A Persistent Lock-Free Queue for Non-Volatile Memory

Extended Abstract

Michal Friedman[1], Maurice Herlihy[2], Virendra Marathe[3], and Erez Petrank[1]

[1]Technion, Israel [2]Brown University, USA [3]Oracle Labs, USA

Abstract– This paper proposes three novel implementations of a concurrent lock-free queue for non-volatile byte-addressable memory. These implementations illustrate algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. In presenting these challenges, the proposed algorithmic designs, and the different durability guarantees, we hope to shed light on ways to build a wide variety of durable data structures.

1 Introduction

Memory is said to be *non-volatile* if it does not lose its contents after a system crash. Non-volatile memory is soon expected to co-exist with or even displace volatile DRAM for main memory (but not caches or registers) in many architectures. This has led to increasing interest in the problem of designing and specifying *durable* data structures, that is, data structures whose state can be recovered after a system crash.

A major challenge in designing durable data structures is that caches and registers are expected to remain volatile. Thus, the state of main memory following a crash may be inconsistent, missing all previous writes to the data structure that were present in the cache but not yet written into the main memory.

Previous work focuses on search trees (e.g., [3–7]), especially on B-tree implementations. The interest in B-trees is natural given their prevalence in file system and database implementations. However, other foundational data structures are also used in application domains that care about persistence, e.g., hash tables in key-value stores and persistent message queues. Since traditional storage media have been block-based, all these applications persist these data structures by marshaling them to a block-based format. Doing so involves non-trivial overhead that was dwarfed by the high cost of disk access. As a result, the in-memory representation and on-disk (-SSD) representation of these data structures are quite different. Following this work, further data structures have been presented, such as linked-lists, hash-tables and BSTs [10].

Designing such concurrent data structures for upcoming non-volatile memories requires meeting the combined challenge of high concurrency and non-volatile durability. We study these challenges by designing a durable version of the lock-free concurrent queue data structure of Michael and Scott [8], which also serves as the base algorithm for the queue in java.util.concurrent. This concurrent data structure is complicated enough to demonstrate the challenges raised by concurrent durable data structures, and simple enough to demonstrate solutions. Careful thought is needed to define what it *should mean* for a concurrent structure to be correct and durable.

Various definitions were proposed to formalize durability, and here we adopt and work with the definition of linearizable durability by Izraelevitz *et al.* [2]. Informally, durable linearizability guarantees that the state of a data structure following a crash reflects a consistent subhistory of the operations that actually occurred. This subhistory includes all operations that completed before the crash, and may or may not include operations in progress when the crash occurred. The main tool for achieving durable linearizability for a concurrent data structure is the use of explicit instructions that force volatile cached data to be written to non-volatile memory. While such *persistence barrier* instructions enforce correctness, they also carry a performance cost and their use should be minimized: forcing data from caches into non-volatile memory can take hundreds of cycles.

An alternative, weaker condition is *buffered durable linearizability*. Informally, this condition guarantees that the state of the object following a crash reflects a consistent subhistory of the operations that actually occurred, but this subhistory *need not* include all operations that completed before the crash. Buffered durable linearizability is potentially more efficient than durable linearizability, because it does not require such frequent persistence fences. Unfortunately however, buffered durable linearizability is not compositional: the composition of two buffered durably linearizable data structures is not itself buffered durably linearizable.

When crashes occur during an execution, it is often difficult to tell which operations were executed and which operations failed to execute. Durable linearizability guarantees the completion of all operations that were executed before a crash but does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once.

We enable a more robust use of the queue, by defining a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The *log* queue provides durable linearization and detectable execution. If the program that uses the queue follows a similar procedure for detecting execution, then it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible.

Our main contribution is the proposal of three novel designs of durable concurrent queues. It is easy to obtain a durable linearizable queue by adding many persistence barrier operations automatically. But, in general, the obtained performance can be very low. We attempt to minimize the overhead and still achieve robustness to crashes. The first implementation, denoted *durable* queue, provides durable linearization. The second implementation, denoted *log* queue, provides durable linearization, as well as *detectable execution*. The third implementation, denoted *relaxed* queue, provides buffered durable linearizability with an implementation of a sync() operation.

We have implemented the three queue designs and measured their performance. As expected, implementations that

provide durable linearization have a noticeable cost. Interestingly, however, implementations providing detectable execution do not add significant overhead over durable linearization and may be worthwhile in this case. Also as expected, implementations that provide only buffered durable linearizability obtain good performance.

2 Measurements

We evaluated the performance of the proposed three queue implementations by comparing them one against the other and also against the original MS queue. We ran measurements on a 64-core machine, featuring 4 **AMD** Opteron(TM) 6376 2.3GHz processors, each with 16 cores. The machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). The operating system is Ubuntu 14.04 (kernel version 3.16.0).

Since the queue is not a scalable data structure, executions with many threads are not relevant and we only measured 1-8 threads. Each execution lasted 5 seconds. Memory management was handled with hazard pointers. In our implementation, we followed the hazard pointer scheme provided in [9]. Following [8], we evaluated the performance of the queue algorithms with a workload that lets several threads run enqueue-dequeue pairs concurrently. The queue is either initiated with 5 enqueued elements (for a small queue), or 1,000,000 enqueued elements (for a large queue).

We depict the difference in the throughput of the MS queue and our three new algorithms across different numbers of threads. Each test was repeated 10 times and the average throughput is reported. The x-axis denotes the number of threads, and the y-axis stands for millions of operations per second. A high number is better, meaning that the measured scheme has higher throughput. With hazard pointers (designed in [9]) the memory management overhead is large and the results of Figure 2 and Figure 3 are less indicative of the bare queue actual performance. This is why we also provide the measurements without memory management in Figure 1. As expected, queues that provide weaker durability guarantees perform better in most cases, with the exception being when the queue is very large and the garbage collection costs dominate performance. We believe that the reason for this is that large queues employ many hazard pointers and this cost is similar to all queue variants. In contrast, small queues can avoid some flushes. When the sync() function is infrequently called, the new head may pass the old tail between snapshots, reducing the required number of flushes, and eliminating most of the hazard pointer uses. Surprisingly, the relaxed queue performs better than the MS queue without garbage collection. We believe this is due to an implicit back-off effect that the slower queue creates.

We ran the relaxed queue and let each thread execute the sync() function every $K*N$ operations, where K varies between, 10, 100, 1000 and 10000 and N is the number of the threads. We omitted the $K = 10000$ results because they are similar to the $K = 1000$ results. As each of the N threads executes a sync every $K*N$ operations, we get that on average a sync is executed in the system after each thread executes K operations.

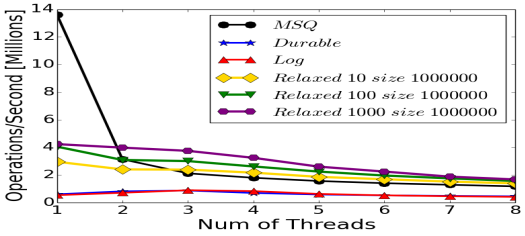

Figure 1. Throughput of the various queues with no object reuse.

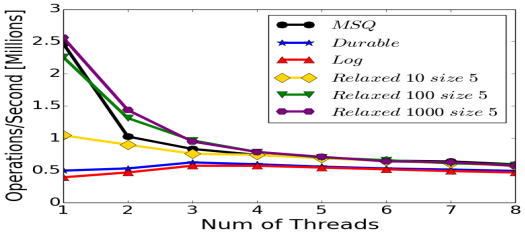

Figure 2. Throughput of the various queues with memory management. Initial size of the queue is 5.

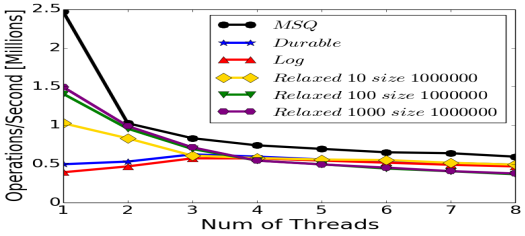

Figure 3. Throughput of the various queues with memory management. Initial size of the queue is 1,000,000.

References

[1] Friedman Michal, Herlihy Maurice, Marathe Virendra and Petrank, Erez *A persistent lock-free queue for non-volatile memory* PPoPP, 2018. https://dl.acm.org/citation.cfm?id=3178490

[2] Joseph Izraelevitz, Hammurabi Mendes and Michael L. Scott *Linearizability of Persistent Memory Objects under a Full-System-Crash Failure Model* DISC, 2016.

[3] Andreas Chatzistergiou, Marcelo Cintra and Stratis D. Viglas. *REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures*. PVLDB, 2015.

[4] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam and Sam H. Noh. *WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems* USENIX, 2017

[5] Chi Ping, Lee Wang-Chien and Xie Yuan. *Making B^+-tree Efficient in PCM-based Main Memory* ISLPED, 2014.

[6] Lulian Moraru, David G. Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz and Parthasarathy Ranganathan. *Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores* CMU-PDL-11-114, 2011.

[7] Venkataraman Shivaram, Tolia Niraj, Ranganathan Parthasarathy and Campbell Roy H. *Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory* USENIX, 2011.

[8] Maged M. Michael and Michael L. Scott. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms* PODC, 1996.

[9] Maged M. Michael. *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects* IEEE, 2004.

[10] Tudor David, Aleksandar Dragojevi, Rachid Guerraoui and Igor Zablotchi. *Log-Free Concurrent Data Structures* USENIX, 2018