Implementation of a Wait-Free Synchronization Primitive That Solves n-Process Consensus

Dror G. Feitelson Larry Rudolph
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, ISRAEL

Abstract

Several synchronization primitives were compared in a recent paper by Herlihy [7th PODC, 1988, pp. 276–290], using their ability to implement wait-free shared data objects as a measure of relative strength. Wait-free n-process consensus was shown to be a useful generic problem, whose solution for different values of n creates a hierarchy of primitives. Well known primitives such as test-and-set, swap, and fetch-and-add, can only solve 2-process consensus, and thus cannot be used to implement arbitrary wait-free objects. Stronger primitives that solve the n-process version were also suggested in the paper; these include memory-to-memory swap, compare-and-swap, and fetch-and-cons. However, it is not clear that these primitives can be implemented in a satisfactory manner. In this note a new primitive that solves the wait-free n-process consensus problem is presented: the fetch-and-conditional-swap. This primitive is easily implementable by a combining network.

Keywords: consensus, read-modify-write, wait-free synchronization

1 Introduction

Shared memory allows multiple processes to interact using few data objects; for example, a broadcast can be implemented by having the transmitter write to a shared cell and then having the receivers read from it. The problem with such a scheme is that if the transmitter is delayed for some internal reason, the receivers must wait for it. The solution is to try not to use operations such as broadcast, where many processes are dependent on the activities of one other process; rather, *symmetric* operations are used in which all processes play a similar non-critical role. In addition, the interactions should be *wait-free*, meaning that any process is guaranteed to complete its part in the interaction regardless of the progress of the others.

A comparison of some well-known primitives, based on the wait-free objects that they can implement, was reported recently by Herlihy [2]. At the heart of his work lies the problem of wait-free n-process consensus: n processes each start out with individual preferred values, and they must reach an agreement on one of these values. A naive solution would be to decompose the problem into two parts: first choose a leader, and then broadcast the leader's preferred value. However, the chosen leader might go to a party to celebrate its victory and forget to broadcast the value, thus leaving his followers in suspense. Therefore the actions of choosing the value and broadcasting it must be combined in some way into one atomic operation.

Herlihy uses n-process consensus in two ways. First, he uses it to form a hierarchy of operations:

1. Read/write memory cells cannot solve wait-free 2-process consensus.
2. Any non-trivial read-modify-write operation can solve wait-free 2-process consensus, but most cannot solve 3-process consensus. This includes test-and-set, swap, fetch-and-add, and FIFO queues.
3. Several primitives do solve the general n-process consensus problem. These include memory-to-memory move and swap, compare-and-swap, n-register assignment, fetch-and-cons, and an augmented FIFO queue.

Second, he shows that primitives that solve the wait-free n-process consensus problem are *universal*, in the sense that they can implement any wait-free object.

The main problem with Herlihy's results is that the primitives he suggests are not so primitive; in

fact, it seems doubtful whether they can be implemented efficiently at all. The purpose of this note is to present a new primitive that is simple to implement, and solves the wait-free n-process consensus problem; this is done in the next section. The final section evaluates the significance of this result.

2 Fetch-and-Conditional-Swap

The only way to implement a shared object that supports highly concurrent access by multiple processes is by use of a combining multi-stage interconnection network [3]. Such networks are indeed utilized by research prototypes like the NYU ultracomputer [1] and IBM RP3 [4]. Other methods use arbitration mechanisms to perform only one of a set of concurrent requests addressed at the same object, and thus force serial execution when concurrent access is attempted.

Combining proceeds as follows. The interconnection network is a multi-stage packet-switched network that connects processors to memory modules. Access requests are taken to be read-modify-write operations of the form $< addr, f >$, where $addr$ is an address in the shared memory and f is a function that specifies how to update the contents of that address. Such requests return the old contents of the address. When two requests $< addr, f >$ and $< addr, g >$ directed at the same address arrive at the same network node, they are combined and the unified request $< addr, g \circ f >$ is forwarded to the relevant memory module (where $g \circ f$ is the composition of g and f, i.e. $g \circ f(x) = g(f(x))$). When the return value v is received by the node, it is propagated back as a response to the first request. Then $f(v)$ is calculated, and sent as a response to the second request. The final effect is equivalent to a scenario in which the first request was served before the second one [3].

Not all types of functions can be combined, however. Using the symbols of the above example, we can identify the following two requirements that must be fulfilled for combining to be practical:

1. The composition of functions $g \circ f$ must be easy to compute.

2. The representation of $g \circ f$ should not require more space than the representations of f or of g.

The synchronization primitives suggested by Herlihy are not suitable for combining:

- Memory-to-memory swap: this instruction addresses *two* memory locations, and thus undermines the whole basis of a combining network's operation. These networks can handle multiple requests that propagate from the processors to the memory modules, but cannot handle direct interaction among the processors or among the memory modules.

- Compare-and-swap: each compare-and-swap requires the address contents to be compared with another specified value; when such requests are combined all the values must propagate on, and actually all the work must be done serially at the memory module.

- Fetch-and-cons: each fetch-and-cons requires another element to be added to the head of a list. Therefore when many are combined, many new elements must be transferred with the request.

The proposed primitive, fetch-and-conditional-swap, is based on the observation that a restricted version of compare-and-swap is sufficient for solving the n-process consensus problem. The initial value of the shared memory location is $\perp$. All the processes check for this same value, to see if they are first; if so, they substitute it with their preferred value. If the value they see is not $\perp$, they infer that another process arrived before them; in this case they just read its preferred value.

Formally, fetch-and-conditional-swap is defined as a read-modify-write operation such that the function f associated with a memory access request is

$$\lambda\, x\, y.\ \text{if } x = \perp \text{ then } y \text{ else } x,$$

where x is the contents of memory and y is the preferred value of the process. Two requests with preferred values y_1 and y_2 are combined by arbitrarily choosing one of them and sending it on; assume it is y_1. When the return value v arrives, the two return values are generated as follows: if $v = \perp$, return $\perp$ to the process that had sent y_1 and return y_1 to the process that had sent y_2. If it is not, return v to both. Obviously, the hardware needed to implement the combining is small, and there is no overhead in the size of the combined request.

It should be noted that fetch-and-conditional-swap is a restricted version of the data-level synchronization primitives suggested by Kruskal, Rudoplph, and Snir [3]. These are primitives that perform an operation on a shared variable, conditioned on the *state* that the variable is in. If the number of

possible states is small, as it is in our case, efficient combining is possible. This also explains why the full compare-and-swap cannot be combined: it effectively uses the whole memory-word contents as a state variable, and thus has an exponential number of states.

3 Evaluation

The fetch-and-conditional-swap primitive introduced in this note allows for a wait-free solution to the n-process consensus problem; this is interesting in its own right, as consensus is a basic problem in concurrent systems. In addition, it is a simple and practical universal wait-free primitive, using Herlihy's construction for implementing fetch-and-cons based on consensus and then inplementing any wait-free object by use of fetch-and-cons. As Herlihy notes, however, this construction is too inefficient to be considered useful for practical purposes. Therefore is seems that the universality of the new primitive is of academic interest only.

It is natural to ask whether this deficiency is specific to fetch-and-conditional-swap, in which case other primitives ought to be sought, or maybe it is inherent in the concept of universal wait-free primitives. The answer probably lies in between, specifically in the use of fetch-and-cons as an intermediate step in the reduction of an arbitrary object to a wait-free implementation using n-process consensus. The problem is that fetch-and-cons implies access to two shared memory locations at once: one is the pointer to the head of the list, which is made to point at the new element, and the other is the "next" pointer of the new element, which is made to point at the element which was previously pointed at by the head. Thus fetch-and-cons is similar to a memory-to-memory swap at a very basic level.

Realization of a practical universal wait-free primitive therefore requires a solution to one of the following problems:

1. Find an efficient and practical implementation of memory-to-memory swap.

2. Find a direct method to generate a wait-free implementation of a concurrent object, using n-process consensus but avoiding fetch-and-cons.

References

[1] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "*The NYU Ultracomputer — designing an MIMD shared memory parallel computer*". *IEEE Trans. Computers* **C-32(2)**, pp. 175–189, Feb 1983.

[2] M. P. Herlihy, "*Impossibility and universality results for wait-free synchronization*". In *Proc. 7th Symp. Principles of Distributed Computing*, pp. 276–290, 1988.

[3] C. P. Kruskal, L. Rudolph, and M. Snir, "*Efficient synchronization on multiprocessors with shared memory*". *ACM Trans. Programming Languages and Systems* **10(4)**, pp. 579–601, Oct 1988.

[4] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "*The IBM research parallel processor prototype (RP3): introduction and architecture*". In *Intl. Conf. Parallel Processing*, pp. 764–771, 1985.