

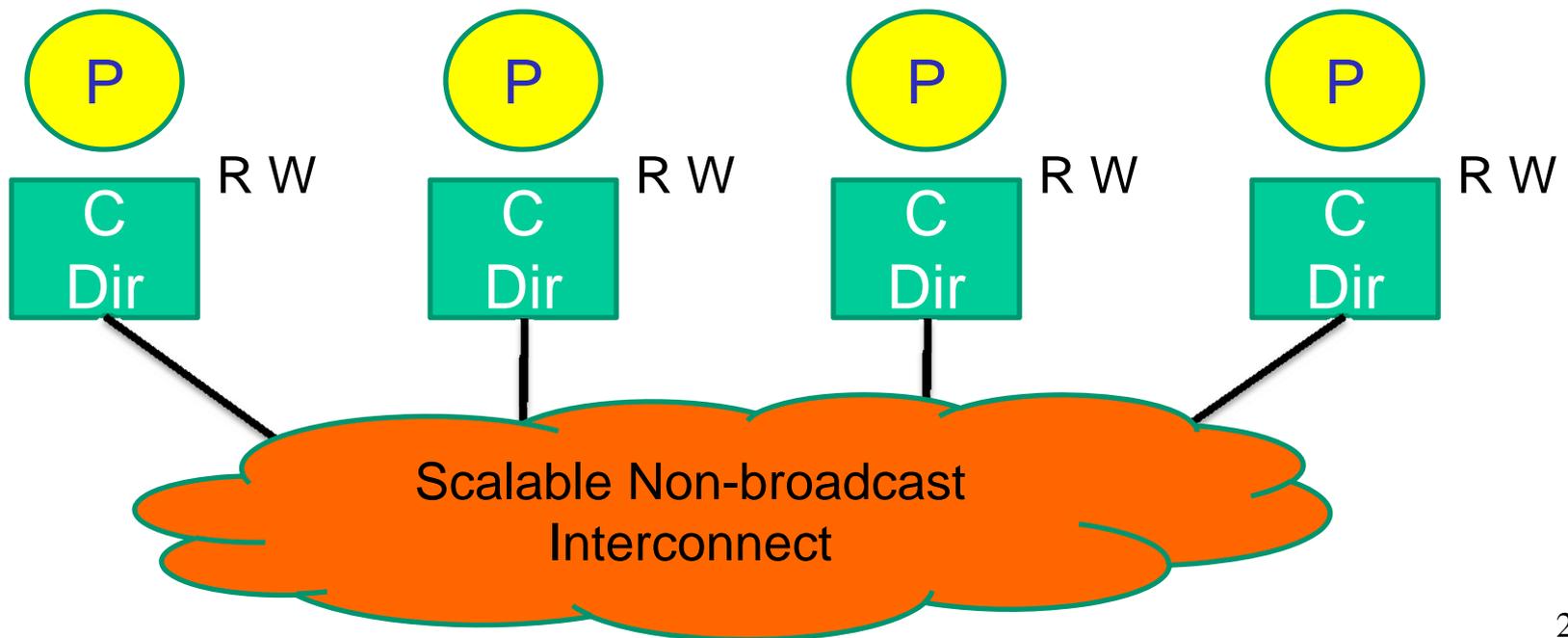
Lecture 8: Eager Transactional Memory

- Topics: implementation details of eager TM, various TM pathologies

“Eager” Overview

Topics:

- Logs
- Log optimization
- Conflict examples
- Handling deadlocks
- Sticky scenarios
- Aborts/commits/parallelism



“Eager” Implementation (Based Primarily on LogTM)

- A write is made permanent immediately (we do not wait until the end of the transaction)
- Can't lose the old value (in case this transaction is aborted) – hence, before the write, we copy the old value into a log (the log is some space in virtual memory -- the log itself may be in cache, so not too expensive)

This is eager versioning

Versioning

- Every overflowed write first requires a read and a write to log the old value – the log is maintained in virtual memory and will likely be found in cache
- Aborts are uncommon – typically only when the contention manager kicks in on a potential deadlock; the logs are walked through in reverse order
- If a block is already marked as being logged (wr-set), the next write by that transaction can avoid the re-log
- Log writes can be placed in a write buffer to reduce contention for L1 cache ports

Conflict Detection and Resolution

- Since Transaction-A's writes are made permanent rightaway, it is possible that another Transaction-B's rd/wr miss is re-directed to Tr-A
- At this point, we detect a conflict (neither transaction has reached its end, hence, *eager conflict detection*): two transactions handling the same cache line and at least one of them does a write
- One solution: requester stalls: Tr-A sends a NACK to Tr-B; Tr-B waits and re-tries again; hopefully, Tr-A has committed and can hand off the latest cache line to B
→ neither transaction needs to abort

Deadlocks

- Can lead to deadlocks: each transaction is waiting for the other to finish
- Need a separate (hw/sw) contention manager to detect such deadlocks and force one of them to abort

Tr-A
write X
...
read Y

Tr-B
write Y
...
read X

- Alternatively, every transaction maintains an “age” and a young transaction aborts and re-starts if it is keeping an older transaction waiting and itself receives a nack from an older transaction

Block Replacement

- If a block in a transaction's rd/wr-set is evicted, the data is written back to memory if necessary, but the directory continues to maintain a “sticky” pointer to that node (subsequent requests have to confirm that the transaction has committed before proceeding)
- The sticky pointers are lazily removed over time (commits continue to be fast)

Paper on TM Pathologies

- LL: lazy versioning, lazy conflict detection, committing transaction wins conflicts
- EL: lazy versioning, eager conflict detection, requester succeeds and others abort
- EE: eager versioning, eager conflict detection, requester stalls

Pathology 1: Friendly Fire

- Two conflicting transactions that keep aborting each other
- Can do exponential back-off to handle livelock
- Fixable by doing requester stalls?

- VM: any
- CD: eager
- CR: requester wins

Pathology 2: Starving Writer

- A writer has to wait for the reader to finish – but if more readers keep showing up, the writer is starved (note that the directory allows new readers to proceed by just adding them to the list of sharers)

- VM: any
- CD: eager
- CR: requester stalls

Pathology 3: Serialized Commit

- If there's a single commit token, transaction commit is serialized

- VM: lazy
- CD: lazy
- CR: any

- There are ways to alleviate this problem

Pathology 4: Futile Stall

- A transaction is stalling on another transaction that ultimately aborts and takes a while to reinstate old values

- VM: any
- CD: eager
- CR: requester stalls

Pathology 5: Starving Elder

- Small successful transactions can keep aborting a large transaction
- The large transaction can eventually grab the token and not release it until after it commits

- VM: lazy
- CD: lazy
- CR: committer wins

Pathology 6: Restart Convoy

- A number of similar (conflicting) transactions execute together – one wins, the others all abort – shortly, these transactions all return and repeat the process

- VM: lazy
- CD: lazy
- CR: committer wins

Pathology 7: Dueling Upgrades

- If two transactions both read the same object and then both decide to write it, a deadlock is created
- Exacerbated by the Futile Stall pathology
- Solution?

- VM: eager
- CD: eager
- CR: requester stalls

Four Extensions

- Predictor: predict if the read will soon be followed by a write and acquire write permissions aggressively
- Hybrid: if a transaction believes it is a Starving Writer, it can force other readers to abort; for everything else, use requester stalls
- Timestamp: In the EL case, requester wins only if it is the older transaction (handles Friendly Fire pathology)
- Backoff: in the LL case, aborting transactions invoke exponential back-off to prevent convoy formation

Title

- Bullet