How Hard Is It to Take a Snapshot?

Faith Ellen Fich

University of Toronto
Toronto, Canada
fich@cs.utoronto.ca

Abstract. The snapshot object is an important and well-studied primitive in distributed computing. This paper will present some implementations of snapshots from registers, in both asycnhronous and synchronous systems, and discuss known lower bounds on the time and space complexity of this problem.

1 Introduction

An important problem in shared memory distributed systems is to obtain a consistent view of the contents of the shared memory while updates to the memory are happening concurrently. For example, in a sensor network, simultaneous readings of all the sensors may be needed, even when the measured readings change frequently. In a distributed computation, periodically recording the global state can facilitate error recovery, since the computation can be restarted from the most recent backup, rather than from the beginning of the computation. Ideally, one wants to collect these checkpoints without interrupting the computation. Checkpoints can also be useful for debugging distributed programs, allowing invariants that have been violated to be found more easily.

To obtain a consistent view, it is not enough to simply collect the entire contents of memory by reading one value at a time. The difficulty is that the first values that have been read may be out of date by the time that later values are read. The problem can be formalized as the implementation of a snapshot object that can be accessed concurrently by different processes. A snapshot object consists of a set of $m > 1$ components, each capable of storing a value. Processes can perform two different types of operations: `UPDATE` any individual component or atomically `SCAN` the entire collection to obtain the values of all the components. A single-writer snapshot object is a restricted version in which there are the same number of processes as components and each process can UPDATE only one component.

It is often much easier to design fault-tolerant algorithms for asynchronous systems and prove them correct if one can think of the shared memory as a snapshot object, rather than as a collection of individual registers. Snapshot objects have been used to solve randomized consensus [3, 4], approximate agreement [8], and to implement bounded concurrent time stamps [15] and other types of objects [5, 16].

M. Bieliková et al. (Eds.): SOFSEM 2005, LNCS 3381, pp. 27–36, 2005.

Unbounded time stamps are particularly easy to generate using a single-writer snapshot object. Each process stores the time stamp from its last request (or 0, if has not yet requested one) in its component. To obtain a new time stamp, a process performs a SCAN and adds 1 to the maximum time stamp contained in the result. Then the process updates its component with its new time stamp. This ensures that a request that is started after another request has completed will receive a larger time stamp. Note that concurrent requests can receive the same or different time stamps.

Because snapshot objects are much easier for programmers to use, researchers have spent a great deal of effort on finding efficient implementations of them from registers, which, unlike snapshot objects, are provided in real systems.

Our asynchronous shared memory system consists of n processes, $p_1, \ldots, p_n$ that communicate through shared registers. At each step of a computation, one process can either read from one register or write a value to one register. In addition, it can perform a bounded number of local operations that do not access shared registers. The order in which processes take steps is determined by an adversarial scheduler. In synchronous shared memory, at each step, every active process either reads from one register or writes a value to one register, in addition to performing a bounded amount of local work. An active process is a process that has neither completed its task nor crashed. The order in which the shared memory operations occur during a step is determined by an adversarial scheduler.

We consider linearizable (atomic) implementations [17], where each SCAN and UPDATE operation appears to take effect at some instant during the time interval in which it is executed. Implementations should also be wait-free. This means that there is an upper bound on the number of steps needed for any process to perform any operation, regardless of the behaviour of other processes. In particular, processes do not have to wait for other processes to finish. Thus wait-free implementations tolerate crash failures.

The rest of this paper will present an overview of some different implementations of snapshot objects from registers and discuss known lower bounds. Both the time complexity and the number of registers are considered, but not the size of the registers. Various techniques are known to reduce register size [9, 11, 12, 15, 19], but these are beyond the scope of this paper.

2 An Asynchronous Implementation Using m Registers

Our first implementation uses one shared register for each component of the snapshot object. Although simply collecting the values of the registers may give an invalid view, a valid view *is* guaranteed if two collects are performed by some process and, meanwhile, the values of none of the registers are changed. Note that the value of a register may change and then change back again. To distinguish between this possibly bad situation and the situation when no changes have occurred, each process performing an UPDATE to a component writes its

identifier and a sequence number to the corresponding register together with the new value.

However, a process can repeatedly collect the values without ever getting two consecutive identical results. To overcome this problem, when a process performs UPDATE, it first performs SCAN and writes the result, together with the new value, its identifier, and a sequence number to the appropriate register. Now, when a process performs two collects that have different results, it could use the result of a SCAN that is stored in one of the registers.

Unfortunately, the stored result may be out of date: It may be a memory state that only occurred before the process began its SCAN. To ensure that a process p performing a SCAN gets a memory state that occurs while it is executing, it repeatedly performs collects until either two consecutive collects are the same (in which case, the result of this collect is its result) or it sees that some other process has performed at least two different UPDATES since p performed its first collect. In the second case, p returns the last result written by this other process, which is from a SCAN that is part of an UPDATE that began after p began its SCAN.

One of these two cases will occur by the time a process has performed $n + 2$ collects. Each collect takes takes $\Theta(m)$ steps. Thus the time complexity of a SCAN is $O(mn)$. Since an UPDATE contains an embedded SCAN, its time complexity is also $O(mn)$. This algorithm, although presented in a slightly different form, is from Afek, Attiya, Dolev, Gafni, Merritt, and Shavit's paper [1]. It also appears in [13].

3 An Implementation of a Snapshot Object From a Single-Writer Snapshot Object

Anderson [2] shows how to construct a snapshot object with m components shared by n processes using a single-writer snapshot object shared by n processes. The idea is that each process stores information about every component of the snapshot object in its component of the single-writer snapshot object. Specifically, it stores the value of the last UPDATE operation it performed to each component, together with a time stamp for each of those UPDATES.

To perform a SCAN of the snapshot object, a process performs a SCAN of the single-writer snapshot object and, for each component, returns the value with the latest time stamp. Process identifiers are used to break ties.

To perform an UPDATE to component i of the snapshot object, a process performs a SCAN of the single-writer snapshot object. The result of the SCAN gives it the time stamp of the last UPDATE to component i performed by each process. It generates a new time stamp that is later than all of these. Then the process UPDATES its single-writer component with the new value and time stamp for component i and the old values and time stamps for all other components.

Thus, if there is an implementation of a single-writer snapshot object in which SCAN takes time $S(n)$ and UPDATE takes time $U(n)$ in the worst case, then

there is an implementation of a snapshot object in which SCAN takes $O(S(n))$ time and UPDATE takes $O(S(n) + U(n))$ time.

4 Asynchronous Single-Writer Snapshot Implementations Using Lattice Agreement

Lattice agreement is a decision problem that is closely related to the single-writer snapshot object. In this problem, each process gets an element of some lattice as input and must output an element that is at least as large, but no larger than the least upper bound of all the inputs. Furthermore, all outputs must be comparable with one another.

It can easily be solved in $O(1)$ time using a single-writer snapshot object: Each process UPDATES its component with its input, performs a SCAN, and takes the least upper bound of all the lattice elements in the result as its output. Conversely, a single-writer snapshot object in which each process performs at most one operation can be implemented using single-writer registers and an instance of the lattice agreement problem [7]. To perform an UPDATE, a process writes the new value to its single-writer register and then performs a SCAN, throwing away the result. To perform a SCAN, a process collects the values in all the single-writer registers and uses this view as an input to lattice agreement, where one view is less than or equal to another view in the lattice if the set of components that have new values in the first view is a subset of those in the other view. A process uses its output from the lattice agreement problem as the result of its SCAN. The reason a process performing an UPDATE with some value v performs an embedded SCAN is to ensure that there isn't a later UPDATE by another process whose new value is collected by some SCAN that doesn't collect v.

Attiya, Herlihy, and Rachman [7] extend this idea to allow processes to perform an unbounded number of SCAN and UPDATE operations using an unbounded number of instances of lattice agreement. First, each process writes a sequence number to its single-writer register together with its new value. In the lattice, one view is less than or equal to another view if each component of the first view has a sequence number that is less than or equal to the sequence number in the same component of the other view. A process p that wants to perform a SCAN (or an embedded SCAN as part of an UPDATE) uses the view it collects as input to an instance of lattice agreement that it has not already solved. There may be multiple instances of lattice agreement that are active simultaneously. Process p either chooses the latest of these, if it has not already solved it, or starts a new instance. After getting an output from this instance, p checks whether a later instance of lattice agreement is active. If not, it can use this view as its result. Otherwise, it performs a second collect and uses the view from it as input to the latest instance that is now active. If, after getting its output to this instance, there is an even later active instance, there is some other process that participated in the same instance and got a valid view. Then p uses this view as its result.

Inoue, Chen, Masuzawa, and Tokura [18] show how to solve agreement in this lattice in $O(n)$ time. From the discussion above, this implies the existence of an n-component single-writer snapshot implementation from an unbounded number of registers in which SCAN and UPDATE take $O(n)$ time. They consider a complete binary tree of height $\lceil \log_2 n \rceil$, with each process assigned to a different leaf. At its leaf, a process has a view consisting of only one component, which contains its new value. The process proceeds up the tree, at each successive level doubling the length of its view, until, at the root, it has a view that includes all n components. At each internal node of height k, a process solves an instance of lattice agreement among the 2^k processes that are associated with leaves in its subtree, using an input formed from its resulting view at the previous level. This takes $O(2^k)$ time. The fact that the inputs of processes arriving from the same child are comparable with one another ensures that all the outputs are comparable with one another.

Attiya and Rachman [9] also use lattice agreement to show that an n-process single-writer snapshot object can be implemented from single-writer registers so that the time complexities of both `SCAN` and `UPDATE` are in $O(n \log n)$. Like the previous implementation, they use a complete binary tree of height $\lceil \log_2 n \rceil$ that contains a single-writer register for each process at every node. However, their approach to solving lattice agreement is quite different. First, they consider the case when at most n SCAN and UPDATE operations are performed. To perform an UPDATE, a process writes its new value, together with a sequence number, to its single-writer register at the root of the tree and then performs an embedded SCAN. To perform a SCAN, a process traverses the tree from the root down to a leaf. All the processes that end at the same leaf will return the same view. A process collects the views written to the single-writer registers at each internal node it reaches. It decides whether to go left or right depending on how the sum of the sequence numbers it sees compares to some appropriately chosen threshold. A process with a sum that is at or below the threshold goes left and writes the same view at this child as it wrote at the node from which it came. A process that goes right computes a new view by taking the union of the views that it collected and writes this view at this child. In either case, the process repeats the same procedure at the child, until it reaches a leaf. An important observation is that the sequence number in a component of a view written to a right child is at least as large as the sequence number in the same component of a view written to its left sibling. This ensures that the views that processes write at different leaves are comparable with one another. To handle the general case, the operations are divided into virtual rounds, each containing n operations. They use a variety of mechanisms, including bounded counters and handshakes, to separate the operations from different rounds so that they do not interfere with one another.

5 Faster Asynchronous Snapshot Implementations

Israeli, Shaham, and Shirazi [20] present an implementation of an n-component single-writer snapshot object $\mathcal{A}$ with $O(n)$ SCAN time and $O(S(n) + U(n))$

UPDATE time from a single n-component single-writer snapshot object $\mathcal{B}$ with SCAN time $S(n)$ and UPDATE time $U(n)$ and n single-writer registers. Each process uses its component in the base object $\mathcal{B}$ to store both the current value of its component and a sequence number that it increments each time it performs an UPDATE. After performing an UPDATE, a process then SCANS $\mathcal{B}$ and writes the resulting view to its register. Since $\mathcal{B}$ is linearizable, all the views that are written can be totally ordered, using the sequence numbers contained in the views. To perform a SCAN of $\mathcal{A}$, a process simply reads the views written in the registers and chooses the most recent one. Applying the transformation in Section 3 yields a snapshot object with $O(n)$ SCAN time and $O(S(n)+U(n))$ UPDATE time. In particular, using Attiya and Rachman's implementation in Section 4 yields a snapshot object with $O(n)$ SCAN time and $O(n \log n)$ UPDATE time.

The same paper also gives a similar implementation of an n-component single-writer snapshot object $\mathcal{A}'$ with $O(n)$ UPDATE time and $O(n + S(n) + U(n))$ SCAN time. Here, each process stores a possibily inconsistent view in its register. To UPDATE $\mathcal{A}'$, a process reads the views of all processes and, from them, constructs a new view whose i'th component contains the maximum of the sequence numbers in component i together with its associated value. It increments the sequence number in its own component of this view and replaces its value with the new value. It then writes this view to its register. To SCAN $\mathcal{A}'$, a process first reads the registers of all processes and constructs a new view, as above. It UPDATES its component of the base object $\mathcal{B}$ with this view and then SCANS $\mathcal{B}$. Finally, it constructs a new view, as above, from the views returned by the SCAN, which it returns as its result. Again, using Attiya and Rachman's implementation in Section 4 yields a single-writer snapshot object with $O(n)$ UPDATE time and $O(n \log n)$ SCAN time.

6 Lower Bounds for Asynchronous Snapshots

Jayanti, Tan, and Toueg [23] use a carefully constructed covering argument to prove that, in any implementation of an n-component single-writer snapshot object, there is an execution of a SCAN that must read from at least $n - 1$ different registers. Since a $\min\{m, n\}$-component single-writer snapshot object is a special case of an m-component snapshot object shared by n processes, this implies a lower bound of $\min(m, n) - 1$ on the space and SCAN time of any implementation of the latter.

Fatourou, Fich, and Ruppert [14, 13], using a different covering argument, prove that any implementation of an m-component snapshot object shared by $n \geq m$ processes requires at least m registers, matching the upper bound in Section 2. They introduce the concept of a *fatal configuration*, which has a set of processes each of which is about to write to a different register and are in the midst of performing UPDATES to a smaller number of components. No correct implementation using m or fewer registers can reach a fatal configuration; otherwise, there is an insufficient number of registers to record UPDATES to

the other components. This implies a significant amount of information about such implementations: SCANS never write to registers, each UPDATE operation writes to only one register, UPDATE operations to the same component write to the same register, and UPDATE operations to different components write to different registers. Hence, at least m registers are needed.

Using these results together with an intricate covering argument, they prove that, for $m < n$, any space-optimal implementation takes $\Omega(mn)$ time to perform a SCAN in the worst case. This also matches the upper bound in Section 2. For $m = n - 1$, the lower bound is $\Omega(n^2)$, in contrast to the implementations in Section 4 that use single-writer registers and run significantly faster. Note that any implementation using single-writer registers only needs to use one single-writer register per process: All of the single-writer registers into which a process can write can be combined into one single-writer register having many fields.

Israeli and Shirazi [IS98] proved an $\Omega(n)$ lower bound on UPDATE time for implementations of an n-component single-writer snapshot object using only single-writer registers. Their proof uses an elegant combinatorial argument. This can be extended to an $\Omega(m)$ lower bound on UPDATE time for implementations of an m-component snapshot object from m (multi-writer) registers.

7 Synchronous Snapshot Implementations and Lower Bounds

There is a simple implementation of a synchronous m-component snapshot object due to Neiger and Singh [24] in which SCAN and UPDATE each take $m+1$ steps in the worst case. It uses m registers, each of which holds the current value of one component. Their idea is to divide time into blocks of $m + 1$ slots, one for reading each component and one for writing. To perform an UPDATE, a process waits until the next write slot and then writes the new value into the appropriate register. To perform a SCAN, a process just reads the values from the registers in the order specified by the next m read slots. It is possible to linearize every operation that includes a write slot in some order during that write slot so that all the views that result are consistent. They also prove that in any implementation using m registers, each of which can hold the value of only one component, $S(n) + U(n) \in \Omega(m)$, where $S(n)$ is the worst case SCAN time and $U(n)$ is the worst case UPDATE time.

A small variant of this idea allows SCANS to be performed in one step, while UPDATES take at most $2m + 1$ steps. In this case, one (larger) register holds the value of the entire snapshot object, which can be read at any time. Time is divided into blocks of $2m$ slots. A process that wants to UPDATE component i reads the register during slot $2i-1$ and then write backs its contents in the next step, with its new value replacing the old value of component i. Brodsky and Fich [10] show that, with $2m-1$ registers, it is possible to improve the UPDATE time to $O(\log m)$. The registers are conceptually organized as a strict balanced binary tree, where each of the m leaves contains the value of a different component and, provided no UPDATES are in progress, each internal node contains

the concatenation of the values of its children. A process performs a SCAN by reading the register at the root. To perform an UPDATE, a process writes the new value to the corresponding leaf and then propagates this information up the tree to the root. Information from its siblings and from the siblings of each of its ancestors is also propogated to the root at the same time.

Brodsky and Fich present another implementation in which UPDATE takes at most 3 steps and SCAN takes at most $3m - 1$ steps. It uses m registers, one for each component. Each register has two fields, a top-field and a bottom-field, each of which hold a component value. Time is divided into two alternating phases, each consisting of m consecutive steps. During top-write phases, a process performing an UPDATE writes its new value to the top-field of the specified component and processes performing SCANS read from the bottom-fields. Similarly, during bottom-write phases, a process performing an UPDATE writes its new value to the bottom-field of the specified component and processes performing SCANS read from the top-fields. This ensures that writes to top-fields don't interfere with reads from top-fields and writes to bottom-fields don't interfere with reads from bottom-fields. In addition, each register contains a third field to which processes write when they UPDATE the corresponding component. The information in this field allows a process performing a SCAN to determine whether the top-field or the bottom-field contains the value of the most recent UPDATE to that component.

They also show how to combine their two implementations to perform UPDATE in $O(\log(m/c))$ time and SCAN in $O(c)$ time, for any positive integer $c \leq m$. For $n < m$, they get the same results, but with m replaced by n.

Finally, using an information theoretic argument, they prove a general tradeoff $U(n) \in \Omega(\log(\min\{m, n\}/S(n)))$ between the worst case UPDATE time $U(n)$ and the worst case SCAN time $S(n)$, matching their upper bounds.

8 Conclusions

This paper presents a brief survey of different implementations of snapshot objects from registers, together with some complexity lower bounds. It is not intended to be comprehensive. There are a number of interesting aspects this paper does not cover, for example, randomized implementations [7], implementations that are adaptive to contention among processes [6], and implementations from more powerful objects [7, 25, 22].

Acknowledgements

I am grateful to Hagit Attiya for many interesting discussions about snapshot objects and to Alex Brodsky, Danny Hendler, and Kleoni Ioannidou for helpful comments about this paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, *Atomic Snapshots of Shared Memory*, JACM, volume 40, number 4, 1993, pages 873–890.
2. J. Anderson, *Multi-Writer Composite Registers*, Distributed Computing, volume 7, number 4, 1994, pages 175–195.
3. James Aspnes, *Time- and Space-Efficient Randomized Consensus*, Journal of Algorithms, volume 14, number 3, 1993, pages 414–431.
4. James Aspnes and Maurice Herlihy, *Fast, Randomized Consensus Using Shared Memory*, Journal of Algorithms, volume 11, number 2, 1990, pages 441–461.
5. James Aspnes and Maurice Herlihy, *Wait-Free Data Structures in the Asynchronous PRAM Model*, 2nd ACM Symposium on Parallel Algorithms and Architectures, 1990, pages 340–349.
6. Hagit Attiya, Arie Fouren, and Eli Gafni, *An Adaptive Collect Algorithm with Applications*, Distributed Computing, volume 15, 2002, pages 87–96.
7. Hagit Attiya, Maurice Herlihy, and Ophir Rachman, *Atomic Snapshots Using Lattice Agreement*, Distributed Computing, volume 8, 1995, pages 121–132.
8. Hagit Attiya, Nancy Lynch, and Nir Shavit, *Are Wait-free Algorithms Fast?* JACM, volume 41, number 4, 1994, pages 725–763.
9. H. Attiya and O. Rachman, *Atomic Snapshots in O(n log n) Operations*, SIAM Journal on Computing, volume 27, number 2, 1998, pages 319–340.
10. Alex Brodsky and Faith Ellen Fich, *Efficient Synchronous Snapshots*, 23rd Annual ACM Symposium on Principles of Distributed Computing, 2004, pages 70-79.
11. C. Dwork, M. Herlihy, and O. Waarts, *Bounded Round Numbers*, 12th Annual ACM Symposium on Principles of Distributed Computing, 1993, pages 53–64.
12. C. Dwork and O. Waarts, *Simple and Efficient Concurrent Timestamping or Bounded Concurrent Timestamping are Comprehensible*, 24th Annual ACM Symposium on Theory of Computing, 1992, pages 655–666.
13. Panagiota Fatourou, Faith E. Fich, and Eric Ruppert, *A Tight Time Lower Bound for Space-Optimal Implementations of Multi-Writer Snapshots*, 35th Annual ACM Symposium on Theory of Computing, 2003, pages 259–268.
14. Panagiota Fatourou, Faith E. Fich, and Eric Ruppert, *Space-Optimal Multi-Writer Snapshot Objects Are Slow*, 21st Annual ACM Symposium on Principles of Distributed Computing, 2002, pages 13–20.
15. Rainer Gawlick, Nancy Lynch, and Nir Shavit, *Concurrent Timestamping Made Simple*, Israel Symposium on the Theory of Computing and Systems, LNCS volume 601, 1992, pages 171–183.
16. Maurice Herlihy, *Wait-Free Synchronization*, ACM Transactions on Programming Languages and Systems, volume 13, number 1, 1991, pages 124–149.
17. Maurice Herlihy and Jeannette Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, ACM Transactions on Programming Languages and Systems, volume 12, number 3, 1990, pages 463–492.
18. Michiko Inoue, Wei Chen, Toshimitsu Masuzawa, and Nobuki Tokura. Proceedings of the 8th International Workshop on Distributed Algorithms, LNCS volume 857, 1994, pages 130–140.
19. A. Israeli and M. Li, *Bounded Time Stamps*, Distributed Computing, volume 6, number 4, 1993, pages 205–209.
20. A. Israeli, A. Shaham, and A. Shirazi, *Linear-Time Snapshot Implementations in Unbalanced Systems*, Mathematical Systems Theory, volume 28, number 5, 1995, pages 469–486.

21. A. Israeli and A. Shirazi, *The Time Complexity of Updating Snapshot Memories*, Information Processing Letters, volume 65, number 1, 1998, pages 33–40.
22. Prasad Jayanti, *f-Arrays: Implementation and Applications*, 21st Annual ACM Symposium on Principles of Distributed Computing, 2002, pages 270–279.
23. P. Jayanti, K. Tan, and S. Toueg, *Time and Space Lower Bounds for Nonblocking Implementations*, SIAM Journal on Coomputing, volume 30, 2000, pages 438–456.
24. G. Neiger and R. Singh, *Space Efficient Atomic Snapshots in Synchronous Systems*, Technical Report GIT-CC-93-46, Georgia Institute of Technology, College of Computing, 1993.
25. Yaron Riany, Nir Shavit, and Dan Touitou, *Towards a Practical Snapshot Algorithm*, Theoretical Computer Science, volume 269, 2001, pages 163–201.