# LITERATURE REVIEW: Achieving Lock-Free Writes and Wait-Free Reads with Persistence

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

November 20, 2016

## 1 Introduction

As soon as parallel computing became more than an idea, a question had to be answered. How to get multiple threads to work on the same data? The solution at the time was to use locks and mutexes to prevent multiple threads from writing to the data, or reading while another is writing. While this works it is not without its issues. For instance in this model threads can enter deadlock by which they have a lock and are waiting for another which is currently possessed by a different thread that is waiting on the original threads lock. In this situation the threads will be stuck and simply wait until the process is terminated.

Enter lock-free programming. This new paradigm is not one where, as the name might suggest, a thread never has to wait or that locks are not used, but rather one that simply guarantees progress in the program by at least one thread. The problem though is that lock-free programming is usually very hard and error prone so despite its benefits, it is rarely used.

The goal of this paper then is to demonstrate a method for which any pointer based data structure can be easily made lock-free for writes and wait-free for reads by making the data structure persistent and leveraging the immutability of the structure at a given timestamp.

## 2 Literature Review

The current method with which lock-free programming is achieved is primarily through the CAS (Compare And Swap) operation. This is demonstrated in a lock free queue implementation in [**?**] which we will be using as our lock-free operation queue. Beyond directly borrowing their implementation of the queue though, this method of lock-free programming is not used anywhere else in the structure being presented.

The process by which we can make arbitrary pointer based data structures persistent is demonstrated in [**?**]. 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 updates. 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 timestamp 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, dropping all but the most up-to-date information. This means the new node will have an empty extra pointer array. This though means that other nodes will have to relink to this node while still maintaining the old pointers which could potentially cause them to duplicate as well. 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, this still amortizes to constant space per single update, or linear space overall.