lfbench: a lock-free microbenchmark suite

Mahita Nagabhiru
Department of Computer and Electrical Engineering
North Carolina State University
Raleigh, USA
mnagabh@ncsu.edu

Greg Byrd
Department of Computer and Electrical Engineering
North Carolina State University
Raleigh, USA
gbyrd@ncsu.edu

***Abstract*—In this work, we present *lfbench*: a micro-benchmark suite intended as a one-stop shop representing all the popular lock-free data structures. Lock-free programming is very complex and so hard that there hasn't been a generalized lock-free algorithm designed; instead, lock-free data structures are individually developed and optimized for the specific use-cases. In spite of this difficulty, lock-free programs are indispensable; OS kernel codes, popular databases, networking buffers, and so forth, all rely on lock-free data structures for the performance and scalability they provide. We attempt for the first time to bring all the popular lock-free data structures under one roof, primarily to enable development of new H/W semantics needed for easy lock-free programming and help evaluate the same. Additionally, the benchmark suite can be used for:**

1) **Performance analysis of any new S/W algorithms/ libraries developed.**
2) **Building blocks for complex multi-threaded applications.**

I. Introduction

Using locks to provide correct concurrent execution is easier to reason when considering isolation, and hence it is a more common practice. We can guard a critical section using locks such that only one writer thread can execute at a time, while other threads spin and wait for the release of the lock. Despite this, writing performance efficient fine-grain locking is hard and suffers from deadlocks, priority inversion and lack of composability [1]. Lock-free programming, on the other hand, lets all threads attempt the critical section concurrently: each thread makes thread-local copies as it progresses, but only one can publish this thread-local work to the shared memory space, while others fail and re-try. This is achieved by keeping the critical section inside an *always-true while-loop* and exiting on a successful write using efficient H/W atomics like *compare-and-swap* (CAS), *fetch-and-add*, etc. Thus, Lock-free programming inherently avoids the above listed problems owing to its "non-blocking" nature while it enjoys disjoint-access parallelism, giving better performance and scalability. To be called lock-free, a program needs to avoid deadlocks and live-locks and provide a program wide forward progress guarantee.

II. Related Work

Lock-free programming is harder than lock-based programming, because it inherently suffers from memory reclamation problem and the ABA-problem [9]. It also requires stronger H/W semantics like a multi-compare-multi-swap (MCMS), spanning multiple words to make linearizable updates to the data structure getting modified. For example, consider a B-tree shown in Fig. 1. Consider a writer thread removing node-9. As shown in the figure, we need at least a 4-way atomic CAS operation to do the removal correctly: update `node-14->left`, `node-10->left`, `node-9->left` and `node-9->right`. We need to update node-9's `left` and `right` pointers such that no concurrent writer accidentally adds children to a node that is getting removed. Additionally, once node-9 is detached from the tree, we cannot delete this node and reclaim the memory immediately, because we do not know if any other reader threads are still referring to this node as the writer did not have any exclusive rights to this node while it was removing it.

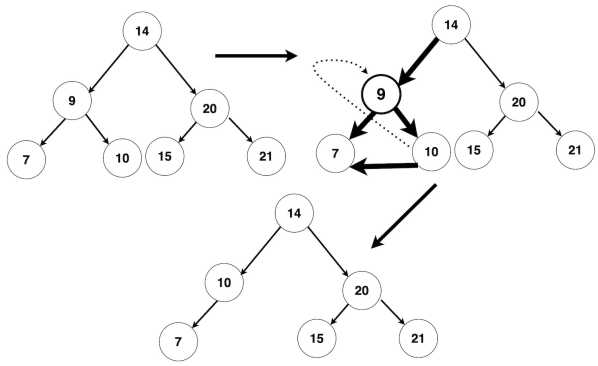

Fig. 1. Example: B-tree removal

In the absence of a H/W assisted multi-word CAS (MCAS), lock-free programmers developed custom algorithms while augmenting the base data structure with additional metadata and inferring any updates through this metadata; e.g., for the above B-tree, there is a map-table created that captures the various incremental updates to the B-tree, consolidates these lazily and updates using a single-CAS at a page granularity [2]. A S/W based MCAS has been created by [1] and optimized by several others [3], but these still suffer from heavy complexity, non-trivial memory reclamation and indirection to data access. Hardware transactional memory (HTMs) [10] are a good alternative solution but lack of forward progress in most commercial HTMs make them prone to live-locks

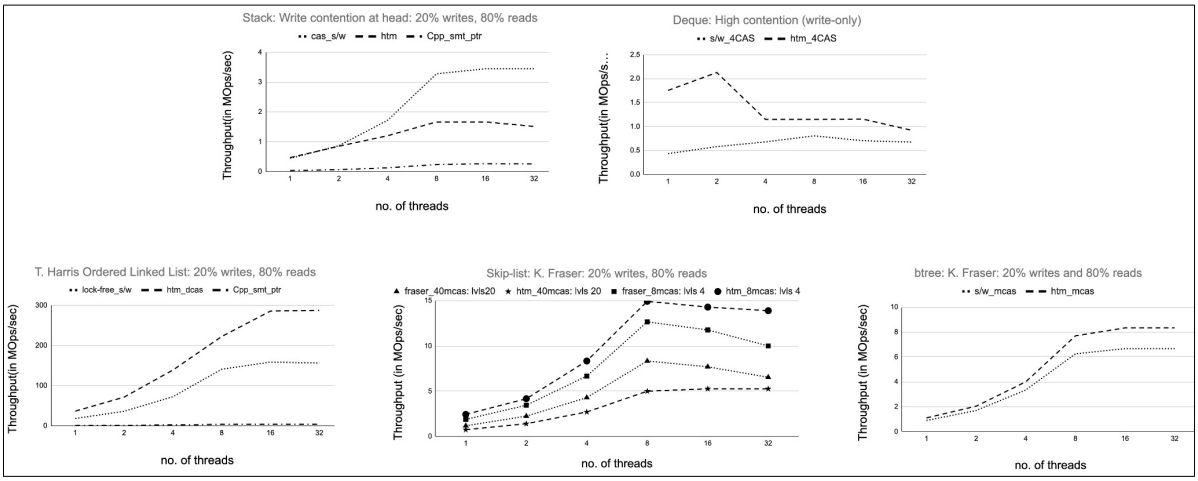

Fig. 2. Experimental results: Shows Throughput (in Million Operations/sec) scaling as thread count increases on an 8-core multi-threaded system

needing a lock-based fall-back path for forward progress. HTMs also suffer from read/write set limitations, and they lack debugging infrastructure, rendering them prone to performance bugs. Recent work on H/W multi-word atomics [4] [5] comes close to addressing some of these issues, but it is focused on a H/W MCAS and evaluation of the design is only on a subset of lock-free data structures. Their notable work on [8] is complementary to our work, where they port traditionally lock-based multi-threaded programs to lock-free style but still rely on single-word atomics to ensure lock-freedom. Our work is distinct for the following reasons:

1) We provide lock-free data structures in both S/W and H/W using MCMS-like style to provide like-for-like comparison of any new H/W semantics that are developed.
2) We use HTM not as a coarse-grain synchronization semantic but as a building block for powerful multi-word atomics.
3) Using our benchmark suite, we want to address all the inherent problems in lock-free programming like memory reclamation, enabling faster reads and building a foundation for our future work that motivates not just for an MCAS but a much stronger MCMS [7]. Key difference in MCMS is that unlike in MCAS, we may chose to write only to a subset of words that we read, providing more potential for optimization at H/W level.

III. Experiments

We used the ARM-TME model on gem5-v20 [11] for our architectural exploration work but our benchmark suite is H/W independent. This suite consists of the following data structures and workloads:

- Stack: A LIFO stack is significant for its simplicity, and is instrumental in comparing raw performance on various semantics without having complexity of the data structure. It has three variants: S/W based on single-CAS, HTM based and C++ smart pointer based for safe memory reclamation. In Fig. 2, we see that single-CAS is the most efficient one, while HTM does not scale well at higher thread count due to live-lock. Smart pointers are inherently slow because they use lock-based version counting to maintain portability across platforms.
- Deque: A doubly-ended queue represents an important workload for OS kernel code and is a write-intensive workload that adds and removes nodes from both ends. We provide a 4-CAS based implementation using S/W based MCAS from [1] and alternatively, use HTM-based MCAS for the hardware assisted model. This is an important workload that highlights again how the HTM can tend to be stuck in live-lock under heavy contention.
- Ordered singly-linked list: We use a custom S/W thread-cooperation based model from [6] as a base variant and then compare it against a HTM-based model using a Double-word CAS (DCAS). DCAS one shows a significant improvement we expect on avoiding lot of S/W based indirection. This is a read-intensive workload.
- Skip-list: This builds on the previous ordered linked list but significantly puts pressure on MCAS, as the height (levels) of the skip-list increases. The S/W one uses MCAS from [1] and scales better, even for higher levels of the list. The HTM-based MCAS suffers as soon as it starts experiencing cache capacity/associativity aborts, highlighting another limitation of the HTM.
- B-tree: This is a custom MCAS based implementation of the B-tree that uses an 8-way CAS at max but scales well in both HTM and S/W based MCASes. We can get rid of most of the complexity of the S/W based traversal in the HTM-based one.

The suite can be accessed here: https://github.com/mahita649/lfbench_suite

References

[1] T. L. Harris, K. Fraser, and I. A. Pratt, "A Practical Multi-word Compare-and-Swap Operation." DISC 2002, pp. 265-279.

[2] D. Makreshanski, J. Levandoski, and R. Stutsman, "To lock, swap, or elide," vol. 8, no. 11, pp. 1298–1309, Jul. 2015, doi: 10.14778/2809974.2809990.
[3] M. Pavlovic, A. Kogan, V. Marathe, and T. Harris, "Brief Announcement: Persistent Multi-Word Compare-and-Swap." PODC 2018, pp. 37–39, doi: 10.1145/3212734.3212783.
[4] S. Patel, R. Kalayappan, I. Mahajan, and S. R. Sarangi, "A hardware implementation of the MCAS synchronization primitive," DATE 2017, pp. 918–921, doi: 10.23919/DATE.2017.7927120.
[5] E. J. Gómez-Hernández, J. M. Cebrian, R. Titos-Gil, S. Kaxiras, and A. Ros, "Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations." MICRO 2021, pp. 337–349.
[6] T. L. Harris., "A Pragmatic Implementation of Non-blocking Linked-Lists," DISC 2001, pp. 300–314.
[7] Trevor Brown, William Sigouin, Dan Alistarh, "PathCAS: an efficient middle ground for concurrent search data structures.", PPoPP 2022, pp. 385–399.
[8] Eduardo José Gómez-Hernández, Ruixiang Shao, Christos Sakalis, Stefanos Kaxiras, Alberto Ros, "Splash-4: Improving Scalability with Lock-Free Constructs." ISPASS 2021, pp. 235–236.
[9] D. Dechev, P. Pirkelbauer and B. Stroustrup, "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs," ISORC 2010, pp. 185–192, doi: 10.1109/ISORC.2010.10.
[10] Ravi Rajwar, James R. Goodman, "Transactional lock-free execution of lock-based programs.", ASPLOS 2002, pp. 5–17
[11] Timothy Hayes, "Arm's Transactional Memory Extension support in gem5", https://www.gem5.org/project/2020/10/27/tme.html, accessed 07/Mar/2023.