Persistence Programming Models for Non-Volatile Memory

Hans-J. Boehm, Dhruva R. Chakrabarti

HP Laboratories
HPL-2015-59

Keyword(s):

non-volatile memory; locks; transactions; consistency; semantics

Abstract:

It is expected that DRAM memory will be augmented, and eventually replaced, by one of several up-and-coming memory technologies, all of which are non-volatile, in that they retain their contents without power. This allows primary memory to be used as a fast disk replacement. It also enables more aggressive programming models that directly leverage persistence of primary memory. However, it is challenging to maintain consistency of memory in such an environment. There is no consensus on the right programming model for doing so, and subtle differences can have large, and sometimes surprising, effects on the implementation and its performance. The existing literature describes several programming systems that provide selective persistence for user data structures. We more carefully and precisely describe the semantics of those systems, and thus the associated programming rules. We expose subtle and generally ignored trade-offs of programming generality vs implementation difficulty, as well as additional interesting points in the design space.

External Posting Date: August 6, 2015 [Fulltext] Approved for External Publication
Internal Posting Date: August 6, 2015 [Fulltext]

Persistence Programming Models for Non-Volatile Memory

Hans-J. Boehm [*]   Dhruva R. Chakrabarti
Hewlett-Packard Laboratories
{hboehm@google.com, dhruva.chakrabarti@hp.com}

Abstract

It is expected that DRAM memory will be augmented, and eventually replaced, by one of several up-and-coming memory technologies, all of which are non-volatile, in that they retain their contents without power. This allows primary memory to be used as a fast disk replacement. It also enables more aggressive programming models that directly leverage persistence of primary memory. However, it is challenging to maintain consistency of memory in such an environment. There is no consensus on the right programming model for doing so, and subtle differences can have large, and sometimes surprising, effects on the implementation and its performance.

The existing literature describes several programming systems that provide selective persistence for user data structures. We more carefully and precisely describe the semantics of those systems, and thus the associated programming rules. We expose subtle and generally ignored trade-offs of programming generality vs implementation difficulty, as well as additional interesting points in the design space.

Categories and Subject Descriptors D.3.3 Programming Languages [*Language Constructs and Features*]: Concurrent Programming Structures

General Terms Languages, Design, Reliability, Performance

Keywords non-volatile memory, locks, transactions, consistency, semantics

1. Introduction

It is widely believed [1] that conventional DRAM primary memory technology will not continue to scale, and is thus likely to be replaced over the next 5 or 10 years. At the same time. a number of new memory (`NVRAM`) technologies, such as memristors [33] and phase change memory (PCM) [24], are being developed. These provide an interesting twist to programming, since they are non-volatile. They allow the CPU to store data that will persist across power failures directly at `DRAM`-like speed through ordinary CPU store instructions.

Current applications maintain their data structures in DRAM, and periodically copy important data, often in a different format to disk, in order to ensure that it persists beyond the end of the process. A system based on `NVRAM` potentially eliminates that need by allowing in memory data structures to be persistent. 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.

Several fundamentally different approaches to handling failures in such a scenario have been proposed. All of them are complicated by the fact that only primary memory, and not CPU caches, are likely to be nonvolatile; thus it is hard to preserve the entire system state.

One approach, suggested in [29] is to restart the whole system in exactly the state before it failed, and to keep the whole process transparent to software. This approach is clearly worthy of further investigation, but it currently imposes two challenges:

- There is no actual hardware that makes this possible, and building such hardware is extremely challenging. It requires that all register and cache state be saved in a form suitable for restart. This is challenging for CPU registers and cache, and seems borderline intractable for state held in peripheral devices.
- It can deal with a relatively limited set of failure scenarios, like power failures. There is no clear way of coping with any kind of software failure, for example.

Here we instead follow [7, 10, 35], and look at restarting an application with a subset of its data structures explicitly identified as persistent by the programmer. This results in other challenges that we address here:

- When the program is restarted, data structures must be in a consistent state, i.e. crashes half-way through an update must be hidden.
- As mentioned above, some of the program state at the time of the crash may still have been in the CPU cache and will be lost. Thus we need to ensure that "important" program state reaches `NVRAM` soon enough to allow us to recover.

Many of the consistency issues encountered in this context are reminiscent of those encountered for shared memory communication between threads. And indeed we show that we can leverage synchronization constructs already found in most existing applications to infer most of the required consistency properties for `NVRAM`.

Most prior work in this area [10, 32, 35] has extended transactional memory facilities [23, 34] to provide a transactional programming construct that not only ensures isolation, i.e. that other threads do not see incomplete transactions, but also guarantees atomicity in the event of failures, i.e. transactions appear to have completely executed or not at all when the program is restarted after a failure.

[*] Current affiliation: Google. Most of this work was done while both authors were at HP Labs.

```
if (persistent_data_exists()) {
  restart_from_persistent_data();
} else {
  initialize_persistent_data();
}
use_persistent_data();
```

Figure 1. Structure of an NVRAM Program

We instead follow our approach from [7] and take a lock-based view, for several reasons:

- A lock-based model is strictly more general. From our perspective, which is concerned only with correctness and not performance, critical sections acquiring a single global lock are a good model of transactions.[1]
- It is unlikely that transactional memory will completely replace locks. There are reasons to expect transactional memory to become much more popular[13, 18]. But we expect future systems to still contain some locks. Notably, there is a non-trivial effort required to convert lock-based programs to ones based on TM [4], and we expect such conversions to be gradual. Additionally, some constructs, such as condition wait, arguably do not lend themselves well to the TM paradigm (cf. [19] but also [26]). We expect the relatively small amounts of code that need to directly use condition variables to continue to use locks. The draft specification of TM constructs for C++ [34] requires the co-existence of locks with transactions.
- As we pointed out in [7], a lock-based approach, by allowing thread communication within critical sections, raises issues hidden by the transactional memory approach.

This paper explores several different possible semantics for lock-based programs in the context of NVRAM all based on this general approach. We sketch corresponding implementation variants. In the process we highlight the simplifications that would be enabled if we restricted ourselves to transactions.

2. Setting and Core Issues

Applications will normally be structured as in Figure 1, though we will simplify this in the more precise treatment in Section 4.3. For this paper, we will assume that there are persistent variables, such as persistent_data here, that are initially set to zero, but will have their old value if the program is being restarted.[2] The programmer adds restart code that checks the state of these persistent variables, and either reuses the persistent data, or initializes it if there was none. Our goal is to ensure that if the programmer finds an existing data structure, then it is sufficiently consistent for use.

We assume a fail-stop or crash-recovery model. Persistent variables survive a tolerated failure,[3] other program state does not.

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. For many applications, this is naturally true. If not, the programmer can add "critical sections", using locks acquired only by a single thread, to explicitly delimit other regions during which data structures are inconsistent. This establishes our assumption without adding contention between threads.[4]

Since we assume that data structures may be inconsistent while any lock is held, we must effectively guarantee, as in [7], that "outermost critical sections" are failure atomic, i.e. intermediate states within an outermost critical section are never visible to restart code. Note that

```
lock(l1);
x = a;
lock(l2);
unlock(l1);
y = b;
unlock(l2);
```

has a single outermost critical section; an outermost critical section may be delimited by lock and unlock on different locks.

Using locks to ensure atomicity of persistent updates gives rise to situations not encountered in a purely transactional setting [7]. Consider the program in figure 2 with 2 threads. Each thread executes a single outermost critical section. Note that at step 8, thread 1 has completed its outermost critical section, but there is no way to advance to a consistent state unless thread 2 also completes steps 8 and 9. In the event of a crash at step 8, even if all of Thread 1's updates have reached persistent memory, implementations will generally have to undo Thread 1's *completed* critical section [7]. This preserves causality and ensures that while the unit of durability is an outermost critical section, any happens-before dependences between such units are maintained in NVRAM.

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

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

Updates to persistent memory outside critical sections or non-CS code (i.e. when no locks are held) may have to be handled in a special manner since restoring a consistent state may require undoing updates in non-CS code. Restoring a consistent state either requires

- Maintaining a single undo log and backing up to an actually encountered state in which no thread held a lock.
- Tracking happens-before between thread-specific undo logs.

The former adds contention and may require extra operations to be reverted, so we focus on the latter.

3. Contributions

NVRAM enables a variety of new and different programming models. Even if we restrict our attention, as we do, to those solutions

[1] This model was used in [18], and then refined slightly quite late in the process. We ignore explicit transaction aborts, whose utility is controversial [19].

[2] Our implementation supports a slightly different model in which persistent data is allocated in named persistent regions containing a root pointer that can be used to access the data, as in [7]. This is orthogonal to the issues discussed here.

[3] 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.

[4] Implementations may wish to add additional syntax for this case.

ri = x;	Load a global variable
x = ri x = c	Store into a global variable
`lock l`j	Acquire lock `l`j
`unlock l`j	Release lock `l`j

Figure 3. Simple statements in base language

that persist only a user-specified subset of program data and rely on lock- or transaction-like update mechanisms, and if we ignore a number of interesting issues related to, for example, combining pointer-containing persistent data regions occupying the same address space in a single process, as we also do, some profound tradeoffs remain.

Prior work in this area has selected a point in that space and explored an implementation on that space, with only nominal attention to semantic implications. Most other work (cf. [10, 35]) assumes a restrictive database-like setting of transactions (for our purposes a single lock) and restricting persistent memory updates to within those transactions. This model prohibits idioms that are common in existing lock-based or transactional-memory-based programs, such as first building a data structure privately, unprotected by transactions or locks, and then “publishing” it on completion by setting a flag or pointing to it. Quite recently, [7] developed a much more general model, supporting both multiple locks, and updates to persistent memory outside of critical sections, at the expense of added implementation cost and complexity.

Our contribution consists of precisely defining the semantics and corresponding programming rules for these models in a single framework, and analyzing at a foundational level, the implementation impact of those choices. In the process we explore some potentially interesting intermediate models that have not been previously studied.

4. A minimal language

Consider a tiny programming language subset, similar to that used in [5], in which a program consists of a collection of code sequences, each denoting the actions performed by a particular thread. We denote local variables (or **r**egisters) with ri, locks with names lj, and global variables with names pi and ti. Informally, variables pi are **p**ersistent, and their contents are preserved across application restarts. Variables ti are **t**ransient, and reset on restart.

We assume that variables take integer values. We assume that the sets of local variables ri appearing in the code sequence for each thread are disjoint. We allow the statements S shown in Figure 3 in each code sequence. Here c refers to an integer constant.

By restricting ourselves to loop- and recursion-free code, we do not need to distinguish programs and traces, simplifying the discussion. We avoid relying on this oversimplification in any substantive ways.

We assume that only the final values of transient variables are observable at program termination. This is yet another assumption made only for convenience: It significantly simplifies the later presentation, without hiding essential issues. It is again not needed in practice.

The grammar for a code sequence is given in figure 4. Here x represents either a persistent variable pi or a transient variable ti. The conditional tests for a nonzero value of ri. A program consists of a set of statement sequences corresponding to its threads.

We assume that a given local (register) variable ri is mentioned in the code sequence for at most one thread.

$$
\begin{aligned}
SS ::= {} & \epsilon \\
& \mid S;\ SS \\
& \mid \texttt{if (r}i\texttt{)}\ \{\,SS\,\}\ \texttt{else}\ \{\,SS\,\};\ \ SS
\end{aligned}
$$

Figure 4. Code sequences

4.1 Sequential failure-free semantics

We can define sequential, failure-free semantics of a code sequence as follows:

Code sequences update a state consisting of two components:

- The variable state V maps local (register) and global (persistent and transient) variables to integer values. (We distinguish register and global transient variables only in that the former are mentioned by a single thread. We distinguish persistent and transient variables later.)
- The lock state L maps locks to integers informally corresponding to the number of times each (reentrant) lock is held.

Statements and code sequences define a map from a state, i.e. an element of $V \times L$ to $V \times L \cup \{abort\}$.

Assignment statements copy or set the appropriate state component. A `lock l`i statement increments the count associated with the lock li. An `unlock l`i statements decrements the count associated with li, or *aborts* if the count was already zero.

The mapping for a statement sequence is the composition of that for the initial statement with that for the remaining statement sequence, except that it produces *abort* if the initial statement aborts. The code sequence `if (r`i`) then` *SS1* `else` *SS2*; *SS3* is equivalent to *SS1*; *SS3* if $R($r$i)$ is nonzero, and *SS2*; *SS3* otherwise.

A single threaded program P allows a final state s_f for an initial state s_0 if the mapping for its statement sequence does not abort, and produces the final state s_f.

Throughout this paper we will assume that no locks are held in initial states, i.e. $s_0.L(l) = 0$ for all l.

4.2 Concurrent failure-free semantics

Most mainstream languages [6, 27] assign semantics to this language essentially as follows.[5]

An execution consists of a triple (X, W, S), where

1. X is a set of pairs (s, i) denoting program actions, where s is a statement in the program or a conditional and i is an integer. Informally, the presence of s in X indicates s was executed, and assigned the corresponding value i, or resulted in the final lock count i. We sometimes write $X(s)$ to denote the value assigned by s.[6] If a conditional c is in X, the condition was executed, and $x(c)$ indicates the value of c and hence which branch was executed.
2. The write-seen-by function W maps each load or conditional (i.e. each read) l to the corresponding store (write) of the same variable, or to a special `init` action indicating an implied assignment of its initial value. Intuitively $W(l)$ is the store seen by l.
3. A *synchronization order* S is a total order on the lock acquisitions and releases in X.

We say that statement or conditional S_1 is *sequenced before* statement statement or conditional S_2 if either

[5] The Java situation is actually somewhat more complicated, in that it tries, somewhat unsuccessfully, to also assign semantics to programs with data races.

[6] If our programs had loops, the first element of each pair would be a specific program action, not a statement.

- S_1 is immediately followed by S_2 in the statement sequence for a thread, or
- S_1 is a conditional and S_2 is the first statement or conditional in either the then or the else clause, or
- S_1 is sequenced before S_m and S_m is sequenced before S_2.

We say that an action is sequenced before another action if that relation applies to the statements they contain.

Given a synchronization order S, we say that an `unlock l`j statement *synchronizes with* a `lock l`j statement referring to the same lock, if the `lock` statement follows the `unlock` statement in S. In addition, the special `init` action synchronizes with the first action (in sequenced before order) of every thread.

We say that a statement a *happens before* b if

- a is sequenced before b, or
- a synchronizes with b, or
- There exists a statement c such that a happens before c and c happens before b.

We say that a (potentially incomplete) execution is valid with respect to an initial state if:

- The synchronization order is consistent with the behavior of reentrant locks: For every `lock` operation, either the last preceding (in S) lock operation on the same lock was performed by the same thread, or $X(l)$ is 0, i.e. the lock is not held. For every unlock operation u, $X(u) > 0$, and the last lock operation on the same lock was performed by the same thread.
- The happens before relation is irreflexive, i.e. there are no cycles that result in a happens before a.
- $W(l)$ happens before l (or is *init*), i.e. every load "sees" a store that happens before it.[7]
- It is never the case that $W(l) = s$ and there is another store s' to the same variable that "happens between" them, i.e. such that s happens before s' and s' happens before l.
- If $W(l) = w$, then $X(l) = X(w)$, i.e. the value observed by a load or conditional is the stored value. If $W(l) = init$, then $X(l)$ is the initial value of the variable read by l.
- If statement or conditional S_2 was executed, and S_1 is sequenced before S_2, then so was S_1.
- The executed statements are consistent with values loaded by conditionals: If c is a conditional in X and $X(c)$ is nonzero, then, and only then, the first statement in the `then` clause is in X. Otherwise, and only then, is the first statement in the `else` clause in X.
- A `lock` or `unlock` operation appears in S iff it appears in X.

We say that an execution is complete, if also:

- If the first of two consecutive (non-conditional) statements is in X, then so is the second. If a conditional is in X, then so is the first statement of the then- or else-clause, and so is the first statement in the subsequent statement sequence (if any).

We say that a valid execution has a *data race* if there are two store statements to the same variable, or a load and a store statement accessing the same variable, that are not ordered by the happens before relation, i.e. that are not prevented from executing concurrently.

[7] Loads never see "racing" stores. This is realistic since we do not assign meaningful semantics to programs with data races. It is also essential; see [6] for details.

```
Main program:
    initialize_persistent_data();
    use_persistent_data();

Restart code:
    restart_from_persistent_data();
    use_persistent_data();
```

Figure 5. Simplified structure of an NVRAM Program

The semantics of a particular program define the allowable final values of variables, given a set of initial values. This is defined as follows:

- If there exists an execution with a data race, we consider the program to be erroneous.
- Otherwise a multithreaded program is said to allow a final state s_f if there is an execution in which the value of each variable v in s_f is $X(a)$, where a is last (in happens-before order) assignment to v. (If this last preceding store were not unique, there would be a data race.)[8] It is equal to the initial value of the variable if there is no such assignment.

We say that (X', W', S') is a prefix of (X, W, S) if both are valid executions for the same initial state, $X' \subseteq X$, W' is W restricted to X' and S' is S restricted to X'.

4.3 Desired failure semantics

Informally, we expect to be able to restart a program after a failure, and for tolerated failures, the new execution of the program should find the heap in a consistent state, i.e. in a state that could have occurred in the original execution, and in which no locks were held.

We will continue to simplify the discussion by assuming that each statement is executed at most once. To this end, we will assume a single failure, and that a *restartable program pair*, or just *restartable program* consists of two programs as defined before[9]: The *main program* and the *restart code*. Rather than having the code test for preexisting persistent data as in 1, we completely separate out the recovery code and replicate the actual body of the program accesses the persistent data, as in Figure 5. (Both figures show a simplified single-threaded case.)

We define a state s to be *consistent* with respect to a main program m and a given input state s_0, if:

- A (partial) execution of m from s_0 allows s, and
- there are no locks held in s, i.e. $\forall i \, . \, s.L(\texttt{l}i) = 0$.

We define the semantics of a restartable program pair as follows. An execution of a restartable program (m, r) from initial state s_0 to final state s_f consists of either an execution of m with final state s (the failure-free case) or a pair of executions (the failure case)[10]:

1. An execution of m from s_0 to a failure state $s_\dagger$, consistent with respect to m and s_0, and
2. An execution of r from state $clean(s_\dagger)$ to s_f

where $clean(s)$ is defined to be s with all transient and register variables reset to zero.

[8] In fact, $W(l)$ is uniquely determined by the synchronization order. For non-erroneous executions, our semantics are equivalent to sequential consistency [22].

[9] Each of these in turn consists of statement sequences corresponding to each thread.

[10] The fully general repeated failure case would require a sequence.

We say that (m, r) on s_0 allows final state s_f if there is an execution of (m, r) from s_0 to s_f.

Our goal is to provide implementation strategies, possibly coupled with program restrictions, that ensure that the observable program behavior, i.e. final values of transient variables, correspond to an allowed final state.

For example, if our main program is `p0 = 1000; p1 = 1000;`, and our initial state maps all variables to zero, these semantics preclude starting the restart code in a state in which p1 = 1000 and p0 = 0. As a result of compiler and hardware optimizations, this is typically nontrivial to enforce. In fact, programming languages like C and C++ currently carefully avoid even specifying that the individual assignments here are indivisible; they may be performed e.g. a byte at a time. Thus, with a naive implementation and/or in the absence of program restrictions, restart code may run in a context in which assignments were only partially completed when they were interrupted by the crash, something that we clearly also wish to preclude.

These semantics are clearly desirable, in that they allow the restart code to only see consistent failure states, and hide effects of caching and the like. It is less clear that they are reasonably implementable, or required for reasonable programs. Understanding those issues requires an (abstract) look at possible implementations. We begin by presenting an abstract machine to use as a translation target.

5. The target language

Implementations of our minimal language consist of a translation from minimal language programs to augmented programs in a slightly extended *target language*, together with an algorithm for reconstructing the program state after a crash of such an augmented program. This target language exposes the addition of three additional state components:

- An undo log U, which, for each thread contains a sequence of $(statement, variable\_or\_lock, old\_value)$ triples.[11]
- A synchronization history H, which maps each lock to the last statement to acquire or release that lock, and each `lock` or `unlock` statement to the previous statement to update the lock.
- A subset P of the persistent variables that have reached `NVRAM` and will thus be visible after a failure.

We also explicitly include the program counters C, one for each thread, in the state. C maps the thread index to the next statement or conditional to be executed by each thread, or to a special *done* value.

Thus our state is now a 6-tuple (C, V, L, U, H, P), comprising program counters, the variable state, the lock state, the undo log, the synchronization history, and the persistence state P.

When talking about running a target program in a given state, we will typically specify only V, and then only the bindings to persistent variables. We implicitly assume that initially all locks and other variables are bound to zero, U, and H are empty, P contains all persistent variables, and C refers to the initial statement in each statement sequence.

In reality, an implementation would include mechanisms to occasionally prune U and H, thus limiting their size, as is discussed in [7]. We'll omit that here, since it is orthogonal to our goal of understanding the semantic issues.

[11] This clearly biases our implementation strategies towards those based on an undo log. Although those are not the only viable strategies, we believe they suffice to illustrate the trade-offs between implementation and semantics.

`flush`	Flush persistent variables into `NVRAM`
`log x`i	Append (a way to identify the) immediately following statement, `x`i and its current value to log

Figure 6. Added statements in target language

Our abstract target language is the same as our minimal language, except that we add the statements in Figure 6.

We (somewhat informally) specify the operational semantics of the target language as follows. Each execution step chooses a thread t to execute next. If its next instruction, i.e. C_t is `lock l`i, then $L(\texttt{l}i)$ must be zero or $H(\texttt{l}i)$ must be in the same thread. An execution step for statement s in thread t is then performed as follows:

- If s is an assignment, update V as in the sequential semantics. If the assigned variable is in P, remove it. Update C_t to refer to the next statement, or *done* if there is none.
- If s is a `lock` or `unlock` operation on `l`i, acquire or release the lock and update L as in the previous semantics. After acquiring the lock, or before releasing it, set $H(s)$ to $H(\texttt{l}i)$. Then set $H(\texttt{l}i)$ to s. Update C_t as above.
- If s is `flush`, add all persistent variables in V to P. Update C_t as above.
- If s is `log p`i, and the immediately following statement in t is s', append $(s', i, V(\texttt{p}i))$ to $U(t)$. If s is `log l`i, instead append $(s', i, L(\texttt{l}i))$ to $U(t)$. Update C_t as above.
- If s is a conditional `if(r`i`) ...`, set $C(t)$ to the first statement in the `then` or `else` clause, depending on $V(\texttt{r}i)$
- At any point, without modifying C, optionally add one or more persistent variables to P. This reflects the fact that a processor cache may choose to evict variables from the cache without an explicit `flush`.

For a data-race-free and abort-free program in the base language, this process produces a final state consistent with the semantics in Section 4.2. It is easy to derive X and W for the corresponding Section 4.2 semantics from the above program interpretation. The synchronization order S is encoded in the synchronization history H.

In the event of a failure at state s, a target language program produces a crash state containing U and an incomplete version of V mapping all persistent variables in P to their values in s. Other persistent variables modified by the computation since the last `flush` may contain any values whatsoever.

For convenience we will assume that the synchronization history H is also available; in practice it might be reconstructed from an augmented version of the undo log.

The first step of recovering after a failure will be to reconstruct L from U by counting lock and unlock operations in U.

It is the job of the implementation to add `flush` and `log` statements to the main program, and to transform the crash state to a consistent state for the restart code. We describe several such implementation strategies below.

5.1 Our target language vs. reality

Note that this target language reflects an optimistic view of actual hardware capabilities. Most existing hardware and operating systems at best allow user-level programs to at most flush individual cache lines to the underlying memory, be it `NVRAM` or `DRAM` [15]. In such environments, our `flush` statement could be implemented with a software "write barrier" that maintains a set of possibly dirty cache lines, so the the `flush` implementation knows which lines to flush. Alternatively, most architectures provide a "flush entire cache" operation that could possibly be exposed to user programs via a new system call. A third option is to map the affected memory range as write-through or uncacheable, in which case we expect a relatively light-weight fence instruction to suffice as the `flush` implementation. We expect the implementation options to improve as `NVRAM` gains more practical significance.

Note that our `flush` statement is entirely different from, and typically more expensive than, the memory fences often used to enforce memory visibility among multiple threads. Memory fences typically only ensure that memory operations are visible to the cache, so that cache coherency mechanisms ensure a consistent view. Our `flush` operation typically must ensure that stores have reached actual physical non-volatile memory.

In order to ensure that U survives a crash, every `log` operation must also ensure that log entries have reached `NVRAM` before continuing, or at least before the corresponding assignment reaches `NVRAM`. This can be done directly with cache line flushes, or probably with other architecture-specific mechanisms, such as x86 "non-temporal" stores.[35] Various optimizations are possible and useful. Consecutive writes to the log are combinable, and repeated log entries for the same variable can often be elided.

Current memory controllers add other challenges, but we expect those to disappear with `NVRAM` support, such as Intel's newly added `PCOMMIT` instruction [16].

6. Implementation Strategy 1: Log all updates to persistent memory locations

Our first implementation strategy roughly models that in [7] and reconstructs a consistent state from the crash log by undoing all updates in reverse-happens-before order until a consistent state is reached such that the consistent state corresponds to a point that is just after an `unlock` operation.

We "compile" the main program of a restartable program pair by inserting a few statements into the original:

- Each assignment to a persistent global variable pi in the original program is preceded by a `log` pi statement.
- Each `lock` statement is preceded by `log` li.
- Each `unlock` statement s is preceded by `flush; log` li.

The combination of the undo log U and synchronization history H allows us to compute a happened-before relationship between log entries:

- If a and b are both entries in U_t, i.e. they reflect actions by the same thread, and a was added to the log before b, then a happened before b.
- If a is an `unlock` operation, and b is a `lock` operation, and $H(b) = a$, i.e. b acquired the lock immediately after a released it, then a happened before b.
- If a happened before b and b happened before c, then a happens before c.

This is essentially the same as the happens before relation from Section 4.2, but defined on log entries.

Given an undo log U and a synchronization history H, we say that an entry u in U is undo-eligible if either

- (directly eligible) the prefix of U_t ending with u contains more `lock` than `unlock` actions (i.e. t still held a lock after u), and no longer prefix contains an equal number of `lock` and `unlock` actions, or
- (indirectly eligible) u happened after another undo-eligible `unlock` action in U.

Given (V, U, H) generated when main program m started from s_0 failed, we can use the following algorithm to construct a consistent state from the crash state:

ALGORITHM 1. *We reconstruct L from U by subtracting the number of unlock operations for any given lock from the number of lock operations for the same lock.*[12] *We start with the variable state V at the time of the failure, but reset all transient and register variables to zero.*

We then repeatedly update this lock state L, together with the partial variable state V and undo log U, from the crash state, based on the synchronization history H, as long as one of the following applies to any thread t:

- *If the last entry of a thread t's undo log refers to an assignment to* pi*, remove the undo log entry, and replace the value for* pi *in V with the one saved in U_t.*
- *If the last entry of a thread t's undo log refers to a* `lock` *statement, remove the corresponding entry and decrement the corresponding lock value in L.*
- *If the last entry u of a thread t's undo log refers to an* `unlock` *statement, the* `unlock` *entry is undo-eligible, and there is no other* `lock` *entry l in U, such that $H(l) = u$, then remove the entry for u from U_t and increment the corresponding lock value in L. The last condition ensures that this critical section does not happen before another one in a different thread that should be undone first. We always undo actions in an order consistent with reverse happens-before order; a critical section cannot be undone before those that might have observed its effects.*

LEMMA 1. *If the execution of m on s_0 was data-race-free, then the state produced after repeated application of the above 3 undo steps depends only on the resulting U, i.e. which unwind steps were applied, not the order in which they were applied.*

Proof. State updates are constrained to be undone in reverse happened-before order. For different orders to produce different outcomes, there would have to be two updates to the same lock or persistent variable that are not ordered by happens-before. For operations on the same lock, that's impossible, since they are always ordered. Since there are no data races, there also cannot be unordered assignments to the same persistent variable. ●

LEMMA 2. *For every U generated during the undo process, there is a partial execution E' of m from s_0 resulting in that U, and a synchronization history H', such that for every statement S for which $H'(S)$ is defined, $H'(S) = H(S)$, i.e. lock acquisitions and releases were performed in the same order as in the actual execution. Furthermore, the execution E' results in lock and variable states L' and V' such that*

- *L' is identical to that generated during the unwind process.*
- *$V($*p$i)$ *generated during unwinding is either equal to its value at the time of the failure (if no undo log entries for that variable were processed), or to $V'($*p$i)$ *(if undo actions for $V($*p$i)$ *were processed).*

Proof. Take E' to be the original execution restricted to any log actions that are reflected in U, together with the immediately following statement corresponding to that log action, and any actions that are sequenced before any of those actions. We then show, by induction on the number of performed unwind actions, that E' defined this way always preserves this property. As each undo action performed during the process, the state is restored to correspond to

[12] This gives us an L consistent with U. It will include the effect of lock or unlock operations that were logged, but not actually executed before the failure.

a new E' execution that terminated just after the previous action logged by that thread, but is otherwise identical. ●

LEMMA 3. *Algorithm 1 always terminates in a consistent state in which no locks are held, i.e. $L(\mathtt{l}i)$ is zero, for all i.*

Proof. Algorithm 1 can clearly not terminate while any thread's undo log ends in something other than an `unlock` entry. Since happened-before is a strict partial order consistent with the total order in which entries were added, there must always be a thread undo log whose last entry e did not happen before any other entries. Once algorithm 1 terminates, the last remaining undo log entry of every thread must be an `unlock entry`, and the last entry corresponding to every such happens-before-maximal thread must be undo-ineligible. If any undo-eligible `unlock` remained in U it would have happened-before one of these undo-ineligible maximal entries, leading to a contradiction. Thus the algorithm terminates when all U_t end with an undo-ineligible `unlock` entry. It follows from the definition of undo-eligibility that each U_t contains an equal number of `lock` and `unlock` entries. ●

We complete the implementation by taking the crash state, reverted by Algorithm 1, $s_\dagger$, constructing a new state from the resulting V and (all-zero) L, and using that as the start state for the restart code.

THEOREM 1. *For data-race-free programs, the above implementation achieves our desired failure semantics, i.e. it produces only valid executions corresponding to a consistent failure state.*

Proof. By Lemma 2 the state $s_{\dagger'}$ reconstructed by Algorithm 1 corresponds to a partial execution of of m on s_0. By Lemma 3 there are no locks held at the end of this partial execution. Since the undo process terminates just after each thread has performed an `unlock` operation (or has just started), and every `unlock` operation is preceded by a `flush` operation, every update to a persistent variable sequenced before this final unlock action would have been correctly reflected in the state at failure. If it was not subsequently modified in its original execution, it would have had, and retained, its correct value. If it was subsequently modified, the undo action corresponding to that modification would have restored its correct value. Hence all persistent variables in $s'_\dagger$ correspond to the partial execution from Lemma 2, and all other variables are zero. Thus we've achieved our desired failure semantics: It appears the recovery action was restarted in a consistent persistent state of the original execution. ●

This algorithm achieves our goal. It also has the practical advantage that it preserves the validity of common compiler optimizations: Since we always restore the state at the time of an `unlock` operation, transformations such as sequentially correct reorderings of assignments are never visible to the restart code; only transformations that reorder with respect to `unlock` operations would be visible.

However, the approach in this section still has several important shortcomings:

- It requires that we log persistent operations outside of critical sections. Ideally we would like to be able to run existing unmodified code outside of critical sections without slowing it down. The need to eventually flush stored values out of processor caches to `NVRAM` may already make that challenging, but the need to maintain logs further removes us from this goal.
- We must undo long sequences of synchronization-free code outside critical sections completely, even if they do not depend on (happen after) critical sections that need to be undone. A thread that performs no synchronization will have to be undone to its start state. As a practical matter, we may have to keep very large logs before we can be sure that they are no longer needed for undo operations.

In the next three sections, we explore ways to lessen these implementation constraints at the expense of additional constraints on the programmer.

7. Implementation Strategy 2: Restart-race-freedom

This strategy models the requirement used in [7] where code executed outside a critical section should not arbitrarily update persistent locations. Since no assumption can be made about atomicity of such updates, such updates may appear to the restart code to have been partially completed. Indeed any implicit atomicity assumptions often don't make sense in higher level code: If a templatized C++ function assigns to a variable of type `T`, where `T` is a template parameter, it has no way to tell whether `T` is represented by a single byte, which the hardware could atomically update, or a megabyte data structure, which it could not. This is similar to disallowing data races in multithreaded code [6, 14] or disallowing a signal handler from accessing data that might be in the midst of an update [17].

Algorithm 1 avoids any such restrictions by reverting more actions than usually necessary. By always reverting all actions till the last `unlock` action, it effectively ensures that statement sequences containing no `lock` or `unlock` statements are also treated as atomic. The downside is that we may have to revert long sequences of actions unnecessarily.

Similarly to [7], we say that an execution is *restart-race-free* if, an update u of a persistent variable $\mathtt{p}i$, performed by the main program outside a critical section, can only be read by restart code if that restart code also observes an update u' performed in a critical section, such that u happens before u'. This means that if $\mathtt{p}i$ is accessed by the restart code at all, the restart code must first read a variable set in the later critical section to check that this is safe.

A program is restart-race-free on a given input if all executions are restart-race-free.

Note that this continues to allow stores to persistent variables outside of critical sections. The following common idiom is restart-race-free, provided p1 is initially zero:

```
main program:
    p0 = 42;
    lock l1;
    p1 = 1;
    unlock l1;

restart code:
    r1 = p1;
    if (r1) then
        r2 = p0;
```

The persistent variable *p0* is not accessed by the restart code unless the main program successfully executes the `l1` critical section.[13]

If we require restart-race-freedom, the following variant of Algorithm 1 suffices, with the same code instrumentation as before:

ALGORITHM 2. *Identical to Algorithm 1, except that assignments are reverted only if they are undo-eligible. We no longer revert assignments after the final* `unlock` *action in each thread.*

THEOREM 2. *For data-race-free and restart-race-free programs, Algorithm 2 achieves our desired failure semantics.*

[13] In real code, it is likely that the access of `p1` from restart code would also be inside an `l1` critical section, to avoid conventional data races. Purely from a restart race perspective, that's not necessary.

```
x, y are persistent and initially x=y=0
    t1                  t2
                     1: lock(lj)
                     2: lock(li)
                     3: ...
                     4: unlock(li)
5: lock li
                     8: ...
                     9: unlock(lj)
```

Figure 7. Lock of `li` causes `lj` to be added to $UL_{\texttt{t1}}$

Proof. The only difference from the preceding algorithm is that we may fail to undo assignments to persistent variables that are not followed by a critical section in the reconstructed execution to a consistent state. Since the program is restart-race-free, these stores cannot be read by the restart code. Thus the execution of the restart code and the value of transient variables are unaffected. We've carefully defined the observable behavior of the program execution to exclude the persistent variables we failed to undo; thus observable behavior is unchanged.[14] •

7.1 Logging elision

Insisting on restart-race-freedom gives us another important benefit: If a program is restart-race-free, updates to persistent memory outside critical sections do not have to be logged unless there is a need to undo those updates after a failure. The condition under which such logging can be elided is discussed in [7]. Here we present an algorithm to realize log elision during program execution.

The following data structures are maintained:

1. A global lock release history H_{rel} which maps each lock to the total number of releases of that lock. We only count "outermost" releases of a reentrant lock, i.e. releases of Li such that afterwards the total number of acquisitions and releases of Li by that thread are equal.
2. A thread-specific "undo locks" set, UL_t, is maintained by every thread. At any point of execution, this set captures the set of locks on which execution is currently conditioned; a thread's execution is no longer speculative when all such locks are released at least one more time. Each element of UL_t is a pair $(\texttt{l}i, c)$, where c is a lock release count corresponding to the lock li. A pair $(\texttt{l}i, c)$ is included in UL_t if li is currently held by t and li was acquired and released a total of c times before this last acquisition.

   As depicted in Figure 7, given a `lock` action A_{lock} for lock li by thread t_1, if there exists an unlock action A_{unlock} (executed by thread t_2 such that $t_1 \neq t_2$) such that A_{unlock} happens before A_{lock}, and if a lock lj is held by the thread t_2 at the time of execution of A_{unlock}, then a pair $(\texttt{l}j, cj)$ is added to the *UL* set corresponding to t_1, where cj = $H_{rel}(\texttt{l}j)$ at the point A_{unlock} is executed. t_1's execution is now speculative until lj is released another time.
3. A global history of undo locks H_{undo} which maps each lock li to the value of UL_t at the point of the last release of li, where t is the thread that executed the corresponding `unlock` statement. Effectively $H_{undo}(\texttt{l}i)$ gives the set of critical sections on which the last release of li depended at the time it was performed.

[14] In a more realistic setting, if we want to use some persistent memory locations as an output medium, and thus make them observable, we would still have to revert assignments to those specific "output" locations.

On an acquire of lock li by thread t, the following steps are executed:

- Using H_{rel}, obtain $c_{\texttt{l}i}$, the number of times li has been released.
- Replace UL_t by $UL_t \cup \{(\texttt{l}i, c_{\texttt{l}i})\} \cup H_{undo}(\texttt{l}i)$

On a release of lock li by thread t, the following steps are executed:

- Remove $(\texttt{l}i, c_{\texttt{l}i})$ from UL_t.
- Traverse UL_t and if for a given lock lj, its corresponding counter from $H_{rel}(\texttt{l}i)$ is greater than that found in UL_t, remove that element from UL_t. We know that there was an intervening release of that lock, and we can no longer be forced to undo as a result of that critical section.
- Add a mapping from li to the current value of UL_t in H_{undo}.
- Add a mapping from li to $c_{\texttt{l}i} + 1$ in H_{rel}, where $c_{\texttt{l}i}$ is remembered from the lock acquisition.

Given the above algorithm, the following holds: During an update outside a critical section, if for all elements in UL_t, the counter value for a lock from H_{rel} is greater than the corresponding value found in UL_t, the update does not depend on any data written within a critical section that can be undone. If so, the update does not have to be logged. Consequently, if the program crashes while a thread is in the midst of an unlogged update outside a critical section, that update will not be undone. But that does not matter since that updated location should not be accessed by restart code and no critical section that happened before the update can be undone.

Since log entries outside critical sections are elided whenever possible, this strategy overcomes the two shortcomings of strategy 1: possibly unnecessary logging of updates outside critical sections during program execution and possibly unnecessary undoing of such updates. However, strategy 2 still incurs some runtime overhead for log elision analysis. The next strategy explores a constraint on the programmer that enables log elision outside critical sections without any analysis.

8. Implementation Strategy 3: Log only in critical sections

Although the insistence on restart-race-freedom often allows us to optimize away logging at run-time, the need to instrument non-critical-section stores remains. This may be undesirable, both for performance and other reasons. For example in a traditional compilation setting it means that all library code that potentially touches persistent data needs to be recompiled. It also differs from, for example, the requirements of volatile-memory transactional memory systems.

The most common use cases for updating persistent memory outside of critical sections (or transactions) are similar to the example in the previous section, and actually *never* require undo logging in a critical section. The variable `p0` will only be read if the subsequent critical section also completes. If it does, there is no reason to undo the assignment. If the critical section fails to complete, there is no reason to log either, since `p0` will never be looked at. We can make the required condition precise as follows:

We say that in a main program execution E of a program (m, r) on s_0, a persistent variable pi is guarded by an `unlock` action u, if

- pi is only assigned to before u or inside a critical section, and
- restart code r, when started in a consistent state resulting from executing a prefix E' of E, only reads pi if E' includes u.

We say that a main program execution E of (m, r) on s_0 is *strongly restart-race-free* if every persistent variable assigned to outside a critical section in that execution is guarded by a corresponding unlock action. We say that a program (m, r) on s_0 is strongly restart-race-free if all its main program executions are.

In a strongly restart-race-free program, each persistent location pi written outside a critical section is "published" by a critical section c that makes it available to restart code. If the program fails before c is completed, pi is not read by restart code at all. And pi is not written by the main program outside of a critical section after u'.

The difference between this and our earlier definition of restart-race-freedom is that the publishing write, or more properly the critical section in which it occurs, depends only on the persistent variable pi, and hence must apply to all updates of pi. For plain restart-race-freedom, there can be a different "guard" critical section for each update.

This property is satisfied by the most common real-life use cases for non-critical-section persistent writes. The non-critical-section code builds up a persistent data structure S, which is followed by a critical section write w of a well-known location that or signals the availability of S. In real code, w most commonly writes a pointer to S. In our minimal language, we are restricted to writing a flag variable. In the event of a failure before w, S becomes garbage and can be reclaimed.

Strong restart-race-freedom precludes the following example, which uses a zero value for p1 to signal that p0 is currently being modified:

```
main program, thread 0:
    p0 = 42;
    lock l1; p1 = 1; unlock l1;
    ...
    lock l1; p1 = 0; unlock l1;
    p0 = 17;
    a:
    lock l1; p1 = 2; unlock l1;

restart code:
    r1 = p1;
    if (r1) then
        r2 = p0;
```

Consider a failure at a, such that there is a need to undo the middle critical section because the ellipsized code had observed the effects of an incomplete critical section in another thread. If we did not have a way to undo the second assignment to p0 we would end up unwinding to a state in which p0 is still 17, but p1 is 1. This is not a consistent state of the main program.

This code is not strongly restart-race-free because there is no single guard critical section for all assignments to p0.

This program models real code that temporarily removes an element from a persistent data structure, treats it as private data, since it now has exclusive access, and then adds it back to the persistent data structure. Such code is again restart-race-free, but not strongly restart-race-free.

THEOREM 3. *For a strongly restart-race-free program, Algorithm 1 yields a valid execution, even if we fail to insert* log *statements for persistent assignments outside of critical sections.*

Proof. The only difference is that we may fail to correctly restore a persistent variable pi that was modified outside a critical after the state $s_\dagger$ restored by Algorithm 1, but before the actual failure. If $s_\dagger$ reflects the execution of pi's guard, then there can be no such modification. If it does not, then pi may not be accessed by the recovery code, and hence does not impact observable behavior. ●A similar implementation strategy can also be applied to other programming restrictions that further restrict the amount of speculative state that may need to be undone after a failure. The next section revisits a fairly draconian restriction along these lines that has been pursued in other work.

9. Implementation Strategy 4: Undo only active critical sections

We restrict our implementation strategy to modify Algorithm 1 to only undo statements, separately for each thread t, until the number of lock acquisitions and releases in U_t matches. We thus undo only actions in incompletely executed critical sections.

THEOREM 4. *Theorem 1 holds for Strategy 4 for data-race-free and restart-race-free programs that never hold more than one (possibly reentrant) lock at a time.*

Proof. No action in another thread can happen after an undo-eligible unlock action. The unlock can only be directly undo-eligible because it is a release of a nested acquisition of a lock, and the thread still holds the lock. But that would have prevented another thread from acquiring it. Thus this case is equivalent to Algorithm 2, and the same result holds. ●

THEOREM 5. *Theorem 1 holds for Strategy 4 for data-race-free and restart-race-free programs that always release all simultaneously held locks at once.*

Proof. The argument is similar to before. This time there are no directly undo-eligible unlock actions, since we assume that there are no failures between unlock actions for simultaneously held locks. Thus no undo-eligible unlock action happens before anything else, and there are no indirectly undo-eligible actions either. ●

Note that since we do not flush non-critical-section updates we still need the restart-race-freedom assumption to ensure that relevant non-critical-section updates from before the last unwound critical section are visible. Otherwise the restart code could rely on unflushed persistent variables written after the last critical section. If we also disallow non-critical-section updates, as in [35] that need disappears.

Single-global-lock-atomicity transactional memory systems can be viewed as satisfying this restriction. Disjoint-lock-atomicity [18, 28] can be viewed as releasing all concurrently held locks at once, making it subject to Theorem 5.

10. Input and Output

In any of these strategies, batch input is easily modeled as providing the initial state of the computation, and batch output can be modeled as updating distinguished locations in NVRAM. However any form of interactive or embedded input or output is more difficult to accommodate. This is primarily because of the delayed commit that may result from lock nesting. Even if a critical section completes, it may have to be undone after a failure (see Algorithm 1), which in turn may result in undoing updates, if any, outside critical sections. The delayed commit point for an update in a lock-based program is reached when it can be guaranteed that the update will not be undone even in the event of a crash.

Transactions [18] have a similar issue with I/O. But there it suffices to defer I/O within a transaction to its end, i.e. when it commits. But with all of our models except for Strategy 4, there may be a need to defer I/O outside a critical section till its (delayed) commit point. This clearly is a constraint on the program.

To help understand where a programmer can place an I/O operation that cannot be deferred, we can expose a new interface wait_until_committed() that blocks until it can be ensured that

```
Thread 1                        Thread 2

lock l1
lock l3
unlock l3
                                lock l3
lock l2                         unlock l3
buffer_not_full.wait(l2)        wait_until_committed()
unlock l2                       empty_buffer()
unlock l1                       buffer_not_full.signal()
```

Figure 8. `wait_until_committed` and condition wait

any update at the program point containing this call cannot be undone. The implementation of this interface must ensure that all prior deferred I/O operations are executed before returning. This interface thus allows the program to send a message acknowledging that prerequisites have been persisted in `NVRAM` by calling `wait_until_committed` and then performing I/O conventionally.

Theoretically, `wait_until_committed` may acquire any lock whatsoever, and it can be thought of as acquiring all locks in the system, so some care is required to ensure deadlock freedom. A thread must not call `wait_until_committed` while holding a lock. This is sufficient to prevent purely lock-based deadlocks.

Simple uses of condition wait will also work. However, condition wait with nested locks may not. Consider Figure 8 and assume that the release of lock `l3` in Thread 1 synchronizes with the acquire of `l3` in Thread 2. While holding lock `l1`, Thread 1 acquires lock `l2` and tries to communicate with I/O Thread 2 through a buffer. I/O Thread 2 calls `wait_until_committed` before emptying the buffer and signaling Thread 1. But `wait_until_committed` will never return since it is waiting for the delayed commit point to be reached which in turn requires the outer critical section (protected by lock `l1`) in thread 1 to complete (because of the synchronizes-with relation induced by lock `l3`) which in turn requires thread 2 to signal Thread 1. Thus, we have a deadlock. A solution is to not call condition wait under an outer lock in anticipation that there may be another thread calling `wait_until_committed` . An outer lock can be used as long as care is taken to ensure that offending synchronizes-with relations (such as the one in Figure 8) do not exist.[15]

11. Empirical Comparison

As in [7], we used `DRAM` to simulate `NVRAM`. In Table 1, we present results comparing the 4 persistence strategies using a Linux x86 machine (Intel Xeon quad-core X5355 @ 2.66 GHz). All reported numbers are averages over 6 runs.

We picked some of the same applications that were used in [7]. MDB [9] is a B-tree based key/value store that has been made `NVRAM`-aware. `mtest` [7, 9] is a workload that belongs to the `MDB` test suite and performs insertion, traversal, and deletion over 3000 key/value pairs. We chose all the applications in the SPLASH2 [37] benchmark suite and adapted them. As in [7], we persist all non-stack-allocated locations in SPLASH2. This is likely to create a large number of accesses to persistent data, both inside and outside critical sections, and is a good way to stress-test our strategies. On the other hand, this methodology probably exposes tradeoffs that might not be exhibited by real applications, such as MDB/mtest.

As discussed in Section 5.1, cache line flushes tend to be expensive, perhaps unrealistically so for future `NVRAM` systems, and often

[15] Similar issues also appear to arise in other related scenarios. For example, [26] seems to inherently introduce blocking at the end of certain transactions, also creating a need for a non-obvious, at least to us, deadlock analysis.

	Statistics			Speedup vs s2 (%)	
Applications	#cs	#str	#out (%)	s3	s4
mtest	0.10M	179K	98	0.5	9.9
barnes	0.13M	586M	99	7.7	10.3
fmm	6.6K	87M	99	10.6	12.9
ocean	788	98M	99	10.5	10.8
radiosity	0.25M	18M	96	0.8	NC
raytrace	74K	18M	99	0.1	37.9
volrend	72K	391M	99	8.7	13.0
water-nsq	6K	45K	99	-38	13.3

Table 1. Comparison of implementation strategies 2-4. 4 user threads were used in every SPLASH2 application (K=10^3, M=10^6, NC = non-conforming)

hide the overheads of logging. Hence, for all results reported in this paper, we turned off cache flushing altogether. This helps us isolate the overheads and tradeoffs surrounding logging. Strategy 1 is implemented as discussed in Section 6. Strategy 2 (denoted by s2) uses log elision analysis described in Section 7. In programs where a large number of accesses are outside critical sections, a large reduction in logging overhead is possible with strategy 2. We implemented strategy 3 (denoted by s3) by inferring whether a store to persistent memory is outside a critical section and simply not logging that access, without performing any additional analysis. We implemented strategy 4 (denoted by s4) by simply not building the synchronization history H (Section 5). As discussed in Section 9, this is safe if locks do not nest during program execution.

Table 1 summarizes our findings including some program statistics. #cs denotes the total number of critical sections (both nested and non-nested). #str denotes the total number of stores to persistent memory encountered during execution. #out denotes the percentage of the above stores that were dynamically not in any critical section. We found that for the programs and configurations we tested, strategy 2 could reduce logging of stores in more than 95% of encountered stores, considering stores both within and outside critical sections. It turns out that `s2` never needed to log outside critical sections, except for `radiosity`. We found that log elision outside critical sections can lead to a large reduction in overheads. While not shown in Table 1, compared to strategy 1, the workload `mtest` ran 25% faster using strategy 2. For the SPLASH2 applications, the runtimes using s2 were better than those using s1 by a number of times, ranging between 3x and 43x[16]. From these results, it appears that for performance reasons, we should really consider s2 as the baseline strategy, perhaps relegating s1 to automatic repair of "buggy" code with known restart races. S2 imposes minimal constraints on the programmer and as Section 7 explains, they are quite reasonable.

In the last 2 columns of Table 1, we present further reduction in overheads that we can expect from strategies 3 and 4 with strategy 2 as the baseline. We saw an improvement in performance up to around 10% using s3 compared to s2[17]. This is a modest improvement but the applications we experimented with did not have much nesting of locks either. Heavy nesting of locks could lead to a larger cost of log elision analysis. Since we do not perform any additional analysis to establish whether a program conforms to strategy 3, it may not be safe all the time to simply not log

[16] We may not see such a large reduction in overheads in real applications simply because there may not be such a large number of stores to persistent memory outside critical sections. The much more modest reduction for the real application MDB/mtest is an indicator as well.

[17] In the application `water-nsq`, we saw an unexpected performance degradation using s3 compared to s2. It is possible that the additional log elision analysis in s2 changes the concurrency pattern in its favor.

outside critical sections (as discussed in Section 8). In fact, for the application `radiosity`, s2 needed to log some stores outside critical sections (around 1% of total stores). While we did not observe any incorrect behavior of `radiosity` with s3, it is left to future work to automatically determine when a program conforms to strategy 3.

It turns out that, among the applications we experimented with, `radiosity` is the only one with dynamic nesting of locks[18]. Since we are not assuming that these locks are released all at once, we consider `radiosity` as non-conformant to strategy 4. For all the other applications, simply turning off building and maintaining the synchronization history is not only safe but also achieves performance improvements upto 37%, compared to strategy 2. This indicates that if a program is known to have no lock nesting, strategy 4 is the best option. In reality, strategy 4 should have even better performance than what we have presented here. An optimal implementation need not log lock and unlock operations at all. It is sufficient to log just the persistent update operations for the current critical section, treat those log entries as unnecessary on encountering an unlock operation, and reuse the existing logs for the next critical section. Such an implementation should have lower logging overheads and significantly better cache performance. Implementing strategy 4 in this optimized manner is left to future work.

12. Related Work

There has been a fair amount of work on providing durable, isolated transactions [10, 35]. While this is a very useful abstraction for maintaining crash-resilience of persistent data structures, and can be seen as an instance of our Strategy 4, we take the more general approach of exploring persistence models in the context of locks. Section 1 compares a lock-based view of persistence with a transaction-based one.

The persistence model that is closest to our work is the one presented in [7]. It presents what we call "Strategy 2" using a more realistic, but less precise formulation. We effectively place this work in a broader context, by expanding it to a set of models with different tradeoffs, while giving it a more precise foundation, and quantifying the expected performance differences between these models. We present novel models that come with some programmer restrictions but we make the case that, for many programs, the performance improvement justifies the tradeoff.

`NVM Direct` [2] is a C library and a set of C extensions for use in persistent memory programming. The APIs of `NVM Direct` support `NVM locking`, `volatile memory mutexes`, and transactions. Their `NVM locks` must be used together with transactions to emulate our lock semantics. Their `NVM locks` are owned by a transaction rather than a thread of execution. An explicit acquire operation on an `NVM lock` is part of the API but no explicit corresponding release operation is available. Locks are released if and when the owning transaction commits or rolls back. The volatile memory mutexes, intended to be used for manipulating transient memory, are exactly the same as traditional mutexes with both acquire and release operations available for them. [2] also supports some other features, notably open transaction nesting, that we do not address.

Although we are not completely confident about the semantics, our reading is that this is essentially another implementation of our Strategy 4 and, in the absence of volatile memory mutexes, it benefits directly from Theorem 5. They appear to use a more aggressive cache-flushing strategy, and do not rely on restart-race-freedom in the way that we do.

In the presence of volatile memory mutexes or other conventional thread synchronization, it is presumably possible, without a conventional data race, to observe the nonvolatile effects of a subsequently rolled back transaction in another thread, and as a result commit a non-volatile memory transaction in that other thread. This would result in an inconsistent state, and such code is presumably discouraged. By combining isolation and `NVRAM` functionality, we avoid the potential for such inconsistencies.

`REWIND` [8] is a user-mode library that enables transactional updates to persistent data structures in `NVRAM`. The API is a `persistent atomic` construct that encloses a set of updates to persistent memory. The guarantee provided by `REWIND` is essentially that of failure-atomicity of a bunch of updates. Write-ahead logs in `NVRAM` are used to provide recoverability in the event of a failure during execution. The logs are maintained in a thread-safe manner but thread-safe access to user data is up to the programmer. In other words, the `persistent atomic` construct in `REWIND` provides no isolation semantics for user data. It appears that based on the isolation properties and consistency requirements of user data structures, a programmer is responsible for correct placement of the `persistent atomic` construct. The NVML Library [3] provides a similar construct whereby a single transaction block works in the context of a single thread. The programmer is responsible for thread isolation in the context of parallel code, an approach again closely related to our Strategy 4. Our more general models use locks to provide both isolation and persistence semantics in a more unified manner.

In this paper, we discussed cache line flushes that can be used to constrain the persist order in which updates are visible in `NVRAM`. A number of papers [11, 25, 31] have tried to relax this constraint. Analogous to memory consistency, [31] introduces memory persistency models that facilitate low-level reasoning about the persist order with respect to failures. A failure is reasoned as a *recovery observer* that atomically reads all of persistent memory at the moment of failure. [31] explains that the order enforced by memory consistency models need not be the same as the order enforced by memory persistency models. A spectrum of models ranging from strict to relaxed persist ordering are described with corresponding programming and performance tradeoffs. More recently, notations were introduced to precisely define the memory persistency models and show code patterns that can leverage them [21]. Along similar lines, `Loose-Ordering Consistency` (LOC) [25] has been introduced to reduce the overhead of enforcing persistence order of updates. Our work targets higher-level programming language models, while this line of work explores hardware-level failure-atomicity models. Clearly we need both, and eventually a mapping of our work to theirs.

Recently, the concept of *Timely Sufficient Persistence* (TSP) [30] has been introduced which makes persist barriers unnecessary for crash resilience. Broadly speaking, TSP encompasses measures that, instead of being applied continually during program execution, can be applied when a failure is imminent in such a way that correct recovery is still possible. A specific example is flushing the cache on failure – if all of the cache can be reliably flushed at that point, the order in which persistent updates reach `NVRAM` does not matter since the recovery observer will see all of them anyway. [30] applies TSP to both lock-based and lock-free programs and shows that for lock free programs, no additional changes are required for crash-resilience, enhancing both usability and performance.

There has been a fair amount of interest in using `NVRAM` for databases. `FOEDUS` [20] is an OLTP engine for a large number of cores and `NVRAM`. Snapshot pages are stored in `NVRAM` in a manner designed to optimize performance. [36] evaluates distributed logging in the context of `NVRAM`. File systems designed to take advantage of `NVRAM` have been implemented [11, 12]. We believe that our work on programming models for `NVRAM` will provide insights that

[18] We found that 3% of the critical sections were nested in `radiosity`.

will guide system software and applications as they try to exploit the advantages offered by NVRAM.

13. Conclusions

We showed that even the somewhat limited space of NVRAM programming approaches and implementations we considered (variables selectively persistent, no concerns about cross-process or cross-machine data sharing, undo-log-based implementation style) is surprisingly rich. There are interesting trade-offs that had largely been overlooked by prior work. At least the last three strategies we described seem worthy of serious consideration. We have shown that various existing approaches, as well as some interesting new ones, notably our Strategy 3, can be viewed and contrasted in the same framework.

References

[1] Process Integration, Devices, and Structures. International Technology Roadmap for Semiconductors, 2007.

[2] NVM Direct Library. At `http://www.oracle.com/technetwork/oracle-labs/open-nvm-download-2440119.html`, retrieved 07/2015, .

[3] pmem.io: Persistent Memory Programming. `http://pmem.io`, retrieved 07/2015, .

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2005.

[5] H. Boehm. Performance Implications of Fence-based Memory Models. In *MSPC*, pages 13–19, 2011.

[6] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 2008.

[7] D. R. Chakrabarti, H. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, Oct. 2014.

[8] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data structures. In *VLDB*, pages 497–508, 2015.

[9] H. Chu. MDB: A memory-mapped database and backend for OpenLDAP. At `http://www.openldap.org/pub/hyc/mdm-paper.pdf`, retrieved 03/2014.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS '11: Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–117, Mar 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP 22*, pages 133–146, Oct 2009.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Eurosys '14*, Apr 2014.

[13] J. E. Gottschlich and H.-J. Boehm. Generic programming needs transactional memory. In *TRANSACT*, Mar 2013.

[14] IEEE and The Open Group. *Posix.1-2008, IEEE Std 1003.1, 2013 Edition.* 2013.

[15] 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, .

[16] Intel Corp. Intel ISA Extensions. `https://software.intel.com/en-us/isa-extensions`, retrieved 07/2015, .

[17] ISO JTC1/SC22/WG14. *ISO/IEC 9899 - Programming languages - C.* 2011.

[18] ISO JTC1/SC22/WG21. Technical Specification for C++ Extensions for Transactional Memory. Available through ISO, or possibly at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf`.

[19] H. J-Boehm. 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.

[20] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD '15*, pages 691–706, Jun 2015.

[21] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. Persistency programming 101. In *6th Annual Non-volatile Memories Workshop*, Mar 2015.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM Transactions on Computers*, C-28(9):690–691, Sep 1979.

[23] J. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2007. ISBN 1–59829–124–6.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09: Proc. of the 36th International Symposium on Computer Architecture*, pages 2–13, Jun 2009.

[25] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, pages 216–223, 2014.

[26] V. Luchangco and V. J. Marathe. Transaction communicators: enabling cooperation among concurrent transactions. In *PPoPP*, pages 169–178, 2011. doi: 10.1145/1941553.1941578. URL `http://doi.acm.org/10.1145/1941553.1941578`.

[27] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[28] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[29] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[30] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT)*, pages 689–694, Mar 2015.

[31] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proc. of the 41st International Symposium on Computer Architecture*, Jun 2014.

[32] M. Saxena, M. Shah, S. Harizopoulos, M. Swift, and A. Merchant. Hathi: Durable transactions for memory using flash. In *DaMoN: Proceedings of 8th ACM/SIGMOD International Workshop on Data Management on New Hardware*, 2012.

[33] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[34] *Draft specification of transactional language constructs for C++*. Transactional memory specification drafting group, Feb 2012. At https://sites.google.com/site/tmforcplusplus.

[35] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–103, Mar 2011.

[36] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. In *PVLDB*, pages 865–876, 2014.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.