Atlas: Leveraging Locks for Non-volatile Memory Consistency

Dhruva R. Chakrabarti, Hans-J. Boehm, Kumud Bhandari

HP Laboratories
HPL-2013-78

.

Abstract:

Non-volatile memory, such as memristors and PCRAM, can revolutionize the way programs persist data. In-memory objects can be persistent as the program executes, thus removing the need for a separate data storage format. However, the challenge is to ensure that such data remains consistent if a failure occurs during execution. In this paper, we add durability semantics to lock-based code, typically allowing us to automatically maintain a globally consistent state even in the presence of failures. We identify failureatomic sections of code based on existing critical sections and describe a log-based implementation that can be used to recover a consistent state after a failure. We discuss several subtle semantic issues and implementation tradeoffs. We confirm the ability to rapidly flush caches as a core implementation bottleneck and suggest partial solutions. Experimental results confirm the practicality of our approach and provide insight into the overheads of such a system.

Internal Posting Date: January 21, 2014 [Fulltext] - HP Restricted

© Copyright 2014 Hewlett-Packard Development Company, L.P.

Atlas: Leveraging Locks for Non-volatile Memory Consistency

Dhruva R. Chakrabarti
HP Labs
dhruva.chakrabarti@hp.com

Hans-J. Boehm
HP Labs
hans.boehm@hp.com

Kumud Bhandari
HP Labs & Rice University
kumud.bhandari@hp.com

Abstract

Non-volatile memory, such as memristors and PCRAM, can revolutionize the way programs persist data. In-memory objects can be persistent as the program executes, thus removing the need for a separate data storage format. However, the challenge is to ensure that such data remains consistent if a failure occurs during execution.

In this paper, we present Atlas, a system with durability semantics to lock-based code, typically allowing us to automatically maintain a globally consistent state even in the presence of failures. We identify failure-atomic sections of code based on existing critical sections and describe a log-based implementation that can be used to recover a consistent state after a failure. We discuss several subtle semantic issues and implementation tradeoffs. We confirm the ability to rapidly flush caches as a core implementation bottleneck and suggest partial solutions. Experimental results confirm the practicality of our approach and provide insight into the overheads of such a system.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages, Performance, Reliability

Keywords non-volatile memory, transactions, locks

1. Introduction

Applications that need to take advantage of the parallelism available on modern multicore computers are most commonly written using threads and locks. This programming model, though low-level and often error-prone, is well-established, and quite general.

Transactional memory (TM) [11, 20] attempts to raise the abstraction level by borrowing the idea of transactions from databases and incorporating them into parallel programs. A program transaction is a block of code that appears to execute indivisibly. A programmer is only required to specify the code that should be part of the block thus transferring the onus of synchronizing shared memory references to the implementation. This simplifies the construction of modular parallel programs.

New non-volatile memory (NVRAM) technologies such as memristors [19] and phase change memory (PCM) [12] provide an interesting twist to programming since they allow CPU stores to persist data directly at DRAM-like speed. Existing programs, incognizant of NVRAM, will continue to work but will not be able to take any advantage of the newly available persistence properties. Our goal is to require minimum changes in the programming model that enable applications utilize the persistence properties of NVRAM at the byte granularity.

Data in NVRAM lives beyond the lifetime of the creating process, and across system restarts. The programmer is able to persist data reliably through CPU store instructions and retrieve them using CPU load instructions. This model removes the frequent need to maintain both an in-memory object format and a separate persistent file format, together with the substantial amounts of code needed to keep them consistent. Data structures persist in NVRAM as they are created and modified and the evolved state can be reused when an application is restarted. But since hardware and software failures cannot be ignored, this model requires care to ensure that the persisted program is consistent, and hence really reusable. This is often no easy task, especially for multithreaded programs.

```
1: int *t = malloc(...);          // allocate
2: *t = 10;                       // initialize
3: l.lock(); p = t; l.unlock();   // publish
```

Figure 1. Typical allocation sequence

Consider Figure 1 where a persistent memory location is allocated, initialized, and then "published", by assigning it to a shared variable p. If the store to persistent pointer p in line 3 becomes visible in NVRAM before the initial store to the persistent location in line 2 (e.g. because the store to *t is still cached and not yet in NVRAM)[1] and the program crashes, a post-restart dereference of the persistent pointer will read uninitialized data. In some settings, e.g. if p were a complicated C++ "smart pointer" it is also conceivable that only part of the pointer p had made it to NVRAM, causing a slightly different kind of failure on restart.

The problem in Figure 1 can be addressed by ensuring the updates to NVRAM, such as the assignment to p, are atomic, and ordered correctly with respect to other updates. In this paper we argue that for multithreaded lock-based programs such as in Figure 1, the locking operations usually give us enough information to infer NVRAM atomicity and ordering requirements, and that it is useful to extend locking primitives with durability semantics. We build on the idea of durability semantics [5] that associate failure-atomicity with code sections in lock-based programs.

This paper presents Atlas, a system with appropriate semantics to automatically provide lock-based programs with useful durability semantics in the context of NVRAM. A log-based implementation is described along with primary sources of overheads and possible optimizations. Experimental results evaluate our programming model with respect to existing transient and durable programs.

It is useful to compare this approach with recent work on transactional memory (TM). As the name suggests, TM differs from database transactions in that it stores data in (volatile) memory as opposed to a durable medium such as a hard disk. Consequently, TM provides Atomicity, Consistency, and Isolation (ACI)[2] but no durability. However, recent work explored adding durability to TM in the context of NVRAM and Flash [6, 17, 22]. Durable TM is clearly a very attractive model for NVRAM. But TM, even in its original form,

[1] Note that hardware or compiler optimizations may have a similar effect.

[2] And hopefully some ordering guarantees with respect to non-transactional memory accesses.[8, 14]

has at this point not been widely adopted. Aside from performance issues, there is a non-trivial effort required to convert lock-based programs to TM programs [3]. Additionally, some constructs, such as condition wait, arguably do not lend themselves well to the TM paradigm.[10] Consequently, lock-based multithreaded programs will continue to be popular even when TM is adopted. As a confirmation of this trend, the draft specification of TM constructs for C++ [20] requires the co-existence of locks with transactions. So, to take advantage of NVRAM-based data reuse in general multithreaded programs, durability semantics should also be added to lock-based programs.[3]

By generalizing durability semantics beyond transactions to lock-based programs, we expose a number of interesting semantic issues that have not been well-explored by prior work that limited itself to durable transactions.

2. System Assumptions and Programming Environment

We make the following assumptions for the underlying system.

- NVRAM devices are expected to be connected as memory and accessed using regular CPU loads and stores.
- The read and write latencies of NVRAM devices are assumed to be roughly similar to DRAM.
- Machines will continue to have volatile CPU caches. The current multi-level cache organization has huge performance benefits, and we do not expect drastic changes in the near future.
- Given a memory address, we assume the availability of a low-level interface for flushing the corresponding cache line into NVRAM. (See Section 5.2 for details.)
- We assume a fail-stop or crash-recovery model. If a program encounters a tolerated failure, such as a power failure, data already in NVRAM survives, those in DRAM or caches do not. If a CPU fails, we assume that the data in NVRAM is still reachable from another CPU, either because the data is accessible from a second CPU, or because the CPU has been replaced.
- Memory objects are specified by the programmer to be either persistent or transient. Persistent data lives in NVRAM, is reachable beyond the lifetime of the process that creates it, and survives a (tolerated) failure. Transient data, such as thread stacks, software caches, or possibly security keys, are not accessible once the creating process terminates. Transient data may be stored in DRAM or NVRAM. In the latter case it is erased on restart.
- We assume a data-race-free programming model, as in C11 or C++11.
- It would be impractical to require the programmer to ensure that no persistent memory is leaked in the event of a failure. Newly allocated memory is typically referenced only by (transient) expression temporaries, and avoiding this would require a dramatic change from, and complication of, existing programming practice. We assume that, at a minimum, persistent regions are garbage collected (possibly conservatively, but tracing only from persistent region roots) in the event of a failure.

Given the above assumptions, we support the following abstractions.

2.1 Programming with Persistent Regions

Programmers use containers called *persistent regions* (PR) [1, 6, 22] for identifying persistent data. A PR consists of an identifier and a list of contiguous virtual address ranges. Every PR has at least one entry point called a root. A root stores the start address of a set of connected persistent objects. A root of a PR lives within that PR and is hence persistent; it provides a way to traverse the set of connected objects reachable from it[4]. In a quiescent state (such as the one reached at successful program termination or immediately after a failure), any data within a PR that is not reachable from any of its roots is assumed to be garbage and reclaimed. (Such reclamation is relatively easy to implement, since the root set is clearly defined and limited.) Data not in a PR is considered logically transient.

```
pr = find_or_create_persistent_region(nm);
persistent_data = get_root_pointer(pr);
if (persistent_data) {
  A: // restart code
} else {
  B: // initialize persistent_data
}
C: // use persistent_data
```

We say that "restart code accesses a particular datum", if it is accessed by the code run after a failure, i.e either by A or by C when it is re-executed.

Figure 2. Structure of an NVRAM Program

2.2 Failures and Recovery

Program state in persistent regions survives a tolerated failure,[5] other program state does not.

If a failure occurs, a recovery phase will be initiated to ensure that data structures are in a state of harmony.

A particular store to persistent memory becomes *durable* at a point in program execution. If a failure occurs before that point, it will not be visible when the program is restarted, either because it had not yet reached NVRAM, or because the recovery phase could not reconstruct a harmonious state that included the update. If a failure occurs after this point, it will be visible after the program is restarted.

2.3 Application Structure

Applications will normally be structured as in Figure 2. Data within a PR is allocated by using a special *malloc*-like interface that maps NVRAM physical pages directly onto the address space of a process. The programmer adds restart code that runs when an application starts and detects whether a prior data structure version is already available in NVRAM. Proper use of consistency mechanisms ensures that if a data structure exists, then it is consistent; this property enables restart from an evolved state, leading to saved computation, and potentially avoiding loss of data, such as user input.

2.4 Pointers from and to persistent regions

For purposes of this paper, we presume that pointers between transient and persistent objects are allowed, and that mechanisms exist to

- Ensure that a given persistent region is always allocated at the same address (e.g. by providing a suitable address reservation scheme) or can be safely relocated (e.g. by using relative pointers).
- "De-fang" dangling pointers from a persistent region to transient objects after a failure, e.g. by clearing such pointers when the region is garbage collected.

[3] For our purposes, transactions can be modeled as a single global reentrant lock, so our lock-based solutions are strictly more general than TM.

[4] A program traversing a PR needs to know the layout of objects within it, e.g. by including a header file that describes the objects.

[5] We assume that the hardware defines the notion of "tolerated failure", and that it includes at least power failures.

- Garbage collect in the presence of cross-persistent-region pointers, possibly by disallowing them, or possibly with a more complex garbage collection scheme extending across regions.

3. Durability semantics for lock-based code

Our goal is to preserve all existing properties of lock-based code and incorporate additional durability semantics. Updates to shared locations are exposed to threads exactly as before. Isolation is provided by holding locks and it remains the responsibility of the programmer to follow proper synchronization disciplines. Consistency tends to be a function of the application and we do not change that. The only semantics we add is failure-atomicity or durability, only for memory locations that are persistent, i.e. those within a PR. There is no change to the threads memory model and memory visibility rules with respect to other threads.

3.1 Treating unlocked program points as consistent

Program data structures must satisfy invariants that hold in consistent states. Programs that are written using atomic sections (or transactions) [11], as opposed to locks, typically mutate these data structures and temporarily violate invariants only within those sections — so the invariants hold at the start and end of every (outermost) atomic section as well.

Existing lock-based applications generally satisfy a similar property. In order to avoid data races and ensure thread-safety, they acquire locks to form critical sections (CS) in which data structures are modified and thus temporarily inconsistent.

We thus assume that data structures are inconsistent only in critical sections, and hence treat lock operations as indicators of consistent program points. We will call program points at which the executing thread holds no locks *thread-consistent*. If no locks are held by any thread, all data structures should be in a consistent state.

This assumption fails, for example, if the client implements its own mutexes in terms of system-provided mutexes (or transactions), or if the application is single-threaded. In this case, the programmer may explicitly introduce additional critical sections, using a lock local to the thread, to explicitly indicate code regions in which data structures are inconsistent, and which should be treated as failure-atomic. Since the lock is local to the thread, no actual synchronization is introduced. We actually provide less obscure syntax to handle this case but, unlike in the transactional case, such a facility is not fundamental.

The durability semantics of critical sections differ from the transactional case for two primary reasons: Locks do not necessarily nest perfectly, and the semantics of nested critical sections are very different from simply ignoring the inner critical section. We will see in the next sections that both are important, but neither inhibits our approach.

3.2 Identifying a Failure-Atomic SEction (FASE)

Consider a dynamic execution trace $A_1A_2...A_n$ of a single thread t, where A_i refers to an instruction executed. We define a section of this trace comprising $A_i...A_j$ to be a Failure Atomic SEction(FASE) if the following hold:

- The point just before A_i is thread-consistent, i.e. t holds no locks there.
- The point after A_j is again thread-consistent.
- No point between A_i and A_j is thread-consistent, i.e. a lock is held at every intervening point.

It follows that A_i acquires a lock and A_j releases a lock, though not necessarily the same lock. Locks inside a FASE do not have to nest perfectly. For example, hand-over-hand locking poses no problem. Atlas then ensures the following fundamental property:

```
tmp = pointer_to_new_nvm_data structure();
l1.lock();
ptr_to_persistent->x = tmp;
l1.unlock();
```

Figure 3. Initialization with atomic pointer update

PROPERTY 1. *If any update within a* FASE *is durable, i.e. visible in* NVRAM *after a failure, then all updates within that* FASE *must be.*

3.3 Happens-Before Relation between FASEs

Semantics of nested critical sections are different from those of nested atomic sections in a TMsystem. While locks provide isolation between threads at the granularity of individual critical sections, we are providing failure-atomicity at larger granularity (Section 3.2), an apparent mismatch. It follows that even if updates made by thread T_1 within an inner CS C within a FASE F are exposed to thread T_2, C is not durable until F is durable. Hence, any update of a persistent location made by T_2 and dependent on a change made by T_1 in C cannot be durable either until F is durable.

Instead of tracking write-read dependency at the memory access level, we use an ordering on entire FASEs induced by the happens-before relation [4, 13], to capture the durability constraints. We say that a FASE f_1 is durability-ordered before another FASE f_2 (denoted by $f_1 \leq_d f_2$) if there is a release operation in f_1 that dynamically happens before an acquire operation in f_2. The durability constraint among FASEs is captured by the following property.

PROPERTY 2. *If $f_1 \leq_d f_2$, f_2 is durable only if f_1 is.*

As the notation suggests, and unlike the situation for atomic sections with TM, the $\leq_d$ relation may not be antisymmetric. If FASE f_1 contains a release operation r_1 that synchronizes with an acquire operation a_2 in FASE f_2 and if FASE f_2 contains a release operation r_2 that synchronizes with an acquire operation a_1 in FASE f_1, $f_1 \leq_d f_2$ and $f_2 \leq_d f_1$. If a cycle exists among FASEs, i.e. if $f_1 \leq_d f_2 \ldots \leq_d f_1$, then all of the constituent FASEs must be durable if one of them is.

We observe that all synchronization operations must participate in creating the $\leq_d$ relation. Happens before relations between references to persistent locations are often imposed by synchronizing through critical sections that contain references to only transient data. Consider two FASEs f_1 followed by f_2 executed by thread $t_1$and another two FASEs f_3 followed by f_4 executed by thread t_2. Assume that f_1 and f_4 update disjoint persistent locations and are protected by distinct locks. Assume too that f_2 and f_3 contain accesses to only transient locations that conflict and hence use the same lock for synchronization. In order to make sure that we do not lose the $\leq_d$ relation from f_1 to f_4, we must track the happens-before relation from f_2 to f_3 even though they have no accesses to persistent locations.

3.4 Durability-safety for non-critical-section updates

Consistent with publication semantics in multithreaded code [14], we provide durability-safety for updates made without holding a lock as per the following properties.

PROPERTY 3. *If a* FASE *f_1 is durable, i.e. remains visible after a failure, then all updates that happened before f_1(and can be observed by restart code) must also be durable.*

Consider the code in Figure 3, which is similar in spirit to Figure 1: Since the assignment to the x field occurs inside a critical

```
// Initially ptr_to_persistent->valid is true.
l1.lock();
ptr_to_persistent->valid = false;
l1.unlock();
ptr_to_persistent->data = inconsistent_junk;
A: // Consider failure here
...
ptr_to_persistent->data = good_stuff;
l1.lock();
ptr_to_persistent->valid = true;
l1.unlock();
```

Figure 4. Initialization with atomic pointer update

section, we ensure (Property 1) that it is made durable atomically. Property 3 ensures that if the assignment to the `x` field is visible after a failure, then so are the assignments to the new data structure performed during its creation by `pointer_to_new_nvm_data structure()`; thus `x` or the data structure it points to cannot acquire dangling pointers as the result of a failure.

We expect that applications meet the following *restart-race-freedom* requirement:

REQUIREMENT 1. *No update, outside of a* `FASE`*, of a persistent location u can be seen by (i.e. the stored value read by) restart code after a failure, unless the failure occurs after a* `FASE` *following u.*

Figure 3 satisfies Requirement 1. The `NVRAM` updates performed during the data structure construction in `pointer_to_new_nvm_data structure()` are all to memory locations that will be unreachable garbage if a failure occurs before the `FASE` containing the assignment to `x`. Thus the constructed data structure can be observed by restart code only if the subsequent `FASE` completes.

Code violating this requirement is in some sense "unreasonable". Updates not followed by a `FASE` at the time of the failure could be partially completed and thus inherently inconsistent. If they were visible to restart code, we could ensure any form of consistency only by essentially requiring atomicity enforcement even outside `FASEs`.

Figure 3 also satisfies the following *strong restart-race-freedom* requirement:

REQUIREMENT 2. *All updates of persistent memory locations outside of a* `FASE` *must be to locations that can never be accessed by restart code unless the subsequent* `FASE` *completes.*

This strengthens Requirement 1 by insisting that the written location cannot be read at all, as opposed to requiring only the particular written value cannot be seen.

To see the difference, consider Figure 4, where the restart code is assumed to use the `data` field only if the `valid` field is `true`. This code satisfies Requirement 1 but not Requirement 2.

For code that satisfies Requirement 2, like Figure 3, Properties 1 and 3 suffice to imply the following, highly desirable property:

PROPERTY 4. *After a failure, the program state observed by restart code is as if each thread had stopped at a particular point in its execution at which no locks were held.*

If any updates from a `FASE` are visible to restart code then all of them are, and so are all prior persistent updates. Thus all updates up to and including those in the last completed `FASE` are visible to restart code. It does not matter whether the implementation preserves any later updates across the restart; by Requirement 2 the restart code never looks at those locations.

This does not hold for code such as Figure 4. If the application crashes at `A`, and the implementation, consistent with Properties 1 and 3, makes the update to `data` visible without exposing the initial update to `valid`, the restart code sees an inconsistent state in which `valid` is `true`, but `data` is `inconsistent_junk`.

In cases like this, where only Requirement 1 is enforced, the implementation needs to satisfy the following additional property:

PROPERTY 5. *If an update at a thread-consistent program point P is durable and can be observed by restart code, then all* `FASEs` *that happened before P must also be durable.*

Together with Requirement 1, Properties 1, 3, and 5 again imply Property 4. The first three properties ensure that we see all updates up to and including some `FASE` plus possibly some later updates preceding the next `FASE`. Requirement 1 guarantees that those later updates are not observed by the restart code.

Note that all observations in this subsection apply equally to systems based on transactions, so long as non-transactional updates to `NVRAM` are allowed. Prior work has generally disallowed them.

Although we believe that systems insisting on Requirement 2 may be of interest, here we continue to pursue a model that is as close to a conventional lock-based programming model as possible. For the rest of this paper, we assume that the programmer must only obey Requirement 1, and the implementation must thus ensure Properties 1, 3, and 5 to guarantee Property 4. As we point out in Section 5, this assumption comes at some cost.

3.5 What about I/O operations?

Our durability model imposes similar and additional constraints on I/O operations compared to a transactional setting [21]. Since code outside a `FASE` may have to be undone, it should be possible to buffer not only I/O within `FASEs` but also those outside `FASEs`[6]. Currently, we advocate the following solution:

- Outputs: Regardless of where they are issued, defer them by recording the operation in `NVRAM`. An output operation `x` is considered deferrable if there is no requirement to execute `x` before another non-output operation `y` that dynamically happens after it in that program. For example, an input operation that requires the execution of an output operation before it is not handled. When the global consistent state reaches the point at which an output was originally issued, perform actual externalization. In the presence of multiple deferred outputs, they must be externalized in happens-before order.
- Inputs: Invoke an input operation as it is encountered. If the input operation needs to be undone during recovery, undoing an input operation is a `NOP`. The result is indistinguishable from normal operation that never reached the input program point.

4. Implementation

We now describe our implementation of the durability semantics so that a consistent program snapshot can be realized efficiently.

4.1 Initialization and shutdown

The initialization phase must be called at program start-up. This phase performs two main tasks:

1. Creation of a process-private `PR` named `LPR`, where the log entries will be written. At any given point in time, the log has a single entry point `LEP`, which is initially `NULL`. The root of `LPR` points to `LEP`; once set, the root remains constant throughout program execution.
2. A helper thread is created to perform consistent state identification and log pruning.

[6] If it can be proved that certain code outside a `FASE` will never be undone, the latter requirement can be relaxed. See also Section 5.1.

```
// Atomically updateable entry point of the log
atomic<LHead*> LEP;
// Lock-to-log mapper
atomic<ReleaseInfo*> ReleaseTab[TAB_SIZE];
enum LE_TYPE {ACQ, REL, STORE, OTHER};
// Log Entry                 // Thread-specific header
Type LE: 32 bytes            Type LHead:
  void *Addr;                  LE *LePtr;
  uintptr_t ValOrPtr;          LHead *Next;
  atomic<LE*> Next;          // Mapper Node
  size_t Size: 60;           Type ReleaseInfo:
  LE_TYPE Type: 4;             atomic<LE*> LeAddr;
                               ReleaseInfo *Next;
```

Figure 5. Log and Accessory Structures/Declarations

The application may optionally call a function to join the helper thread and ensure that everything is in a consistent state, thus eliminating the need for recovery processing in the case of a normal process shutdown.

4.2 Log Structure and its Creation

As an application is running, Atlas client threads log "undo" information for stores to persistent memory, as well as happens-before relations between synchronization operations. The result is a network of log entries, in NVRAM that by itself is sufficient to allow identification of FASEs and computation of a consistent state.

Left to itself, this log grows without bound. It is the job of the helper thread created above to identify consistent states and prune log entries predating this state, since such entries will not be used. Each pruning step performed by the helper thread is an atomic operation, essentially transitioning the durable program state from one consistent snapshot to another.

In the event of a failure, the recovery process applies remaining unpruned log entries, reverting the process state to the last consistent state identified by the helper thread.

As indicated in Figure 5, the log is a shared data structure with a single atomically updateable entry point, LEP. LEP points to a linked list of LHead nodes, each of which contains the first non-deleted log entry (LE) created for a given thread. Every persistent-store[7] and synchronization operation[8] are potential logging candidates. There is no need to log non-synchronization read operations.

4.2.1 Structure of a Log Entry

As shown in Figure 5, a log entry (LE) is a fixed size object and its fields are shown in Figure 5. LE_TYPE denotes the type of a log entry. For a store, the field Addr is the address of the location written into; for acquire and release synchronization operations, it is the address of the lock; otherwise it is unused (or NULL). For a store, ValOrPtr maintains the value of Addr just before the store instruction, allowing this log entry to be used in undoing the effect of the store during a post-crash recovery step. For an acquire synchronization operation, ValOrPtr points to the log entry, if any, for the last release of the same lock, essentially capturing a happens-after (inverse of happens-before) relationship. For a release synchronization operation, ValOrPtr is unused (or NULL). The Next field of LE points to the next log entry of the same thread in program order. When an LE l_1 is created, its Next field is set to NULL and the Next field of the previous log entry for the thread, if any, is set to point to l_1. The Next field is atomic, allowing multiple threads to concurrently write and read the Next field. Note that all fields of LE except Next are write-once.

[7] Our implementation also directly handles library interfaces to modify a persistent location, such as memcpy, explaining the 60 bit size field in the log entry.

[8] We discuss mutual exclusion locks here. Our implementation also handles reader-write locks, etc.

```
var=val              l.lock()       l.unlock()
-------              --------       ----------
log_str(&var,sz);    l.lock();      log_rel(&l);
var=val;             log_acq(&l);   l.unlock();
vbarrier(&var);
```

Figure 6. Result of Code Interposition

4.2.2 Memory Allocation for the Log

Atlas supports a number of modes of NVRAM memory allocation. The base interface (nvram_alloc) is modeled on libc malloc and has similar performance characteristics. The interface nvram_alloc_calign is similar to nvram_alloc, except that it ensures that the returned address is at the start of a cache line[9].

Since log entries have specific access patterns and high performance requirements, we support a mode (nvmram_alloc_cb) in which CBs are allocated once by a thread, and then repeatedly reused as a circular buffer for log entries created by that thread. Only one user thread writes new log entries to a given CB while the helper thread is the only entity removing from it. Addition of log entries requires a lock only when the CB needs to be grown. Other operations are lock-free. The addition of a log entry does not require manipulation of any NVRAM-resident allocation metadata, thus minimizing cache-line flushes.

4.2.3 Lock-to-log Mapper

A chained hash table (referred to as ReleaseTab in Figure 5) shared among all threads maintains a map from a lock address to the log entry (LeAddr) corresponding to the last release operation of that lock.

4.2.4 Interposing on Store, Acquire, and Release Operations

The major components of Atlas are implemented in a library. We use an LLVM-based compiler pass to interpose on persistent updates and synchronization operations of the original program to insert calls into the Atlas library. Figure 6 shows the resulting code for each type of instruction interposed, where log_str, log_acq, log_rel, and vbarrier are routines implemented in the Atlas library, and described further below. Note that both log_acq and log_rel will be executed under the protection of the lock.

4.2.5 Library logging routines

We provide a brief overview of these logging operations here. More detail is provided in Appendices A and B.

The log_str routine explicitly checks that both the target location is indeed persistent (to cover cases in which the compiler could not determine this statically) and that logging is really needed (see Section 5.1). Assuming that both these conditions are met, a log entry is created and published by attaching it to the shared log. In addition to similar creation and publication phases, log_acq and log_rel help track the happens-before relationships between synchronization operations.

The log_rel routine creates a log entry and updates the ReleaseTab entry for the lock to point to it, either by allocating a new entry or redirecting the existing one. The log_acq routine

[9] Given an address, the start of a cache line can be obtained by applying a mask on it based on the cache line size.

creates a log entry that refers to preceding release of the same lock, obtained from ReleaseTab.

Publishing a Log Entry: In any of these cases, once a log entry of the appropriate type is created, it has to be attached to the shared log, so that it can be seen by the helper thread. This is accomplished with an atomic update of either the header (first log insertion) or of the last Next field in the linked list comprising the log for the current thread. A pointer to this last Next field for each thread can be maintained in a transient memory location.

All operations are lock-free except in rare cases when allocation is required, e.g. when a CB needs to be grown.

4.3 Introducing Visibility Barriers

In Section 1, we mentioned the need for an ordering constraint in the context of Figure 1. Atlas internally needs to enforce similar constraints. In particular:

1. We ensure that log entries become visible in order.
2. We ensure that log entries become visible before the actual stores to the corresponding locations, so that all stores not part of the latest consistent state can be properly undone.
3. We ensure that properties 3 and 5 hold.

To ensure the first property, we rely on the sizes and cache-line alignment of log entries, presuming that two stores to the same cache line are evicted from the cache either simultaneously or, if there is an intermediate eviction, in the correct order. See Appendix B for details.

4.4 Consistent, Durable State Computation

While the log is concurrently written out by user threads, consistent states are computed serially in a helper thread. A consistent program state consists of the effects of a set of FASEs, such that Property 2 is satisfied by each of them. The helper thread identifies the log entries corresponding to a set of FASEs and removes them from the log. If a failure occurs after this removal step, the recovery phase will apply all the undo log entries found in NVRAM, returning the program state to the consistent state computed by the helper thread.

The helper thread may fall behind the user threads implying that, in the event of a failure, the last computed consistent state can be further advanced. We ensure that this happens by having the recovery phase (Section 4.5) advance the consistent state, if possible. Computing the consistent state by a distinct non-blocking helper thread helps minimize the perturbation on user threads. The availability of a distinct processor core for the helper thread should also help in this regard. We outline the steps performed by the helper thread.

4.4.1 Creating a durability-ordered graph G on FASEs

The helper thread reads the LEP atomically, retrieving the thread-specific log headers. It traverses log entries for a given thread, until there are no more logs to traverse or it reaches a configurable limit. During this traversal, it uses a simple count of held locks (heldlocks) to determine the start and end of a FASE. An increment of heldlocks from 0 to 1 indicates the start of a FASE. A decrement from 1 to 0 indicates the end of the current FASE.

Every time a new FASE is encountered, a graph node is created and appropriate durability-ordered edges are added from other graph nodes to this new node according to semantics outlined in Section 3. Every graph node is assigned a sequence of log entries corresponding to all actions since the end of the preceding FASE, if any, and including all actions in this FASE.

4.4.2 Pruning the durability-ordered graph G

Following Property 1 in Section 3, if the FASE corresponding to a node $\mathtt{n}_1$, has not yet ended, then $\mathtt{n}_1$ is marked non-durable. Following Property 2, any node that is reachable from a non-durable node (or with acquire operations corresponding to a not-yet-traversed release operation) is also marked non-durable. All non-durable nodes are then removed from G. The remaining nodes in G correspond to a consistent durable program state.

4.4.3 Atomic pruning of the log

Let $\mathtt{L}_1$ and $\mathtt{L}_2$ denote the states of the log corresponding to two consecutive consistent program states $\mathtt{S}_2$ and $\mathtt{S}_2$. Since failures can occur at any point of time, the transition from $\mathtt{S}_2$ to $\mathtt{S}_2$ must be atomic. Consequently, the log must change from $\mathtt{L}_1$ to $\mathtt{L}_2$ atomically as well. The helper thread achieves this atomic switch by versioning the log as follows:

- Create a new log pointer, $\mathtt{LEP}_{new}$, initially pointing to an empty linked list of thread-specific header nodes.
- For every existing thread-specific header node h reachable from LEP, identify the first log entry $\mathtt{L}_{first}$ in program order that does not belong to the pruned G. Create a new thread-specific header node $\mathtt{h}'$ with its field LeAddr pointing to $\mathtt{L}_{first}$. Insert $\mathtt{h}'$ at the head of the linked list pointed to by $\mathtt{LEP}_{new}$.
- Atomically set LEP to the header node pointed to by $\mathtt{LEP}_{new}$ in a CAS-loop. The CAS may fail if a new thread got spawned during versioning and its logs were not reflected in the new version. In such a case, the subsequent CAS-iteration must incorporate the log entries for the new thread before attempting the atomic pointer switch.

LEMMA 1. *The consistent durable state satisfies properties 1-5 from Section 3.*

Proof. Property 1 holds since FASEs are pruned as unit. Property 2 is enforced since pruning respects durability order. Properties 3 and 5 follow since non-FASE updates are also pruned in happens-before order. Property 4 follows. •

4.5 Recovery Phase

The recovery phase first applies the algorithm from Section 4.4 to ensure that the consistent state is advanced as much as possible. The remaining log entries are then traversed, reverting any updates remaining in the log, and reestablishing the consistent state as established in Section 4.4. The log entries are traversed in an order consistent with the inverse of the happens-before relation recorded in the log, i.e. in reverse order within a thread, and such that critical sections corresponding to a given lock are undone in the reverse order of the original lock acquisition order. Since the original execution is presumed to have been data-race-free, updates to the same memory location must be ordered by happens-before, and thus any order consistent with the inverse happens-before relation will produce the same deterministic result, reflecting a state in which each thread has advanced just past the last FASE pruned by the algorithm in Section 4.4. This state may not be one that occurred at a particular point in time during the original execution, but it is one that could have occurred in a valid execution consistent with the observed happens-before ordering.

Applying this recovery phase is idempotent. If it is interrupted by a failure, it is attempted again in its entirety. Once successful, the persistent region LPR is removed from the system.

See Appendix C for details.

5. Optimizations

We briefly discuss two classes of optimizations: Those related to reducing the required amount of logging, particularly outside FASEs, and those related to enforcing visibility after a crash.

5.1 Log Elision outside FASEs

A cross-thread hb-relation may stem from a lock release operation that is dynamically within a FASE. The inner critical section may expose effects of updates to other threads by releasing a lock before the FASE is finished. If this FASE is undone for some reason, another FASE that happens after it will have to be undone as well. It follows from Property 5 that if the effects of a completed FASE f_1 are undone, the effects of code after f_1 and outside a FASE need to be undone as well which requires that such updates be logged in the first place.

However, further analysis can be used to determine whether logging can be elided for an update to a persistent location outside a FASE[10]. If a FASE f_1 in thread T1 dynamically happens before another another FASE f_2 in thread T2, then any write to a persistent location, executed by T2 outside a FASE and executed after f_2 in program order, will have to be logged unless it can be proved that f_1 cannot be undone. A FASE cannot be undone if it has completed and every FASE that happens before it has completed successfully. We implemented this optimization in Atlas and found that it drastically reduced the overhead of logging in many applications where many updates to persistent locations occur outside FASEs.

5.2 Reducing the cost of persistence

On systems with NVRAM, ensuring that data has been made persistent in NVRAM often becomes performance critical, and our system is no exception. Unfortunately, current architectures often provide (1) only unoptimized support for forcing data out of the CPU caches, and often (2) no support for forcing data out of volatile memory controller buffers into NVRAM. This is entirely understandable, since neither facility is important on today's hardware without NVRAM. User-level applications cannot tell whether data is stored in volatile caches, volatile buffers, or volatile memory. But this changes entirely with NVRAM.

In this paper, we ignore (2), since it must be addressed at the hardware level, and could be addressed in many different ways, possibly by arranging that controller buffers are themselves non-volatile, or are written back in the event of any tolerated failure. We instead focus on (1), flushing caches.

Again consider code along the lines of Figures 1 or 3, and assume that logging can be avoided outside of a FASE using the techniques of the preceding section. We still must ensure that all the stores to initialize a data structure become visible before the FASE that stores the pointer to it. It is generally infeasible to use "flush entire cache" operations provided by some processors, since the required worst-case latency is too high, and this interferes with kernel interrupt-response guarantees. Two other obvious approaches are:

1. (Eager) Immediately flush every store, as it's completed, out of the processor cache. This can be done with either an instruction to flush a single cache line (e.g. CLFLUSH from [9]) or by using a write-through memory mapping.
2. Another alternative is to maintain a data structure that contains recently written addresses, and then flush all affected cache lines at once, when needed. We support 2 such modes. The first one, called $\texttt{Delayed}_l$, tracks all dirty locations since the end of the last FASE, and flushes all of the corresponding cache lines at the end of the current FASE. In the second one, called $\texttt{Delayed}_g$, the user threads do not flush any user data, instead delegating this task to the helper thread. When the helper thread computes a globally consistent state, it computes the affected cache lines from the dirty addresses found in the corresponding log entries, and then flushes all of the lines synchronously. $\texttt{Delayed}_g$ has an implication on the recovery phase and has some interesting performance characteristics. Unlike the design outlined in Section 4.5, the recovery phase must not advance the consistent state if $\texttt{Delayed}_g$ mode was in use during execution since there is no guarantee that all of user data corresponding to the potential advanced state was flushed out of caches before the failure. Regarding performance, $\texttt{Delayed}_g$ may suffer (as it does today on x86-64) if the cache line flushes are synchronous since in this scheme, a lot of data usually gets flushed out at a consistent point. On the other hand, since this scheme flushes cache lines only when consistent states are found, the total amount of flushed data is usually lower than $\texttt{Delayed}_l$.

The first option has the serious problem that every store results in a cache line flush, a very expensive operation. The second allows flushes to be merged, but at the expense of possibly performing a large number of cache line flushes at the last possible point, when there is no hope of overlapping them with useful computation.

We expect that a compromise, called Hybrid, outperforms both of these extremes. We remember a small set R of cache lines that have been written, but not flushed. When a new store is executed, if the address is already part of a cache line in R, we do nothing else. Otherwise we add it to R, if necessary after removing and flushing a line in R to make room. This admits an extremely efficient software implementation if the set R is represented as a hash table, where collisions are handled by evicting the previous entry, as in a direct-mapped CPU cache.

This compromise scheme both avoids flushing the same cache line repeatedly, if it is written in rapid succession, and it avoids synchronously flushing large numbers of cache lines at once. Unfortunately, current x86 hardware appears to not support much concurrency between cache line flushes and other operations, but we nonetheless see some benefit from this scheme. We show results with these different flushing techniques in Section 6. Variants of this scheme that instead use x86 "MOVNT" operations also appear promising, but raise other issues, and are left to future work.

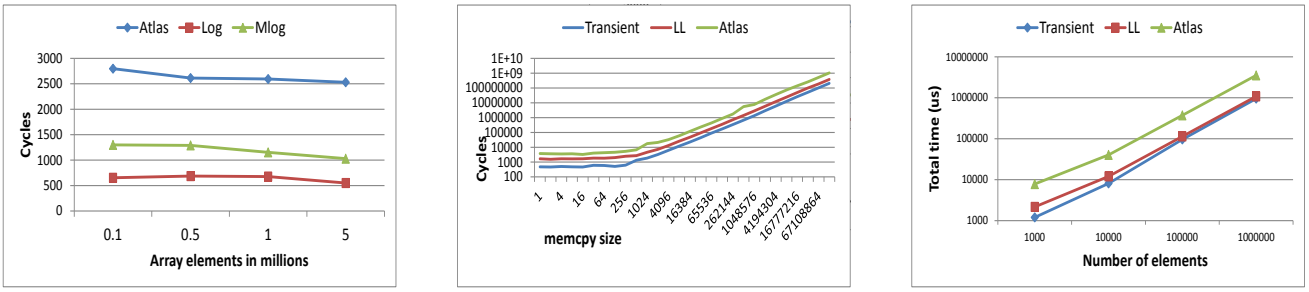
6. Experimental results

6.1 Methodology

Since real NVRAM devices are not yet available, DRAM was used for simulating NVRAM. Linux *tmpfs* [18] was used for "persisting" data and logs. Although data on *tmpfs* does not persist past a system shutdown, it otherwise provides a directly mapped, byte-addressable persistent (across process shutdowns) memory. We successfully performed crash-recovery testing of these programs but a more extensive testbed is required for full correctness testing. All experiments were performed on a Red Hat Linux Intel 4 socket quad-core (16 total) Xeon E7330 machine with 4 sockets running at 2.4GHz. Results are averages over 6 runs.

In general, we found that NVRAM-based programs using our consistency model are 2 to 3 orders of magnitude faster than programs persisting data on disks through mmap. However, this depends on the workload and the amount of data persisted. We describe similar results with a large real-life application later in this section. But our primary focus here is to understand the costs and inherent bottlenecks in our system. A number of modes were used in the results. (1) Default: This is the default mode in Atlas and uses CB-based log allocation, log elision, and the Hybrid cache flushing scheme. (2) Log: Same as Default, except that all cache flushes are turned

[10] All updates to persistent locations within FASEs have to be invariably logged, identical to updates within ACID transactions.



(a) Array Write (b) Failure-atomic memcpy (c) Perfect shuffle singly linked list

Figure 7. Comparison of execution cycles and timings for different configurations

off. (3) `MLog`: Same as `Log`, except that the log entries are allocated using `libc malloc`.

6.2 Serial Array Write Microbenchmark

To understand the cost of logging and cache flushing, we begin with a serial program `TransientArray`. It has a single tight loop which in iteration `i`, writes to the i^{th} element of an array of integers. With a million elements, the loop takes around 9 million cycles to complete. When the array is allocated out of `NVRAM` (let's call it `PersistentArray`) and compiled using Atlas, the program takes around 30 million cycles. Thus for simple stores, the runtime becomes 3.3x if we want durability.

Atlas correctly determines that logging can be elided for the stores but the dirty cache lines still have to be flushed out. We find that log elision analysis takes 12 million cycles. Flushing user data from caches using the default `Hybrid` scheme takes 8 million cycles. 62K cache line flushes were issued, implying 129 cycles for each of them on an average. This is an order of magnitude reduction in the number of flushes compared to the `Eager` mode, which would issue a million of them.

We then created a failure-atomic version of `PersistentArray`, called $\texttt{PersistentArray}_{fa}$, where every store occurs within a separate `FASE`. Each iteration creates 3 log entries,[11] for the start and end of the `FASE` and the store itself. The number of cache line flushes per iteration is found to be 5.6 on an average, 1 for data and the rest for logging. For each log entry, 1.5 cache line flushes are required on an average. As the frequency of flushes was higher in this case than that of `PersistentArray`, their average latency increased to around 300 cycles. We also found that the helper thread completed processing and pruning of 40% of the log structure, giving an indication of the latency of the helper's job. It also shows that the helper thread did fall behind the user threads and hence, in the event of a crash, the recovery phase would have to advance the consistent state. We found that if the helper was disabled, the overall runtime increased by 5%, indicating minimum perturbation of the user thread.

As the array size is varied, Figure 7(a) plots the number of cycles required by 3 different modes of execution: `Default`, `Log`, and `MLog`. On an average, the default scheme takes 2633 cycles or 1097 ns per iteration, providing a throughout of 911K transactions per second. Flushing caches takes 75% of total execution time. If we ignore cache flushing, `Log` takes 641 cycles or 267 ns per iteration, providing a throughput of 3.7 million transactions per second. Note that, on an average, logging a single write takes 213 cycles. Instead of using a circular buffer for log management, if we use regular libc malloc (`MLog`), the cost of logging goes up significantly as shown in Figure 7(a).

[11] This uses a special form of "critical section" that avoids actual synchronization. Such an `FASE` could be optimized to avoid any logging. We leave that for future work.

6.3 Comparison with low-level hand-written code

Given that Atlas is a high-level programming model, we would like to understand how close it is to low-level, expert-written code where logs and cache flushes are managed manually by the programmer. The first benchmark we considered was a `memcpy`. The transient version (`Transient`) is the baseline with no atomicity guarantees and the low-level persistent version (`LL`) is written with manual logging. The manually maintained log is a record of what is being copied, containing just the `from` and `to` pointers and a `size`, and is created and flushed before attempting the `memcpy`. If the `memcpy` is interrupted, the recovery process looks at the log and re-issues the copy. Once the `memcpy` returns, the log is removed. Using Atlas on the other hand, the programmer only needs to place the `memcpy` within a `FASE`.

Note that the manual logging approach (and the recovery phase) here is quite different from the Atlas implementation. The low-level version takes advantage of the `memcpy` semantics and consequently needs to persist a very small log for every `memcpy`. On the other hand, Atlas needs to ensure that the old values are made durable in the log before attempting the memcpy, typically requiring more cache line flushes. Figure 7(b) shows the comparison for copying the specified size of data. Atlas runtimes are within 4-5x of the `Transient` and within 2-3x of the `LL` runtimes.

We consider another program that is multithreaded, a variant of a singly linked list. A total of `N` elements are inserted starting with an empty list. The `LL` version is written using copy-on-write technique. Figure 7(c) shows the results with a varying number of inserted elements. Again, the durability costs are found to be around 4x on an average.

6.4 Overheads and characteristics in different benchmarks

6.4.1 Persistent data structures

We present detailed results on more programs collected from a variety of sources. `queue` is a multithreaded benchmark we wrote based on the blocking algorithm of Michael and Scott [15]. Two threads insert and remove 100,000 elements from the persistent queue, making it a write-intensive benchmark. `cow_al` is a multithreaded copy-on-write array-based persistent list.[12] The driver maintains 2 threads, each of which repeatedly performs a mutation followed by a traversal of the list such that each thread makes a total of a million writes to persistent locations. A distinguishing characteristic of this benchmark is that most of the code in this benchmark is outside critical sections.

[12] This benchmark is inspired by Java's CopyOnWriteArrayList.

	$\frac{Def}{Base}$	$\frac{NoFlush}{Base}$	$\frac{LogAll}{Base}$	$\frac{Eager}{Base}$	$\frac{Delayed_l}{Base}$	$\frac{Delayed_g}{Base}$
queue	3.9	2.1	4.4	3.5	4.7	3.5
cow_al	3.0	2.7	8	7	4.3	3.3
radiosity	5	4	25	16	5	5
raytrace	4.1	3.3	3.9	46	4.3	4.3
volrend	6	4	6.4	67	5.7	5.5
dedup	1.3	1.2	1.4	5	1.7	1.7
memcached	1.6	1.2	2.8	2	2.2	1.8
mdb	4	2.5	12	5.2	5.3	5.5

Table 1. Runtime ratios in different modes

6.4.2 SPLASH2 programs with pervasive persistent accesses

The next 3 programs, radiosity, raytrace, and volrend are taken from the SPLASH2 benchmark suite [23][13]. Typically, in order to convey to the underlying system that some data is persistent, the program has to allocate it out of a PR. However, for the SPLASH2 programs, instead of adding allocator calls, we use a mode available in Atlas whereby all non-stack locations are treated as persistent locations. Typically overheads are proportional to the amount of data made durable. Here we take an essentially arbitrary application and add durability to *all* data structures that are not stack-allocated, providing a near-worst-case picture of the amount of overhead that could be introduced.

6.4.3 Persisting the hash table of a deduplication benchmark

dedup is a deduplication kernel that is part of PARSEC 1.0 [2]. This program removes duplicate chunks of repeating data, mimicking compression techniques used in backup storage systems. We focus on the stage that computes a hash value for a unique chunk and builds a global database of chunks indexed with the hash values. We take the central hashtable data structure in dedup and make it durable in NVRAM. It is useful to have a persistent version of the hashtable since it acts as a cache of the unique key-value pairs and would enable a quick restart in the event of a server crash. We examined a number of synchronized regions to ensure that they conformed to the model we described in this paper. All of them did.

6.4.4 A Persistent Version of memcached

Memcached [7] is a main memory key-value cache used in cloud and web workloads. In the current incarnation, the cache is transient; so all of the cached information is lost in the event of a failure. But it would be nice to have the cache available across system restarts.

Memcached has a few key data structures. There is a hashtable for the key-value pairs. There are least recently used (LRU) lists (heads and tails) to determine eviction order from the cache when it is full. A slab-allocation based memory manager is used for efficiency purposes. We persist all of the above data structures. In addition to the cache, persisting the LRU lists and the slab allocator information allow maintenance of eviction order and memory management information across server crashes.

While manipulating the hash table entries or the LRU lists, memcached always holds a global cache lock. At the boundaries of the resulting critical sections, the program state is always consistent. During some operations in slab management, the global cache lock and a global slab lock are held leading to a FASE with an inner CS. Like before, data structures are kept consistent at FASE boundaries.

[13] We arbitrarily chose three programs we knew to be heavy lock users. We did not experiment with other members of the suite.

	Logs	Elision (%)	# FASEs	Flushes
queue	$0.9M$	50	$0.3M$	0.95
cow_al	$2.4K$	99	$1.2K$	11
radiosity	$1.4M$	95	$0.3M$	33
raytrace	$0.7M$	99	$0.15M$	3
volrend	$0.2M$	99	$0.07M$	3.8
dedup	$1.4M$	5	$0.26M$	1
memcached	$11M$	87	$4M$	1
mdb	$4K$	97	$1.3K$	1

Table 2. Miscellaneous Statistics

6.4.5 Memory-mapped Database (MDB)

MDB is a B-tree based key/value store designed to replace Berkeley DB as OpenLDAP backend. It stores key/values in a memory-mapped file where retrievals happen through memory and updates to the database happen through file I/O at transaction commit points. Using the Atlas durability and consistency semantics, we were able to completely eliminate file I/O and shorten the length of critical sections, thus improving the performance of MDB.

6.4.6 Detailed results

Table 1 shows the runtimes of different modes as compared to that of the base mode without any durability. The default is a clear winner in essentially all the cases, its usual cost being around 3x-4x on an average, even for programs with worst-case characteristics. When flushes are removed (NoFlush), the runtime drops further as expected. If log elision were not present in the default technique, the runtimes would go up as in LogAll. The last 3 columns show the results with alternative flushing strategies.

Table 2 shows the number of logs created and the number of times logging is avoided (as a percentage). The log elision percentage is quite high in general, indicating a large savings in runtimes. The number of FASEs found in each program is tabulated as well. The last column presents the ratio of cache line flushes from eager and default modes. This number is often substantial too, explaining why the default Hybrid mode can sometimes vastly outperform the Eager mode.

6.5 Log Elision Failures (outside a FASE)

Section 5.1 outlines the condition under which a persistent store outside a FASE must be logged. Other than impacting runtime performance adversely, such a failure in log elision implies that I/O in corresponding code regions cannot be immediately externalized. So an interesting data point is finding out the frequency of log elision failures. It turns out that for the applications we studied (e.g. the ones shown in Table 2), only 2 had non-zero log elision failures: dedup had a handful of them and though radiosity had a noticeable number of log elision failures, it was still less than 1%.

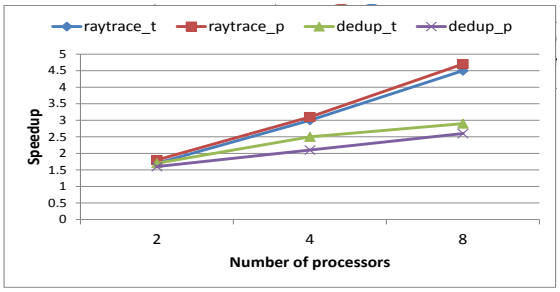


Figure 8. Impact of durability on scalability

	Disk	DRAM	NVRAM
mtest	962057	4840	50014
fillseqsync	238682	7316	60262
fillrandsync	238310	8123	59577

Table 3. MDB-based workloads performance

While more studies are required, this may indicate that in many lock-based applications, log elisions are mostly successful.

6.6 Impact of durability on scalability

Figure 8 shows the speedup for transient and persistent versions of raytrace and dedup on 2, 4, and 8 processors. Even with durability support, the scalability curve appears to hold. We saw similar results for other applications that were designed to scale[14].

6.7 Speedup over existing durability support using mdb

Table 3 shows the performance of Atlas compared to no durability on one hand and disk-based durability on the other. MDB was used as the database for these results. mtest first inserts 1500 key/value pairs, where each key is a 4-byte integer and value is a 32-byte string. Next, it iterates through all the key/value inserted. Then, it deletes some entries where each deletion is a separate transaction, followed by two iterations - forward and reverse - of the entries remaining. Fillseqsync and Fillrandsync are parts of Google levelDB benchmark suite. Fillseqsync inserts 1.5 million pairs of 16-byte key and 100-byte value in a sequential order of keys. Likewise, fillrandsync inserts 15 thousand pairs in random key order.

Note that `NVRAM`-based durability is around an order of magnitude faster than disks[15]. The timing ratio between using `DRAM` and `NVRAM` is similar to what we saw earlier for other applications.

6.8 Summary of results

In-memory durability can be orders of magnitude faster than when using disks. But compared to executing on volatile RAM, there is clearly a cost to it. However, our results indicate that it can be kept reasonably low. We have seen overheads varying from a small percentage to around 3-4x of execution time; however, much of this is due to cache line flushes that are expected to improve over time. In our system, cache flushing and logging are the predominant costs but substantial optimizations can be performed to keep their costs down. What appears to be the most important aspect is that Atlas provides the capability to persist data at a very low programming cost but with encouraging performance characteristics for emerging memory technologies.

[14] Note that some applications such as memcached do not scale inherently, even for the transient version.

[15] In this case, the disk's write-caching was on, otherwise the difference should be even bigger.

7. Related work

Transparent user-level checkpointing of the entire process state has been explored [16]. The model is different from ours; it does allow a more transparent application restart, at the expense of what appear to be substantial open implementation challenges, and probably tolerance for a reduced set of failures. For example, we expect that our approach will tolerate a significant set of software failures, though clearly not all of them.

Systems such as `Mnemosyne` [22] and `NV-heaps` [6] built consistency mechanisms on top of persistent regions and durable transactions. Our work aims to provide similar semantic guarantees for lock-based programs. The problem of providing meaningful semantics for safe re-execution of multithreaded programs has been looked at earlier [24]. A checkpointing mechanism was developed for Concurrent ML using a language abstraction called stabilizers. In contrast, our technique provides failure-atomic semantics that lead to a globally consistent state and all of this is done without any annotations from the programmer. In general, it appears to us that using constructs like stabilizers can be error-prone without global knowledge of the dynamic nature of the program.

8. Conclusions

We presented failure-atomic semantics for lock-based programs. We showed how that can be implemented to capture a globally consistent snapshot. This has a significant impact on programming with non-volatile memory. Results from our implementation show that in-memory durability on `NVRAM` using our programming model has encouraging performance characteristics. We presented a number of optimizations and pointed out the primary sources of overheads in our system.

References

[1] ATKINSON, M. P., DAYNES, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. An Orthogonally Persistent Java. *ACM SIGMOD Record 25*, 4 (Dec 1996), 68–75.

[2] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, Jan. 2008.

[3] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. K. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)* (2005).

[4] BOEHM, H.-J., AND ADVE, S. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), pp. 68–78.

[5] CHAKRABARTI, D. R., AND BOEHM, H.-J. Durability semantics for lock-based multithreaded programs. In *HotPar '13: 5th USENIX Workshop on Hot Topics in Parallelism* (Jun 2013).

[6] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS '11: Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar 2011), pp. 105–117.

[7] DORMANDO. Memcached: a distributed memory object caching system. At `http://memcached.org`, retrieved 11/2013.

[8] GROSSMAN, D., PUGH, B., AND MANSON, J. What do High-Level Memory Models Mean for Transactions? In *Proceedings of the 2006 ACM SIGPLAN Workshop on Memory System Performance and Correctness* (Oct. 2006), pp. 63–69.

[9] INTEL CORP. Intel64 and IA-32 Architectures Software Developer's Manuals Combined. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, retrieved 11/2013.

[10] J-BOEHM, H. Transactional Memory should be an Implementation Technique, Not a Programming Interface. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism* (Mar. 2009).

[11] LARUS, J., AND RAJWAR, R. *Transactional Memory*. Morgan and Claypool Publishers, 2007.

[12] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proc. of the 36th International Symposium on Computer Architecture* (Jun 2009), pp. 2–13.

[13] MANSON, J., PUGH, W., AND ADVE, S. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005).

[14] MENON, V., BALENSIEFER, S., SHPEISMAN, T., ADL-TABATABAI, A., HUDSON, R., SAHA, B., AND WELC, A. Single global lock semantics in a weakly atomic stm. In *TRANSACT '08* (Feb 2008).

[15] MICHAEL, M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1996), pp. 267–275.

[16] RIEKER, M., ANSEL, J., AND COOPERMAN, G. Transparent user-level checkpointing for the native posix thread library for linux. In *PDPTA '06: Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications* (Jun 2006), pp. 492–498.

[17] SAXENA, M., SHAH, M., HARIZOPOULOS, S., SWIFT, M., AND MERCHANT, A. Hathi: Durable transactions for memory using flash. In *DaMoN: Proceedings of 8th ACM/SIGMOD International Workshop on Data Management on New Hardware* (2012).

[18] SNYDER, P. tmpfs: A virtual memory file system. In *Autumn European Unix Users' Group Conference* (1990).

[19] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *Nature 453* (2008), 80–83.

[20] TRANSACTIONAL MEMORY SPECIFICATION DRAFTING GROUP. *Draft specification of transactional language constructs for C++*, Feb 2012. At https://sites.google.com/site/tmforcplusplus.

[21] VOLOS, H., TACK, A. J., GOYAL, N., SWIFT, M. M., AND WELC, A. xcalls: safe i/o in memory transactions. In *EuroSys* (2009), pp. 247–260.

[22] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar 2011), pp. 91–103.

[23] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[24] ZIAREK, L., SCHATZ, P., AND JAGANNATHAN, S. Modular Checkpointing for Atomicity. *Electr. Notes Theor. Comput. Sci. 174*, 9 (2007), 85–115.

Appendix

A. Implementation of log_str, log_acq, and log_rel

Figures 9 and 10 show the main steps of each of the primary entry points for logging. In `log_str`, a check is made to make sure that the location is indeed persistent. This filter is required because, in the event of incomplete information, a compiler may generate a logging call for a transient access. Next, the implementation checks whether logging can be elided for this instruction, as described in Section 5.1. Assuming that both these conditions are met, a log entry is created and published by attaching it to the shared log. In addition to similar creation and publication phases, `log_acq` and `log_rel` help track the happens-before relationships between synchronization operations.

```
1: void log_str(void *addr, size_t sz) {
2:   if (!is_persistent(addr, sz)) return;
3:   if (elide_log(addr, sz)) return;
4:   LE *le = create(addr, sz, STORE);
5:   publish_log(le);
6: }
1: void log_acq(void *addr) {
2:   LE *le = create(addr, ACQ);
3:   ReleaseInfo *ri = find_rel_info(addr);
4:   if (ri) le->ValOrPtr = ri->LeAddr;
5:   publish_log(le);
6: }
1: void log_rel(void *addr) {
2:   LE *le = create(addr, REL);
3:   publish_log(le);
4:   add_rel_info(le);
5: }
1: void publish_log(LE *le) {
2:   if (last_le == NULL) {
3:     LHead *new_lh = allocate_from_NVRAM(...);
4:     vbarrier(le);
5:     new_lh->LePtr = le;
6:     LHead *old, *tmp;
7:     do {
8:       tmp = LEP;
9:       new_lh->Next = tmp;
10:      vbarrier(new_lh);
11:      old = CAS(&LEP, tmp, new_lh);
12:    }while (old != tmp);
13:    vbarrier(&LEP);
14:  } else {
15:    if (isCacheLineAligned(le)) vbarrier(le);
16:    last_le->Next = le;
17:    vbarrier(last_le);
18:  }
19:}
```

Figure 9. Overview pseudo-code for logging entry points

Publishing a Log Entry: Once a log entry of the appropriate type is created, it has to be attached to the shared log. This publish allows the helper thread to examine the created log entry for computation of a consistent state. As shown in Figure 9, `publish_log` first checks whether this is the first entry created by the executing thread. If that is the case, it creates a thread-specific log header, sets its fields appropriately, and inserts the header at the head of the linked list pointed to by `LEP` in a CAS-loop. In the steady state, publishing will translate to a single store (line 12) setting the `Next` field of the previous log entry (created by the executing thread) to the log entry being published.

Creating Happens-before Relations between Log Entries: Consider the hash table `ReleaseTab` again. Two operations, `add_rel_info` and `find_rel_info` are defined on this hash table. Both are implemented in a non-blocking manner. `add_rel_info` checks whether a hash table record already exists for the lock address (`le->Addr`) corresponding to the log entry `le` under consideration. If it exists, its field `LeAddr` is made to point to `le` as shown in line 3 in `add_rel_info`. This is why the field `LeAddr` is of atomic type. Additionally, the above assignment is correct since the insertion happens under the protection of the lock (denoted by `le->Addr`) and hence it is guaranteed that no other thread can concurrently attempt an insertion of a log entry corresponding to the same lock. Without this guarantee, a CAS-loop would be required.

If an existing record is not found (lines 4-12 in add_rel_info), a new record is created and inserted at the head of the linked list for the bucket corresponding to le->Addr. A CAS-loop is required since another thread may be concurrently attempting an insertion of a log entry with a lock address whose hash collides with that of le->Addr. The non-blocking insertion also requires that the bucket pointers or the entries in ReleaseTab be of atomic type. Since an insertion always happens at the head of the linked list for a given bucket, find_rel_info is allowed to identify the bucket, read off the bucket pointer atomically, and scan through the obtained linked list in a concurrent but safe manner. A find operation succeeds only if the lock-address matches as shown in line 4 of find_rel_info. Note that while hash collisions are possible here, no false conflicts are possible and the log entry for the last release of a lock-address is computed precisely. While not shown here, a delete operation on ReleaseTab is provided by maintaining a valid bit in every hash table record and it can be combined with eventual garbage collection of deleted records.

```
1: ReleaseInfo *find_rel_info(void *lock_addr) {
2:   ReleaseInfo *ri = get_header(lock_addr);
3:   while (ri) {
4:     if (ri->LeAddr->Addr==lock_addr) return ri;
5:     ri = ri->Next;
6:   }
7:   return NULL;
8: }
1: void add_rel_info(LE *le) {
2:   ReleaseInfo *ri = find_rel_info(le->Addr);
3:   if (ri) ri->leAddr = le;
4:   else {
5:     ReleaseInfo *new_ri = create_rel_info(le);
6:     ReleaseInfo *old, *first;
7:     do {
8:       first = get_header(le->Addr);
9:       new_ri->Next = first;
10:      old = CAS(get_header_pointer(le->Addr),
                   first, new_ri);
11:    }while (old != first);
12:  }
13:}
```

Figure 10. Overview pseudo-code for logging entry points

B. Visibility barriers for logging

In the introduction, we mentioned the need for an ordering constraint in the context of Figure 1. Atlas provides an internal interface called the visibility barrier (or vbarrier) that accepts the address of a memory location. A call to this interface explicitly flushes out the corresponding cache line.[16].

In our implementation, vbarrier is used for user data as well as for logging. Consider the use of vbarrier in the instrumented assignment from Section 4. This call ensures that the effect of a write to user data reaches NVRAM.

The visibility barriers necessary for logging are shown in publish_log in Figure 9. Note that a thread-specific log header is 16 bytes whereas a log entry is 32 bytes in size. Recall that our log allocator guarantees that an object of either of these types resides on a single cache line. Lines 2-14 handle the first log entry created by a thread requiring three barriers: one for the log entry, one for the header, and another for LEP. Lines 15-17 show the requirement at steady state. If the log entry under consideration $\texttt{LE}_i$ starts at a cache line boundary, it implies that the previous log entry $\texttt{LE}_{i-1}$ is in a different cache line. This scenario requires two barriers, one for $\texttt{LE}_i$ and another for $\texttt{LE}_{i-1}$ after setting the Next field of $\texttt{LE}_{i-1}$. On the other hand, if $\texttt{LE}_i$ does not start at a cache line boundary, it implies that $\texttt{LE}_i$ and $\texttt{LE}_{i-1}$ are in the same cache line, following the guarantees provided by the log allocator. In this case, we set the Next field of $\texttt{LE}_{i-1}$ and then flush out the cache line containing both $\texttt{LE}_i$ and $\texttt{LE}_{i-1}$. This optimization takes advantage of the insight that since cache lines are flushed out atomically, writes to the same cache line become visible in memory in the same order that they are executed. Thus only one barrier is sufficient to guarantee that when $\texttt{LE}_i$ is attached to the log, the contents of $\texttt{LE}_i$ are either already visible or become visible atomically with the attachment.

[16] Note that the cache line may already have trickled out to memory following usual replacement policies. This would make the explicit call unnecessary overhead but there is no way for the program to tell.

```
void recover() {
  for (i=0;i<num_threads;++i) recover(i);
}
void recover(tid) {
  LE *le = last_log(tid);
  while (le) {
    if (le->Type == ACQ) visited(le)=true;
    else if (le->Type == STR) undo(le);
    else if (le->Type == REL) {
      LE *acq_le = ha(le);
      if (acq_le && !visited(acq_le)) {
        last_log(tid) = prev(le);
        recover(get_tid(acq_le));
      }
    }
    le = prev(le);
  }
}
```

Figure 11. Recovery Algorithm

C. More Details on Recovery Phase

The recovery phase first applies the same algorithm as in the helper thread to ensure that the consistent state is advanced as much as possible. The log entries are then traversed, honoring the embedded happens-before relations, reverting any inconsistencies in user data.

Figure 11 is a sequential algorithm that replays the log and undoes the effects of updates to persistent memory in reverse happens-before order. Before the algorithm is run, the log entries of each thread are traversed in execution order to create data structures representing the following maps:

- Given a log entry $\texttt{LE}_1$, prev($\texttt{LE}_1$) returns the log entry, if any, created by the same thread just before $\texttt{LE}_1$ during program execution.
- Given a log entry $\texttt{LE}_1$ of type REL, if $\texttt{LE}_2$ is the next acquire operation on the same object, then ha($\texttt{LE}_1$) = $\texttt{LE}_2$.
- Given a thread tid, last_log(tid) is set to the last log entry created in execution order by thread tid. last_log(tid) is updated as the log entries are replayed during recovery.
- Given a log entry $\texttt{LE}_1$, get_tid($\texttt{LE}_1$) returns the id of the thread that created $\texttt{LE}_1$.

The recovery algorithm starts from each thread tid, retrieves its last log entry, and starts visiting its log entries in reverse creation order. If a log entry for an acquire is encountered, no action is taken other than marking it visited. If a log entry for a store is encountered, the undo value is written to the stored memory location. If a log entry for a release is encountered and this release happens before an acquire in another thread $\texttt{tid}'$, a switch is made to the logs of $\texttt{tid}'$ and replay starts with the last log of $\texttt{tid}'$. This

process is repeated until each remaining log entry has been visited exactly once.

LEMMA 2. *The recovery process terminates and every log entry corresponding to STR is replayed exactly once.*

Proof. Every step through the log entries of a given thread retrieves the previously created log entry. So if a switch to another thread is not done, all log entries of the thread will be visited and visited only once. A switch to another thread is done only if the happens before relation is induced by an acquire in the switched thread such that the acquire node has not yet been visited. This, coupled by the fact that happens-before relation is acyclic, a new log entry for some thread is visited by the recovery process at every step. This shows that the process will terminate and a given STR entry will be replayed exactly once. •

THEOREM 1. *Every memory location undone during recovery is restored to its value found in the globally consistent state.*

Proof. Given a memory location `loc`, consider the log entries that contain undo values for `loc` and available to the recovery process. Among them, the one (say u_1) that was written out first during execution contains the globally consistent value in its undo entry. So we must show that u_1 is replayed after replaying all the others. From the preceding lemmas, we know that every log entry is replayed exactly once. Hence, without loss of generality, consider a second log entry u_2 for the same memory location `loc`. Since the program is assumed to be free of data races, u_1 happens before u_2. If u_1 and u_2 are written out by the same thread, u_2 will be visited before u_1 by the recovery process since the algorithm starts with the last log entry and works its way backwards to the first log entry available for a thread. If u_1 and u_2 are created by different threads, let u_1 be created by t_1 and u_2 be created by t_2. If the recovery phase starts with thread t_1, there will be a log entry u_{r1} of type `REL`, created by t_1 and encountered before u_1. Since u_1 happens before u_2, there must be a log entry u_{a1} of type `ACQ`, created by thread t_2, such that u_{a1} HB u_2. This causes the recovery process to switch to thread t_2 and start replaying log entries in reverse creation order. Now, the only way for the recovery process to switch back to thread t_1 before replaying u_2 would be encountering a log entry u_{r2} of type `REL`, created by thread t_2 such that u_{r2} HB u_{a2}, where u_{a2} is a log entry of type `ACQ` created by thread t_1. Note that u_{a2} HB u_{r1}, otherwise u_{a2} is already visited and the switch will not happen. This would mean that u_{a2} HB u_{r1} HB u_{a1} HB u_{r2} HB u_{a2}, a cycle in the happens before relation, an impossibility. •