Thanks for the introduction. Good Morning Everyone. I am Deli from University of Central Florida and I am presenting Lock-free Transactions without Rollbacks for Linked Data Structures. This is a joint work with my advisor Damian Dechev. There are many lock-free data structures, and we’ve seen a lot of them being compiled into concurrent libraries. They guarantee linearizablity for individual method invocation, but the problem is that user cannot safely compose multiple operations together.

Here I show a simple Move function that deletes a key from set A and insert it into set B. In sequential scenario, this would be trivially correct. But in concurrent execution, the thread might be interrupted righted after deleting the key from set A, leaving the two sets in an inconsistent state. Even with only one container such composition is not possible without additional synchronization. In the second example, we try to compute some value and insert the new key-value pair into the map only if the key does not exist. The thread may get interrupted after the condition check and wrongly overwrite an existing value in the map. A study conducted by Shacham et al. showed that many programmers are not aware of this issue, which causes concurrency bugs when using the Java concurrent library.

In order to build reusable software in a modular fashion, we need to support transactions in concurrent data structures. Transactional data structures should execute a sequence of operation atomically and in isolation. Atomicity requires that if one operation fails, the entire transaction should abort. Isolation requires that concurrent execution of transactions appears to take effect in some sequential order that respect real-time ordering. Now, let’s review some of the applicable approaches.

Given a sequential data structure, we can easily apply transactional memory, such that the operations executed within a transaction are guaranteed to be atomic. However, since STM instruments read and write memory accesses, it has large runtime overhead. Moreover, STMs cause excessive aborts because low-level memory access conflicts do not necessarily correspond to data structure level semantic conflicts. The graph here illustrate such a case. Thread 1 and 2 try to insert key 4 and 1 respectively into a linked list. Any good concurrent linked list would allow these two operations to proceed concurrently because they write to disjoint memory locations. But they do have read/write conflict on node 0 so an STM has to abort one of them.

Transactional boosting proposed by Herlihy and Koskinen can be used obtain transactional data structures from linearizable base data structures. Each operation needs to acquire an abstract lock before execution. The abstract locks are designed based on the knowledge of commutativity of data structure operations, so they ensure non-commutative operations will never occur concurrently. Transactional boosting provides significantly better performance than STMs, but the use of locks degrades the progress guarantee when it’s applied to lock-free data structures. Also, as shown in the graph, when a transaction aborts due to failed lock acquisition, boosted data structures need to execute the inverse operations of partially executed transaction in order to restore the abstract state. This causes additional overhead.

Golan-Gueta et al. proposed automatic semantic locking. Given a piece of code on the left that needs to be executed atomically, lock acquisition code is inserted through static analysis. Similar to transactional boosting, they also use abstract locks so it is blocking and need additional data structure to store all the locks. Unlike transactional boosting their approach acquire all locks prior to the execution of any operation. So no rollback is needed.

The challenges for supporting lock-free transactions are that we need to effectively buffer write operations so that they are not visible to operation outside the scope of the active transaction; and to efficiently rollback failed transactions to restore data structure consistency. We present Lock-free transactional transformation, a methodoly to transform lock-free linked data structures into lock-free transactional data structures. Our key contributions are that we introduced the first lock-free semantic conflict detection; a logical status interpretation procedure to eliminate rollbacks; and cooperative transaction execution to minimize aborts. Currently we can apply to this methodoly to the set abstract data type and linked data structures such as list, skiplist and trees.

Our approach uses node-based conflict detection. As the graph shows, for each node we embed a NodeInfo structure which serves as a monitor for that node. It stores an operation id and a reference to a transaction descriptor, which is shared among all nodes participating in the same transaction. A transaction descriptor contains an array operation type and operand as well as a transaction status. The status can be committed, aborted, or active. The concept here is that before modifying a node, a thread must first read the contents in the transaction descriptor to make sure that the previous transaction is not active. This prevent concurrent access to the node, which is equivalent to preventing non-commutative operations on the same key in a set.

The workflow of a transformed Insert function will look like this. The grey blocks are extracted from the base data structure. We introduced a new code path to update the NodeInfo structure on the new node. For Delete and Find function, we don’t need to insert new node, so they will directly return false when failed to locate the node with K.

In the newly introduced UpdateNodeInfo function, we first test if NodeInfo is bit marked. If so, we use the delete function from the base data structures to physically remove the node. Next, it helps finishing up the previous transaction if it is still active. This enforces serialization so that the current thread observes the final results of the previous transaction. We then test if the key is logically present based on transaction status and operation type, which will be detailed in the next slide. For Insert function, we only update the NodeInfo structure using CAS if the key is not logically present.

Here is the truth table for interpreting the existence of a key. For an Insert operation, a committed transaction means the key exist; while an aborted transaction means the key does not. The key is also considered present for an active transaction only for operations within the same transaction. For delete operations the interpretations are opposite. For Find operations the key is considered present for all cases because it is a read-only operation.

So, in the transformed data structure all operations are invoked by creating and filling a transaction descriptor and passing it to the transaction execution function. This function invokes the operations in sequence and only continues if the previous operation is successful. After that it commits the transaction by atomically updating the transaction status to Committed or Aborted depending on if all operations completes successfully. If the transaction is successful, it also needs to bit mark all the NodeInfo structures on the deleted nodes and then physically remove them. There are a few subtle things to note. Very rarely the threads may stuck in an endless cycle of helping if there is a cyclic dependency among operations from different transactions. We use a HelpStack and push the current transactions descriptor before set out to help others. Cyclic dependencies are detected as duplicate descriptors in the stack and we abort one of them. Also, if we disable the helping mechanism the transaction execution becomes obstruction-free. In this case, we don’t need to store the operation context in the descriptor and may use many exiting contention management scheme such as aggressive, polite, or karma.

Here is an example of how LFTT works. We have three transactions. Thread 1 finishes t1 and inserted node 1 and 3. Thread 2 inserted node 4 and is working on node 2. Now thread 3 deleted node 3 and tries to delete node 4 concurrently. It sees that node 4 is still being accessed by an active transaction, so thread 3 will help executing the remaining operations in t2 before replacing the NodeInfo on node 4. After t3 commits, key 3 and 4 will be considered absent from the set by subsequent operations.

In our performance test, we use a 64-core NUMA system with 4 AMD cpus. For each data structure we test them with three types of work-load and with transactions sizes from 1 to 16. We compare our transactional skiplist with Frasers’s object-based STM and a boosted skiplist. We also compare our transactional linked list with the list in the RSTM using Norec backend and with a boosted linked list.

Due to limited time, I only show the result for mixed work load. The top half of graph shows the throughput; higher is better. The total number operation is measured by the product of number of operations in a transaction and number of committed transactions. For 2 operations per transaction, the throughput increases as the number of thread increases. As the transaction size increases, we see the scalability of all approach decreases. Because larger transaction has longer execution window, and are more likely to abort due to failed operations. Overall, our approach is on average 60% faster than boosted skiplist, and 3 times faster than the object-based STM. Also it experiences no spurious aborts. Spurious aborts are the number of aborts not counting ones that caused by failed operations. It is a measure of efficiency of the contention management schemes.

For transactional linked list, we see similar scalability trends. Our approach achieves an average of 40% speed up over boosted list and 3 to 10 times speedup over the word-based STM. It also produces 2 to 3 order of magnitude less spurious aborts.

To summarize, LFTT allows for built-in transactions for lock-free sets. It excels at large transactions and has greater success rate with minimal spurious aborts. We are working on supporting map data type as well as wait-free data structures. A preliminary version of our transactional data structure library that supports cross-container transaction is available on our lab website.

TODO: not physically removing nodes, e.g., for skiplist