Concurrent Data Structures

Faith Ellen
Department of Computer Science
University of Toronto, Canada
faith@cs.toronto.edu

Trevor Brown
Department of Computer Science
University of Toronto, Canada
tabrown@cs.toronto.edu

Data structures are an important component of efficient and well-structured programs. In shared memory distributed computing, correct data structures are difficult to construct because concurrent accesses by different processes can conflict with one another. One simple approach is to use a global lock to restrict access to one process at a time. But this can severely affect performance. This talk will present a survey of some of the interesting techniques that have been developed to build efficient concurrent data structures. It will also discuss how we think concurrent data structures should be evaluated.

1. TECHNIQUES

One natural technique is for a process to only lock a small portion of the data structure that is currently being accessed. This is known as fine grain locking. An example of this is hand-over-hand locking [2, 19]. However, such data structures are not fault tolerant. If a process crashes while holding a lock, part of the data structure may become inaccessible.

Another approach is to build lock-free (non-blocking) or wait-free data structures. These ensure that some processes or all processes can always complete an operation on the data structures, no matter how the processes are scheduled. However, there typically needs to be some way for processes accessing the same part of a data structure to synchronize with one another. For example, nodes that are going to be deleted can be *marked* so that no other processes will later modify these nodes [14] or auxillary nodes can be added between real nodes be prevent conflicts between updates of adjacent nodes [25].

For complicated updates that involve changing multiple pointers, processes must record information about the updates they are performing, so that other processes can help complete the updates, in case the processes are slow or crash. Nodes involved in such an update can be *flagged* and store information in specially dedicated fields or store a pointer to an information node describing the update [12, 10].

PODC'16 July 25-28, 2016, Chicago, IL, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3964-3/16/07.
DOI: http://dx.doi.org/10.1145/2933057.2933123

If an operation takes effect by applying a single atomic primitive, there is no need to record information about the update, because it is never interrupted. Atomic multi-word primitives (such as multi-word compare&swap) can simplify the implementation of complicated updates. However, such primitives are typically not available. An alternative is to use lock-free linearizable implementations of more powerful primitives from weaker primitives such as compare&swap [7]. The key is to choose powerful primitives that are well-suited for implementing data structure operations, but are not so powerful that they preclude efficient implementations.

Transactional memory enables concurrent data structures to be built very easily: simply put each operation inside a transaction [16]. However, the high overhead of software transactional memory means that the resulting data structures are inefficient. Using hardware transactional memory reduces the overhead, but does not provide any progress guarantees. Therefore, data structures built using hardware transactional memory must provide a software fallback path. To allow the hardware and software paths to run concurrently (avoiding concurrency bottlenecks), hardware transactions must be instrumented, which adds significant overhead. With two hardware paths, one instrumented and one not, together with a software fallback path, high efficiency can be obtained without introducing concurrency bottlenecks [5].

It is typically easier to implement a data-structure with weak progress guarantees, for example, obstruction-freedom. However, there are techniques that can automatically transform (a class of) implementations to have stronger progress guarantees [11, 13, 24].

Instead of trying to implement an efficient sequential data structure in a concurrent setting, it is sometimes better to implement a version having relaxed structural properties. For example, with lazy deletion, a node is marked to logically delete it from a set and may be physically deleted later [14, 4]. In a chromatic tree, updates (such as rotations), which are used to help rebalance the tree, are decoupled from the insertion and deletion of nodes and can be freely interleaved with other updates [3, 8]. Breaking up a big operation into a number of smaller pieces can be especially beneficial for implementations using hardware transactional memory, because there are capacity limits that cause transactions accessing too many memory locations to fail.

Alternatively, one can relax the definition of the abstract data type to be implemented. A pool (bag) maintains a set which supports an operation that returns an an *arbitrary* element, rather than a specified element (such as the element that was inserted earliest or latest). For some applications where a queue or stack is employed, a pool is sufficient and can be implemented more efficiently [17, 23, 1]. Other examples of relaxed concurrent data types can be found in [22].

An atomic snapshot of a data structure is useful for iterating through the data structure, for example, when performing a range query or for generating a consistent backup copy of the data structure. A *persistent data structure*, in which each update creates a new version of the data structure without destroying previous versions, automatically provides atomic snapshots. However, it can use a lot of space, even when large parts of the data structure are shared by consecutive versions. For example, in a persistent binary search tree, each update creates a copy of all the nodes it changes as well as all ancestors of these nodes. Other implementations are more space efficient [4, 21, 20].

For sequential and lock-based data structures, memory reclamation is easy. For lock-free and wait-free data structures, it can be difficult to determine when a node can safely be freed (and then reused) after it has been deleted from the data structure. Before a process can free a node, it must know that no other process is currently accessing the node or can reach it (by following a sequence of pointers). Hazard pointers [15, 18] provide a very popular memory reclamation scheme for use with concurrent data structures. However, when a deleted node can point to other deleted nodes, as in the chromatic tree, hazard pointers cannot be used. There are alternative memory reclamation schemes for such data structures (surveyed in [6]), but they are significantly more complicated. Because many concurrent data structures involve a lot of copying and, thus, create a lot of garbage, memory reclamation can have a significant impact on their performance. Thus, choosing (or possibly constructing) a good memory reclamation scheme is an important component of building a concurrent data structure.

2. EVALUATION

Papers about concurrent data structures range from the presentation of new algorithmic techniques and the design and analysis of new data structures to experimental comparison of different data structures and algorithmic engineering.

If the focus of the paper is algorithmic, the work must include rigorous proofs of correctness. Researchers should not rely on high level proof sketches. Concurrent data structures can be sensitive to seemingly insignificant reordering of operations and some errors are quite subtle. Unfortunately, there are numerous examples of incorrect concurrent data structures that have been published (or would have been published, but for a dedicated expert reviewer). It is the responsibility of researchers to ensure that their data structures are correct. Verification tools are being developed, but, so far, they are difficult to use and have only been applied to relatively simple concurrent data structures.

Complexity analyses of concurrent data structures should also be provided. For wait-free algorithms, determining the worst-case step complexity is usually straightforward and is often part of the proof of wait-freedom. However, lock-free data structures require an amortized or average-case analysis. This is typically much more involved [9]. As is the case for papers about sequential data structures, an interesting analysis (and, especially, new analytical techniques) can be an important contribution of a paper.

If the focus of the paper is experimental, the experiments should be reproducible and, ideally, vetted (for example, through artifact evaluation, as is done in some systems conferences). It is important that experiments are measuring the right aspects of concurrent data structures. For example, one concurrent data structure may appear to be more efficient than another, but, in reality, the differences can be due to small optimizations, better coding, or a different memory reclamation scheme. Even when two different data structures are evaluated using the same experimental setup, on the same hardware, using the same operating system and the same memory reclamation scheme, there may be significant overhead in the experimental setup that masks substantial differences in their performance. For example, if an operation takes 1 millisecond and another operation takes 1 microsecond, but the overhead is 1 millisecond, then the first operation appears to be only a factor of 2 slower, rather than a factor of 1000 slower. Even more troubling, a set of experiments performed on two concurrent data structures on the same machine can lead to opposite conclusions when performed on a different machine.

When evaluating concurrent data structures, such as balanced search trees, which are designed for good worst-case behaviour, it is insufficient to only consider randomly generated input data. Since binary search trees are likely to remain relatively balanced under random sequences of insertions and deletions, the overhead of rebalancing is wasted. Naturally, unbalanced binary search trees perform significantly better in these experiments.

Even when one wants to compare the average case performance of concurrent data structures, one has to be careful how one chooses the data. For example, for multi-sets (which can contain multiple copies of the same key), using different proportions of insertions and deletions might lead to data structures that become empty or grow without bound. However, this is provably not the case for sets.

Comprehensive experiments that fairly evaluate a variety of concurrent data structures can be very valuable. Experiments should be chosen to not only present the relative performance of different concurrent data structures, but provide some understanding about *why* one performs better than another under various conditions. For example, one might use experiments to determine when it is faster to replace a small subtree with a new copy rather than update information in that subtree. It is also interesting to compare concurrent data structures using real-world data to determine which is best for a particular application or class of applications and which optimizations are useful.

Papers which describe a slightly new concurrent data structure (for example, using known techniques to obtain a small variation of an existing data structure), together with experiments on which it performs better than some other existing data structures, are not worthy of publication in top conferences and journals. Unfortunately, many such papers are submitted.

On the other hand, a paper with a new and interesting algorithmic idea, but with performance results that are comparable to (or slightly worse than) existing data structures (or even with no experiments), should not be automatically rejected "since it isn't practical." Other researchers may be able to build on the approach or combine it with other existing approaches to engineer a very efficient data structure, possibly for a specific application.

Acknowledgments

This work was supported by by the Natural Science and Engineering Research Council of Canada.

3. REFERENCES

[1] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. *Distributed Computing*, 26(4):243–269, 2013.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inf.*, 9:1–21, 1977.

[3] J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[5] T. Brown. Brief announcement: Faster data structures in transactional memory using three paths. In *Proc. 29th International Symposium on Distributed Computing (DISC)*, pages 671–672, 2015.

[6] T. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proc. 34rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 261–270, 2015.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 13–22, 2013.

[8] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, 2014.

[9] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 332–340, 2014.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2010.

[11] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proc. 19th International Conference on Distributed Computing (DISC)*, pages 78–92, 2005.

[12] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, 2004.

[13] G. Giakkoupis, M. Helmi, L. Higham, and P. Woelfel. An O($\sqrt{n}$) space bound for obstruction-free leader election. In *Proc. 27th International Symposium on Distributed Computing (DISC)*, pages 46–60, 2013.

[14] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[15] M. Herlihy, V. Luchangco, P. A. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2012.

[17] U. Manber. On maintaining dynamic information in a concurrent environment (preliminary version). In *Proc. 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 273–278, 1984.

[18] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[19] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications.* Chapman and Hall/CRC Press, 2004.

[20] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Proc. 27th International Symposium on Distributed Computing (DISC)*, pages 224–238, 2013.

[21] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 151–160, 2012.

[22] N. Shavit and G. Taubenfeld. The computability of relaxed data structures: Queues and stacks as examples. In *Proc. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 414–428, 2015.

[23] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory Comput. Syst.*, 30(6):645–670, 1997.

[24] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 357–368, 2014.

[25] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222, 1995.