Brief Announcement: Towards Reduced Instruction Sets for Synchronization*

Rati Gelashvili[1], Idit Keidar[2], Alexander Spiegelman[2], and Roger Wattenhofer[4]

1 MIT, Cambridge, MA, USA
2 Technion, Haifa, Israel
3 Technion, Haifa, Israel
4 ETH, Zürich, Switzerland

Abstract

Contrary to common belief, a recent work by Ellen, Gelashvili, Shavit, and Zhu has shown that computability does not require multicore architectures to support "strong" synchronization instructions like *compare-and-swap*, as opposed to combinations of "weaker" instructions like *decrement* and *multiply*. However, this is the status quo, and in turn, most efficient concurrent data-structures heavily rely on *compare-and-swap* (e.g. for swinging pointers).

We show that this need not be the case, by designing and implementing a concurrent linearizable Log data-structure (also known as a History object), supporting two operations: *append*(*item*), which appends the item to the log, and *get-log*(), which returns the appended items so far, in order. Readers are wait-free and writers are lock-free, hence this data-structure can be used in a lock-free universal construction to implement any concurrent object with a given sequential specification. Our implementation uses atomic *read*, *xor*, *decrement*, and *fetch-and-increment* instructions supported on X86 architectures, and provides similar performance to a *compare-and-swap*-based solution on today's hardware. This raises a fundamental question about minimal set of synchronization instructions that the architectures have to support.

1998 ACM Subject Classification C.1.2 Multiple Data Stream Architectures (Multiprocessors), D.1.3 Concurrent Programming

Keywords and phrases Consensus hierarchy, universal construction, synchronization instruction

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.53

1 Introduction

In order to develop efficient concurrent algorithms and data-structures in multiprocessor systems, processes that take steps asynchronously need to coordinate their actions. In shared memory systems, this is accomplished by applying hardware-supported low-level atomic instructions to memory locations. An atomic instruction takes effect as a single indivisible step. The most natural and universally supported instructions are *read* and *write*, as these are useful even in uniprocessors to store and load data from memory.

A concurrent implementation is *wait-free*, if any process that takes infinitely many steps completes infinitely many operation invocations. An implementation is *lock-free* if in any infinite execution infinitely many operations are completed. Binary consensus is a synchronization task where processes start with input bits, and must agree on an output bit

* A full version of the paper is available at `http://arxiv.org/abs/1705.02808`.

that was an input to one of the processes. For one-shot tasks like consensus, wait-freedom and lock-freedom are equivalent. Herlihy's Consensus Hierarchy [2] assigns a *consensus number* to each object, namely, the number of processes for which there is a wait-free binary consensus algorithm using only instances of this object and *read-write* registers. An object with a higher consensus number is hence a more powerful tool for synchronization. Moreover, Herlihy showed that consensus is a fundamental synchronization task, by developing a universal construction which allows n processes to wait-free implement any object with a sequential specification, provided that they can solve consensus among themselves.

Herlihy's hierarchy provides an explanation as to why, for instance, the *compare-and-swap* instuction can be viewed "stronger" than *fetch-and-increment*, as the consensus number of a Compare-and-Swap object is n, while the consensus number of Fetch-and-Increment is 2.

However, key to this hierarchy is treating synchronization instructions as distinct objects, an approach that is far from the real-world, where multiprocessors do let processes apply supported atomic instructions to arbitrary memory locations. In fact, a recent work by Ellen et al. [1] has shown that a combination of instructions like *decrement* and *multiply-by-n*, whose corresponding objects have consensus number 1 in Herlihy's hierarchy, when applied to the same memory location, allows solving wait-free consensus for n processes. Thus, in terms of computability, a combination of instructions traditionally viewed as "weak" can be as powerful as a *compare-and-swap* instruction, for instance.

The practical question is whether we can really replace a *compare-and-swap* instruction in concurrent algorithms and data-structures with a combination of weaker instructions. *compare-and-swap* is ubiquitous in practice and used heavily for various tasks like swinging a pointer. Also, the protocol given by Ellen et al. solves only binary n-process consensus. It is not clear how to use it for implementing complex concurrent objects, as utilizing Herlihy's universal construction is not a practical solution. On the optimistic side, there exists a concurrent queue implementation based on *fetch-and-add* that outperforms *compare-and-swap*-based alternatives [3]. Both a Queue and a Fetch-and-Add object have consensus number 2, and this construction does not "circumvent" Herlihy's hierarchy by applying different non-trivial synchronization instructions to the same location. Indeed, we are not aware of any practical construction that relies on applying different instructions to the same location.

We develop a lock-free universal construction using only *read*, *xor*, *decrement*, and *fetch-and-increment* instructions. The construction could be made wait-free by standard helping techniques. In particular, we implement a Log object (also known as a History object), which supports high-level operations *get-log*() and *append*(*item*), where *get-log*() returns all previously appended items in order. This interface can be used to agree on a simulated object state, and thus, provides the universal construction [2]. In practice, we require a *get-log*() for each thread to return a suffix of items after the last *get-log*() by this thread. We design a lock-free Log with wait-free readers, which performs as well as a *compare-and-swap*-based solution on modern hardware. We could replace *fetch-and-increment* and *decrement* with the atomic *fetch-and-add* instruction, reducing the instruction set size even further.

2 Algorithm

We work in the bounded concurrency model where at most n processes will ever access the Log implementation. The object is implemented by a single *fetch-and-increment*-based counter C, and an array A of b-bit integers on which the hardware supports atomic *xor* and *decrement* instructions. We assume that A is unbounded. Otherwise, processes can use A to agree on the next array A' to continue the construction. C and the elements of

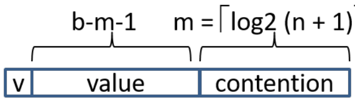

Figure 1 Element of A.

A are initialized by 0. We call an array location *invalid* if it contains a negative value, i.e., if its most significant bit is 1, *empty* if it contains value 0, and *valid* otherwise. The least significant $m = \lceil \log_2 (n+1) \rceil$ bits are *contention bits* and have a special importance to the algorithm. The remaining $b - m - 1$ bits are used to store items. See Figure 1 for illustration.

For every array location, at most one process will ever attempt to record a $(b-m-1)$-bit item, and at most $n-1$ processes will attempt to invalidate this location. No process will try to record to or invalidate the same location twice. In order to record item x, a process invokes $xor(x')$, where x' is x shifted by m bits to the left, plus $2^m - 1 \geq n$, i.e., the contention bits set to 1. To invalidate a location, a process calls a *decrement*. The following properties hold:

1. After a *xor* or *decrement* is performed on a location, no *read* on it ever returns 0.
2. If a *xor* is performed first, no later read returns an invalid value. Ignoring the most significant bit, the next most significant $b - m - 1$ bits contain the item recorded by *xor*.
3. If a *decrement* is performed first, then all values returned by later *read*s are invalid.

A *xor* instruction *fails to record an item* if it is performed after a decrement. To implement a *get-log* operation, process p starts at index $i = 0$, and keeps reading the values of $A[i]$ and incrementing i until it encounters an empty location $A[i] = 0$. By the above properties, from every valid location $A[j]$, it can extract the item x_j recorded by a *xor*, and it returns an ordered list of all such items $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. In practice, we require p to return only a suffix of items appended after the last *get-log*() invocation by p. This can be accomplished by keeping i in static memory instead of initializing it to 0 in every invocation. To make *get-log* wait-free, p first performs $l = C.read()$. Then, if i becomes equal to l during the traversal, it stops and returns the items extracted so far. To implement *append*(x), process p starts by $\ell = C.\textit{fetch-and-increment}()$. Then it attempts to record item x in $A[\ell]$ using an atomic *xor* instruction. If it fails to record an item, the process does another *fetch-and-increment* and attempts *xor* at that location, and so on, until it is able to successfully record x. Suppose this location is $A[\ell']$. Then p iterates from $j = \ell' - 1$ down to $j = 0$, reading each $A[j]$, and if $A[j]$ is empty, performing a *decrement* on it. Afterwards, process p can safely return. The proofs of lock-freedom and linearizability can be found in the full version at `http://arxiv.org/abs/1705.02808`.

We implemented the algorithm on X86 processor and with 32 threads. It gave the same performance as an implementation that used *compare-and-swap* for recording items and invalidating locations. It turns out that in today's architectures, the cost of supporting *compare-and-swap* is not significantly higher than that of supporting *xor* or *decrement*. This may or may not be the case in future Processing-in-Memory architectures [4]. Finding a compact set of synchronization instructions that, when supported, is equally powerful as the set of instructions used today is an important question to establish in future research.

References

1 Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based hierarchy for multiprocessor synchronization:[extended abstract]. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing*, 2016.

2 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

3 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 48 of *PPoPP'13*, pages 103–112, 2013.

4 David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.