Delay-Free Concurrency on Faulty Persistent Memory

Naama Ben-David
Carnegie Mellon University
nbendavi@cs.cmu.edu

Michal Friedman
Technion
michal.f@cs.technion.ac.il

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

YuWHao Wei
Carnegie Mellon University
yuwhao1@cs.cmu.edu

ABSTRACT
Non-volatile memory (NVM) promises persistent main memory that remains correct despite loss of power. This has sparked a line of research into algorithms that can recover from a system crash. Since caches are expected to remain volatile, concurrent data structures and algorithms must be redesigned to guarantee that they are left in a consistent state after a system crash, and that the execution can be continued upon recovery. However, the prospect of redesigning every concurrent data structure or algorithm before it can be used in NVM architectures is daunting.

In this paper, we present a construction that takes any concurrent program with reads, writes and CASs to shared memory and makes it persistent, i.e., can be continued after one or more processes fault and have to restart. The converted algorithm has constant computational delay (preserves instruction counts on each process within a constant factor), as well as constant recovery delay (a process can recover from a fault in a constant number of instructions). We show this first for a simple transformation, and then present optimizations to make it more practical, allowing for a trade-off between computation and recovery delay. We also provide an optimized transformation for normalized lock-free data structures, thus speeding up a large class of concurrent algorithms.

Finally, we experimentally evaluate these transformations by applying them to a queue. We compare the performance of our transformations to that of a persistent transactional memory framework, Romulus, and to a hand-tuned persistent queue. We show that our optimized transformation performs favorably when compared to Romulus. Furthermore, our optimized transformation is even comparable to the hand-tuned version, showing that the generality we provide comes at very little performance cost.

1 INTRODUCTION
A new wave of memory technology, known as Non-Volatile Memory (NVM), have now been released on the newest machines from Intel. NVM is expected to co-exist or replace DRAM for main memory, and promises many attractive features, including persistence under transient failures (e.g., a power failure). This persistence introduces the possibility of recovering the data structure from main memory after a system failure, sparking a flurry of research in this area. However, it also introduces potential for inconsistencies, since caches are expected to remain volatile in these new architectures, losing their contents upon a crash. A lot of work has focused on developing persistent algorithms for index trees [8, 22, 23, 30, 32], lock based data structures [7, 29], and lock-free data structures [10, 14, 17] in this new model.

A natural question that arises is whether we can find general mechanisms that would port algorithms for current machines over to the new persistent setting. One approach has been the development of persistent transactional memory frameworks [13, 21, 24, 27]. This can be an effective approach, although it does not handle code between transactions. Another approach to achieve arbitrary persistent data structures is the design of persistent universal constructions [10, 20]. In particular, Cohen et al. [10] present a universal construction that only requires one flush per operation, thereby achieving optimality in terms of flushes. However, universal constructions often suffer from poor performance, because they sequentialize accesses to the data structure. Furthermore, universal constructions are only applicable to data structures with clearly defined operations, and cannot apply to a program as a whole. For these reasons, even a seemingly efficient universal construction leaves more to be desired.

In this paper, we consider simulators that take any concurrent program and transform it by replacing each of its instruction with a simulation that has the same effect. We define persistent simulations, which exhibit a trade-off between their computation delay, meaning the overhead introduced by the simulation in a run without crashes, and their recovery delay, which is the maximum time the simulation takes to recover from a system crash. Our first result is the presentation of a general persistent simulator, called the Constant-Delay Simulator, that takes any concurrent program using Reads, Writes, and compare-and-swap (CAS) operations, and simulates it with constant computation and recovery delays.

Theorem 1.1. Any concurrent program that uses Reads, Writes, and CAS operations can be simulated in the persistent memory model with constant computation delay and constant recovery delay.
We assume the Parallel Persistent Memory (PPM) model [3, 6]. The model consists of P processors, each with a fast local ephemeral memory of limited size, and sharing a large persistent memory. The model allows for each processor to fault and restart (independently or together). On faulting all the processor’s state and local ephemeral memory are lost, but the persistent memory remains. On restart, each processor has a location in persistent memory that points to a context from which it loads its registers, including program counter, and restarts. Ideas from this model also apply if all processors share the ephemeral memory as a shared cache [9, 10, 17]—although in this case it only makes sense if all processors fail together. Throughout the paper we will use private cache model to refer to the PPM model, and shared cache model for the shared cache variant.

The Constant-Delay Simulator is achieved by using a technique called capsules [6], which in effect introduces checkpoints on a per processor basis from which the program continues after recovering from a crash. In general, the more capsules there are in a program, the smaller the size of each capsule, and therefore, recovery time decreases. The idea behind the Constant-Delay Simulator is to show that we can have constant sized capsules, and that each capsule can be implemented with constant overhead.

In practice, crashes are relatively rare. It is thus important to minimize the computation delay introduced by a persistent simulator, even at the cost of increased recovery time. In the rest of the paper, we therefore present optimizations that can be applied to the simulator to decrease the computation delay. The first such optimization is just as general as the Constant-Delay Simulator in that it applies to any concurrent program. The difference is that we use fewer capsules; we show where boundaries between capsules can be removed, creating capsules that are larger (not necessarily constant sized), to arrive at a smaller computation delay but a larger recovery delay. The second optimization applies to a large class of lock-free data structures called normalized data structures [31]. In this setting, we show how to further reduce the number of capsules and thus the computation delay.

The idea behind capsules is that the program is broken up into contiguous chunks of code, called capsules. Between every pair of capsules, information about the state of the execution is persisted. When a crash occurs in the middle of a capsule, recovery reloads information to continue the execution from the beginning of the capsule. This means that some instructions may be repeated several times. Blelloch et al. [6] noted the need for idempotent capsules, that is, capsules that are safe to repeat, and gave sufficient conditions to ensure that sequential code is idempotent. However, repetitions of concurrent code can be even more hazardous, as other processes may observe changes that should not have happened. We therefore formalize what it means for a capsule to be correct in a concurrent setting, and show how to build correct concurrent capsules.

To build correct capsules for general concurrent programs, we must be able to determine whether a modification of a shared variable can safely be repeated. This can be problematic, because often, the execution of a process depends on the return value of its accesses to shared memory. A bad situation can occur if a process has already made a persistent change in shared memory, but crashed before it could persist its operation’s return value [6]. Attiya et al. [3] consider this problem and define nesting-safe recoverable linearizability (NRL), a correctness condition for persistent objects that allows them to be safely nested. In particular, Attiya et al. [3] introduced the recoverable CAS, a primitive which ensures that if a compare-and-swap by process \( p \) has successfully changed its target, this fact will be made known to \( p \) even if a crash occurs.

At a high-level we show how to combine the ideas of capsules with a recoverable CAS in a careful way to achieve our results for programs with shared reads and CASes. To make the simulation work with writes requires some additional ideas. We use a modification of this recoverable CAS primitive in our capsules to ensure that a program can know whether it should repeat a CAS. We show that the recoverable CAS algorithm satisfies a stronger property than NRL, allowing the recovery to be called even if a crash occurs after the operation has terminated. This property is very important for use in capsules, because after a crash, all operations of the capsule must be recovered, rather than just the most recent one.

We test our simulations by applying them to the lock-free queue of Michael and Scott (MSQ) [28], and comparing their performance with two other state-of-the-art implementations: one using the transactional memory framework Romulus [13], and the other a hand-tuned detectable queue, known as the LogQueue [17]. We did not expect general constructions to match the performance of specialized implementations, but it turns out to be very close. The LogQueue outperforms our transformations on 2-6 running threads, but only by an average of 5%; our most optimized transformation even outperforms the LogQueue on 1, and 7-8 threads. In comparison, the original MSQ is usually between 3.57x to 1.9x faster than the LogQueue, showing that the inevitable cost of persistence outweighs the extra cost paid for generality by our transformations. We further show that our optimized transformation outperform Romulus for the thread counts we tried.

In summary, the contributions of our paper are as follows.

- We define persistent simulations, which consider computation and recovery delay, thereby providing a measure of how faithful a simulation is to the original program, and how fast it can recover from crashes.
- We present a constant-computation and constant-recovery delay simulation that applies to any concurrent program.
- We show optimized simulations that trade-off computation delay for recovery delay, both for general programs and for normalized data structures.
- We show that our transformations are practical by comparing experimentally to state-of-the-art persistent algorithms.

2 MODEL AND PRELIMINARIES

We use the Parallel Persistent Memory (PPM) model [3, 6]. It consists of a system of \( n \) asynchronous processes \( p_1 \ldots p_n \). Each process has access to a persistent shared memory of size \( M \) that it may access with Read, Write, and compare-and-swap (CAS) instructions, as well as a smaller private ephemeral memory, which can be accessed with standard RAM instructions. A process can persist the contents of its ephemeral memory by writing them into a persistent memory location. The ephemeral memory is explicitly managed; it does not behave like a cache, in that no automatic evictions occur. Memory operations are sequentially consistent, and all memory locations can hold \( O(\log M) \) bits.
Each process may fault (crash) between any two instructions. Upon a fault, the contents of a process’s private ephemeral memory is lost, but the persistent memory remains unchanged. After a process fault, it restarts. On restart, the ephemeral memory can be in an arbitrary state, but the persistent memory is not affected by the fault. To allow for a consistent restart, each processor has a fixed memory location in the persistent memory referred to as its restart pointer location. This location points to a context from which to restart (i.e., a program counter and some constant number of register values). On restart this is reloaded into the registers, much like a context switch. Processors can checkpoint by updating the restart pointer. Furthermore a process can know whether it has just faulted by calling a special fault() function that returns a boolean flag, and resets once it is called. We call programs that are run on a processor of such a machine persistent programs.

A similar model for non-volatile memories allows a shared cache and automatic cache evictions [9, 10, 17]. In this model only the whole system can fault. In Section 7, we discuss the difference between the models in more detail, and point out that all our ideas are also valid on the shared cache variant. Throughout the paper we will use private cache model to refer to the PPM model, and shared cache model for the shared cache variant.

For our simulations we also consider a standard concurrent RAM (C-RAM) consisting of \( n \) asynchronous processes with local registers and a shared memory of size \( M \). As with the PPM we assume the C-RAM supports Reads, Writes and CAS instructions on the shared memory, that the memory is sequentially consistent, and registers and memory locations can hold \( O(\log M) \) bits.

For algorithms that implement concurrent objects, we define executions as follows. An execution, \( E \), involves three kinds of events for each process \( p_i \) in the system: invocation events \( i_i(op, obj) \), which invoke operation \( op \) on object \( obj \), response events \( R_i(op, obj) \), in which object \( obj \) responds to \( p_i \)'s operation, and fault events \( C_i \). Fault events are process-specific. On a fault event, \( p_i \) loses all information stored in its ephemeral variables (but shared objects remain unaffected). A process \( p \) executes instructions in an execution, which together make an implementation of the high-level operations represented by the invocation and response events of the execution. The time complexity of an implementation is defined to be the number of instructions process \( p \) executes in the worst case between the invocation and response. Note that this notion of time counts local instructions as well.

### 2.1 Capsules

Our goal is to create concurrent algorithms that can recover their execution after a fault. The main idea in achieving this is to periodically persist checkpoints, which record the current state of the execution, and from which we can continue our execution after a fault. We call the code between any two consecutive checkpoints a capsule, and the checkpoint itself a capsule boundary. At a boundary, we persist enough information to continue the execution from this point when we recover from a fault. This approach was use by Blelloch et al. in [6] and similar to approaches by others [12, 15, 25, 26]. An encapsulation of a program is the placement of such boundaries in its code to partition it into capsules.

**Capsule Correctness.** When executing recoverable code that is encapsulated, it is possible for some instructions to be repeated. This happens if the program crashes in the middle or a capsule, or even at the very end of it before persisting the new boundary, and restarts at the previous capsule boundary. To reason about the correctness of encapsulated programs after a crash, we define what it means for a capsule to be correct in a concurrent setting, intuitively meaning that it can be repeated safely.

**Definition 2.1.** An instruction \( I \) in an execution history \( E \) is said to be invisible if \( E \) remains legal even when \( I \) is removed.

**Definition 2.2.** A capsule \( C \) inside algorithm \( A \) is correct if:

1. Its execution does not depend on the local values of the executing thread prior to the beginning of the capsule, and
2. For any execution \( E \) of \( A \) in which \( C \) is restarted from its beginning at any point during \( C \)'s execution an arbitrary number of times, there exists a set of invisible operations performed by \( C \) such that when they are removed, \( C \) appears to only have executed once in the low-level execution history.

**Definition 2.3.** A program is correctly encapsulated if all of its capsules are correct.

### 3 \( k \)-Delay Simulations

We are interested in efficiently simulating arbitrary computations on a reliable concurrent RAM (C-RAM) on a faulty PPM machine. The performance of concurrent algorithms heavily depends on factors like contention, disjoint access parallelism, and remote accesses. While these factors are difficult to theoretically characterize, they are monumental in their effect on the algorithm’s performance. Therefore, when creating algorithms for the persistent setting, it is important to be able to preserve the structure of tried and tested efficient concurrent algorithms to their persistent counterparts. We formalize this notion of ‘preserving structure’ with the definitions of computation delay, recovery delay and contention delay. Roughly, the first refers to the the number of instructions on the PPM required to simulate each instruction on the C-RAM when there is no fault, the second refers to the number of instructions needed to recover from each fault, and the third takes into account the delay caused by concurrent accesses to the same object.

Before talking about delays, we define what we mean by a simulation. We allow for arbitrary computations on the machine that is being simulated, as long as the machine is sequentially consistent. For simplicity we consider computations from beginning to end.

**Definition 3.1 (Linked Simulation).** Consider a concurrent source machine \( S \) with \( n \) processes for which each process executes a sequence of atomic (linearizable) operations, and a concurrent target machine \( T \), also with \( n \) processes. A linked simulation of \( S \) on \( T \) is a simulation that allows for any computation on \( S \), preserves its behavior, and for each process \( p \) the operations taken by \( T \) (the history) can always be partitioned, contiguously, so that:

1. there is a one-to-one order-maintaining correspondence between these partitions and operations for \( p \) in \( S \), and
2. each partition atomically simulates the corresponding source operation, linearized at some point between the partition’s first and last instruction.
We refer to each partition on each process as a step of the simulation.

Note that our definition includes all local operations as well as any access to shared objects. In our initial simulation, the operations are machine instructions, but more generally they could include linearizable operations on shared objects. We say a step of a linked simulation is fault free if no fault happened on the step’s process on T during that step. For a sequence of instructions s on a single process of either the source or target machine let t(s) be the number of instructions, i.e., the time.

Definition 3.2 (Computation Delay). A linked simulation has k computation delay if for each fault-free step s of the target machine simulating an operation of the source machine, t(s) = O(t(0) + k).

Ben-David et al. use a similar concept of delay for measuring the efficiency of transactions [4]. Persistent computations are tightly coupled with their recovery mechanisms. When discussing a computation for a persistent setting, it is important to also discuss how it recovers from faults. Note that, if all processes fault together during a system fault, simply running a concurrent program as is in a persistent setting yields a trivial 1 computation delay simulation of itself; all steps of the program remain exactly the same. However, upon a fault, the entire computation has to be restarted, and all progress is completely lost. Thus, the recovery time of this ‘simulation’ is unbounded; it grows with the length of the execution. We therefore also formalize the notion of a recovery delay; how long it takes for a persistent program on any process to recover from a fault (processes can fault independently).

Definition 3.3 (Recovery Delay). A linked simulation of a source machine S on a faulty target machine T has k recovery delay if each fault that occurs within a step of the simulation on T adds at most k instructions to the step.

Note that the k for computation delay and recovery delay need not be the same. Furthermore, although k implies a constant, and our simulations guarantee a constant, in general the k could be a function of other parameters of the machine or computation (e.g. input size or number or processes).

A stronger notion of a k computation delay simulation is one in which the amount of contention experienced by an algorithm cannot grow by more than a factor of k either. Accounting for contention helps to capture the structure of an algorithm, since scalability is highly associated with keeping contention as low as possible on all accesses. To be able to discuss contention formally, we follow the definition of contention presented by Dwork et al. in [16]; the amount of contention experienced by an operation op on object O is the number of responses to operations on O received between the invocation and response of op. We now extend the definition of computation delay to account for contention.

Definition 3.4 (Contention Delay). A linked simulation has k contention delay if for any operation during the simulated computation on the source S the contention is C, then the corresponding step in the simulation on the target T has contention at most k × C.

In this paper, we consider persistent simulations that have small computation, recovery and contention delay. We say that an algorithm is X-delay free or if it has c X delay for a constant c, where X is one of computation, contention, or recovery.

4 BUILDING BLOCKS

4.1 Capsule Implementation

We now briefly discuss how to implement the capsule boundaries themselves, and in particular, how we handle persisting the ephemeral variables between capsules. We keep a copy of each ephemeral variable in persistent memory. At a high level, we update the persistent copy of each ephemeral variable with its current value at the end of each capsule, and read these values into ephemeral memory after a crash. However, we must be careful to avoid inconsistencies that could arise if a crash occurs after updating some, but not all, of the persistent copies. Such inconsistencies could occur if ephemeral variables suffer write-after-read conflicts in a capsule [6]. In a nutshell, these conflicts occur if a variable can be read and then written in the same capsule. If the write was persistent and a crash occurs after such a scenario, causing the code to repeat itself from the read instruction, then the read sees a different value than it did originally. These conflicts can be avoided if it can be guaranteed that after a new value is persisted, the program will never repeat an earlier instruction that reads it. Note that in general, it is not a problem if an ephemeral variable suffers a write-after-read conflict, since their writes are to ephemeral memory and thus are not persistent. However, since we persist their new values at the capsule boundary, we must be careful not to overwrite their previous value if we might need to use it again.

Thus, for ephemeral variables that are read and subsequently written to in this capsule, we do not write out their new values into their persistent copies right away. Instead, we keep two persistent write buffers, along with a persistent bit indicating which one of them is currently valid. At the boundary of the end of the capsule, the invalid write buffer is cleared, and for each write-after-read conflicted ephemeral variable v, a tuple of the form (v, newVal) is written into it, where newVal is the value of v at the end of the capsule. After writing out all of these tuples to the write buffer, the validity bit for the write buffers is flipped. At the beginning of each capsule, before executing any of its code, the valid write buffer is read, and the persistent copy of each ephemeral variable that appears in it is updated with this ephemeral variable’s value in the buffer.

Note that persisting the values of ephemeral variables that are either not updated at all, or updated before being read in a given capsule, is not difficult. We don’t have to do anything at the capsule boundary for an ephemeral variable that was not updated in this capsule. For ephemeral variables that were updated before being read in this capsule, we can simply copy in their new value into their persistent copy, without going through the write buffer. This is because the current capsule’s execution does not depend on the value of such an ephemeral variable. This copying is done before the validity bit of the write buffer is flipped.
4.2 Recoverable Primitives

One problem that arises from volatile registers and caches is that the return values of atomic operations can be lost. For example, consider a CAS operation that is applied to a shared memory location. It must atomically read the location, change it if necessary, and return whether or not it succeeded. Return values are stored in volatile registers. If a crash occurs immediately after a CAS is executed, the return value could be lost before the process can view it. When the process recovers from the crash, it has no way of knowing whether or not it has already executed its CAS. This is a dangerous situation; repeating a CAS that was already executed, or skipping it altogether, can render a concurrent program incorrect. In fact, any primitive that changes the memory suffers from the same problem.

This issue was pointed out by Attiya et al. in [3]. To address the problem, they present several recoverable primitives, among them a recoverable CAS algorithm. This algorithm is an implementation of a CAS object with three operations: read, CAS, and recover. The idea of the algorithm is that when CASing in a new value, a process writes in not only the desired value, but also its own ID. Before changing the value of the object, a process must notify the process whose ID is written on the object of the success of its CAS operation. The recovery operation checks this notification to see whether its last CAS has been successfully executed. Attiya et al. show that their recoverable CAS algorithm satisfies nesting-safe recoverable linearizability (NRL), intuitively meaning that as long as recovery operations are always run immediately after crashes, the history is linearizable. Attiya et al.'s algorithm uses classic CAS as a base object, and assumes that CAS operations are ABA-free. This is easy to ensure by using timestamps.

It turns out Attiya et al.'s recoverable CAS algorithm satisfies strict linearizability [2], a stronger correctness property than NRL. The main difference between the two properties is that while NRL only allows recovering operations that were pending when the crash happened, strict linearizability is more flexible. This means that we can define the recovery function to work even on operations that have already completed at the time of the crash. This property is very important for use in the transformations provided in the rest of the paper, in which we may not know exactly where in the execution we were when a crash occurred. To satisfy strict linearizability, we need to tweak the recoverable CAS algorithm slightly, to include the use of sequence numbers on each CAS. In contrast to Attiya et al., we treat the recovery function as another operation of the recoverable CAS object, whose sequential specification is as follows.

Each Recover(i) operation $R$ returns a sequence number seq and a flag $f$ with the following properties:

- If $f = 1$, then seq is the sequence number of the last successful CAS operation with process id $i$.
- If $f = 0$, all successful CAS operations before $R$ with process id $i$ have sequence number less than seq.

We also further modify the recoverable CAS algorithm to create a version that has constant recovery time (instead of $O(P)$), and uses less space ($O(P)$ instead of $O(P^2)$). The pseudocode of our version of the recoverable CAS is given in Algorithm 1. This code also shows the tweaks that we do to the original recoverable CAS algorithm of Attiya et al. Also, in the full version of the paper [5], we give a more detailed description of how it works, and prove it satisfies strict linearizability.

Theorem 4.1. Algorithm 1 is a strictly linearizable, contention-delay free and recovery-delay free implementation of a recoverable CAS object.

The problem of recoverability also applies to atomic write operations; if a process $p$ executes a write that may be seen and possibly overwritten by other processes, it is important that $p$ never repeat its write after a crash, since this can cause an inconsistent state. To handle shared write operations, we reduce the problem to shared CAS operations, and then use the recoverable CAS primitive presented above. We note that often, a shared write can be replaced with a CAS with no effect on the algorithm; that is, an algorithm $A$ that uses CASs and Writes can be simulated by algorithm $A_s$ in which each write operation is implemented with a single CAS. If the CAS fails, this is treated as if the simulated write succeeded, but was overwritten before any other process saw the value. However, in some cases, replacing a write with a CAS does not have the same effect. Intuitively, this could occur if in algorithm $A$, the write races between CAS and write operations on the same location. At a high level, our algorithm does this by adding a level of indirection, leading the racy CAS and Write to actually access different locations. We describe the CAS-Write algorithm in detail in Section 4.3, and show that it is computation-delay free. Once the CAS-Write algorithm is applied, all writes can be replaced with CAS operations, which can be recovered with the recoverable CAS algorithm presented above. Hence, all of our presented simulations apply not only to programs that use CAS and read operations on shared memory, but also to those that additionally use write operations.

4.3 Handling Write-CAS races

Recall from Section 4.2 that it is often safe to replace each shared write with a read followed by a CAS without impacting correctness.
In this section, we handle the cases where this is not possible by giving a constant time implementation of \( M \) writable CAS (WCAS) objects using \( O(M + P^2) \) classic CAS objects. The idea is to use a level of indirection to prevent writes and CASs from racing on the same memory location and then replace all the writes with CASs.

Aghazadeh, Golab and Woelfel [1] presented a general technique which can be used to implement a WCAS object with constant time complexity using \( O(P^2) \) CAS objects. We extend their technique to be more space efficient when implementing many WCAS objects.

Our implementation uses two types of objects: WCAS objects \((W)\) and physical objects \((P)\). The field \( W_p \) stores a pointer to a physical object and \( P_v \) stores the actual value. As a first try, suppose we have unbounded memory. Then to do a read or CAS on \( W \), we simply follow the pointer \( W_p \) to arrive at a physical object \( P \), and read or CAS on \( P_v \). To write value \( v \) into a WCAS object \( W \), we allocate a new physical object \( P \), initialize \( P_v \) with \( v \), and then try to change \( W_p \) to point to \( P \) with a CAS. If this CAS fails, then this high-level write must have been concurrent with another high-level write (whose CAS succeeded) so we can linearize immediately before that write.

To make this more practical, we will recycle physical objects so that we use at most \( O(M + P^2) \) of them. More specifically, we keep around \( M + 2P^2 \) physical objects so that exactly \( 2P^2 \) of them are not pointed to by any WCAS object. These \( 2P^2 \) objects are said to be \( unused \) and they are divided evenly between the processes. Whenever process \( p_i \) performs a CAS causing a physical object \( p_h \) to be unused, it becomes the new owner of \( p_h \). Performing a write boils down to finding an unused physical object to recycle and ensuring that it will not be accessed by any other process. Performing a read or CAS boils down to reserving a physical object so that cannot be recycled by a concurrent write.

Both these properties can be achieved having read and CAS operations reserve physical objects using an announcement array. Process \( p_i \) makes a reservation by first CASing into \( A[i] \) a pointer to the WCAS object \( W \) it is interested in (this CAS cannot fail), reading \( W_p \), and then trying to CAS this physical pointer into \( A[i] \). This CAS can only fail if it is interrupted by a helping write operation that helps us CAS some physical object pointer into \( A[i] \). Regardless of whether \( p_i \)’s CAS succeeds or fails, \( A[i] \) now contains a physical object pointer and \( p_i \) can freely perform reads and CASs on that object until it is cleared. Using this reservation algorithm, high-level read and CAS operations can be implemented in constant time.

Now consider write operations. For simplicity, we first describe an amortized constant time algorithm and then explain how to deamortize it. To find a set of unused physical objects that are safe to recycle, \( p_i \) first cycles through the announcement array and helps whenever it sees a pointer to a WCAS object. To help \( p_i \), process \( p_j \) reads the physical pointer inside the announced WCAS object and then tries to CAS this physical object pointer into \( A[j] \). After helping everyone reserve a physical object, \( p_i \) makes a second pass through the announcement array and recycles any unused physical objects that it owns that were not announced. This recycling algorithm can be implemented in \( O(P) \) time and it finds at least \( P \) objects to recycle.

This is because \( p_i \) owns \( 2P \) unused objects and at most \( P \) of them can be reserved. Each successful write operation consumes one recycled object, so writes take \( O(1) \) amortized time. Pseudo-code can be found in the full version of the paper [5].

To prevent ABA problems, \( A[j] \) also contains a sequence number that is incremented at the start of each high-level read or CAS by \( p_j \). Verifying correctness of the recycle algorithm involves checking two main points. First, we need to make sure that when a physical object \( p_h \) is recycled by \( p_i \), this object cannot be announced until it is reused by some write. This is why the recycle algorithm performs two passes through the announcement array. The first pass of process \( p_i \) atomicity, announcing objects in constant time, and then that process has to execute the entire recycle algorithm. The recycle algorithm helps all processes announce a physical object, so it would have caused \( C \) to fail, a contradiction.

One way to deamortize the write algorithm would be to maintain two lists of recycle-able locations. Whenever we consume a location from one list, we perform a constant amount of work towards populating the other list. We also allow dynamic allocation of WCAS objects. WCAS objects and physical objects are always allocated in pairs to ensure there is always \( O(P^2) \) unused objects.

5 PERSISTING CONCURRENT PROGRAMS

One way to ensure that a program is tolerant to crashes is to place a capsule boundary between every two instructions. We call these Single-Instruction capsules. Can this guarantee a correctly encapsulated program? Even with single-instruction capsules, maintaining the correctness of the program despite crashes and restarts is not trivial. In particular, a crash could occur after an instruction has been executed, but before we had the chance to persist the new program counter at the boundary. This would cause the program to repeat this instruction upon recovery.

Trivially, if the single instruction \( I \) in a capsule \( C \) does not modify persistent memory, then \( I \) is invisible, and thus \( C \) is correct. But what if \( I \) does modify persistent memory? Using the algorithm from Section 4.3, we can implement any writable CAS object using just CAS, so we only need to worry about CASs to shared memory and non-cas writes to private persistent memory. A private persistent write is invisible because the process simply overwrites the effect of its previous operation, and no other process could have changed it in between. So, we only have CASs left to handle. This is where we employ the recoverable CAS operation. We replace every CAS object in the program with a recoverable CAS. We show that it is safe to repeat a recoverable CAS if we wrap it with a mechanism that only repeats it if the recovery operation indicates it has not been executed; any repeated CAS will become invisible to the higher level program. When recovering from a crash, we simply call a checkRecovery function, that takes in a sequence number, and calls the Recover operation of the recoverable CAS.
object. The checkRecovery function returns whether or not the CAS referenced by the sequence number was successful. If it was, then we do not repeat it, and instead continue on to the capsule boundary. Otherwise, the CAS is safe to repeat. Pseudocode for the checkRecovery function is given in Algorithm 2.

With this mechanism to replace CAS operations, single-instruction capsules are correct. The formal proof of correctness is implied by the proof of Theorem 6.1, which we show later.

![Algorithm 2: Check Recoverable CAS](image)

We now show that this transformation applied to any concurrent program $C$ is a contention-delay free simulation of $C$.

**Theorem 5.1.** For any concurrent algorithm $A$ written in the C-RAM model, if $A_s$ is the program resulting from encapsulating $A$ using single-instruction capsules, then $A_s$ is a $c$-contention-delay; $c'$-recovery-delay simulation of $A$, where $c$ and $c'$ are constants.

To prove the theorem, we first show a useful general lemma, that relates the way a simulated object is implemented to the contention-delay of a simulation algorithm. This lemma is proven in the full version of the paper [5].

**Lemma 5.2.** Let $A_s$ be a $k$-computation-delay simulation of $A$. If for every two base objects $O_1$ and $O_2$, the set of primitive objects used to implement $O_1$ is disjoint from the set used to implement $O_2$ in $A_s$, then $A_s$ is a $k$-contention-delay simulation of $A$.

**Proof of Theorem 5.1.** Since each recoverable CAS and each capsule can be used to recover in constant time, it is easy to see that $A_s$ has constant recovery delay. We implement each base object $O$ of $A$ by calling the operations of $O$, followed by a capsule boundary. For CASs, we implement it by replacing the CAS object with a recoverable CAS object and also calling a capsule boundary. Because both the recoverable CAS algorithm and the capsule boundary take a constant number of instructions, we have shown that our transformation is a $k$-computation-delay simulation of $A$. Furthermore, each recoverable CAS object uses primitive objects that are unique to it, and not shared with any other object. Note that while the capsule boundary does use primitive objects that are shared among other capsule boundaries, the capsule boundaries are in fact local operations, since each process uses its own space for persisting the necessary data. So capsule boundaries do not introduce any contention. The rest of the proof therefore follows from Lemma 5.2. □

6 OPTIMIZING THE SIMULATION

Although capsule boundaries only consist of a constant number of uncontended instructions, they can still be expensive in practice, as they require persisting several pieces of data and use up to two fence instructions. We now discuss how to reduce the number of required capsule boundaries in a program, while still maintaining correctness. Fewer capsule boundaries means more instructions per capsule, so more progress could be lost due to a crash.

In this section, we focus on programs that use CASs and reads as their mechanisms for accessing shared data. Using our implementation of writable CAS objects (presented in Section 4.3), we can extend this to programs that use writes as well, thereby covering many concurrent programs.

6.1 CAS-Read Capsules

We begin by showing that at a high level, as long as there is only one CAS operation per capsule, and this operation is the first of the capsule, the program remains correctly encapsulated.

We note that Blelloch et al. [6] comprehensively showed how to place capsule boundaries in non-racey persistent code to ensure idempotence. Their guideline is to create capsules that avoid write-after-read conflicts (See Section 4.1). Therefore, in addition to the capsule boundaries dictated by instructions on shared memory as outlined above, we also place a capsule boundary between a read of a persistent private location in memory and the following write to that location. However, note that we don’t always add this extra boundary; if a capsule begins with a persistent write of a private variable $v$, any number of reads and writes to $v$ may be executed in the same capsule, since there is no write-after-read conflict.

Another potentially dangerous situation arises when we have a branch that depends on a shared memory read and the two paths after the branch write to different persistent memory locations. If this is all placed within a single capsule, then crashing could cause an incorrect execution where both persistent memory locations are written to. For example, there could be a crash immediately following the write to one location, and after recovering, the process could see a different shared value and decide to follow the other branch, leading to a write on the other persistent memory location.

We call this construction a CAS-Read capsule. We also allow for capsules that do not modify any shared variables at all. We call such capsules Read-Only capsules. Intuitively, all read operations are always invisible, as long as their results are not used in a persistent manner. So, a capsule that has at most one recoverable CAS operation, followed by any number of shared reads, is correct.

Note that we assume that every process has a sequence number that it keeps locally, and increments once per capsule. At the capsule boundary, the incremented value of the sequence number is persisted (along with other ephemeral values, like, for example, the arguments for the next recoverable CAS operation). Therefore, all repetitions of a capsule always use the same sequence number, but different capsules have different sequence numbers to use.

We now describe in more detail how to use the recovery function of the recoverable CAS object. This assumption is realistic, since in most real systems, there is a way for processes to know that they are now recovering from a fault. We use the fault() function to optimize some reads of persistent memory— if we are recovering from a fault, we read in all ephemeral values we need for this capsule from the place where the previous capsule persisted them. Otherwise, there is no need to do so, since they are still in our ephemeral memory. We show pseudocode for the CAS-Read capsule in Algorithm 3. Read-Only capsules are a subset of the code for CAS-Read capsules.
We now show that the CAS-Read capsule is correct. This fact trivially implies that Read-Only capsules are correct as well, so we do not prove their correctness separately. We wrap up this section by showing that a transformation that applies CAS-Read and Read-Only capsules is a $c$-contention-delay simulation for constant $c$. Intuitively, removing capsule boundaries can only improve the contention delay.

Note that the definition of correctness is with respect to an algorithm that contains the capsule. Here, we prove the claim in full generality; we show that this capsule is correct in any algorithm that could use it. For this, we argue that its repeated operations are invisible in any execution, despite possible concurrent operations. Note that this implies correctness for any context in which the capsule might be used. Theorem 6.1 is proven in the full version [5].

**Theorem 6.1.** If $C$ is a CAS-Read capsule, then $C$ is a correct capsule. We also require that each process increments the sequence number before calling CAS.

**Theorem 6.2.** A program that uses only CAS-Read, Read-Only, and Single-Instruction capsules is correctly encapsulated, and is a contention-delay-free simulation of its underlying program.

**Proof.** Since CAS-Read, Read-Only, and Single-Instruction capsules are all correct (corollary of Theorem 6.1), by definition, a program that uses only these capsules is correctly encapsulated. Furthermore, since by Theorem 5.1, a program encapsulated with single-instruction capsules only is a constant-contention-delay simulation of its underlying program, and CAS-Read and Read-Only capsules use strictly less instructions, programs encapsulated with these capsules are also contention-delay free simulations.

### 6.2 Normalized Data Structures

Timnat and Petrank [31] defined normalized data structures. The idea is that the definition captures a large class of lock-free algorithms that all have a similar structure. This structure allows us to reason about this class of algorithms as a whole. In this section, we briefly recap the definition of normalized data structures, and show optimizations that allow converting normalized data structures into persistent ones, with less persistent writes than even our general Low-Computation-Delay Simulator would require. We will show two optimizations; one that works for any normalized data structure, and one that is more efficient, but requires a few more (not-too-restricting) assumptions about the algorithm.

Normalized lock free algorithms use only CAS and Read as their synchronization mechanisms. At a high level, every operation of a normalized algorithm can be split into three parts. The first part, called the **CAS Generator**, takes in the input of the operation, and produces a list of CASs that must be executed to make the operation take effect. The second part, called the **CAS Executor**, takes in the list of CASs from the generator, and executes them in order, until the first failure, or until it reaches the end of the list. Finally, the **Wrap-Up** examines which CASs were successfully executed by the executor, and determines the return value of the operation, or indicates that the operation should be restarted. Interestingly, the Generator and Wrap-Up methods must be parallelizable, intuitively meaning that they do not depend on a thread’s local values, and can be executed many times without having lasting effects on the semantics of the data structure.

**Optimizations.** Our Low-Computation-Delay Simulator works for all concurrent algorithms that access shared objects using only read, write and CAS. In particular, works for normalized data structures. However, we can exploit the additional structure of normalized algorithms to optimize the simulation.

Note that placing capsule boundaries around a parallelizable method yields a correct capsule. This is implied from the ability of parallelizable methods to be repeated without affecting the execution, which is exactly the condition required for capsule correctness. The formal definition of parallelizable methods is slightly different, but a proof that this definition implies capsule correctness appears in [11]. Thus, there is no need to separate the code in parallelizable methods into several capsules. Furthermore, there is also no need to use recoverable CAS for some of the CAS operations performed by parallelizable methods; for normalized data structures, we can simply surround the CAS generator and the Wrap-Up methods in a capsule, and do not need to alter them in any other way.

All that remains now is to discuss the CAS executor, which simply takes in a list of CASs to do, and executes them one by one. No other operations are done in between them. Note that we can convert CAS operations to use the recoverable CAS algorithm, and then many consecutive CASs could be executed in the same capsule, as long as they access different objects. In the case of normalized data structures, however, we do not have the guarantee that the CASs all access different shared objects. Therefore, we cannot just plug in that capsule construction as is. However, we note another quality of the CAS executor that we can use to our advantage: the executor stops after the first CAS in its list that fails. Translated to the language of persistent algorithm, this means that we do not actually need to remember the return values of each CAS in the list separately; we only need to know the index of the last successful CAS in the list. Fortunately, the recovery operation of the recoverable CAS algorithm actually gives us exactly that; it provides the sequence number of the last CAS operation that succeeded. Therefore, as long as we increment the sequence number by exactly 1 between each CAS call in the executor, then after a crash, we can use the recovery function to know exactly where we left off. We can then continue execution from the next CAS in the
list. Note that if the next CAS in the list actually was executed to completion but failed before the crash, there is no harm in repeating it. We simply execute it again, see that it failed, and skip to the end of the executor method.

For a recoverable CAS to work correctly, all CASs to that object must use the recoverable CAS algorithm. Whenever a generator or wrap-up method performs CAS on an object that could also be modified by a CAS-executor, it must use recoverable CAS instead of classic CAS. This is because even though the generator or wrap-up method never needs to recover a CAS’s results, it is still important to notify other processes of the success or failure of their last CAS.

We now discuss a method that allows removing the capsule boundary between the executor and the wrap-up. We argue that as long as we can recover the arguments and results of each executor CAS, it is safe restart the execution from the beginning of the executor. Suppose a combined executor plus wrap-up section faults and repeats multiple times, we first argue that as long as the wrap-up part cannot overwrite the notification of any CAS in the cas-list, then in every repetition, the executor returns exactly the same index in the cas-list. Recall that we assume CAS operations are ABA-free in the original program (i.e. the object cannot take on a value twice the capsule restarts. Also, if the executor of a capsule succeeds. If the wrap-up does not access any CAS location on each capsule, we can still ensure that we can recover. To remove the capsule immediately after the executor, we need to ensure that the wrap-up does not corrupt the ability of the recoverable CAS to tell whether the most recent executor CAS on each object succeeded. If the wrap-up does not access any CAS location accessed by the executor, this property is guaranteed. However, if there is a CAS in the wrap-up that accesses the same location as some CAS in the executor, we can still ensure that we can recover. Let $C_w$ be such a CAS in the wrap-up executed by process $p$. Note that $C_w$ never needs to use the recovery function for itself; since the wrap-up is recoverable, it is always safe to repeat $C_w$ after a crash. Therefore, when $C_w$ is executed using a recoverable CAS, it can leave out its own ID and sequence number, so that other processes do not notify $p$. Thus, the previous notification that $p$ received (i.e. a notification about $p$’s executor CAS on the same object) remains intact.

Notice that if we have two parallelizable methods, $A$ and $B$, next to each other, we can actually put them in a single capsule as long as the inputs to $A$ and $B$ are the same whenever the capsule restarts. Since $A$ and $B$ have the same inputs, we know by parallelizability that $A$ and $B$ each appear to execute once regardless of how many times the capsule restarts. Also, $A$ must appear to finish before $B$ because there was a completed execution of $A$ before any invocation of $B$. Therefore, this capsule appears to have executed only once.

So, we can avoid an additional capsule boundary between the current iteration’s wrap up method and the next iteration’s generator method, as long as we now use the same notification trick in the CASs of the generator as well. So as long as there are capsule boundaries before and after each call to a normalized operation, we only need one capsule boundary in each iteration of the main loop: only before the executor. We call this simulation the Persistent Normalized Simulator. The details of our encapsulation are shown in Algorithm 4. The results of this section are summarized in Theorem 6.3.

**Theorem 6.3.** Any normalized data structure $N$ can be simulated in a persistent manner with constant-contention-delay using one capsule boundary per repetition of the operation.

---

**Algorithm 4: Persistent Normalized Simulator**

```cpp
result_type NormalizedOperation(arg_type input) {
  do {
    cas_list = CAS_Generator(input);
    capsule_boundary();
    if (fault()) {
      cas_list = read(CAS_list);
      seq = read(seq);
    }
    idx = CAS_Executor(cas_list, seq);
    (output, repeat) = Wrap_Up(cas_list, idx);
    if (repeat == true) return output;
  } while (repeat == true);

  seq = read(seq);
  cas_list = read(CAS_list);
  for (i = 0; i < CASs.size(); i++) {
    if (RCAS(CASs[i])) return i;
    done = check_recovery(CASs[i].obj, seq, p);
  }
  idx = CAS_Executor(cas_list, seq);
  (output, repeat) = Wrap_Up(cas_list, idx);
  if (repeat == true) return output;

  return CASs.size();
}
```

---

### 7 PRACTICAL CONCERNS

**Implementing Capsules in Real Programs.** When interpreting real program code in terms of our model, we assume that all stack-allocated local variables, including the program counter, are updated in ephemeral memory, and that heap-allocated variables are created and updated directly on persistent memory. That is, the ephemeral variables in our model map to stack-allocated variables, and we add to each stack frame the program counter at the last capsule boundary. We therefore keep a copy of the stack in persistent memory, and use a doubling trick similar to the one discussed in Section 4.1 to handle stack-allocated variables with write-after-read conflicts. Instead of using write buffers to only duplicate the variables that have write-after-read conflicts, another possibility is to duplicate all stack-allocated variables, that is, have two persistent copies of each stack frame, and toggle between the two, copying all values over from one to the other at the end of every capsule. This technique seems wasteful, but is very robust, and works well when the number of variables is small.

Note that from the control flow graph of the program, we can know which variables suffer write-after-read conflicts in a capsule.
If these variables stay the same across many capsules, we can duplicate only them, and have only one copy for the rest of the variables. A validity bit can indicate which copy of the duplicated variables is valid, similarly to how the write buffers work. The validity bit must be flipped after all other changes are made, and flipping it represents the atomic transition between capsules. This technique allows us to use space only as necessary, as the write buffer technique does, but reduces the number of times values are copied over to different locations. If between two capsules the variables with write-after-read conflicts change, we can use the copy-all technique to remap the memory usage to have copies of the correct variables.

Note also that generally, flushes are done at the granularity of a cache line. Thus, if all stack-allocated variables (with the necessary duplications) fit in a single cache line, we can simply ensure that they are all updated before the validity bit is flipped, and only need a single fence to commit the capsule.

Since there may be many heap-allocated variables, we do not handle them in the same way as stack-variables. Instead, they are directly written on persistent memory. Therefore, we must make sure that repeated code does not corrupt their values by setting capsule boundaries in between instructions that may harm each other, just like we do for the shared memory instructions.

**Shared vs Private Model.** Recall that in the private cache model, we assume that the only way for processes to communicate is through persistent memory (i.e., all shared memory is persistent). Furthermore, the volatile memory is explicitly managed, and no automatic flushes occur. A similar model has been considered in several other works [10, 17, 20]. On the other hand, in a shared cache model, processes communicate through objects in volatile memory rather than persistent memory. The values in these objects are persisted when the program issues an explicit flush instruction or when a cache line is evicted automatically. The shared model is more faithful to current cache coherent machines, while the private model helps to abstract away machine-specific flushes.

A simple transformation to convert an algorithm for the private cache variant into an algorithm that works in the shared cache variant was presented by Izraelevitz et al. in [20]. This transformation applies to programs written in the release-consistency memory model. After every load-acquire, it adds a flush and a fence. Before every store-release, it adds a fence, and after every store and store-release, it adds a flush. When transforming an algorithm from the shared cache model to a model in which cache lines may be automatically evicted, one needs to consider not only shared variables, but also local ones. Inconsistencies can occur in code that might repeat changes to persistent local variables (due to a crash). This can be handled again by avoiding write-after-read conflicts [6], as is discussed for heap-allocated variables in Section 6.

**Compiler.** Note that we treat shared variables differently from private ones; private heap-allocated variables may be written to many times in a single capsule, but this is disallowed for shared variables. It is thus important that the compiler be able to distinguish between these two types of variables. One way to achieve this is to have the user annotate shared variables. We also need annotations to allow the compiler to determine which of our constructions should be used; for normalized data structures, we assume the user can annotate the generator, executor, and wrap-up sections. We also assume that for simple cases, the compiler can determine if a variable is ever used again. Therefore a capsule boundary only needs to persist the variables that may be used in the future.

**Constant Stack Frames.** Recall that for the stack-allocated variables, we assume that there is only a constant number of them (around the same as the number of bits in a word) in each stack frame. This is important to be able to atomically update the validity mask of the variables in each capsule boundary, as in Section 2.

**CAS.** Also recall that the recoverable CAS algorithm requires storing not only the value, but also an ID and sequence number in each CAS location. This can be achieved by using a double-word CAS, which is common in modern machines.

**Flushes.** In a capsule boundary, if all the local variables fit on the same cache line, then we only need one fence for the capsule since the cache line gets flushed all at once in the private model. On the other hand, note that on modern machines with automatic cache evictions, writing all variables on one cache line does not guarantee atomicity, since an eviction can happen part way through updating the cache line. However, we can still assume, as is done in [9], that writes to the same cache line are flushed in the order they are written. Intuitively, this is because on real machines, the following three properties generally hold: (1) total store order (TSO) is preserved, (2) individual words are written atomically, and (3) each cache line is evicted atomically.

**Memory mapping.** Note that for our algorithms to recover, we assume that after a crash, the each process can always find the memory in which the capsule boundary stored information. This requires persisting the page table. We assume that this is done by the operating system. We further assume that each process is assigned the same virtual address space as it was before the crash. These two assumptions together ensure that each process receives the same physical address space before and after a crash. More details about the virtual to physical mapping for persistent memory is given in [7].

**Definitions.** A linearizable data structure is said to be *durably linearizable* if at any time, the state of the data structure in persistent memory is consistent with a linearization of the execution up to that point [20]. A data structure is said to be *detectable* [17] if the operations and their return values are persisted, indicating whether an operation has been executed or not, thus allowing a process to recover and continue its execution after a crash.

8 EXPERIMENTS

We measured the overhead of our general and normalized data structure transformations by applying them to the lock-free queue of Michael and Scott [28]. We also compare against Romulus [13], a persistent transactional memory framework, and LogQueue [17], a hand-tuned persistent queue. Both Romulus and LogQueue provide durability [20] and detectability [17] in the shared cache model. We use the shared cache model for our experiments because it’s closer to the machine that we test on. But this means we need to somehow translate our detectable queues from the private cache model to the shared cache model. We consider two different ways of doing this translation: automatically by Izraelevitz et al.’s durability transformation [20] or manually by hand.

We ran our experiments on Amazon’s EC2 web service with Intel(R) Xeon(R) Platinum 8124M CPU model (8 physical cores,
2-way hyperthreading, 3GHz and 25MB L3 cache), and 16GB main memory. The operating system is Ubuntu 16.04.5 LTS. Just like in [7, 13, 17], we assume that the cost of flushing on this system will be similar to what we will see in real NVM systems.

All functions were implemented in C++ and compiled using the g++ 5.4.0 with -O3. We only measured the performance on 1-8 threads (each on a separate core), as queues are not a scalable data structure. As in previous work [17, 28], we evaluated the performance with threads that run enqueue-dequeue pairs concurrently. In all the experiments we present, the queue is initiated with 1M nodes; however, we also tested on a nearly-empty queue and verified that the same trends occur. The flush operation consists of two instructions: clflushopt, and sfence. Clflushopt has store semantics as far as memory consistency is concerned. It guarantees that previous stores will not be executed after the execution of the clflushopt. According to Intel, flushing with clflushopt is faster than executing flushes using clflush [19]. We also tried using clwb instead of clflushopt and found no difference in performance. The sfence instruction guarantees that the clflushopt instruction is globally visible before any following store instruction in program order becomes globally visible. We omitted some fences when the ordering of the flushes is not important. All the presented results use the recoverable CAS algorithm that was proposed by Attiya et al. [3]. In our experiments, their algorithm performed slightly better than ours and thus was the one that was presented.

Each of our tests were run for 5 seconds and we report the average throughput over 10 runs. In general, queues that contain less flushes perform better, which is consistent with what we expected.

In our experiments, our goal is to understand the overhead general programs would observe if they were made persistent using various methods. When we run the queue experiments, we keep in mind that these queues should be used within general programs. So, before calling each of the queue operations, the general program has to execute a capsule boundary. This is true for all queues that we test, including the LogQueue and Romulus. Therefore, since this additional overhead would be the same for all queues tested, we omit it in our comparative experiments. However, we note that the LogQueue and Romulus produce stand-alone data structures, that maintain more information than our queues do if the initial capsule boundary is removed. This means that in some specific contexts, for example when a few queue operations are executed consecutively, a capsule boundary can be avoided before calling LogQueue or Romulus operations, whereas our constructions still require it. It is possible to store some extra information in our queue constructions to match the properties of the other queues, but this requires some careful manipulations, which are outside of the scope of this paper.

Using the Izraelevitz Construction. One general way to automatically achieve correctness in the shared model is to use the construction presented by Izraelevitz et al. [20] (described in Section 7). Figure 5 a) shows the result of applying our transformations along with Izraelevitz’s construction to the Michael-Scott queue (MSQ) [28]. To isolate the overhead of our transformations, we also show the performance of a Michael Scott queue with just the Izraelevitz construction. We call this the IZ queue and it is an upper bound on how well our transformations can perform. The result of the Low-Computation-Delay Simulator is called General. Normalized represents the normalized data structure transformation introduced in Section 6.2.

As the General queue contains more capsule boundaries than the Normalized queue, we can see that Normalized performs 1.25x better when there are 2 running threads, and 1.17x better when there are 8 threads. Without any of the detectability transformations, the IZ queue performs better than Normalized by 1.13x and 1.26x for 2 and 8 threads respectively.

Competitors. Another way to make our transformed queues correct in the shared model is to add flushes manually. The flushes we add are very similar to those in Friedman et al.’s Durable Queue [17]. The difference is that we flush both the head and tail to allow for faster recovery and we omit the return value array; Friedman et al. used the return value array to recover return values following a crash, but this functionality is handled by our transformations.

We compared the manual flush version of our transformed queues with the Log [17] as well as a queue written using Romulus [13]. We chose the RomulusLR version as it performed better for every thread count. The results are depicted in Figure 5 b). For scalable memory allocation, our transformations use jemalloc and Romulus uses its own scalable memory allocator. We ran Log queue both with and without jemalloc and found that it tends to be faster without jemalloc. The faster set of numbers are reported.

Our Normalized queue performs much better than Romulus when the thread count is low, which could partly be because Romulus incurs the extra overhead of implementing a persistent memory allocator and general transactional memory. Romulus scales very well and outpaces General because it uses flat combining [18], a technique where update transactions are aggregated and processed with a single lock acquisition and release.

When compared to Log queue, we found that Normalized performs better by 29% on one running thread and by up to 9% on 7-8 threads. The Log queue is better by up to 10% on 2-6 threads. We believe this is because Normalized performs less overall fences compared to Log queue, however, in some places, Normalized performs more work in between a read and its corresponding CAS. These instructions bottleneck performance at higher thread counts. With clever cache line usage, it is also possible to reduce Log queue enqueues by one flush, but we did not implement this in our experiments. Given that the Log queue was tailored to one particular data structure, we were impressed that our Normalized automatic construction gets comparable performance. Although we did not measure this experimentally, our transformation also has lower recovery time compared to Log queue in situations where the size of the queue is larger than the number of processes. This is because the recovery function of Log queue traverses the entire queue starting from the head. In contrast, our recovery function just involves loading the previous capsule and performing the recovery function of a recoverable CAS object. Since we are using Attiya et al’s implementation of recoverable CAS [3], recovery time is linear in the number of threads.

Figure 5 c) shows how these persistent queues compare to the original MSQ. From the graph, it looks like the cost paid by our transformations to ensure generality and quick recovery is not so much compared to the inevitable cost of persistence.
ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444.

REFERENCES


Figure 5: Throughput of persistent and concurrent queues under various thread counts. Normalized and General are the result of our transformations.