Dalí: A Periodically Persistent Hash Map

Faisal Nawab*[1], Joseph Izraelevitz*[2], Terence Kelly*[3], Charles B. Morrey III*[4], Dhruva R. Chakrabarti*[5], and Michael L. Scott[6]

1 **University of California, Santa Barbara, CA, USA**
nawab@cs.ucsb.edu
2 **University of Rochester, NY, USA**
jhi1@cs.rochester.edu
3 **Palo Alto, CA, USA**
4 **Woodside, CA, USA**
5 **Palo Alto, CA, USA**
dhruvac@gmail.com
6 **University of Rochester, NY, USA**
scott@cs.rochester.edu

Abstract

Technology trends suggest that byte-addressable nonvolatile memory (NVM) will supplant many uses of DRAM over the coming decade, raising the prospect of inexpensive recovery from power failures and similar faults. Ensuring the consistency of persistent state remains nontrivial, however, in the presence of volatile caches; cached values can "leak" back to persistent memory in arbitrary order. To ensure consistency, existing persistent memory algorithms use expensive, explicit write-back instructions to force each value back to memory before performing a dependent write, thereby incurring significant run-time overhead.

To reduce this overhead, we present a new design paradigm that we call *periodic persistence*. In a periodically persistent data structure, updates are made "in place," but can safely leak back to memory *in any order*, because only those updates that are known to be valid will be heeded during recovery. To guarantee forward progress, we periodically force a write-back of all dirty data in the cache, ensuring that all "sufficiently old" updates have indeed become persistent, at which point they become semantically visible to the recovery process.

As an example of periodic persistence, we present a transactional hash map, Dalí, together with an informal proof of safety (buffered durable linearizability). Experiments with a prototype implementation suggest that periodic persistence can offer substantially better performance than either file-based or incrementally persistent (per-access write-back) alternatives.

1998 ACM Subject Classification D.3.3 Concurrent Programming Structures

Keywords and phrases data structure, nonvolatile memory, durable linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.37

1 Introduction

For decades, programmers have been accustomed to partitioning program state into *memory*, which is transient – used during a single program run – and *storage*, which is persistent –

* This work was supported in part by the US Department of Energy under Cooperative Agreement no. DE-SC0012199 while the indicated authors were members of Hewlett Packard Labs. At the University of Rochester, the work was supported in part by NSF grants CNS-1319417, CCF-1337224, and CCF-1422649, and by a Google Faculty Research award.

intended for use across program runs and even system crashes. The design of data structures is rooted in the use of memory; data in storage is typically relegated to a file or database.

Since the 1970s, memory has been virtually synonymous with DRAM, accessed (since the 1980s) through a rich hierarchy of caches. Storage has been the province of magnetic disks or, more recently, flash drives. Several new memory technologies, however, promise to provide byte-addressable nonvolatile memory (NVM) with access latencies and costs comparable to those of DRAM. These technologies provide the opportunity to re-think the memory–storage divide, and to entertain the possibility of maintaining traditional in-memory data structures across program runs and crashes.

We are particularly interested in crashes, as they present unique consistency challenges. For simplicity, and in keeping with the real-world common case, we assume a "whole system crash" failure model (caused, for example, by power failure or an OS kernel panic). We wish to ensure, in the wake of a crash, that data in memory are consistent. At first blush, it is tempting to model this goal as a conventional concurrency problem: "normal" execution entails one or more threads performing atomic updates to the data; a *recovery procedure* runs in the wake of a crash and (since the crash can occur at any time) functions as if it were merely an additional concurrent thread (with the possible simplifying assumption that it runs in isolation).

The problem with this model is that the recovery procedure does not have access to the view of memory shared by threads during normal execution. Caches are likely to remain volatile, at least for the foreseeable future, so what the recovery procedure sees is whatever has been written back to nonvolatile memory prior to the crash. Unfortunately, hardware capacity and associativity constraints require that caches be permitted to perform their writes-back in essentially arbitrary order. When this order differs from the happens-before order of the running program, the values that happen to have "leaked back" to memory at any particular time may be mutually inconsistent. If, for example, a program creates an object and then aims a pointer at it, it is possible for the pointer to reach memory before the object to which it points. Persistent data structures must be carefully designed to avoid this sort of problem.

In current real-world processors, instructions to control the ordering, timing, and granularity of writes-back from caches to memory are rather limited. On Intel processors, for example, the CLFLUSH instruction [16] takes an address as argument, and blocks until the cache line containing the address has been both evicted from the cache and written back to the memory controller. When combined with an MFENCE instruction to prevent compiler and processor instruction reordering, CLFLUSH allows the programmer to force a write-back that is guaranteed to *persist* (reach nonvolatile memory) before any subsequent store. The overhead is substantial, however – on the order of hundreds of cycles. Future processors may provide less expensive persistence instructions, such as the `pwb`, `pfence`, and `psync` assumed in our earlier work [17], or the `ofence` and `dfence` of Nalli et al. [21]. Even in the best of circumstances, however, "persisting" an individual store (and ordering it relative to other stores) is likely to take time comparable to a memory consistency fence on current processors – i.e., tens of cycles. Due to power constraints [8], we expect that writes and flushes into NVM will be guaranteed to be failure-atomic only at increments of eight bytes – not across a full 64-byte cache line.

We use the term *incremental persistence* to refer to the strategy of persisting store w_1 before performing store w_2 whenever w_1 occurs before w_2 in the happens-before order of the program during normal execution (i.e., when $w_1 <_{hb} w_2$). Given the expected latency of even an optimized persist, this strategy seems doomed to impose significant overhead on the operations (method calls) of any data structure intended to survive program crashes.

As an alternative, we introduce a strategy we refer to as *periodic persistence*. The key to this strategy is to design a data structure in such a way that modifications can safely leak into persistence *in any order*, removing the need to persist locations incrementally and explicitly as an operation progresses. To ensure that an operation's stores eventually become persistent, we periodically execute a *global fence* that forces all cached data to be written back to memory. The interval between global fences bounds the amount of work that can ever be lost in a crash (though some work may be lost). To avoid depending on the fine-grain ordering of writes-back, we arrange for "leaked" lines to be ignored by any recovery procedure that executes before a subsequent global fence. After the fence, however, a known set of cache lines will have been written back, making their contents safe to read. Like naive uninstrumented code, periodic persistence allows stores to persist out of order. It guarantees, however, that the recovery procedure will never use a value v from memory unless it can be sure that all values on which v depends have also safely persisted.

In contrast to checkpointing, which creates a consistent *copy* of data in nonvolatile memory, periodic persistence maintains a *single* instance of the data for both the running program and the recovery procedure. This single instance is designed in such a way that recent updates are nondestructive, and the recovery procedure knows which parts of the data structure it can safely use.

In some sense, periodically persistent structures can be seen as an adaptation of traditional *persistent data structures* [12] (in a different sense of the word "persistent") or of multiversion transactional memory systems [3], both of which maintain a history of data structure changes over time. In our case, we can safely discard old versions that predate the most recent global fence, so the overall impact on memory footprint is minimal. At the same time, we must ensure not only that the recovery procedure ignores the most recent updates but also that it is never confused by their potential structural inconsistencies.

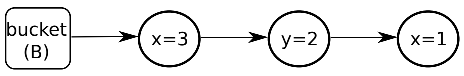
As an example of periodic persistence, we introduce Dalí,[1] a transactional hash map for nonvolatile memory. Dalí demonstrates the feasibility of using periodic persistence in a nontrivial way. Experience with a prototype implementation confirms that Dalí can significantly outperform alternatives based on either incremental or traditional file-system-based persistence. Our prototype implements the global fence by flushing (writing back and invalidating) all on-chip caches. Performance results would presumably be even better with hardware support for whole-cache write-back without invalidation.

The remainder of this paper is organized as follows: Section 2 elaborates on the motivation for our work in the context of persistent hash maps. We describe Dalí's design in Section 3 and prove its correctness in Section 4. Section 5 then presents experimental results. Section 6 reviews related work. Section 7 summarizes our conclusions.

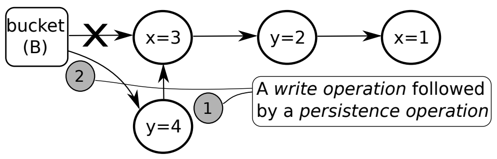
2 Motivation

As a motivating example, consider the construction of a persistent hash map, beginning with the nonblocking structure of Schwalb et al. [24]. To facilitate transactional update of entries in multiple buckets, we switch to a blocking design with a lock in each bucket, enabling the use of two-phase locking (and, for atomicity in the face of crashes, undo logging).

[1] The name is inspired by Dalí's painting *The Persistence of Memory*.



(a) A bucket containing three records.



(b) An example of the write-ordering overhead entailed in updating a data object.

Figure 1 A hash map data structure that demonstrates the overhead of write ordering.

This hash map, which is incrementally persistent, consists of an array of buckets, each of which points to a singly-linked list of *records*. Each record is a key-value pair. Figure 1a shows a bucket with three records. For the sake of simplicity, each list is prepend-only: records closer to the head are more recent. It is possible that multiple records exist for the same key – the figure shows two records for the key x, for instance, but only the most recent record is used. Deletions are handled by inserting a "not present" record. Garbage collection / compaction can be handled separately; we omit the description here.

Figure 1b shows an update to change the value of y to 4. The update comprises several steps: (1a) A record, r_{new} with the new key-value pair is written. The record points to the current head of the list. (1b) A persist of r_{new} serves to push its value from cache to NVM. (2a) The bucket list head pointer, B, is overwritten to point to r_{new}. (2b) A second persist pushes B to NVM. The first persist must complete before the store to B: it prevents the incorrect recovery state in which r_{new} is not in NVM and B is a dangling pointer. The second persist must complete before the operation that updates y returns to the application program: it prevents misordering with respect to subsequent operations.

On current hardware, a persist operation waits hundreds of cycles for a full round trip to memory. On future machines, hardware support for ordered (queued) writes-back might reduce this to tens of cycles. Even so, incremental persistence can be expected to increase the latency of simple operations several-fold. The key insight in Dalí is that when enabled by careful data structure design, periodic persistence can eliminate fine-grain ordering requirements, replacing a very large number of single-location fences with a much smaller number of global fences, for a large net win in performance, at the expense of possible lost work. In practice, we would expect the frequency of global fences to reflect a trade-off between overhead and the amount of work that may be lost on a crash. Fencing once every few milliseconds strikes us as a good initial choice.

3 Dalí

Dalí is our prepend-only transactional hash map designed using periodic persistence. It can be seen as the periodic persistence equivalent of the incrementally persistent hash map of Section 2 and Figure 1. As a transactional hash map, Dalí supports the normal `get`, `set`, `delete`, and `replace` methods. It also supports ACID transactions comprising any number of the above methods.

Dalí updates or inserts by prepending a record to the appropriate bucket; the most recent record for a key is the one closest to the head of the list (duplicates may exist, but only the most recent record matters). Records in a bucket are from time to time consolidated to remove obsolete versions. Dalí employs per-bucket locks (mutexes) for isolation. A variant of strong strict two-phase locking (SS2PL) is used to implement transactions.

```
class node:
  key k; val v
  node* next
class bucket:
  mutex lock
  int stat<a, f, c, ss> // 2/2/2/58 bits
  node* ptrs[3]
class dali:
  bucket buckets[N_BUCKTS]
  int list flist
  int epoch
```

Figure 2 Dalí globals and data types.

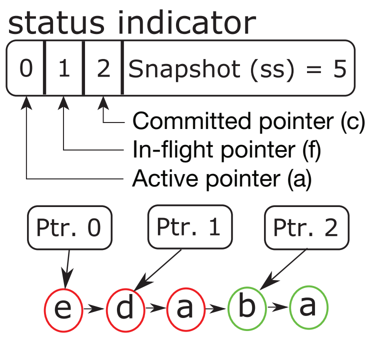


Figure 3 The structure of a bucket.

3.1 Data Structure Overview

As mentioned above, Dalí uses a periodic global fence to guarantee that changes to the data structure have become persistent. The fence is invoked by a special worker thread in parallel with normal operation by application threads. We say that the initiation points of the global fences divide time into *epochs*, which are numbered monotonically from the beginning of time (the numbers do not reset after a crash). Each update (or transactional set of updates) is logically confined to a single epoch, and the fence whose initiation terminates epoch E serves to persist all updates that executed in E. The execution of the fence, however, may overlap the execution of updates in epoch $E+1$. The worker thread does not initiate a global fence until the previous fence has completed. As a result, in the absence of crashes, we are guaranteed during epoch $E+1$ that any update executed in epoch $E-1$ has persisted. If a crash occurs in epoch F, however, updates from epochs F and $F-1$ cannot be guaranteed to be persistent, and should therefore be ignored. We refer to epochs F and $F-1$ as *failed epochs*, and revise our invariant in the presence of crashes to say that during a given epoch E, all updates performed in a non-failed epoch prior to $E-1$ have persisted. Failed epoch numbers are maintained in a persistent *failure list* that is updated during the recovery procedure.

In Dalí, hash map records are classified according to their persistence status. Assume that we are in epoch E. *Committed records* are ones that were written in a non-failed epoch at or before epoch $E-2$. *In-flight records* are ones that were written in epoch $E-1$ if it is not a failed epoch. *Active records* are ones that were written during the current epoch E. Records that were written in a failed epoch are called *failed records*. By steering application threads around failed records, Dalí ensures consistency in the wake of a crash.

Dalí's hash map buckets are similar in layout to those of the incrementally persistent hash map presented in Figure 1. Dalí adds metadata to each bucket, however, to track the persistence status of the bucket's records. The metadata in turn allows us to avoid persisting records incrementally. Specifically, a Dalí bucket contains not only a singly-linked list of records, but also a 64-bit *status indicator* and, in lieu of a head pointer for the list of records, a set of three *list pointers* (see pseudocode in Figure 2 and illustration in Figure 3). The status indicator comprises a *snapshot* (*SS*) field, denoting the epoch in which the most recent record was prepended to the bucket, and three 2-bit *role IDs*, which indicate the roles of the three list pointers. A single STORE suffices to atomically update the status indicator on today's 64-bit machines.[2]

[2] With 6 bits devoted to role IDs, 58 bits remain for the epoch number. If we start a new epoch every millisecond, roll-over will not happen for 9 million years.

Each of the three list pointers identifies a record in the bucket's list (or NULL). The pointers assume three roles, which are identified by storing the pointer number (0, 1, or 2) in one the three role ID fields of the status indicator. Roles are fixed for the duration of an epoch but can change in future epochs. The roles are:

Active pointer (a): provided that epoch SS has not failed, identifies the most recently added record (which must necessarily have been added in SS). Each record points to the record that was added before it. Thus, the active pointer provides access to the entire list of records in the bucket.

In-flight pointer (f): provided that epochs SS and $SS-1$ have not failed, identifies the most recent record, if any, added in epoch $SS-1$. If no such record exists, the in-flight role ID is set to invalid ($\perp$).

Committed pointer (c): identifies the most recent record added in a non-failed epoch equal to or earlier than $SS-2$.

To establish these invariants at start-up, we initialize the global `epoch` counter to 2 and, in every bucket, set SS to 0, all pointers to NULL, the in-flight role ID to $\perp$, and the active and committed IDs to arbitrary values.

Figure 3 shows an example bucket. In the figure SS is equal to 5, which means that the most recent record was prepended during epoch 5. The active pointer is Pointer 0. It points to record e, which means that e was added in epoch 5, even if we are reading the status indicator during a later epoch. Pointer 1 is the in-flight pointer, which makes d the most recently added record in epoch 4. Because a record points only to records that were added before it, by transitivity, records a, b, and the prior a were added before or during epoch 4. Finally, Pointer 2 is the committed pointer. This makes record b the most recently added record before or during epoch 3. By transitivity, the earlier record a was also added before or during epoch 3. Both record b and the earlier record a are therefore guaranteed persistent (shown in green) as of the most recent update (the time at which e was added), while the remainder of the records may not be persistent (shown in red).

It is important to note that the status indicator reflects the bucket's state at SS (the epoch of the most recent update to the bucket) even if a thread inspects the bucket during a later epoch. For example, suppose that a thread in epoch 10 reads the bucket state shown in Figure 3. Given the status indicator, the thread will conclude that all records were written during or before epoch 5 and thus are all committed and persistent (assuming that epochs 4 and 5 are not in the failure list). If one or both epochs are on the failure list, the thread can navigate around their records using the in-flight or committed pointers.

3.2 Reads

The task of the read method is to return the value, if any, associated with a given key. A reader begins by using a hash function to identify the appropriate bucket for its key, and locks the bucket. It then consults the bucket's epoch number (SS) and the global failed epoch list to identify the most recent, yet valid, of the three potential pointers into the bucket's linked list (Figure 4). Call this pointer the *valid head*. If SS is not a failed epoch, the valid head will be the active pointer, which will identify the most recently added record (which may or may not yet be persistent). If SS is a failed epoch but $SS-1$ is not, the valid head will be the in-flight pointer. If SS and $SS-1$ are both failed epochs, the valid head will be the committed pointer.

Starting from the valid head, a reader searches records in order looking for a matching key. Because updates to the hash map are prepends, the most recent matching record will

```
// Bucket is assumed locked via SS2PL
val bucket::read(key k):
  node* valid_head =
    if ss ∉ flist then ptrs[a]
    elsif ss-1 ∉ flist && f ≠ ⊥ then ptrs[f]
    else ptrs[c]
  return search(k, valid_head)
```

Figure 4 Dalí read method.

```
// Bucket is assumed locked via SS2PL
void bucket::update(key k, val v):
  bool curr_fail = ss ∈ flist
  bool prev_fail =
      ss-1 ∈ flist || f == ⊥
  node* valid_head =
    if !curr_fail then ptrs[a]
    elsif !prev_fail then ptrs[f]
    else ptrs[c]
  node* n = new node(k, v, valid_head)

  // Get new pointer roles from table
  int new_stat = lookup(epoch,
      curr_fail, prev_fail, stat)
  ptrs[new_stat.a] = n
  stat = new_stat
```

Figure 5 Dalí update method.

	SS	SS ∈ flist	SS −1 ∈ flist or f = ⊥	new a	new f	new c
1	E	N/A	N/A	a	f	c
2	$E-1$	✗	✗	c	a	f
3	$E-1$	✗	✓	f	a	c
4	$E-1$	✓	N/A	a	⊥	c
5	$< E-1$	✗	N/A	c	⊥	a
6	$< E-1$	✓	✗	a	⊥	f
7	$< E-1$	✓	✓	a	⊥	c

Figure 6 Lookup table for pointer role assignments. Current epoch is E.

be found first. If the key has been removed, the matching value may be NULL. If the key is not found in the list, the value returned from the read will also be NULL.

3.3 Updates

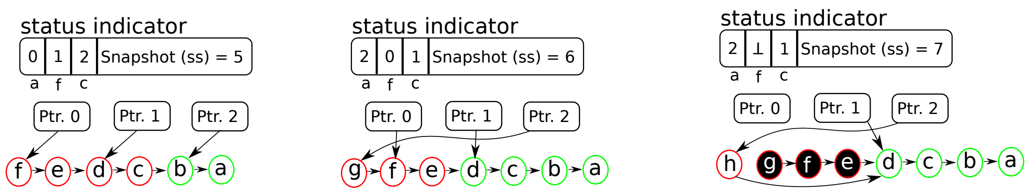
Updates in Dalí prepend a new version of a record, as in the incrementally persistent hash map of Section 2. Deletions / overwrites of existing keys and inserts of new keys are processed identically by a unified `update` method. Like the `read` method, `update` locks the bucket. An update to a Dalí bucket comprises several steps:

1. Determine the most recent, valid pointer (as in the `read` method).
2. Create a new record with the key and its new value (or NULL if a `remove`).
3. Determine the new pointer roles (if the new and old epochs are different).
4. Retarget the new active pointer to the new record node.
5. Update *SS* and the role IDs by overwriting the status indicator.

Pseudocode appears in Figure 5.

Step 3 is the most important part of the update algorithm, as it is the part that allows the update's component writes to be reordered. The problem to be addressed is the possibility that writes from neighboring epochs might be written back and become mixed in the persistent state. We might, for example, mix the snapshot indicator from the later epoch with the pointer values from the earlier epoch. Given any combination of update writes from bordering epochs, and an indication of epoch success or failure, the `read` procedure must find a correct and valid head, and the list beyond that head must be persistent.

The details of step 3 appear in Figure 6. They are based on the following three rules. First, the new committed pointer was last written at least two epochs prior, guaranteeing



(a) Initial state in epoch 5. **(b)** Adding record g in epoch 6. **(c)** Adding record h in epoch 7; epochs 5 and 6 have failed.

Figure 7 A sequence of Dalí updates.

that its value and target have become persistent (and would survive a crash in the current epoch). Second, the new active pointer was either previously invalid or pointed to an earlier record than the new committed pointer. In other words, according to both the old and new status indicators, the new active pointer will never be a valid head, so it is safe to reassign. Third, the new in-flight pointer is the most recent valid record set in the previous epoch, or $\perp$ if no such record exists. These rules are sufficient to enumerate all entries in the table.

Because each bucket is locked throughout the update method, there is no concern about simultaneous access by other active threads. We assume that each of the two key writes in an update – to a pointer and to the status indicator – is atomic with respect to crashes, but the order in which these two writes persist is immaterial: neither will be inspected in the wake of a crash unless the global epoch counter has advanced by 2.

Figure 7 displays two example updates. In Figure 7a, an update to the bucket has occurred in epoch 5. In Figure 7b, record g is added to the bucket in epoch 6. First, we initialize the new record to point to the most recent valid record, f. Then, we change the status indicator to update pointer roles and the epoch number. As we are in epoch 6, the most recent committed record was added in epoch 4 (the previous in-flight pointer). Therefore, pointer 1 is now the committed pointer. The new in-flight pointer is the one pointing to the most recent record added in the previous epoch (pointer 0). The remaining pointer, pointer 2, whose target is older than the new committed pointer, is then assigned the active role and is retargeted to point to the newly prepended record, g.

In Figure 7c, an additional record, h, is added to the bucket after a crash has occurred in epoch 6 (after the update of Figure 7b). Because of the crash, epochs 5 and 6 are on the failure list. Records e, f, and g are thus failed records, because they were added during these epochs and cannot be relied upon to have persisted. The new record, h, refers to the valid head d instead. Then, the status indicator is updated. The snapshot number SS becomes 7. The committed pointer is the one pointing to the most recent persistent record, d. Pointer 1, which points to d, is assigned the committed role. One currently invalid pointer (pointer 2) will point to the newly added record, h. Since the previous epoch is a failed one, there are no in-flight records, so we set the in-flight role as invalid. The net effect is to transform the state of the bucket in such a way that the failed records, e, f, and g, become unreachable.

3.4 Further Details

Global Routines. As noted in Section 3.1, our global fences are executed periodically by a special worker thread (or by a repurposed application thread that has just completed an operation). The worker first increments and persists the global epoch counter under protection of a sequence lock [19]. It then waits for all threads to exit any transaction in the previous epoch, thereby ensuring that every update occurs entirely within a single epoch. (The wait employs a global array, indexed by thread ID, that indicates the epoch of the thread's current transaction, or 0 if it is not in a transaction.) Finally, the worker initiates the actual whole-cache write-back. In our prototype implementation, this is achieved with a custom system call that executes the Intel WBINVD instruction. This instruction has the side effect of invalidating all cache content. We hypothesize that future machines with persistent memory will provide an alternative instruction that avoids the invalidation.

Following a crash, a *recovery procedure* is invoked. This routine reads the value, F, of the global epoch counter and adds both F and $F-1$ to the failed epoch list (and persists these additions). The crashed epoch, F, is added because the fence that would have forced its writes-back did not start; the previous epoch, $F-1$, is added because the fence that would have forced its writes-back may not have finished. Significantly, the recovery procedure does not delete or modify failed records in the hash chains: as illustrated in Figure 7c, recovery is performed incrementally by application threads as they access data.

Transactions. Transactions are easily added on top of the basic Dalí design. Our prototype employs strong strict two-phase locking (SS2PL): to perform a transaction that includes multiple hash map operations, a thread acquires locks as it progresses, using timeout to detect (conservatively) deadlock with other threads. To preserve the ability to abort (when deadlock is suspected), it buffers its updates in transient state. When it has completed its code, including successful acquisition of all locks, it performs the buffered updates, as described in Section 3.3, and releases all its locks.

In-place Updates. A reader executing in epoch E is interested only in the most recent update of a given key k in E. If there are multiple records for k in E, only the most recent will be used. As a means of reducing memory churn, we modify our update routine to look for a previous entry for k in the current epoch, and to overwrite its associated value, atomically and in place, if it is found.

Multiversioning. Because historical versions are maintained, we can execute read-only operations efficiently, without the need for locking, by pretending that readers execute two epochs in the past, seeing the values that would persist after a crash. This optimization preserves serializability but not strict serializability. It improves throughput by preventing readers from interfering with concurrent update transactions. To ensure consistency, read-only transactions continue to participate in the global array that stalls updates in a new epoch until transactions from the previous epoch have completed.

Garbage Collection. Garbage collection recycles obsolete records that are no longer needed because newer persistent records with the same key exist; it operates at the granularity of a bucket. At the end of an update operation, before releasing the bucket's lock, a thread will occasionally peruse the committed records and identify any for which there exists a more recent committed record with the same key. Removal from the list entails a single atomic pointer update, which is safe as the bucket is locked. Once the removal is persistent (two epochs later), the record can safely be recycled. If memory pressure is detected, we can use incremental persistence to free the record immediately. Otherwise we keep the record on a "retired" list and reclaim it in the thread's first operation two epochs hence.

Because the retired list is transient, we must consider the possibility that records may be lost on a crash, thereby leaking memory. Similar concerns arise when bypassing failed records during an `update` operation, as illustrated in Figure 7b, and when updating the free list of the memory allocator itself. To address these concerns, we can end the recovery procedure with a sweep of the heap that reclaims any node not found on a bucket list [2]. Since the amount of leakage is likely to be small, this need not occur on every crash.

4 Correctness

We here present an informal proof of Dalí's safety. Specifically, we argue that it satisfies *buffered durable linearizability* [17], an extension of traditional linearizability that accommodates whole-system crashes. For clarity of exposition (and for lack of space), we consider only `read` and `update` operations, omitting garbage collection, in-place updates, multiversioning, and transactions. We begin by arguing that a crash-free parallel history of Dalí is linearizable. We then show that the operations preserved at a crash represent a consistent cut of the history prior to the crash, so that when crashes and lost operations are removed from the history, what remains is still linearizable.

4.1 Linearizability

The code of Figures 4 and 5 defines a notion of `valid_head` for a Dalí bucket. Let us say that a bucket is *well formed* if `valid_head` points to a finite, acyclic list of nodes. We define the *valid content* of a well-formed bucket to comprise the initial occurrences of keys on this list, together with their associated values.

▶ **Theorem 1.** *In the absence of crashes, Dalí is a linearizable implementation of an unordered map.*

Proof. All Dalí operations on the same bucket acquire the bucket's lock; by excluding one another in time they trivially appear to take effect atomically at a point between their invocation and response. While the roles of the various pointers may rotate at epoch boundaries, inspection of the code in Figure 5 confirms that, in the absence of crashes, each newly created node in `update` links to `ptrs[a]` (which is always `valid_head`), and `ptrs[a]` is always updated to point to the new node. A trivial induction (starting with initially empty content) shows that this prepending operation preserves both well formedness and the desired sequential semantics. ◀

4.2 Buffered Durable Linearizability

Buffered durable linearizability [17] extends linearizability to accommodate histories with "full-system" crashes. Such crashes are said to divide a history into *eras*, with no thread executing in more than one era.[3] Information is allowed to be lost in a crash, but only in a consistent way. Specifically, if event e_1 happens before event e_2 ($e_1 <_{hb} e_2$ – e.g., e_1 is a store and e_2 is a load that sees its value), then e_1 cannot be lost unless e_2 is also.

Informally, a history is buffered durably linearizable (BDL) if execution in every era can be explained in terms of information preserved from the consistent cut of the previous era. More precisely, history H is BDL if, for every era ending in a crash, there exists a happens-before consistent cut of the events in that era such that for every prefix P of H, the history P' is linearizable, where P' is obtained from P by removing all crashes and, in all eras other than the last, all events that follow the cut. A concurrent object or system is BDL if all of its realizable histories are.

[3] With apologies to geologists, eras here are generally longer than epochs.

Our BDL proof for Dalí begins with the following lemma:

▶ **Lemma 2.** *An epoch boundary in Dalí represents a consistent cut of the happens-before relation on the hash map.*

Proof. Straightforward: The worker thread that increments the epoch number does so under protection of a sequence lock, and it doesn't release the lock until (a) no thread is still working in the previous epoch and (b) the new epoch number has persisted (so no thread will ever work in the previous epoch again). ◀

Suppose now that we are given a history H comprising read, update, and epoch boundary events, where some of the epoch boundaries are also marked as crashes. The two epochs immediately preceding a crash are said to have *failed*; the rest are *successful*. An update operation is said to be successful if it occurs in a successful epoch and to have failed otherwise. Let us define the "valid content" of bucket B at a point between events in H to mean "a singly linked chain of update records reflecting all and only the successful updates to B prior to this point in H." The following is then our key lemma:

▶ **Lemma 3.** *For any realizable history H of a Dalí bucket B, and any prefix P of H ending with a successful update u, `ptrs[a]` will refer to valid content immediately after u.*

Proof. By induction on successful updates. We can ignore the reads in H as they do not change state. As a base case, we adopt the convention that the initial state of B represents the result of a successful initialization "update." The lemma is trivially true for the history prefix consisting of only this single "update," at the end of which `ptrs[a]` is NULL.

Suppose now that for some constant k and all $0 \leq i < k$, the lemma is true for all prefixes P_i ending with the ith successful update, u_i. We want to prove that the lemma is also true for P_k. First consider the case in which there is no crash between the previous successful update, u_{k-1}, and u_k. By the same reasoning used in the proof of Theorem 1, u_k will prepend a new record onto the chain at `ptrs[a]`, preserving valid content.

If there *is* at least one crash between u_{k-1} and u_k, there must clearly be at least two failed epochs between them. This means that the valid content as of the end of u_{k-1} will have persisted as of the beginning of u_k – its chain will be intact. We wish to show that no changes to the pointers and status indicator that occur between u_{k-1} and u_k – caused by any number of completed or partial failed updates – can prevent u_k from picking up and augmenting u_{k-1}'s valid content. We do so by reasoning on the transitions enumerated in Figure 6.

Let E_{k-1} denote the epoch of u_{k-1} and E_k the epoch of u_k. We note that all failed updates between u_{k-1} and u_k occur in epochs numbered greater than E_{k-1}. Further, let v denote the value of `a` (0, 1, or 2) immediately after u_{k-1}. Any update that sees the state generated by u_{k-1} will use row 2, 3, or 5 of Figure 6, and will choose, as its "new `a`" a value *other* than v. Over the course of subsequent failed updates before u_k, `ptrs[v]`'s role may transition at most twice, from `a` to `f` to `c`. As a consequence, the code of Figure 5 will never change the value of `ptrs[v]` – that pointer will continue to reference u_{k-1}'s valid content until the beginning of u_k.

Reasoning more specifically about the ID roles, a status indicator change persisted by a failed update that happens in epoch $E_{k-1}+1$ will, by necessity, make `ptrs[v]` the in-flight pointer. A subsequent update that sees this change in epoch $E_{k-1}+2$ or later will by necessity make `ptrs[v]` the committed pointer. Alternatively, a failed update in epoch $E_{k-1}+2$ or later, without having seen a previous failed update in epoch $E_{k-1}+1$, will also make `ptrs[v]` the committed pointer. A subsequent update that sees this change will leave `ptrs[v]`'s role alone. The net result of all these possibilities is that u_k will chose `ptrs[v]` as the `valid_head` regardless of which failed update's status indicator is read. It will then copy this value to the `next` field of its new node and point `ptrs[a]` at that node, preserving valid content. ◀

▶ **Theorem 4.** *Dalí is a buffered durably linearizable implementation of an unordered map.*

Proof. Straightforward: Given history H, containing crashes, we choose as our cut in each era the end of the last successful epoch. In the era that follows a crash, the visible content of each bucket (the records that will be seen by an initial `read` or `update`) will be precisely the valid content of that bucket. ◀

5 Experiments

We have implemented a prototype version of Dalí in C/C++ with POSIX threads. As described in Section 3.4, we implemented the global fence by exposing the privileged WBINVD instruction to user code using a syscall into a custom kernel module. Since non-volatile memory is not yet widely available, we simulated NVM by memory mapping a `tmpfs` file into Dalí's address space. This interface is consistent with industry projections for NVM [25].

As a representative workload for a hash map, we chose the transactional version of the Yahoo! Cloud Serving Benchmark (YCSB) [9, 11]. Each thread in this benchmark performs transactions repeatedly, for a given period of time. Keys are 8 bytes in length, and are drawn randomly from a uniform distribution of 100 million values. Values are 1000 bytes in length. We initialize the map with all keys in the key range.

The tested version of **Dalí** uses both mentioned optimizations (in-place updates and multiversioning) and our prototype SS2PL transaction processing system. Garbage collection is enabled. Epoch duration is a configurable parameter in Dalí; our experiments use a duration of 100 ms. We compared Dalí with three alternative maps: Silo [26], FOEDUS [18], and an incrementally persistent hash map (IP).

Silo [26] is an open source in-memory database for large multi-core machines.[4] It is a log-based design that maintains both an in-memory and a disk-resident copy. A decentralized log, maintained by designated logging threads, is used to commit transactions. We configured Silo to use NVM for persistent storage – i.e., Silo writes logs to main memory instead of disk.

FOEDUS [18] is an online transaction processing (OLTP) engine, available as open source.[5] The engine is explicitly designed for heterogeneous machines with both DRAM and NVM. Like Silo, FOEDUS is a log-based system with both an transient and persistent copy of the data. Unlike Silo, FOEDUS adopts a dual paging strategy in which a logical page may exist in two physical forms: a mutable volatile page in DRAM and an immutable snapshot page in NVM. FOEDUS commits transactions with the aid of a decentralized logging scheme similar to Silo. FOEDUS offers both key-ordered and unordered storage, based respectively on a B-tree variant and a hash map; our experiments use the latter. Like Dalí, both Silo and FOEDUS may lose recent transactions on a crash (their decentralized logs are reaped into persistence in the background).

We also implemented a data store called **IP**, an incrementally persistent hash map [24], as described in Section 2. As in Dalí, transactions in IP are implemented using SS2PL. To

[4] `https://github.com/stephentu/silo`
[5] `https://github.com/HewlettPackard/foedus`

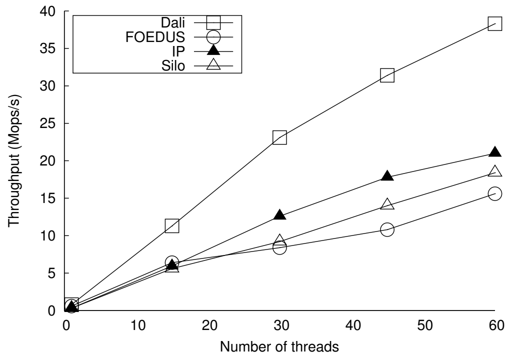


Figure 8 Scalability (75% reads).

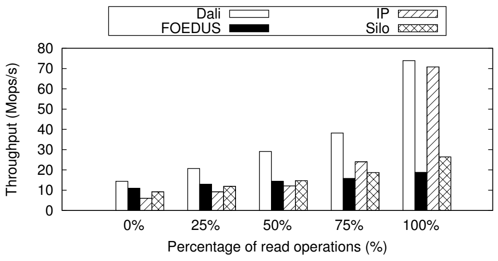


Figure 9 Impact of read:write ratio.

ensure correct recovery, per-thread undo logging is employed. In contrast to Dalí, Silo, and FOEDUS, transactions are immediately committed to persistence.

We benchmarked all four systems on a server-class machine with four Intel Xeon E7-4890 v2 processors, each with 15 cores, running Red Hat Enterprise Linux Server version 7.0. The machine has 3 TB of DRAM main memory. Each processor has a 37.5 MB shared L3 cache, and per-core private L2 and L1 caches of 256 KB and 32 KB, respectively.

Figure 8 shows the transaction throughput of Dalí and the comparison systems while varying the number of worker threads from 1 to 60; transactions here comprise three reads and one write. Dalí achieves a throughput improvement of 2–3× over Silo and FOEDUS across the range of threads. The removal of write-ordering overhead in Dalí reduces the time spent blocking per transaction, thereby improving throughput.

Figure 9 shows experiments that vary the read-to-write ratio at 60 threads across transactions containing four operations. Dalí's performance advantages compared to Silo and FOEDUS are larger for workloads with more reads due to the multiversioning optimization, whereas IP's advantage lies in the reduction in persist instructions at high read percentages.

6 Related Work

Dalí builds upon years of research on in-memory and NVM-centric designs, and upon decades of research on traditional database and multiversioning algorithms. As the promise of NVM is fast and fine-grained durable storage, tailored NVM systems have focused on specific types of applications: namely transactional memory and data storage.

Transactional memory systems are a natural fit for NVM, since a common challenge is to ensure consistent persistent state. The transaction-based NV-Heaps [7] and REWIND [5] and the lock-based Atlas [4] use undo logs to track writes to persistent state as they occur; on system crash, changes are rolled back. In contrast, the redo-logging Mnemosyne [27] redirects writes of persistent state to a thread-private location; on transaction commit, it copies changes to the shared state. All these systems are fine-grained "incrementally persistent" designs. A more novel design is SoftWrAP, which uses aliasing to keep both a transient and a persistent copy of data, thus avoiding inconsistencies caused by leaking cache lines [13].

Other authors have built intricate NVM data structures for data storage and transaction processing. Several projects use custom NVM-adapted trees that support atomic and durable updates [5, 6, 23, 28]. Schwalb et al. present a lock-free NVM hash map [24] similar to the incrementally persistent design of Section 2. These data structures all use incremental persistence, either within individual updates or in transaction logging.

Recent research on in-memory databases has also investigated NVM-based durability. Both DeBrabant et al. [10] and Arulraj et al. [1] explore how traditional database designs can be adapted for architectures with NVM, while Kimura's FOEDUS [18] builds a custom DBMS for NVM from the ground up.

Like Dalí, traditional disk-resident databases maintain a single persistent copy of the data (traditionally on disk, but for Dalí in NVM) and must move data into transient storage (traditionally DRAM, but for Dalí CPU caches) in order to modify it. Viewed in this light, CPU caches in Dalí resemble a database's STEALING, FORCEABLE buffer cache [15]. The updating algorithm of the incrementally persistent hash map is similar to traditional shadow paging [14, 29], but at a finer granularity. To the best of our knowledge, no prior art in this space has allowed writes to be reordered within an update or transaction, as Dalí does.

The prepend-only buckets of Dalí resemble several structures designed for RCU [20]. Dalí also resembles work on *persistent* data structures, where "persistent" here refers to the data structure's ability to preserve its own history [12]. Data structures of this sort are widely used in functional programming languages, where their ability to share space among multiple versions provides an efficient alternative to mutating a single version [22]. In the notation of this field, Dalí resembles a *partially persistent* data structure – one in which earlier versions can be read but only the most recent state can serve as the basis for new versions [12].

7 Conclusion

We have introduced *periodic persistence* as an alternative to the incremental persistence employed by most previous data structures designed for nonvolatile memory. Dalí, our periodically persistent hash map, executes neither explicit writes-back nor persistence fences within updates; instead, it tracks the recent history of the map and relies on a *periodic global fence* to force recent changes into persistence. Experiments with a prototype implementation suggest that Dalí can provide nearly twice the throughput of file-based or incrementally persistent alternatives. We speculate other data structures could be adapted to periodic persistence, and that the paradigm might be adaptable to traditional disk based architectures.

Acknowledgments. The authors sincerely thank Hideaki Kimura and Tianzheng Wang for their helpful suggestions and assistance.

References

1 Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage: Recovery methods for non-volatile memory database systems. In *SIGMOD*, Melbourne, Australia, 2015.

2 Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA*, Amsterdam, Netherlands, 2016.

3 João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, December 2006.

4 Dhruva Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Leveraging locks for NVM consistency. In *OOPSLA*, Portland, OR, USA, 2014. doi:10.1145/2660193.2660224.

5 Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5), January 2015. doi:10.14778/2735479.2735483.

6 Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7), 2015.

7 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Making persistent objects fast and safe with NVM. In *ASPLOS*, Newport Beach, CA, USA, 2011. `doi:10.1145/1950365.1950380`.

8 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, Big Sky, MT, USA, 2009.

9 Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, Indianapolis, IN, USA, 2010.

10 Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya R. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. *Proc. VLDB Endow.*, 7(14), 2014.

11 Anamika Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW*, Chicago, IL, USA, 2014.

12 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *STOC*, Berkeley, CA, USA, 1986.

13 Eric R. Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *MSST*, Santa Clara, CA, USA, 2015.

14 Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. *ACM Computing Survey*, 13(2):223–242, June 1981.

15 Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Survey*, 15(4):287–317, December 1983.

16 Intel Corp. Intel architecture instruction set extensions programming reference. Technical Report 319433-022, Intel Corp., October 2014.

17 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, Paris, France, 2016.

18 Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, Melbourne, Australia, 2015.

19 Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, San Jose, CA, USA, 2005.

20 Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Ottawa Linux Symposium*, Ottowa, Canada, 2002.

21 Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, Xi'an, China, 2017.

22 Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

23 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, San Francisco, CA, USA, 2016.

24 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *IMDM*, Kohala Coast, HI, USA, 2015.

25 Storage Networking Industry Association (SNIA) Non-Volatile Memory Programming Model. `http://www.snia.org/tech_activities/standards/curr_standards/npm`.

26 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, Farmington, PA, USA, 2013.

27 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Lightweight persistent memory. In *ASPLOS*, Newport Beach, CA, USA, 2011. `doi:10.1145/1950365.1950379`.

28 Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, Santa Clara, CA, USA, 2015.

29 Tatu Ylönen. Concurrent shadow paging: A new direction for database research. Technical Report 1992/TKO-B86, Helsinki University of Technology, Helsinki, Finland, 1992.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of DISC 2017 (LIPIcs, volume 91, `http://www.dagstuhl.de/dagpub/978-3-95977-053-8`, published in October, 2017), in which a missing funding acknowledgment has been included.

Dagstuhl Publishing – July 19, 2018.