WFPO Reads via Persistence in Pointer-Based Data Structures

Kyle Thompson
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
kylejthompson@cmail.carleton.ca

December 15, 2016

1 Introduction

One fundamental problem if the field of parallel data structures is ensuring that the multiple threads reading from and writing to a data structure do not interfere with one another. For instance, what would happen if one thread read from some data as it was being modified by another? The most prevalent solution to this is to use locks wherein a single thread gains exclusive access to a data structure while it is using it. There are a few different variations of this concept like a read-lock where multiple threads can read from a structure at once but even with this variation it is not possible to have more than one thread interacting with the structure when one is writing. This can be especially costly when modification operations can take long times to complete, leaving the reading threads just waiting for the lock to open.

This paper presents a novel approach to solving this problem by augmenting the pointer based data structure in question to a partially persistent version of itself, allowing Wait-Free Population Oblivious reads.

2 Literature Review

Partial Persistence

Partially persistent in data structures is the ability to query the state of the structure at a particular point in the past. The benefit this serves us is that these previous states are immutable. Regardless of any operation that happens at time t, the state at $t-1$ is unchanged. We will see in section 3 how this property can be leveraged for WFPO reads.

The process by which we can make arbitrary pointer based data structures persistent is demonstrated in [1]. Essentially each node in the data structure can be augmented with extra pointers and back pointers along with timestamps. The back pointers point to all nodes which point to it and the extra pointers are used to handle updating normal pointer (a pointer that exist in the non-persistent data structure's nodes) values after they have been set. For example, in a red-black tree, if a node's right child is deleted, that pointer must be updated. Rather than changing it though a new pointer with the correct new right value is marked as being a right pointer, assigned an up-to-date time stamp and is added to

the extra pointers. When querying in a particular time then, simply follow the path with the largest times that are no greater than the time being searched in. In the event that a node runs out of space in it's extra pointers when an update occurs, the node is simply copied, bringing only the most up-to-date information to the copy. This means the new node will have an empty extra pointer array. It also means that other nodes that pointed to the old copy will have to add a new pointer to this node. Note that the addition of a new pointer in the preceding nodes can cause them to require a copy too. For this reason it is important to choose a data structure where on any operation not too many pointers get changed. In our example with the red-black tree this could cause copies from a leaf all the way to the root. Despite its poor worst case though, because only $O(1)$ pointer changes occur during any particular update this still amortizes to constant space per single update, or linear space overall.

Other forms of persistence

Relevant to the discuss in section 4 are other forms of persistence. Beyond partial where it is possible to query the past there is full persistence where it is also possible to modify the past. In this model of time travel any modification in the past will result in a new, branching time line. Beyond that is confluent persistence which is the same as full, only with the ability to merge two distinct time lines into one. The mechanisms by which this is accomplished is not relevant to this paper.

3 Bringing it Together

Leveraging partial persistence in data structures to achieve Wait-Free Population Oblivious reads becomes rather intuitive once the mechanics are understood. Due to the immutability of previous states, simply maintain a time stamp to the most recent stable time t. Then when a new writer modifies the state at time $t + 1$ the readers of the structure at t are unaffected.

Naturally there may be some races behind the scenes if a reader tries to read a particular pointer while it's being modified but making them atomic solves this problem easily. Since each "pointer" now has to maintain additional information like a time stamp and potentially other information, (The extra pointers in a red-black tree would need to know whether they are right or left pointers for example) turn them into their own data type containing all the information they need. Then replace the old pointers with pointers to these types. (Which have the actual pointer to the next node inside them) This allows us to have all the pointers inside the actual node fit into the size of an actual pointer, rather than grow arbitrarily large with additional information. These fields then are small enough to be made atomic. (Note that on a 32bit machine an assignment to something the length of a pointer is an atomic operation already so this wouldn't be necessary. It is on 64bit machines though since they require two 32bit assignments per single pointer assignment.)

Breaking the pointers out into their own types and maintaining normal pointers to these new types and then making those normal pointers atomic allows us to read past states without worry of any interference from a writer allowing wait free population oblivious reads into the structure.

4 Future Work

What about writes?

One natural follow up question is what about writes? It would be nice to pair our wait free population oblivious reads with at least lock free writes. There is no known way yet to achieve this in the augmenting to persistence model proposed here aside from some conventional methods or "hacks". (Like a lock-free operation queue.)

In keeping with the theme of persistence though, one theory worth pursuing is using confluent persistence. Described in part 2, confluent persistence allows for branching timelines and the ability to fuse those branches into one new one. The idea then is that for any write operation, simply make a branch in time off of the most recent time, make any and all changes desired and then merge it back into the structure. If there have been no updates between the branch point and the merge point, then the operation is considered successful and left alone. Otherwise, like in the typical CAS based lock-free programming method, the work is scrapped and retried from the most current version. (Note that confluent persistence maintains it's timelines and versions in a DAG rather than a line but we can artificially constrain it to function as a linear time line for our needs here.) In theory, this would not impact the WFPO reads already available and would provide a lock-free writes via another form of persistence.

Managing overhead

Despite the constant asymptotic space overhead, (using the right structures like red-black trees. Can be far worse with some others.) using partial persistence in the way we are can be considered wasteful. Within partial persistence all previous states are available to be queried and thus nodes with very outdated information can never be removed. For our uses though, there reaches a point where some nodes are no longer needed. Say a particular node has been copied from many times and it's pointer with the largest time stamp has a time stamp of t. Given that readers only read from the most current stable time stamp, after there exists a single time stamp $t + 1$ and all readers at time $\leq t$ are done reading, that node is no longer required. To save space and prevent this structure from growing in size with useless information it would be nice to have some method with which outdated nodes can be easily detected and deleted.

5 Conclusion

This paper addresses a long standing issue of reading to a contested parallel data structure by augmenting the pointer-based structure with partial persistence. Using the immutability past states given by partial persistence along with minimal atomics inside each of the nodes in the structure it is possible to achieve Wait-Free Population Oblivious reads.

An interesting problem remains though on whether or not this idea can be extended to achieve lock-free writes with confluent persistence.

References

[1] D. Sleator R. Tarjan J. Driscoll, N. Sarnak. Making data structures persistent. *Computer and System Sciences*, 38(1):86–124, 1989.