Binary Snapshots

J-H Hoepman, J. Tromp

Computer Science/Department of Algorithmics and Architecture

CS-R9319 1993

Binary Snapshots

Jaap-Henk Hoepman
jhh@cwi.nl

John Tromp
tromp@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

This paper considers the shared memory wait-free *atomic snapshot* object in its simplest form where each cell contains a single bit. We demonstrate the 'universality' of this binary snapshot object by presenting an efficient linear-time implementation of the general multibit atomic snapshot object using an atomic binary snapshot object as a primitive. Thus, the search for an efficient (subquadratic or linear time) wait-free atomic snapshot implementation may be restricted to the binary case.

AMS Subject Classification (1991): 68M10, 68Q22, 68Q25
CR Subject Classification (1991): B.3.2, B.4.3, C.1.2, D.1.3, D.4.1, E.1
Keywords & Phrases: snapshot, shared memory, atomicity, linearisability, wait-free implementations
Note: Partially supported by the Dutch foundation for scientific research (NWO) through NFI Project ALADDIN, under contract number NF 62-376

1. Introduction

Consider a concurrent shared memory system. A snapshot memory object shared between n processes is a vector of n memory cells, one 'owned' by each process. All processes can independently and concurrently write to (*update*) the cell they own, and all processes can 'instantaneously' collect (*scan*) all values in the vector in a single operation.

The problem of implementing a wait-free atomic snapshot object was independently proposed and solved by Anderson [And89a, And89b, And90] and Afek et al. [AAD$^+$90]. Anderson gives an exponential time[1] solution to this problem using single-writer multi-reader registers, and also considers the multi-writer case in which more than one process may update a particular cell. In his solution for the multi-writer case he uses the single-writer snapshot object as a primitive, so his solution does not rely on multi-writer multi-reader registers. Afek et al. give a polynomial time implementation of a single-writer atomic snapshot object, also using single-writer multi-reader registers. They also consider the multi-writer case, but give a solution using multi-writer multi-reader registers instead.

[1] In the literature on this subject the time complexity is usually measured by the number of shared register accesses per operation as a function of the number of processes.

The atomic snapshot memory object is a powerful tool to construct other atomic wait-free objects, for instance counters, logical clocks, or bounded concurrent time-stamp schemes. Aspnes and Herlihy [AH90] give a general method to convert a sequential specification of a shared memory object that satisfies certain constraints to a wait-free implementation of that object using an atomic snapshot memory object as a primitive. They also give a a polynomial-time implementation of a wait-free atomic snapshot object.

The main question remains whether it is possible to deterministically implement an atomic snapshot object with single-writer multi-reader registers such that the time complexity of both the update and the scan operations is linear. Much research has focused on affirming this, by imposing certain restrictions on the applicability of the solutions. In [KST91], Kirousis et al. present a linear-time solution for the case in which no two scans ever overlap. Dwork et al. [DHPW92] introduce the weaker time-lapse snapshot object, and give a linear time implementation of this object. Time-lapse snapshots satisfy the same properties atomic snapshots do, except that the former allow concurrent scans to contradict each other. In [ISS92], Israeli et al. present linear-time implementations for either the update or the scan operations, or for unbalanced systems in which the number of updaters is substantially smaller than the number of scanners, or vice versa. Finally, Attiya et al. [AHR92] introduce the lattice agreement decision problem and show that a solution to this problem can be converted to a wait-free atomic snapshot implementation.

In this paper we take a similar approach, and reduce the general atomic snapshot problem to a simpler one. We present a bounded, linear time construction of a wait-free implementation of the general atomic snapshot object from an atomic wait-free *binary* snapshot object (where each cell can contain only two values) and a small amount of safe and regular single-writer registers. Thus the search for an efficient atomic snapshot implementation may be restricted to the binary case.

We will use a proof technique proposed in [AKKV88], also used in [LTV89] to prove the correctness of some atomic register constructions. The technique is a derivation of Lamport's system as described in [Lam86], where his two precedence relations *precedes* and *can affect* are replaced by a single interval order. We first present the model in section 2, then we state the atomic wait-free snapshot problem in section 3. The protocol is presented in section 4, and is proven correct in section 5.

2. The Model

A concurrent shared memory system is a collection of sequential processes communicating asynchronously through shared memory data structures. At any time a process is executing at most one action. A process can at any time decide to start a new action when it is idle, or to finish an ongoing action. The start time of an action a is denoted by $s(a)$ and the finish time by $f(a)$.

We model an execution of the shared memory system by a tuple $\langle \mathcal{A}, \rightarrow \rangle$, where $\mathcal{A}$ is the set of all executed actions ordered by $\rightarrow$ such that a precedes b, $a \rightarrow b$, if $f(a) < s(b)$. We require for any execution $\langle \mathcal{A}, \rightarrow \rangle$ that for any $a \in \mathcal{A}$ there are only a finite number of actions $b \in \mathcal{A}$ with $\neg(a \rightarrow b)$. This way we require an execution to start at some point in time, rather than extending into the infinite past [Lam86, AKKV88]. With this definition, $\rightarrow$ is a special kind of partial order called an *interval order* (i.e. a transitive binary relation such that if $a \rightarrow b$ and $c \rightarrow d$ then $a \rightarrow d$ or $c \rightarrow b$). Now we have abstracted away from the actual time an action occurred, and we can specify the behaviour of actions involving access to the shared memory in terms of the interval order.

If one wishes to implement a certain *compound* shared memory object, one first assumes a set of *primitive* shared memory objects used in the implementation. Every operation on the compound object is implemented by a *protocol* which invokes actions on these primitive objects. Using the compound object will result in an *implementation* execution. Since every operation on the compound object is implemented by a sequence of actions on the primitive objects, an implementation execution induces a *basic* execution $\langle \mathcal{A}, \rightarrow \rangle$ on the shared memory system. In an implementation execution we model an operation as the set of actions it invokes. The implementation execution itself is modeled by a tuple $\langle \mathcal{O}, \xrightarrow{o} \rangle$, where $\mathcal{O}$ contains all operations invoked during the execution, and where for operations $A, B \in \mathcal{O}$, $A \xrightarrow{o} B$ iff all actions $a \in A$ precede all actions $b \in B$ in $\langle \mathcal{A}, \rightarrow \rangle$.

3. Atomic Snapshot Memories

A snapshot memory object on n processes is a vector of n memory cells. A process P_i can both write a new value to the i-th cell in the vector or instantaneously collect all values in the vector in a single operation. In the first case it performs an *update-operation*, in the latter case it performs a *scan-operation*.

We require our implementation to be *wait-free* to allow maximal concurrency, and failure-resiliency in the case of crash-failures. An implementation is wait-free if and only if all update and scan operations performed by any process will complete in an a priori bounded number of steps, regardless of the behaviour of the other processes.

Secondly, we require our implementation to be *atomic*. This means that all operations must appear to take effect at one instant of time during the actual time the operation executed[2]. This allows us to 'shrink' the actual execution interval of an operation to a point, and we require a scan to return the values written by the most recent preceding updates. The next paragraph formalises this.

Let $\mathcal{O}$ be the set of all scan and update operations invoked in an implementation execution $\langle \mathcal{O}, \xrightarrow{o} \rangle$ of a snapshot object. Assume for ease of presentation that $\mathcal{O}$ includes n initialising updates, one per processor, that precede all other operations in $\mathcal{O}$. The implementation of an atomic snapshot object is *correct* if for any of its executions $\langle \mathcal{O}, \xrightarrow{o} \rangle$ we can extend $\xrightarrow{o}$ to a total order $\xRightarrow{o}$ such that for all scan operations $S \in \mathcal{O}$, S returns for any cell i the value written by the last update $U_i \in \mathcal{O}$ executed by P_i preceding S in $\xRightarrow{o}$.

4. The Solution

In the next two sections we give our implementation of the n process wait-free atomic snapshot object. The *architecture* describes all primitive shared memory objects used by the *protocols*—one for each type of operation on the shared memory object. The architecture

[2]Although here we refer to the global time model for its more intuitive appeal, we will actually prove atomicity by linearisation (cf. [Lam86], and the previous section).

also specifies the initial values of the primitive objects, the operations each process is allowed to perform on them, and the type of values it holds.

The intuition behind our implementation is quite straightforward: Suppose update operations of P_i write the new value alternatingly to two registers $val_i[0]$ and $val_i[1]$ (this idea was independently put forward by Haldar and Vidyasankar [HV92]), after which they use an update on the binary snapshot to inform the scans of the position they wrote to. A scan first performs a scan on the binary snapshot, and tries to read the values from the registers val_i at the positions returned by the binary scan. As later updates may overwrite values before they are read by a concurrent scan, updates perform a scan operation as well, the result of which they write in the register $view_i$. A scan uses a *handshaking* mechanism to detect overwriting updates, in which case it copies the view written by an interfering update.

4.1 The Architecture

Our implementation of an n process atomic snapshot memory—with cells of type T—will use one n process binary atomic snapshot object with operations $B\text{-}Update_i$ and $B\text{-}Scan_i$, performed by process P_i. Each cell of this binary snapshot object is initially 0. In addition to this, our n-process atomic snapshot protocol will use the following shared registers. For each $i \in \{1, \ldots, n\}$:

- 2 safe registers of type T, $val_i[0]$ and $val_i[1]$, written by process P_i and read by all. Initially, $val_i[1]$ may be arbitrary, but $val_i[0]$ must be initialised to the desired initial value of cell i of the snapshot vector.
- 1 regular register, $view_i$ (an n-value vector with elements of type T), written by process P_i and read by all, initially arbitrary.
- for each $j \in \{1, \ldots, n\}$: a safe bit c_{ij} (the 'complement'-bit), an atomic bit s_{ij} (the 'start'-bit) and a regular bit e_{ij} (the 'end'-bit). All written by process P_i, read by process P_j and initially 0.

4.2 The Protocols

Each of the n processes P_i can execute both updates and scans according to the following protocols

```
**Procedure** $Update_i(value)$
    $b := 1 - b$
    write $val_i[b] \leftarrow value$
    $B\text{-}Update_i(b)$
    **for** $j \in \{1, \ldots, n\}$ **do**
        write $s_{ij} \leftarrow$ (read $c_{ji}$)
    write $view_i \leftarrow Scan_i$
    **for** $j \in \{1, \ldots, n\}$ **do**
        write $e_{ij} \leftarrow s_{ij}$
```

```
**Procedure** $Scan_i$
    **for** $j \in \{1, \ldots, n\}$ **do**
        write $c_{ij} \leftarrow 1-$(read $s_{ji}$)
    $b[1..n] := B\text{-}Scan_i$
    **for** $j \in \{1, \ldots, n\}$ **do**
        read $v[j] \leftarrow val_j[b[j]]$
        **if** $c_{ij} =$ (read $s_{ji}$) $=$ (read $e_{ji}$)
        **then return** (read $view_j$)
    **return** $v[1..n]$
```

The Update-protocol uses local variables j (ranging over $\{1, \ldots, n\}$), and b, a static bit variable initially 0, which retains its value inbetween successive invocations of the protocol.

The Scan-protocol uses local variables b (an n-bit vector), j (ranging over $\{1, \ldots, n\}$), and v (an n-value vector with elements of type T).

A few words on the programming notation are in order. Some assignments involve both a write and a read or Scan. These are to be executed sequentially, the read/Scan first and then the write. E.g. 'write $s \leftarrow$ (read c)' is shorthand for ' read $t \leftarrow c$; write $s \leftarrow t$'. This should not to be confused with read-modify-write operations that execute atomically. We assume that the value of a shared register written by a process also belongs to that process's local state. This means that the value of for instance the shared variable c_{ij} in the Scan-protocol need not be explicitly read. The return statements in the Scan-protocol serve to return the indicated value to the caller, and to terminate the protocol immediately.

The for loops are indexed over a set to make clear that the n loop bodies may be interleaved arbitrarily. Since the registers accessed in the loop bodies are all disjoint, such a for statement can also be interpreted as a do-in-parallel construct. Thus the parallel time complexity [AGTV92] of snapshots equals the parallel time complexity of binary snapshots (up to a constant factor).

5. Proof of Correctness

To prove correctness we assume the usual correctness conditions on the read write registers that we use in our implementation. We also assume the correctness of the atomic binary snapshot object used by our implementation. I.e. in an execution $\langle \mathcal{A}, \rightarrow \rangle$ we assume there exists a total order $\Rightarrow$ extending $\rightarrow$ such that every binary scan BS returns for bit i the value written by the last binary update BU_i executed by P_i preceding BS in $\Rightarrow$.

We write U for *Update* and S for *Scan*. For operation $O \in \{U, S, BU, BS\}$, O_i^x denotes the x-th execution of O by process P_i, including scans S_i that are invoked by some update U_i^y. These scans are sometimes written as US_i^y. Note that BS_i^x is invoked by S_i^x, and BU_i^x is invoked by U_i^x. $\mathcal{O}$ contains all invocations S_i^x and U_i^x for $i \in \{1, \ldots, n\}$ and $x \geq 0$. Note that this also includes updates U_i^0 that wrote the initial values for the cells i, and scans US_i^x invoked by updates U_i^x.

If scan S_i^x sees $c_{ij} =$ (read s_{ji}) $=$ (read e_{ji}), then process P_j (or some update U_j) is said to *interfere* with S_i^x. A scan is *direct* if no process interferes with it. S_i^x *contains* S_j^y iff $s(S_i^x) < s(S_j^y) < f(S_j^y) < f(S_i^x)$. The next lemma shows that direct scans will return correct values.

Lemma 1 *Assume P_j does not interfere with some scan S_i^x, and let S_i^x scan the value $b[j]$ updated by some U_j^y, i.e. $BU_j^y \Rightarrow BS_i^x \Rightarrow BU_j^{y+1}$. Then the value $val_j[b[j]]$ read by S_i^x was written there by U_j^y.*

Proof: Assume scan S_i^x does not see $c_{ij} =$ (read s_{ji}) $=$ (read e_{ji}) and that BS_i^x scanned the value $b[j]$ for cell j updated by BU_j^y, i.e. $BU_j^y \Rightarrow BS_i^x \Rightarrow BU_j^{y+1}$.

The write of $val_j[b]$ by U_j^y precedes BU_j^y in $\rightarrow$, and the read of $val_j[b]$ by S_i^x follows BS_i^x in $\rightarrow$. Since $BU_j^y \Rightarrow BS_i^x$, we have $\neg(BS_i^x \rightarrow BU_j^y)$, so the write of $val_j[b]$ by U_j^y precedes the read of it by S_i^x. So if S_i^x does not read the value written by U_j^y it must be concurrent with or occur after a write to $val_j[b]$ by a later update U_j^z Note that this later update cannot be U_j^{y+1}, since this update will write to $val_j[1-b]$.

Suppose the read of $val_j[b]$ is concurrent with or occurs after a write to it by an update U_j^z, $z > y+1$. Now $BS_i^x \Rightarrow BU_j^{y+1}$, so by a similar argument as before the read of c_{ij} by U_j^{y+1} occurs after the write of c_{ij} by S_i^x in $\rightarrow$. U_j^{y+1} writes the value of c_{ij} to s_{ji} and later to e_{ji} before S_i^x reads these, since the read of $val_j[b]$ by S_i^x is concurrent with or occurs after a write to it by update U_j^z. Now the values of c_{ij}, s_{ji} and e_{ji} must be equal, and as long as S_i^x does not finish, c_{ij} will not change. This implies that any later writes to s_{ji} and e_{ji} will not change their value and thus, as they are atomic and regular, S_i^x should see $c_{ij} =$ (read s_{ji}) $=$ (read e_{ji}), a contradiction. ■

The next lemma shows that scans that cannot collect the values directly due to interfering updates can copy the result from such an interfering update. This interfering update will have stored the result, called a view, of a direct scan contained in the interfered scan.

Lemma 2 *If process P_j interferes with scan S_i^x, then the view S_i^x copied from $view_j$ is the result of a direct scan S_k^z, contained in S_i^x.*

Proof: Since S_i^x sees $c_{ij} =$ (read s_{ji}) and s_{ji} is atomic and S_i^x sets $c_{ij} = 1-$(read s_{ji}), there must be an update U_j^y that changed s_{ji} after S_i^x read s_{ji}. This implies that the scan US_j^y of U_j^y started after S_i^x did. Note that after U_j^y changes s_{ji}, e_{ji} holds the old value of s_{ji} which is unequal to the current value of s_{ji}. Then if S_i^x also sees $c_{ij} =$ (read e_{ji}), U_j^y must have written e_{ji} before or concurrent with the read of e_{ji} by S_i^x. This implies that S_i^x reads $view_j$ after the result of US_j^y was written to it by U_j^y. This also shows that S_i^x contains this US_j^y. Note that $view_j$ must be a regular register, since views written by later updates may interfere with the read of the view by S_i^x. ■

We conclude by proving the correctness of our implementation of the atomic snapshot object. The implementation is obviously wait-free.

Theorem 3 *For any execution $\langle \mathcal{O}, \xrightarrow{o} \rangle$ there exists a total extension $\overset{o}{\Longrightarrow}$ of $\xrightarrow{o}$ such that any scan S_i^x with $U_j^y \overset{o}{\Longrightarrow} S_i^x \overset{o}{\Longrightarrow} U_j^{y+1}$ returns for cell j the value written by U_j^y.*

Proof: For direct scans S_i^x, let $\beta(S_i^x) = BS_i^x$. For indirect scans S_i^x that copied the view collected by a direct scan S_j^y (see lemma 2), let $\beta(S_i^x) = BS_j^y$. Finally, for updates, let $\beta(U_i^x) = BU_i^x$.

For any two $A, B \in \mathcal{O}$, define $A \overset{o}{\Longrightarrow} B$ if $\beta(A) \Rightarrow \beta(B)$. Note that neither $A \overset{o}{\Longrightarrow} B$ nor $B \overset{o}{\Longrightarrow} A$ iff $\beta(A) = \beta(B)$. By lemma 2, $\beta(S)$ occurs inside S for any indirect scan S. This implies that if $A \xrightarrow{o} B$ we have $\beta(A) \Rightarrow \beta(B)$ and thus $A \overset{o}{\Longrightarrow} B$. So $\overset{o}{\Longrightarrow}$ extends $\xrightarrow{o}$. Now extend $\overset{o}{\Longrightarrow}$ to a total order.

If for some scan S_i^x, $U_j^y \overset{o}{\Longrightarrow} S_i^x \overset{o}{\Longrightarrow} U_j^{y+1}$, then $BU_j^y \Rightarrow \beta(S_i^x) \Rightarrow BU_j^{y+1}$ by the definition of β and $\overset{o}{\Longrightarrow}$ (Note that if $\beta(A) = \beta(B)$, then both A and B are scans). If S_i^x is a direct scan, then $\beta(S_i^x) = BS_i^x$ and by lemma 1 the theorem is proved. If S_i^x is not a direct scan, then it copied the result from a direct scan S_k^z, and thus $\beta(S_i^x) = BS_k^z$. But again by lemma 1 the theorem is satisfied. ■

6. Future Research

Further research might be directed at finding an implementation of atomic binary snapshots with subquadratic or linear time complexity.

It is interesting to note that all atomic snapshot implementations we are aware of use at least $O(n)$ registers with $O(nv)$ size (where v is the maximal number of bits contained in any cell of the snapshot object). However, Dwork et al. [DHPW92] have shown that for time-lapse snapshots $O(n^2)$ registers with size $O(n+v)$ suffice. It is an interesting open question whether registers with size $O(nv)$ are necessary to implement atomic snapshot objects.

References

[AAD+90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *9th PODC*, pages 1–13, Aug. 1990.

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *6th WDAG*, LNCS 647, pages 85–94. Springer Verlag, Nov. 1992.

[AH90] J. Aspnes and M. P. Herlihy. Wait-free data structures in the asynchronous pram model. In *2nd PAAA*, pages 340–349, July 1990.

[AHR92] H. Attiya, M. Herlihy, and O. Rachman. Efficient atomic snapshots using lattice agreement. In *6th WDAG*, LNCS 647, pages 35–53. Springer Verlag, Nov. 1992.

[AKKV88] B. Awerbuch, L. M. Kirousis, E. Kranakis, and P. M. B. Vitányi. On proving register atomicity. In *8th FSTTCS*, pages 286–303, 1988.

[And89a] J. H. Anderson. Composite registers. Technical Report TR-89-25, Department of Computer Science, The University of Texas at Austin, Sept. 1989.

[And89b] J. H. Anderson. Multiple-writer composite registers. Technical Report TR-89-26, Department of Computer Science, The University of Texas at Austin, Sept. 1989.

[And90] J. H. Anderson. Composite registers. In *9th PODC*, pages 15–29, Aug. 1990.

[DHPW92] C. Dwork, M. Herlihy, S. A. Plotkin, and O. Waarts. Time-lapse snapshots. In *Israel Symposium Theory of Computing and Systems*, LNCS 601, pages 154–170. Springer Verlag, May 1992.

[HV92] S. Haldar and K. Vidyasankar. Elegant constructions of atomic snapshot variables. Unpublished manuscript, May 1992.

[ISS92] Amos Israeli, Amnon Shaham, and Asaf Shirazi. Linear-time snapshot protocols for unbalanced systems. Technical Report CS-R9236, CWI, Sept. 1992.

[KST91] L. M. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *5th WDAG*, LNCS 579, pages 229–241. Springer Verlag, Oct. 1991.

[Lam86] L. Lamport. On interprocess communication part i: basic formalism. *Distr. Comput.*, 1(2):77–85, 1986.

[LTV89] M. Li, J. Tromp, and P. M. B. Vitányi. How to share concurrent wait-free variables. Technical Report CS-R8916, CWI, Apr. 1989.