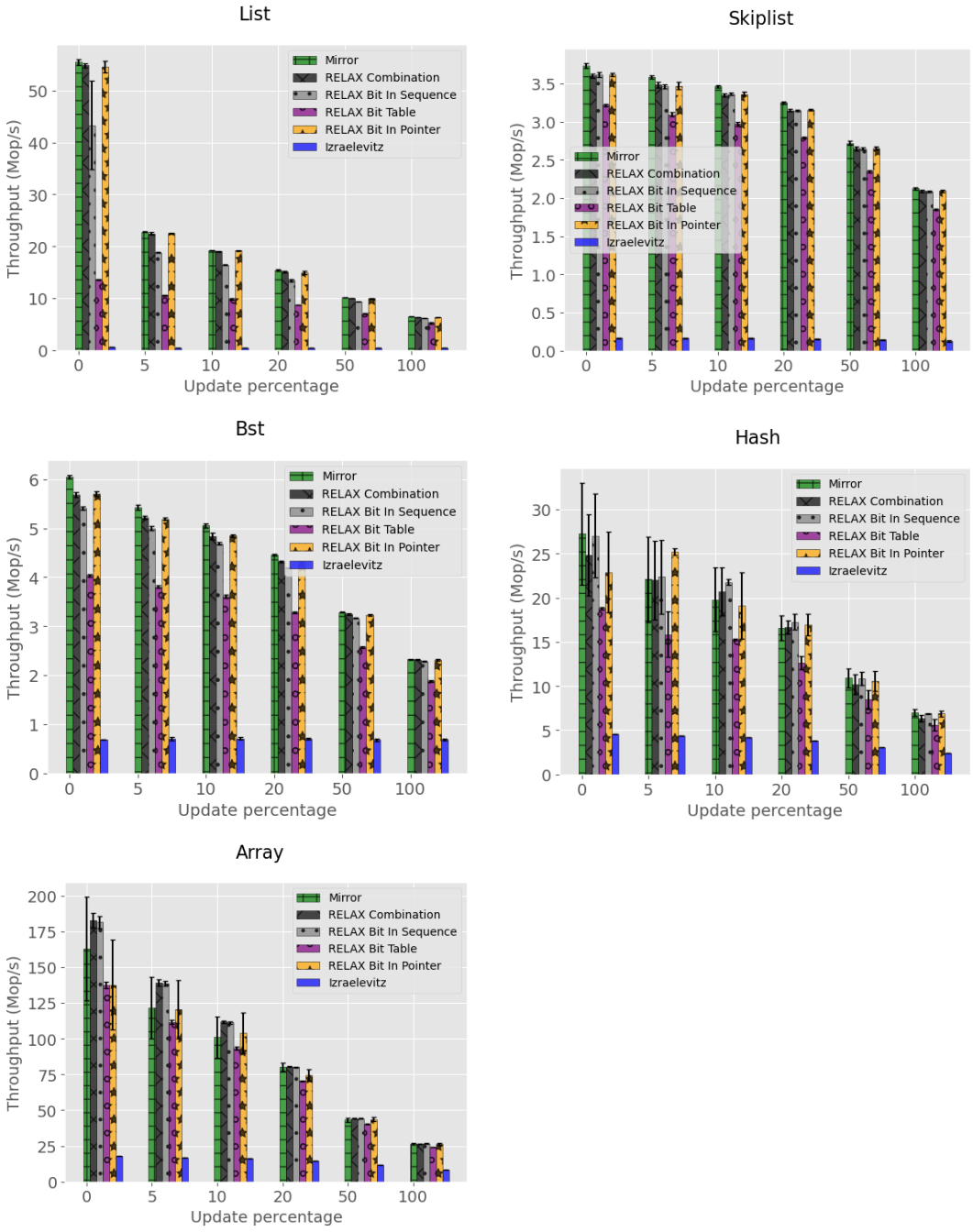


Figure 8.4: Varying updates ratio - Average throughput with average size of 128 keys for list and 8 M keys for all other data structures. all tests run using 8 threads. Error bars represent 95% confidence intervals.

8.2.4 Recovery

Figure 8.5 shows the effect of recovery on the throughput of RELAX compared to Mirror. To fully demonstrate the benefits of fast recovery we use large data structures, with 128 M nodes. RELAX allows all data structure operations to execute (with non-zero throughput) after 0.1ms. In the Mirror implementation, recovery time of Skiplist, Bst, and Hash data structure takes 27s, 20.23s and 14s, respectively, which can severely impact the users of the system.

Both the Skiplist and Bst show better than linear recovery, which can be attributed to the highly skewed probability of encountering each node. Given a random query, the probability of passing through a node in the top layers is much higher than the probability of reaching a specific node at a lower level. This means that Mirror spends a lot of time at the beginning, recovering nodes that are seldom accessed. By the time that Mirror has started operating, RELAX is already running at more than 65% throughput for Skiplist and more than 89% throughput for the Bst.

Unlike the Bst and Skiplist, the probability of visiting a specific node in the Hash data structure is the same for all nodes, making lazy recovery less effective. In particular, RELAX's Hash shows a linear recovery until recovery is mostly complete due to the uniform distribution of the keys. Moreover, the array of the Hash offers high locality for copying, which allows Mirror to recover quickly, resulting in the shortest Mirror's recovery time among the three data structures.

In many real-world applications, the distribution of keys is not uniform, but rather Zipfian. We show recovery results for this case in Figure 8.6. It demonstrates that when the keys are biased, even the Hash shows a better than linear recovery, suggesting better efficiency compared to Mirror for the first 15.03s, when it reaches more than 40% of the throughput.

We also note that table-based implementations, `C`ombination and `B`it Table, show slower recovery time. This is due to the need to access the mark table during recovery.

We excluded linked-list and arrays from this measurements. Linked-list are infeasible for 128 M nodes. For arrays, Mirror is able to recover 1 billion keys in 2.4 seconds, making the benefits of RELAX less apparent.

8.2.5 Memory Manager

We compared RELAX and its various versions with the original Mirror, and with a modified Mirror that uses our memory management and allocation. The new memory manager can be slower due to being in the NVMM, or faster due to its simple allocator. In addition, the allocated memory is positioned differently, which can influence performance both ways. We tested the differences between the two versions of Mirror, presented in Figure 8.7. The differences are noticeable and can have both positive or negative impact, depending on the data structure. Even at 0% updates there are major differences between the versions. This can be caused by the differences in the memory layout (e.g., [13]). Since a persistent SMR memory manager is an orthogonal problem to our problem of creating persistent data structures, and RELAX could use any such memory manager, we believe that the more interesting measurement is the overhead of RELAX compared to Mirror with the new memory manager, and that is what is measured in this section. By using the new memory manager we simplified the recovery process of the original Mirror by copying the data structure only to DRAM. The original Mirror's recovery time is longer than its recovery time with the new memory manager since it includes more steps or a more complicated interaction with the memory manager.

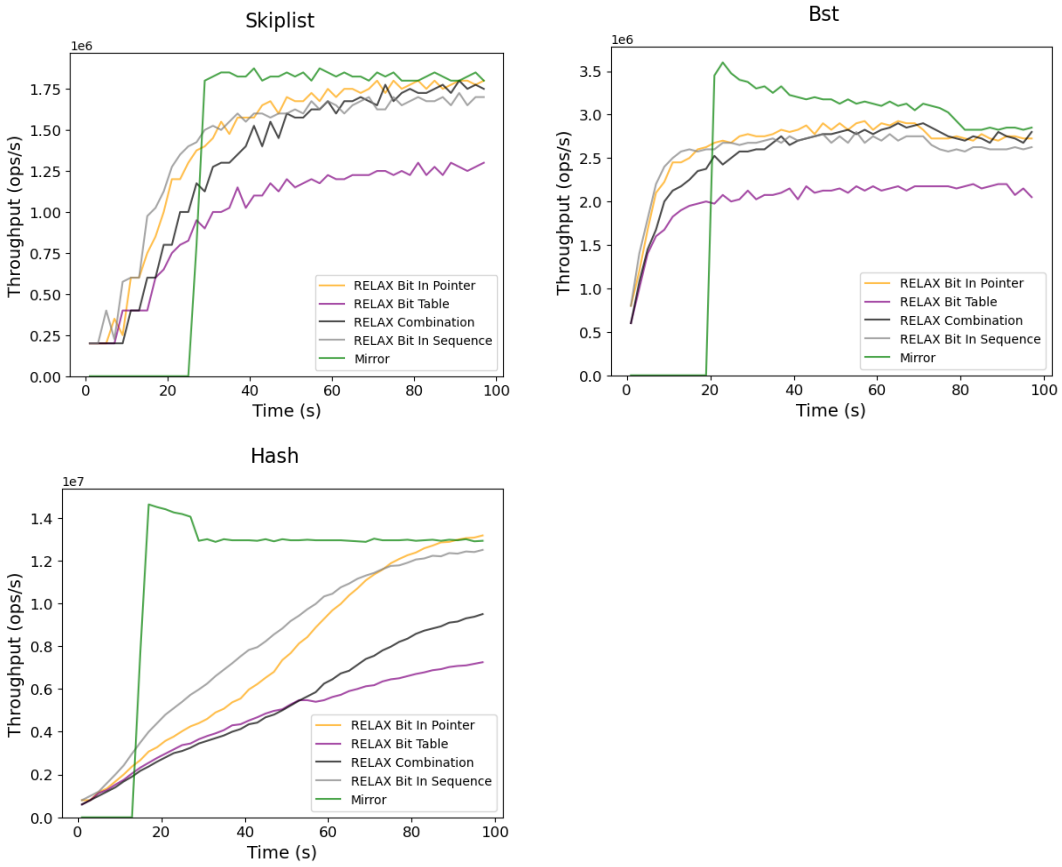


Figure 8.5: recovery - Estimated throughput during recovery over 100 seconds with uniform distribution. Measured using 8 threads, with an update ratio of 20% and a default size of 128M keys.

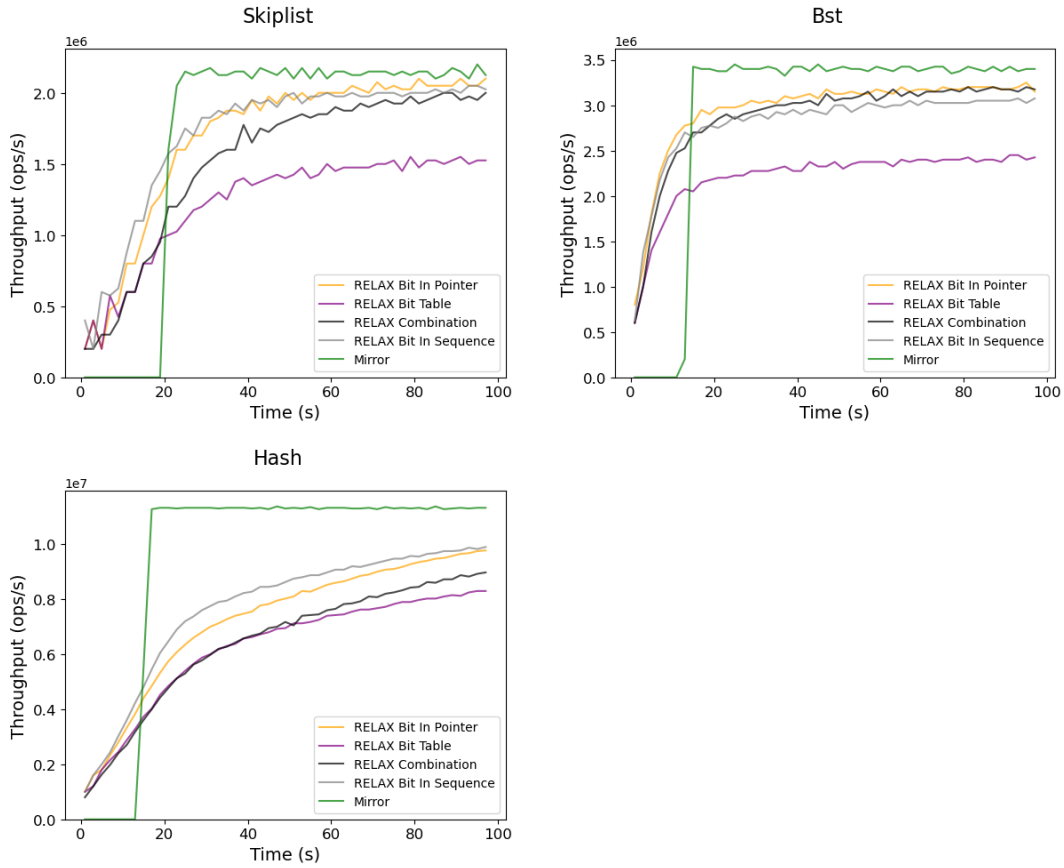


Figure 8.6: Zipfian distribution recovery - Estimated throughput during recovery over 100 seconds with Zipfian distribution. Measured using 8 threads, with an update ratio of 20% and a default size of 128M keys. Keys randomized according to zipfian distribution

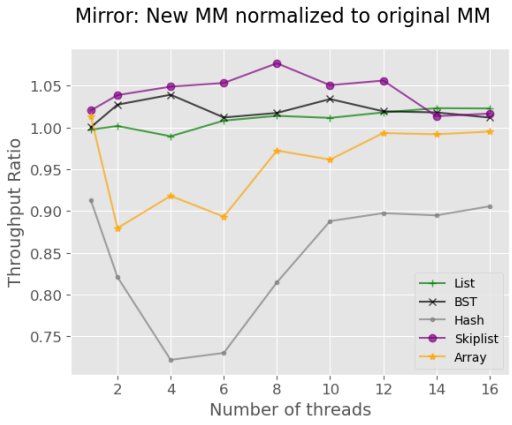


Figure 8.7: Memory manager comparison - Average throughput of Mirror with the new memory manager divided by the original Mirror's average throughput for each data structure. Varying number of threads, with 80% reads. Average size for list is 128 keys, for all other data structures it is 8 M keys.

9 Conclusions

In this paper we presented RELAX, a simple transformation from linearizable lock-free data structures to durably linearizable lock-free data structures. The data structures generated by RELAX minimize the downtime after recovery, while still preserving the high performance of the data structures. RELAX extends Mirror. It can work in the most general manner with all linearizable lock-free data structures, but with some performance degradation. For recursive data structure on widely-used modern operating systems optimizations can be applied to almost entirely eliminate performance overheads, obtaining the best of both worlds: high performance with minimal downtime after a crash. RELAX employs a lazy recovery, that executes concurrently with program execution. So while the original Mirror required a noticeable downtime to recover from a crash (copying the entire data structure to DRAM), RELAX almost immediately enables execution of data structure operations with moderate throughput, by lazily copying only relevant parts of the data structure that were not yet recovered. Upon concurrent completion of the recovery, the data structure returns to serving requests at full speed. Evaluation shows that RELAX indeed provides immediate response after a crash, and that the overhead over the original Mirror during normal (non-crashing) execution is very low.

Bibliography

[1] AMD. (2021) Amd64 architecture programmer's manual. [Online]. Available: https://www.amd.com/system/files/TechDocs/24594.pdf

[2] ARM. (2018) Arm architecture reference manual armv8. [Online]. Available: https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf

[3] N. Ben-David, G. Blelloch, M. Friedman, and Y. Wei, "Delay-free concurrency on faulty persistent memory„" 2019.

[4] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," 2016, pp. 677–694.

[5] C++. (2011) Std::atomic library. [Online]. Available: https://en.cppreference.com/w/cpp/atomic

[6] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, *FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory*, 2020, p. 1077–1091.

[7] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency." Association for Computing Machinery, 2013, p. 228–243.

[8] J. Coburn, A. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *asplos*, 2011.

[9] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," vol. 1. ACM, 2017, p. 67.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," 2009, p. 133–146.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," 2010.

[12] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory." ACM, 2018, pp. 271–282.

[13] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 219–228, mar 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451141

[14] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, "Nvalloc: Rethinking heap metadata management in persistent memory allocators," 2022, p. 115–127.

[15] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," 2018.

[16] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," 2015.

[17] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for openjdk," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–9.

[18] K. Fraser, "Practical lock-freedom," 2003.

[19] M. Friedman, N. Ben-David, Y. Wei, G. Blelloch, and E. Petrank, "Nvtraverse: In nvram data structures, the destination is more important than the journey," 2020, pp. 377—392.

[20] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," vol. 53, no. 1. ACM, 2018, pp. 28–40.

[21] M. Friedman, E. Petrank, and P. Ramalhete, "Mirror: Making lock-free data structures persistent," 2021, pp. 1218—1232.

[22] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: Efficient, htm-compatible persistent transactions," 2020, pp. 59–74.

[23] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: Efficient, htm-compatible persistent transactions," 2020, p. 59–74.

[24] E. Giles, K. Doshi, and P. J. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–14.

[25] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," p. 46–61, 2018.

[26] S. Haria, M. D. Hill, and M. M. Swift, *MOD: Minimally Ordered Durable Datastructures for Persistent Memory*, 2020, p. 775–788.

[27] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists." Springer, 2001, pp. 300–314.

[28] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," vol. 12, no. 3, pp. 463–492, 1990.

[29] Intel. (2021) Intel architecture instruction set extensions programming reference. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[30] ——. (2022) Developers intel64 and ia-32 architectures software manuals combined. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[31] P. Intel. (2018) Persistent memory programming. [Online]. Available: https://pmem.io

[32] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," p. 427–442, 2016.

[33] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model." Springer, 2016, pp. 313–327.

[34] S. Jayanti and J. Shun, "Fast arrays: Atomic arrays with constant time initialization," in *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[35] J. Jeong and C. Jung, *PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency)*, 2021, p. 517–529.

[36] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," 2016, pp. 399–411.

[37] R. M. Krishnan, W.-H. Kim, X. Fu, S. K. Monga, H. W. Lee, M. Jang, A. Mathew, and C. Min, "Tips: Making volatile index structures persistent with dram-nvmm tiering," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 773–787.

[38] P. Lidén and S. Karlsson, "The z garbage collector," 2018.

[39] Linux. (2021) mmap(2) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html

[40] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory." ACM, 2017, pp. 329–343.

[41] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory." IEEE, 2018, pp. 258–270.

[42] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," 2020, p. 789–806.

[43] Microsoft. (2021) CreateFileMappingA function (winbase.h). [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga

[44] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, p. 94–162, 1992.

[45] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees." ACM, 2014.

[46] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard, "Stopless: a real-time garbage collector for multiprocessors," in *Proceedings of the 6th international symposium on Memory management*, 2007, pp. 159–172.

[47] F. Pizlo, E. Petrank, and B. Steensgaard, "A study of concurrent real-time garbage collectors," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 33–44.

[48] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, no. POPL, 2019.

[49] ——, "Persistency semantics of the intel-x86 architecture," 2019.

[50] A. Raad, J. Wickerson, and V. Vafeiadis, "Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models," 2019.

[51] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," 2017, p. 91–104.

[52] T. Shull, J. Huang, and J. Torrellas, "Autopersist: An easy-to-use java nvm framework based on reachability," 2019, p. 316–332.

[53] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proceedings of the international symposium on Memory management*, 2011, pp. 79–88.

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

[55] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," vol. 39, no. 1. ACM, 2011, pp. 91–104.

[56] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Luján, "Pmthreads: Persistent memory threads harnessing versioned shadow copies," 2020, p. 623–637.

[57] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: Log less, re-execute more," 2021, p. 346–359.

[58] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank, "Efficient lock-free durable sets," 2019.

המערכת לכמה שיותר סביבות הרצה אפשריות, וכיוון שקיימות מערכות הפעלה שלא מבטיחות מראש מה יהיה בזיכרון לאחר קריסה, בחרנו להניח שלאחר קריסה הזיכרון הרגיל יכול להכיל כל ערך אפשרי. הנחה זו, ביחד עם ריבוי העותקים והאתגרים הרגילים של מקביליות דורשים אלגוריתם העתקה מיוחד שמוודא את נכונות ריצת מבנה הנתונים על ידי מניעת חזרת הערך בזיכרון.

פיתחנו 4 שיטות מעקב עיקריות. ראשית, שיטה כללית לחלוטין שמשתמשת במערך שמאותחל בזמן קבוע על מנת לעקוב אחרי אילו שדות הועתקו ואילו עדיין לא הועתקו. שנית, אם ידוע לנו כי מרחב הזיכרון הרגיל מאותחל ל0 בתחילת הריצה, תכונה שנפוצה למערכות הפעלה רבות מסיבות אבטחה, אזי אנחנו יכולים לסמן את השדות שהועתקו באותה שורת מטמון. כך אנחנו יכולים לבדוק את מצב השדה על ידי גישה למטמון ולא לזיכרון, וכך להקטין משמעותית את תקורת המערכת. אם ידוע לנו כי מבנה הנתונים מורכב מצמתים המחוברים על ידי מצביעים, כגון רשימה מקושרת או עץ חיפוש, אז אנחנו יכולים לעבוד ברזולוצית הצומת ולעקוב אחרי מצב ההעתקה של כל צומת במצביע אליו. היתרון בשיטה זו הוא שאפילו לא צריך לגשת למטמון אלא ניתן לקרוא רק את ערך המצביע, וכל שאר הערכים לא יוצרים תקורה כלל. החיסרון הוא שיכולים להיות מספר מצביעים עבור צומת יחיד, ולכן צריך מנגנון נוסף שימנע העתקות מיותרות. השיטה השלישית אותה פיתחנו משתמשת במעקב דרך המצביעים ומשלבת מערך המאותחל בזמן קבוע על מנת למנוע העתקות מיותרות. השיטה הרביעית מתאימה במקרה ומבנה הנתונים מורכב מצמתים המחוברים על ידי מצביעים ובנוסף לכך ידוע כי הזיכרון מאותחל ל0 לאחר קריסה. בשיטה הרביעית אנחנו משתמשים בידע שלנו על הסביבה על מנת לחסוך את המנגנון הנוסף ורק לאחסן את מצב ההעתקה של הצומת על המצביע אליו.

בניסויים שביצענו מצאנו כי השיטה הראשונה גורמת לתקורה של $10-30\%$ על פני Mirror , השיטה השנייה גורמת לתקורה של $5-10\%$ בעוד השיטות השלישית והרביעית גורמות לתקורה של אחוזים בודדים. בנוסף מצאנו כי המנגנון העצל שימושי במיוחד במבני נתונים היררכיים, כגון עץ חיפוש שבהם יש יותר גישות לצמתים העליונים, וכאשר התפלגות המפתחות אינה אחידה, לדוגמה בהתפלגות Zipfian . כאשר הגישות אינן אחידות המנגנון העצל מחזיר את החלקים החשובים במהירות וחוזר לתפוקה כמעט מלאה בעוד ש Mirror נאלץ להעתיק צמתים בעלי חשיבות נמוכה.

תקציר

בימינו כולם מסתמכים על מערכות מחשבים. לכן אנחנו שואפים לבנות מערכות עמידות ככל האפשר. אבני הבניין למערכות עמידות הם מבני נתונים עמידים. בעבר מבני נתונים עמידים דרשו הסתמכות על כוננים קשיחים, שהציבו תקורה משמעותית. כיום ניתן לבחור בזיכרון עמיד, שמספק אחסון עמיד עם שבריר מאובדן הביצועים. מבני נתונים עמידים שמסתמכים על זיכרון עמיד מתנהגים יותר כמו מבני נתונים בזיכרון גישה אקראית בממשק ובביצועים שלהם לעומת מבני הנתונים העמידים שמסתמכים על מכשירי אחסון בבלוקים כגון טכנולוגית SSD .

השימוש בזיכרון עמיד אינו טריוויאלי, שכן המטמונים אינם עמידים ולכן בעת קריסה תוכנם נאבד. המטמונים מעבירים את תוכנם לזיכרון בסדר לא צפוי, שיכול להשאיר את הזיכרון העמיד במצב לא תקין. על מנת לשלוט בסדר הפינוי של המטמונים לזיכרון ניתן לפנות באופן יזום מידע מהמטמון לזיכרון. פינוי זה לוקח זמן רב ולכן צריך למזער את השימוש בו.

איזון זה בין ביצועים לנכונות מקשה על עיצוב מבני נתונים עמידים שמסתמכים על זיכרון עמיד, ודורש ידע מומחה. קיימות שיטות הנקראות בניות כלליות שמאפשרות להמיר מבנה נתונים לא עמיד למבנה נתונים עמיד ללא ידע מומחה. בניות כלליות שימושיות כיוון שלאחר שנבנו אין צורך בידע מומחה על מנת לבנות מבני נתונים עמידים, במחיר של אובדן ביצועים.

נעשה מחקר רב על איך לשפר את הביצועים של בניות כלליות, בשיטות כגון טרנסקציות או רישום פעולות. שיטה נוספת מסתמכת על העובדה שזיכרון רגיל עדיין מהיר יותר מזיכרון עמיד, ולכן אחסון עותק של מבנה הנתונים בזיכרון רגיל מאפשר להאיץ חלק מהפעולות במחיר של האטת השאר.

כל השיטות הנ"ל משאירות חלק מהעבודה לאחרי הקריסה, בין אם מדובר בשחזור הפעולות הרשומות או העתקת מבנה הנתונים חזרה לזיכרון הרגיל שנמחק. כתוצאה מכך לאחר קריסה מבנה הנתונים לא נגיש מיד. זמן המתנה זה יכול לפגוע בשימושיות של מבני נתונים עמידים, ולכן ברצוננו למזערו. עבודות עד כה לא התמקדו על אספקט זה של מבני נתונים עמידים ולכן כיום יש לבחור בין ביצועים סבירים וזמן תגובה נמוך.

לצורך כך אנחנו מציגים את המערכת שלנו RELAX , שמאפשרת גישה כמעט מיידית למבני הנתונים לאחר קריסה, תוך שמירה על הספק גבוה בזמן ריצה רגילה. RELAX נבנתה על גבי Mirror , בנייה כללית המאפשרית להמיר מבנה נתונים חופשי ממנעולים למבנה נתונים עמיד חופשי ממנעולים.

Mirror מספקת ביצועים מהגבוהים בתחום, על ידי שיקוף מבנה הנתונים העמיד בזיכרון הרגיל, וקריאת נתונים מהזיכרון הרגיל וכתיבתו גם למבנה הנתונים וגם לשיקוף שלו. Mirror שומרת על נכונות מבני הנתונים על ידי עדכון מתוחכם של שתי הגרסאות. החיסרון העיקרי של Mirror הוא זמן ההמתנה הארוך לאחר קריסה, שבו יש לעבור על מבנה הנתונים ולהעתיק אותו לזיכרון הרגיל.

RELAX מבצעת את ההעתקה באופן עצל ברזולוצית השדה, וכך מאפשרת גישה למבנה הנתונים ישר לאחר הקריסה, במחיר הארכת תקופת ההחלמה שבה הביצועים ירודים. ל RELAX יש שני אתגרים מרכזיים: ראשית, RELAX משתמשת באלגוריתם העתקה מיוחד על מנת למנוע קונפליקטים בין חוטים שונים בעת תקופת ההחלמה. שנית, RELAX מבצעת מעקב על אילו שדות כבר הועתקו על מנת למנוע העתקה חוזרת. שיטת מעקב זו גורמת לרוב התקורה בזמן ריצה רגילה ולכן פיתחנו מספר שיטות מעקב, בהתאמה למגוון סביבות הרצה ומבני נתונים על מנת למקסם את הביצועים בזמן ריצה רגילה. על מנת להתאים את

המחקר נעשה בהנחיית פרופ' ארז פטרנק בפקולטה למדעי המחשב.

מחבר חיבור זה מצהיר כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו',נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

התאוששות עצלה מריצה כושלת בעזרת זיכרון עמיד

חיבור על מחקר לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אלמוג צור

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
אדר ה'תשפ"ג, חיפה, מרץ 2023

התאוששות עצלה מריצה כושלת בעזרת זיכרון עמיד

אלמוג צור