# Space-Optimal Multi-Writer Snapshot Objects Are Slow

Panagiota Fatourou, University of Ioannina
Faith Fich, University of Toronto
Eric Ruppert, York University

## Snapshot Object

- shared data structure
- stores array of $m$ components
- $\texttt{UPDATE}(i, val)$ changes $i$th element to $val$
- $\texttt{SCAN}$ atomically reads entire array

Single-Writer (SW) snapshot:
Only process $i$ can do $\texttt{UPDATE}(i, val)$.

Multi-Writer (MW) snapshot:
Every process can update every component.

## Implementing Snapshots

Model:
Asynchronous system
Linearizable objects
Wait-free (any number of crash failures)

- Snapshots are easier to use than registers.
- Snapshots have wait-free implementations from registers.
  [AADGMS, Anderson, Aspnes/Herlihy]
- How efficient can the implementation be?

## Best MW Snapshot Implementations

$m$ = number of components ($m > 1$)
$n$ = number of processes

From unbounded number of MW registers:
$O(n)$ time. [Anderson + (ICMT or ISS)]

From $n$ bounded-size SW registers:
$O(n \log n)$ for UPDATES, $O(n)$ for SCANS.
[Anderson + Attiya/Rachman]

From $m$ MW registers:
$O(mn)$ time. [modification of AADGMS]

From $m$ bounded-size MW registers and
$n$ SW registers:
$O(n^2 + mn)$ time. [AADGMS]

## The Lower Bound

A MW snapshot implemented from $m$ MW registers uses $\Omega(n)$ time for a `SCAN`.

(Previous lower bound: $\Omega(\min(m, n))$ [Jayanti/Tan/Toueg])

Additional results from the proof:

At least $m$ MW registers are required.
(Previous lower bound: $m - 1$ [Jayanti/Tan/Toueg])

In space-optimal implementations,
`SCANS` cannot `write`, and
`UPDATES` to component $i$ `write` only to register $i$.

## Ideas for the Proof

Adversary argument: design an execution where troublesome `SCAN` takes many steps.

Ensure troublesome `SCAN` never has enough up-to-date information to produce correct output.

Before troublesome `SCAN` **`reads`** a register, a process **`writes`** useless information in it.

## Ideas, contd

First, prove lower bound using 2 assumptions:

`SCANS` cannot `write`, and
`UPDATES` to component $i$ `write` only to register $i$.

Then use covering argument to show these assumptions hold for any implementation that uses at most $m$ registers.

## The bad execution

We have $n$ processes:
- 1 process executing troublesome `SCAN`,
- n-2 processes each doing an `UPDATE` to **`write`** very old information in registers before troublesome `SCAN` **`reads`** them.
- 1 process performing lots of `UPDATES`

Ensure that
(1) troublesome SCAN sees only red values, and no blue values, and
(2) at all times, at least one component of the snapshot object contains a blue value.

To guarantee (2), we add extra `SCANS` to fix the linearization order of the `UPDATES` and `UPDATES`.

## The bad execution

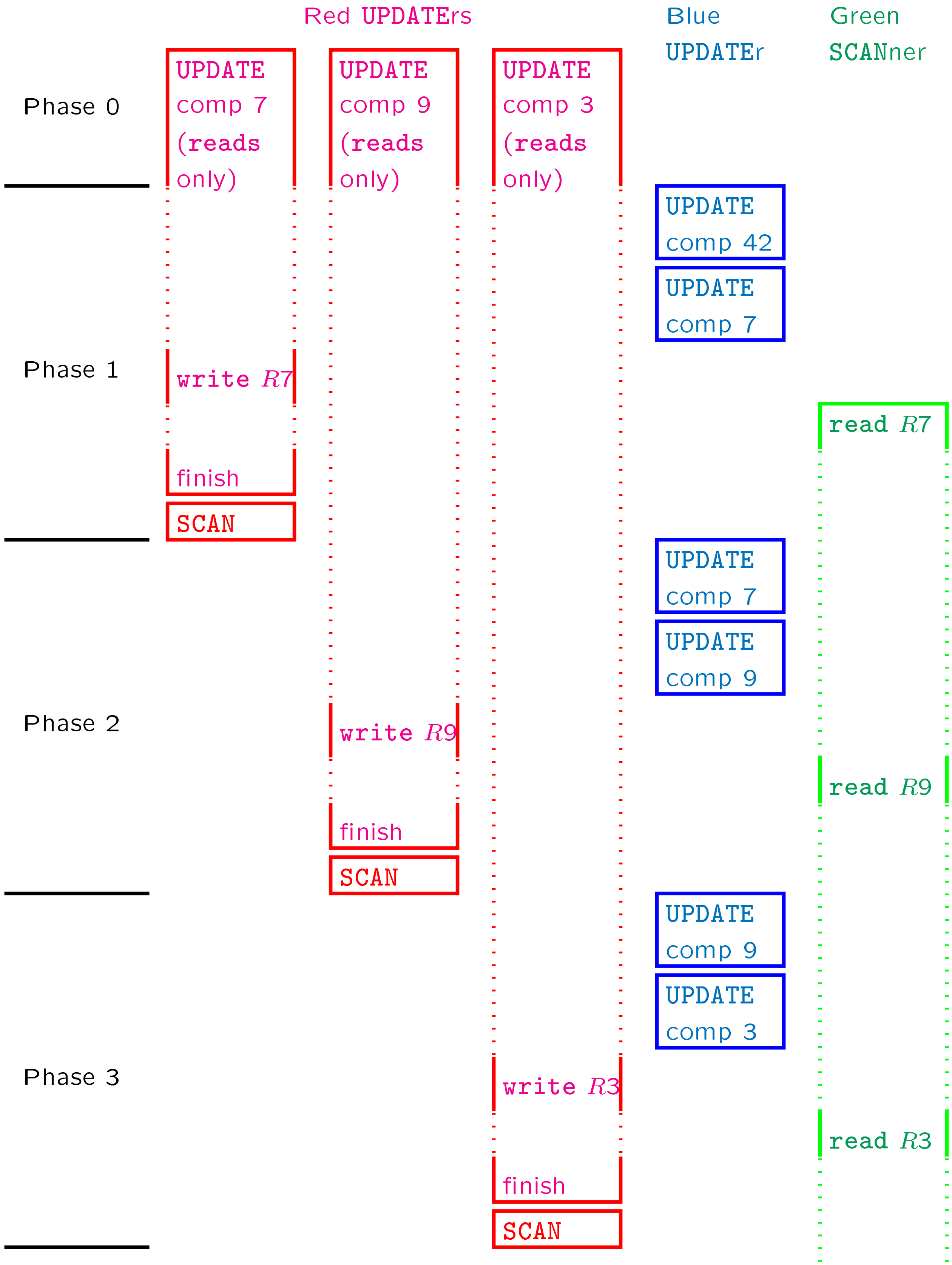
Phase 0:

- Run each red process until just before its first `write`.

Phase i:

- blue process `UPDATES` 2 components.
- $i$th red process `writes` to a register.

- troublesome `SCAN` `reads` the register.
- $i$th red process finishes its `UPDATE`, then `SCANS`.



Red UPDATErs
Blue UPDATEr
Green SCANner
Phase 0
Phase 1
Phase 2
Phase 3
UPDATE comp 7 (reads only)
UPDATE comp 9 (reads only)
UPDATE comp 3 (reads only)
UPDATE comp 42
UPDATE comp 7
write $R7$
read $R7$
finish
SCAN
UPDATE comp 7
UPDATE comp 9
write $R9$
read $R9$
finish
SCAN
UPDATE comp 9
UPDATE comp 3
write $R3$
read $R3$
finish
SCAN

## Linearization

We have following linearization:

`UPDATE` component 42
`UPDATE` component 7
`UPDATE` component 7
`SCAN`
`UPDATE` component 7
`UPDATE` component 9
`UPDATE` component 9
`SCAN`
`UPDATE` component 9
`UPDATE` component 13
`UPDATE` component 13
`SCAN`

Note: after first update, snapshot always has blue value in at least one component.

Conclusion: Troublesome `SCAN` cannot terminate before doing $n - 1$ **`reads`**.

## Removing the Assumptions

We assumed:

`SCANS` cannot `write`, and
`UPDATES` to component $i$ `write` only to register $i$.

To remove these assumptions:

Take any implementation using at most $m$ MW registers.

We prove that any set of `UPDATES` to fewer than $k$ components cannot cover $k$ different registers.

## More Precisely...

A set of processes $\mathcal{P}'$ is hidden in configuration $C$ if, starting from $C$, no process outside $\mathcal{P}'$ can ever tell that processes inside $\mathcal{P}'$ are working on `UPDATES`.

A configuration $C$ is $k$-fatal if
$\exists$ set $\mathcal{P}$ of $k$ processes s.t.
- processes in $\mathcal{P}$ cover $k$ different registers in $C$,
- processes in $\mathcal{P}$ are working on `UPDATES` to fewer than $k$ components, and
- $\forall$ sets $\mathcal{P}'$ of processes that are not updating those components, $\mathcal{P}'$ is hidden in $C$.

If an implementation ever reaches a fatal configuration, the implementation is wrong!

## Fatality

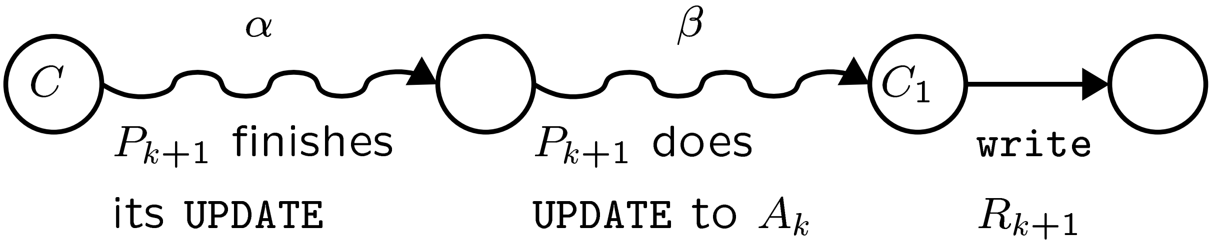
Theorem No correct implementation using at most $m$ registers ever reaches a $k$-fatal configuration.

Proof idea: Reverse induction on $k$.

Base case ($k = m + 1$): trivial, since there are only $m$ registers.

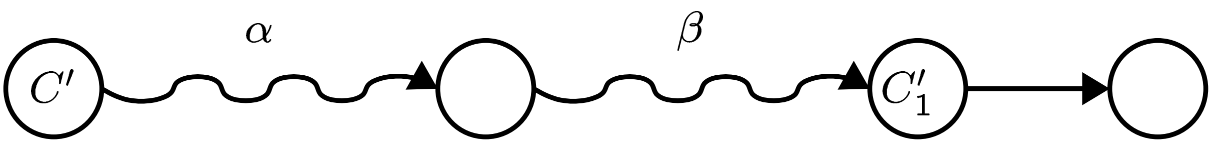
Induction step:
Assume there is a $k$-fatal configuration.
Pick a new process.
Have it update a different component.
Run it until it covers a different register.
Prove resulting configuration is $(k + 1)$-fatal.

Assume $C$ is $k$-fatal: Processes $P_1, \ldots, P_k$ are updating components $A_1, \ldots, A_{k-1}$ and covering registers $R_1, \ldots, R_k$.




- Show $C_1$ is $(k+1)$-fatal.
- Let $\mathcal{P}'$ be set of processes not updating $A_1, \ldots, A_k$ in $C_1$.
- Must show $\mathcal{P}'$ is hidden.

By definition of $k$-fatal, $\mathcal{P}'$ hidden in $C$.
I.e. $C$ looks (to processes outside $\mathcal{P}'$) like $C'$, and no process in $\mathcal{P}'$ has pending update in $C'$.




$C_1$ looks (to processes outside $\mathcal{P}'$) like $C_1'$, and no process in $\mathcal{P}'$ has pending update in $C_1'$.

## Future Directions

Close gap between $\Omega(n)$ and $O(mn)$ for space-optimal implementations.

Time/space tradeoff for snapshots?
(Some progress for the case where additional SW registers are available.)

Applying technique to get time lower bounds for other space-constrained algorithms.