Brief Announcement: A General Technique for Non-blocking Trees

Trevor Brown[1], Faith Ellen[1], and Eric Ruppert[2]

[1] University of Toronto, Canada
[2] York University, Canada

We introduce a template that can be used to implement a large class of non-blocking tree data structures efficiently. Using this template is significantly easier than designing the implementation from scratch. The template also drastically simplifies correctness proofs. Thus, the template allows us to obtain provably correct, non-blocking implementations of more complicated tree data structures than those that were previously possible. For example, we use the template to obtain the first non-blocking balanced binary search tree (BST) using fine-grained synchronization.

Software transactional memory (STM) makes it easy to turn sequential implementations into concurrent ones, but non-blocking implementations of STM are currently inefficient. Tsay and Li [6] gave a general approach for implementing wait-free trees using LL and SC primitives. However, their technique severely limits concurrency, since it requires every process accessing the tree (even for read-only operations) to copy an entire path of the tree starting from the root. For library implementations of data structures, and applications where performance is critical, it is worthwhile to expend effort to get more efficient implementations than those that can be obtained using these techniques.

The tree update template. We can implement any down-tree, which is a directed acyclic graph with indegree one. Each update operation on the data-structure replaces a contiguous part of the tree with a new tree of nodes (which can point to nodes that were previously children of the replaced nodes). As with all techniques for concurrent implementations, our method is more efficient if the operations are smaller. For instance, an insertion into a (leaf-oriented) chromatic tree is performed by one update operation to replace a leaf by a new internal node and two new leaves, followed by a sequence of rotations, each of which is an update operation. If an operation has to access several nodes, implementing it in a non-blocking way requires synchronization among processes. Our template takes care of all process coordination, so provably correct implementations can be obtained fairly mechanically.

The template uses the recently introduced LLX and SCX primitives [2], which are extended versions of LL and SC that can be efficiently implemented from CAS. At a high level, an update operation built using the template consists of a sequence of LLXs on a small set V of nodes, the creation of a (typically small) tree of new, replacement nodes, and an SCX, which atomically swings a pointer to do the replacement, only if no node in V has changed since the aforementioned LLX on it. If the SCX successfully swings the pointer, it also prevents any removed nodes from undergoing any further changes.

The operations of a data structure built using the template are automatically linearizable. Moreover, simply reading an individual field of a node always returns the result of the most recently linearized operation that modified the field. Some queries can be performed efficiently using only reads. For example, in a BST where the keys of nodes are immutable, a search can follow the appropriate path by simply reading keys and child pointers, and the value in the node reached. Such a search is linearizable even if concurrent update operations occur along the path. The implementation of LLX/SCX also provides an easy way to take a snapshot of a set of nodes. This facilitates the implementation of more complex queries, such as successor or range queries. The template allows some update operations to fail, but guarantees that update operations will continue to succeed as long as they continue to be performed. Moreover, if all update operations in progress apply to disjoint parts of the tree, they will not prevent one another from succeeding.

Implementing non-blocking balanced BSTs. Chromatic trees [4] are a relaxation of red-black trees which decouple rebalancing operations from operations that perform the inserts and deletes. Rebalancing operations can be performed in any order, and can be postponed and interleaved freely with operations that perform the inserts and deletes. Amortized O(1) rebalancing operations per insert or delete are sufficient to rebalance the tree into a red-black tree. Using the template, we implemented a non-blocking chromatic tree. The height of a tree containing n keys is $O(\log n + c)$, where c is point contention. Our implementation is easily described in 10 pages, and rigorously proved in five pages. In contrast, a non-blocking implementation of a B+tree (a considerably simpler sequential data structure) is described in 30 pages and proved correct in 33 pages [1].

We are presently working on relaxed versions of (a, b)-trees and AVL trees. Performance experiments (available in the full paper, at http://www.cs.utoronto.ca/~tabrown) indicate that our Java implementations rival, and often significantly outperform, highly tuned industrial-strength data structures [3, 5].

References

1. A. Braginsky and E. Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.
2. T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. 32nd ACM Symp. on Principles of Distr. Comput.*, 2013.
3. D. Lea. Java's `java.util.concurrent.ConcurrentSkipListMap`.
4. O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.
5. M. Spiegel and P. F. Reynolds, Jr. Lock-free multiway search trees. In *Proc. 39th International Conference on Parallel Processing*, pages 604–613, 2010.
6. J.-J. Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Proc. Int. Conf. on Parallel and Distributed Sys.*, pages 544–549, 1994.