POSTER: Reuse, don't Recycle: Transforming Algorithms that Throw Away Descriptors

Maya Arbel-Raviv
Technion - Israel Institute of Technology

Trevor Brown
University of Toronto

Abstract

Lock-free algorithms guarantee progress by having threads help one another. Complex lock-free operations facilitate helping by creating *descriptor* objects that describe how other threads should help them. In many lock-free algorithms, a new descriptor is allocated for each operation. After an operation completes, its descriptor must be reclaimed by a memory reclamation scheme. Allocating and reclaiming descriptors introduces significant space and time overhead.

We present a transformation for a class of lock-free algorithms that allows each thread to efficiently reuse a single descriptor. Experiments on a variety of workloads show that our transformation yields significant improvements over implementations that reclaim descriptors.

Keywords Concurrent data structures, Synchronization, Lock-free

1. Introduction

As core counts continue to rise in modern processors, applications' scalability is increasingly important. Designing scalable concurrent software is notoriously difficult, and programmers must rely on efficient concurrent library code. Concurrent data structures represent some of the most fundamental building blocks in these libraries.

Lock-free data structures are beneficial since they guarantee that some thread always makes progress, even if processes halt unexpectedly. This progress guarantee is typically achieved with *helping*. When a thread is blocked by an operation performed by another thread, it helps the other thread to make progress before continuing its own operation.

In complex lock-free data structures (e.g., [2, 3, 5]), threads publish descriptors for their operations, and helpers look at these descriptors to determine how to help. Existing lock-free algorithms that use descriptors fall into two categories: those that initially allocate a fixed number of descriptors and reuse them, and algorithms that allocate many descriptors throughout an execution and rely on garbage collection to reclaim them. We call the former *reusable descriptor* algorithms and the latter *throwaway descriptor* algorithms. Existing reusable descriptor algorithms use complicated ad-hoc techniques that are difficult to separate from the algorithms that use them. On the other hand, throwaway descriptor algorithms are faced with the limitations of lock-free memory reclamation algorithms, which introduce significant time and/or space overhead.

We introduce a transformation that takes a large class of lock-free throwaway descriptor algorithms and produces lock-free reusable descriptor algorithms with desirable properties. Throwaway descriptor algorithms that use k descriptors *per operation* are transformed into reusable descriptor algorithms that use k descriptors *per thread*.

We applied our transformation to lock-free double-compare-single-swap (DCSS) and k-compare-and-swap (k-CAS) primitives, and to a lock-free binary search tree. We compare our reusable descriptor implementations against throwaway descriptor implementations that use state of the art lock-free memory reclamation algorithms. Our results show that the reusable descriptor implementations perform at least as well as their throwaway descriptor counterparts, and *significantly* outperform them in some workloads.

2. Transformation

We start by describing the class of non-blocking algorithms to which our transformation can be applied. Each update operation allocates a new *descriptor* object, and fills it with information that describes any modifications it will perform. This information will be used by any threads that help the operation. (For example, if the non-blocking algorithm performs its modifications with a sequence of CAS steps, then the descriptor might contain the addresses, expected values and new values for the CAS steps.) An operation then "locks" an object by publishing a pointer to its descriptor in the object, typically using CAS.

PPoPP '17 February 04-08, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-4493-7/17/02. . . $15.00

DOI: http://dx.doi.org/10.1145/3018743.3019035

When a thread encounters a pointer ptr to a descriptor (for a different operation), it may decide to help the other operation by invoking a function $Help(ptr)$. If all accesses to fields of a descriptor occur inside *Help*, then it is straightforward to apply our transformation. Otherwise, some algorithm specific knowledge may be necessary.

In our transformation, each thread has a single descriptor of each type, and these descriptors are reused repeatedly. Conceptually, each of these descriptors has multiple versions. Each time an operation in the original algorithm would create a new descriptor, it instead reuses its single descriptor and increments its version. Whenever a thread tries to access another thread's descriptor, it must specify which version of that descriptor it is trying to access. If the version it specifies does not match the current version of the descriptor, then the thread fails to access the descriptor. Thus, whenever a thread accesses a descriptor, it always sees a consistent view of its contents. This introduces new semantics for reading and writing descriptor fields. In particular, these steps can now fail (due to a descriptor being reused).

Naturally, the original algorithm must be modified to take some special action when these steps fail. A failed step means that the operation being helped has already finished. Since descriptors are accessed only when a thread is helping another operation complete, the failed step means that operation *no longer needs help*, and the correct response is to stop helping.

We implement descriptor versions by storing a sequence number in each descriptor. To avoid the ABA problem with descriptor pointers, we represent a pointer to a descriptor by the concatenation of the thread name (which identifies the location of the descriptor) and the descriptor's current sequence number. Each time a process accesses a descriptor, it verifies that the sequence number has not changed. If it has changed, then the access fails.

3. Experiments

For brevity, we describe a small subset of our experiments. Our experiments were run on a 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads.

We compared our reusable descriptor implementation of k-CAS [4] to several implementations that use memory reclamation schemes. Specifically, we implemented k-CAS using a lock-free memory reclamation scheme called *hazard pointers* (HP) [7], and two blocking epoch-based reclamation schemes *DEBRA* [1] and read-copy-update (RCU) [6].

We ran a simple array-based microbenchmark. For each algorithm $A \in \{Reuse, DEBRA, HP, RCU\}$, array size $S \in \{2^{14}, 2^{20}, 2^{26}\}$ and k-CAS parameter $k \in \{2, 16\}$, we run ten timed *trials* for several thread counts n. In each trial, an array with fixed size S is allocated and each entry is initialized to zero. Then, n concurrent threads run for one second, during which each thread repeatedly chooses k uniformly random locations in the array, reads those locations, and then performs a k-CAS (using algorithm A) to increment each location by one.

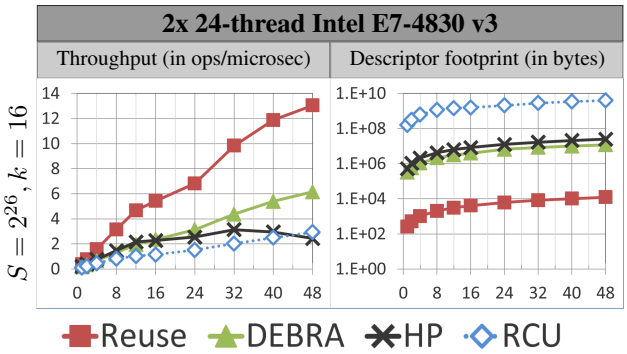

Figure 1: Results for the k**-CAS microbenchmark**. The x-axis represents the number of concurrent threads.

Due to the lack of space, Figure 1 shows only graphs for array size 2^{26} and $k = 16$. The left graph shows the throughput (total number of operation per microsecond). *Reuse* outperforms every other algorithm and its throughput is *2.2 times* that of the next best algorithm at 48 threads. The right graph shows the approximated descriptor footprint, i.e., the largest amount of memory occupied by descriptors at any point during a trial. Note that the y-axis is a logarithmic scale. The results show that *DEBRA* and *HPs* use almost three orders of magnitude more memory than *Reuse* at their peaks, and *RCU* uses nearly three orders of magnitude more memory than *DEBRA* and *HPs*.

Acknowledgments

This work was supported by the Israel Science Foundation (grant 1749/14), the Yad-HaNadiv foundation, the Natural Sciences and Engineering Research Council of Canada, and Global Affairs Canada. Maya Arbel-Raviv is supported in part by the Technion Hasso Platner Institute Research School.

References

[1] T. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *PODC '15*, pages 261–270.

[2] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *PPoPP '14*, pages 329–342.

[3] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *PODC '10*, pages 131–140.

[4] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC '02*, pages 265–279.

[5] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *SPAA '12*, pages 161–171.

[6] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[7] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6): 491–504, June 2004.