Durability Semantics for Lock-based Multithreaded Programs

Dhruva R. Chakrabarti
Hewlett-Packard Laboratories, USA
dhruva.chakrabarti@hp.com

Hans-J. Boehm
Hewlett-Packard Laboratories, USA
hans.boehm@hp.com

Abstract

Non-volatile storage connected as memory (NVRAM) offers promising opportunities for simplifying and accelerating manipulation of persistent data. Load and store latency is potentially comparable to that of ordinary memory. The challenge is to ensure that the persisted data remains consistent if a failure occurs during execution, especially in a multithreaded programming environment. In this paper, we provide semantics for identifying a globally consistent state for a lock-based multithreaded program. We show how to conveniently ensure that programs are returned to such a globally consistent state after a crash. We discuss challenges and opportunities along the way, and explain why adding durability to transactional programs may be less expensive.

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) [9, 16] 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 [15] and phase change memory (PCM) [10] provide an interesting twist to programming since they allow CPU stores to persist data directly at DRAM-like speed. Data in NVRAM lives beyond the lifetime of the creating process. 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 that it be possible to identify and persist program states that are consistent and this is no easy task for multithreaded programs.

Transactional memory (TM), as the name suggests, 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) but no durability. However, recent work explored adding durability to TM in the context of NVRAM and Flash [8, 18, 13].

Durable TM is clearly a very attractive model for NVRAM. But TM, even in its original form, 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 ones based on TM [6]. Additionally, some constructs, such as condition wait, arguably do not lend themselves well to the TM paradigm. 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++ [16] requires the co-existence of locks with transactions.

So, to take advantage of NVRAM-based data reuse in general multithreaded programs, durability semantics should be added to lock-based programs. But durable data has to be consistent as well, otherwise it is not reusable. Data structures mutated by lock-based programs are trivially consistent at normal program termination. However, this is not adequate in the presence of NVRAM once failures are taken into consideration. We

need some guarantee that data structures are consistent, even if a failure occurs at an inopportune moment of execution. Currently there is no well-established notion of what this means or how to achieve it. In the absence of NVRAM, it does not matter. But the emergence of NVRAM changes the fundamental assumptions and opens up new opportunities if we can capture intermediate consistent states.

This paper explores appropriate semantics that would be required for lock-based programs in the context of NVRAM and sketches an implementation route. Section 2 presents our programming model and elaborates on the properties we want. Section 3 presents results obtained while persisting data structures in a PARSEC [5] kernel and a widely used caching system. The goal is to understand the overheads and any effects on scalability. We then present some related work and conclude.

2 System Assumptions and Programming Model

NVRAM devices are expected to be connected as memory and accessed using regular CPU loads and stores. The result is fast persistence of program objects. Potentially in-memory data structures that are already maintained by the application can be reused directly. But even in the presence of NVRAM, there will be volatile buffers and caches in the memory hierarchy, simply because of the performance advantages they provide. This implies that during program execution, some of the state may reside in volatile structures and the rest in NVRAM. Low-level interfaces such as memory fences and cache-line flushes can be used to ensure that certain data is forced out of volatile structures and into NVRAM at appropriate program points.

In our programming model, programmers use containers called *persistent regions* (PR) [3, 18, 8] 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[1]. 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. Data not in a PR is considered logically transient.

We assume a fail-stop or crash-recovery model. Program state in persistent regions survives a tolerated failure,[2] other program state does not. After a failure, persistent regions should contain a consistent program state, which can be used to recover the data when the application is restarted. This state must be consistent in spite of the fact that contents of machine registers, caches, and DRAM memory regions have disappeared.

[1] 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.

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

Figure 1: Structure of an NVRAM Program

Applications will normally be structured as in Figure 1. 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, read by the application.

2.1 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) [9], 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. They also already indicate when important data structures are in a consistent state. In order to ensure thread-safety, they acquire locks to form critical sections (CS) in which data structures are modified and thus temporarily inconsistent. This is typically required to ensure the absence of data races.[3]

[2] We assume that the hardware defines the notion of "tolerated failure", and that it includes at least power failures. Clearly some failures, e.g. a direct hit by a large meteorite, are not tolerated.

[3] Our technique, like the underlying programming language, expects the input program to be free of data races. In the presence of data races, the program state identified as consistent may not be what the programmer intended. We expect the consequences to be similar to the

As in the transactional case, we 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 doesn't hold, for example, if the client implements its own mutexes on top of the system-provided ones. At that point we no longer recognize critical sections, and thus no longer guarantee consistency in spite of the absence of data races. Programmer-inserted annotations, indicating the appropriate synchronization operations, may be helpful in that context.

Our assumption above is mildly restrictive, and it may take some programmer effort to ensure that it holds. For example, data accessed by only a single thread will typically not be protected by a lock, and thus may be inconsistent outside of a critical section. This can be addressed by introducing a "dummy" critical section around such updates, or by introducing an additional language construct for such dummy critical sections.

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 neither inhibits the approach, though the latter does add significant implementation expense over the transactional case.

2.2 Outermost critical sections are failure-atomic

What is the unit of durability of a lock-based program? Consider a dynamic execution trace $A_1A_2...A_n$ of a single thread t, where A_i refers to an instruction executed. A section of this trace comprising $A_i...A_j$ forms an outermost critical section (OCS) if the following hold:

- The point just before A_i is thread-consistent, i.e. t holds no locks there.
- A_i acquires a lock.
- A_j releases a lock, not necessarily the one acquired in A_i.
- The point after A_j is again thread-consistent.

Thus, in the above trace, two dynamically occurring thread-consistent program points are separated by (and adjacent to) an OCS. Locks do not have to nest perfectly for an OCS to be identified. For example, hand-over-hand locking poses no problem. We ensure that each OCS is failure atomic: If any updates performed within an OCS are visible after a failure, then all of them will be. We do not treat the point after a non-OCS critical section as thread-consistent, since updates of invariants may be in progress in an outer one.

existing concurrent execution hazards caused by data races.

```
x, y are persistent and initially x=y=0
     T1                        T2
1: lock(l1)              4: lock(l2)
2: y = x                 5: lock(l1)
3: unlock(l1)            6: x = 1
                         7: unlock(l1)
                         8: <T1 executes here>
                         9: ...
                         10: unlock(l2)
```

Figure 2: An example program: Outermost critical sections are failure-atomic and can have happens-before relations among them.

It may often be the case that invariants that matter are indeed maintained at boundaries or even within arbitrary critical sections, but there is no way to automatically verify that — we believe that identifying an OCS as the unit of failure-atomicity provides a conservative abstraction that allows persisting consistent data without any additional effort on the part of the programmer. Providing failure-atomicity for critical sections at a finer granularity appears to be intractable since the association between a shared datum and the corresponding protecting lock cannot be automatically inferred in the general case.

2.3 Persistence of one outermost critical section may depend on another

Semantics of nested critical sections are different from those of nested atomic sections. While isolation is implicit at the granularity of individual critical sections, we are providing durability or failure-atomicity at the OCS-granularity (Section 2.2), an apparent mismatch. It follows that even if updates within an inner CS (say C) within a given OCS are exposed to another thread, those updates within the inner CS may have to be rolled back if for some reason the corresponding OCS fails to complete or has to be rolled back. If that happens, any update of a persistent location, dependent on a change made by C and executed by another thread, must also be rolled back. These cross-thread constraints should be identified by the underlying implementation. Note that our model preserves the pessimistic nature of lock-based programs, so any roll-back happens only during recovery after abnormal program termination, unlike the transactional model.

Consider the program in figure 2 with 2 threads, T1 and T2. There are 2 OCSes: o_1 in lines 1-3 in T1 and o_2

in lines 4-10 in T2. Consider the indicated dynamic interleaving where o_1 executes in full between the statements in lines 7 and 9. However, if the program crashes after o_1 has completed but before T2 can finish executing line 9, some of the updates in T2 may be incomplete. By the reasoning in Section 2.2, T2 has to be failure-atomic and hence the update in line 6 must not be visible in persistent memory. In order to preserve a globally consistent state in persistent memory, the effects of updates made to persistent locations in o_1 must not be visible either — otherwise, after recovery, the persistent state will have values (x=0, y=1) that cannot be obtained in a failure-free execution. This constraint arises because there is a happens-before (hb) dependency, in the sense of e.g. [11] or [7], between o_2 and o_1.

An OCS o_1 dynamically happens before another OCS o_2 if there is a release operation in o_1 that happens before an acquire operation in o_2. It follows that, if one OCS dynamically happens before another, the effects of the latter are visible in persistent memory only if the effects of the former are. Two concurrently executing OCSes may each depend on the results of inner critical sections in the other, resulting in a cyclic hb-relation among OCSes — in such a case, the effects of all of the constituent OCSes must be visible in NVRAM if the effects of any one of them are. We believe that these are the right properties to maintain after crash recovery since they are the ones a newly created thread, playing by the proper synchronization rules, usually relies on.

2.4 Relationship to ACID semantics

Comparison of our proposed semantics to transactional ACID semantics is unavoidable. We do not change any of the ACI semantics of a CS or an OCS in any way. 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. However, the data visible in NVRAM after a restart may be a conservative approximation of what the programmer intended, but should preserve consistency.

The only semantics we add is failure-atomicity or durability, but only at the level of an OCS and only for memory locations that are persistent, i.e. those within a PR. There is no change in privatization and publication semantics as far as thread visibility is concerned. We add publication safety for durability: If the effects of an OCS are visible in NVRAM after a failure, then the effects of all updates to persistent data appearing before that OCS (including those at thread-consistent points) must also appear to be visible in NVRAM. Since OCSes become persistent atomically, a similar guarantee implicitly holds for all critical sections. Publication safety has its costs that are discussed in more detail in Section 2.7.2.

```
x, y, m, p, and q are shared and persistent.
t is local, ready is shared. Both are transient.
Initially x = y = m = p = q = t = ready = 0
     T0              T1               T2
a1: lock(l1)    b11: lock(l2)    d21: lock(l3)
a2: t = ready   b12: p = q       d22: lock(l2)
a3: unlock(l1)  b13: unlock(l2)  d23: q = 1
 4: if (t)       14: x = 1       d24: unlock(l2)
 5:   y = x     c15: lock(l1)    d25: m = 1
                c16: ready = 1   d26: unlock(l3)
                c17: unlock(l1)
```

Figure 3: Example showing why all hb-relations must be tracked: Happens-before relations between references to persistent locations are often imposed by synchronizing through critical sections that contain references to only transient data.

2.5 What about I/O operations?

In order to support our durability model for lock-based programs, it should be possible to buffer an I/O operation in case it needs to be rolled back. This may be achieved by reflecting the semantics of I/O operations in NVRAM and playing them in the current order at an appropriate time. I/O in a transactional setting is a sticky issue [17] and our situation is no different.

2.6 Sketch of an implementation

We log writes to persistent memory locations as well as happens-before relations between synchronization operations. Capturing happens-before relations between lock releases and acquires is sufficient to maintain the necessary dependences between OCSes. The logs are kept in NVRAM so that in the event of a crash, the recovery phase has enough information to reconstruct a consistent state of the persistent data according to the properties outlined in Sections 2.2 and 2.3. Memory fences and cache line flushes at appropriate program points are added automatically to ensure that the log is failure-resilient and can be relied on after a crash. Depending on the dynamic program state and its consistency properties, unnecessary log entries are removed.

2.7 Pitfalls, optimizations

2.7.1 hb-relationships of all synchronization operations must be captured

In our model, all lock acquires and releases must be analyzed for dynamic happens-before relations, including

those for critical sections not containing any updates to persistent memory. Figure 3 illustrates such a scenario. We have numbered the lines and labeled the 4 OCSes with the letters a through d. Since a hb-relation (induced by synchronizing on l1) must exist from the write to x in line 14 (T1) to any read from x in line 5 (T0), there are no data races in the program. For the purpose of this example, assume that ready is a transient flag, so OCS a and OCS c do not update any persistent data. We assert that, in spite of that, the appropriate hb-relation between such OCSes needs to be captured. Consider the interleaved execution: lines 21-24 (T2), lines 11-17 (T1), lines 1-5 (T0), followed by a program crash. If all hb-relations are captured, OCS d hb OCS b hb OCS c hb OCS a. On recovery, OCS d will be rolled back and since OCSes a–c transitively depend on OCS d, all persistent updates in Figure 3 will be rolled back and the resulting snapshot in NVRAM will be rendered consistent. But if only hb-relations among OCSes containing updates to persistent locations were captured (i.e. OCS d hb OCS b) for failure-atomicity purposes, OCS a will not be rolled back. This implies that the update to y on line 5 will not be rolled back either leading to an inconsistent state (y = 1, with other persistent variables zero) in NVRAM, something that cannot be obtained in a failure-free execution.

As figure 3 shows, hb-relations between references to persistent locations is often imposed by synchronizing through critical sections that contain references to only transient data. This requires that all synchronization operations be tracked for hb-relations. Note that the situation is different for transactional programs, potentially leading to performance tradeoffs. The effects of updates within an atomic section can be made durable atomically along with the commit. Consequently, the effects of a committed atomic section never need rolling back. In contrast, the effects of a completed OCS may be rolled back during recovery if, for example, the program crashes during execution of another OCS that happens before it (see Section 2.3). To ensure precise identification of such OCSes that need rolling back, explicit happens-before relations must be maintained in lock-based programs, unlike transactional ones.

2.7.2 Optimizations for thread-consistent updates

Unlike atomic sections that provide ACID transactional guarantees, a cross-thread hb-relation may stem from a lock release operation that is dynamically within an OCS. The OCS may expose effects of updates to other threads by releasing a lock before the OCS is finished. Consequently, as we saw in earlier sections, the effects of a completed OCS may have to be rolled back if another OCS executed by another thread, that it depends on, can be rolled back. Consider Figure 3 again with the same interleaving discussed earlier. If OCS d can be rolled back, all the other OCSes have to be rolled back as well. This requires that the effects of updates in lines 5 and 14 be rolled back as well.

Further analysis can be used to determine whether logging can be elided for an update to a persistent location outside an OCS such as those in lines 5 and 14:[4] If an OCS o_1 in thread T1 dynamically happens before another another OCS o_2 in thread T2, then any write to a persistent location, executed by T2 outside an OCS and executed after o_2 in program order, will have to be logged unless it can be proved that o_1 cannot be rolled back. Generally speaking, an OCS cannot be rolled back if it has completed and every OCS that happens before it has completed successfully. As an example, the above condition would determine that, for the interleaving considered earlier in Figure 3, the updates in lines 5 and 14 will have to be logged. But for another interleaving such as lines 21-25, 11-13, line 26, lines 14-17, lines 1-5, the updates in lines 5 and 14 do not have to be logged. We believe that this optimization may be very effective in drastically reducing the overhead of logging in many applications where many updates to persistent locations happen outside OCSes.

This is another scenario where the performance tradeoffs are different for lock-based and transactional programs. Since a committed transaction never needs rolling back, any non-transactional update to a persistent location does not need to be logged[5]. However, that may not be the case for lock-based programs and even when such logging can be elided, it is based on dynamic checking (as outlined above) which has its own cost.

3 Initial experimental results

3.1 Methodology

We developed a prototype implementation of a persistent memory manager, a logging infrastructure, a consistency and recovery manager, and compiler support for inserting calls to the relevant runtime libraries. In order to take advantage of NVRAM durability, a program needs to create a PR and use a special allocator to add data to a PR. The rest is programmer-oblivious with the compiler automatically inserting instrumentation for synchronization operations and updates to persistent locations and the runtime creating and manipulating logs to ensure consistency.

DRAM was used for simulating NVRAM since their access latencies are expected to be comparable. Linux

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

[5] A non-transactional update may still have to be flushed out of cache.

tmpfs [14] 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(R) quad-core Xeon(R) E7330 machine with 4 sockets running at 2.4GHz. Results are averages over 3 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. Keeping the above number as context, our goal is to understand the cost of adding durability to a program that starts off with transient data structures. To this end, we report results for 2 programs. For both, we show that our model is useful by enabling transitions from one consistent state to another. In addition, we present runtimes for 4 configurations – `orig`: the original program that uses transient data structures, `nvram`: the persistent version, `nvram_noflush`: the persistent version without cache line flushes, and `nvram_noflush_opt`: same as the previous but with the optimization from Section 2.7.2 applied. The 3^{rd} and 4^{th} configurations primarily indicate the cost of logging and consistency maintenance for enforcing failure-atomicity of lock-based code; more elaborate description of the cost of flushing processor caches is outside the scope of this paper and is discussed elsewhere [4]. In addition, note that optimizations that reduce the cost of logging and cache flushing are likely as our implementation and NVRAM hardware evolve.

3.2 Persisting the hash table of a deduplication benchmark

We added durability support to `dedup`, a deduplication kernel that is part of PARSEC 1.0 [5]. This program removes duplicate chunks of repeating data, mimicking compression techniques used in backup storage systems. The program breaks the input stream into chunks that are processed in parallel in a pipelined fashion. A number of stages are employed where each stage fetches items from its queue, processes the items, and puts them in the queue for the next stage. 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. For example, the routine `ChunkProcess` computes the hash of a chunk, acquires a lock for the corresponding hashtable bucket, and performs one of the following while holding the above lock. If it is a cache miss, the unique key is inserted into the hashtable and a corresponding item is inserted into the queue for the compress thread. If it is a cache hit, the existing hashtable metadata is updated and a corresponding item is inserted into the queue for the write thread. Since the queues are multithreaded as well, the enqueue operation acquires a queue-specific lock leading to an outer critical section (`OCS`) with an inner critical section. Using `OCS` boundaries for globally consistent states ensures that both the hashtable and the relevant queue are properly updated.

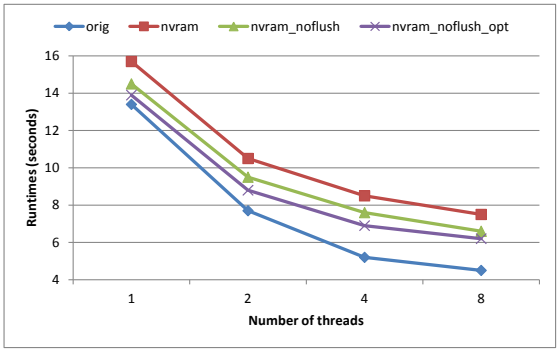

Figure 4: Cost of adding durability to dedup

Running dedup using the input *simlarge*, we found (at a thread count of 4) 260K `OCSes` and 4K inner critical sections dynamically. 900K stores to persistent locations were tracked out of which 857K were inside `OCSes`. 1.4M log entries were created. Figure 4 shows the runtimes for different thread counts for the above input. Persisting data structures in a consistent, reusable manner is useful but does not come for free. At a thread count of 4, the NVRAM version is 63% slower than the transient version. If we ignore cache flushing[6], this number drops to 46%. As stores outside `OCSes` need not be tracked for this workload, the overhead drops further to 33%. In spite of the overheads, `dedup` continues to scale.

3.3 A persistent version of memcached

Memcached [1] is a main memory key-value cache used in cloud and web workloads. It is typically used by front-end services to cache key-value pairs so that expensive

[6]Cost of cache flushing cannot be ignored but will likely be lower in NVRAM-enabled architectures.

trips to the back-end databases are minimized. 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. Though the current architecture of memcached would have to change to accommodate a persistent cache[7], we explored the adequacy of our model for the existing codebase.

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 an `OCS` with an inner `CS`. Like before, data structures are kept consistent at `OCS` boundaries.

We collected results by starting memcached with 4 threads and performing a series of sets and gets. Statistics showed that 4M `OCSes` and 600K inner critical sections were executed dynamically. 22M stores to persistent locations were tracked and 3M of them were inside `OCSes`. 30M log entries were created. The transient original version finished in 17 seconds. The nvram version took 55 seconds, i.e. 3.2x time as the original. Ignoring the cache flushes, the time dropped to 38 seconds, or 2.2x. Optimizing logging for persistent stores outside `OCSes` brings down the time to 28 seconds, or 1.6x. We expect that further tuning of our code will reduce logging overhead further.

3.4 Summary of results

We showed that providing failure-atomicity for lock-based programs helps persist a consistent snapshot that is reusable. This property does not come for free but the overheads need not be prohibitive either. There are some fundamental challenges such as tracking changes and getting the updates out of volatile buffers and caches; our initial results provide an idea of the costs but we believe substantial optimizations are possible. If logging can be eliminated outside critical sections, data can be made durable more efficiently. But this can only be done based on dynamic analysis for general lock-based programs.[8] The situation is different for atomic sections where committed updates need not be undone, indicating that atomic-section-based programs may provide durability more efficiently than their lock-based counterparts.

[7] Depending on the use case, an old cache may cause problems. But a persistent cache does have benefits. See the section titled *Persistent Storage* in [2].

4 Related work

Transparent user-level checkpointing of the entire process state has been explored [12]. 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` [18] and `NV-heaps` [8] 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 [19]. 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.

5 Conclusions

We explored failure-atomic semantics for lock-based programs. We showed how that can be used to reason about a globally consistent snapshot that can in turn be used for persisting consistent states. This has a significant impact on programming with non-volatile memory. We implemented a prototype and presented initial results that show the viability of our approach in addition to pointing out overheads in the system.

6 Acknowledgments

This work benefited from discussions with many people including Pramod Joisha and Prith Banerjee. Comments from the anonymous reviewers on an earlier version helped improve the paper.

[8] We are also exploring weaker semantic guarantees that may enable this to be done statically. It is not yet clear whether that is a practical alternative.

References

[1] Memcached: a distributed memory object caching system. At http://memcached.org.

[2] Memcached: what is this thing? At http://code.google.com/p/memcached/wiki/NewOverview.

[3] 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.

[4] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H.-J. Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming. Technical Report HPL-2012-236, HP Labs, 2012.

[5] 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.

[6] 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).

[7] 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.

[8] 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.

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

[10] 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.

[11] 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).

[12] 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.

[13] 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).

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

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

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

[17] 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.

[18] 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.

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