Towards A Practical Snapshot Algorithm

Yaron Riany, Nir Shavit, Dan Touitou
Tel Aviv University

***Abstract*— An *atomic snapshot memory* is an implementation of a multiple location shared memory that can be atomically read in its entirety without having to prevent concurrent writing. The design of wait-free implementations of *atomic snapshot memories* has been the subject of extensive theoretical research in recent years. This paper introduces the *coordinated-collect* algorithm, a novel wait-free atomic snapshot construction which we believe is a first step in taking snapshots from theory to practice. Unlike former algorithms, it uses currently available multiprocessor synchronization operations to provide an algorithm that has only $O(1)$ update complexity and $O(n)$ scan complexity, with very small constants. Empirical evidence collected on a simulated distributed shared-memory multiprocessor shows that *coordinated-collect* outperforms all known wait-free, lock-free, and locking algorithms in terms of overall throughput and latency.**

I. Introduction

An *atomic snapshot memory* [1], [3] is an implementation of a multiple location shared memory that can be atomically read in its entirety. The ability to collect such an instantaneous view is a powerful tool for designing concurrent data structures, as it greatly reduces the need to argue about inconsistent views of memory. Snapshots are useful for applications such as checkpointing, generating concurrent backups, debugging of multiprocessor programs, and even in a multiprocessor server of a radar tracking system, where multiple sensors generate updates concurrently with multiple requests for consistent system states. The design of wait-free implementations of an *atomic snapshot memory* has been the subject of extensive research in recent years [1], [3], [5], [7], [9], [12], [15], [23].

An atomic snapshot memory is an abstract data type equivalent to a memory partitioned into n segments, one for each processor. There are two types of operations on the object, a *scan* and an *update*. In an update operation, a processor writes the contents of its associated segment, while in a scan, each obtains an instantaneous "global picture" of all n segments. Snapshots should be fault-tolerant and non-interfering, that is, applications (for example, programs being checkpointed) on the system should have minimal disruption or loss of performance as a result of ongoing snapshots, and in the extreme should continue to run even in the face of severe timing anomalies. Fault-tolerance and non-interference are the major advantages of wait-free methods over standard lock-based implementations.

This paper takes a practical look at the question of providing wait-free implementations of atomic snapshots on multiprocessor architectures. A snapshot implementation that is to be practical should have the following properties:

- The complexity of performing an *update* operation should be within a small constant of that of a simple "write" to memory, since the typical user will not want to sacrifice the speed of updating memory to support efficient snapshots.
- Register sizes and hardware synchronization primitives should conform with ones available on multiprocessor architectures.
- Memory contention should be minimized by distributing and load-balancing work, otherwise good asymptotic complexity will not result in good performance.

A. Atomic Snapshots

The main contribution of this paper is in introducing the *coordinated-collect* algorithm, a novel atomic snapshot construction which we believe is a first step in taking snapshots from theory to practice. It uses *Load linked/Store conditional* and *Fetch&Increment* operations [13], [20], [25] to provide a multi-scanner algorithm[1] that uses real-world registers, each containing at most $O(1)$ values, having only $O(1)$ update complexity (in fact, at most *four* operations), $O(n)$ scan complexity, and $O(n^2)$ space complexity.

Though one might think that the use of strong primitives like *Load linked/Store conditional* would allow to readily modify the elegant snapshot algorithms in the literature [1], [3], [5], [7], [9], [12], [15] to achieve similar complexity, it turns out that this is not the case (See table in Figure 1). These multi-scanner snapshot protocols have an algorithmic structure in which each updater and/or scanner collects a view of memory in its register, and then have processors try to agree which of the views to return. This leads to a situation where, even with the added power of a *Load linked/Store conditional* operation to speed up the view-agreement process, the complexity of an update remains an unacceptable $\Omega(n)$, and the registers used in the algorithms are required to hold $\Omega(n)$ values.

Our presentation begins with the introduction of a new single scanner protocol – a greatly simplified version of the innovative single scanner protocol of Kirousis, Spirakis, and Tsigas [23]. [2] We build the *coordinated-collect* multi-scanner algorithm based upon our single scanner protocol. The algorithm has updaters perform the same $O(1)$ sequence of operations as in our single scanner algorithm, but uses a novel collection methodology to allow multiple scanners to return coherent scans of memory. Instead of deciding on one of many collected views as in former algorithms, *coordinated-collect* has all the active scanners

Contact author e-mail: shanir@math.tau.ac.il
This work was supported by a Digitial Equipment Corporation ERP Equipment Grant

[1] A multi-scanner algorithm is one in which concurrent scan operations by different processors are allowed.

[2] The version we present is unbounded but can be easily made bounded using a sequential time-stamp system [21].

0-8186-6915-2/95 $04.00 © 1995 IEEE

distribute the work and 'help' each other to collect values from the n registers into a pre-agreed shared view area. This allows us to achieve an $O(n)$ scan complexity without increasing the update complexity. The helping process is tailored to maintain low contention by load-balancing processors over the shared view locations. We effectively reduce the problem of designing a practical snapshot algorithm to that of maintaining a re-usable pool of view-areas.

To solve the latter task, we introduce a novel ***pool*** abstract data type to trace and recycle outdated shared-view areas in the face of multiple concurrent view requests by scanners. We provide two pool implementations, of $O(n^2)$ and $O(n^3)$ space, and compare their performance. The overall efficiency of our implementations is in minimizing concurrent access to shared pointers so as to eliminate possible memory "hot-spots" [19] (which would have left us with good asymptotic behavior but bad performance).

B. A Comparison of Atomic Snapshot Algorithms

The second contribution of our paper is a comparison of the performance of several single and multi-scanner algorithm snapshot techniques, including our own, on a simulated distributed shared-memory multiprocessor using the well accepted Proteus Parallel Hardware Simulator [10], [11]. Our choice of algorithms was driven not only by their asymptotic complexity, but also by the feasibility of actually implementing them on multiprocessor machines.

The first two compared methods are an algorithm that blocks updates during a scan and a lock-free algorithm that never blocks updates but does not guarantee scan termination in the face of repeated updating. Of the known wait-free methods, we chose to implement the unbounded-register versions of the algorithms of [1] and [9], and the consensus based algorithm of [12]. The first two algorithms use n-valued read/write registers to have processors agree among collected views, and the last uses n-valued registers and an agreement mechanism which we implemented using the powerful *Load linked/Store conditional* operation. We did not implement the *test&set* based algorithms of [7] which achieve agreement among views using an unbounded number of *test&set* registers. Transforming them into bounded algorithms would introduce a substantial overhead in space and in memory contention, making them inferior to the *Load linked/Store conditional* based agreement scheme which we tested. Given that the above algorithms assume the availability of atomic n-value registers, we tested them both under the (unrealistic) assumption that such operations are available in hardware, and under the (more realistic) one that each n-valued read operation takes at least n local operations. We found that their performance was only slightly improved by assuming n-valued registers were available in hardware.

We found that our single-scanner and multi-scanner *coordinated-collect* algorithms outperform all known algorithms both in throughput and latency. Surprisingly, their update throughput is as good as that of the lock free method which lets updates succeed at the price of very low scan throughput. [3] The scan throughput of our algorithms remains consistently high as the number of processors increases, even though the size of the collected views grows linearly. However, it has an associated overhead and generates a certain level of contention which prevent it from reaching the throughput of the blocking algorithm (which blocks all updates during a scan).

In summary, we believe our algorithm is an example of using current multiprocessor synchronization operations to develop snapshot algorithms that are more "realistic" in terms of register size and the complexity of update operations. Our hope is that it will lead to the development of a wait-free algorithm that has superior latency and throughput for both scans and updates.

Section V presents benchmark results of some of the implementable snapshot algorithms and compares them to our algorithms.

Our computation model follows [8], [9], [16]. An *atomic snapshot memory* is an object partitioned into n segments $Mem[1], .., Mem[n]$ each of type *Data*. There are two types of allowable actions for any process i: $Update_i(r)$, and $Scan_i(r_1, \ldots, r_n)$ for any $i \in \{1..n\}$. $Update_i(r)$ changes the value of $Mem[i]$ to r and $Scan_i(view[1..n])$ returns a *view* of memory, a collection view$[1..n] = Mem[1..n]$ of values. Scan and update operations are atomic, that is, behave as if they were executed instantaneously within their associated execution interval. This is captured by requiring that the implementation of any atomic snapshot memory be linearizable [16].

Formally, assume that each processor is sequential, that is, does not start a scan or update operation until it has finished the previous one. We define a partial *real-time* order on scan and update operations and denote it as "$\longrightarrow$" [24], where $A \rightarrow B$ means that the execution of operation A was completed before that of B was started. Operations are concurrent if neither $A \rightarrow B$ nor $B \rightarrow A$.

An atomic snapshot memory implementation is correct if for every execution there is a sequence containing all scan and update events, each completely preceding the other, such that:

1. It extends the real-time order of operations as defined by $\rightarrow$, and
2. Maintains the sequential semantics of scan and update operations; that is, if *view* is returned by some *scan* operation, then each $view[i]$ is the value written by the last $update_i$ operation which precedes the scan in the sequence.

We will be interested in getting an implementation that is *wait-free*, that is, the execution of any implemented scan or update operation completes within a bounded number of machine operations [17].

[3] In the lock free algorithm, the scanner repeatedly collects the contents of the registers. If it has read twice the content of the registers and no register has been changed, it returns the collected values as a result.

Snapshot Algorithm		Primitive Used	Update Complexity	Scan Complexity	Register Size	Space Complexity
Lock free		r/w register	$O(1)$	∞	$O(1)$	$O(n)$
Block update		r/w register	∞	$O(n)$	$O(1)$	$O(n)$
Anderson [3]		r/w register	$O(2^n)$	$O(2^n)$	$O(1)$	$O(n^3 logn)$
Aspnes,Herlihy [5]		r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$
Afek et al. [1]	Unbounded	r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
	Bounded	r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$
Kirousis et al. [23]	One scanner	r/w register	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Dwork et al. [15]	Weak snapshot	r/w register	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$
Chandra Dwork [12]		LL/SC	$O(n)$	$O(n)$	$O(n)$	∞
Attiya, Herlihy Rachman [7]	Version 1	T&S	$O(nlog^2 n)$	$O(nlog^2 n)$	$O(n)$	∞
	Version 2	dyn. T&S	$O(n)$	$O(n)$	$O(n)$	∞
Attiya,Rachman [9]	Unbounded	r/w register	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	∞
Coordinated Collect	Version1	LL/SC,F&I	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
	Version2	LL/SC,F&I	$O(1)$	$O(n)$	$O(1)$	$O(n^3)$

Fig. 1. A Comparison of Atomic Snapshot Algorithms

II. The Single-Collect algorithm

We begin by presenting a greatly simplified variant of the Kirousis, Spirakis and Tsigas one scanner snapshot algorithm [23]. This algorithm uses only atomic read/write register operations, and achieves optimal time and space complexity: $O(n)$ for a scan operation and $O(1)$ for an update operation. The algorithm uses a sequential time-stamp scheme (a register of 64 bits will suffice, though one can replace it by a bounded sequential time-stamp system [14], [21]). The code for *scan* and *update* operations appears in Figure 2. [4]

```
Scan()
{
        curr_seq = curr_seq+1;
        for j=1 to n {
            high_r=r[j].high;
            if (high_r.seq < curr_seq)
                view[j]=high_r;
            else
                view[j]=r[j].low;
        }
        return(view);
}

Update(val)
{
        seq=curr_seq;
        high_r=r[i].high;

        if (seq==high_r.seq)
                r[i].high=[val,seq];
        else {
                r[i].low=high_r;
                r[i].high=[val,seq];
        }
}
```

Fig. 2. The Single-Collect algorithm - code for P_i

The algorithm uses a shared array `r[1..n]` of records, each record having two components *high* and *low*, and a shared variable `curr_seq` which holds a current time-stamp (sequence number) of the latest scan operation, incremented at the start of each scan operation. Each update operation first reads this number, and the idea is to make sure that the scanner returns only values that were written by update operations that did not read its time-stamp. This means that these updates started before the scan, and so all their associated values *could have existed* in memory at the point in time when the increment of the time-stamp was performed. To guarantee that such a value is found for each updater, we must keep in memory its latest update value having a timestamp preceding that of the current scan. For this purpose we use the low field of each update location.

[4] The code is presented in C-style programming.

III. The Coordinated-Collect algorithm

The key idea behind our new multi-scanner algorithm is to have updaters perform the same sequence of operations as in the *single-collect* algorithm (Figure 2), and have concurrent scanners *work together* in collecting a view. The difficulty is that this coordinated collecting must be performed efficiently and in a wait-free manner on real architectures using only bounded space. We do so using a combination of *Load-linked/Store-Conditional*, *Fetch&Increment*, and *Fetch&Decrement* operations.

The crux of our algorithm lies in replacing the commonly used idea of collecting separate views and agreeing, using an n-process consensus protocol or primitive, on one of them, with the idea of agreeing on a memory location, and coordinating the collecting of a view into it without using n-consensus. The high level description of the algorithm appears in Figure 3.

On a high level, a process wishing to perform a scan proposes its `own_view` as a space in which all scanners will perform a coordinated-collect. Each scanner first invokes a `get_view` operation, which returns as result a "free" view, that is, a view which no other scanner needed its use. An initialization operation is performed on this returned view, and it is becomes the scanner's suggested view. We use a *pool* of view spaces, from which groups of concurrent scanners pick "free" view spaces. The *pool* behaves as if it contains an unbounded number of free views, though in reality we will use a view recycling mechanism.

Processes agree which view to collectively work on using a "helping" methodology in the style of Herlihy's announce

array [17]: every process proposes its **own_view** and posts it in its announce entry. The agreed view, on which the "filling up" takes place, is determined by a shared pointer, which is advanced in an agreement upon completion of filling up a view. At any point in time, only one view is being filled up, that is, filling up operations are totally ordered. In order to help filling up a view, each scanner has to successfully **acquire** it, and after filling it up, the scanner **releases** it. A scanner helps filling up at most two views, since the second filled up view can be returned as a result.

The **acquire** of a view fails only when all scanners which participated in its filling up, have already **released** it, and the view is being *recycled* by a scanner for a new scan operation. Any failed **acquire** operation, is followed by an advance of the shared counter. Therefore, after at most n failed **acquire** operations, the scanner is assured that its **own_view** has been filled up.

The filling up of a view is done in the same style of the single-collect scan operation. The main difference is that there might be many processors concurrently participating in the filling process. In order to achieve better performance, we distribute the the work onto many locations in a coordinated way using a shared counter from which processes get the next address to update. [5]

Theorem III.1: Given a *wait-free* implementation of a *pool* object with operations *get_view* ,*init* ,*acquire* and *release* having time complexity $O(g(n))$, $O(i(n))$, $O(a(n))$ and $O(r(n))$ respectively, and with $O(s(n))$ space complexity in terms of $O(1)$ value registers then:

1. The *coordinated-collect* multi scanner algorithm is correct and *wait-free* .
2. The *scan* complexity is $O(g(n) + i(n) + n * a(n) + r(n) + n)$ steps, the update complexity is $O(1)$ steps and the space complexity is $O(s(n))$.

Using the pool constructions we provide, we conclude:

Corollary III.2: The *coordinated-collect* multi scanner algorithm achieves $O(n)$ *scan* complexity and $O(1)$ *update* complexity.

IV. Pool Objects

We define a *pool* object as an abstract datd type whose elements are objects of type *view* each consisting n register fields. The *pool* object allows the following operations on its *view* elements:

get_view returns a view V from the *pool* .

init(V) initializes *view* V of the *pool* .

acquire(V) marks view V as acquired and returns boolean (True/False).

release(V) marks view V as released.

is_empty(V,j) checks the j-th register field of *view* V of the pool, and returns boolean (True/False).

We denote a successful *acquire* operation (i.e. returns True), as *successful_acquire*.

Definition IV.1: A correct scanner is a processor P_i which satisfies the following conditions:

[5]We could have replaced that single location shared counter by a faster and lower contention counting network or diffracting tree, but this goes beyond the scope of this paper.

```
Scan()
{
    own_view = get_view();
    init(own_view);
    assign own_view to my announce entry;
    while (own_view is not full) {
        curr_view = view pointed by
          the current entry in the announce array;
        if (curr_view is full)
            advance on the announce array;
        else
            help_fill(curr_view);
    }
    release(own_view);
    return(own_view);
}

help_fill(curr_view)
{
    if ( successful_acquire(curr_view) ) {
        first = successful assignment of
          increased curr_seq in curr_view;
        fill curr_view's registers in a
          "single-collect" style;
        mark curr_view as full;
        advance on the announce array;
        if ((first) or (help_fill once before) or
            (own_view is full)) {
                if (own_view is not curr_view) then
                    release(own_view);
                release(curr_view);
                return(curr_view);
        }
        release(curr_view);
    }
}

}
```

Fig. 3. High-level code of Coordinated-Collect algorithm

1. After P_i performs an *init(V)* or a *successful_acquire(V)* it performs no other *init(V)* or *acquire(V)* until it performs *release(V)* .
2. P_i never performs a *release(V)* without a preceding *init(V)* or *successful_acquire(V)* .

Definition IV.2: An pool implementation is correct if for every execution there is a sequence containing the above defined *pool* operations, each completely preceding the other, such that:

1. It extends the real-time order of operations as defined by $\rightarrow$.
2. When accessed only by correct scanners, the following invariants are kept:
    (a) The *get_view* operation always returns a view V, for which the number of *release(V)* operations ever performed, is equal to the sum of the *init(V)* and *successful_acquire(V)* operations ever performed.
    (b) If some scanner P_i performs *get_view* operation and receives a returned view V, then until it performs an *init(V)* , there can be no *successful_acquire(V)* occurrences.
    (c) The *is_empty(V,j)* operation returns True iff it occurs after the completion of an *init(V)* , and before the first write to the j-th register of V.

(d) Upon completion of an *init(V)* and before any other operation, the *view* V in its initial state.

We implemented two versions of the *pool* object. Both versions yield the same asymptotic time complexity for the multi scanner algorithm: $O(n)$ for *scan* and $O(1)$ for *update*.

The first version , which is much simpler to prove and implement, has minimized time complexity in practice, having smaller constants, but has space complexity of $O(n^3)$, i.e. array of n^2 *view* objects. The main idea behind the construction is that each scanner has its own separate part of the *pool* object. Each time a scanner performs a **get_view**, it searches for a view in its own pool containing n *view* objects. The **acquire** and **release** operations use *Fetch&Increment* and *Fetch&Decrement* operations appropriately, in order to maintain for each *view* object, the current number of scanners which have performed a successful **acquire** on it. The **init** operation just initializes the *view* object.

In the second version, which has better space complexity but inferior performance, the *pool* object contains only $2n$ *view* objects, due to a major change in the **get_view** operation implementation. All *view* objects are shared by all the scanners, and within the **get_view** operation each scanner coordinates its search using the "helping" methodology.

V. Performance Evaluation of Snapshot Algorithms

We compared a collection of snapshot algorithms on a 64 processor simulated Alewife cache-coherent distributed-memory machine [2] using the Proteus simulator developed by Brewer, Dellarocas, Colbrook and Weihl [10]. Each processor had a cache with 2048 lines of 8 bytes and a memory access cost of 4 cycles. The cost of switching or wiring in the Alewife architecture was 1 cycle/packet. The current version of Proteus does not support *Load-linked/Store-Conditional* instructions. Instead we used a slightly modified version that supports a 64-bit *Compare-and-Swap* operation where 32 bits serve as a time stamp. Naturally this operation is less efficient than the theoretical *Load-linked/Store-Conditional* [18] (which we could have built directly into Proteus), since a failing *Compare-and-Swap* will cost a memory access while a failing *Store-Conditional* wont. However, we believe the 64-bit *Compare-and-Swap* is closer to the real world then the theoretical *Load-linked/Store-Conditional* since existing implementations of *Load-linked/Store-Conditional* as on Alpha [13] or PowerPC [20] do not allow access to the shared memory between the *Load-Linked* and the *Store-Conditional* operations. On existing machines, the 64 bits compare-and-swap may be implemented by using a 64 bits *Load-linked/Store-Conditional* as on the Alpha.

We measured for each scan and update implementation:

Throughput The total number of completed operations by all the processors in the system running for 10^6 cycles.

Latency The average amount of time between the start and the end of an operation for all the processors in the system.

We evaluated the algorithms by having each scanner/updater processor execute scan/update operations repeatedly as in Figure 4. Between each two operations, a processor waits for an amount of time chosen uniformly at random in the interval 0 to **scan_wait** for a scanner, and 0 to **update_wait** for an updater. We used the following benchmarks:

Checkpoint The system has only one processor which executes scan operations (scanner) and the other processors execute update operations (updaters). This benchmark models the behavior of a "checkpoint" mechanism for collecting consistent backups of a multiprocessor system or for concurrent debugging. The algorithms were tested with **MAX_TIME** equal to 10^6 cycles, and **scan_wait** and **update_wait** equal to 10^3 cycles.

Concurrent data structure The system has half of the processors execute scans and the other execute updates. This benchmark models the the use of snapshots for concurrent-data-structure design, where multiple processors update or request an atomic view of the state of the shared object. The algorithms were tested with **MAX_TIME** equal to 10^6 cycles, and **scan_wait** and **update_wait** equal to 10^3 cycles. We added a second set of tests with **update_wait** equal to 100 cycles.

```
SCANNER:
        while (current_time < MAX_TIME) {
                repeat random(scan_wait) times
                        /* do nothing */ ;
                scan();
        }
UPDATER:
        while (current_time < MAX_TIME) {
                repeat random(update_wait) times
                        /* do nothing */ ;
                update(val);
                val.seq = val.seq + 1;
        }
```

Fig. 4. Benchmarks

A. The Algorithms

In the concurrent data structure benchmark, we tested two versions of the *coordinated-collect* multi scanner algorithm, using the the two *pool* object implementations. In the checkpoint benchmark, we tested the *single-collect* algorithm, as described in section II.

We compared our algorithms, with the following previously known snapshot algorithms:

A+ The unbounded version of the single-writer algorithm in [1], which has $O(n^2)$ scan and update complexity, and uses $O(n)$ values registers. Each register, in this version, has a *data,seq* and n-value *view* component. Since we could not implement $O(n)$ value registers, and by observing that this algorithm control flow is not dependent on the contents of the n-value *view*

component of each register, we excluded that component.

AR The unbounded version of the algorithm in [9], which has $O(nlogn)$ scan and update complexity and uses $O(n^2)$ values registers, was simulated. Each register, which participates in the *lattice agreement* procedure, contains n vectors, where each vector represents a view of $O(n)$ values. The implementation of the algorithm had to deal with the manipulation of the contents of those registers, i.e. *unionizing* the views vectors. However, since each view vector contents is not referenced, we implemented a bit value as a reference of this view vector, and thus enabling the unionizing of the views vectors.

CD The unbounded algorithm in [12], which has a scan complexity of $O(n)$ and update complexity of $O(n + C(n))$, where $C(n)$ is the consensus complexity. We used the strong *Load linked/Store conditional* primitive, to implement the consensus, thus theoretically achieving $O(n)$ scan and update complexity. Yet, this algorithm uses $O(n)$ value registers, and also, atomic multi-write operation of these registers. The algorithm uses for each scanner, two structures a new one and an old one, where each has an $O(n)$ value *view* register and an appropriate $O(1)$ value *time-stamp*. However, the control flow of the algorithm is dependent only on the values of the new and old *time-stamp* components. We included only these components in our implementation's registers, without making the algorithm pay for the added $O(n)$ values that must be stored in other registers.

Lock-free The simple algorithm in which a scanner repeatedly tries to perform a successful *double collect*, during which no change to memory occured, and an updater which writes to its register in a straightforward manner.

Block-update The scanner uses a multi-valued semaphore to "block" any updaters from performing a write to any of the registers, while it collects their values. The updater uses a random backoff method, while "waiting" for the semaphore to be cleared.

The first three algorithms (A+,AR,CD) has unbounded space complexity (in the strong sense of unbounded number of new locations), and therefore its time complexity is much less than any of its appropriate bounded implementation. Thus, our test results give an advantage for these algorithms with respect to our practical bounded algorithm implementation.

In our benchmarks we make the realistic assumption that the implementation of registers containing $\Omega(n)$ values requires at least n local steps for each read operation (we avoid making this assumption on write operations). However, we performed tests under the unrealistic assumption of availability of atomic $O(n)$-value registers and hardware operations, with no significant changes in our conclusions (appears in the full paper).

B. Checkpoint Benchmark Results

The checkpoint benchmark results, as can be seen in Figure 5, show that the *block-update* and the *lock-free* algorithms are at the extreme ends with respect to their scan and update throughput [6].

The *block-update* has the highest scan throughput since its scan operations are performed without any 'interference' from the updaters (the interference is in terms of interconnect contention and cache misses). However, it has very low update throughput, since the updates can be executed only between scan operations. Nevertheless, there is a performance increase due to having more concurrent update attempts.

The *lock-free* algorithm presents very poor scan throughput because of repeated double collect failures that increase with the number of updaters. Nevertheless, its update throughput scales linearly with the number of updaters. This is clearly due to the small number of operations executed to complete an update.

The A+ algorithm has similar degradation in its scan and update throughput. This is due to failures of its double collects which increase with the number of updaters, therefore, its scan latency increases, and that holds true for its update which is essentially a scan. The AR algorithm, because of the hidden constants involved in local work, is much worst than the A+ algorithm and does not manage to complete a single scan when n is larger than 20, and for the same reasons mentioned for the A+ algorithm, its update throughput degrades rapidly. The CD algorithm has low update throughput which degrades moderately as the number of updaters grow, due to the high contention and the intense local work executed in each update operation. Its scan throughput has good throughput for small number of updaters which degrades rapidly for larger numbers, due to an appropriate increase in the updaters 'interference'.

The *single-collect* algorithm update throughput is nearly the same as the *lock-free*'s due to the small number (four) of update operations, and also has high scan throughput, which is close to the non-interference scan of the *block-update*'s algorithm, since its scan collects the updaters values in a straightforward manner.

C. Concurrent Data Structure Benchmark Results

The results of the Concurrent data structure benchmark, appear in Figure 6, which includes both throughput and latency results for the first set of tests. For most algorithms these results have a lot in common with those of the checkpoint benchmark. We will therefore concentrate on the major differences.

The *block-update* algorithm never seems to succeed in completing an update for any number of processors due to the increased number of scanners which disable the updaters progress. The scan throughput of the *lock-free* algorithm degrades rapidly due to the increased failure of the double collects as the number of updaters increases. The

[6] The scan and update latency results are not presented since they are appropriately inversed to the scan and update throughput.

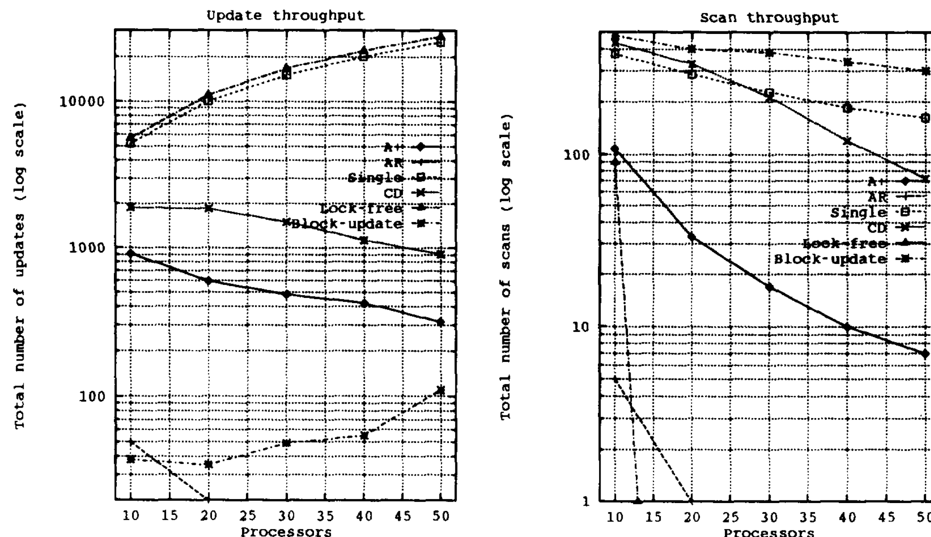

Fig. 5. One scanner throughput (Log scale)

CD algorithm starts with a very good scan throughput and some scaling but as the updaters 'interference' grows it degrades substantially.

The *coordinated-collect* algorithm maintains consistent high scan throughput and linear scaling of scan latency. Unlike in the case of the checkpoint benchmark, there is a better throughput since a single view can be the returned result of several scanners. One must remember that the size of the view that needs to be collected increases linearly with the number of processes. Furthermore, the method we used for collecting the view is a little more subtle than the straightforward way of iteration on the view entries, since the latter would cause major contention of the interconnect and would degrade the scan throughput.

The performance results of the $O(n^3)$ version of the *coordinated-collect* algorithm, shows an improvement over the $O(n^2)$, in terms of scan latency and throughput. This strengthens our intuition about the need to simplify the **get_view** procedure, in order to achieve better results. We measured the relative latency of the **get_view** procedure, which resulted in 80-90% of the scan latency, for the original *coordinated-collect* algorithm, and 60-70% for the $O(n^3)$ version. These results show that the space limit and the need to recycle views is a major factor in the overall performance. The coordination costs in the views filling up is minimized in our implementation, therefore, further space recycling simplification can still obtain performance enhancement.

In the second set we repeated our experiments with **update_wait** equal to 100 cycles, in order to simulate a "heavy load" of updaters, noting no significant changes.

VI. Conclusions

Though the asymptotic complexity of our algorithms is optimal, there are various practical directions in which their performance can be enhanced. First and foremost would be a more efficient implementation of the pool object. Other enhancements would involve eliminating some of the constant overheads, and make the algorithm complexity more closely dependent on the actual number of scanners and updaters accessing it at a given time. Finally, the current trend towards running multiprocessors applications in message passing architectures (farms of workstations) raises the interesting question of an efficient wait-free message passing implementation of an atomic snapshot object.

VII. Acknowledgments

We wish to thank the anonymous PODC and ISTCS referees for their helpful comments.

References

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

[2] Agrawal, A. et al. The MIT Alewife machine: A large scale distributed memory multiprocessor. *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] Anderson, J. H. Composite registers. *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1990) 15-29.

[4] Anderson, J. H., Singh, A. k., and Gouda, M. G. The elusive atomic register revisited. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing,* (Aug. 1987) 206-221.

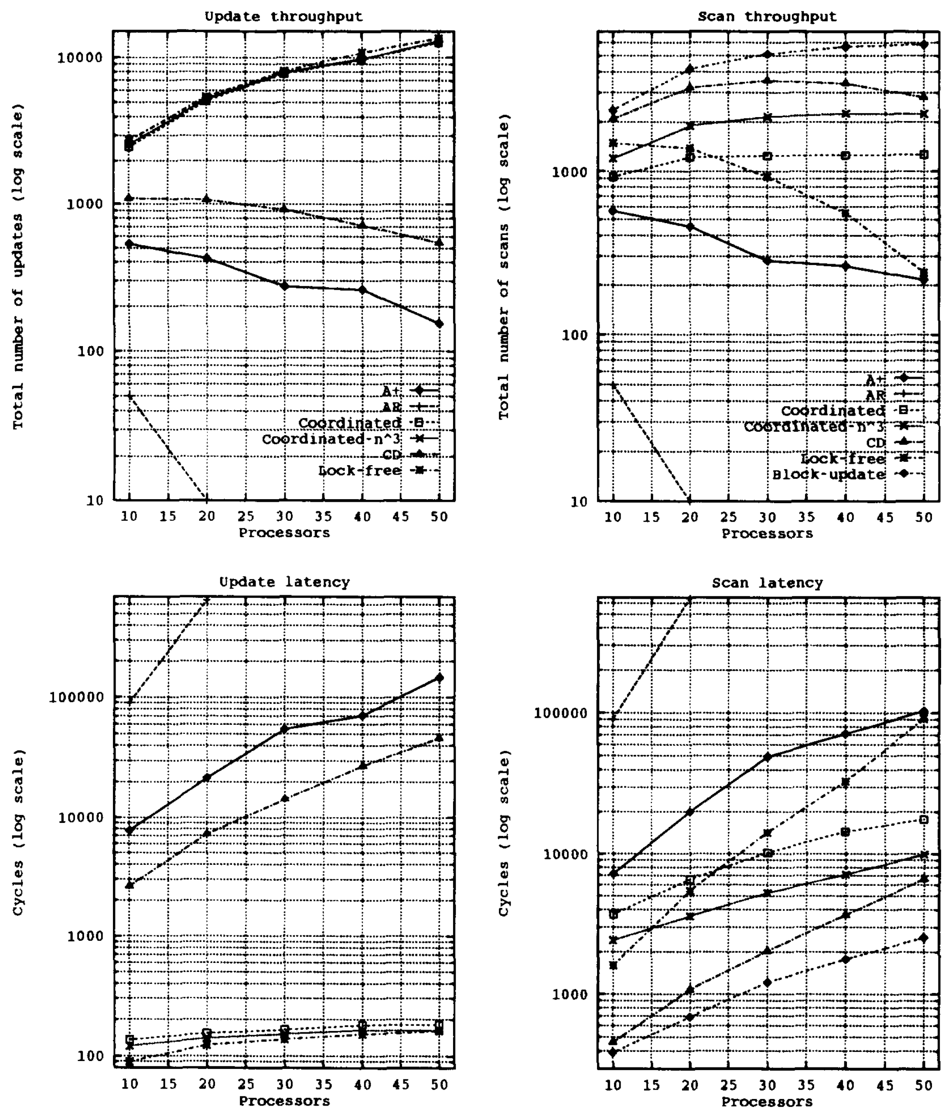

Fig. 6. Multi-scanner throughput and latency results (Log scale)

[5] Aspnes, J., and Herlihy, M. Wait-free data structures in the asynchronous PRAM model. *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures* (July 1990) 340–349.

[6] Attiya, H., Dolev, D., and Shavit, N. Bounded polynomial randomized consensus. *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, (Aug. 1989) 281–293.

[7] Attiya, H., Herlihy, M., and Rachman, O. Efficient atomic snapshots using lattice agreement. *Proceedings of the 6th International Workshop on Distributed Algorithms.* Haifa,Israel,November 1992(Segall A. and Zaks S. eds.) 35–53.

[8] H. Attiya, N. Lynch and N. Shavit, "Are Wait-Free Algorithms Fast?" *Journal of the ACM*, Vol. 41, No. 4 (July 1994), 725–763.

[9] Attiya, H., and Rachman, O. Atomic snapshots in O(nlogn) operations. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, (Aug. 1993) 29–40.

[10] Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. Proteus: A High performance parallel architecture simulator. MIT Technical report /MIT/LCS/TR-561, September 1992.

[11] Brewer, E. A. and Dellarocas, C. N. Proteus User Documentation, Version 4.0, march 1992.

[12] Chandra T. D. and Dwork, C. Using Consensus to solve Atomic Snapshots. Manuscript, 1993.

[13] Digital Equipment Corporation. ALPHA system reference manual.

[14] Dolev, D., and Shavit, N. Bounded concurrent time-stamp systems are constructible! *Proceedings of the 21st Annual ACM Symposium on the Theory of Computing*, (May 1989) 454–465.

[15] Dwork, C., Herlihy, M., Plotkin, S. A., and Waarts, O. Time lapse snapshots. *Proceedings of the Israel Symposium*

on the Theory of Computing and Systems. Haifa, Israel, May 1992(Dolev D., Galil Z., and Rodeh M. eds.) 154–170.

[16] Herlihy, M., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems, 12,* 3 (July 1990) 463–492. Preliminary version appeared as Axioms for concurrent objects. in *Proc. 14th ACM Symp. on Principles of Programming Languages,* (Jan. 1987) 13–25.

[17] Herlihy, M. Wait free synchronization. *ACM Transactions on Programming Languages and Systems, 13,* 1 (January 1991) 124–149.

[18] Herlihy, M. A Methodology For Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745-770, November 1993.

[19] D.Gawlik. Processing 'hot spots' in high performance systems. In *Proceedings COMPCON'85,* 1985.

[20] IBM Corporation. POWER PC reference manual.

[21] Israeli, A., and Li, M. Bounded time stamps. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science,* 1987.

[22] Israeli A., Shaham A., and Shirazi A. Linear time snapshot protocols for unbalanced systems. *Proceedings of the 7th International Workshop on Distributed Algorithms..*

[23] Kirousis, L. M., Spirakis, P., and Tsigas, Ph. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. *Proceedings of the 5th International Workshop on Distributed Algorithms.* October 1991 229–241.

[24] L. Lamport. On Interprocess Communication, Parts I and II. *Distributed Computing,* Volume 1, 77–101, 1986.

[25] MIPS Computer Company. The MIPS RISC Architecture.