Lock-Free Algorithms for Thread Safe Programming

Patrick Garrity '12 – St. Olaf College – garrity@stolaf.edu

ABSTRACT

An object is *lock-free* if it guarantees that in a system where multiple threads are manipulating the object, some thread will complete its operation on the object in a finite number of steps, no matter what happens to other threads. More practically, lock-free programming is a technique to achieve thread safety without the use of locks, which are the current standard. Lock-free objects (LFOs) have a number of useful properties that make them more broadly useful than locked objects, and also tend to perform well under high contention (being accessed by many threads at once). Though lock-free objects are not very common in practice, there is an opportunity for this to change. Lock-free algorithms (LFAs) are notoriously difficult to design and prove correct, and implementing them presents a number of technical challenges, but over the course of around ten years several commonly used data structures have been developed. This paper approaches lock-free objects from a practical standpoint. It addresses their features and problems, along with specific implementation issues. This paper also summarizes a large set of useful lock-free algorithms that would be broadly useful. Finally, it reports on the author's efforts to implement a collection of lock-free data structures (LFDS) for general use.

1. Introduction

Researchers have developed lock-free algorithms since before the nineties, though many of the practically useful algorithms were not published until the past decade. The rise of multi-core computers and promise of highly concurrent systems provided a market for efficient concurrent algorithms. LFAs are particularly desirable in this market as they tend to perform very well under high contention.

Specifically, most lock-free research has gone toward designing algorithms for scalable lock-free data structures, which offer a number of useful properties in the context of concurrent access. By definition, LFDS are immune to deadlock and livelock, which is useful to any developer [12]. Additionally, they maintain this property in the event of arbitrary thread failure – any number of threads may be killed, and an LFDS will remain functional [12]. LFDS are also tolerant of priority inversion, as there is no way for them to 'lose' anything (a lock) in a low-priority thread [12]. Similarly, they are tolerant of thread preemption. In all of these situations, data structures based on locks may suffer or cease to function. Finally, LFDS have asynchronous signal safety, which allows them to be used in signal handlers safely [12]. Locks are generally unsafe to use in signal handlers due to the situation where a handler and a thread are in deadlock. This further prevents many implementations of `malloc` and `free` from use in signal handlers, as they are lock-based.

2. Problems and Solutions

Despite the impressive features of LFDS, they suffer from a number of problems that have prevented them from gaining a strong mainstream presence. The first of these is that LFDS are difficult to design and prove correct. Furthermore, implementing an algorithm is difficult as well, and presents a number of technical challenges and system dependency issues, some of which are addressed in section 4. Beyond this, there are three major problems that affect LFDS: the ABA problem, non-decreasing memory use, and memory allocation. At the time of writing, solutions exist for all of these problems.

The ABA problem occurs when a process that has read some shared memory value *A* is preempted by another process. The second process changes the shared memory value *A* to *B*, and then back to *A* again before the first process continues. Once the first process resumes execution, it does not notice that *A* has changed, and so acts as though this were true. This causes problems when a lock-free object deletes some value in one thread, then another thread allocates a new value at the same location. In this case, the first thread would have a pointer that has not changed, but the value being pointed at changed. Solutions to the ABA problem include tagging, and garbage collection or similar processes. Tagging involves using spare pointer bits to count the number of times a pointer has been modified or used [11]. This tentatively solves the ABA problem, but a counter may wrap around to the same value the old pointer held. A more full-proof solution to the ABA problem is to use garbage collection or a similar strategy for memory management [11]. Rather than being the goal of memory management, the ABA problem is usually solved as a side effect. Since garbage collection does not exist on all systems, another solution is required for languages such as C.

This leads directly into the issue of non-decreasing memory use of LFDS. This is caused by an inability to physically delete any data from an LFDS. Data may be logically disconnected, but there is no way to determine whether or not another thread is accessing the removed data. Physically deleting this data could lead to segmentation faults. The result is that LFDS will continue to consume memory until none is left, even if data is removed from them.

Garbage collection solves this problem, but is not generally available. One of the best-known solutions to this problem is an algorithm called *Hazard Pointers*, developed by Michael [10,11]. This algorithm relies on a global list of pointers to data that all threads may access. When a thread acquires a pointer that must point to valid data, this pointer is placed in the global list. Each thread also maintains a list of pointers to data it wishes to physically delete. Once this local list reaches a certain threshold size, every pointer that is not also a member of the global list is destroyed [11]. This manual form of garbage collection effectively solves the memory-use problem, and as a side-effect also prevents the ABA problem [11].

The memory allocation problem stems from the fact that most implementations of `malloc` (or other memory allocators) are not lock-free. Therefore, LFDS that use these allocators are not completely lock-free. This problem requires an implementation of a lock-free allocator, algorithms for which do exist [12]. There is debate as to whether these algorithms perform well or comparably to non-lock-free implementations.

Finally, it should also be mentioned that there are performance issues associated with lock-free programming. The atomic operations used may be expensive relative to locks, may have memory barriers, and using too many can lead to an overall slowdown. Care needs to be put into algorithm design to minimize or eliminate these calls when possible. This slowdown is also architecture-dependent, as different core/cache setups may produce different performance. Other problems aside, producing a lock-free algorithm that performs well requires additional insight.

3. Overview of Algorithms

Over the past twenty years, lock-free algorithms have undergone significant work. This paper examines this progress in the context of LFDS which have a wide range of applications and are commonly useful in concurrent code.

3.1 Stacks and Queues

Various stack algorithms currently exist, and the stack is arguably the simplest LFDS to design and implement. One of the oldest lock-free stack algorithms was created through IBM research based on a traditional stack implementation. This style of implementation has survived through time and serves as a starting point in lock-free stack design. Michael has also designed a similar stack to which he applied hazard pointers, making it one of the first LFDS to support memory reclamation. However, both of these stack algorithms suffer due to the single point of access that is the top of the stack. Since LFDS are designed to be used in highly concurrent situations, these stacks may become bottlenecks. Recently, researchers designed more scalable algorithms that attempt to manage contention for the top of the stack. One of these, presented by Hendler, Shavit, and Yerushalmi keeps track of which stack operation (push, pop, top) each thread is performing and streamlines the calls internally [2]. The author of this paper also implemented a stack independently, though it is equivalent to that presented by Michael.

Queues are slightly more complicated to implement, but can still take advantage of having a limited number of access points. One algorithm, designed by Michael and Scott, originated in 1996 and has been implemented by the author of this paper [8]. At one point the Intel Threading Building Blocks package experimented with using a lock-free algorithm for their concurrent queue class. However, tests showed that the atomic CAS operation cost them too heavily in performance [16].

3.2 Lists, Skip Lists, and Hash Tables

The (singly) linked list algorithm is significantly more complex than that of the stack or queue, due to the possibility of arbitrary insertions and removals. This class of LFDS has been heavily studied, though not all of the proposed designs are feasible. For instance, some linked list algorithms are based on DCAS (double compare and swap), which is generally not available. One practical linked list algorithm was presented by Fomitchev and Ruppert in 2004, and relies on CAS combined with the use of two flag bits [13]. The original solution is based on a list of ordered keys (which store the list values) but was reworked and generalized by the author of this paper [13]. This generalization supports any type of data, does not pay attention to order, and removes the reliance on keys – the list only stores values.

An extension of the linked list is the skip list. Skip lists are ordered linked lists where each node is assigned a random height, and this height can be used to skip nodes while searching. Skip lists perform in amortized time O(log(N)), making them useful in structures such as dictionaries. In the lock-free field, skip lists have been used in place of binary search trees due to good BST algorithms not existing until very recently (2010). Fomitchev and Ruppert presented a skip list algorithm in 2004 alongside their linked list algorithm [13]. Additionally, Fraser, Sundell, and Tsigas have worked to develop lock-free skip list algorithms.

These are directly relevant to hash tables, which rely on sets to provide buckets. So far hash tables have been implemented using list-based sets (skip lists) though it may now be possible to use binary search trees to create hash tables. In 2002, Michael presented a lock-free hash table algorithm dependent on the list-based set presented in [8].

3.3 Binary Search Trees and Deques

A good binary search tree algorithm is one of the most recent contributions to the lock-free field. In 2010, Ellen, Ruppert, Fatourou and van Breugel presented a leaf-based tree that implements a dictionary (each leaf is a key-value pair) [5]. Other similar algorithms have been developed (as mentioned in the previous section) but do not utilize trees as their primary structure. For example, Sundell and Tsigas produced a lock-free dictionary in [6] that instead relies on a skip list. Finally, significant work has been applied to development of deques, circular lists, and related data structures. A prime example of this effort is presented by Chase and Lev in [3].

4. Implementation Details

Lock-free algorithms generally rely on a special processor instruction called compare and swap or CAS. This operation is the common point of many implementations, and must be understood in order to work in the current field of LFAs. CAS is an atomic operation that changes a value if and only if it is equivalent to some expected value. The CAS instruction can be described by the following C code:

Listing 4.1: Compare and Swap

```
1: int compare_and_swap(int* dest, int cmp, int swap)
2: {
3:    int old_val = *dest;
4:    if (old_val == cmp)
5:        *dest = swap;
6:    return old_val;
7: }
```

The strength of compare and swap is that multiple operations are achieved in a single instruction. Since CAS is a processor instruction, this behavior is achieved without any locks. This atomicity allows locks to be replaced by clever applications of this function. An example use-case of CAS is popping the top of a stack. Listing 4.2 attempts to do so by finding the top of the stack, getting the next element, then doing a CAS operation on the top of the stack.

Listing 4.2: Example use of CAS

```
1: top = top_of_stack;
2: do {
3:     cmp = top;
4:     next = cmp->next();
5: } while (!cas(top, cmp, next));
```

Listing 4.2 accounts for concurrent operations - if the top of the stack is changed between the lines 3 and 5, the call to CAS will fail. Since CAS is an atomic operation, if call to it begins, it is guaranteed to complete without interruption.

The CAS instruction is available on most modern 32- and 64-bit systems, and is therefore a reliable component of these algorithms. However, using this instruction can be platform dependent. The GCC, Mac OS X, and Windows libraries all provide functions for CAS, though each behaves differently. Additionally, for systems that do not provide CAS, the instruction must be used through inline assembler. This situation makes implementing a general-use CAS function that works on all platforms a non-trivial task.

As an alternative to CAS, the LL/SC (load-linked/store conditional) instruction is also used. This is an implementation detail, but has the same effect. One pitfall of several lock-free algorithms, however, is the use of DCAS which performs a CAS operation on two non-adjacent words at once. This operation is potentially useful, but is not generally supported on hardware. If more than a single pointer needs to be compared and swapped at once, other methods must be used. One way to transport additional data is through the use of spare pointer bits. On a 32-bit system, 2 bits may go unused. On a 64-bit system, 16 bits may be available. Manipulating spare bits in pointers can be a risky solution, but it can also provide the solutions to algorithm implementation without resorting to functions that may or may not be supported.

There are also efforts to implement lock-free algorithms by minimizing the use of CAS and atomic primitives. These generally lead to wait-free implementations (a stronger version of lock-free – all threads make progress) with high throughput [1].

V. Library Development

Over the course of several months, the author of this paper developed a library of lock-free data structures. This library is implemented in C++, and makes an effort to correspond to the standard template library data structures when possible. Over the course of several iterations, a functional stack, queue, and linked list were completed. Additionally, a flexible hazard pointer system was implemented that could easily be used to manage hazard pointers for any LFDS. Both the stack and queue are memory managed, while the list is a work in progress. Finally, binary search tree and dictionary implementations are in development. This library is cross-platform, and compiles under GCC 4.4+ and Visual Studio 2010 for both 32- and 64-bit systems.

Unfortunately, this project is unavailable to the public and is maintained as a research project. The original goal of the project was to create a robust, open-source template library. Due to patenting issues with certain algorithms that are currently being investigated, the source cannot be released at this time. There are at least two growing C libraries, and the open source community appears to be producing work in this direction.

VI. Conclusion

There is currently a rich set of lock-free algorithms that have been published over the course of roughly twenty years. Although lock-free code has some inherent issues, these have mostly been addressed and remedied in some way. Additionally, lock-free objects have strong benefits and a wider range of use than their locked counterparts at the occasional cost of performance. Even so, use of these algorithms has not been standard and implementations are scattered. There are growing libraries of code designed to be lock-free, and the author of this paper has also made progress in this regard.

References

[1] Alex Kogan and Erez Petrank. 2011. Wait-free queues with multiple enqueuers and dequeuers. In Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11). ACM, New York, NY, USA, 223-234. DOI=10.1145/1941553.1941585 http://doi.acm.org/10.1145/1941553.1941585

[2] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '04). ACM, New York, NY, USA, 206-215. DOI=10.1145/1007912.1007944 http://doi.acm.org/10.1145/1007912.1007944

[3] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '05). ACM, New York, NY, USA, 21-28. DOI=10.1145/1073970.1073974 http://doi.acm.org/10.1145/1073970.1073974

[4] Dmitri Perelman, Rui Fan, and Idit Keidar. 2010. On maintaining multiple versions in STM. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (PODC '10). ACM, New York, NY, USA, 16-25. DOI=10.1145/1835698.1835704 http://doi.acm.org/10.1145/1835698.1835704

[5] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (PODC '10). ACM, New York, NY, USA, 131-140. DOI=10.1145/1835698.1835736 http://doi.acm.org/10.1145/1835698.1835736

[6] Hakan Sundell and Philippas Tsigas. 2004. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM symposium on Applied computing* (SAC '04). ACM, New York, NY, USA, 1438-1445. DOI=10.1145/967900.968188 http://doi.acm.org/10.1145/967900.968188

[7] M. Herlihy. 1990. A methodology for implementing highly concurrent data structures. *SIGPLAN Not.* 25, 3 (February 1990), 197-206. DOI=10.1145/99164.99185 http://doi.acm.org/10.1145/99164.99185

[8] Maged M. Michael and Michael L. Scott. 1995. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. Technical Report. University of Rochester, Rochester, NY, USA.

[9] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (SPAA '02). ACM, New York, NY, USA, 73-82. DOI=10.1145/564870.564881 http://doi.acm.org/10.1145/564870.564881

[10] Maged M. Michael. 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (PODC '02). ACM, New York, NY, USA, 21-30. DOI=10.1145/571825.571829 http://doi.acm.org/10.1145/571825.571829

[11] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491-504. DOI=10.1109/TPDS.2004.8 http://dx.doi.org/10.1109/TPDS.2004.8

[12] Maged M. Michael. 2004. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.* 39, 6 (June 2004), 35-46. DOI=10.1145/996893.996848 http://doi.acm.org/10.1145/996893.996848

[13] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* (PODC '04). ACM, New York, NY, USA, 50-59. DOI=10.1145/1011767.1011776 http://doi.acm.org/10.1145/1011767.1011776

[14] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. 2004. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* (PODC '04). ACM, New York, NY, USA, 31-39. DOI=10.1145/1011767.1011773 http://doi.acm.org/10.1145/1011767.1011773

[15] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (SPAA '10). ACM, New York, NY, USA, 253-262. DOI=10.1145/1810479.1810528 http://doi.acm.org/10.1145/1810479.1810528

[16] Wooyoung Kim. The Concurrent Queue Container With Sleep Support. Intel Software Network Blogs. June 3, 2008. http://software.intel.com/en-us/blogs/2008/06/03/the-concurrent-queue-container-with-sleep-support/.