Directory-based Cache Coherency

To read more...

This day's papers:

Lenoski et al, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor"

Supplementary readings:

Hennessy and Patterson, section 5.4
Molka et al, "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture"
Le et al, "IBM POWER6 Microarchitecture"

Coherency

single ‘responsible’ cache for possibly changed values

can find out who is responsible

can take over responsibility

snooping: by asking everyone

optimizations:

- avoid asking if you can remember (exclusive)
- allow serving values from cache without going through memory

Scaling with snooping

shared bus

even if not actually a bus — need to broadcast

paper last time showed us little benefit after approx. 15 CPUs

(but depends on workload)

worse with fast caches?

DASH topology

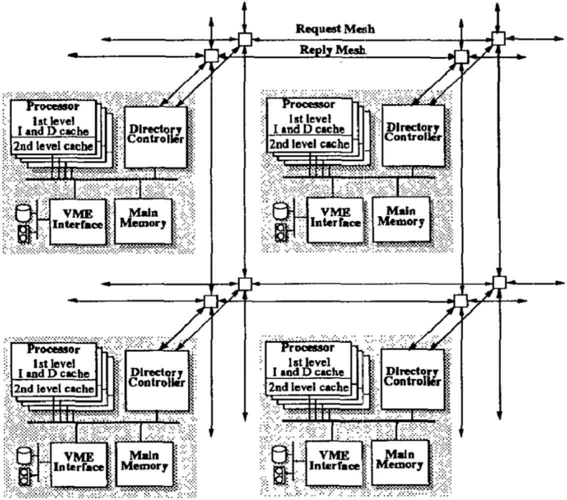


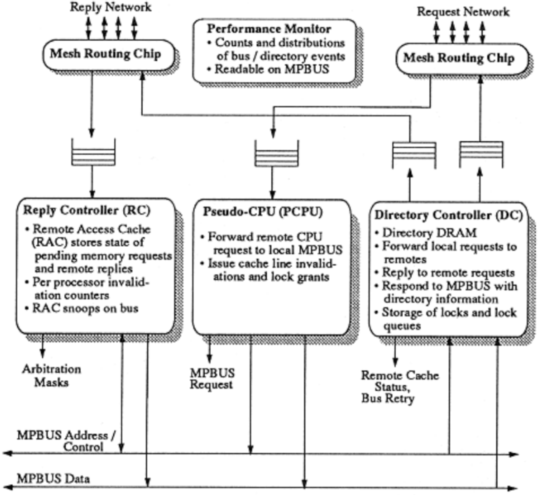
Figure 2: Block diagram of sample 2 x 2 DASH system.

DASH: the local network

shared bus with 4 processors, one memory

CPUs are unmodified

DASH: directory components



Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

directory controller pretending (1)

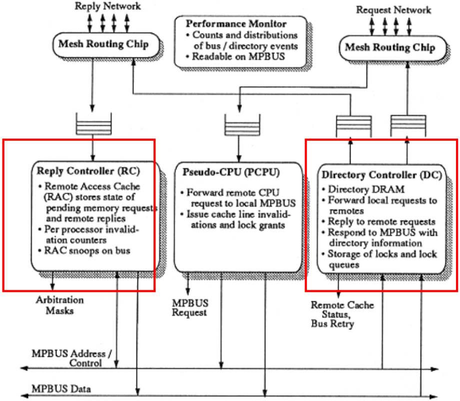

Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

directory board pretends to be another memory

… that happens to speak to remote systems

directory controller pretending (2)

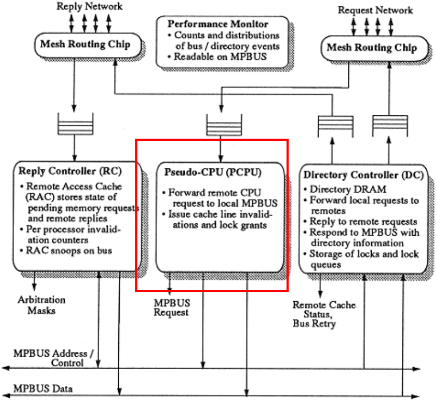

Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

directory board pretends to be another CPU

… that wants/has everything remote CPUs do

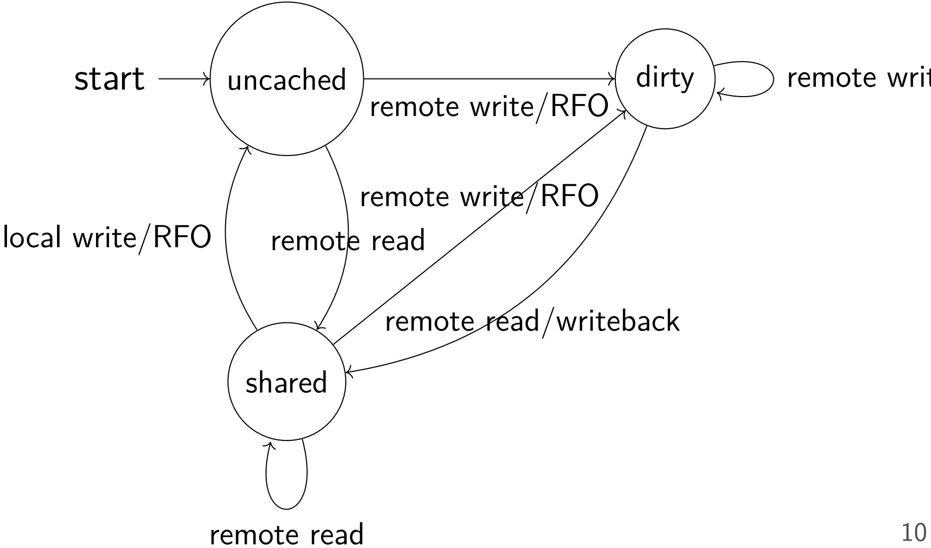
directory states

Uncached-remote value is not cached elsewhere

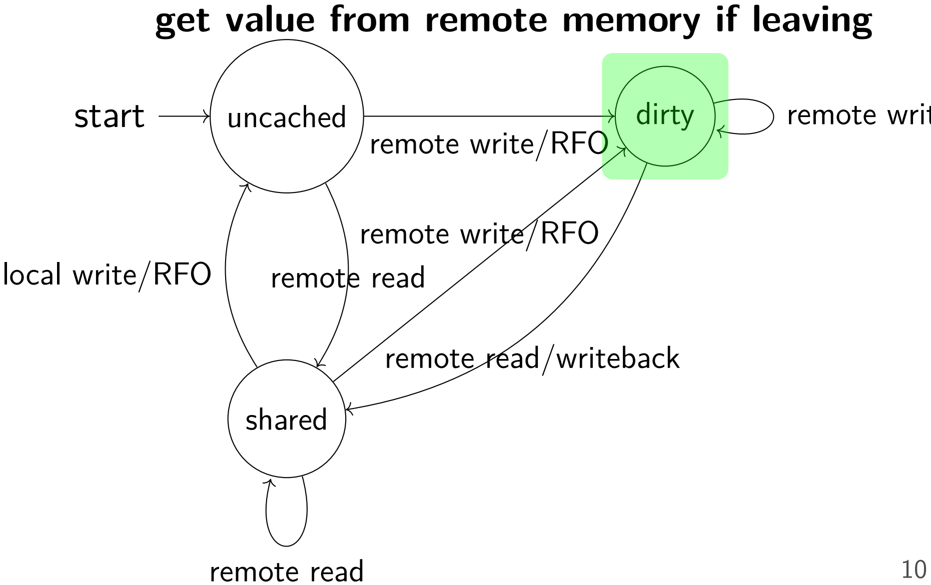
Shared-remote value is cached elsewhere, unchanged

Dirty-remote value is cached elsewhere, possibly changed

directory state transitions



directory state transitions



directory information

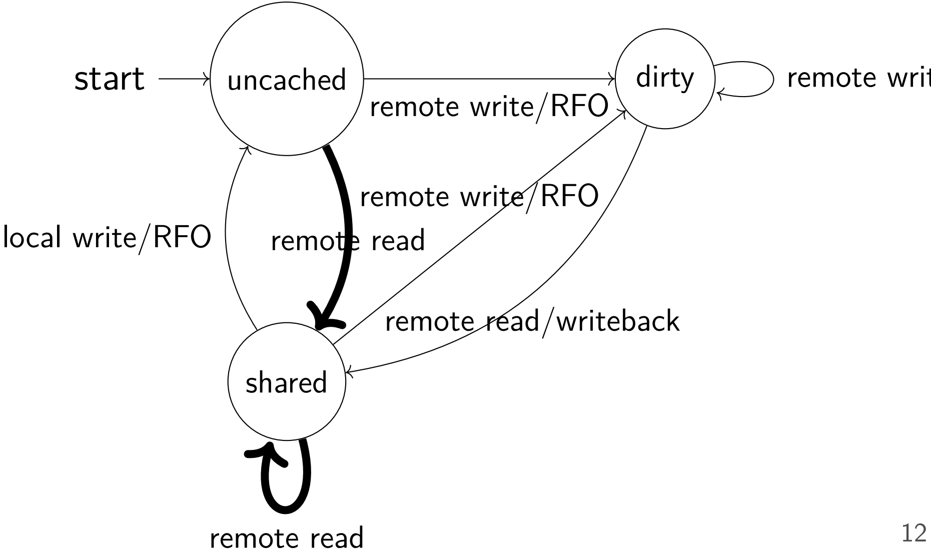
state: two bits

bit-vector for every block: which caches store it?

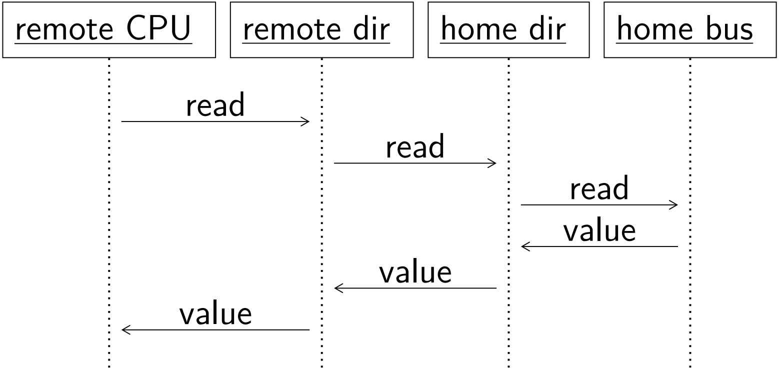
total space per cache block:

- bit vector: size = number of nodes
- state: 2 bits (to store 3 states)

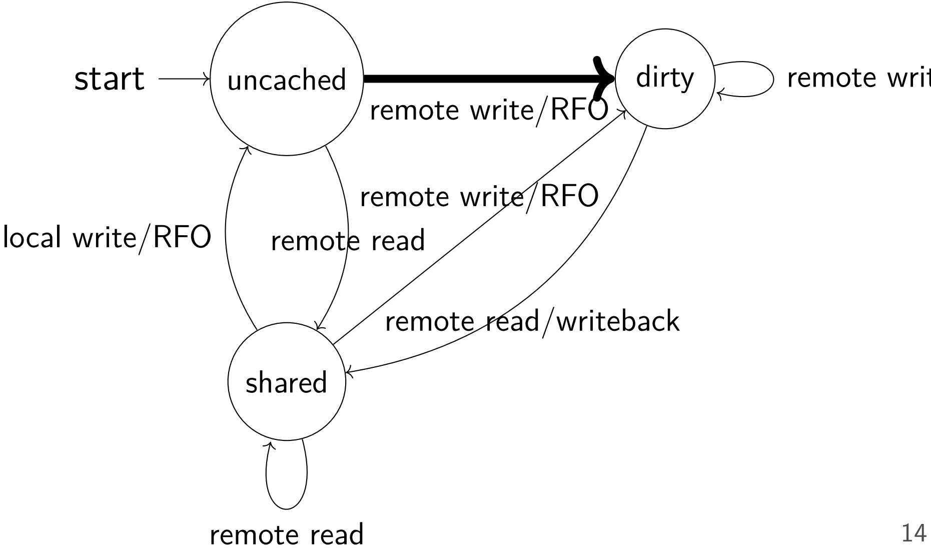
directory state transitions



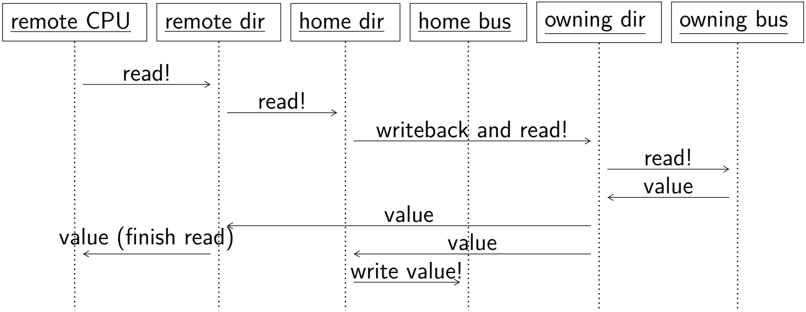
remote read: uncached/shared



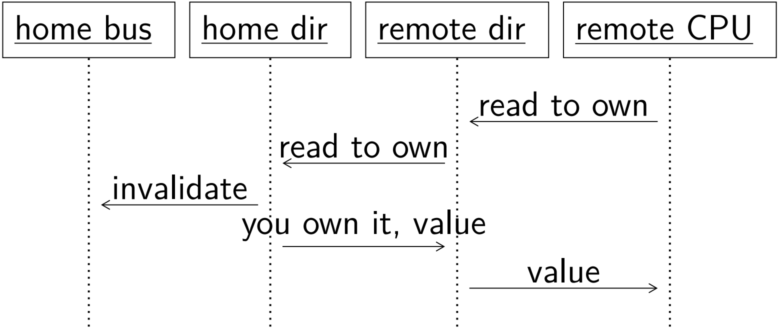
directory state transitions



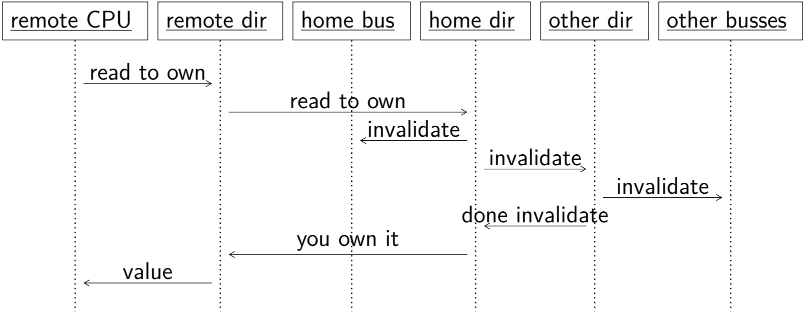
read: dirty-remote



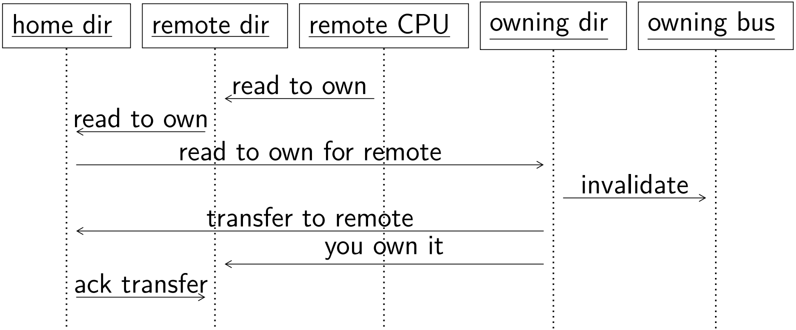
read-for-ownership: uncached



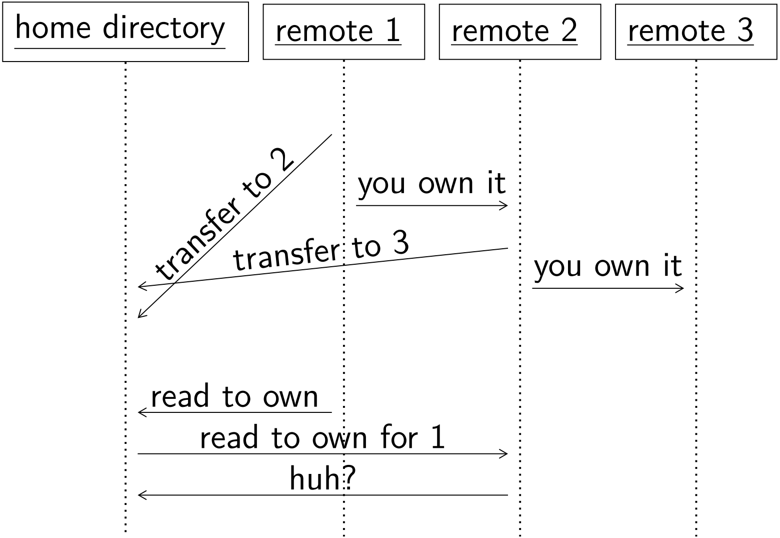
read-for-ownership: shared



read-for-ownership: dirty-remote



why the ACK



home directory

remote 1

remote 2

remote 3

transfer to 2

you own it

transfer to 3

you own it

read to own

read to own for 1

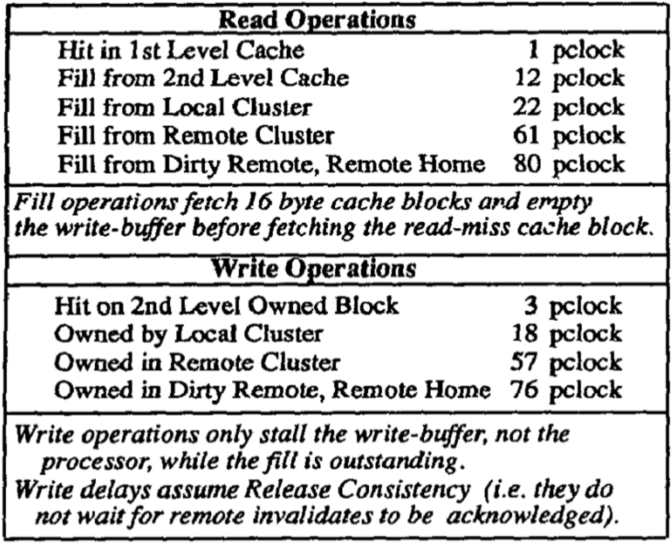
huh?

dropping cached values

directory holds worst case

a node might not have a value the directory thinks it has

NUMA



Read Operations	
Hit in 1st Level Cache	1 pclock
Fill from 2nd Level Cache	12 pclock
Fill from Local Cluster	22 pclock
Fill from Remote Cluster	61 pclock
Fill from Dirty Remote, Remote Home	80 pclock

Fill operations fetch 16 byte cache blocks and empty the write-buffer before fetching the read-miss cache block.

Write Operations	
Hit on 2nd Level Owned Block	3 pclock
Owned by Local Cluster	18 pclock
Owned in Remote Cluster	57 pclock
Owned in Dirty Remote, Remote Home	76 pclock

Write operations only stall the write-buffer, not the processor, while the fill is outstanding.

Write delays assume Release Consistency (i.e. they do not wait for remote invalidates to be acknowledged).

Big machine cache coherency?

Cray T3D (1993) — up to 256 nodes with 64MB of RAM each

32-byte cache blocks

8KB data cache per processor

no caching of remote memories (like T3E)

hypothetical today: adding caching of remote memories

Directory overhead: adding to T3D

T3D: 256 nodes, 64MB/node

32 bytes cache blocks: 2M cache blocks/node

256 bits for bit vector + 2 bits for state = 258 bits/cache block

64.5 MB/node in overhead alone

Decreasing overhead: sparse directory

most memory not in any cache

only store entries for cached items

worst case?

- 8KB cache/node * 256 nodes = 2MB cached

2MB: 64K cache blocks

64K cache blocks * 258 bits/block ≈ 2 MB overhead/node

Decreasing overhead: distributed directory

most memory only stored in small number of caches

store linked list of nodes with item cached

each node has pointer to next entry on linked list

around 80 KB overhead/node

… but hugely more complicated protocol

Real directories: Intel Haswell-EP

2 bits/cache line — in-memory

- .4% overhead
- stored in ECC bits — loss of reliability

14KB cache for directory entries

cached entries have bit vector (who might have this?)

otherwise — broadcast instead

Real directories: IBM POWER6

1 bit/cache line — possibly remote or not

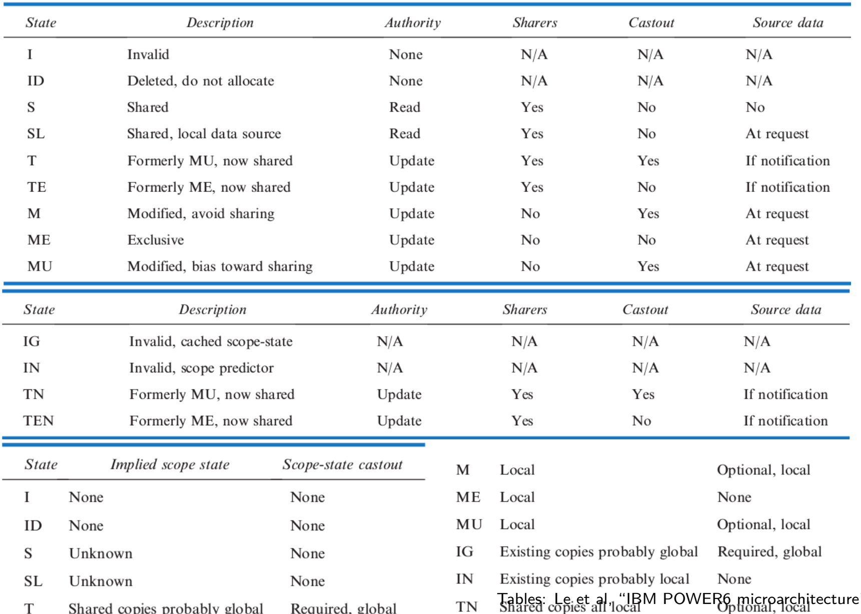
.1% overhead

stored in ECC bits — loss of reliability

extra bit for each cache line

no storage of remote location of line

Aside: POWER6 cache coherency



State	Description	Authority	Sharers	Castout	Source data
I	Invalid	None	N/A	N/A	N/A
ID	Deleted, do not allocate	None	N/A	N/A	N/A
S	Shared	Read	Yes	No	No
SL	Shared, local data source	Read	Yes	No	At request
T	Formerly MU, now shared	Update	Yes	Yes	If notification
TE	Formerly ME, now shared	Update	Yes	No	If notification
M	Modified, avoid sharing	Update	No	Yes	At request
ME	Exclusive	Update	No	No	At request
MU	Modified, bias toward sharing	Update	No	Yes	At request

State	Description	Authority	Sharers	Castout	Source data
IG	Invalid, cached scope-state	N/A	N/A	N/A	N/A
IN	Invalid, scope predictor	N/A	N/A	N/A	N/A
TN	Formerly MU, now shared	Update	Yes	Yes	If notification
TEN	Formerly ME, now shared	Update	Yes	No	If notification

State	Implied scope state	Scope-state castout
I	None	None
ID	None	None
S	Unknown	None
SL	Unknown	None
T	Shared copies probably global	Required, global
M	Local	Optional, local
ME	Local	None
MU	Local	Optional, local
IG	Existing copies probably global	Required, global
IN	Existing copies probably local	None
TN	Shared copies all local	Optional, local

Tables: Le et al., “IBM POWER6 microarchitecture”

software distributed shared memory

can use page table mechanisms to share memory

implement MSI-like protocol in software

using pages instead of cache blocks

writes: read-only bit in page table

reads: remove from page table

really an OS topic

handling pending invalidations

can get requests while waiting to finish request

could queue locally

instead — negative acknowledgement

retry and timeout

what is release consistency?

“release” does not complete until prior operations happen

idea: everything sensitive done in (lock) acquire/release

example inconsistency

possibly if you don't lock:

- writes in any order (from different nodes)
- reads in any order

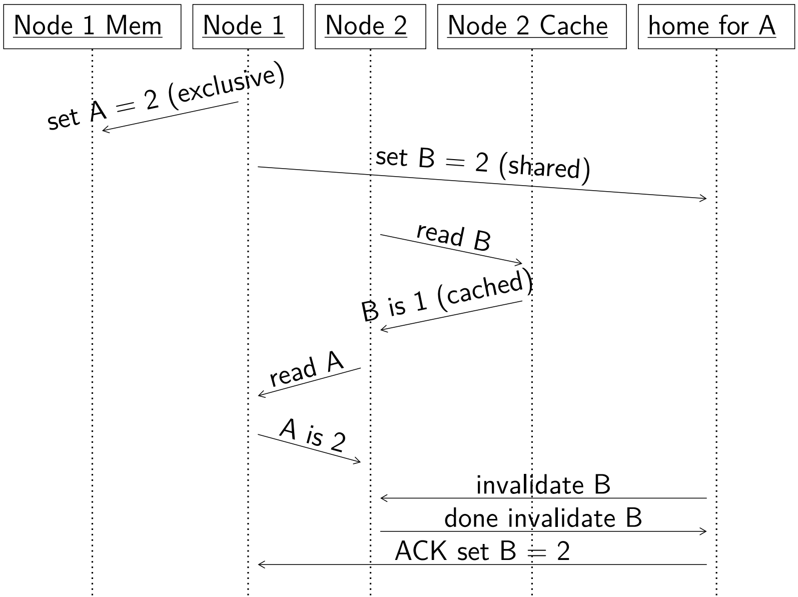
simple inconsistencies

starting: shared $A = B = 1$

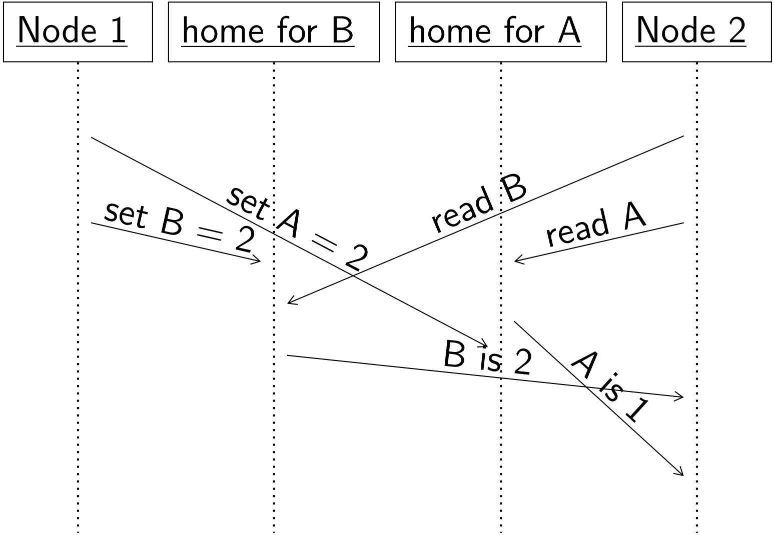
Node 1	Node 2
$A = 2$	$x = B$
$B = 2$	$y = A$

possible for $x = 2, y = 1$

timeline: out-of-order writes



timeline: out-of-order reads



cost of consistency

wait for each read before starting next one

wait for ACK for each write that needs invalidations

release consistency utility

acquire lock — wait until someone else's release finished

release lock — your operations are visible

programming discipline: always lock

inconsistency

gets more complicated with more nodes

very difficult to reason about

topic of next Monday's papers

implementing the release/fence

need to wait for all invalidations to actually complete

if a full fence, need to make sure reads complete, too

otherwise, let them execute as fast as possible

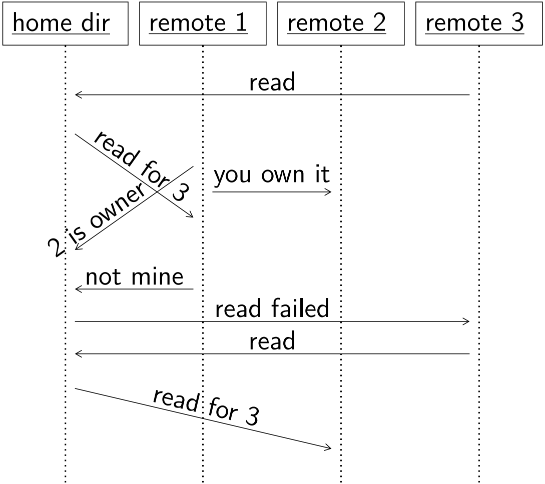
cost of implementing sequential consistency

better consistency would stop pipelining of reads/writes

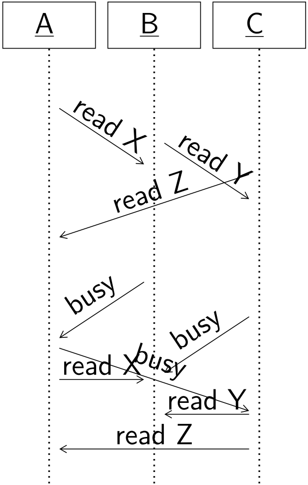
recall: big concern of, e.g, T3E

dramatically increased latency

“livelock”

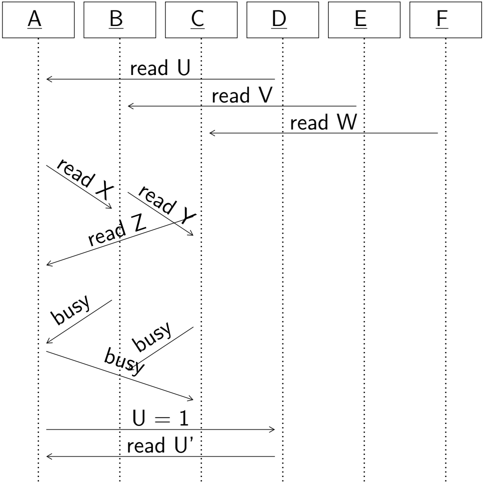

home dir
remote 1
remote 2
remote 3
read
read for 3
you own it
2 is owner
not mine
read failed
read
read for 3

deadlock



buffer for one pending request
everyone out of space!

deadlock: larger buffer



Example: two buffered requests
everyone out of space!

mitigation 1: multiple networks

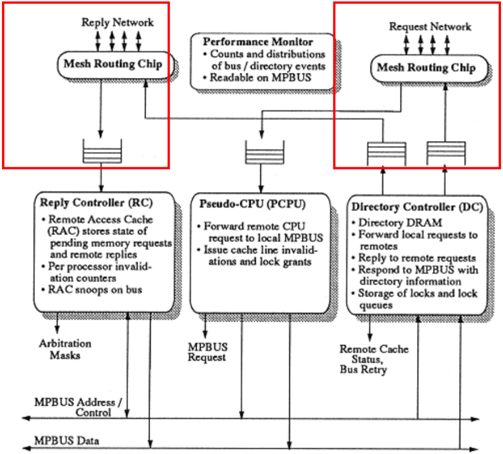
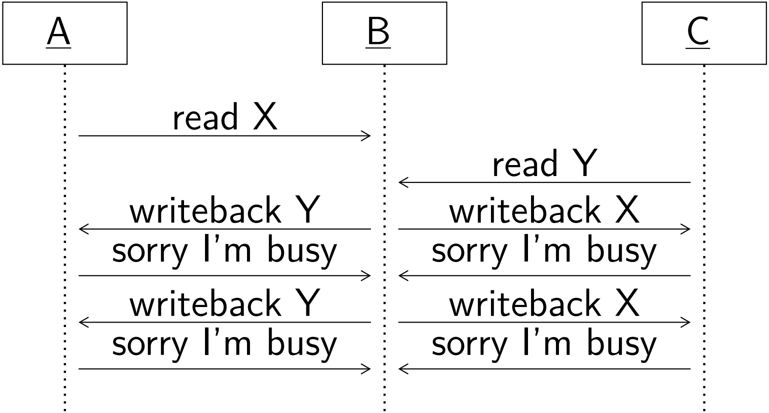

Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

deadlock in requests



A, C waiting for ACK for it's operation

out of space for new operations

deadlock detection

negative acknowledgements

timeout for retries

takes too long — enter deadlock mitigation mode

refuse to accept new requests that generate other requests

deadlock response

validation: what they did

generated lots of test cases

deliberately varied order of operations a lot

better techniques for correctness (1)

techniques from program verification

usually on abstract description of protocol

challenge: making sure logic gate implementation matches

better techniques for correctness (2)

specialized programming languages for writing coherency protocols

still an area of research

efficiency of synchronization

special synchronization primitive — queue-based lock

problem without: hot spots

contended lock with read-modify-write

best case: processors check value in cache, wait for invalidation

on invalidation: every processor tries to read-for-ownership the lock

one succeeds, but tons of network traffic

other directions in cache coherency

identify access patterns — write-once, producer/consumer, etc.

can handle those better

pattern: processors read, then write value a lot?

optimization: treat those reads as read-exclusives

new states in coherency protocol to track pattern

next week: focus group

last approx 20 minutes of class: consultant from CTE (Center for Teaching Excellence)

hope to get actionable feedback on how I can improve this class (this semester and in the future)

please stay, but I won't know

next time: papers

Adve and Gharachorloo. “Shared Memory Consistency Models: A Tutorial”

Section 1 (only) of Boehm and Adve, “Foundations of the C++ Concurrency Memory Model”