POSTER: State Teleportation via Hardware Transactional Memory*

Nachshon Cohen

Technion
nachshonc@gmail.com

Maurice Herlihy

Brown University
mph@cs.brown.edu

Erez Petrank

Technion
erez@cs.technion.ac.il

Elias Wald

Brown University
elias_wald@brown.edu

Abstract

State teleportation is a new technique for exploiting hardware transactional memory (HTM) to improve existing synchronization and memory management schemes for highly-concurrent data structures. When applied to fine-grained locking, a thread holding the lock for a node launches a hardware transaction that traverses multiple successor nodes, acquires the lock for the last node reached, and releases the lock on the starting node, skipping lock acquisitions for intermediate nodes. When applied to lock-free data structures, a thread visiting a node protected by a hazard pointer launches a hardware transaction that traverses multiple successor nodes, and publishes the hazard pointer only for the last node reached, skipping the memory barriers needed to publish intermediate hazard pointers. Experimental results show that these applications of state teleportation can substantially increase the performance of both lock-based and lock-free data structures.

1. Introduction

The emerging widespread availability of *hardware transactional memory* (HTM) [4] on commodity processors, such as Intel's x86 [5], provides an opportunity to rethink how we

design and implement concurrent data structures. This paper introduces *state teleportation*, a novel way to exploit hardware transactional memory to improve the performance of existing lock-based and lock-free data structures.

Many data structures, such as lists, skip-lists, B-trees, queues and hash maps, consist of a collection of nodes linked by pointers. Threads navigate through these links, sometimes adding or removing nodes from the structure. Highly-concurrent data structures must typically solve two interrelated tasks: *synchronization*, and *memory management*. For synchronization, we consider data structures that use either *fine-grained locking*, where individual nodes are locked and released, or *lock-free synchronization*, where locks are replaced by atomic instructions such as *compare-and-swap*.

For memory management, we consider data structures that use *lock-free navigation*, such as the *lazy list* [2, 3], where threads traverse the data structure without acquiring locks, but then validate the target nodes (using locks or atomic operations) before making changes. In these data structures, it is not safe to reuse a node immediately after it is unlinked from the data structure, because another thread may still be traversing that node. Instead, each traversing thread publishes the address it is about to read, a so-called *hazard pointer* [6], preventing other threads from recycling that memory. Hazard pointers are expensive, because a thread must execute a memory barrier each time a new node is traversed. Recently, various new memory reclamation schemes were introduced to reduce the overhead of hazard pointers, but those require special compiler and runtime support or require to present the algorithm in a special normalized form.

In highly-concurrent data structures, a high-level operation (such as adding a value to a hash set) is executed as a sequence of atomic state transitions, where each transition may require a memory barrier. Examples of such transitions include critical sections, atomic reads and writes, and read-modify-write operations such as compare-and-swap (CAS).

* This work was supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2012171 and by the National Science Foundation (NSF) grant 1331141.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '17, Feb. 4–8, 2017, Austin, Texas, USA..

Copyright is held by the owner/author(s).

ACM 978-1-4503-4493-7/17/02.

<http://dx.doi.org/10.1145/3018743.3019026>

Hardware transactional memory (HTM) [4] is a synchronization mechanism that has recently become available on commodity processors such as Intel’s x86 [5] architecture. HTM provides the ability for a thread to execute a sequence of machine instructions as an atomic transaction: the sequence appears to take effect instantaneously, without interleaving with other threads’ instructions. For the foreseeable future, however, HTM transactions will be *best-effort*: no transaction is guaranteed to commit. Transactions abort because of data conflicts, cache overflow, or cache associativity issues. As a result, HTM should be considered to be a potentially unreliable short-cut: any data structure or algorithm that uses HTM must have a fallback path that does not rely on HTM for progress.

State teleportation is a simple idea that exploits HTM’s short, best-effort hardware transactions to short-cut an operation’s state transitions. Starting from one state, a thread launches a hardware transaction that traverses multiple successor states, halting in a state that may not be directly reachable from the start. To an outside observer, that thread appears to have *teleported*, moving instantaneously from the starting state to the final state without visibly traversing the intermediate states.

In this paper we consider two applications of state teleportation. First, we show how state teleportation can be superimposed on a well-known fine-grained locking algorithm: *lock-coupling* [1] (or hand-over-hand locking). By teleporting a lock from one node to a non-adjacent node, we avoid the memory traffic produced by locking and unlocking the intermediate nodes. Next, we show how teleportation can be superimposed on two hazard pointer-based memory management schemes. By teleporting a hazard pointer from one node to a non-adjacent node, we avoid the memory barriers need to publish the intermediate hazard pointers.

2. State Teleportation

The heart of the state teleportation algorithm is the `teleport()` method. This method takes as an argument an *initial state*: an internal data structure node from which the navigation begins. This node is assumed to be protected by the chosen synchronization primitive, e.g., locked in lock-coupling algorithms or pointed by a hazard pointer for hazard pointer memory reclamation. The `teleport()` atomically reaches a node which is closer to the target node (on which the operation should be applied); the returned node is protected by the chosen synchronization primitive, while the initial node is released. `teleport()` employs HTM to avoid the synchronization overhead of traversing intermediate nodes.

Due to the best-effort nature of HTM, it is not possible to assume that a teleportation of any fixed number of nodes would succeed (not even one). On the other hand, it is desirable to teleport over several nodes as this reduces the overhead of synchronization. Toward this end, the `teleport()` method dynamically adjusts the attempted

teleportation distance. Every time a transaction succeeds, the thread increases the attempted teleportation distance by one. When a transaction fails, the teleportation distance is halved. However, we do not let the distance fall below a threshold, because short teleportation distances harm performance. When too many failures occur, the algorithm reverts to standard traversal.

2.1 Evaluation

We measure the effectiveness of teleportation on three linked-list data structures: lock-coupling, lazy, and lock-free. All measurements were done on a 8-core Broadwell processor with hyperthread enabled (total 16 hardware threads). Figure 1 provides the teleportation speedups when applied to hand-over-hand locking and when applied to the hazard pointers with lazy list and lock-free list.

For the hand-over-hand implementation, teleportation yields an an approximate 1.5-fold speedup for a single thread, increasing to a 4-fold speedup at 16 threads. The teleportation scheme is able to avoid most overhead associated with repeated acquiring and releasing the locks. The effect is especially dominant when hyperthreading is enabled (more than 8 threads). For the hazard pointer scheme, teleportation improves the performance by 21% – 65%, while keeping the strong progress guarantee provided by hazard pointers.

Figure 1. Effects of teleportation on various lists. The bars show the ratio between original list and teleportation. Higher means teleportation is more effective.

References

- [1] BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on b-trees. *Acta Informatica* (1977).
- [2] HARRIS, T. A pragmatic implementation of non-blocking linked-lists. In *DISC’01* (2001).
- [3] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., III, W. N. S., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *OPODIS’05*.
- [4] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA’93*.
- [5] INTEL CORPORATION. Transactional Synchronization in Haswell, 2012.
- [6] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.* (2004).