Partial Snapshot Objects

Hagit Attiya
Technion

Rachid Guerraoui
EPFL

Eric Ruppert
York University

ABSTRACT

We introduce a generalization of the atomic snapshot object, which we call the *partial snapshot object.* This object stores a vector of values. Processes may write components of the vector individually or atomically scan any subset of the components. We investigate implementations of the latter *partial scan* operation that are more efficient than the complete scans of traditional snapshot objects. We present an algorithm that is based on a new implementation of the *active set* abstraction, which may be of independent interest.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *distributed data structures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; F.2.2 [**Analysis of Algorithms and Problems**]: Nonnumerical algorithms and problems

General Terms

Algorithms, Theory

Keywords

snapshot, active set, asynchronous, wait-free, shared memory

1. INTRODUCTION

A fundamental problem in distributed computing is that of obtaining a consistent view of a collection of shared data while other processes are concurrently updating the data. The naive solution of simply reading different portions of the data piece-by-piece may yield inconsistent results. For example, consider the problem of computing the total assets of a stock portfolio by checking the value of each stock one by one, while, concurrently, the values of the stocks are fluctuating, and stocks are constantly being added to the portfolio or removed from it. The result might exceed the maximum value the portfolio had at any time during the day if each stock is checked when it is at its peak value for the day.

The *snapshot object* [1, 5, 8] was introduced as an abstraction of the problem of obtaining a consistent view of several data items. The snapshot object stores a vector of m components and provides two atomic operations: `update`(i, v), which writes the value v into component i of the vector, and `scan`, which returns the entire contents of the vector. (We focus on multi-writer snapshot objects, where any process is allowed to update any component.) The snapshot object has proved to be an enormously useful abstraction. It has been used as a building block for solving many other problems, including approximate agreement [11], timestamping [16], randomized consensus [6, 7] as well as several concurrent object constructions [8, 17]. It could also be used in garbage collection, debugging distributed programs and storing checkpoints for data recovery.

Many algorithms implementing snapshots have been published in the literature, using either read/write registers or more sophisticated objects. (See [15] for a survey.) However, in all of these implementations `scan` operations remain costly. In many applications of snapshot objects, the total number of components, m, is very large, and this can contribute significantly to the cost of a `scan`.

Often, however, users need a consistent view of only a small portion of the vector. In the stock portfolio example above, the vector might store an entire database of stock information, but individual queries might require a consistent view of only a few entries in the database, for example, the stocks in one person's portfolio, or the stocks for a particular type of industry.

If we know, in advance, the portions of the vector for which a consistent view must be obtained and, furthermore, those portions do not overlap, then the vector can be split into smaller pieces, with each piece stored in a separate snapshot object. However, this solution works only under rather specialized conditions. Such conditions clearly do not hold for the stock example above, where queries are unpredictable and could require views of overlapping portions of the database. Algorithms that use snapshots as a building block often assume, impractically, that the entire memory is a giant snapshot object to simplify the design of the algorithm.

This paper introduces a more flexible kind of snapshot object with the goal of handling unpredictable queries as efficiently as possible. We define a *partial snapshot object* which, just like a traditional snapshot object, stores a vector

SPAA'08, June 14–16, 2008, Munich, Germany.

of m components and allows processes to update a single component. However, unlike traditional snapshot objects, processes may scan any subset of the components. (A formal definition is given in Section 2.1.)

The partial snapshot object is a generalization of an ordinary snapshot object, since an ordinary `scan` operation is equivalent to a partial `scan` of the set of all components of the object. Conversely, a snapshot object trivially implements a partial snapshot object: the components required by any partial `scan` can be extracted from a global `scan` that returns all components. Such implementation would however be wasteful because it does not take advantage of the fact that partial `scans` need only a small amount of information. The motivation of this work is to make the complexity of partial `scan` operations dependent only on the number of components they access (we talk about a *local implementation*) rather than the total number of components in the shared object.

Consider a simple variant of the original non-blocking snapshot algorithm of Afek *et al.* [1]. Each component of the partial snapshot object is represented by a register. To update a component, a process writes the value in the corresponding register (together with its id and a counter). A partial `scan` can be performed by repeatedly reading all registers of the components to be scanned until two sets of reads return identical results. However, individual `scans` may never terminate: a slow scanner can keep seeing different collects if fast `updates` are concurrently being performed. The implementation is thus not wait-free. The classical way to transform such a non-blocking implementation into a wait-free one is to rely on a helping mechanism where every `update` embeds a `scan` whose result is written into the shared memory [1]. A slow scanner can then eventually return the result of one such embedded `scan` that it sees. If we use this helping mechanism to implement a partial snapshot object, we must ensure that the embedded `scans` include enough information to help slow concurrent `scans` produce their outputs, and at the same time avoid gathering too much information, which would be inefficient. (This is not an issue for the original snapshot objects, since all `scans` must return the values of all components.)

We propose here a solution with embedded `scans` that record only the states of components that are actually needed by concurrent partial `scans`. The scanners *announce* which components they are currently attempting to scan, and updaters *consult* these announcements in order to perform their embedded `scans` (these embedded `scans` need not announce the components they are scanning). We use an *active set* abstraction [3] as a building block to handle the announcements.

The active set problem is to maintain a group with dynamic membership. Processes may join and leave the group and perform queries that return a list of the current members of the group. We use a solution to this problem to keep track of the processes that are currently performing partial `scans`. This information is used by the `update` operations to determine which components their embedded `scans` must read. We first show how this approach can be used to obtain an implementation of a partial snapshot object using only registers by adapting the classical snapshot algorithm of Afek *et al.* [1].

The implementation from registers provides a blueprint for the main algorithm of this paper, which gives an implementation of a partial snapshot object from stronger base objects where `scan` operations are local, and `updates` are efficient in an amortized sense.

To obtain this algorithm, we present a new solution to the active set problem which, we believe, is interesting in its own right. We use a compare&swap and a fetch&increment object to expedite the `join` and `leave` operations so that they run in constant time (unlike the original active set implementation of [3]). We also use compare&swaps instead of writes when storing values in the snapshot object to improve the efficiency.

We provide a brief description of the model of computation and a formal definition of the partial snapshot object and active set problem in Section 2. In Section 3, we describe the partial snapshot implementation from registers. In Section 4, we give our new active set algorithm and the partial snapshot implementation that provides local partial `scans`. Sections 5 and 6 provide a discussion of related work and some concluding remarks.

2. MODEL

We use a fairly standard model of asynchronous shared-memory systems. Processes run at arbitrarily varying speeds and may experience halting failures. The processes communicate by accessing linearizable shared objects of various types. Because the objects are linearizable, we can think of an execution as a sequence of steps, where that sequence is obtained by interleaving the steps of different processes, each of which is following its algorithm. The interleaving can be done arbitrarily, and the algorithm must behave correctly for all possible interleavings. In describing our algorithms, we use the convention that names of shared objects begin with a capital letter and names of local objects begin with a lower-case letter.

A distributed implementation of a data structure provides an algorithm for each process to follow to perform each operation on that data structure. Our implementations of partial snapshot objects are *linearizable*. Linearizability means that, in any execution, it is possible to choose a linearization point for each operation during the interval of time that operation is being performed such that the responses given by all operations are the same as they would be if they were performed sequentially in the order of their linearization points. If an execution includes incomplete operations, those operations may or may not be assigned linearization points. Our implementations are *wait-free*, meaning that every process completes its operation within a finite number of its own steps.

We are primarily concerned with the time complexity of implementations. For wait-free implementations, we can measure time complexity in terms of the worst-case number of steps a process must perform to complete an operation. The (worst-case) *amortized time per operation* is the maximum, over all finite executions, of the total number of steps in the execution divided by the number of operations in the execution. We can also state amortized time more precisely in terms of several different types of operations. If there are k different types of operations that can be performed, $op_1, \ldots, op_k$, we can say that the amortized time of the implementation is t_1 per op_1, ..., t_k per op_k if, in any finite execution that has M_i invocations of operations of type op_i (for all i), the total number of steps by all processes is at most $\sum_{i=1}^{k} M_i \cdot t_i$.

An algorithm is *adaptive* if its (possibly amortized) time complexity is independent of the number of processes in the system. Ordinarily, it will depend, instead, on the contention, which can be measured in several ways. The *point contention* of an operation *op*, denoted $\dot{C}(op)$ is the maximum number of processes that run simultaneously at any time during the interval that *op* is active. The notation $\dot{C}$ is used to denote the maximum, over all operations, of $\dot{C}(op)$. A subscript s or u is added to $\dot{C}$ if we are interested only in the number of simultaneous `scan` operations or the number of simultaneous `update` operations, respectively. Thus, $\dot{C}_s$ is the maximum number of `scans` that are ever simultaneously active. The *interval contention* of an operation *op*, denoted $\overline{C}(op)$ is the total number of processes with operations whose active intervals overlap *op*'s interval. Again, $\overline{C}$ refers to the maximum, over all operations, of $\overline{C}(op)$ and subscripts s or u can be added as for point contention.

2.1 Problem Definitions

A partial snapshot object is similar to a snapshot object, but permits the user to scan a subset of the components of the object, rather than requiring all `scans` to return the complete state of the object. More formally, a *partial snapshot object* is a linearizable object that stores a vector from D^m, where D is the domain. It provides two operations:

- `update`(i, v), where $1 \leq i \leq m$ and $v \in D$, changes the ith component of the state to v and returns *ack*, and
- `scan`$(i_1, \ldots, i_r)$, where $r \leq m$ and $1 \leq i_j \leq m$ for all $j \in \{1, \ldots, r\}$, does not change the state of the object and returns the vector $(x_{i_1}, \ldots, x_{i_r})$ if the state of snapshot object is $(x_1, \ldots, x_m)$.

For (partial) snapshot implementations, linearizability means that the value of a component returned by a `scan` is the value written by the `update` to that component with the latest linearization point prior to the linearization point of the `scan` (or the initial value of the component, if no such `update` exists). We say a partial snapshot implementation is *local* if the complexity of the `scan` depends only on the number of components it accesses, rather than m; we also strive to have `updates` with adaptive complexity that is independent of m.

As we shall see, devising local implementations of the partial snapshot object is closely linked to the active set problem [3]. Intuitively, a solution to the *active set* problem keeps track of a set of processes. Processes may join or leave the set and get a list of processes currently in the set. However, if a process is joining or leaving the set while another process is getting the list, the latter process may consider the former in the set or outside the set.

More formally, an active set abstraction provides three operations: `join`, `leave` and `getSet`. The `join` and `leave` operations return *ack*. In any execution, calls to `join` and `leave` by the same process alternate, starting with a `join`. A process is *active* from the time it completes a `join` operation until it next calls `leave`. A process is *inactive* from the time it completes a `leave` operation until it next calls `join`. A process is also called inactive for the period before it begins its first `join`. A process is neither active nor inactive while it is executing `join` or `leave`. The `getSet` operation returns a set S of process ids that contains all active processes, and does not contain any inactive process; it may contain any subset of the processes that are neither active nor inactive.

3. AN IMPLEMENTATION FROM REGISTERS

We now describe how to adapt the snapshot algorithm of Afek *et al.* [1] to achieve an implementation of a partial snapshot object from registers with limited `scan` and `update` complexity. The implementation uses a register to represent each of the components of the partial snapshot object. An `update` to a component is accomplished by writing to the corresponding register. Processes also write their ids and a counter along with the value to be stored. This avoids the ABA problem: no two write operations write exactly the same contents into a register, so if two reads of a register return the same result that register's contents cannot have changed between the two writes. The partial `scan` algorithm repeatedly reads the registers corresponding to the components begin scanned. Each set of reads is called a *collect*. If two collects ever return identical results, the `scan` returns those values. To make the algorithm wait-free there is an additional helping mechanism: each `update` writes the result of a `scan` (called an `embedded-scan`), and a slow scanner can eventually return the result of one such `embedded-scan` that it sees. This result is written along with the value, process id and counter value into a single large register. (If all of this information cannot be stored in a single register, one can instead store a pointer to a set of registers that stores the information, but that will increase the time and space complexity of the algorithm.)

To achieve our goal of low complexity, an `embedded-scan` does not determine the values of all components in the snapshot object, but must find the values of enough components to be useful in helping other `scans` complete. To accomplish this, we use an embedded *partial* `scan` that records only the states of components that are needed by concurrent scans. Thus, scanners must announce which components they are currently attempting to scan, and updaters must read these announcements in order to perform their `embedded-scans`. We use an active set algorithm [3] for these announcements.

The wait-free implementation is given in Figure 1. It uses an array of registers $R[1..m]$ with one element for each component of the snapshot object. It also uses an array of single-writer registers $A[1..n]$, where each process can announce which components it is currently scanning, and the registers required to implement the active set algorithm. The `embedded-scan` operation carries out the `scan` operation, but without announcing the components it is scanning. The result of an `embedded-scan` is a list of index-value pairs (i, v), such that component i of the partial snapshot object has value v at the moment the `embedded-scan` is linearized. In general, the indices appearing in this list will be a superset of the arguments given to the `embedded-scan`. All other variables (*scanners*, *counter*, *view*) are local.

We outline the proof of correctness, which closely follows the proof technique of Afek *et al.* [1]. Each `update` is linearized at its write operation. An `embedded-scan` that terminates by condition (1) in the pseudocode is linearized between its two identical collects. An embedded-scan that terminates by condition (2) is linearized at the same time as the embedded-scan whose result it borrows. Finally, each

```
embedded-scan(i_1, ..., i_r)
    repeatedly read R[i_1], ..., R[i_r] until either
        (1) two sets of reads return the same vector, (x_1, ..., x_r);
            then return ((i_j, first field of x_j))_{1≤j≤r},
        or (2) three different values written by the same process have been seen (in any locations);
            then let (v, view, c, id) be the one of these three values with the highest counter field.
            return view
end embedded-scan
```

```
update(i, v)
    scanners ← getSet
    (i_1, ..., i_r) ← ⋃_{p∈scanners} A[p]
    view ← embedded-scan(i_1, ..., i_r)
    R[i] ← (v, view, counter, id)
    counter ← counter + 1
end update
```

```
scan(i_1, ..., i_r)
    A[id] ← (i_1, ..., i_r)
    join
    ((i'_1, v_1), ..., (i'_k, v_k)) ← embedded-scan(i_1, ..., i_r)
    leave
    component j of the result vector is v_ℓ, where i'_ℓ = i_j
end scan
```

Figure 1: A wait-free implementation from registers

scan is linearized at the same time as its embedded-scan.

We argue that each `embedded-scan` returns a result consistent with its linearization point. First, suppose the `embedded-scan` terminates by condition (1). As remarked above, whenever two reads of a register return the same result, that value must have been in the register for the entire interval between the two reads. Thus, if the `embedded-scan` returns $((i_1, v_1), \ldots, (i_r, v_r))$, then each $R[i_j]$ must have contained v_j at the time the scan is linearized (for $j \in \{1, \ldots, r\}$).

Now consider an `embedded-scan` E that terminates by condition (2). Let E' be the `embedded-scan` whose result is borrowed by E. In this case, the `update` that performs E' must have started after E did: this is because the process that performs E' did at least one other write during E before writing the output of E', and that earlier write must have been part of a different `update` operation. It follows by an induction argument that the linearization point of each `embedded-scan` that terminates by condition (2) is between the invocation and the termination of the `embedded-scan`.

Now, it remains to show that the last line of the `scan` routine is well-defined. Let S be an invocation of `scan`$(i_1, \ldots, i_r)$ by some process p that reaches the last line of the `scan` routine. We must show that, for all $j \in \{1, \ldots, r\}$, there is an i'_k in the result of S's `embedded-scan` E that is equal to i_j, meaning that the result of E contains enough information to produce the output of S. If E terminates by condition (1), then this is obvious: the components in the result of E are identical to the arguments of S. Otherwise, if E terminates by condition (2), the result of E was originally produced by some other `embedded-scan` E' that terminated by condition (1). As argued above, the update U that performed E' began after E. The `getSet` performed by U must therefore include process p in its output because p completed its `join` operation before calling E. So the arguments given to E' must include all of S's arguments and, thus, the view returned by E' and written by U will contain sufficient information to produce the output for S.

We now consider the step complexity of the update and scan operations. Let T_{join}, T_{leave} and T_{getSet} be the step complexities of the three active set operations. The number of processes returned by a `getSet` operation is always bounded by $\overline{C}_s$. Thus, in an execution where all partial scans access at most r_{max} components, the number of arguments that an `update` passes to its `embedded-scan` is at most $\overline{C}_s \cdot r_{max}$. An `embedded-scan` will satisfy termination condition (2) after performing at most $2\overline{C}_u + 1$ collects. The time for an `update` is thus $O(\overline{C}_u \cdot \overline{C}_s \cdot r_{max}) + T_{getSet}$. The time for a `scan` of r components is $O((\overline{C}_u + 1) \cdot r) + T_{join} + T_{leave}$. In the best known solution to the active set problem [12], all operations have step complexity $O(\dot{C}_s^2)$. Thus, we have the following theorem.

Theorem 1. *The algorithm in Figure 1 is a wait-free, linearizable implementation of a partial snapshot object from registers where processes perform* $O((\overline{C}_u + 1) \cdot r + \dot{C}_s^2)$ *steps per* `scan` *and* $O(\overline{C}_u \cdot \overline{C}_s \cdot r_{max} + \dot{C}_s^2)$ *steps per* `update`.

If the implementation is altered to use small registers, as mentioned above, the time complexity increases slightly. An `update` performs an additional $O(\overline{C}_s \cdot r_{max})$ steps to write its view, sorted by indices, so this just increases its time by a constant factor. A `scan` that satisfies exit condition (2) can use binary searches within a recorded view to read the r components it must return using an additional $O(r \log \overline{C}_s + r \log r_{max})$ steps.

4. USING STRONGER PRIMITIVES TO ACHIEVE LOCAL PARTIAL SCANS

We describe here an implementation of a partial snapshot object where `scan` operations are local, and `updates` are efficient in an amortized sense. We do this using compare&swap and fetch&increment objects in addition to registers.

A compare&swap object stores a value and provides an operation compare&swap(*old*, *new*), which changes the object's value to *new* if and only if it is currently equal to *old*. The operation returns the previous value stored in the object. A fetch&increment object stores an integer, and provides an operation that atomically increments the value and returns the new value. For convenience, we assume a fetch&increment object can also be read without changing its value.

We modify the snapshot algorithm in Figure 1 in two ways to make use of these stronger primitives. First, we create a new algorithm for the active set problem which accomplishes `joins` and `leaves` in a constant number of steps. Secondly, we change the write performed by an `update` to a compare&swap. This allows us to bound the number of collects done by a partial `scan` of r components in terms of r rather than the contention. Together, these changes yield a wait-free snapshot algorithm where a partial `scan` finishes in $O(r^2)$ steps. This increases the time required for `update` operations, but the amortized time for `updates` and `scans` is still reasonable: in particular, the amortized time depends only on r_{max} and the contention.

4.1 A New Active Set Algorithm

We design an active set algorithm where `joins` and `leaves` happen very quickly. Since this is done by pushing most of the real work into the `getSet` operations, the worst-case time complexity of `getSets` becomes unbounded. However, in an amortized sense, `getSets` are still efficient.

Our active set algorithm uis given in Figure 2. It uses an array of registers $I[1..]$, each element of which stores the id of one active process. The algorithm also uses one fetch&increment object H that stores the highest index in I that has been written to, and one compare&swap object C. To join the set, a process performs a fetch&increment on H to obtain an index of a free entry of I into which it can write its id. To leave the set, the process simply writes 0 into this entry of I. The compare&swap object C holds a list of intervals of array indices that are known to contain only 0's, which can be safely skipped by a process doing a `getSet` operation. A `getSet` operation will read through the entries of I up to the location indexed by H, skipping all entries that appear in an interval of C, using the values of C and H that are read at the beginning of the `getSet` operation. While reading I, any entries of I that have been vacated are added to a local list of intervals that can be safely skipped, and the process attempts to put this updated list into C using a compare&swap at the end of the `getSet` to ensure that subsequent `getSets` do not have to check those vacated entries of I again. While locally constructing the updated list, any consecutive intervals that have no gaps between them should be coalesced into a single interval in order to keep the length of the list as small as possible. To make the local operations on the list efficient, the intervals in the list should be kept in sorted order.

Correctness of the active set algorithm follows from one simple invariant: An index appears in an interval stored in C only after the corresponding entry of I is set to 0 (and that entry of I never changes thereafter). Thus the `getSet` operation finds the id of every process that has completed its `join` before the `getSet` begins. Furthermore, the `getSet` does not return the id of any process whose `leave` is completed before the `getSet` begins since the `leave` erases the process's id from the array I.

The `join` and `leave` operations take $O(1)$ steps. The number of steps that have to be taken by a `getSet` can be bounded only by the number of `joins` in the entire execution. For example, if k `joins` and `leaves` occur with no `getSet` operations, a subsequent `getSet` will have to read through all k entries of the I array. However, we shall show that the amortized complexity of all operations is bounded in terms of contention. When analyzing an active set algorithm, active processes are counted, along with processes performing operations, when measuring contention [3]. This measure of contention is appropriate for the active set problem because it is usually studied as a subroutine in the context of solving some larger problem, and active processes are those that are in the middle of performing some operation within the large problem; indeed, this is exactly what we do when we implement partial snapshot objects using the active set algorithm as a subroutine.

```
join
    l ← fetch&increment(H)
    I[l] ← id
end join

leave
    I[l] ← 0
end leave
getSet
    oldC ← C
    h ← H
    newC ← oldC
    result ← {}
    for j ← 1..h
        if j is not in one of the intervals in oldC
            entry ← I[j]
            if entry = 0 then add j to an interval in newC
            else result ← result ∪ {entry}
            end if
        end if
    end for
    compare&swap(oldC, newC) on object C
    return result
end getSet
```

Figure 2: A wait-free active set algorithm

Consider any execution. A `getSet` operation G reads at most $\overline{C}(G)$ non-zero values. If a `getSet` operation reads a 0 value in I, this read is charged to the `leave` operation that wrote the 0. Thus the amortized time per `getSet` is bounded by $\overline{C}$. The amortized time per `join` is just its actual cost, which is $O(1)$. We can also show the amortized cost per `leave` is $O(\dot{C})$. Let T_0 be the beginning of the execution. Let T_i be the moment that the ith successful compare&swap is performed on C. Notice that no `getSet` starts after T_i and ends before T_{i+1} (since then its compare&swap on C would be successful, contradicting the definition of the T_i's). Thus, every `getSet` that takes steps between T_i and T_{i+1} is running at time T_i or at time T_{i+1}. If a `leave` operation writes 0 in $I[l]$ between T_i and T_{i+1}, then, at all times beyond T_{i+2}, l is included in some interval stored in C. Thus, the `getSets` that can charge a read to this `leave` operation are all active either at T_i or T_{i+1} or T_{i+2}. Each such operation can charge at most one read to the `leave` operation. Thus, each `leave` operation is charged for at most $3\dot{C}$ reads and its overall amortized complexity is at most $3\dot{C} + 1$. This analysis is summarized in the following theorem.

Theorem 2. *The algorithm in Figure 2 is a wait-free solution to the active set problem in which* `joins` *and* `leaves` *take $O(1)$ steps. Moreover, the amortized time complexity of any execution is $O(1)$ per* `join` *operation, $O(\dot{C})$ per* `leave` *operation and* $O(\overline{C})$ *per* `getSet` *operation.*

We remark that the size of the compare&swap object in this algorithm is quite large: it could have to store up to $\Theta(\overline{C})$ intervals. If this is a concern, we can instead store the list of intervals in a set of $O(\overline{C})$ registers and store in C a pointer to this set of registers. This just adds $O(\overline{C})$ steps to the complexity of `getSet` operations but it ensures that all objects used are of a reasonable size.

Although our algorithm achieves our primary goal of having good time complexity, it does so using an unbounded number of registers. When a bound on the number of `joins` that can be performed in an execution is known *a priori*, the space can be bounded. Finding a way to recycle the registers in the case where no bound is known is left as an open question.

4.2 A Snapshot Algorithm with Local Scans

We now give our partial snapshot algorithm that uses the new active set algorithm. The snapshot algorithm uses an array $R[1..m]$ of compare&swap objects, and an array $S[1..n]$ of single-writer registers. Pseudocode is given in Figure 3. Besides using our new implementation of `join`, `leave` and `getSet`, there are only a few ways that this algorithm differs from the one in Figure 1: the termination condition (2) for `embedded-scans` is different, and `updates` perform a compare&swap in place of a write.

If an `update` U does a successful compare&swap on R, it is linearized when it performs that step. If U's compare&swap is unsuccessful, then there must have been some other successful compare&swap by another process updating the same component of the snapshot object between U's first read and U's compare&swap; U is linearized immediately before that successful compare&swap. All `embedded-scan` and `scan` operations are linearized as in the algorithm of Figure 1.

The proof of correctness follows the same line of reasoning as in Section 3. Here, we describe only the points at which the proof differs. If an `update` performs an unsuccessful compare&swap, it leaves no trace of its existence in shared memory. This means that no `scan` will ever see the value of this `update`. This is correct, since the `update` is linearized immediately before another `update` to the same component. The argument that the linearization point assigned to each `embedded-scan` E that terminates by condition (2) is within the interval of E is slightly different. If E has seen three different values in the same location, the second one was put into the object during the operation E. This means that the third value was put into the object by an `update` that read the object after it contained the second value, so it is safe for E to borrow the results of that `update`'s `embedded-scan` because that `embedded-scan` began after E did.

We now look at the time complexity of this implementation. The worst-case time for a `scan` of r components is $O(r^2)$, since condition (2) of its `embedded-scan` will be satisfied after $2r+1$ collects and the `join` and `leave` subroutines take $O(1)$ time. Since the `update` uses the `getSet` operation, there is no bound on the number of steps that an individual `update` may take in the worst case. However, we can again bound the amortized time per operation using the amortized analysis of the active set subroutines. Let r_{max} be the maximum number of components accessed by one partial `scan` in an execution. Since the number of components an `embedded-scan` must read is bounded by $\overline{C}_s \cdot r_{max}$, the time complexity of an `embedded-scan` is $O(\overline{C}_s^2 \cdot r_{max}^2)$. Using this, together with the amortized complexity of active set operations found in Section 4.1, we get an amortized complexity of $O(r^2 + \dot{C}_u)$ per `scan` operation and $O(\overline{C}_s^2 \cdot r_{max}^2 + \overline{C}_s) = O(\overline{C}_s^2 \cdot r_{max}^2)$ per `update` operation.

THEOREM 3. *The algorithm in Figure 3 is a wait-free, linearizable implementation of a partial snapshot object with worst-case time* $O(r^2)$ *for partial* `scans`. *Moreover, the amortized complexity of any execution is* $O(r^2 + \dot{C}_u)$ *per* `scan` *and* $O(\overline{C}_s^2 \cdot r_{max}^2)$ *per* `update`.

Using smaller objects, as described in the comments following Theorems 1 and 2 would add $O(\overline{C}_s \cdot r_{max})$ steps to each `update` and $O(r \log(\overline{C}_s \cdot r_{max}))$ steps to each `scan`.

5. RELATED WORK

There are implementations of ordinary adaptive snapshots from registers, whose step complexity depends only on the point contention [9, 12, 4]. As discussed in the introduction, these can be used to implement a wait-free single-writer partial snapshot object by simply ignoring irrelevant components, with $O(\dot{C}_s^2)$ step complexity per `scan` and `update` (using [12]). Although our implementation of partial snapshots from registers has higher step complexity, it provides a blueprint for the local algorithm using stronger primitives. In addition, our implementation supports multi-writer snapshot objects and stores smaller values.

While most algorithms for atomic snapshots use only read and write operations, a few papers studied implementations of atomic snapshots from primitives that are stronger than reads and writes. Attiya *et al.* [10] present an atomic snapshot implementation that uses $O(n)$ steps for a combined `update` and `scan` operation; the algorithm uses 2-processor test&set registers. Riany *et al.* [22] implement a single-writer atomic snapshot object with $O(1)$ time complexity for an `update` and $O(n)$ time complexity for a `scan`; their algorithm uses compare&swap, fetch&increment, and fetch&decrement primitives. Jayanti [21] shows that the same complexity bounds can also be achieved for the more general, multi-writer atomic snapshot object; this algorithm uses only compare&swaps. When the number of components m is smaller than the number of updaters n, Fatourou and Kallimanis [14] use compare&swaps to implement a multi-writer atomic snapshot object with $O(1)$ time complexity for an `update` and $O(m)$ time complexity for a `scan`.

These algorithms provide a complete view of all the components: none of them provides partial `scans` with lower time complexity, depending only on the number of components scanned.

Some adaptive implementations of the *collect* abstraction use strong synchronization primitives [2, 18], which can be used to obtain adaptive implementations of atomic snapshot objects (at least single-writer). Again, none of these implementations is local, in the sense that scanning a smaller subset of the components does not have a cost proportional to the total number of components. The collect algorithm of Herlihy *et al.* [18] is *dynamic* and can be translated into an active set algorithm, which bears some similarities to our active set algorithm. However, because we are less concerned about space complexity, we use an array rather than a linked list to make our `join` and `leave` run in constant

```
embedded-scan(i_1, ..., i_r)
    repeatedly read R[i_1], ..., R[i_r] until either
        (1) two sets of reads return the same vector, (x_1, ..., x_r);
            then return ((i_j, first field of x_j))_{1≤j≤r},
        or (2) three different values have been seen in some location;
            then let (v, view, c, id) be the third value seen in that location.
            return view
end embedded-scan

update(i, v)
    old ← R[i]
    scanners ← getSet
    (i_1, ..., i_r) ← ⋃_{p∈scanners} S[p]
    view ← embedded-scan(i_1, ..., i_r)
    compare&swap(old, (v, view, counter, id)) on object R[i]
    if the compare&swap was successful then counter ← counter + 1
end update

scan(i_1, ..., i_r)
    S[id] ← {i_1, ..., i_r}
    join
    ((i'_1, v_1), ..., (i'_k, v_k)) ← embedded-scan(i_1, ..., i_r)
    leave
    component j of the result vector is v_ℓ, where i'_ℓ = i_j
end scan
```

Figure 3: Partial snapshot algorithm with fast scans.

time, allowing us to obtain a local implementation of a partial `scan`. In addition, their algorithm is lock-free whereas ours is wait-free.

Jayanti [20] presented the *f-array*, an object with m components; a process can either update a component of the array or obtain the value of some function f applied to all the components of the array. He presents an implementation of f-array, in which an update operations requires $O(m)$ steps, while f operation on all the components is performed in $O(1)$ steps; this assumes an LL/SC object that can store any value of the function f. For certain aggregation functions f, the update operation can be performed in $O(\log n)$ steps. The multi-writer snapshot object is a simple special case of an f-array; the function f can also be specified so that an f-array provides an active set algorithm. However, in these cases, the object to which LL/SC operations are applied is large, since its size is proportional to the number of processes or the number of components of the snapshot object; moreover, the improvement in the scan operation is achieved by making the cost of an update proportional to the size of the f-array, regardless of the current contention and number of components scanned.

6. CONCLUDING REMARKS

We have introduced partial snapshots, a generalized version of snapshots, which we believe could be widely applicable. We have given an algorithm for implementing these partial snapshots in a local manner, using compare&swap as well as fetch&increment. Finding a local implementation of partial snapshots that uses only reads and writes, or even just compare&swap, or proving this is impossible, is the main technical open question we leave for further research.

Other ways in which our algorithms might be improved include adapting them to use smaller objects, bounding the timestamps they use, and possibly improving the complexity bounds to depend on point contention rather than interval contention. We were focusing on time complexity, without being overly concerned with space complexity. In particular, the number of registers used by our second algorithm is bounded only by the number of operations performed in an execution. It would be interesting to see whether the registers could be recycled to improve the space complexity.

Further research using different approaches to implementing partial snapshots may yield more efficient or more practical algorithms. Lower bounds on the complexity of local implementations would also be of great interest, particularly because they would have implications on the complexity of implementing transactional memory [19, 24]. Indeed, a partial `scan` can be viewed as a read-only transaction that declares the objects it wishes to access in advance. Any lower bound on the implementation of a partial `scan` would yield a lower bound on the implementation of such transactions. In fact, it would be interesting to see how efficient implementations of partial snapshots can help devise efficient implementations of general transaction memory systems, along the lines of [13, 23].

We also hope this work will help revive interest in the active set problem, which elegantly captures a fundamental problem in distributed computing, and was very useful in understanding how to achieve local implementations of partial snapshots.

Acknowledgements

The authors thank Panagiota Fatourou and the anonymous reviewers for their helpful comments. This research was supported by the Natural Sciences and Engineering Research Council of Canada, the Israel Science Foundation (grant number 953/06), and Intel Corporation.

7. REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4), pages 873–890, September 1993.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[3] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Proc. 40th IEEE Symposium on Foundations of Computer Science*, pages 262–272, 1999.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symposium on Principles of Distributed Computing*, pages 71–80, 2000.

[5] James H. Anderson. Composite registers. *Distributed Computing*, 6(3), pages 141–154, April 1993.

[6] James Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3), pages 414–431, May 1993.

[7] James Aspnes and Maurice Herlihy. Fast, randomized consensus using shared memory. *Journal of Algorithms*, 11(2), pages 441–461, September 1990.

[8] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

[9] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4), pages 444–468, July 2003.

[10] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3), pages 121–132, March 1995.

[11] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4), pages 725–763, July 1994.

[12] Hagit Attiya and Idan Zach. Incremental calculation for fully adaptive algorithms. Brief announcement in DISC04, available in `www.cs.technion.ac.il/~hagit/publications/AZ03.pdf`.

[13] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. Distributed Computing, 20th International Symposium*, volume 4167 of *LNCS*, pages 194–208, 2006.

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using CAS. In *Proc. 26th ACM Symposium on Principles of Distributed Computing*, pages 33–42, 2007.

[15] Faith Ellen Fich. How hard is it to take a snapshot? In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *LNCS*, pages 28–37, 2005.

[16] Rainer Gawlick, Nancy Lynch, and Nir Shavit. Concurrent timestamping made simple. In *Proc. of the Israel Symp. on the Theory of Computing and Systems*, volume 601 of *LNCS*, pages 171–183, 1992.

[17] Maurice Herlihy. Randomized wait-free concurrent objects. In *Proc. of the 10th ACM Symp. on Principles of Distributed Computing*, pages 11–21, 1991.

[18] Maurice Herlihy, Victor Luchangco, and Mark Moir. Space- and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78, pages 260–280, April 2003.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[20] Prasad Jayanti. f-arrays: implementation and applications. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 270–279, 2002.

[21] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *Proc. 37th ACM Symposium on Theory of Computing*, pages 723–732, 2005.

[22] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2), pages 163–201, October 2001.

[23] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. Distributed Computing, 20th International Symposium*, volume 4167 of *LNCS*, pages 284–298, 2006.

[24] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2), pages 99–116, 1997.