Fast Implementations of Shared Objects using Fetch&Add

Panagiota Fatourou
Department of Computer Science
University of Crete
& FORTH ICS
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Department of Computer Science
University of Ioannina
nkallima@cs.uoi.gr

February 17, 2010

Abstract

In this paper we present efficient implementations of shared objects from Fetch&Add registers. We present the first snapshot implementation which has time complexity $O(1)$ for both scan and update. Using this implementation we can directly obtain an active set with constant time complexity for all three operations, join, leave and getSet. We also present the first universal construction which has time complexity $O(1)$. This construction proves that a lower bound of $\Omega(\log n)$ on the number of LL/SC registers required for implementing a universal object can be beaten if we employ a constant number of Fetch&Add registers (in addition to one LL/SC register). Our algorithms improve upon existing implementations of these objects and they have several interesting implications on the complexity of many known implementations of other shared objects that use any of the implemented objects as one of their components (i.e., existing partial snapshot algorithms and others). Despite these implications, they are themselves very simple and some of them are of practical interest in several cases.
1 Introduction

In shared memory systems, where processes communicate by (concurrently) accessing shared objects, the design of efficient such objects from simpler that are provided by the hardware is of major importance. The most basic shared object is a read-write register which atomically supports the operations read and write. Unfortunately, such registers are not strong enough to simulate other useful objects [22] in a wait-free manner. (wait-freedom is a strong progress condition according to which each process should finish the operation it executes within a finite number of its own steps independently of the speeds of other processes or their failures). More specifically, Herlihy [22] introduced the consensus hierarchy which characterizes the power of a shared object to wait-free simulate (using additionally read-write registers) other objects. A shared object with consensus number $n$ can simulate any other object in a system of $n$ processes.

The strongest types of registers are CAS and LL/SC registers which have infinite consensus number. These registers are strong enough to simulate any other shared object for any number of processes. Such registers have also been used for speeding up the simulation of weaker shared objects which, when implemented from read-write registers, is relatively slow. An indicative example of such an object is the snapshot object [1, 4, 8] which consists of $m$ components, each capable of storing a value and supports the operations scan to get a ”consistent” vector of the values of the components and update($i, v$) to update the $i$-th component with the value $v$. It is remarkable that all snapshot implementations from read-write registers [1, 9, 4, 5, 11, 12, 14, 27, 26, 19, 16] are much slower than others using LL/SC registers [11, 31, 30, 35, 17]). Although CAS (or LL/SC) registers are currently provided by several systems, using as few such registers as possible is highly desirable since their current implementation is much slower than that of simpler types of registers, and therefore they should not be used in a wide scale.

In this paper, we study efficient implementations of shared objects using a weaker type of register, namely the Fetch&Add register. A Fetch&Add register $R$ supports, in addition to read, the operation $\text{Fetch&Add}(R, x)$ which adds some value $x$ to register $R$ and returns the previous value of $R$. Fetch&Add registers stand on the second level of the consensus hierarchy and therefore they are fairly weaker than LL/SC and CAS. Fetch&Add has performance advantages [21] in comparison to other synchronization primitives (like CAS, or LL/SC); in brief, a Fetch&Add requires only one memory access which minimizes serialization delays, it is combinable [20], and excessive contention for Fetch&Add variables can be reduced or eliminated by using appropriate software techniques [37]. In some architectures (i.e., the Origin2000 which is a ccNUMA machine) a Fetch&Add is implemented in-memory (bypassing the cache and its coherence protocol). This implementation of Fetch&Add was proved to be much faster under contention than the implementation of LL/SC which is integrated with the coherence protocol in this architecture [34]. Another difference between CAS and Fetch&Add in favor of Fetch&Add comes from the fact that to issue a CAS the processor has to provide three source operands (an address, an old value, and a new value). By contrast, to issue a Fetch&Add, it only needs to provide two source operands (address, increment). Store instructions do provide two sources, an address and a data word; however, especially in RISC architectures, it is very rare for an instruction to provide three sources. On the other hand, both Fetch&Add and CAS return a value, i.e. they need a destination operand as well; this is incompatible with store instructions (at least in RISC architectures), which do have two source operands but no destination operand inside the processor. However, all but one of the implementations presented in this paper do not use the returned value of the Fetch&Add (i.e., they work correctly with the weaker type of an Add register which atomically supports just the operation Add). For all these reasons, implementing
**Fetch&Add** (or **Add**) in hardware seems to be more appealing than **LL/SC**, **CAS** or other powerful instructions.

We first present a simple and efficient implementation, called **BSW-Snap**, of a single-writer, binary snapshot object from a single **Fetch&Add** register storing \( n \) bits, where \( n \) is the number of processes (in a single-writer snapshot, each of the \( n \) component can be updated by just one process; a snapshot is binary if each component stores a binary value). **BSW-Snap** has (shared-access) time complexity \( O(1) \) for both **SCAN** and **UPDATE**. In current systems where registers of 64 bits (or even 128 bits) are available, the algorithm works with just one single-word **Fetch&Add** register for up to 64 (or 128, respectively) processes. Thus, the algorithm is of some practical interest.

From a theoretical perspective, our results are much more interesting. If a larger **Fetch&Add** register of \( nd \) bits is used, we obtain a \( d \)-value (single-writer) snapshot implementation (called **SW-Snap**) with constant time complexity for **SCAN** and **UPDATE** using a single **Fetch&Add** register. To the best of our knowledge, this is the first snapshot algorithm which exhibits constant time complexity for both **SCAN** and **UPDATE**. The algorithm significantly improves upon the (single-writer version of the) **f-arrays** snapshot algorithm [30] which (from a theoretical perspective) has the best known time complexity to now, namely \( O(1) \) for **SCAN** and \( O(\log n) \) for **UPDATE**. **SW-Snap** improves upon it by exhibiting constant time complexity for both operations. Moreover, the **f-arrays** snapshot uses \( O(n) \) **LL/SC** registers, some of which store \( O(nd) \) bits, whereas **SW-Snap** uses just one **Fetch&Add** register (which is weaker than **LL/SC**) and this register is not of a larger size than some of the registers employed by the **f-arrays** snapshot. The **f-arrays** algorithm works for multi-writer snapshots, whereas our algorithms (**BSW-Snap** and **SW-Snap**) are single-writer snapshot implementations. However, using more **Fetch&Add** registers of unbounded size, we can get the first multi-writer snapshot algorithm, called **MW-Snap**, which achieves \( O(1) \) time complexity for both **SCAN** and **UPDATE** (notably without using as strong types of registers as **CAS** or **LL/SC**).

As a direct application of **BSW-Snap**, we obtain an implementation of an active set object (called **F-ActSet**) which has time complexity \( O(1) \) and uses a single **Fetch&Add** register of \( n \) bits. An active set object [3] supports three operations, **JOIN**, **LEAVE**, and **GETSET**. A process executes operations **JOIN** and **LEAVE** to identify its intention to join or leave the active set and **GETSET** to discover the processes that participate in the set. **F-ActSet** is the first implementation of active set that exhibits constant time complexity for all operations. We believe that this algorithm is of practical interest. In systems where \( n \) is too large to have \( n \) bits fitting in a memory word, **F-ActSet** can slightly be modified to obtain an active set algorithm which uses \( n/b \) **Fetch&Add** registers of \( b \) bits each, and has time complexity \( O(n/b) \).

Attiya, Guerraoui and Ruppert [10] presented an adaptive implementation (which we will call **AGR**) of an active set object using **CAS** and **Fetch&Add** (a restricted version of **Fetch&Add** where the increase can be just by one). **AGR** has time complexity \( O(1) \) for **JOIN** and **LEAVE** but the time complexity of **GETSET** is unbounded since it depends on the number of **JOIN** operations executed in the entire execution. **AGR** uses a **Fetch&Inc** register which takes an unbounded number of values, and a large **CAS** register which stores up to \( \Theta(k) \) intervals, where \( k \) is the maximum, over all operations, interval contention of the operation (this, in the worst case, can be \( n \)). Attiya, Guerraoui and Ruppert [10] employ **AGR** to design a partial snapshot object where a **SCAN** of \( r \) components runs in \( O(r^2) \) but the **UPDATE** is unbounded (since it calls **GETSET**). **F-ActSet** improves these results, since it has constant time complexity for **GETSET**. Moreover, **F-ActSet** uses just one **Fetch&Add** register which in the worst case it is not larger than the **CAS** register used in **AGR**. Incorporating **F-ActSet** into **AGR** gives a partial snapshot algorithm which has time complexity.
to store the state of the object. We can apply similar techniques to size as one of the \(O\) which store complexity \(O\). In this paper, we present the first universal construction, called \(O\) in terms of time complexity (a summary of known universal algorithms is presented in \(O\)). These algorithms employ some of the techniques described by Anderson and Moir [7] and combine cope with large objects (i.e., objects that need a large amount of storage to maintain their states). Apparently, the employment of Fetch&Add register). Active sets have been used to solve other synchronization problems [36, 3]. Apparently, the employment of F-ActSet may impact their performance.

Fatourou and Kallimanis have presented in [18] a family of wait-free, adaptive, universal constructions, called RedBlue. The first algorithm (F-RedBlue) has time complexity \(O(min(k, \log n))\), where \(k\) is the point contention of any operation. This is the best known wait-free universal construction in terms of time complexity (a summary of known universal algorithms is presented in Table 1). In this paper, we present the first universal construction, called SimOpt, that has time complexity \(O(1)\). SimOpt is much simpler than F-RedBlue, uses significantly less registers and achieves better time complexity. More specifically, F-RedBlue uses \(O(n)\) LL/SC registers some of which store \(O(n)\) values, whereas SimOpt uses just three registers, an LL/SC register of the same size as one of the LL/SC registers of F-RedBlue, and two Fetch&Add registers, one storing \(n\) bits and the other storing \(n\) values.

The second RedBlue algorithm (S-RedBlue) uses smaller registers than F-RedBlue but has time complexity \(O(k)\). S-RedBlue employs \(O(n)\) LL/SC registers of \(n\) bits each, a big LL/SC register, and \(n+1\) single-writer read-write registers, one for each process. SimOpt uses two Fetch&Add registers and a big LL/SC register, and achieves constant time complexity. In systems where \(n\) bits fit in one memory word, S-RedBlue can be implemented by single-word LL/SC registers using the technique in [32]. This increases its time complexity to \(O(k + B)\), where \(B\) is the number of words required to store the state of the object. We can apply similar techniques to SimOpt to get a much simpler algorithm (S-SimOpt) than S-RedBlue. S-SimOpt has the same time complexity \(O(k + B)\), and uses registers of the same size as those used by S-RedBlue. However, S-SimOpt uses \(O(n)\) less LL/SC registers. Also, each execution of S-SimOpt performs \(\Omega(\log k)\) less LL/SC instructions per operation than those performed in S-RedBlue.

Table 1: Summary of Universal Algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Primitives</th>
<th>Time Complexity</th>
<th>Space Overhead (# of regs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Herlihy [23]</td>
<td>consensus objects, r/w regs</td>
<td>(O(n))</td>
<td>(O(n^2W))</td>
</tr>
<tr>
<td>GroupUpdate, Afek et al. [2]</td>
<td>LL/SC, consensus objects, r/w regs</td>
<td>(O(k\log k + W + kD))</td>
<td>(O(n^2W \log n))</td>
</tr>
<tr>
<td>IndividualUpdate, Afek et al. [2]</td>
<td>LL/VC/SC</td>
<td>(O(kD\log D))</td>
<td>(O(nD + W))</td>
</tr>
<tr>
<td>F-RedBlue, Fatourou &amp; Kallimanis [18]</td>
<td>SC</td>
<td>(O(\min(k, \log n)))</td>
<td>(O(n^2 + W))</td>
</tr>
<tr>
<td>S-RedBlue, Fatourou &amp; Kallimanis [18]</td>
<td>LL/VC/SC, r/w regs</td>
<td>(O(k + W))</td>
<td>(O(n^2 + nW))</td>
</tr>
<tr>
<td>Anderson &amp; Moir [6]</td>
<td>LL/VC/SC</td>
<td>(O((n/ \min(k, M/T)))</td>
<td>(O(n^2 + n(B + MS)))</td>
</tr>
<tr>
<td>LS-RedBlue, Fatourou &amp; Kallimanis [18]</td>
<td>LL/VC/SC, r/w regs</td>
<td>(O(B + k(D + TS)))</td>
<td>(O(n^2 + n(B + kTS)))</td>
</tr>
<tr>
<td>BLS-RedBlue, Fatourou &amp; Kallimanis [18]</td>
<td>LL/VC/SC, r/w regs</td>
<td>(O((k/ \min(k, M/T)))</td>
<td>(O(n^2 + n(B + MS)))</td>
</tr>
<tr>
<td>(this paper)</td>
<td>LL/SC or CAS, Fetch&amp;Add</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

\(O(r^2)\) for scan and \(O(r^2_{\text{max}} k)\) for update, where \(r_{\text{max}}\) is the maximum number of components accessed by one partial scan in an execution.

A partial snapshot implementation from read-write registers is presented in [25]. The implementation uses an active set object, so it can be sped up by employing F-ActSet (although this would enroll the use of a Fetch&Add register). Active sets have been used to solve other synchronization problems [36, 3]. Apparently, the employment of F-ActSet may impact their performance.

Fatourou and Kallimanis have presented in [18] a family of wait-free, adaptive, universal constructions, called RedBlue. The first algorithm (F-RedBlue) has time complexity \(O(min(k, \log n))\), where \(k\) is the point contention of any operation. This is the best known wait-free universal construction in terms of time complexity (a summary of known universal algorithms is presented in Table 1). In this paper, we present the first universal construction, called SimOpt, that has time complexity \(O(1)\). SimOpt is much simpler than F-RedBlue, uses significantly less registers and achieves better time complexity. More specifically, F-RedBlue uses \(O(n)\) LL/SC registers some of which store \(O(n)\) values, whereas SimOpt uses just three registers, an LL/SC register of the same size as one of the LL/SC registers of F-RedBlue, and two Fetch&Add registers, one storing \(n\) bits and the other storing \(n\) values.

The second RedBlue algorithm (S-RedBlue) uses smaller registers than F-RedBlue but has time complexity \(O(k)\). S-RedBlue employs \(O(n)\) LL/SC registers of \(n\) bits each, a big LL/SC register, and \(n+1\) single-writer read-write registers, one for each process. SimOpt uses two Fetch&Add registers and a big LL/SC register, and achieves constant time complexity. In systems where \(n\) bits fit in one memory word, S-RedBlue can be implemented by single-word LL/SC registers using the technique in [32]. This increases its time complexity to \(O(k + B)\), where \(B\) is the number of words required to store the state of the object. We can apply similar techniques to SimOpt to get a much simpler algorithm (S-SimOpt) than S-RedBlue. S-SimOpt has the same time complexity \(O(k + B)\), and uses registers of the same size as those used by S-RedBlue. However, S-SimOpt uses \(O(n)\) less LL/SC registers. Also, each execution of S-SimOpt performs \(\Omega(\log k)\) less LL/SC instructions per operation than those performed in S-RedBlue.

Two additional adaptive RedBlue universal constructions [18] (LS-RedBlue and BLS-RedBlue) cope with large objects (i.e., objects that need a large amount of storage to maintain their states). These algorithms employ some of the techniques described by Anderson and Moir [7] and combine
them with the techniques of the RedBlue family. Using the results proved here, we can obtain much simpler versions of these algorithms which achieve the same time complexities as LS-RedBlue and BLS-RedBlue but employ $\Omega(n)$ less LL/SC registers, and reduce the number of LL/SC instructions performed in any execution significantly (by a factor of $\Omega(\log k)$ per operation).

Jayanti [29] has proved a lower bound of $\Omega(\log n)$ on the time complexity of any oblivious implementation of a universal object using LL/SC registers; an oblivious implementation of universal objects that do not exploit the semantics of the object being implemented. One of the open problems mentioned in that paper is the following: "If shared-memory supports all of read, write, LL/SC, swap, CAS, move, Fetch&Add, Fetch&Multiply, would the $\Omega(\log n)$ lower bound still hold?" SimOpt is oblivious. It follows that by using just two Fetch&Add registers (in addition to one LL/SC register), this lower bound can be beaten. It is worth-pointing out that SimOpt can be built from a constant number of single-writer snapshots (or of collect objects) in addition to one LL/SC register. This implies that any implementation of single-writer snapshot (or of collect) from LL/SC registers has time complexity $\Omega(\log n)$.

Jayanti [28] has proved a lower bound of $\Omega(n)$ on the number of local steps performed by any oblivious universal construction; SimOpt has local time complexity $O(n)$. Lower bounds on the space and time complexity of snapshot implementations from read-write registers are presented in [13, 14, 15, 33]. The lower bounds described in this paper focus solely on the time complexity of implementations of single-writer snapshots and collect objects from LL/SC registers.

Although our algorithms have a lot of interesting implications on the complexity of many current implementations of shared objects, they are themselves very simple (as expected by any algorithm that has constant time complexity). We consider this to be a major ingredient of their significance.

The universality result [22] has motivated major hardware manufacturers to include strong instructions, like LL/SC or CAS, in the instruction set of most modern processors. Weaker instructions, like Fetch&Add, can possibly be implemented in hardware more easily than LL/SC or CAS. However, this is currently performed on top of LL/SC (or CAS) in some architectures (e.g., SPARC), sometimes with uncertain progress guarantees. Obviously, these architectures have serious performance disadvantages compared to architectures that directly support Fetch&Add. Fortunately, the great majority of multicore processors (x86 and x86_64 machines) provide Fetch&Add in their instruction sets. We hope that the results proved in this paper provide some motivation for seeing primitives such as Fetch&Add (or Add) provided in the instruction set of more architectures in the future.

2 Model

The system is asynchronous and contains $n$ processes, $p_1, \ldots, p_n$, that may fail by crashing. Processes communicate via shared objects. Each shared object stores some information and offers the ability to processes to access and modify the stored information using atomic methods that may be executed by processes concurrently.

The most basic shared objects are read-write registers which are provided by the hardware in all systems. A read-write register $R$, stores a value $v$ from a set $V$ and supports two atomic operations, read($R$) and write($R, v$). Operation read($R$) returns the current value of register $R$, while write($R, v$) stores value $v$ to $R$ and returns ack. A Fetch&Add register stores a value from a set $V$ and supports two atomic operations, read($R$) and Fetch&Add($R, v$). Operation read($R$) returns the stored value of register $R$, while Fetch&Add($R, v$), where $v \in V$, adds $v$ to the value of $R$ and returns the previous value (that before the application of Fetch&Add). We remark that $v$ may
be a negative value. A single writer register can be updated only by one process. An algorithm to simulate \texttt{FetchAdd} supporting negative values from \texttt{FetchAdd} supporting only positive values is provided in the Appendix (we remark though that most current systems provide \texttt{FetchAdd} that supports any integer value).

An LL/SC register $R$ stores a value from some set and supports the atomic operations LL and SC; LL$(R)$ returns the current value of $R$; the execution of SC$(R,v)$ by a process $p$ must follow the execution of LL$(R)$ by $p$, and it is successful only if no process has performed a successful SC on $R$ since the execution of $p$’s latest LL on $R$; if SC$(R,v)$ is successful the value of $R$ changes to $v$ and true is returned. Otherwise, the value of $R$ does not change and false is returned.

A snapshot object consists of $m$ components $A_1, \ldots, A_m$, each of which stores a value from a set $V$. We assume that $V = \{0, \ldots, 2^d - 1\}$ (i.e., the value stored in each component is of $d$ bits). A process may access the information of the snapshot object by executing \texttt{SCAN} or \texttt{UPDATE}. An UPDATE$(i,v)$ stores the value $v \in V$ in component $A_i$. A \texttt{SCAN} returns a ”consistent” vector of $m$ values, one for each component. A snapshot implementation is single writer, if each component is updated by exactly one process, so only this process can apply UPDATE operations to the component. In multi writer snapshots, each process can execute UPDATE operations to any component.

An active set object supports three operations \texttt{GETSET, JOIN} and \texttt{LEAVE}. A correct implementation of an active set object satisfies the following conditions: (1) the returned set by a \texttt{GETSET} operation \texttt{GS} contains any process $p$ that has finished the execution of a \texttt{JOIN} operation $J$ before the beginning of the execution of \texttt{GS} and it has not started the execution of a \texttt{LEAVE} operation in the execution interval between the end of $J$ and the end of \texttt{GS}, and (2) the returned set by \texttt{GS} does not contain any process $p$ that has finished the execution of a \texttt{LEAVE} operation $L$ before the beginning of the execution of \texttt{GS} and it has not started the execution of a \texttt{JOIN} operation in the execution interval between the end of $L$ and the end of \texttt{GS}.

A universal object simulates any other distributed shared object. A universal object supports an operation, called APPLYOP(operation op), which simulates the execution of operation $op$ on the simulated object; APPLYOP returns the return value of operation $op$.

An implementation of an object from registers provides an algorithm for each process to simulate each operation supported by the object using the registers. An implementation of a universal object is called oblivious if it does not exploit the semantics of the type of the simulated object.

A configuration $C$ is a vector containing the states of the processes and the values of the registers at any point in time. At the initial configuration, each register contains an initial value and processes are at an initial state. A process completes the execution of a computation step, each time it accesses a shared register. An execution is a sequence of execution steps by processes.

A shared object implementation is wait-free, if each process completes the execution of any operation in a finite number of its own execution steps. The time complexity of an operation is the maximum number of steps that a process performs to complete the operation in any execution. The time complexity of an implementation is the maximum between the time complexities of its operations. The space complexity of an implementation is determined by the number, the type, and the size of the registers used by the implementation.

Linearizability [24] imposes a total order, called linearization order, to all operations performed in an execution $\alpha$. The linearization order must respect the partial order imposed by the execution intervals of operations. If $\alpha$ is linearizable, each of its operations should have the same response as the corresponding operation in the serial execution derived by the linearization order. When this holds for some operation, we say that the response of the operation is consistent. When this
Algorithm 1 Pseudocode for BSW-Snap.

```java
// Code for process p
<ack> UPDATE(boolean v){
    static boolean prev = 0;
    1. constant FValue offset = 2\text{i}-1;
    2. fetchAdd(R, (v - prev) * offset);
    3. prev = v;
    4. return ack;
}
```

holds for all operations of \( \alpha \), we say that \( \alpha \) is consistent. An implementation is linearizable if all its executions are linearizable.

A process is active at some configuration \( C \), if it has started the execution of an operation at \( C \) but it has not yet finished it. The execution interval of some operation \( op \) is the part of the execution that starts with \( op \)'s invocation and ends with \( op \)'s response. The interval contention of an operation \( op \) is the maximum number of processes that take at least one step in the execution interval of \( op \). The interval contention of an implementation is the maximum interval contention of any operation performed in any of its executions. The point contention of an operation is the maximum number of processes that are active in any configuration of the execution interval of any instance of \( op \) in any execution. An algorithm is adaptive if its time complexity depends on the (point or interval) contention of the implementation and not on the total number \( n \) of processes in the system.

3 Single-Writer Snapshots - Active Set

The BSW-Snap Algorithm. We first present BSW-Snap, a single-writer, binary snapshot implementation. Pseudocode for BSW-Snap is presented in Algorithm 1. The algorithm uses a shared fetchAdd register \( R \) of \( n \) bits, one for each process. At any configuration \( C \), the \( i \)-th bit of \( R \) stores the value used by the last UPDATE of \( p_i \) among those that have executed their fetchAdd before \( C \) (i.e., the last value written by \( p_i \) into \( R \) before \( C \)). Process \( p_i \) stores a copy of this bit in the persistent local variable \( prev \). The value of \( prev \) is appropriately used by the next UPDATE of \( p_i \) to ensure that the value of this UPDATE will successfully be stored in the \( i \)-th bit of \( R \).

An UPDATE \( u \) by \( p_i \) with the value \( v \) first performs a fetchAdd to ensure that \( v \) is written into the \( i \)-th bit of \( R \) (line 2), and then keeps a copy of \( v \) into \( prev \) (line 3). To ensure that \( v \) is stored into the \( i \)-th bit of \( R \) (let is be \( R_i \)), \( p_i \) does the following. Assume that the value of \( R_i \) is \( v_i \) at the configuration \( C \) just before \( u \) executes its fetchAdd operation. Recall that \( prev \) stores also \( v_i \) at \( C \). To ensure that \( v \) will be stored in the \( i \)-th bit of \( R \) after the execution of fetchAdd, \( u \) subtracts \( prev \) from \( v \). This results to the value \( x = v - v_i \), which when added to \( R_i \) results to \( v \) since \( v_i + x = v_i + v - v_i = v \). To ensure that only the value of \( R_i \) is affected by the fetchAdd and not the rest of the bits of \( R \), \( x \) is shifted by \( i - 1 \) bits; this is performed by multiplying it with
the value $2^{i-1}$ (line 2) (this value is stored in a local variable of $p_i$ called offset (line 1)).

Whenever, $p_i$ executes a scan operation, it simply reads the value stored in register $R$. Recall that the $i$-th bit of $R$ contains the value written by the last update of $p_i$ into component $A_i$. To extract these values from the value read in $R$, $p_i$ applies appropriate divisions (and modulo operations) to the value read in $R$ (lines 7-8).

We remark that BSW-Snap works well even on machines that do not support binary representation of data (which are though uncommon), since all the bit shifts are performed by integer multiplications and divisions (i.e., they are virtual). It is also worth mentioning that $x$ may be a negative value. More specifically, $x$ is negative whenever the value of an update by $p_i$ is smaller than the current value of $R_i$; we remark though that $x$ always stores some value in $\{-1,0,1\}$.

A scan is linearized just after the execution of its read instruction to register $R$ (line 5). An update is linearized just after the execution of its Fetch&Add instruction to register $R$ (line 2). Obviously, each operation is linearized in its execution interval. To prove that scans return consistent vectors, we introduce the following notation. Let $a$ be any execution, and let $U_j$, $j > 0$, be the update that its Fetch&Add instruction (line 2) is the $j$-th Fetch&Add that is executed in $a$. Assume that $U_j$ is executed by process $p_i$. Denote by $v_j$ the value that $U_j$ stores into component $A_i$. We prove that the following claims hold: (1) $U_j$ modifies only the $i$-th bit of register $R$, and (2) after the execution of $U_j$’s Fetch&Add instruction on $R$, the value of the $i$-th bit of $R$ is $v_j$.

To prove consistency, assume that a scan $S$ returns the vector $\vec{v} = < v_1, ..., v_n >$. Denote by $U_i^S$, $i \in \{1,..,n\}$ the last update on $A_i$ (by $p_i$) that executes its Fetch&Add before the read of $R$ by $S$. Recall that update operations on other components do not affect the value of the $i$-th bit of $R$. Since $S$ returns $v_i$ for $A_i$, it follows that $U_i^S$ writes $v_i$ into $R$. By the way linearization points are assigned, no other update on $A_i$ is linearized between $U_i^S$ and $S$. Thus, $S$ returns a consistent value for $A_i$. Due to lack of space, the full proof of correctness of BSW-Snap is provided in the attached appendix.

In BSW-Snap, scan and update operations execute just one step each (i.e., the access to $R$; the rest of the steps of both operations are local). Thus, the time complexity of of BSW-Snap is one step (i.e., $O(1)$) which is optimal. Apparently, BSW-Snap is also optimal in terms of the number of registers that it uses and their size (it uses one register of $n$ bits, one for each of the $n$ components).

**Theorem 3.1** BSW-Snap is a linearizable, wait-free implementation of a binary, single-writer snapshot object from one Fetch&Add register of $n$ bits. The time complexity for both scan and update is one step.

**The SW-Snap Algorithm.** BSW-Snap can be generalized to provide a $d$-value, single-writer snapshot (called SW-Snap) in a straightforward way. As BSW-Snap, SW-Snap uses a Fetch&Add register $R$ but now $R$ contains $dn$ bits. $R$ is partitioned into $n$ chunks of $d$ bits each, one for each process. Process $p_i$ owns the $i$-th chunk of $d$ bits. Each chunk stores a value from the set $\{0,\ldots,2^d-1\}$. Register $R$ stores values from the set $\{0,\ldots,2^{nd}-1\}$. SW-Snap works in a similar way as BSW-Snap. Pseudocode for SW-Snap is presented in Algorithm 2.

The proof of correctness for SW-Snap is similar to that for BSW-Snap (and therefore it is omitted). As BSW-Snap, SW-Snap is optimal in terms of its time complexity, the number of registers it uses and their size.

**Theorem 3.2** SW-Snap is a linearizable, wait-free implementation of a single-writer, $d$-value snapshot object from one Fetch&Add register of $dn$ bits. The time complexity for both scan and update is one step.
The F-ActSet algorithm. Using an instance of BSW-Snap, we can get an implementation (called F-ActSet) of an active set object in a direct way. All components of the binary snapshot are initialized to false. Each process \( p_i \) updates its component with the value true (false) whenever it executes a JOIN (LEAVE, respectively). GETSET performs a scan and includes a process \( p_i \) in its returned set if the value of \( A_i \) equals true. If the binary, single-writer snapshot object used by F-ActSet is implemented using an instance of BSW-Snap, an execution of any of the three operations of F-ActSet executes just one step. Moreover, the implementation uses one Fetch&Add register which stores \( n \) bits.

**Theorem 3.3** F-ActSet is a correct implementation of an active set object from one Fetch&Add register of \( n \) bits. The time complexity for JOIN, LEAVE and GETSET is one step.

In case \( n \) is big enough to fit in one memory word, we employ \( w = \lceil n/b \rceil \) (where \( b \) is the number of bits comprising a memory word) snapshot objects to obtain an active set implementation with time complexity \( O(1) \) for JOIN and LEAVE, and \( O(w) \) for GETSET. In this implementation (called MWF-ActSet), a process \( p_i \) is assigned the \( (i \mod b) \) component of the \( \lceil i/b \rceil \) snapshot object. When \( p_i \) executes JOIN (LEAVE), it updates the component assigned to it with the value true (false, respectively). When \( p_i \) executes GETSET it performs a scan operation on each of the \( w \) snapshot objects used by F-ActSet and includes a process in its set if the component assigned to the process contains the value true. The pseudocode of MWF-ActSet is presented in Algorithm 3.

Let \( \alpha \) be any execution and let \( GS \) be any GETSET executed by some process \( p_j \) in \( \alpha \). Obviously, if any process \( p_j \) has finished the execution of a JOIN operation \( J \) before the beginning of \( GS \), and it has not started the execution of a LEAVE operation between the end of \( J \) and the end of \( GS \), the SCAN of \( p_i \) on \( S_{\lceil j/b \rceil} \) will return true for the \( j \mod b \) component, and therefore, \( p_j \) will be included in the set returned by \( GS \). Similarly, if any process \( p_j \) has finished the execution of a LEAVE operation \( L \) before the beginning of \( GS \), and it has not started the execution of a JOIN operation between the end of \( L \) and the end of \( GS \), the SCAN of \( p_i \) on \( S_{\lceil j/b \rceil} \) will return false for the \( j \mod b \) component, and therefore, \( p_j \) will not be included in the set returned by \( GS \).
Algorithm 3 Pseudocode for MWF-ActSet.

```plaintext
type Pindex {1, ..., n}; // Type of variables that store values from the set {1,...,n}
shared binary single-writer snapshots S1,...,Sw; // each of the S1,...,Sw is comprised of b components with initial value false;

// Code for process pi
<ack> join () {
    1. S⌈i/b⌉.update(i, true);
    return ack;
}

<ack> leave () {
    2. S⌈i/b⌉.update(i, false);
    return ack;
}

Set getSet () {
    Set ret = \emptyset;
    boolean act[1..b];
    Pindex j, l;
    3. for l = 1 to w do{
        4. act = S_l.scan();
        5. for j = 1 to b do
            6. if(act[j] == true) ret = ret \cup \{p(l-1)b+j\};
    }
    7. return ret;
}
```

Theorem 3.4 MWF-ActSet is a correct implementation of an active set object from w single-word Fetch&Add registers. The time complexity for join and leave is one step, and the time complexity of getSet is w steps.

4 Universal Constructions - Lower Bounds

The SimOpt algorithm. We present a wait-free universal construction (Algorithm 4) which has time complexity O(1). The algorithm (called SimOpt) uses an LL/SC register sp, an active set object AS, and a snapshot object S. The LL/SC register stores the state of the simulated object, a vector of n bits identifying whether the current operation (if any) of each process has been applied to the simulated object, and a set of return values one for each process (techniques to reduce the size of this register are discussed below).

Whenever a process pi wants to apply some operation op to the simulated object, it first announces op in the single-writer snapshot object S (by updating its component). Then, it executes a join operation to inform all other processes that it is currently executing an operation. Afterwards, pi executes the attempt routine to ensure that its operation has been applied to the object (the details of attempt are discussed below). After ensuring that the application of op has been simulated, pi updates its component with the special value ⊥, and executes a leave operation on AS to inform the other processes that its operation has been completed. Then, pi executes attempt once more to eliminate any evidence of the operation it has executed.

We now discuss the details of attempt. First, pi executes an LL to register sp, and then a getSet to get the set of processes currently executing some operation (in order to help them completing their operations). Next, pi executes a scan to discover what other operations are active, and applies them (in addition to its own) to a local copy lsp of the state of the simulated object. It also calculates the return value for each operation it applies and stores it in the appropriate field of lsp. Afterwards, pi tries to update sp by executing an SC instruction. We prove that it is enough for pi to execute these steps twice to guarantee that its operation op has been applied to the simulated object (or the evidence of its last operation has been eliminated).

The snapshot object is implemented using an instance of SW-Snap. Similarly, the active set object is implemented using an instance of F-ActSet. Theorems 3.2 and 3.3 imply that lines 1, 2,
Algorithm 4 Pseudocode for SimOpt.

```c
typedef Pindex {1, ..., n};
struct StatePtr {
    boolean applied[1..n];
    ret_val rvals[1..n];
    state st;
};
shared ActiveSet AS = ∅;
shared single-writer snapshot S;
shared StatePtr sp = <<F, ..., F>, <⊥, ..., ⊥>, ⊥>;

Code for process p_i

ret_val APPLYOp(operation op) {
1. S.UPDATE(i, op);
2. AS.JOIN();
3. attempt();
4. AS.LEAVE();
5. S.UPDATE(i, ⊥);
6. attempt();
7. return sp.rvals[i];
}

<ack> attempt() {
    StatePtr lsp;
    Pindex i, j;
    Set act;
    operation ops[1..n];
8. for j=1 to 2 do{
9.    lsp = LL(sp);
10.   act = AS.GETSET();
11.   ops = S.scan();
12.   for i=1 to n do { // local loop
13.      if(i ∈ act AND lsp.applied[i] == false)
14.         apply ops[i] to lsp.st and store
15.         into lsp.rvals[i] the return value;
16.         if(i ∈ act) lsp.applied[i] = true;
17.         else lsp.applied[i] = false;
18.    }
19.    SC(s, lsp);
    }
20. return ack;
}
```

4 and 5 of attempt are executed in $O(1)$ time. Thus, the time complexity of APPLYOp is bounded by the time complexity of attempt. Theorems 3.3 and 3.2 imply that lines 10 and 11 of APPLYOp are executed in $O(1)$ time. We remark that lines 12-16 perform only local computation. Therefore, the time complexity of APPLYOp is $O(1)$.

**Theorem 4.1** SimOpt is a linearizable, wait-free implementation of a universal object using two Fetch&Add registers and one LL/SC register. The time complexity of SimOpt is $O(1)$.

SimOpt uses three registers, a Fetch&Add register of $n$ bits for the implementation of the active set, a Fetch&Add register of $nd$ bits for the implementation of the single-writer snapshot (where $d$ is the number of bits required to code any operation and its parameters), and a big LL/SC register. We remark that several techniques can be applied to SimOpt to reduce the size of the registers it uses. Doing so would increase its time complexity but it would make the algorithm more practical. Some of these techniques are briefly described below.

The snapshot object can be replaced by a set of $n$ single-writer read-write registers, one for each process. When $p_i$ wants to apply an operation $op$, announces it by writing $op$ (and its parameters) in its single-writer register. When $p_i$ arrives to the root and executes attempt, it should discover the operations that other processes perform (to help them) by reading the single-writer registers of the processes returned by its GETSET. This increases the time complexity of SimOpt to $O(k)$ (where $k$ is the point contention) but it removes the need for the Fetch&Add register of $nd$ bits.

The array of $n$ returned values stored in $sp$ can be replaced by a process id as follows. Each process maintains a set of $n$ additional single-writer registers, one for each process (so there are $O(n^2)$ such registers in total). Each time a process $p_i$ executes attempt, it records a response for the currently active processes (i.e., for those returned by the execution of its GETSET) in its appropriate single-writer registers. Then, it tries to store its id (together with the new state of the object and the updated version of applied) into $sp$. A process finds the response for its current
operation in the appropriate single-writer register of the process recorded in \( sp \). The pseudocode of a modified version of SimOpt (called S-SimOpt) that incorporates the techniques discussed above is presented in Algorithm 6.

Jayanti et al. [32] presented an implementation of a \( w \)-word LL/SC register from single-word LL/SC registers. The time complexity of the transformation is \( O(w) \), and the number of single-word LL/SC registers required is \( O(n^2w) \). We can apply this transformation to implement \( sp \) from single-word LL/SC objects. The resulted version of the algorithm has time complexity \( O(k + W) \) (where \( W \) is the number of words required to store the state of the object), and uses \( O(n^2 \text{LL/SC}) \) single-word read-write registers, one Fetch\&Add register of \( n \) bits and \( O(n^2W) \) single-word LL/SC registers.

In cases where the implemented object is large, and a big amount of storage is required to store its state, techniques similar to that presented in [7] (and later employed by the RedBlue family of algorithms [18]) can be employed to take adaptive, wait-free implementations of a universal object for large objects. The resulted algorithms significantly simplify LS-RedBlue and BL-S-RedBlue [18]. Due to lack of space, the pseudocodes for these algorithms will be provided in the full paper.

**Lower Bounds.** Jayanti [29] has proved that any oblivious implementation of a universal object from LL/SC registers has time complexity \( \Omega(\log n) \). Our results imply that this lower bound can be beaten if a constant number of Fetch\&Add registers is used in addition to an LL/SC register. SimOpt uses a single-writer snapshot object and an active set object implemented using a single-writer snapshot object Thus, the \( \Omega(\log n) \) lower bound can be beaten if a constant number of single-writer snapshot objects are used in addition to an LL/SC register. Finally, it is remarkable that SimOpt works correctly even if the snapshot object \( S \) and the active set \( AS \) are replaced by instances of an implementation of a collect object.

**Theorem 4.2** A lower bound of \( \Omega(\log n) \) holds on the time complexity of any implementation of (1) a single-writer snapshot object, and (2) a collect object, from LL/SC registers.
5 Multi-Writer Snapshots

MW-Snap (Algorithm 5) uses a single-writer snapshot object $S$ (implemented using SW-Snap). Each component $A_i$ of $S$ stores a vector of $m$ pairs (each pair is of type $\langle value, seq \rangle$) representing the local view of process $p_i$ for the values of the $m$ components of the simulated snapshot. The algorithm uses $m$ Fetch&Add registers, one for each component of the simulated snapshot. Whenever a process $p_i$ executes an UPDATE to the $j$-th component of the simulated object, it gets a new sequence number for this component by performing a Fetch&Add (with parameter equal to 1) on $seq[j]$. Then, it updates its local view $lview$ for the component values of the simulated object to contain the pair $\langle$ new value, new sequence number for $j\rangle$ at its $j$-th position. Finally, the process executes an update to write its new local view in the $i$-th component of $S$. To execute a scan on the simulated object, a process performs a scan on $S$ and chooses the value with the largest timestamp among the $j$-th pairs of the vectors read in the components of $S$, as the return value for the $j$-th component.

The proof of correctness of MW-Snap is pretty simple and it will appear in the full version.

Algorithm 6 Pseudocode of S-SimOpt.

type Pindex {1, ..., n};
struct StatePtr
   boolean applied[1..n];
   Pindex pid;
   state st;
};
shared ActiveSet AS = ∅; // Active set for $n$ processes.
shared single-writer snapshot $S$; // Snapshot of $n$ components, each initialized to ⊥
shared StatePtr $sp = <<F,...,F>, <⊥,...,⊥>, ⊥>>$;
shared ret_val rvals[1..n][1..n] = {{⊥, ..., ⊥}, ..., {⊥, ..., ⊥}};
shared operation type $ops[1..n] = {⊥, ..., ⊥}$;

// Code for process $p_i$
void attempt()
{
    StatePtr lsp;
    Pindex j, l, q;
    Set act;
    for $j=1$ to 2 do{
        lsp = LL($sp$);
        q = lsp.pid;
        lsp.pid = $i$;
        act = AS.GETSET();
        for $l=1$ to $n$ do // local loop
            if($l ∈ act$ AND $lsp.applied[l] == false$)
                apply $ops[l]$ to lsp.st and store into $rvals[i][l]$ the return value;
            else if($l ∈ act$)
                $rvals[i][l] = rvals[q][l]$;
                if($l ∈ act$) lsp.applied[l] = true;
                else lsp.applied[l] = false;
        }
    SC($sp$, lsp);
}

Acknowledgments
We would like to thank Vassilios Dimakopoulos, Manolis Katevenis, and Dimitris Nikolopoulos for several useful discussions on the differences of Fetch&Add and LL/SC (or CAS) when implemented on hardware.
References


Algorithm 7 Pseudocode for MI-Snap.

type Pindex {1, ..., n};  // Type of registers that store values from {1,...,n}.
type CValue {0, 1, ..., 2^d - 1};  // Type of registers that store values from {0,1,...,2^d - 1}.
type FValue {0, 1, ..., 2^{nd+T} - 1};  // Type of registers that store values from {0,1,...,2^{nd+T} - 1}.

shared Fetch&Add register R = 0 of type FValue;  // R consists of dn+T bits.

data [] scan(void){
    FValue s;
    CValue snap[1..n+1];
    Pindex i;

    1. s = read(R);
    2. for i = n+1 down to 1 do{
        3. snap[i] = s / 2^d(i-1);
        4. s = s mod 2^d(i-1);
    }

    return snap[1..n];  // Shallow the (n+1)-th element of snap.
}

<ack> update(CValue v){  // For process pi.
    static CValue prev = 0;
    FValue offset;

    5. offset = 2^d(i-1);
    6. Fetch&Add(R, v * offset + (2^{dn} - prev * offset));
    7. prev = v;
    return ack;
}

A Single Writer Snapshot from Fetch&Add that does not Support Negative Values

In SW-Snap it is possible the new value that an update by p_i wants to write to the i-th chunk of R to be smaller than its current value. In this case, the difference between the new and the old value is negative, and thus the parameter of Fetch&Add is a negative value. We present a single writer snapshot implementation, (MI-Snap) from Fetch&Add registers that does not support negative values. MI-Snap is mainly of theoretical interest since in the majority of modern machines, Fetch&Add supports negative values. As in SW-Snap, only a Fetch&Add register R is used. Now, register R is partitioned into n + 1 chunks (and not in n chunks as in SW-Snap). Each of the first n chunks consists of d bits and stores a component’s value. The n + 1-th chunk of bits contains T > 0 bits.

In order to update the value of the i-th chunk with a new value v, the old value of the chunk must first become zero and then the new value should be added. In SW-Snap, Fetch&Add added the difference between the new and the old value which however may be negative. In MI-Snap, the value of the updated chunk becomes zero by adding the difference of the value 2^{dn} and the chunk’s current value. This converts to zero the bits of the i-th chunk, and adds 1 to the first bit of the (n + 1)-th chunk. The new value v is also added to R. This has as an effect an increase of the value of the (n + 1)-th chunk by one each time an update on A_i with any value v takes place, while additionally v is stored in the i-th chunk of R. Thus, at most 2^T updates can be executed during any execution of the algorithm. Pseudocode of MI-Snap is presented in Algorithm 7.

B Correctness proof of BSW-Snap

Let a be any execution. Recall that U_j, j > 0, is the update that its Fetch&Add instruction (line 2) is the j-th Fetch&Add executed in a. Assume that U_j is executed by some process p_i. Recall that v_j denotes the (binary) value that U_j wants to store into component A_i. Obviously, v_j ∈ {0, 1}. Denote by prev_j the value of (p_i's) variable prev just before the execution of line 3 by U_j. By the
pseudocode, $\text{prev}_j$ stores the value of the last UPDATE executed by $p_i$ before $U_j$ (or the initial value if such an UPDATE does not exist). It follows that $\text{prev}_j \in \{0, 1\}$. Denote by $\text{cur}_j$ the value of the $i$-th bit of register $R_i$ just before the execution of line 2 by $U_j$. Obviously $\text{cur}_j \in \{0, 1\}$.

**Lemma B.1** Consider any index $j > 0$ and assume that $U_j$ is executed by some process $p_i$ on component $A_i$. If $\text{cur}_j - \text{prev}_j + v_j \in \{0, 1\}$, only the $i$-th bit of register $R_i$ may change by the execution of line 2 of $U_j$.

**Proof:** Let $\text{add}_j = v_j - \text{prev}_j$. By the pseudocode (line 2), it follows that the `Fetch&Add` instruction of line 2 of $U_j$ adds the value $\text{add}_j \times \text{offset}$ to $R_i$. Since it holds that $v_j \in \{0, 1\}$ and $\text{prev}_j \in \{0, 1\}$, it follows that $\text{add}_j \in \{-1, ..., 1\}$. We distinguish the following cases:

- Assume that $\text{add}_j = -1$. Since $\text{cur}_j + \text{add}_j \in \{0, 1\}$, it follows that $\text{cur}_j = 1$.
- Assume that $\text{add}_j = 0$. Then, no bit is modified by adding 0 to $R_i$.
- Assume that $\text{add}_j = 1$. Since $\text{cur}_j + \text{add}_j \in \{0, 1\}$, it follows that $\text{cur}_j = 0$.

In all these cases, it follows that just the $i$-th bit of register $R_i$ may be modified by the execution of $U_j$’s `Fetch&Add` instruction. \hfill \qed

**Lemma B.2** Consider any index $j > 0$ and assume that $U_j$ is executed by some process $p_i$ on component $A_i$. The following claims hold:

1. $U_j$ modifies only the $i$-th bit of register $R_i$.
2. After the execution of $U_j$’s `Fetch&Add` instruction on $R_i$ (line 2), the value of the $i$-th bit of $R_i$ is $v_j$.

**Proof:** By induction on $j$. Consider any $j \geq 1$. Assume that the claims hold for every $j' < j$. We prove that the claims hold for $j$.

We first prove that $\text{cur}_j = \text{prev}_j$. If $j = 1$, it holds that $\text{cur}_j = \text{prev}_j = 0$ (by initialization). Therefore, $\text{cur}_j = \text{prev}_j$ in this case. If $j > 1$, by induction hypothesis (claim 2), it follows that the $i$-th bit of $R_i$ contains the value $v$ used by the last UPDATE on $A_i$ before $U_j$. By the induction hypothesis (claim 1), UPDATE operations on components other than $A_i$ do not affect the value of the $i$-th component of $R_i$. Thus, $\text{cur}_j = v$. By the code, it follows that $\text{prev}_j = v$. Thus, $\text{cur}_j = \text{prev}_j$.

We now prove claim (1). Assume, by the way of contradiction, that $U_j$ modifies the $l$-th bit of register $R_i$ for some $l \neq i$. By Lemma B.1, it follows that $\text{cur}_j - \text{prev}_j + v_j \notin \{0, 1\}$. Since
cur_j = prev_j, it follows that cur_j - prev_j + v_j = v_j. Thus, it must be that v_j \notin \{0, 1\}, which is a contradiction.

Finally, we prove claim (2). By the code, it follows that the value of the i-th bit of R after the execution of U_i's Fetch&Add will be cur_j - prev_j. Since cur_j = prev_j, it follows that cur_j - prev_j + v_j = v_j, as needed.

To prove consistency, assume that a scan S returns the vector \( \vec{v} = < v_1, ..., v_n > \). Denote by \( U_i^S \), \( i \in \{1, ..., n\} \) the last update on \( A_i \) (by \( p_i \)) that executes its Fetch&Add before the read of R by S. Recall that update operations on other components do not affect the value of the i-th bit of R. Since S returns \( v_i \) for \( A_i \), it follows that \( U_i^S \) writes \( v_i \) into R. By the way linearization points are assigned, no other update on \( A_i \) is linearized between \( U_i^S \) and S. Thus, S returns a consistent value for \( A_i \).

C Correctness proof of SimOpt

Let \( a \) be any execution. Let \( op_i^j \) be the j-th operation executed by some process \( p_i, i \in \{1, ..., n\} \).

By the pseudocode, it follows that \( op_i^j \) executes two instances of function attempt. Let \( \pi_{2j-1} \) be the instance executed at line 3 and let \( \pi_{2j} \) be the instance executed at line 6 (see Figure 2).

In order to assign linearization points, we first prove some technical lemmas. By the pseudocode (lines 8, 9, 17), the following observation is derived.

Observation C.1 Consider any execution \( \pi_i^j, j > 0 \), of attempt. There are at least two successful SC instructions in the execution interval of \( \pi_i^j \).

Lemma C.2 Consider any execution \( \pi_i^j, j > 0 \), of attempt by some process \( p_i \). The following claims hold:

1. \( \text{sp.applied}[i] \) is equal to true just after the end of \( \pi_{2j-1} \).

2. \( \text{sp.applied}[i] \) is equal to false just after the end of \( \pi_{2j} \).

Proof: We first prove Claim (1). By its definition, \( \pi_{2j-1} \) is an instance of attempt called on line 3. Assume, by the way of contradiction, that \( \text{sp.applied}[i] \) is equal to false just after the end of \( \pi_{2j-1} \). By Observation C.1, there are at least two successful SC instructions in the execution interval of \( \pi_{2j-1} \). Thus, the last successful SC instruction in \( \pi_{2j-1} \) stores false to \( \text{sp.applied}[i] \). Let \( SC_x \) be this SC instruction and let \( LL_x \) be its matching LL instruction. Assume that \( SC_x \) is executed by some instance \( x \) of attempt. Since \( SC_x \) stores false to \( \text{sp.applied}[i] \), it follows (code line 15) that \( p_i \) is not returned as an active process by the GETSET executed by \( x \) (line 10). Thus, the execution of GETSET by \( x \) and therefore its \( LL_x \), precedes the execution of JOIN by process \( p_i \) (line 2). Since the execution of JOIN by \( p_i \) precedes the beginning of \( \pi_{2j-1} \), it follows that \( LL_x \) precedes the beginning of \( \pi_{2j-1} \). By Observation C.1, there are at least two successful SC instructions in the execution interval of \( \pi_{2j-1} \). Since, \( SC_x \) is the last one, it follows that there is at least one successful SC instruction between the beginning of \( \pi_{2j-1} \) and \( SC_x \). Therefore, there is at least a successful SC instruction between \( LL_x \) and \( SC_x \). Thus, \( SC_x \) is an unsuccessful SC instruction, which is a contradiction. Thus, claim (1) holds.
We now prove claim (2). By its definition, $\pi_{2j-1}^i$ is an instance of attempt called at line 6. Assume, by the way of contradiction, that $sp.applied[i]$ is equal to true just after the end of $\pi_{2j}^i$. By Observation C.1, there are at least two successful SC instructions in the execution interval of $\pi_{2j}^i$. Thus, the last successful SC instruction in $\pi_{2j}^i$ stores true to $sp.applied[i]$. Let $SC_x$ be this SC instruction and let $LL_x$ be its matching LL instruction. Assume that $SC_x$ is executed by some instance $\pi_x$ of attempt. Since $SC_x$ stores true to $sp.applied[i]$, it follows that $p_i$ is included in the set of processes returned by the getSet executed by $\pi_x$ (line 10). Thus, the execution of getSet by $\pi_x$ and therefore its $LL_x$, precedes the execution of leave by process $p_i$ (line 4). Therefore, $LL_x$ precedes the beginning of $\pi_{2j}^i$. By Observation C.1, there are at least two successful SC instructions in the execution interval of $\pi_{2j}^i$. Since, $SC_x$ is the last one, it follows that there is at least one successful SC instruction between the beginning of $\pi_{2j}^i$ and $SC_x$. Therefore, there is at least a successful SC instruction between $LL_x$ and $SC_x$. Thus, $SC_x$ is an unsuccessful SC instruction, which is a contradiction. Thus, claim (2) holds.

For the rest of the proof we introduce the following notation. Let $C_0$ be the initial configuration. At $C_0$, $sp.applied[i]$ is equal to false. Lemma C.2 implies that just after $\pi_1^1$, $sp.applied[i]$ is equal to true. Let $C_1^1$ be the first configuration between $C_0$ and the end of $\pi_1^1$ at which $sp.applied[i]$ is equal to true. Lemma C.2 implies that just after $\pi_2^1$, $sp.applied[i]$ is equal to false. Let $C_2^1$ be the first configuration after $C_1^1$ such that $sp.applied[i]$ is equal to false. Obviously, $C_2^1$ precedes the end of $\pi_2^1$. Consider any integer $j > 1$. Lemma C.2 implies that just after $\pi_{2j-2}^1$, $sp.applied[i]$ is equal to false, while just after $\pi_{2j-1}^1$, $sp.applied[i]$ is equal to true. Let $C_{2j-1}^1$ be the first configuration between the end of $\pi_{2j-2}^1$ and the end of $\pi_{2j-1}^1$ such that $sp.applied[i]$ is equal to true. Lemma C.2 implies that just after $\pi_{2j}^1$, $sp.applied[i]$ is equal to false. Let $C_{2j}^1$ be the first configuration after $C_{2j-1}^1$ such that $sp.applied[i]$ is equal to false. Obviously, $C_{2j}^1$ precedes the end of $\pi_{2j}^1$. Figure 2 illustrates the above notation.

Since the value of $sp.applied[i]$ can change only by the execution of an SC instruction on register $sp$, it follows that at $C_{2j}^1$ (at $C_{2j}^2$) a successful SC on $sp$ is executed. Let $SC_{2j-1}^2$ ($SC_{2j}^2$, respectively) be this SC instruction and let $LL_{2j-1}^2$ ($LL_{2j}^2$, respectively) be its matching LL instruction. Denote by $C_{2j,2}^i$ the configuration just after execution of line 2 by op$^j_i$ and denote by $C_{2j,4}^i$ the configuration just after the execution of line 4 by op$^j_i$.

Lines 2 – 4 and 10 of the pseudocode imply the following two observations.

**Observation C.3** If the execution of a getSet $G$ (line 10) by some process $p_i$ includes some process $p_l$ in its returned set, then there is some operation op$^j_i$ such that $G$ is executed between $C_{2j,2}^i$ and $C_{2j,4}^i$.

**Observation C.4** If the execution of a getSet $G$ by some process $p_l$ does not include some process $p_i$ in its returned set, then either $G$ is executed before $C_{1,2}^i$ or there is an operation op$^j_i$, $j > 0$, such that $G$ is executed between $C_{2j-1,4}^i$ and $C_{2j,2}^i$.
Lines 2 – 4 and 6 of the pseudocode imply the following observation.

**Observation C.5** (1) Configuration $C_{j,2}^i$ precedes the beginning of $\pi_{2j-1}^i$, and (2) configuration $C_{j,4}^i$ follows the end of $\pi_{2j-1}^i$ and precedes the beginning of $\pi_{2j}^i$.

**Lemma C.6** Consider any $j \geq 1$. Let $\pi_h$ and $\pi_h'$ be the attempt functions that execute $SC_{2j-1}^i$ and $SC_{2j}^i$, respectively.

1. $\pi_h$ executes its getSet $G_h$ after $C_{j,2}^i$

2. $\pi_h'$ executes its getSet $G_h'$ after $C_{j,4}^i$

**Proof:** We first prove Claim 1. By the definition of $SC_{2j-1}^i$, it follows that $sp.applied[i]$ changes from false to true at $C_{2j-1}^i$. Thus, by the pseudocode (lines 2, 10, 15), it follows that $G_h$ includes $p_i$ in its returned set. By Observation C.3, there is some $j' \leq j$ such that $G_h$, is executed between $C_{j',2}^i$ and $C_{j',4}^i$. We prove that $j' = j$. Assume, by the way of contradiction, that $j' < j$. By Observation C.5, operation $G_h$ precedes $\pi_{2j'}^i$. By Observation C.1, there are at least two successful SC instructions in the execution interval of $\pi_{2j'}^i$. Since $SC_{2j'-1}^i$ is executed in the execution interval of $\pi_{2j-1}^i$ and $LL_{2j-1}^i$ precedes $G_h$, $SC_{2j-1}^i$ cannot be successful. This contradicts the definition of $SC_{2j-1}^i$. Thus, $j' = j$ which implies that $G_h$ is executed after $C_{j,2}^i$, as needed.

We now prove Claim 2. By the definition of $SC_{2j}^i$, it follows that $sp.applied[i]$ changes from true to false at $C_{2j}^i$. Thus, by the pseudocode (lines 4, 10, 15), it follows that $G_h'$ does not include $p_i$ in its returned set. By Observation C.4, either $G_h'$ is executed by $\pi_{h'}$ before $C_{j,2}^i$ or there is some $op_{j'}^i$ such that $G_h'$ is executed between $C_{j'-1,4}^i$ and $C_{j',2}^i$. In any of these cases, $G_h'$ is executed before $C_{j,2}^i$. Therefore $LL_{2j}^i$ is also executed before $C_{j,2}^i$. Since $SC_{2j-1}^i$ follows $C_{j,2}^i$ and $SC_{2j}^i$ follows $SC_{2j-1}^i$, it follows that $SC_{2j}^i$ cannot be successful, which contradicts the definition of $SC_{2j}^i$. Thus, $SC_{2j}^i$ should have occurred between $C_{j,4}^i$ and $C_{j+1,2}^i$.

**Lemma C.7** $C_{j,2}^i$ precedes $C_{2j-1}^i$.

**Proof:** Assume that $SC_{2j-1}^i$ is executed by some attempt $\pi_h$. By Lemma C.6 (claim 1), it follows that the getSet by $\pi_h$ is executed after $C_{j,2}^i$. By the pseudocode (lines 10, 17), $\pi_h$ executes $SC_{2j-1}^i$ after its getSet instruction. Therefore, $C_{j,2}^i$ precedes $C_{2j-1}^i$, as needed.

**Lemma C.8** $C_{j,4}^i$ precedes $C_{2j}^i$.

**Proof:** Assume that $SC_{2j}^i$ is executed by some attempt $\pi_h$. By Lemma C.6 (claim 2), it follows that the getSet by $\pi_h$ is executed after $C_{j,4}^i$. By the pseudocode (lines 10, 17), $\pi_h$ executes $SC_{2j}^i$ after its getSet instruction. Therefore, $C_{j,4}^i$ precedes $C_{2j}^i$, as needed.

**Lemma C.9** There is no configuration $C$ such that $C$ is between $C_{2j}^i$ and $C_{2j+1}^i$, and $sp.applied[i]$ is equal to true at $C$. 


Proof: Assume, by the way of contradiction, that there is some configuration between \( C_{2j} \) and \( C_{2j+1} \) such that \( sp.applied[i] \) is equal to true. Let \( C_h \) be the first of these configurations. Since only \( SC \) instructions change the value of register \( sp \), there is a successful \( SC \) instruction \( SC_h \) which occurs just before \( C_h \) and writes true to \( sp.applied[i] \). Let \( LL_h \) be the matching \( LL \) instruction to \( SC_h \) and let \( \pi_h \) be the instance of attempt that executes \( SC_h \). Since \( \pi_h \) writes true into \( sp.applied[i] \), its \( GETSET \) \( G_h \) includes \( p_i \) in its returned set of processes. Observation C.3 implies that there is some \( j' \leq j \) such that \( G_{h'} \) is executed between \( C_{j',2} \) and \( C_{j',4} \). Since \( j' \leq j \), \( C_{j',4} \) either precedes \( C_{j,4} \) or \( C_{j',4} = C_{j,4} \). Since \( C_{j,4} \) precedes \( C_{2j} \) where \( SC_{2j} \) is executed successfully, it follows that \( SC_h \) is not successful. This contradicts the definition of \( SC_h \). 

We say that an operation \( op \) by some process \( p_i \) is applied on the simulated object if (1) procedure \( GETSET \), executed by some operation \( op' \) (that might be \( op \) or any other operation), includes \( p_i \) in the set of processes it returns, (2) procedure attempt, executed by \( op' \) reads in component \( A_i \) of snapshot \( S \), the operation type written there by \( p_i \) for \( op \) and considers it as the new operation type for \( p_i \), (2) attempt by \( op' \) calls apply for \( op \), and (2) the execution of the \( SC \) of line 17 (let it be \( SC_r \)) on \( sp \) by \( op' \) succeeds. When these conditions are satisfied, we sometimes also say that \( op' \) applies \( op \) on the simulated object or that \( SC_r \) applies \( op \) on the simulated object. We next prove that each operation \( op \) is applied on the simulated object exactly once.

**Lemma C.10** For each \( j \geq 1 \), operation \( op_j^i \) is applied to the simulated object at configuration \( C_{2j-1}^i \).

**Proof:** By the pseudocode (lines 13, 14), an operation for process \( p_i \) is applied on the simulated object each time \( sp.applied[i] \) changes from false to true. By the definition of configuration \( C_{2j-1}^i \), it follows that some operation is applied in it. Let \( op' \) be this operation, and assume, by the way of contradiction, that \( op' \neq op_j^i \). Let \( \pi_h \) be the attempt that executes \( SC_{2j-1}^i \). Lemma C.6 implies that \( \pi_h \) executes its \( GETSET \) \( G_h \) after \( C_{j,2}^i \). By the pseudocode, \( \pi_h \) calls its \( SCAN \) after \( G_h \), thus the \( SCAN \) by \( \pi_h \) is executed between \( C_{j,2}^i \) and \( C_{j-1}^i \). Since \( op_j^i \) calls its \( UPDATE \) before \( C_{j,2}^i \) and stores \( \bot \) in its component after \( C_{j,4}^i \) which follows \( C_{2j-1}^i \), the \( SCAN \) by \( \pi_h \) returns \( op_j^i \). Thus, \( \pi_h \) applies \( op_j^i \) as the operation of \( p_i \) in the simulated object.

The following corollary is an immediate consequence of Lemmas C.9 and C.10, and of the definitions of \( C_{2j-1}^i \) and \( C_{2j}^i \).

**Corollary C.11** Each operation \( op \) is applied exactly once.

We are now ready to assign linearization points. Let \( a \) be any execution. For each \( i \in \{1, ..., n\} \) and \( j \geq 1 \), we place the linearization point of \( op_j^i \) at \( C_{2j-1}^i \); ties are broken by the order imposed by process identifiers.

**Lemma C.12** Each operation \( op_j^i \) is linearized within its execution interval.

**Proof:** Lemma C.7 implies that \( C_{j,2}^i \) precedes \( C_{2j-1}^i \). By its definition, \( C_{2j-1}^i \) precedes the end of \( \pi_{2j-1}^i \). Thus, \( C_{2j-1}^i \) is in the execution interval of \( op_j^i \), as needed.
In order to prove consistency, we introduce the following notation. Denote by $SC_i$ the $i$-th successful $SC$ instruction on register $sp$. Obviously, between $SC_i$ and $SC_{i+1}$, the $st$ field of register $sp$ is not modified.

Let $a$ be any execution of the algorithm. Denote by $a_i$, the prefix of $a$ which ends at $SC_i$. Let $a_0$ be the empty execution. Denote by $l_i$ be the linearization order of the operations of $a_i$.

**Lemma C.13** For each $i \geq 0$, $a_i$ is consistent.

**Proof:** We prove the claim by induction on $i$.

**Base case (i=0):** Execution $a_0$ is empty, so the claim holds trivially.

**Induction hypothesis:** Fix any $i > 0$ and assume that the claim holds for $i - 1$.

**Induction step:** We prove that the claim holds for $i$. From the induction hypothesis, it holds that $a_{i-1}$ is consistent, and $l_{i-1}$ denotes the linearization order of its operations. Let $op$ be the operation that executes $SC_i$ and assume that $op$ applies $j > 0$ operations on the simulated object. Denote by $op_1, ..., op_j$ the sequence of these operations ordered with respect to the identifiers of the processes that initiate them.

We prove the following claim: For each $l$, $0 \leq l \leq j$, operation $op_l$ returns a consistent response. The claim is proved by induction on $l$.

**Base case (l=0):** The claim holds vacuously.

**Induction hypothesis:** Fix any $l > 0$ and assume that the claim holds for $l - 1$.

**Induction step:** We prove that the claim holds for $l$. By the induction hypotheses (inner and outer inductions), it follows that all operations in $l_{i-1}op_1, ..., op_{l-1}$ return a consistent response. By Observation C.11, $op_l$ is applied exactly once. By the way function attempt operates, it applies $op_l$ on the simulated object just after it has applied $op_1, ..., op_{l-1}$ on it. Therefore, the response calculated for $op_l$ by $op$ is consistent, as needed.