POSTER: Making Concurrent Algorithms Detectable

Naama Ben-David
CMU

Guy E. Blelloch
CMU

Michal Friedman
Technion

Yuanhao Wei
CMU

Abstract

Non-volatile memory (NVM) promises persistent main memory that remains correct despite loss of power. Since caches are expected to remain volatile, concurrent algorithms must be redesigned to ensure a consistent state after a system crash, and to continue the execution upon recovery.

We give the first general construction to make any concurrent program persistent, and show that the persistent version is guaranteed to have at most a constant factor blow-up in both steps and contention. We also provide an optimized transformation for normalized lock-free data structures. We experimentally evaluate our transformation by comparing it to a persistent transactional memory, as well as a hand-tuned persistent algorithm. We show that our transformation's performance is reasonable given its generality.

CCS Concepts • **Theory of computation** → **Concurrency**; • **Hardware** → *Emerging technologies.*

1 Introduction

Non-Volatile Memory (NVM) is making its way into modern architectures, and is expected to replace DRAM for main memory, promising persistence under loss of power. This persistence introduces the possibility of recovering the data structure from main memory after a system failure. However, it also introduces potential for inconsistencies, since caches will likely remain volatile, losing their contents upon a crash.

A natural question is whether we can find general mechanisms to port algorithms for current machines over to the new persistent setting. Work on achieving general solutions for persistence has taken two different approaches. Blelloch *et al.* [3] persistent a program by dividing it into continguous chunkc of code called *capsules*. The stack frame and program counter are persisted between every pair of capsules. Upon a crash, the most recently persisted information is read, and the program continues from the beginning of the last capsule. Blelloch *et al.* use capsules to make general race-free parallel programs persistent, but leave the problem of persisting concurrent programs largely unaddressed. Attiya *et al.* [1] present algorithms for basic concurrent primitives that satisfy *NRL*, a correctness condition that allows persistent objects to be safely nested. They leave open the question of how to use these primitives for general persistent programs.

Results. We present a general simulation that takes any concurrent algorithm that uses Read, Write and CAS operations on shared memory, and makes it persistent. Furthermore, we show an optimized simulation for a large class of lock-free data structures called normalized algorithms [7]. Our simulation provides *detectability*; the results of each operation can be recovered after a crash [5]. It combines insights from capsules and persistent concurrent primitives.

We show our simulation is an efficient simulator of concurrent programs; roughly, a simulator transforms any concurrent program into another such that every instruction is replaced with a simulation with the same effect. We say the *computation delay* of a simulator is the maximum number of steps taken to simulate any instruction. The *recovery delay* of a simulator is the max number of steps it takes to correctly restart a computation from the place where it failed.

Theorem 1.1. *A p processor shared-memory machine with reads, writes and CAS instructions can be simulated with constant computation and recovery delay on a p processor faulty persistent memory machine.*

We test our simulations by applying them to the lock-free MichaelScott (MS) queue [6], and comparing their performance with Romulus [4] (transactional memory framework) and with LogQueue [5] (a hand-tuned detectable queue). We do not expect general constructions to match the performance of specialized implementations. Indeed, the LogQueue outperforms our transformations, but only by about a factor of 1.55x on 8 threads; our most optimized simulation even outperforms the LogQueue on lower thread counts.

In this short paper, we can only highlight the main ideas and results. More details can be found in the full paper [2].

2 Using Capsules in Concurrent Code

Each process p can access an unbounded *non-volatile* shared memory (NVM) with Read, Write and CAS instructions, as well as a small private volatile memory, accessed with standard RAM instructions. p may *crash* at any time. Upon a crash, the contents of p's private memory are lost, but the persistent memory remains unchanged. p can *persist* the contents of its volatile memory by writing them into NVM.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3299991

We create checkpoints by breaking code into capsules; after a crash, the execution continues from the last capsule's beginning. This means that some instructions inside a capsule may be repeated after a crash. To build correct capsules, we must to determine if a modification of a variable can be safely repeated. To handle this, we replace each CAS with a *recoverable CAS* [1], a primitive which ensures that if a CAS by process p has successfully changed its target, this fact will be made known to p even if a crash occurs. We wrap each recoverable CAS with a mechanism that only repeats it after a crash if its recovery mechanism indicates it has not been executed. In the full version, we show a variant of the recoverable CAS algorithm that has constant recovery time.

3 Constant-Delay Simulation

To create a persistent program from a concurrent one, we can put each instruction in its own capsule (after converting CASes into recoverable CASes). We call this our *simple simulation*. Intuitively, our simple simulation perserves the structure of the original algorithm. We formalize this with the notion of a *k-computation-delay simulation*.

Definition 3.1. *A concurrent algorithm A is a* k-computation-delay simulation *of another concurrent algorithm* A' *if A follows the same steps as* A'*, but replaces each base object* O' *of* A' *with an implementation of* O' *that takes at most* k *steps.*

In the full paper, we extend k-computation-delay simulations to *k-contention-delay* simulations, which ensure at most a k-factor blow-up in the contention experienced by each instruction. We also discuss *k-recovery delay*, meaning that upon a crash, the program takes at most k steps before it can continue from where it left off. We show that our simple simulation achieves both constant contention delay and constant recovery delay, leading to Theorem 1.1.

In practice, capsules can be expensive, so we want as few of them as possible. We can improve upon our simple simulation by defining *CAS-Write-Read* capsules, which contain one CAS or Write followed by any number of reads. In the full version, we show that such capsules are safe to repeat.

4 Normalized Simulation

Timnat and Petrank [7] defined *normalized data structures*. The idea is that the definition captures a large class of lock-free algorithms that all have a similar structure, allowing us to reason about many algorithms at once.

At a high level, every operation of a normalized algorithm can be split into three parts: *CAS Generator*, *CAS Executor*, and *Wrap-Up*. In the full paper, we show that such algorithms allow us to further reduce the number of capsules we use; one capsule can contain both the CAS generator and the Wrap-Up. The CAS executor needs to have its own capsule, in which all CASes are made recoverable. Recoverability is not required of the CAS operations in the generator and wrap-up parts, unless they access the same object as a CAS in the executor. We also show a further optimized version for normalized data structures, that has the same number of capsule boundaries, but makes them more lightweight.

5 Experiments

We measure the overhead of our general (CAS-Write-Read) and normalized simulations by applying them to the MS queue [6], and compare against Romulus [4] and LogQueue [5]. We test on a machine with a shared cache that gets automatically flushed, so to test our algorithms we add flushes to them to ensure safety. The results are depicted in Figure 1. Our normalized simulations perform better than Romulus, but our general simulation is slower than *RomulusLR* for more than 6 threads. We found that *NormalizedO2* performs better than the LogQueue by 1.29x on 1-2 threads and *Log* queue is better by up to 1.52x on 3-8 threads. We believe this is because *NormalizedO2* performs less overall operations compared to *Log* queue, however, in some places, *NormalizedO2* performs more work in between a read and its corresponding CAS. Given that the *LogQueue* was hand-tuned, the performance of the general normalized transformation is impressive.

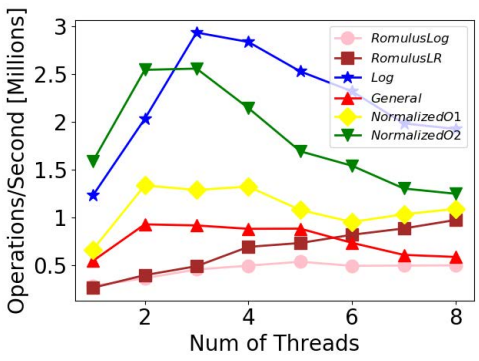

Figure 1. Comparing our transformed queues with manual flushes to prior work.

References

[1] Hagit Attiya, Ohad Ben Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *PODC*.

[2] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. 2018. Delay-Free Concurrency on Faulty Persistent Memory. *arXiv preprint arXiv:1806.04780* (2018).

[3] Guy Blelloch, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The Parallel Persistent Memory Model. In *SPAA*.

[4] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *SPAA*. ACM, 271–282.

[5] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *PPoPP*. ACM, 28–40.

[6] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*. ACM, 267–275.

[7] Shahar Timnat and Erez Petrank. 2014. A practical wait-free simulation for lock-free data structures. In *PPoPP*, Vol. 49. ACM, 357–368.