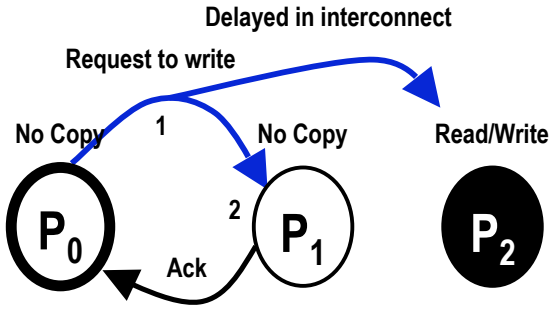
18-742
Lecture 18

Other DSM Systems



Spring 2005
Prof. Babak Falsafi
http://www.ece.cmu.edu/~ece742

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh of University of Illinois, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University.

Readings

Chapter 10 of Culler & Singh

Reader 6:

- **Mike Galles, *Spider: A High-Speed Network Interconnect*, IEEE Micro, February 1997.**
- **W. J. Dally, *Performance Analysis of k-ary n-cube Interconnection Networks*, IEEE Trans. Computers 39(6): 775-785 (1990).**

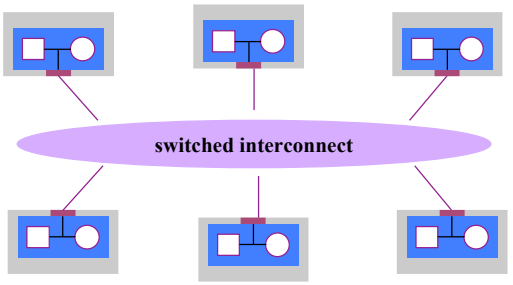
Outline

- **Page-based DSM**
  - **High overhead**
  - **Rely on Relaxed Models**
- **Token Coherence**
  - **Decouple performance from correctness**
  - **2-hop in the common case of no race (noboy is looking)**
  - **3 or more hops in case of race**

Review: Software DSM

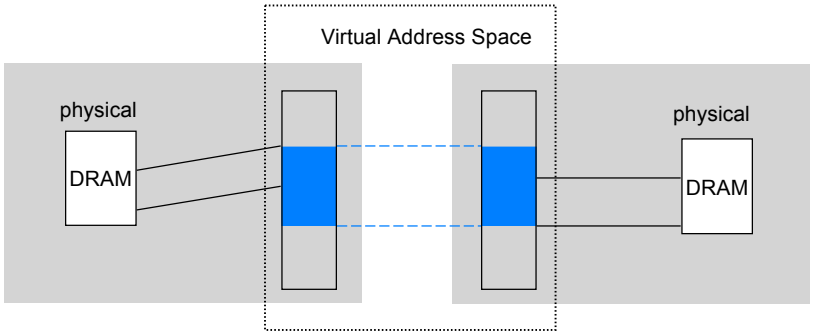
Software-based DSM provides an illusion of shared memory on a cluster

- **remote-fork the same program on each node**
- **data resides in common virtual address space**
  - » **library/kernel collude to make the shared VAS appear consistent**



Review: Page Based DSM

- Also known as Shared Virtual Memory (SVM)
- Virtual address space is shared



Review: Paged DSM/SVM Example

- P1 read virtual address x
  - Page fault
  - Allocate physical frame for page(x)
  - Request page(x) from home(x)
  - Set readable page(x)
  - Resume
- P1 write virtual address x
  - Protection fault
  - Request exclusive ownership of page(x)
  - Set writeable page(x)
  - Resume

Inside Page-Based DSM (SVM)

- ***Write-ownership*** **token protocol on VM pages**
  - **Implemented in software**
  - **Kai Li [Ivy, 1986], Paul Leach [Apollo, 1982]**
  - **System maintains per-node per-page access mode**
    - » **{shared, exclusive, no-access}**
    - » **determines local accesses allowed**
    - » **modes enforced with VM page protection (PTE bits)**

mode	load	store
shared	yes	no
exclusive	yes	yes
no-access	no	no

Write-Ownership Protocol

- **Guarantees coherence (like Hardware protocols):**
  - **Any node with any access has the *latest* copy of the page**
    - » **On any transition from** no-access**, fetch current copy of page**
  - **A node with** exclusive **access holds the *only* copy**
    - » **At most one node may hold a page in** exclusive **mode**
    - » **On transition into** exclusive**, invalidate all remote copies and set their mode to** no-access
  - **Multiple nodes may hold a page in** shared **mode**
    - » **Permits concurrent reads: every holder has the same data**
    - » **On transition into** shared **mode, invalidate the** exclusive **remote copy (if any), and set its mode to** shared **as well**
- **Can be designed for a variety of consistency models**

SVM not Practical Without Relaxed Models

- SC is slow for page-based DSM
  - » Processors cannot observe memory traffic in other nodes
  - » Even if they could, no shared bus to serialize accesses
  - » Protection granularity (pages) is too coarse
- Basic problem: the need for exclusive access to cache lines (pages) leads to *false sharing*
  - » Causes a "ping-pong effect" if multiple writers to the same page
  - » Speculation is not an option (why?)
- Solution: allow *multiple writers* to a page if their writes are "nonconflicting"
  - – Remember the motivation behind "lock elision"

Multiple Writer Protocol

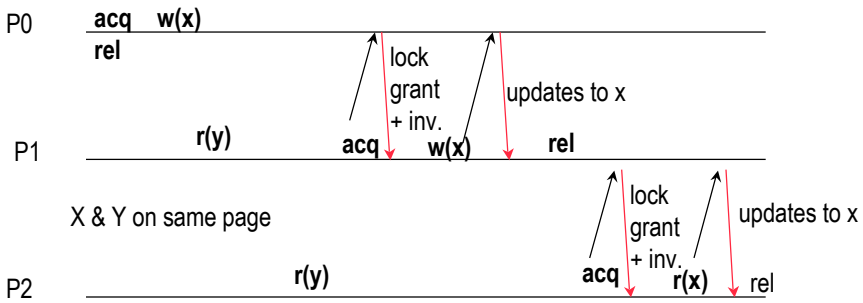
- x & y on same page P1 writes x, P2 writes y
- Don't want delays associated with constraint of exclusive access
- Allow each processor to modify its local copy of a page between synchronization points
- Make things consistent at synchronization point

Treadmarks 101

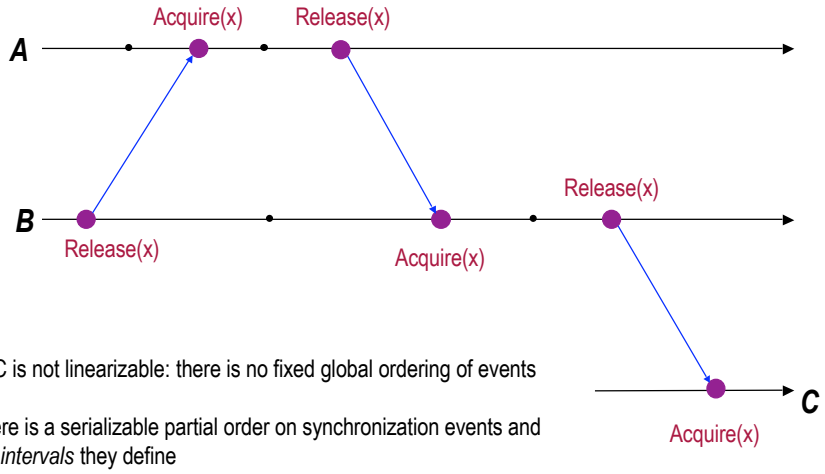
- Goal: **implement the "laziest" page-based DSM**
- **Eliminate false sharing by *multiple-writer protocol***
  - » **Capture page updates at a fine grain by "diffing"**
  - » **Propagate just the modified bytes (deltas)**
  - » **Allows merging of concurrent nonconflicting updates**
- **Propagate updates only when needed, i.e., when program uses shared locks to force consistency**
  - » **Assume program is *fully synchronized***
- ***Lazy Release Consistency* (LRC)**
  - » ***A* need not be aware of *B*'s updates except when needed to preserve potential causality...**
  - » **...with respect to shared synchronization accesses**

Lazy Release Consistency (LRC)

- **Piggyback write notices with *acquire* operations**
  - – **guarantee updates are visible on acquire**
    - » **lazier than Munin, which propagates updates on release**
  - – **implementation propagates invalidations rather than updates**



Ordering of Events in Treadmarks



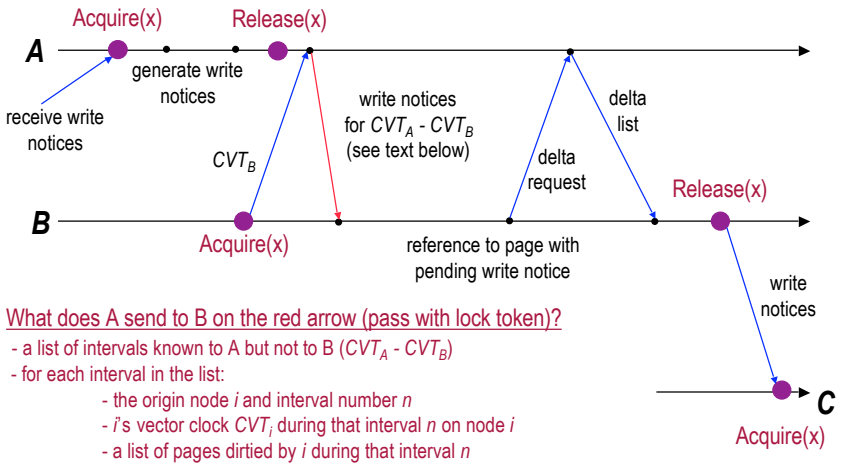
LRC is not linearizable: there is no fixed global ordering of events

There is a serializable partial order on synchronization events and the *intervals* they define

Vector Timestamps in Treadmarks

- **To maintain the partial order on intervals, each node maintains a *current vector timestamp* (CVT)**
  - **Intervals on each node are numbered 0, 1, 2...**
  - ***CVT* is a vector of length *N*, the number of nodes**
  - ***CVT[i]* is number of the last *preceding* interval on node *i***
- **Vector timestamps are updated on lock *acquire***
  - ***CVT* is passed with lock acquire request...**
  - **compared with the holder's *CVT*...**
  - **pairwise maximum *CVT* is returned with the lock**

LRC Protocol



What does A send to B on the red arrow (pass with lock token)?

- a list of intervals known to A but not to B (CVT_A - CVT_B)
- for each interval in the list:
  - the origin node *i* and interval number *n*
  - *i*'s vector clock CVT_i during that interval *n* on node *i*
  - a list of pages dirtied by *i* during that interval *n*
  - these dirty page notifications are called *write notices*

Write Notices

- **Each node must be aware of any updates to a shared page made from prior interval**
  - **Updates are tracked as sets of *write notices*.**
    - » **A *write notice* is a record that a page was dirtied during an interval**
  - **Write notices propagate with locks**
    - » **When relinquishing a lock token, the holder returns all write notices for intervals "added" to the caller's *CVT***
  - **Use page protections to collect and process write notices**
    - » **"First" store to each page is trapped...write notice created**
    - » **Pages for received write notices are invalidated on acquire**

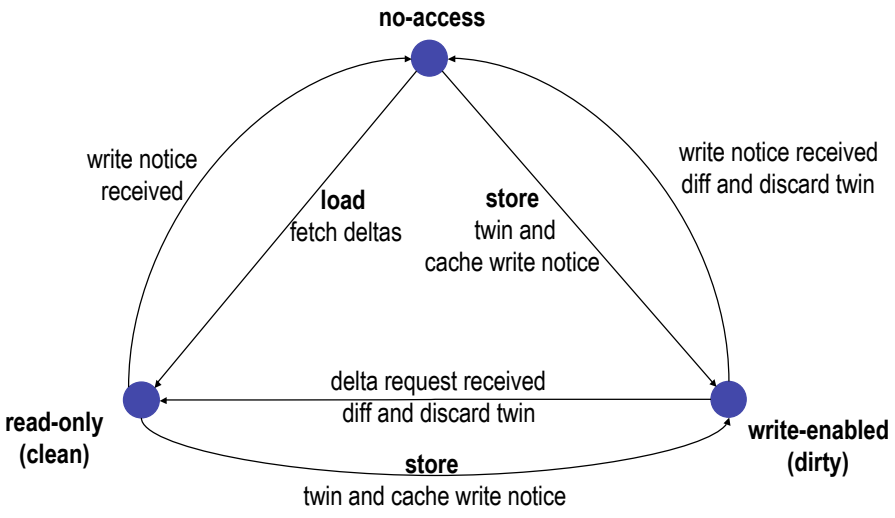
Capturing Updates (Write Collection)

- To permit multiple writers to a page, updates are captured as deltas, made by "diffing" the page
  - "Deltas" include only bytes modified during interval(s) in question
  - On "first" write, make a copy of the page (a *twin*)
    - » Mark the page dirty and write-enable the page
    - » Send write notices for all dirty pages
  - To create deltas, diff the page with its twin
    - » Record deltas, mark page clean, and disable writes
  - Cache write notices by *{node, interval, page}*
    - » cache local deltas with associated write notice

Lazy Interval/Diff Creation

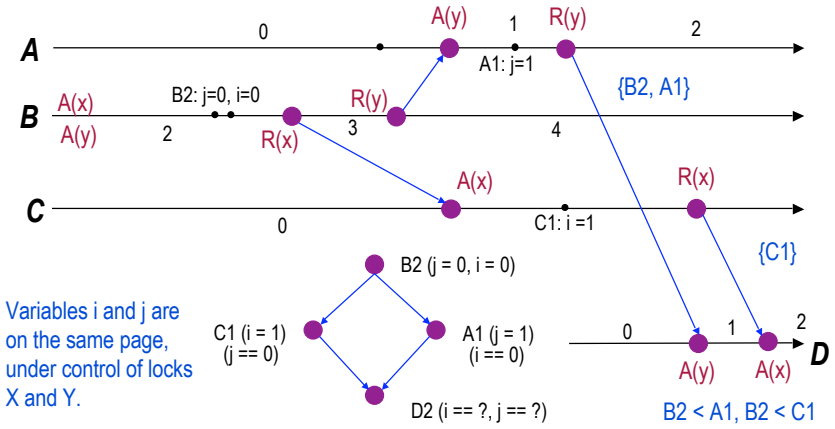
1. Don't create intervals on every acquire/release.
   - Create only if there's communication with another node
2. Delay generation of deltas (diff) until somebody asks
   - When passing a lock token, send write notices for modified pages, but leave them write-enabled
   - Diff and mark clean if somebody asks for deltas.
     - » Deltas may include updates from later intervals (e.g., under the scope of other locks)
3. Must also generate deltas if a write notice arrives.
   - Must distinguish local updates from updates made by peers.
4. Periodic garbage collection is needed.

Treadmarks Page State Transitions



Ordering Conflicting Updates

Write notices must include origin node and CVT.
Compare CVTs to order the updates.



Ordering Conflicting Updates (Cont.)

1. D receives B's write notice for the page from A
2. D receives write notices for the same page from A and C, covering their updates to the page

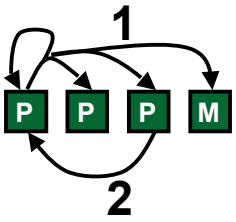
- If D then touches the page, it must fetch updates (deltas) from three different nodes (A, B, C), since it has a write notice from each of them
  - The deltas sent by A and B will both include values for j
  - The deltas sent by B and C will both include values for i
  - D must decide whose update to j happened first: B's or A's
  - D must decide whose update to i happened first: B's or C's
  - I.e., D must decide which order to apply the deltas to its copy
- D must apply these updates in vector timestamp order
- Every write notice (and delta) must be tagged with a vector timestamp

Outline

- Page-based DSM
  - High overhead
  - Rely on Relaxed Models
- Token Coherence (Slides from Milo Martin @ Penn)
  - Decouple performance from correctness
  - 2-hop in the common case of no race (noboy is looking)
  - 3 or more hops in case of race

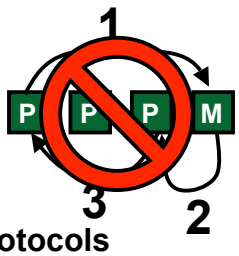
Workload Trends

- **Commercial workloads (e.g., OLTP)**
  - **Many cache-to-cache misses**
  - **Clusters of small multiprocessors**



- **Goals:**
  - Direct cache-to-cache misses (2 hops, not 3 hops)
  - **Moderate scalability**

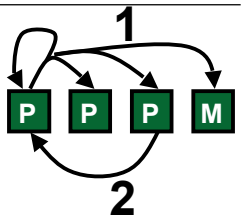
Directory Protocol



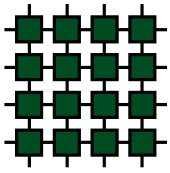
Workload trends → snooping protocols

Basic Approach

- **Fast cache-to-cache misses**
  - Broadcast with direct responses
  - **As in snooping protocols**

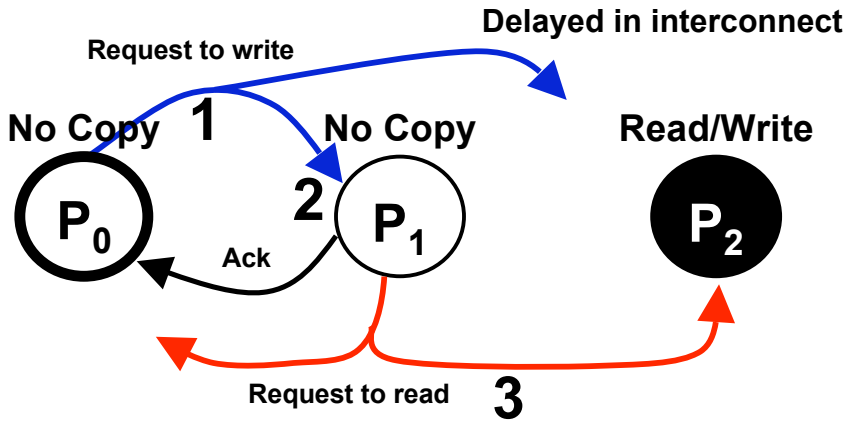


- **Fast interconnect**
  - Use unordered interconnect
  - **As in directory protocols**
  - **Low latency, high bandwidth, low cost**



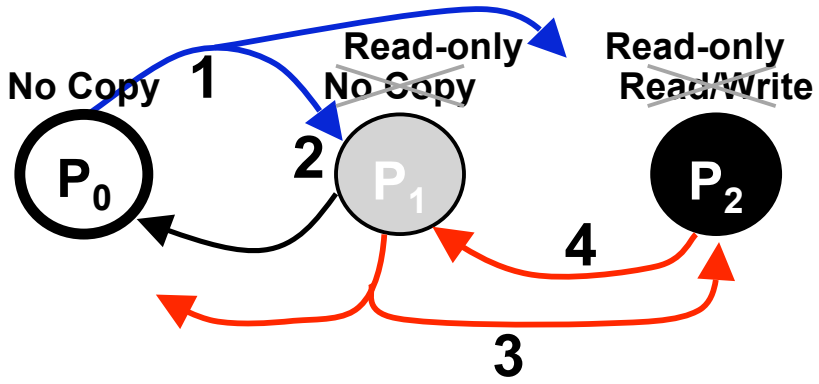
Fast & works fine with **no races**…
…but what happens in the **case of a race?**

Basic approach… but not yet correct



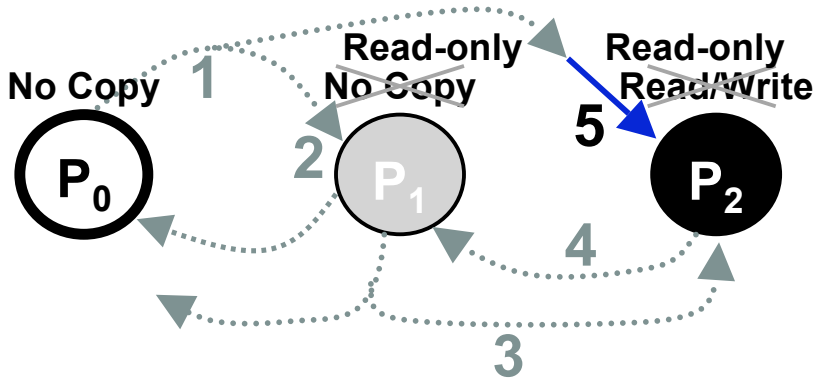
- P_0 issues a request to write (delayed to P_2)
- P_1 issues a request to read

Basic approach… but not yet correct



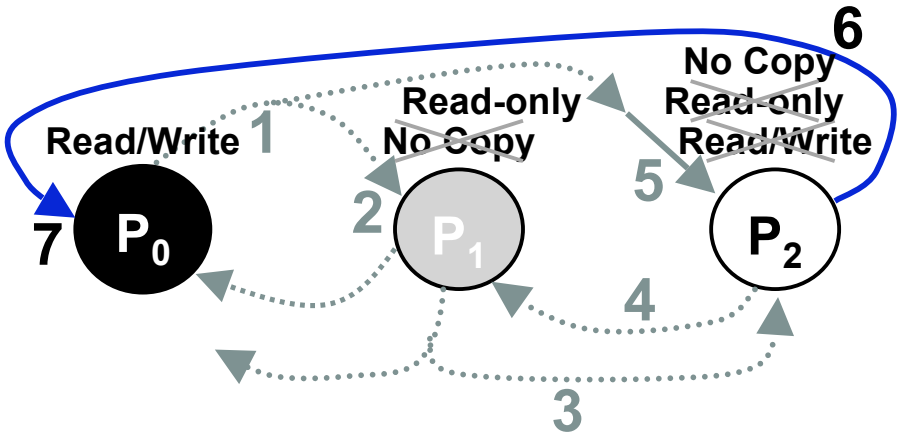
- P_2 responds with data to P_1

Basic approach… but not yet correct



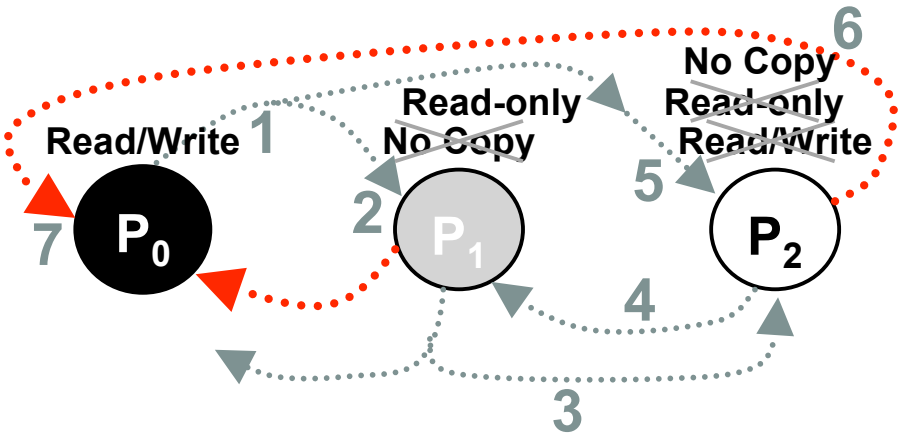
- P_0's delayed request arrives at P_2

Basic approach… but not yet correct



- P_2 responds to P_0

Basic approach… but not yet correct



Problem: P_0 and P_1 are in inconsistent states

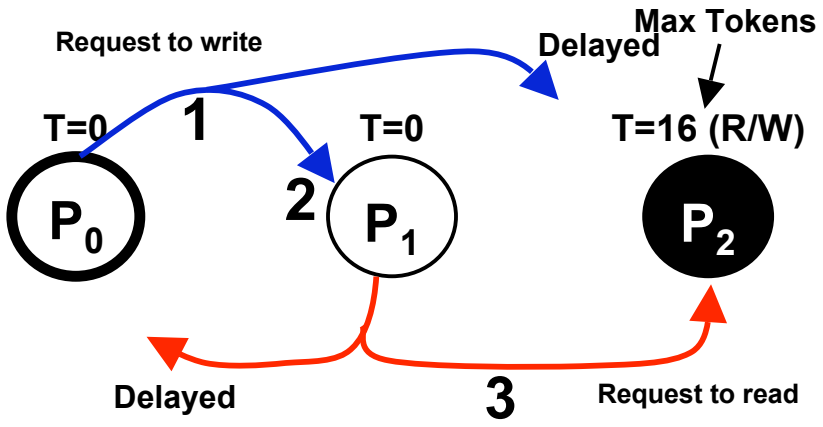
Locally "correct" operation, globally inconsistent

Enforcing Safety with Token Counting

- **Definition of safety:**
  - **All reads and writes are coherent**
  - **i.e., maintain the coherence invariant**
  - **Processor uses this property to enforce consistency**
- **Approach: token counting**
  - **Associate a fixed number of tokens for each block**
  - **At least** one token to read
  - All tokens to write
  - **Tokens in memory, caches, and messages**
- **Present rules as successive refinement**

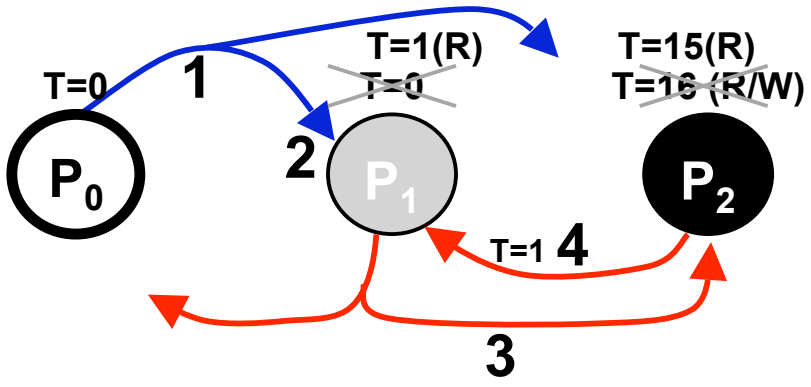
…but first, revisit example

Token Coherence Example



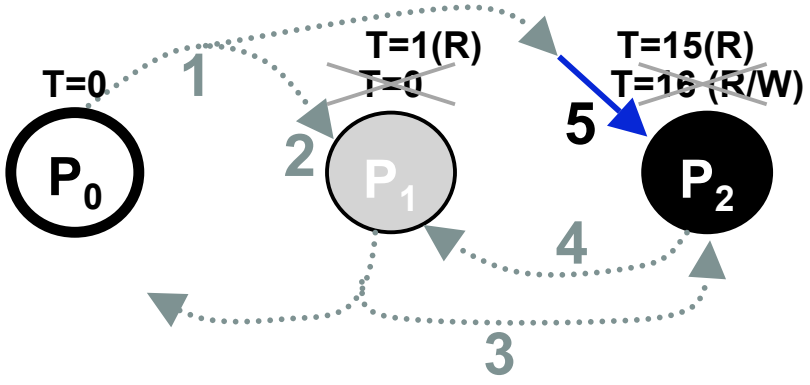
- P_0 issues a request to write (delayed to P_2)
- P_1 issues a request to read

Token Coherence Example



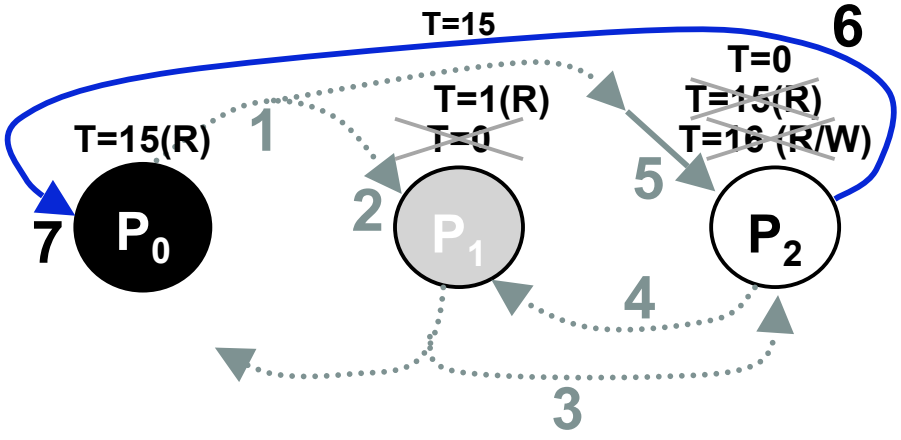
- P_2 responds with data to P_1

Token Coherence Example



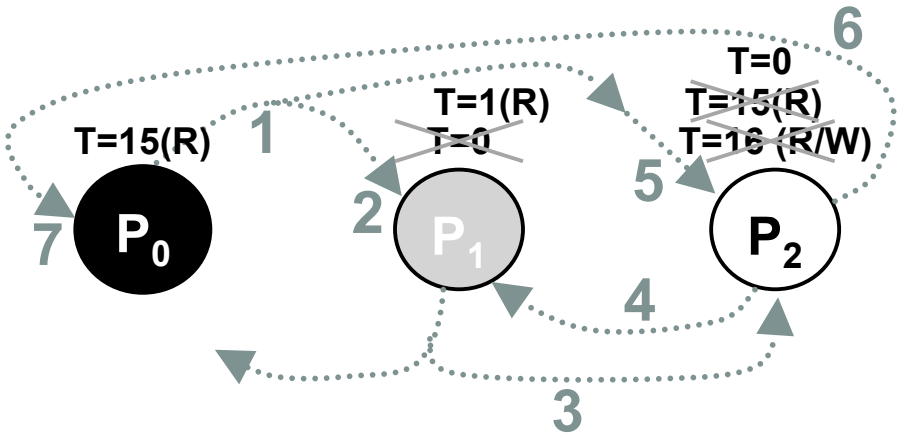
•P_0's delayed request arrives at P_2

Token Coherence Example

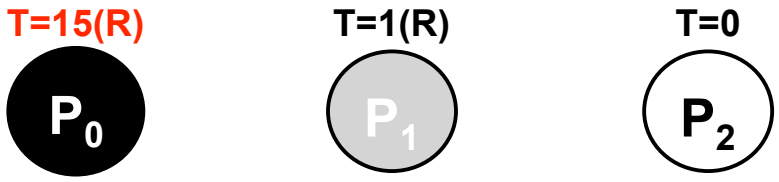


•P_2 responds to P_0

Token Coherence Example



Token Coherence Example



Now what? (P_0 still wants all tokens)

Before addressing the starvation issue, more depth on safety

Token Counting/Preservation Rules

- Conservation of Tokens
  - Tokens may not be created or destroyed
  - *One token is the owner token that is clean or dirty*
- Write Rule
  - Proc can write block only if it holds all block's tokens *and has valid data*
  - *owner token of a block is set to dirty when the block is written*
- Read Rule
  - Proc can read block only if it holds at least one token *and has valid data*
- Data Transfer Rule
  - A message with *a dirty owner token* must contain data

More Rules

- Valid-Data Bit Rule
  - **Set valid-data bit when data and token(s) arrive**
  - **Clear valid-data bit when it no longer holds any tokens**
  - **Memory sets the valid-data bit whenever it receives the** owner token
    - » **even if the message does not contain data**
- Clean Rule
  - **Whenever the memory receives the owner token, the** memory sets the owner token to clean**.**

Result: reduced traffic, encodes all MOESI states

Token Counting Overheads

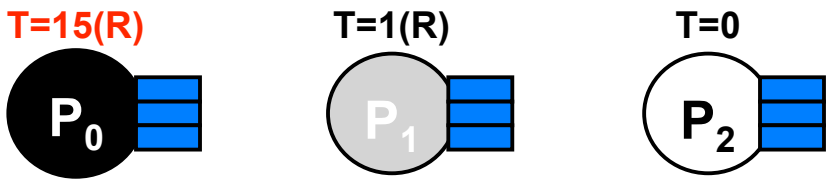
- Token storage in caches
  - 64 tokens, owner, dirty/clear = 8 bits
  - 1 byte per 64-byte block is 2% overhead
- Transferring tokens in messages
  - Data message: similar to above
  - Control message: 1 byte in 7 bytes is 13%
- Non-silent eviction overheads
  - Clean: 8-byte eviction per 72-byte data is 11%
  - Dirty: data + token message = 2%
- Token storage in memory
  - Similar to a directory protocol, but fewer bits
  - Like directory: ECC bits, directory cache

Preventing Starvation via Persistent Requests

- Definition of starvation-freedom:
  - All loads and stores must eventually complete
- Basic idea
  - Invoke after timeout (wait 4x average miss latency)
  - Send to all components
  - Each component remembers it in a small table
  - Continually redirect all tokens to requestor
  - Deactivate when complete
- As described later, not for the common case

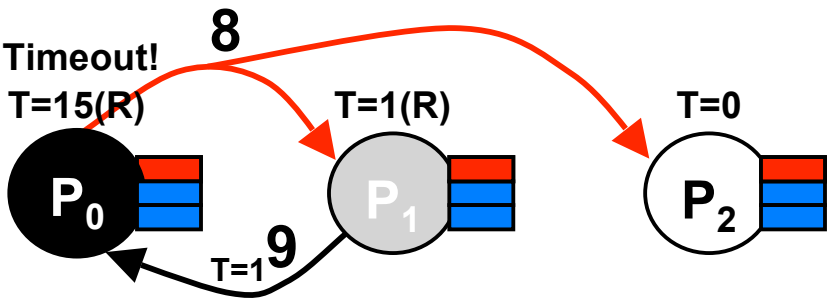
Back to the example…

Token Coherence Example



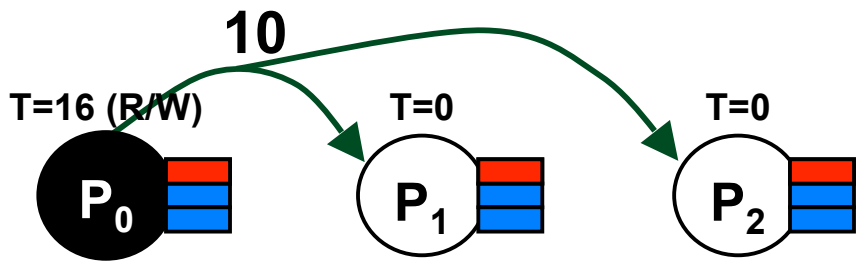
P_0 still wants all tokens

Token Coherence Example



- P_0 issues persistent request
- P_1 responds with a token

Token Coherence Example



- P_0's request completed
- P_0's deactivates persistent request

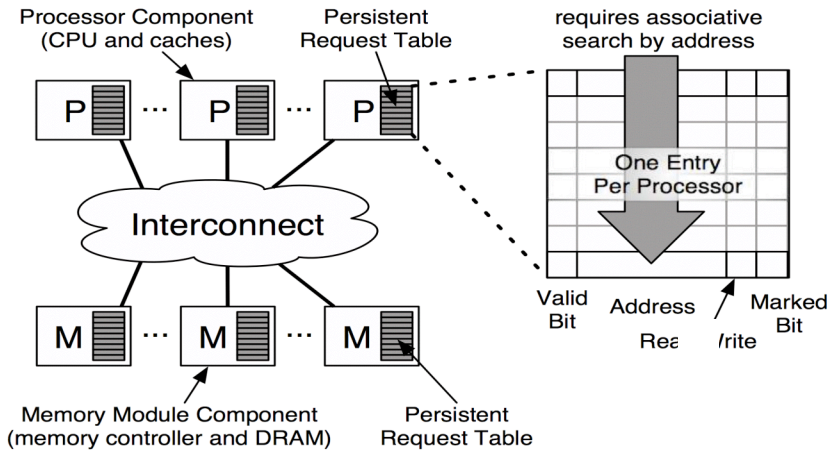
Persistent Request Arbitration

- Problem: **many processors issue persistent requests for the same block**
- Solution: **use starvation-free arbitration**
  - **Single arbiter (in dissertation)**
  - **Banked arbiters (in dissertation)**
  - **Distributed arbitration (my focus, today)**

Distributed Arbitration

- One persistent request per processor
  - One table entry per processor
- Lowest processor number has highest priority
  - Calculated per block
  - Forward all tokens for block (now and later)
- When invoking
  - "mark" all valid entries in local table
  - Don't issue another persistent request until "marked" entries are deactivated
- Based on arbitration techniques (FutureBus)

Distributed Arbitration System



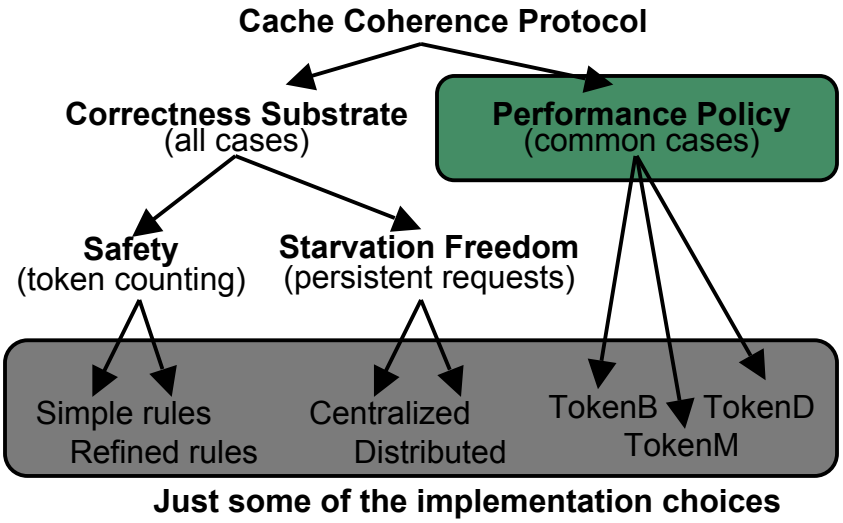
Other Persistent Request Issues

- All tokens, no data problem
  - Bounce clean owner token to memory
- Persistent read requests
  - Keep only one (non-owner) token
  - Add read/write bit to each table entry
- Preventing reordering of activation and deactivation messages
  1. Point-to-point ordering
  2. Explicit acknowledgements
  3. Acknowledgement aggregation
  4. Large sequence numbers
- Scalability of persistent requests

Performance Policies

- Correctness substrate is sufficient
  - Enforces safety with token counting
  - Prevents starvation with persistent requests
- A performance policy can do better
  - Faster, less traffic, lower overheads
  - Direct when and to whom tokens/data are sent
- With no correctness requirements
  - Even a random protocol is correct
  - Correctness substrate has final word

Decoupled Correctness and Performance



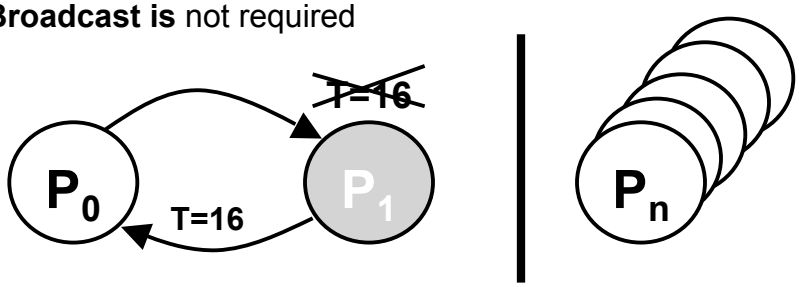
Just some of the implementation choices

TokenB Performance Policy

- Goal**: snooping without ordered interconnect**
- **Broadcast unordered** *transient requests*
  - Hints **for recipient to send tokens/data**
  - **Reissue requests once (if necessary)**
    After 2x average miss latency
  - **Substrate invokes a persistent request**
    As before, after 4x average miss latency
- **Processors & memory respond to requests**
  - **As in other MOESI protocols**
  - **Uses migratory sharing optimization**
    (as do our base-case protocols)

Beyond TokenB

- **Broadcast is** not required



- TokenD: directory-like performance policy
- TokenM
  - Multicast to a predicted *destination-set*
  - **Based on past history**
  - **Need not be correct (fall back on persistent request)**
- **Enables** larger **or** more cost-effective **systems**

TokenD Performance Policy

- Goal**: traffic & performance of directory protocol**
- **Operation**
  - **Send all requests to soft-state directory at memory**
  - **Forwards request (like directory protocol)**
  - **Processors respond as in MOESI directory protocol**
- **Reissue requests**
  - **Identical to TokenB**
- **Enhancement**
  - **Pending set of processors**
  - **Send** *completion message* **to update directory**

TokenM Performance Policy

- **Goals:**
  - **Less traffic than TokenB**
  - **Faster than TokenD**
- **Builds on TokenD, but uses prediction**
  - **Predict a destination set of processors**
  - **Soft-state directory forwards to missing processors**