On Beyond Registers: Wait-Free Readable Objects

Maurice Herlihy

ABSTRACT

Leslie Lamport was the first to pose many of the fundamental questions about synchronization that drive much of our community's research, even today. In this paper, we revisit some of Lamport's classic questions in a modern context. In particular, we consider some of the implications

Prepared for Leslie Lamport's birthday celebration at PODC 2001.

1. INTRODUCTION

Leslie Lamport was the first to pose many of the fundamental questions about synchronization that drive much of our community's research, even today. In this paper, we revisit some of Lamport's classic questions in a modern context, and we explore a few intriguing open questions that are the direct intellectual descendants of Lamport's original approach. These questions are by no means exhaustive. They were chosen because they are easily stated, because they interest the author, and because every approach to these questions inevitably leads to rereading a Lamport paper.

Lamport started by looking at the simplest kind of communication: reading and writing a shared bit [17]. He identified three distinct kinds of behaviors, the now-familiar safe, regular, and atomic registers. The single-bit construction was extended to multiple bits, and the number of readers and writers proliferated. This work was extended by many others, culminating in the notion of an atomic *snapshot scan*: each thread has the ability to update one or more variables, and any process can instantaneously read all of them [3, 5, 4, 6, 2, 7, 9, 18, 19].

Has this research program been mined out? Outside the PODC community, atomic registers as such have faded from view. Modern architectures provide programming models based on a bewildering variety of weaker and more complex memory models, typically some combination of weakly-ordered reads and writes augmented by synchronization primitives such as memory barriers. Hardware memory implementations consist of multi-level cache hierarchies linked by complicated coherence protocols that bear little resemblance to the atomic register algorithms in the literature. No one uses read/write registers for synchronization: instead, modern architectures provide powerful operations such as compare-and-swap or load-linked/store-conditional.

PODC 01 Newport Rhode Island USA

I believe that the research program launched by Lamport's study of wait-free atomic registers is far from exhausted, though some thought is needed to keep it pointed in productive directions. In this area, as in many areas of science, the questions are more important (and more lasting) than this year's answers.

2. READING IS FUNDAMENTAL

Let us review how atomic snapshots work. Central to most snapshot algorithms is the notion of a "clean collect". A *collect* occurs when a thread reads, one-at-a-time, the sequence of memory locations that comprise the snapshot. A *clean collect* is a collect that occurs during an interval when no other thread updates a collected location. Clearly, a clean collect yields an atomic snapshot scan. There are two technical challenges in designing an atomic snapshot scan: ensuring that a clean collect will occur, and recognizing when a clean collect has occurred. The first challenge is typically met by clever use of flags and counters, and the second by requiring each thread that updates the scanned memory to help the others by performing a collect, ensuring that some thread's collect is clean.

Here is the basic premise guiding the rest of this paper. Expressed in the delightful terminology of post-modernism, there is no reason, either pragmatic or mathematical, to "privilege" atomic writes above other operations. One can equally well take snapshots of the effects of any operation that modifies memory, including read-modify-write operations. Moreover, one can in principal) perform an atomic snapshot of the physical memory underlying any object, or even any collection of objects.

This observation leads to the first open question. Can atomic snapshots be made truly practical? By modifying standard cache-coherence protocols, it is relatively easy to provide minimal architectural support for lock-free snapshots of small regions of memory. In a variation of Transactional Memory [13], one can collect the desired region of memory into a processor's local cache. The cache coherence protocol

can be enlisted to notify the processor if any of these cache lines is invalidated during the collect. If not, the collect is clean. Additional mechanisms are needed to deal with memory regions that cannot fit in the cache for one reason or another, and to make the collect wait-free (or perhaps just wait-free for all practical purposes).

3. READABLE OBJECTS

Taking a snapshot of an object's underlying memory is not the same as taking a snapshot of the object itself. Informally, let us say an object is *readable* if it provides a snapshot operation that returns its current state. We can make this definition (slightly) more precise as follows. There is a well-developed theory of sequential objects [11], in which the object's (abstract) state is given by a term in an algebra, axioms are used to define state equivalence, and terms in the algebra correspond (roughly) to the object's operations. For sequential objects, a snapshot operation merely returns some encoding of the object's abstract state.

Because people are usually better at reasoning about sequential behavior than concurrent behavior (whether formally or informally), it is customary to define correctness for concurrent objects in terms of some form of equivalence to sequential objects. For example, *sequential consistency* [16] requires that one can reorder potentially-overlapping concurrent operations in some sequential order, with the restriction that operations executed by a single thread cannot be reordered. *Linearizability* [14] additionally requires that Lamport's "happens-before" ordering be respected. A natural correctness condition for an atomic snapshot operation is that it satisfy the same correctness condition as other concurrent operations, say, sequentially consistent or linearizable.

4. REFINEMENT MAPS

The next open question is characterizing when snapshot operations exist. Snapshot operations are closely related to the issue of refinement maps, a subject studied by Abadi and Lamport [1].

What is the "state" of a concurrent object? For sequential implementations, one defines an *abstraction map* carrying states of the object's representation (for example, an array) to terms in the type's abstract algebra (for example, FIFO queue values). The abstraction map is well-defined when the object is quiescent (no operation is in progress), and is usually left undefined while an operation is in progress.

The notion of an abstraction map extends readily to monitors or other synchronization styles based on mutual exclusion, but it is not helpful for reasoning about wait-free or lock-free concurrent objects, since such an object may never become quiescent.

A *refinement* map is the continuous analog of an abstraction map. It carries the state of one automaton, typically the implementation, to another, typically the sequential type being implemented. Refinement maps are useful because they transform the (awkward) problem of reasoning about histories to the (less awkward) problem of reasoning about states.

Refinement maps are not always easy to construct, however. For example, Herlihy and Wing [14] give an example of a linearizable lock-free FIFO queue implementation which has no refinement map defined in terms of the shared memory and the threads' local states (including registers and program counters). The problem arises when two items are enqueued concurrently. The queue construction has the property that the order in which those items will be dequeued depends on future race conditions, thus any order chosen by a refinement map could be "disproved" by later events. We will return to this example later.

Abadi and Lamport consider the conditions under which refinement maps exist. They show that one can always construct a refinement map provided the object satisfies certain reasonable technical restrictions, and provided one can introduce auxiliary variables. Auxiliary variables come in two flavors: *history* variables track state information that may have been discarded or moved into threads' local states, while *prophecy* variables predict future non-deterministic choices. Prophecy variables are a little like non-deterministic Turing Machines in complexity theory: when faced with a range of choices, they make the right one.

The analogy to Quantum Mechanics is irresistible (to this author) [1]. To an outside observer, a linearizable FIFO queue in which two objects have been enqueued concurrently can be thought of as being in a *superposition* of two states: one cannot tell which object will be dequeued first. A subsequent observation (dequeue or snapshot) will remove this ambiguity, forcing the queue into a sequential "eigenstate". (Beyond this point, the analogy breaks down because probabilities do not come into play.

Although refinement maps and snapshot operations are closely related, there is a substantial difference between defining a map (creating a mathematical object) and implementing a snapshot (writing an algorithm using specific synchronization primitives). When defining a function, we are free to define and manipulate history variables in ways that might be difficult or impossible to reproduce algorithmically. The computational analog of prophecy variables remains even more intriguing.

An intriguing open question is whether one can do for snapshot algorithms what Abadi and Lamport did for refinement maps: characterize the circumstances under which they can be constructed. The analogy between refinement maps and snapshot algorithms can be taken only so far: unlike refinement maps, which always exist, we will see that snapshot algorithms may not exist.

5. CONSENSUS THEORY

It is not hard to contrive an example of a model in which reasonable data types have no snapshot operations. Consider a (hypothetical) architecture that provides read/write registers and the following *poset* data type. The state of a *poset* object is a partially-ordered set of items. The *enq* operation places a maximal new item in the set, the *deq* op-

[1]This analogy between synchronization and Quantum Mechanics is of course loosely inspired by Lamport's well-known analogies between interprocess communication and Special Relativity [17].

eration removes and returns a minimal item in the set, and *enq* operations are required to respect the "happens-before" relation: if *enq(x)* finishes before *enq(y)* starts, then x must precede y in the object's partial order. Naturally, the *poset* type also provides a snapshot operation that returns the object's current partially-ordered set.

Now consider the standard FIFO queue type, which provides *enq* and *deq* operations, which behave in the usual way. A standard FIFO queue provides no snapshot operation, but a *readable* FIFO queue does.

It is not hard to see that a wait-free *poset* object provides a wait-free linearizable (and hence sequentially consistent) implementation of a standard FIFO queue. The non-determinism in the *poset* specification is masked by the non-determinism implicit in the linearizability correctness condition.

Can we use atomic registers and the *poset* object to give the FIFO queue a snapshot operation? The answer is *no*. The following claims are routine exercises [10, 12].

- A *poset* object can solve two-process consensus, but not three-process consensus.
- A standard FIFO queue can solve two-process consensus, but not three-process consensus.
- A readable FIFO queue can solve n-process consensus for any n.

The algorithm underlying the last claim is simple: each process enqueues a unique id in the queue, takes a snapshot, and observes whose id is first in the queue.

The implication is plain: in this (admittedly contrived) model of computation, one can implement a standard FIFO queue, but not a readable FIFO queue [2]. Introducing a snapshot operation can thus have a profound effect on an object's computational power, even for mundane, deterministic objects such as FIFO queues.

Readability is also closely connected to the problem of *robustness* [8, 15, 20, 21]: whether synchronization primitives with low consensus numbers can be combined to implement 'synchronization primitives with higher consensus numbers. Eric Ruppert [22] has shown that the class of readable types is robust, meaning that one cannot combine "weak" readable objects to get "stronger" objects. Violations of robustness seem to occur when an object solves n-process consensus, but (anthropomorphizing slightly) refuses to reveal the outcome unless the threads sharing the object demonstrate they can solve a problem of intermediate difficulty. If this coy object were readable, it could not conceal the consensus outcome from its environment, and no violation of robustness could occur.

In the same paper, ruppert also shows that there is an effective procedure for computing the consensus number of any readable object whose state space is finite. In view of these promising results, it seems reasonable to suggest that the effect of readability (or its absence) on the consensus hierarchy merit further exploration.

[2]Herlihy and Wing[14] give a more elaborate version of this construction using *swap* and *inc* operations.

6. CONCLUSIONS

Should we dismiss non-readable objects as pathological counterexamples? After all, every universal construction of which I am aware admits a straightforward snapshot operation. These constructions center around a single decisive operation that installs the effects of an operation, so a snapshot will yield a clean picture of the object state. Nevertheless, the effect of adding snapshots to a FIFO queue, and the current absence of a truly practical way to take snapshots suggests that it would be premature to come down on either side.

Leslie Lamport was the first to consider the problem of one thread can atomically read bits written (atomically or not) by another thread. In this paper, I have tried to suggest that the very same question endures in other contexts, and remains worth pursuing.

7. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, Sept. 1993.

[3] Anderson. Composite registers. In *PODC: 9th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1990.

[4] Anderson. Multi-writer composite registers. *DISTCOMP: Distributed Computing*, 7, 1994.

[5] Anderson and Groselj. Beyond atomic registers: Bounded wait-free implementations of nontrivial objects. *SCIPROG: Science of Computer Programming*, 19, 1992.

[6] H. Attiya and O. Rachman. Atomic snapshots in o(n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, Mar. 1998.

[7] E. Borowsky and E. Gafni. A simple algorithmically reasoned characterization of wait-free computations (extended abstract). In *Symposium on Principles of Distributed Computing*, pages 189–198, 1997.

[8] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of the wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 334–343, 1994.

[9] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. *SIAM J. Comput.*, 28(5):1848–1874, 1999.

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed commit with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[11] J. Guttag and B. Liskov. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[12] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages And Systems*, 13(1):123–149, Jan. 1991.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[14] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1993.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[17] L. Lamport. On interprocess communications. *Distributed Computing*, 1:86–101, 1986.

[18] M. Li, J. Tromp, and P. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(6):723–746, 1196.

[19] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, 1986.

[20] S. Moran and L. Rappoport. On the robustness of h_m^r. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, volume 1151 of *LNCS*, pages 344–361, 1996.

[21] O. Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 156–163, 1994.

[22] E. Ruppert. Determining consensus numbers. *SIAM Journal on Computing*, 30(4):1156–1168, 2000.