Lecture 10: TM Implementations

• Topics: wrap-up of eager implementation (LogTM), scalable lazy implementation
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)

- This means that if some other transaction attempts a read, the latest value is returned and the memory may also be updated with this latest value

- 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 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                 Tr-B
  write X             write Y
  ...                 ...
  read 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)
Scalable Algorithm – Lazy Implementation

• Probe your write-set to see if it is your turn to write (helps serialize writes)

• Let others know that you don’t plan to write (thereby allowing parallel commits to unrelated directories)

• Mark your write-set (helps hide latency)

• Probe your read-set to see if previous writes have completed

• Validation is now complete – send the actual commit message to the write set
Example

Rd X
Wr X

P1

Rd Y
Wr Z

P2

NS: 1
D: X Z

NS: 1
D: Y
Example

P1 (Rd X, Wr X) to TID Vendor (TID=1)
P2 (Rd Y, Wr Z) to TID Vendor (TID=2)

NS: 1
D: X Z

NS: 1
D: Y
Example

P2 sends the same set of probes/notifications

Probe to write-set to see if it can proceed

No writes here. I’m done with you.
Example

Mark messages are hiding the latency for the subsequent commit
Example

Probe read set and make sure they're done

TID=1
Rd X
Wr X

TID=2
Rd Y
Wr Z

Keep probing and waiting

NS: 1
D: X Z

NS: 3
D: Y
Example

TID Vendor

TID=1

P1

TID=2

P2

Keep probing and waiting

Rd X
Wr X

Invalidate sharers; May cause aborts

NS: 2
D: X Z

Commit

NS: 3
D: Y

Rd Y
Wr Z

TID=1

P1

TID=2
Example

P1: TID=1
Rd X
Wr X

P2: TID=2
Rd Y
Wr Z

NS: 2
D: X Z

NS: 3
D: Y

TID Vendor

P2: Probe finally successful.
Can mark Z.
Will then check read-set.
Then proceed with commit
Title

• Bullet