Distributed Transactional Boosting

Michael Feldman   Maurice Herlihy

Department of Computer Science

Brown University

May 2010

Abstract

We describe a methodology of transforming a large class of linearizable data structures into transactional data structures replicated by a set of networked computers. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent, distributed transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

1 Introduction

As put forth in "Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects" by Maurice Herlihy and Eric Koskinen, transactional boosting is a methodology of transforming linearizable data structures into transactional data structures. Under this methodology, each thread keeps its own copy of the linearizable data structure. Threads update their local copies, and group updates into transactions. When a thread has finished a transaction, it commits, notifying all other threads of that transaction. A system of abstract locks detects conflicts between committing and ongoing transactions. Any ongoing transaction that conflicts with a committed transaction is first undone before the committed transaction updates that thread's data structure.

This methodology works well for multicore systems, but new issues arise in distributed systems. Compare-and-swap operations become difficult, and the primary speed bottleneck is passing messages over the network instead of contention. Still, it is possible that the ideas involved in transactional boosting will provide an efficient way to maintain a distributed data structure.

Our implementation of distributed transactional boosting is built atop Virtual Synchrony, a project originally intended to maintain a distributed hash table using absolutely-ordered broadcast messages over RMI. Virtual Synchrony takes care of adding nodes to the group, broadcasting messages, and deciding upon a coordinator to order the messages.

The DistributedDataStructure class contains all the logic for maintaining a data structure as a black box. Users may create a DistributedDataStructure that wraps any class that extends java.util.Collection. Users first begin a transaction and take an abstract lock (represented by an integer). It is through these abstract locks that conflicts between transactions are detected. Next, they pass in a pair of methods and their arguments. The first method is applied to the data structure and its result is returned. The first method is stored on a redo list, which will be sent to all other distributed nodes when the current transaction commits. The second method is stored on an undo list, which may be used if the current transaction aborts.

Committing a transaction is a two-step process. First, the node broadcasts a lock request message containing all abstract locks that will be used in its transaction. All other nodes reply to

this message, and if any node was in the process of committing a transaction before the first node sent its lock request message, the first node aborts its commit. If there are no conflicts, the first node broadcasts its redo list to all other nodes, which apply it to their copies of the distributed data structure.

If at any point a node receives a transaction that conflicts with an abstract lock currently held by that node, the node that received the transaction aborts. The node first uses its undo list to undo its current transaction. It then applies the incoming transaction and sets a flag. All DistributedDataStructure methods check this flag before executing, and throw an AbortedException if there has been an abort since the last method call.

Synchronization is more difficult in the distributed version of transactional boosting. In multicore transactional boosting, each thread maintains its own undo and redo lists, which are modified only by that thread. However, using Virtual Synchrony, each node has one or more user threads, a thread to send messages, and a thread to receive messages. Even if the underlying data structure, the undo and redo lists, and the local lock set were all concurrent data structures, synchronization would still be necessary in order for all of these data structures to be kept in a consistent state. For example, concurrent calls to apply() and undoTransaction() would not be linearizable, and could result in a difference in the number of methods on the undo and redo lists, therefore leaving the data structure in an inconsistent state. The data structure is never modified without modifying the undo and redo lists, so we might as well be more flexible with which data structures we allow to be used with this methodology.

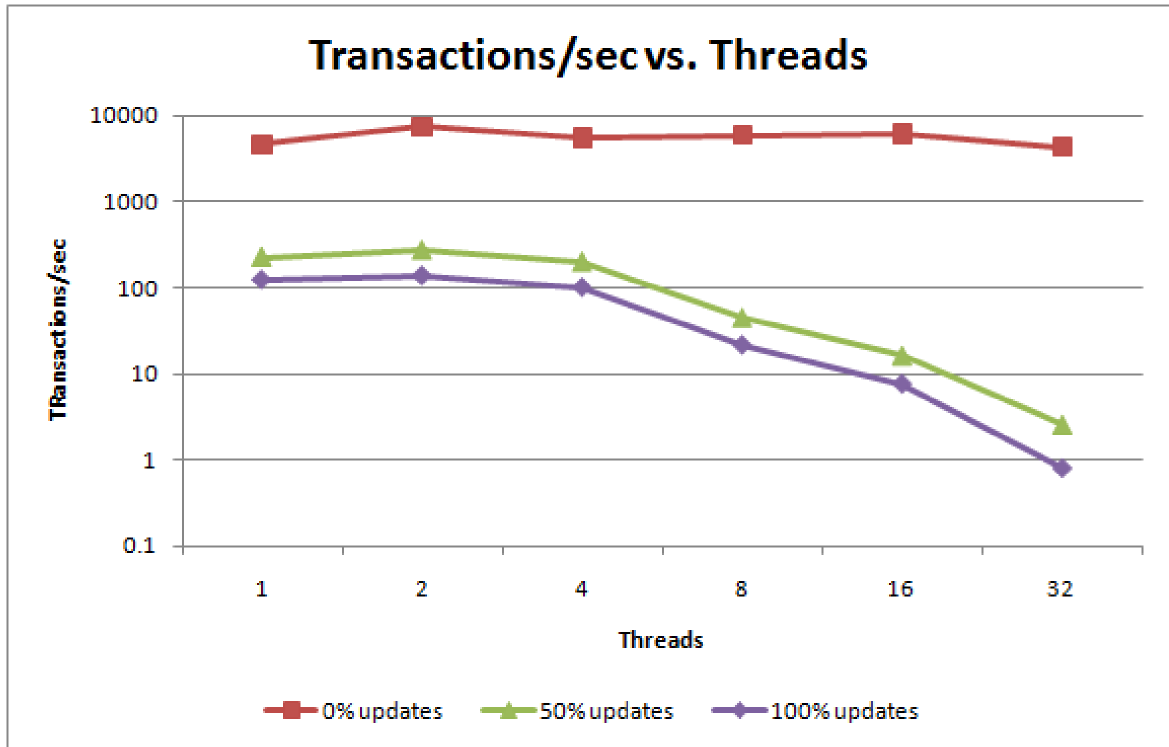
2 Example

As evidence f correctness, a series of DSTM2 (Dynamic Software Transactional Memory) benchmarks were run. DSTM2 benchmarks are intended for software transactional memory systems, not distributed systems. As far as I could tell, there is no set of standard benchmarks for a distributed system such as this one.

2.1 java.util.SortedSet

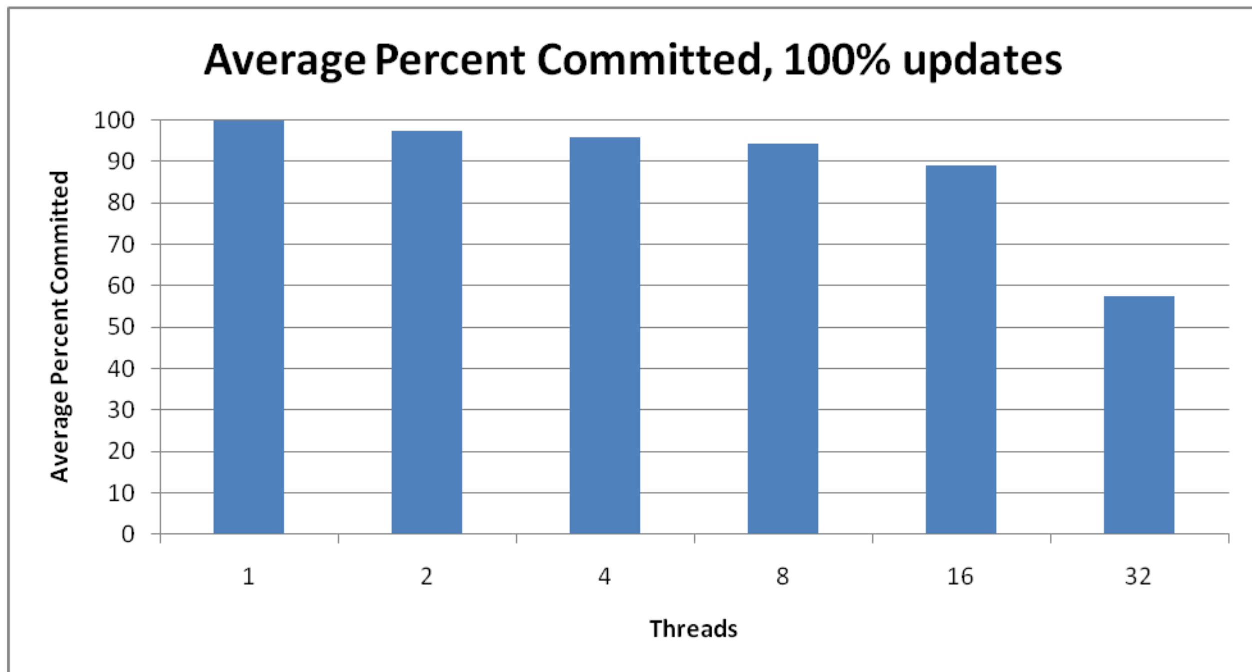
The following chart shows the average results of ten benchmarking runs on a computer with an AMD Phenom 9950 Quad-Core Processor clocked at at 2.6 GHz, running Debian. In order to minimize the randomness associated with networks and for ease of setting up the benchmark, all nodes were run as separate threads on this one computer. The number of threads

is actually the number of user threads, as each user thread also has associated sending and receiving threads.



As would be expected, purely contains transaction greatly increase from 1 to 2 threads. They are completed locally - since each node has its own copy of the data structure, there is no need to even go through the process of starting or committing a transaction.

As the number of threads increases past the number of cores on this computer, throughput decreases. Contention for the limited number of abstract locks increases. It is increasingly likely that a single node's threads will be descheduled during a commit. The longer a thread spends between broadcasting its locks and sending out its transactions, the more likely it is that it will cause another thread to abort a transaction.



This graph shows the percentage of transaction that were successfully committed, as opposed to transactions that aborted mid-completion due to a lock conflict. While aborted transactions do play a part in reducing the throughput at all amounts of threads, aborted transactions only have a significant effect at 32 threads. This shows it may be that this computer simply slows down when it has to handle the overhead of so many more threads than it has processors, and contention due to the design of the system is actually somewhat less common than what the first graph shows.

4 Conclusions

Transactional boosting in distributed systems is effective under certain circumstances. For mainly read-only operations, this methodology is extremely successful. Since each node maintains a complete copy of the data structure, read-only operations can have direct access to the data structure.

Concurrent updates that do not conflict with one another are also fairly efficient. While updates cannot be done fully concurrently, they may still overlap while in transit. In distributed systems, waiting for messages to cross the network is typically the bottleneck. By allowing multiple updates to be on the network simultaneously, some concurrency is achieved.

Concurrent updates that do conflict are a major bottleneck for this methodology. If a node is not the first to commit, it may take some time for the update to receive the transaction.

During that time, it will be doing work that will take an equally long time to undo when the conflicting transaction is received.

4.1 Future Directions

It is possible for some enhancements to be made to this implementation. It may be possible to have fully concurrent updates on highly-concurrent data structures. Multiple sending and receiving threads would be difficult to synchronize properly, but could increase throughput in situations with many updates and few conflicts. Lock request acknowledgments could be sent directly to the requesting node and not abcast (absolutely-ordered broadcast) to the entire group.

5 Acknowledgements

Thank you to Robert Mustacchi for providing a working implementation of Virtual Synchrony, as well as the idea that that some put() calls may be forced to block until they were locally delivered.

This work will become part of Irina Calciu's PhD thesis. Good luck to her in her research and classwork.

6 References

Herlihy, M. and Koskinen, E. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. http://www.cs.brown.edu/~ejk/papers/boosting-ppopp08.pdf, 2008.