Atomic Snapshots from Small Registers

Leqi Zhu and Faith Ellen

University of Toronto, Toronto, ON

Abstract

Existing $n$-process implementations of atomic snapshots from registers use large registers. We consider the problem of implementing an $m$-component snapshot from small, $\Theta(\log n)$-bit registers. A natural solution is to consider simulating the large registers. Doing so straightforwardly can significantly increase the step complexity. We introduce the notion of an interruptible read and show how it can reduce the step complexity of simulating the large registers in the snapshot of Afek et al. [1]. In particular, we show how to modify a recent large register simulation [2] to support interruptible reads. Using this modified simulation, the step complexity of UPDATE and SCAN changes from $\Theta(nm)$ to $\Theta(nm + mw)$, instead of $\Theta(nmw)$, if each component of the snapshot consists of $\Theta(w \log n)$ bits. We also show how to modify a limited-use snapshot [4] to use small registers when the number of UPDATE operations is in $n^{O(1)}$. In this case, we change the step complexity of UPDATE from $\Theta((\log n)^3)$ to $O(w + (\log n)^2 \log m)$ and the step complexity of SCAN from $\Theta(\log n)$ to $O(mw + \log n)$.

1998 ACM Subject Classification

E.1 Distributed data structures

Keywords and phrases
atomic snapshot, limited-use snapshot, small registers, simulation

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Atomic snapshots give processes the ability to obtain a consistent view of shared memory through a SCAN operation, even when other processes are concurrently performing UPDATE operations to the memory. This allows programmers to reason about the concurrency in the system in a higher-level manner and can greatly simplify development and verification of concurrent programs. In their seminal paper on atomic snapshots, Afek et al. [1] cite many applications. In the same paper, they presented an $n$-process, $m$-component snapshot implementation from registers (i.e., using only READ and WRITE) with $\Theta(nm)$ step complexity for both SCAN and UPDATE.

A well-known concern with this implementation, and indeed all known snapshot implementations from registers, is the assumption that the system provides registers large enough to store the result of a SCAN. As the number of components or the size of each component of the snapshot grows, this assumption becomes less and less practical. We consider the problem of implementing snapshots shared by $n$ processes from $\Theta(\log n)$-bit registers. We call such registers words. It is customary to use registers with $\Omega(\log n)$ bits, so they can store process identifiers.

A natural solution is to consider simulating the large registers. Recently, Aghazadeh, Golab, and Woelfel [2] showed how to simulate a $\Theta(w \log n)$-bit register from words with optimal step complexity, $\Theta(w)$, for READ and WRITE. Straightforwardly applying their register simulation to the snapshot implementation by Afek et al. significantly increases the step complexity from $\Theta(nm)$ to $\Theta(nmw)$, if each component of the snapshot consists of $\Theta(w \log n)$ bits.
Most of these extra steps come from (unnecessarily) reading the embedded scans and value fields contained in each of the large registers during the double-collects. However, it is possible to determine if the embedded scans and values are needed by reading only $\Theta(\log n)$ bits from each of the registers. Motivated by this, we introduce the notion of an interruptible read. This means that a process can read part of the large register’s data, pause the simulated READ to perform other operations, and then return to read more of the data, so that the entire read appears to occur at the same time. The large register simulation of Aghazadeh, Golab, and Woelfel can be modified to have interruptible reads. Applying this modified simulation to the snapshot of Afek et al. reduces the step complexity of UPDATE and SCAN from $\Theta(nmw)$ to $\Theta(nm + mw)$.

Recently, Aspnes, Attiya, Censor-Hillel, and Ellen [4] showed that, if the number of UPDATE operations, $b$, is in $n^{O(1)}$, then it is possible to implement a $b$-limited-use $m$-component snapshot from $\Theta(nmw \log n)$-bit registers with step complexity $\Theta((\log n)^3)$ for UPDATE and step complexity $\Theta(\log n)$ for SCAN. Simulating the large registers from words increases the step complexity of UPDATE to $\Theta(n^2 mw)$ and the step complexity of SCAN to $\Theta(nmw)$, even with interruptible reads.

We show how to directly modify their implementation to use $\Theta(\log n)$-bit registers while only slightly increasing the step complexity: $O(w + (\log n)^2 \log m)$ for UPDATE and $O(mw + \log n)$ for SCAN. The idea is similar to interruptible reads. Instead of directly returning a view, a SCAN returns an index into a sequence of views, whose length is proportional to the number of UPDATE operations that have been performed. We call this an implicit SCAN. The view may then be examined by using this index as input to a VIEW operation. We call snapshots implemented in this way implicit. We provide a simple recursive construction of an implicit snapshot from two implicit snapshots with fewer components. Instead of representing a view directly, we represent it implicitly, by a pair of indices into the sequences of views of these two smaller implicit snapshots. The actual view can be recovered recursively. This also allows us to also implement a partial scan [7], in which only $c$ of the components are queried, with step complexity $O(c(w + \log m) + \log n)$.

## 2 Model and Preliminaries

We consider an asynchronous shared memory system with $n$ processes which communicate using shared $\Theta(\log n)$-bit (multi-writer) registers. We call these registers words. We assume that processes may fail at any time by crashing.

An execution in this system is an alternating sequence of configurations and events $C_0, e_1, C_1, \ldots$. Each event $e_i$ (or, step) is either a READ or WRITE operation on a shared register and each configuration $C_i$ consists of the contents of every register and the state of each process after event $e_i$ is applied to configuration $C_{i-1}$. For any two events $a$ and $b$ in an execution, we write $a \rightarrow b$ to mean that $a$ precedes $b$ in the execution.

An implementation of a shared object in this system provides a representation of the object using words and an algorithm for each type of operation supported by the object and for each process sharing the object. We only consider wait-free implementations, where each operation invoked by a non-faulty process is guaranteed to be completed within a finite number of its own steps. The step complexity of an operation $O$ in an implementation is the maximum, over all possible executions, of the number of steps taken by any process to finish an instance of $O$ that it invoked.

Given an execution, the execution interval of an operation is the portion of the execution which begins with the first step in the operation and ends with the last step in the
operation. An implementation is linearizable [8] if, for every execution, we can choose a linearization point in the execution interval of each operation such that operations appear to occur instantaneously at their linearization points.

An $m$-component atomic snapshot (or simply snapshot) has two operations, SCAN and UPDATE$(j, v)$. The SCAN operation returns an instantaneous view of the components, as if all $m$ components were read in a single atomic step. The UPDATE$(j, v)$ operation updates component $j$ to have value $v$ and returns nothing. For $b \geq 1$, we say a snapshot is $b$-limited-use if it supports at most $b$ UPDATE operations in any execution.

A max register has two operations, READ-MAX and WRITE-MAX$(v)$. The READ-MAX operation returns the largest value written thus far, while WRITE-MAX$(v)$ adds a number $v$ to the set of values written and returns nothing. For $b \geq 1$, we say a max register is $b$-bounded if its values are restricted to $\{0, \ldots, b-1\}$. Aspnes, Attiya, and Censor-Hillel [3] showed that:

\begin{itemize}
  \item \textbf{Theorem 1.} There is a $b$-bounded max register implementation with step complexity $\Theta(\log b)$ which uses only 1-bit registers.
\end{itemize}

A 2-component max array has two operations, MAX-SCAN and MAX-UPDATE$(j, v)$. Each component behaves like a max register. The MAX-SCAN operation returns an instantaneous view of the two components, as if both components had READ-MAX performed on them in a single atomic step. The MAX-UPDATE$(j, v)$ operation updates component $j$ as if it had performed WRITE-MAX$(v)$ to it and returns nothing. For $b_1, b_2 \geq 1$, we say a 2-component max array is $(b_1, b_2)$-bounded if the values of its first component are restricted to $\{0, \ldots, b_1-1\}$ and the values of its second component are restricted to $\{0, \ldots, b_2-1\}$. Aspnes, Attiya, Censor-Hillel, and Ellen [4] showed that:

\begin{itemize}
  \item \textbf{Theorem 2.} There is a $(b_1, b_2)$-bounded 2-component max array implementation with step complexity $\Theta(\log b_1 \log b_2)$ which uses only 1-bit registers.
\end{itemize}

### 3 Unlimited-use snapshot from small registers

We show how to obtain an $m$-component snapshot implementation from words with step complexity $\Theta(nm+mw)$ for SCAN and UPDATE, if each component of the snapshot consists of $\Theta(w \log n)$ bits.

Our approach is to simulate the large registers in the $m$-component snapshot by Afek et al. [1] from words. In their implementation, a SCAN performs $\Theta(n)$ collects on an array of $m$ registers, each containing $\Theta(w \log n)$ bits. The large register simulation by Aghazadeh, Golab, and Woelfel [2] has $\Theta(w)$ step complexity for READ and WRITE of a $\Theta(w \log n)$-bit register. Thus, if we directly apply their simulation, the step complexity of SCAN and UPDATE (which contains an embedded SCAN) becomes $\Theta(nmw)$ instead of $\Theta(nm)$.

We observe that not all the bits read during the collects are needed. Indeed, in all but the last collect, only $\Theta(\log n)$ bits from each of the $m$ registers end up being used. Furthermore, after reading only these bits, it is possible to determine which additional bits need to be read. Motivated by this, we introduce the notion of an interruptible read which, intuitively, allows a process to reserve a copy of the value in the large register (BEGIN-IREAD), read particular words in the copy (READ-WORD), and then return the memory for reuse (END-IREAD). In general, this is useful for algorithms in which a process can read only a small fraction of the bits in a large register to determine if it needs to read the rest of the bits.

In Section 3.1, we formally define an interruptible read. In Section 3.2, we explain how to modify the simulation by Aghazadeh et al. to implement BEGIN-IREAD, READ-
4 Atomic Snapshots from Small Registers

WORD, and END-IREAD in constant time. In Section 3.3, we carefully apply this modified simulation with interruptible reads to the standard snapshot of Afek et al. to obtain an m-component snapshot with SCAN and UPDATE step complexity \( \Theta(nm + mw) \). Finally, in Section 3.4, we describe some other, faster snapshot implementations and why it is difficult to modify them to use words while maintaining their step complexity.

3.1 Interruptible reads

Formally, a simulation of a \( \Theta(w \log n) \)-bit register from words supports interruptible reads if it implements 3 operations: BEGIN-IREAD\(_p\), READ-WORD\(_p\), and END-IREAD\(_p\), for each process \( p \). BEGIN-IREAD\(_p\) takes no arguments and returns a pointer to a block of \( w \) words which represents the current value of the large register. END-IREAD\(_p\) takes a pointer returned by a BEGIN-IREAD\(_p\) operation and returns nothing. READ-WORD\(_p\) takes a pointer \( t \) returned by a BEGIN-IREAD\(_p\) operation and an integer \( j \). It returns the value of the \( j \)'th word in the block of memory pointed to by \( t \). Between a BEGIN-IREAD\(_p\) that returns a pointer \( t \) and the next occurrence of END-IREAD\(_p\)(\( t \)), the memory pointed to by \( t \) is not changed. We say that a process \( p \) has an active interruptible read at the end of an execution if the execution contains a BEGIN-IREAD\(_p\) operation that returns some pointer \( t \) which is not followed by a corresponding END-IREAD\(_p\)(\( t \)) operation.

For example, we can use interruptible reads to implement a normal READ by obtaining a pointer \( t \) via BEGIN-IREAD\(_p\), concatenating the values returned by READ-WORD\(_p\)(\( t, j \)) for \( j = 1, \ldots, w \) into a single value \( v \), releasing the memory pointed to by \( t \) via END-IREAD\(_p\)(\( t \)), and then returning \( v \).

Figure 1 gives an example of an execution involving 2 processes, \( p_1 \) and \( p_2 \), using interruptible reads. It features a 2-bit register being simulated by 1-bit registers. At the start, \( p_1 \) writes 00, denoted by W(00). Then \( p_2 \) begins an interruptible read and obtain a pointer \( t_1 \). This is denoted by BI = \( t_1 \). Next, \( p_2 \) reads the first word (in this case, a bit) being pointed to by \( t_1 \), which has value 0. We denote this by RW(\( t_1, 1 \)) = 0. Now \( p_1 \) writes 11. Then \( p_2 \) begins another interruptible read to obtain a pointer \( t_2 \) and reads the first word being pointed to by \( t_2 \), which has value 1. Finally, when \( p_2 \) reads the second word pointed to by \( t_1 \), it is still 0. At the end of this execution, \( p_2 \) has 2 active interruptible reads.

\[
\begin{array}{c|c|c|c|c|c}
\hline
p_1 & W(00) & W(11) \\
\hline
p_2 & BI = t_1 & RW(t_1, 1) = 0 & BI = t_2 & RW(t_2, 1) = 1 & RW(t_2, 0) = 0 \\
\hline
\end{array}
\]

Figure 1 Example of an execution using interruptible reads.

3.2 A large register simulation supporting fast interruptible reads

Not all large register simulations support fast interruptible reads. For instance, Peterson [10] showed how to simulate a large single-writer \( \Theta(w \log n) \)-bit register from single-writer \( \Theta(\log n) \)-bit registers. His simulation represents a large register by collections of \( \Theta(w) \) words, called buffers. The writer alternately writes to two of these buffers. A switch bit indicates which of these two buffers was most recently written to. The writer flips the switch bit after completing a sequence of \( \Theta(w) \) writes to one of these buffers. Ideally, the readers would read from one buffer while the writer writes to the other. However, processes may fall asleep for a long time. Using handshakes, the writer can detect if a reader is concurrent with its WRITE. In this case, it also writes the current value of the register to the reader’s designated copy buffer. The reader performs collects on both of the main buffers as well
as its own copy buffer. By checking the switch and handshake bits, the reader returns the value of a buffer that was not being written to while it was being read. Implementing fast interruptible reads is difficult in this case because the reader cannot quickly determine which pointer \textsc{BEGIN-IREAD} should return.

The simulation by Aghazadeh et al. \cite{2} can easily be modified to support a polynomial (in \(n\)) number of active interruptible reads per process. Like Peterson's simulation, they use buffers. Writers have a pool of buffers to which they may write and there is a pointer to the most recently written buffer. To \textsc{READ}, a reader reads the pointer to the most recently written buffer and announces this pointer. The algorithm guarantees that an announced buffer will not be modified by any writer. Since a writer may miss this announcement, there is also a mechanism for a writer to pass hints to the reader about alternate buffers which have been written to in the meantime, from which it is safe to read. These hints can be read by the reader in a constant number of steps. The algorithm guarantees that, until the reader acknowledges a hint, no writer is allowed to modify the buffers mentioned in the hint. The reader acknowledges a hint when it will no longer read from the buffer to which it points. This takes a constant number of steps. It is possible, but not necessary, for the reader to clear its initial announcement.

To implement interruptible reads, we break this \textsc{READ} operation into pieces. In particular, \textsc{BEGIN-IREAD} is the portion of the \textsc{READ} that determines the buffer to be read. It returns a pointer to that buffer. \textsc{READ-WORD} simply reads the appropriate word from the buffer. By the correctness of the simulation, the words may be read in any order. Finally, \textsc{END-IREAD} is be the portion of the \textsc{READ} that acknowledges hints. Note that each of these operations take a constant number of steps.

This simulation assumes that each process can only have one operation active at a time. To support \(c \in O(1)\) active interruptible reads of the same large register per process, we need to increase the size of the buffer pool for each writer by a factor of \(c\). Then the size of each pointer increases by \(\lceil \log_2 c \rceil = \Theta(\log n)\) bits.

3.3 Application to Afek et al.

We consider the \(m\)-component snapshot implementation of Afek et al. \cite{1}. Suppose that each component of the snapshot consists of \(\Theta(w \log n)\) bits. The implementation uses binary registers, \(q_{i,j}\) and \(q'_{i,j}\), for \(i,j \in \{0,1,\ldots,n-1\}\), which are the handshake bits, \(\Theta(mw \log n)\)-bit registers, \(view_1,\ldots,view_n\), which store views, and \(\Theta(w \log n)\)-bit registers, \(R_1,\ldots,R_m\), each of which stores the current value of the component, a process identifier, and a toggle bit.

A \textsc{SCAN} operation by process \(p_i\) consists of a loop, each iteration of which (1) collects \(q_{1,i},\ldots,q_{n,i}\), (2) writes to \(q'_{1,i},\ldots,q'_{n,i}\), (3) collects \(R_1,\ldots,R_m\) twice, and (4) collects \(q_{1,i},\ldots,q_{n,i}\) again. The iteration is successful if the handshake bits read in both collects of \(q_{1,i},\ldots,q_{n,i}\) are the same and the process identifiers and toggle bits read in both collects of \(R_1,\ldots,R_m\) are the same. In this case, (5) the current value of the components read from the second collect of \(R_1,\ldots,R_m\) are returned. Otherwise, the algorithms (6) checks (by examining the previously read handshake bits, process identifiers, and toggle bits) if some process \(p_j\) has performed at least one complete \textsc{UPDATE} since the start of the \textsc{SCAN} and, if so, (7) reads and returns \(view_j\). They prove that there can be at most \(O(n)\) iterations of the loop. An \textsc{UPDATE} operation by process \(p_i\) consists of a collect of \(q'_{1,i},\ldots,q'_{n,i}\), writes to \(q_{j,i}\), for \(j \in \{0,1,\ldots,n-1\}\), an embedded \textsc{SCAN}, a write to \(view_i\), and a write to \(R_c\), for some \(c \in \{1,\ldots,m\}\).

Steps (1), (2), and (4) have \(\Theta(n)\) step complexity. If we directly apply the simulation of
Aghazadeh et al. [2], the step complexity of step (3) will be \(\Theta(mw)\) and the step complexity of the SCAN is \(\Theta(nmw)\). We can improve this to \(\Theta(m)\) using interruptible reads by reading only the words containing the process identifiers and toggle bits from \(R_1, \ldots, R_m\) in both collects. We can end the interruptible reads started during the first collect immediately after the first collect is finished. If the iteration is successful, then we read the current value of each component and end each interruptible read started during the second collect. If the iteration is unsuccessful, then we end each interruptible read started during the second collect without reading the current value of each component. We note that each process has at most one active interruptible read on each register at any time. Steps (5) and (7) have step complexity \(\Theta(mw)\). Step (6) consists of only local operations. Since steps (5) and (7) are performed just before returning, they contribute \(\Theta(mw)\) steps to the total. Steps (1) to (4) occur \(\Theta(n)\) times in the worst case. Overall, the step complexity of SCAN is \(\Theta(nm + mw)\).

**Theorem 3.** For \(k \in \Omega(\log n)\), there is an \(m\)-component snapshot implementation from \(k\)-bit registers with step complexity \(\Theta(nm + mw)\) for SCAN and UPDATE, if each component of the snapshot consists of \(\Theta(\log n)\) bits.

### 3.4 Other snapshots

There are snapshots with better step complexity. Attiya and Rachman [5] implemented a single-writer snapshot from single-writer registers with \(\Theta(n \log n)\) step complexity. However, they perform too many reads on large registers. In particular, they perform \(\Theta(n \log n)\) reads of \(\Theta(nw \log n)\)-bit registers, if each component of the snapshot consists of \(\Theta(w \log n)\) bits. Moreover, they always use the entire value obtained from each read, so interruptible reads are not helpful. This results in a step complexity of \(\Omega(n^2 \log n)\) using any large register simulation. Inoue and Chen [9] showed how to implement lattice agreement from multi-writer registers with \(\Theta(n)\) step complexity. Attiya, Herlihy, and Rachman showed how to implement a single-writer snapshot from lattice agreement [6]. This implies an implementation of a single-writer snapshot from multi-writer registers with \(\Theta(n)\) step complexity. Unfortunately, the implementation of Attiya, Herlihy, and Rachman uses unbounded size registers. It is unclear whether it is possible to modify these implementations to use small registers while maintaining their step complexity.

### 4 Limited-use snapshot from small registers

An *implicit snapshot* object is like a regular snapshot object except that a SCAN operation is separated into two parts, an ISCAN operation and a VIEW operation. Intuitively, the ISCAN operation is where the actual SCAN occurs. It returns a pointer to a view, which may then be read via the VIEW operation. To facilitate our implementation of a *partial* SCAN [7], a VIEW operation takes a range of components as input and returns the values of the components in that range. We formalize this as follows.

An ISCAN operation \(S\) returns a value \(t(S) \geq 0\), which we call the *index* of \(S\). We require that, for any two ISCAN operations \(S_1\) and \(S_2\), \(t(S_1) < t(S_2)\) if and only if \(S_1\) is linearized before \(S_2\) and there is at least one UPDATE linearized between them. We also require that an ISCAN operation \(S\) has \(t(S) = 0\) if and only if no UPDATE operation is linearized before it.

Given an ISCAN operation \(S\), we define the *view at index* \(t(S)\) to be the \(m\)-component vector whose \(c\)'th component contains the value of the last UPDATE operation to component
A 1-component limited-use implicit snapshot implementation

We can implement a \( b \)-limited-use 1-component single-writer implicit snapshot using an array \( A \) of \( b \) single-writer registers and a single-writer register \( \text{index} \). To UPDATE component 1 to \( v \), the writer increments a local counter \( t \), writes \( v \) to \( A[t] \), and then writes \( t \) to \( \text{index} \). An ISCAN reads \( \text{index} \) and returns it. \( \text{VIEW}(t, 1, 1, V) \) reads \( A[t] \) and writes this value to \( V[1] \).

To extend this implementation to multiple writers, we change \( \text{index} \) to be a \((bn+1)\)-bounded max register. Furthermore, instead of incrementing a local counter, a writer performs \( \text{index}.\text{read-max} \) to determine the current index \( t \), chooses an index \( t' > t \) that no other writer will choose, writes the value \( v \) to \( A[t'] \), and then performs \( \text{index}.\text{write-max}(t') \). An ISCAN consists of performing \( \text{index}.\text{read-max} \) and returning the resulting value. The \( \text{VIEW} \) operation is unchanged. Always choosing \( t' \) to be the smallest integer larger than \( t \) which is congruent to the writer’s process identifier modulo \( n \) ensures that different indices are chosen by different writers and they are all bounded above by \( bn \). See Algorithm 1 for the pseudocode.

Algorithm 1 A \( b \)-limited-use 1-component implicit snapshot object.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure UPDATE(1, u)</td>
</tr>
<tr>
<td>2</td>
<td>( t \leftarrow \text{index}.\text{read-max}() )</td>
</tr>
<tr>
<td>3</td>
<td>( t' \leftarrow \min{j : j &gt; t \text{ and } j \equiv i \mod n} ) &gt; code for process ( p_i )</td>
</tr>
<tr>
<td>4</td>
<td>( A[t'].\text{write}(u) )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{index}.\text{write-max}(t') )</td>
</tr>
<tr>
<td>6</td>
<td>procedure ISCAN</td>
</tr>
<tr>
<td>7</td>
<td>return ( \text{index}.\text{read-max}() )</td>
</tr>
<tr>
<td>8</td>
<td>procedure VIEW(t, 1, 1, V[1..1])</td>
</tr>
<tr>
<td>9</td>
<td>( V[1] \leftarrow A[t].\text{read()} )</td>
</tr>
</tbody>
</table>

Recall that, from Theorem 1, there is a linearizable implementation of a bounded max register from binary registers. Thus, we will assume that all operations on \( \text{index} \) are atomic and treat them as steps in the executions we consider.

For every UPDATE operation \( U \), let \( t(U) \) be the value \( t' \) used as the argument of the \( \text{index}.\text{write-max} \) that \( U \) performed on line 5.
Lemma 4. For all $t \geq 1$, there is at most one UPDATE operation $U$ with $t(U) = t$. No UPDATE operation $U$ has $t(U) = 0$.

Proof. By line 3, if $t \equiv i \mod n$, then only process $p_i$ can choose $t$. Since $p_i$ performs $\text{index.write-max}(t)$ on line 5, the values returned to $p_i$ from $\text{index.read-max}$ on line 2 are strictly increasing. Therefore, $p_i$ will never choose $t$ again. Since $\text{index}$ is initially 0, $t(U) > 0$ for all UPDATE operations $U$. \hfill \square

An UPDATE operation $U$ is linearized at the first point that $\text{index}$ has value at least $t(U)$. This occurs when some process, not necessarily the process performing $U$, performs $\text{index.write-max}(t)$, for some $t \geq t(U)$. If multiple UPDATE operations are linearized at the same point, then they are linearized in increasing order of their indices. By Lemma 4, there will be no ties. An ISCAN operation is linearized when it performs $\text{index.read-max}$. A VIEW operation is linearized when it performs $\text{A[t].read}$.

Since an UPDATE operation $U$ begins with an $\text{index.read-max}$ and $t(U)$ is chosen to be larger than the value it returns, $\text{index} < t(U)$ at the beginning of $U$’s execution interval. Furthermore, since $U$ ends with $\text{index.write-max}(t(U))$, $\text{index} \geq t(U)$ at the end of $U$’s execution interval. Since the value of a max-register is non-decreasing, it follows that each UPDATE operation is linearized at a point within its execution interval.

Lemma 5. If an UPDATE operation $U_1$ is linearized before another UPDATE operation $U_2$, then $t(U_1) < t(U_2)$.

Proof. Let $X$ be the $\text{index.write-max}(t)$ step at which $U_1$ is linearized, so that $t \geq t(U_1)$. If the $\text{index.read-max}$ step of $U_2$ occurs after $X$, then the value returned by the $\text{index.read-max}$ step of $U_2$ is at least $t$, so $t(U_2) > t \geq t(U_1)$. So, suppose that the $\text{index.read-max}$ of $U_2$ occurs before $X$. If $U_2$ is also linearized at $X$, then, from the way the linearization order is defined when multiple UPDATE operations are linearized at the same step, $t(U_2) > t(U_1)$. Otherwise, $U_2$ is linearized after $X$ so, by definition, $t(U_2) > t \geq t(U_1)$. \hfill \square

Lemma 6. Let $S$ be an ISCAN operation. If no UPDATE operation is linearized before $S$, then $t(S) = 0$. Otherwise, if $U$ is the last UPDATE operation linearized before $S$, then $t(U) = t(S)$.

Proof. Every UPDATE operation is linearized at no later than its $\text{index.write-max}$ step. If $S$ is linearized before every UPDATE operation, then its $\text{index.read-max}$ step occurs before any $\text{index.write-max}$ step and, hence, returns 0. So, suppose some UPDATE operation is linearized before $S$ and let $U$ be the last such UPDATE operation. Since $S$ obtained index $t(S)$ from its $\text{index.read-max}$ step, there was an UPDATE operation $U’$ that previously performed $\text{index.write-max}(t(U’))$ with $t(U’) = t(S)$. $U’$ is linearized at no later than its $\text{index.write-max}$ step. Hence, it is linearized before $S$. Suppose, for a contradiction, that $U \neq U’$, so that $U$ is linearized between $U’$ and $S$. By Lemma 5, $t(U) > t(U’).$ By definition, there was a $\text{index.write-max}(t)$ with $t \geq t(U)$ at the linearization point of $U$. This implies that $t(S) \geq t(U)$. This is a contradiction, since $t(U) > t(U’) = t(S)$. \hfill \square

Lemma 7. An UPDATE operation $U$ is linearized before an ISCAN operation $S$ if and only if $t(U) \leq t(S)$.

Proof. Suppose an UPDATE operation $U$ is linearized before an ISCAN $S$. Let $U’$ be the last UPDATE operation linearized before $S$. By Lemma 6, $t(U’) = t(S)$. If $U = U’$, then $t(U) = t(S)$. Otherwise, by Lemma 5, $t(U) < t(U’) = t(S)$. Conversely, suppose that $U$ is linearized after $S$. If no UPDATE operation is linearized before $S$, then $t(S) = 0$ and, by
Lemma 4, \( t(U) > t(S) \). So, suppose that some UPDATE operation is linearized before \( S \) and let \( U' \) be the last such UPDATE operation. Since \( U \) is linearized after \( U' \), by Lemma 5, \( t(U) > t(U') = t(S) \).

Lemma 8. For any two ISCAN operations \( S_1 \) and \( S_2 \), \( t(S_1) < t(S_2) \) if and only if \( S_1 \) is linearized before \( S_2 \) and there is at least one UPDATE linearized between them.

Proof. If \( t(S_1) < t(S_2) \), then \( S_1 \) is linearized before \( S_2 \), since they are linearized at their respective index.read-max steps. By Lemma 4, there is an unique UPDATE operation \( U \) with \( t(U) = t(S_2) \). Since \( t(U) > t(S_1) \), Lemma 7 implies that \( U \) is linearized after \( S_1 \). Since \( U \) is the only UPDATE operation with \( t(U) = t(S_2) \), the index.write-max step in \( U \) occurs before the index.read-max step in \( S_2 \). Since \( U \) is linearized at no later than its index.write-max step, it follows that \( U \) is linearized before \( S_2 \). Conversely, suppose \( S_1 \) is linearized before \( S_2 \) and there is at least one UPDATE operation \( U \) linearized between them. By Lemma 7, \( t(S_1) < t(U) \leq t(S_2) \).

Lemma 9. For any two ISCAN operations \( S_1 \) and \( S_2 \), \( |t(S_1) - t(S_2)| \leq kn \), where \( k \) is the number of UPDATE operations linearized between them.

Proof. Without loss of generality, assume \( S_1 \) is linearized before \( S_2 \). Let \( U_1, U_2, \ldots, U_k \) be the UPDATE operations linearized between \( S_1 \) and \( S_2 \), in that order. By Lemma 7, \( t(S_1) < t(U_1) \). By Lemma 6, \( t(U_k) = t(S_2) \). By Lemma 5, \( t(U_{i+1}) < t(U_i) \) for \( 2 \leq i \leq k \). Suppose the index.read-max step in \( U_i \) returned a value \( t > t(S_1) \). Then there was an index.write-max step \( t' \) after \( S_1 \) with \( t(S_1) < t' < t(U_i) \), and the UPDATE operation \( U \) which performed this index.write-max would be linearized between \( S_1 \) and \( U_i \), contradicting the definition of \( U_i \). Therefore, the index.read-max step in \( U_i \) returned a value which is at most \( t(S_1) \). Similarly, for \( 2 \leq i \leq k \), the index.read-max step in \( U_i \) returned a value which is at most \( t(U_{i+1}) \). It follows by line 3 that \( |t(U_i) - t(S_1)| \leq n \) and \( |t(U_i) - t(U_{i+1})| \leq n \), for \( 2 \leq i \leq k \). It follows that \( |t(S_1) - t(S_2)| = |t(S_1) - t(U_1) + t(U_1) - t(U_2) + \cdots + t(U_k) - t(U_k)| \leq nk \).

Lemma 10. Let \( S \) be an ISCAN operation and let \( U \) be the last UPDATE operation linearized before \( S \). Then every VIEW(t(S), 1, 1, V) operation starting after \( S \) will set \( V[1] \) to the value written by \( U \).

Proof. By Lemma 6, \( t(U) = t(S) \). By Lemma 4, \( U \) is the unique UPDATE operation with \( t(U) = t(S) \). After \( U \) completes, \( A[t(S)] \) contains the value written by \( U \), since each UPDATE operation \( U' \) only writes to \( A[t(U')] \). The index.read-max step in \( S \) occurs after the index.write-max step in \( U \), so \( U \) completes before \( S \). It follows that any VIEW(t(S), 1, 1, V) operation starting after \( S \) will set \( V[1] \) to the value of \( A[t(S)] \), which contains the value written by \( U \).

Theorem 11. There is an implementation of a \( b \)-limited-use 1-component implicit snapshot object from \( w \)-bit registers and 1-bit registers, where \( w \) is the number of bits needed to represent each component of the snapshot. In the implementation, UPDATE consists of a write to a \( w \)-bit register and \( \Theta(\log(bm + 1)) \) writes and reads on \( 1 \)-bit registers, SCAN consists of \( \Theta(\log(bm + 1)) \) reads on \( 1 \)-bit registers, and VIEW consists of a single read on a \( w \)-bit register.

Proof. Lemma 10 shows that a VIEW(t(S), 1, 1, V) operation on an index \( t \) returned by an ISCAN operation \( S \) will set \( V[1] \) to the value of the last UPDATE linearized before \( S \). It follows that the implementation is linearizable. Lemma 8 shows that, for any two ISCANs \( S_1 \) and \( S_2 \), \( t(S_1) < t(S_2) \) if and only if \( S_1 \) is linearized before \( S_2 \) and there is an UPDATE
linearized between them, so the indices returned by ISCAN operations are correct. Lemma 9 shows that index is bounded by \( b_n \), since there can be at most \( b \) UPDATE operations between any two ISCANs. Hence a \((b_n + 1)\)-bounded max register suffices and A needs at most \( b_n + 1 \) entries. By Theorem 1, a READ-MAX or WRITE-MAX on a \((b_n + 1)\)-bounded max register requires \( \Theta(\log(b_n + 1)) \) reads and writes on 1-bit registers.

4.2 An m-component implicit snapshot implementation

For \( m > 1 \), we obtain a \( b \)-limited-use \( m \)-component implicit snapshot recursively. Our implementation is essentially a modification of the implementation in Aspnes, Atiya, Censor-Hillel, and Ellen [4]. The result is a simpler implementation which uses substantially smaller registers.

Let \( \text{snap}_1 \) and \( \text{snap}_2 \) be \( b \)-limited-use \( c_1 \)-component and \( c_2 \)-component implicit snapshots, respectively. Let \( \text{indices} \) be a bounded 2-component max array. We describe a simple, but incorrect implementation of a \( b \)-limited-use \((c_1 + c_2)\)-component implicit snapshot from \( \text{snap}_1 \) and \( \text{snap}_2 \) and then show how to fix it.

UPDATEs on the first \( c_1 \) components are handled by \( \text{snap}_1, \text{update} \) while UPDATEs on the last \( c_2 \) components are handled by \( \text{snap}_2, \text{update} \). The idea is that the ISCAN will use the 2-component max array to keep track of the most recent indices for \( \text{snap}_1 \) and \( \text{snap}_2 \). ISCAN performs \( \text{snap}_1, \text{iscan} \) to obtain index \( u_j \) and performs \( \text{indices}, \text{max-update}(j, u_j) \), for \( j \in \{1, 2\} \). Then it performs \( \text{indices}, \text{max-scan} \) to obtain new indices \( (t_1, t_2) \), which it writes to the register \( T[t_1 + t_2] \), before finally returning \( t_1 + t_2 \). The VIEW operation for an index \( t \) returned by a previously completed ISCAN operation reads \( T[t] \) to obtain indices \( (t_1, t_2) \), and then calls \( \text{snap}_j, \text{view} \) on index \( t_j \), for \( j \in \{1, 2\} \), as necessary to get the appropriate components. By definition of a 2-component max-array, the indices \( (t_1, t_2) \) and \( (t'_1, t'_2) \) seen by two ISCAN operations as a result of their \( \text{indices}, \text{max-scan} \) steps are comparable component-wise. It follows that if \( t_1 + t_2 = t'_1 + t'_2 \), then \( t_1 = t'_1 \) and \( t_2 = t'_2 \). Hence, if two ISCAN operations write to \( T[t] \), for some index \( t \), then they write the same value.

An UPDATE operation consists of a single \( \text{snap}_j, \text{update} \) step, at which it must be linearized. It is tempting to linearize an ISCAN seeing indices \( (t_1, t_2) \) from its \( \text{indices}, \text{max-scan} \) step at the first point in its execution interval that component 1 of indices has index \( t_1 \) and component 2 of indices has index \( t_2 \). It is possible to show that, with these linearization points, for any two ISCAN operation \( S_1 \) and \( S_2 \), if \( t(S_1) < t(S_2) \), then \( S_1 \) is linearized before \( S_2 \) and there was an UPDATE operation linearized between them. The converse, however, does not hold. The problem is that, for \( S_2 \), some UPDATE operations to the first \( c_1 \) components may linearized between the first time that component 1 of indices has index \( t_1 \) and the first time that component 2 of indices has index \( t_2 \), and \( S_2 \) will fail to see these UPDATEs, even though it is linearized after them.

To fix this, we ensure that an UPDATE completes only after it knows it will be seen by all ISCANs linearized after it. Thus, after the UPDATE operation updates \( \text{snap}_j \), it performs \( \text{snap}_j, \text{iscan} \) to obtain an index \( t \), and then performs \( \text{indices}, \text{max-update}(j, t) \). Since UPDATEs now perform \( \text{snap}_j, \text{iscan} \) and \( \text{indices}, \text{max-update}(j, \cdot) \), we can, in fact, remove the \( \text{snap}_j, \text{iscan} \) and \( \text{indices}, \text{max-update}(j, \cdot) \) steps from ISCAN, for \( j \in \{1, 2\} \). The pseudocode is presented in Algorithm 2.

Recall that, from Theorem 2, a bounded 2-component max array can be implemented from 1-bit registers. Thus, we will assume that all operations on indices are atomic and treat them as steps in the executions we consider.

An ISCAN operation \( S \) is linearized at its \( \text{indices}, \text{max-scan} \) step. We use \((t_1(S), t_2(S))\) to denote the value returned by this step, so that \( t(S) = t_1(S) + t_2(S) \). An UPDATE operation
Lemma 12. An UPDATE operation U that updates snap_j is linearized before an ISCAN operation S if and only if X(U) occurs before the first update step returning t_j(S).

Proof. Suppose U is linearized before S. Let Y be the first update step returning t_j(S). Since U is linearized before S, its linearization point, Z(U), occurs before the update step returning t_j(S). If t(Y(U)) < t(Y), then X(U) → Y(U) → Y. Otherwise, if t(Y(U)) = t(Y), then Y → Y(U) by definition of Y. Since t(Y) = t(Y(U)), there are no update steps between Y and Y(U), so we must have X(U) → Y → Y(U).

Conversely, suppose X(U) occurs before the first update step Y returning t_j(S). Let Z be the first update step before S, which is part of some UPDATE operation U', and let Y' be the update step in U' which returned t_j(S). By assumption, X(U) → Y → Y', so Y' ∈ Y(U) and Z ∈ Z(U). Thus, U is linearized no later than Z, which is before the update step in S, so U is linearized before S.

Lemma 13. Let S be an ISCAN operation, and let U_1, . . . , U_p be the UPDATE operations linearized before S that update snap_j in the order in which they are linearized. Then X(U_1) → X(U_2) → · · · → X(U_p) and t(Y(U_1)) ≤ t(Y(U_2)) ≤ · · · ≤ t(Y(U_p)) = t_j(S).
Proof. Let $1 \leq i < j \leq p$. Suppose, for a contradiction, that $t(Y(U_i)) > t(Y(U_j))$, so $X(U_j) \rightarrow Y(U_j) \rightarrow Y(U_i)$ and $Z(U_j) \in Z(U_i)$. It follows that $Z(U_i) = Z(U_j)$, for otherwise $U_j$ is linearized before $U_i$. Thus, $Y(U_j) = Y(U_i)$, so $t(Y(U_i)) = t(Y(U_j))$, which is a contradiction. To see that $t(Y(U_p)) = t_j(S)$, note that the UPDATE operation which performs the first $\text{indices}.\max$-$\text{update}(j, t_j(S))$ step $Z$, is linearized no later than $Z$, which is linearized before $S$.

Similarly, suppose for a contradiction that $X(U_j) \rightarrow X(U_i)$. It follows that $X(U_j) \rightarrow Y(U_i), Y(U_i) \in Y(U_j)$, and $Z(U_i) \in Z(U_j)$. Thus, $U_j$ would be linearized at no later than $Z(U_i)$, and it would be linearized before $U_i$ since $X(U_j) \rightarrow X(U_i)$, a contradiction.

**Lemma 14.** Consider any ISCAN operation $S$ and any $\text{VIEW}((t(S), 1, c_1 + c_2), V[1..c_1 + c_2])$ operation. Then, after this $\text{VIEW}$ operation completes, $V[c]$ is set to the value of the last $\text{UPDATE}$ operation on component $c$ linearized before $S$, or $\perp$ if no such operation exists, for all components $c$.

**Proof.** We only consider when $c \in \{1, \ldots, c_1\}$; the case for $c \in \{c_1 + 1, \ldots, c_1 + c_2\}$ is similar. Suppose that no $\text{UPDATE}$ to component $c$ is linearized before $S$. By Lemma 12, no $\text{snap}_1, \text{update}(c, u)$ step occurs before the first $\text{snap}_1, \text{iscan}$ step returning index $t_1(S)$. Thus, the view of $\text{snap}_1$ at index $t_1(S)$ has component $c$ set to $\perp$, so $V[c] = \perp$.

Now, suppose some $\text{UPDATE}$ to component $c$ is linearized before $S$. Let $U$ be the last such $\text{UPDATE}$, and let $Y$ be the first $\text{snap}_1, \text{iscan}$ step returning index $t_1(S)$. By Lemma 12, any $\text{UPDATE}$ operation $U'$ on component $c$ such that $X(U') \rightarrow Y$ is linearized before $S$. Since $U$ is the last $\text{UPDATE}$ on component $c$ linearized before $S$, Lemma 13 implies that $X(U') \rightarrow X(U)$. It follows that the view of $\text{snap}_1$ at index $t_1(S)$ has component $c$ set to the value of $X(U)$, which is the value of $U$, so $V[c]$ is set to the value of $U$.

**Lemma 15.** For any two ISCAN operations $S_1$ and $S_2$, $t(S_1) < t(S_2)$ if and only if $S_1$ is linearized before $S_2$ and there is at least one $\text{UPDATE}$ linearized between them.

**Proof.** If $t(S_1) < t(S_2)$, then $S_1$ is linearized before $S_2$ since they are linearized at their respective $\text{indices}.\max$-$\text{scan}$ steps. Furthermore, we must have either $t_1(S_1) < t_1(S_2)$ or $t_2(S_1) < t_2(S_2)$. Suppose $t_1(S_1) < t_1(S_2)$; the other case is similar. Since $t_1(S_1) < t_1(S_2)$, there was a $\text{indices}.\max$-$\text{update}(1, t_1(S_2))$ step $Z$ before the start of $S_2$. Let $U$ be the $\text{UPDATE}$ operation which performed $Z$. Let $Y$ be the first $\text{snap}_1, \text{iscan}$ with $t(Y) = t_1(S_2)$. Since $t(Y) = t(Y(U))$, $X(U) \rightarrow Y$ and Lemma 12 implies that $U$ is linearized before $S$. Since $t(Y(U)) = t_1(S_2) > t_1(S_1)$, Lemma 13 implies that $U$ is linearized after $S_1$.

Conversely, suppose $S_1$ is linearized before $S_2$ and there is an $\text{UPDATE}$ operation $U$ linearized between them. Suppose that $U$ updates $\text{snap}_1$. Since $S_1$ is linearized before $S_2$, $t(S_1) \leq t(S_2)$. Since $U$ is linearized after $S_1$, by Lemma 12, $X(U)$ occurred after the first $\text{snap}_1, \text{iscan}$ step $Y$ returning $t_j(S_1)$. Since $Y \rightarrow X(U) \rightarrow Y(U)$, it follows that $t(Y(U)) > t(Y) = t_j(S_1)$. Furthermore, since $U$ is linearized before $S_2$, by Lemma 13, $t(Y(U)) \leq t_j(S_2)$. Therefore $t_j(S_1) = t_j(S_1) + t_3-j(S_1) < t_j(S_2) + t_3-j(S_2) = t_j(S_2)$, where $t_2(S_1) \leq t_2(S_2)$ follows since the $\text{indices}.\max$-$\text{scan}$ step in $S_1$ occurs before the $\text{indices}.\max$-$\text{scan}$ step in $S_2$.

**Lemma 16.** For every $\ell \geq 0$, there is an implementation of a $b$-limited use $2^\ell$-component implicit snapshot from $w$-bit registers, $\Theta(\log(bn + 1))$-bit registers, and 1-bit registers, where $w$ is the number of bits needed to represent each component of the snapshot. The implementation satisfies the following properties:
1. The ISCAN operation consists of \( O((\log(bn + 1))^2) \) reads and writes on 1-bit registers and a write to a \( \Theta(\log(bn + 1)) \)-bit register. Furthermore, for any two ISCAN operations \( S_1 \) and \( S_2 \), \( |t(S_1) - t(S_2)| \leq kn \), where \( k \) is the number of UPDATE operations linearized between \( S_1 \) and \( S_2 \).

2. The UPDATE operation consists of \( O((\log(bn + 1))^2(\ell + 1)) \) reads and writes on 1-bit registers, \( \ell \) writes to \( \Theta(\log(bn + 1)) \)-bit registers, and a write to a \( w \)-bit register.

3. The VIEW operation on an index returned by an ISCAN operation and a range \((i, j)\) of components consists of \( T_{\ell}(i, j) = 1 + \ell + 2(j - i) - \sum_{d=0}^{\ell-1}(yd - xd) \leq 1 + 2(\ell + j - i) \) reads on \( \Theta(\log(bn + 1)) \)-bit registers and \( j - i + 1 \) reads on \( w \)-bit registers, where \( x_{\ell-1} \cdots x_0 \) and \( y_{\ell-1} \cdots y_0 \) are the binary representations of \( i - 1 \) and \( j - 1 \), respectively.

Proof. By induction on \( \ell \). The base case, when \( \ell = 0 \), holds by Theorem 11. Suppose now that the claim holds for \( \ell \) and consider \( \ell + 1 \). We consider what happens when we build a \( b \)-limited use \( 2^{\ell+1} \)-component implicit snapshot object from two \( b \)-limited use \( 2^\ell \)-component implicit snapshot objects using Algorithm 2.

(1) Let \( S_1 \) and \( S_2 \) be any two ISCAN operations. For \( i, j \in \{1, 2\} \), let \( Y_i \) be the first \( \text{snap}_i.\text{iscan} \) step returning \( t_j(S_i) \). By assumption, \( |t_j(Y_{i1}) - t_j(Y_{i2})| \leq q_i n \), where \( k_i \) is the number of UPDATE operations which had their \( \text{snap}_i.\text{update} \) step linearized between \( Y_{i1} \) and \( Y_{i2} \). By Lemma 12, an UPDATE operation \( U \) which updates \( \text{snap}_i \), is linearized between \( S_1 \) and \( S_2 \) if and only if \( X(U) \) occurs between \( Y_{i1} \) and \( Y_{i2} \). It follows that \( k_1 + k_2 = k \). Thus, \( |t(S_1) - t(S_2)| = |t_j(Y_{11}) - t_j(Y_{21}) + t_j(Y_{12}) - t_j(Y_{22})| \leq |t_j(Y_{11}) - t_j(Y_{12})| + |t_j(Y_{21}) - t_j(Y_{22})| \leq (k_1 + k_2)n = kn \). Since \( k \leq b \), this shows that indices can be \( (bn + 1, bn + 1) \)-bounded and \( T \) can be an array of \( bn + 1 \) \( \Theta(\log(bn + 1)) \)-bit registers. By Theorem 2, indices can be implemented from 1-bit registers with step complexity \( \Theta((\log(bn + 1))^2) \) for MAX-UPDATE and MAX-SCAN. Therefore, an ISCAN consists of \( O((\log(bn + 1))^2(\ell + 1)) \) reads and writes on 1-bit registers, and a write to a \( \Theta(\log(bn + 1)) \)-bit register.

(2) \( \text{snap}_i.\text{update} \) consists of \( O((\log(bn + 1))^2(\ell + 1)) \) reads and writes on 1-bit registers, \( \ell \) writes to \( \Theta(\log(bn + 1)) \)-bit registers, and a write to a \( w \)-bit register. \( \text{snap}_i.\text{iscan} \) consists of \( O((\log(bn + 1))^2) \) reads and writes on 1-bit registers and a write to a \( \Theta(\log(bn + 1)) \)-bit register. indices.\text{max-update} \) consists of \( O((\log(bn + 1))^2) \) reads and writes on 1-bit registers. Thus, in total, an UPDATE consists of \( O((\log(bn + 1))^2(\ell + 2)) \) reads and writes on 1-bit registers, \( \ell + 1 \) writes to \( \Theta(\log(bn + 1)) \)-bit registers, and a write to a \( w \)-bit register.

(3) By Algorithm 2, a VIEW operation at an index returned by an ISCAN operation on a range \((i, j)\) of components consists of

\[
T_{\ell+1}(i, j) = \begin{cases} 
T_{\ell}(i - 2^\ell, j - 2^\ell) + 1 & i > 2^\ell \\
T_{\ell}(i, j) + 1 & j \leq 2^\ell \\
T_{\ell}(i, 2^\ell) + T_{\ell}(1, j - 2^\ell) + 1 & \text{otherwise}
\end{cases}
\]

reads on \( \Theta(\log(bn + 1)) \)-bit registers. Let \( x_\ell \cdots x_0 \) and \( y_\ell \cdots y_0 \) be the binary representations of \( i - 1 \) and \( j - 1 \), respectively. If \( i > 2^\ell \), then \( x_\ell = y_\ell = 1 \) and \( 0x_{\ell-1} \cdots x_0 \) and \( 0y_{\ell-1} \cdots y_0 \) are the binary representations of \( i - 2^\ell \) and \( j - 2^\ell \) - 1. In this case, it follows that

\[
T_{\ell+1}(i, j) = T_{\ell}(i - 2^\ell, j - 2^\ell) + 1 = 1 + \ell + 2(j - 2^\ell) - (i - 2^\ell) - \sum_{d=0}^{\ell-1}(x_d - y_d) + 1
\]

\[
= 1 + (\ell + 1) + 2(j - i) - \sum_{d=0}^{\ell}(x_d - y_d) .
\]
Atomic Snapshots from Small Registers

If \( j \leq 2^\ell \), then \( x_t = y_t = 0 \) and \( 0x_{k-1} \ldots x_0 \) and \( 0y_{k-1} \ldots y_0 \) are the binary representations of \( i - 1 \) and \( j - 1 \), respectively. In this case, it follows that

\[
T_{\ell+1}(i, j) = T_{\ell}(i, j) + 1 = 1 + \ell + 2(j - i) - \sum_{d=0}^{\ell-1} (y_d - x_d) + 1
\]

\[
= 1 + (\ell + 1) + 2(j - i) + \sum_{d=0}^{\ell} (y_d - x_d). 
\]

Otherwise, if \( i \leq 2^\ell \) and \( j > 2^\ell \), then \( x_t = 0 \) and \( y_t = 1 \) and \( 0x_{\ell-1} \ldots x_0 \) and \( 0y_{\ell-1} \ldots y_0 \) are the binary representations of \( i - 1 \) and \( (j - 2^\ell) - 1 \), respectively. Note that the binary representation of \( 2^\ell - 1 \) is \( 01 \cdots 1 \) (one zero followed by \( \ell - 1 \) ones). In this case, it follows that

\[
T_{\ell+1}(i, j) = T_{\ell}(i, 2^\ell) + T_{\ell}(1, j - 2^\ell) + 1
\]

\[
= \left[ 1 + \ell + 2(2^\ell - i) - \sum_{d=0}^{\ell-1} (1 - x_d) \right] + \left[ 1 + \ell + 2(j - 2^\ell - 1) - \sum_{d=0}^{\ell-1} y_d \right] + 1
\]

\[
= 1 + (\ell + 1) + 2(j - i) - \sum_{d=0}^{\ell} (y_d - x_d).
\]

In all 3 cases, there are \( j - i + 1 \) reads on \( w \)-bit registers.

Inductively constructing implicit snapshots from implicit snapshots on a smaller number of components via Lemma 16, we obtain the following:

**Theorem 17.** There is an implementation of a \( b \)-limited-use \( m \)-component implicit snapshot from \( w \)-bit registers, \( \Theta((\log(bn+1))^2 \log m) \)-bit registers, and \( 1 \)-bit registers. The implementation satisfies the following:

1. The ISC\( An \) operation consists of \( O((\log(bn+1))^2) \) reads and writes on \( 1 \)-bit registers and a write to a \( \Theta((\log(bn+1))^2) \)-bit register.
2. The UPDATE operation consists of \( O((\log(bn+1))^2 \log m) \) reads and writes on \( 1 \)-bit registers, \( \log m \) writes to \( \Theta((\log(bn+1))^2) \)-bit registers, and a write to a \( w \)-bit register.
3. The VIEW operation on an index returned by an ISC\( An \) operation and a range \((i, j)\) of components consists of at most \( 1 + 2(\log m + j - i) \) reads on \( \Theta((\log(bn+1))^2) \)-bit registers and \( j - i + 1 \) reads on \( w \)-bit registers.

**Algorithm 3** A faster \( b \)-limited-use \( m \)-component implicit snapshot object.

1: procedure UPDATE\((c, v)\)
2: \hspace{1em} old.update\((c, v)\)
3: \hspace{1em} index.write-max\((\text{old.iscan})\)
4: procedure ISC\( An( )\)
5: \hspace{1em} return index.read-max\()
6: procedure VIEW\((t, i, j, V)\)
7: \hspace{1em} old.view\((t, i, j, V)\)

We can reduce the step complexity of ISC\( An \) to \( O((\log(bn+1))^2) \) reads on \( \Theta((\log(bn+1))^2) \)-bit registers. The idea is to use the old implicit snapshot implementation from Theorem 17 and use a \((bn+1)\)-bounded max register index, as in Algorithm 1, to store the most recent
index. An UPDATE operation performs \texttt{old.update} to handle the actual update and then performs \texttt{index.write-max(old.iscan())} to store the most recent index. An ISCAN simply returns the result of \texttt{index.read-max}. A VIEW operation calls \texttt{old.view}. See Algorithm 3 for the pseudocode.

Since \texttt{old} is linearizable, we can assume that the embedded \texttt{old.update} and \texttt{old.iscan} operations are atomic. UPDATEs which write \texttt{t} to \texttt{index} are linearized at the first point that \texttt{index} has value at least \texttt{t}. Ties are broken in order of the embedded \texttt{old.update} operations. ISCANs are linearized when they perform \texttt{index.read-max}.

Finally, by using the large register simulation of Aghazadeh, Golab, and Woelfel [2], we can obtain an implementation from \texttt{k}-bit registers, for \texttt{k} \in \Omega(\log n). Putting this all together, we obtain the following:

\begin{itemize}
  \item Theorem 18. For \texttt{b} \in n^{O(1)} and \texttt{k} \in \Omega(\log n), there is an implementation of a \texttt{b}-limited-use \texttt{m-component snapshot} from \texttt{k}-bit registers with step complexity \texttt{O(w + (\log n)^2 \log m)} for UPDATE and step complexity \texttt{O(mw + \log n)} for SCAN, if each component of the snapshot consists of \texttt{\Theta(wk)} bits.
\end{itemize}

\section{Conclusions}

Aghazadeh, Golab, and Woelfel’s [2] large register simulation requires registers with \Omega(\log n) bits. Peterson’s large register simulation works for arbitrarily small registers, but it is a single-writer register simulation and it is unclear if the simulation can be modified to efficiently implement interruptible reads. It would be interesting to find a large register simulation, which works for arbitrarily small registers, that can implement efficient interruptible reads.

It is not possible to directly apply our techniques to other, more efficient, snapshots [5, 6, 9] without significantly increasing their step complexities because they read too many large registers and they need most of the bits they read. We would like to investigate the possibility of modifying these snapshots to obtain a faster snapshot from \Theta(\log n)-bit registers.

For \texttt{b} \in n^{O(1)}, we showed how to implement a \texttt{b}-limited-use \texttt{m-component snapshot} from \Theta(\log n)-bit registers with UPDATE step complexity \Theta(w + (\log n)^2 \log m) and SCAN step complexity \Theta(mw + \log n), if each component of the snapshot consists of \Theta(w \log n) bits. We had to use multi-writer registers. It would be interesting to see if it is possible to obtain similar results with only single-writer registers.

\section{Acknowledgments}

We thank Hagit Attiya and David Solymosi for helpful discussions. This research was supported by the Natural Science and Engineering Research Council of Canada.

\section{References}


