Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations

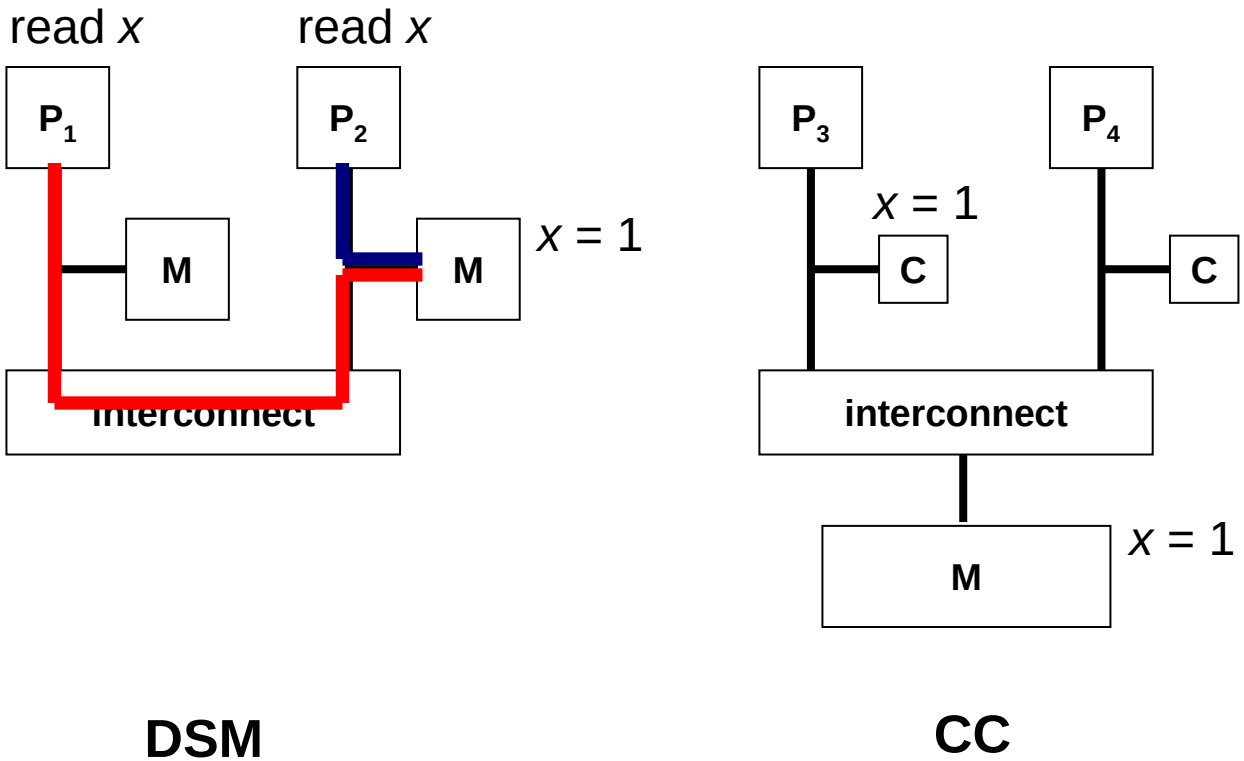
W. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel
University of Toronto, Ben-Gurion University, and University of Calgary

26th PODC
August 13, 2007
Portland, Oregon

Outline

- **Introduction**
- Motivation
- Upper Bounds
  - Name Consensus
  - Compare-and-Swap
- Lower Bound for Bounded Counters

Shared memory architectures



Legend:

P_i – processor *i*
M – memory
C – cache

local memory reference
remote memory reference

Synchronization in asynchronous s.m. multiprocessors – paradigms

- *blocking*
  - active processes are live and do not crash
  - e.g., mutual exclusion
- *non-blocking*
  - processes may crash
  - e.g., wait-free, lock-free, obstruction-free

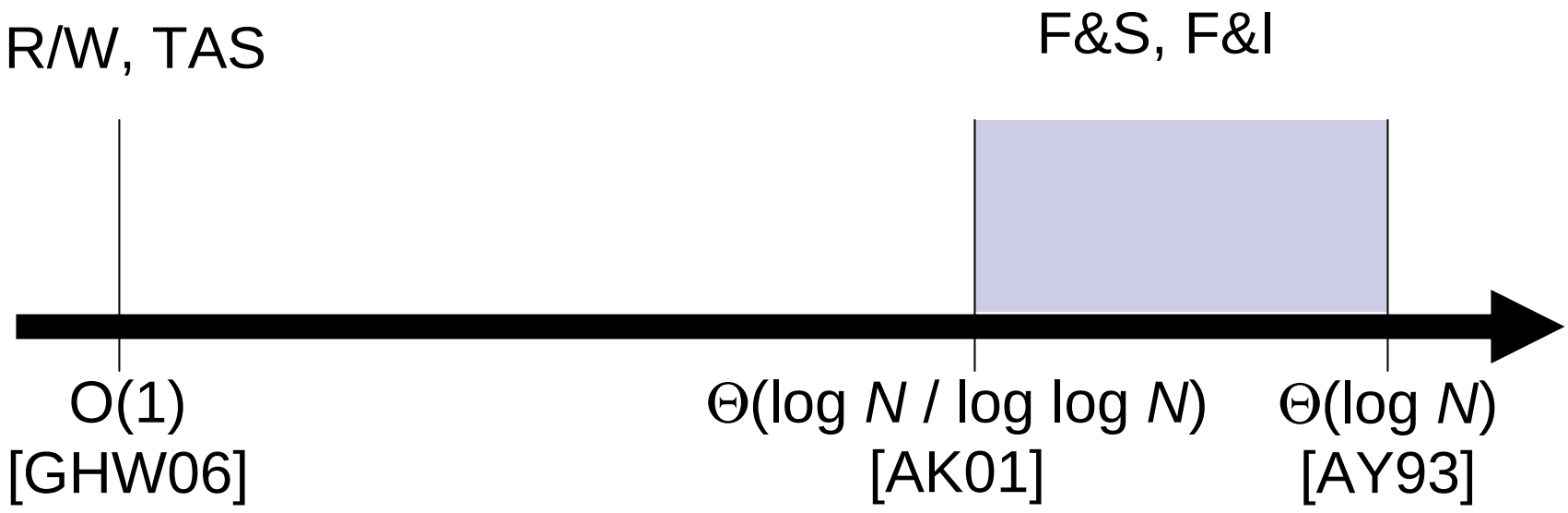
Outline

- Introduction
- **Motivation**
- Upper Bounds
  - Name Consensus
  - Compare-and-Swap
- Lower Bound for Bounded Counters

Motivation

Other related work:
- wait-free hierarchy [Her91]
- queuing locks, e.g., [And90], [Cra93]
- l.b. on state changes [FL06]

CAS, LL/SC, 2-bounded counter



worst-case RMRs for *N*-proc. implementation using R/W only

- Introduction
- Motivation
- **Upper Bounds**
  - Name Consensus
  - Compare-and-Swap
- Lower Bound for Bounded Counters

Simulation of common primitives using reads and writes only

- linearizable implementations of
  1. all comparison primitives (see [AK01])
  2. LL/SC
- O(1) RMRs per operation
- models: DSM and CC

Simulation of algorithms using reads and writes only

- simulate algorithms that use
  1. reads and writes
  2. comparison primitives
  3. LL/SC
- O(1) blowup in RMR complexity
- models: DSM and CC

- Introduction
- Motivation
- Upper Bounds
  - **Name Consensus**
  - Compare-and-Swap
- Lower Bound for Bounded Counters

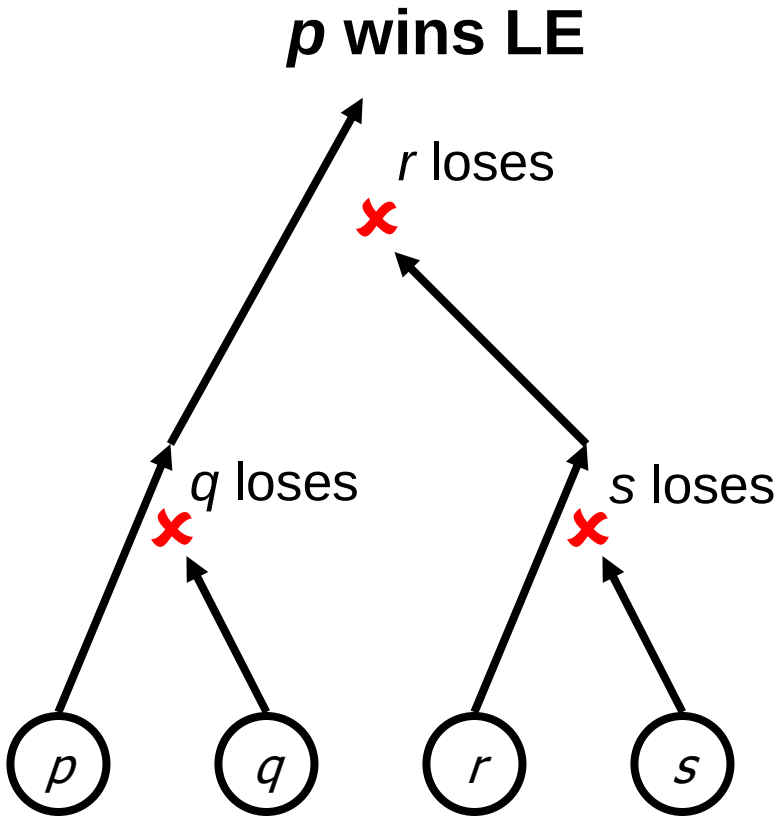
Specification of leader election (LE) and name consensus (NC)

	LE	NC
idea	exactly one active process wins (the "winner")	
response	**Boolean**	**ID of winner**
algo. using reads and writes	O(1) RMRs [GHW06]	O(1) RMRs [GHHW07]

From LE to NC using reads and writes: CC model

```
    /* initially  winner := -1 */
1.  if  (electLeader() = TRUE)  then
2.      winner := PID
3.  else
4.      await  winner ≠ -1
5.  return  winner
```

From LE to NC using reads and writes: DSM model



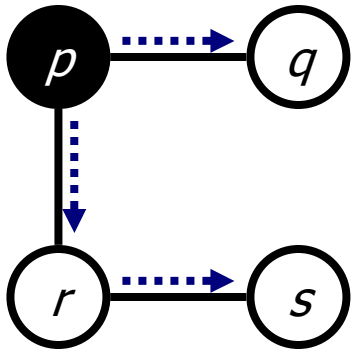
processes competing in LE

Data flow graph of an execution

- *definition:*
  - $G = (V, E)$ (undirected)
  - V = set of processes
  - $\{p, q\} \in E \Leftrightarrow$ p read a value last written by q (or vice versa)
- *observation:*
  - G connected in certain executions of LE

Idea behind DSM NC algorithm

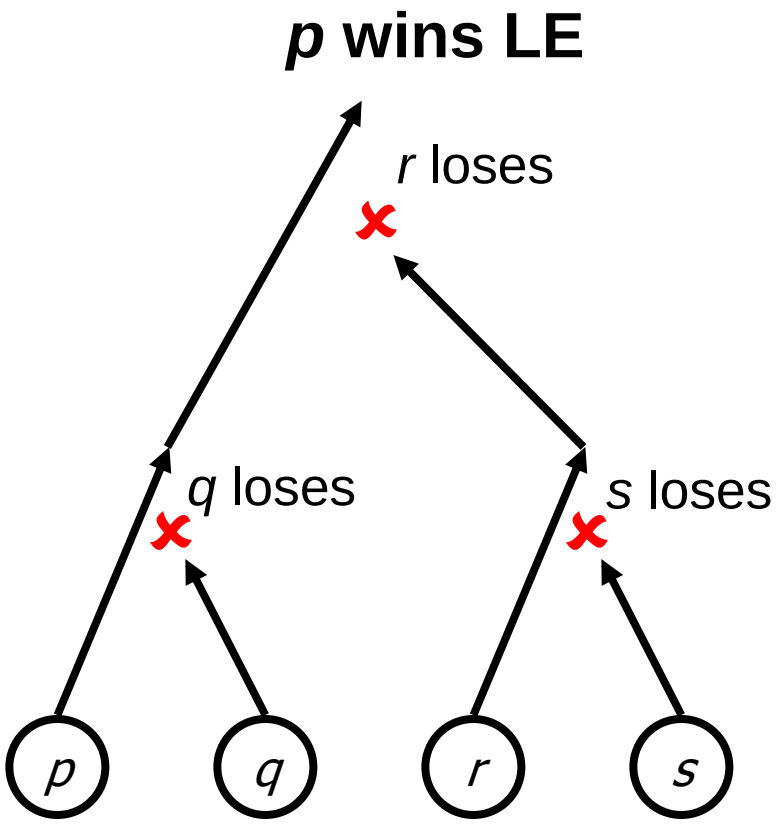
data flow graph *G*



= "*p* won"

Challenges in using *G*:

- *G* difficult to construct
- teamwork required



processes competing in LE

Generalization

- given
  - LE with $f(N)$ RMRs worst-case
  - reads and writes
- obtain
  - NC with $O(f(N))$ RMRs worst-case

- Introduction
- Motivation
- Upper Bounds
  - Name Consensus
  - **Compare-and-Swap**
- Lower Bound for Bounded Counters

Specification of CAS

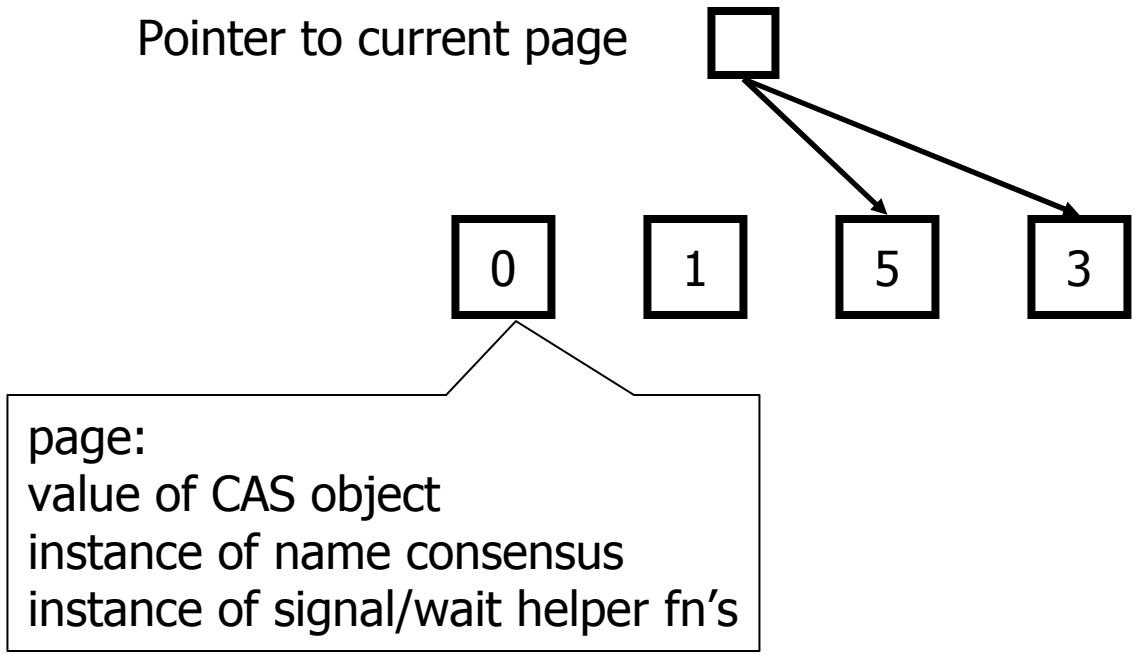
var.CAS(*cmp*, *new*)

1. *old* := *var.value*
2. **if** (*old* = *cmp*) **then**
3. *var.value* := *new*
4. **return** *old*

successful ⇔ returns *cmp*

Idea behind CAS algorithm

- CAS resembles repeated name consensus



***p* – CAS(5, 3)**
concurrently
***q* – CAS(5, 4)**

***p* wins NC**
q* loses NC, waits for *p

***p* allocates next page**

***p* swings pointer**

***p* tells *q* to return 3**
***p* returns 5**

***q* returns 3**

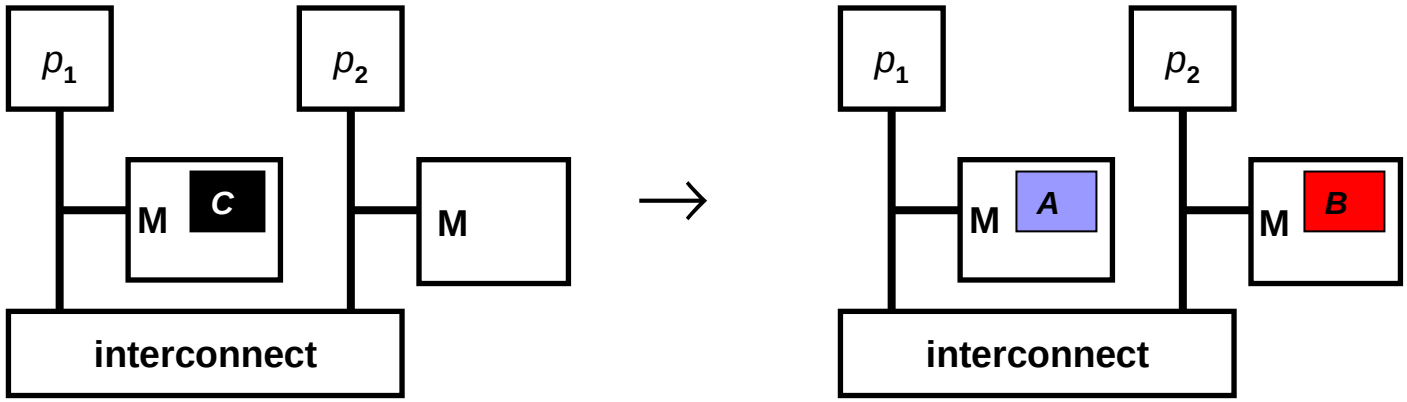
Implications

- sim. of common primitives with O(1) RMRs
  - LL/SC (using [Moir97], [Jayanti98])
  - any comparison primitive (using CAS)
  - using reads and writes only
- sim. of algorithms with O(1) blow-up in RMRs
  - reads and writes, comparison primitives, and LL/SC

    $\rightarrow$

    reads and writes only
- **caveat:** locality property

Locality property: DSM model



C implemented using { A, B }

- Introduction
- Motivation
- Upper Bounds
  - Name Consensus
  - Compare-and-Swap
- **Lower Bound for Bounded Counters**

Object specification [AK03]

k-bounded counter:

- states: 0,1,…, *k*
- operations:
  - Increase: $x \rightarrow \min\{k, x + 1\}$
  - ResetOnOne: $1 \rightarrow 0$
  - BlindReset: $x \rightarrow 0$

return previous value

1-bounded counter ≈ resettable TAS

Result

- lower bound for 2-bounded counter
  - linearizable implementation
  - using
    - reads and writes
    - comparison primitives
    - LL/SC
  - for N processes
  - $\Omega(\log \log N)$ RMRs per operation (amortized)

Implications

N-process ME worst-case RMR complexity

O(log *N* / log log *N*) ME algo [AK03]
(R/W, 2-bounded counters)

+ simulate 2-bounded counter using R/W with O(1) RMRs (hypothetically)

$\Rightarrow$

O(log *N* / log log *N*) ME algo
(using R/W only)

- would match [AK01] lower bound
- would beat [AY93] upper bound
- would *not* contradict [FL06] $\Omega(\log N)$ lower bound

Open Problems

- tight RMR bound for ME using R/W (and comparison primitives, LL/SC)
- is RMR complexity the right measure?

Questions and answers

Thank you!