Lecture: Transactional Memory, Networks

- Topics: TM implementations, on-chip networks

Summary of TM Benefits

- As easy to program as coarse-grain locks
- Performance similar to fine-grain locks
- Avoids deadlock

Design Space

- Data Versioning
  - Eager: based on an undo log
  - Lazy: based on a write buffer

- Conflict Detection
  - Optimistic detection: check for conflicts at commit time (proceed optimistically thru transaction)
  - Pessimistic detection: every read/write checks for conflicts (reduces work during commit)

“Lazy” Implementation

- An implementation for a small-scale multiprocessor with a snooping-based protocol
- Lazy versioning and lazy conflict detection
- Does not allow transactions to commit in parallel

“Lazy” Implementation

- When a transaction issues a read, fetch the block in read-only mode (if not already in cache) and set the rd-bit for that cache line

- When a transaction issues a write, fetch that block in *read-only* mode (if not already in cache), set the wr-bit for that cache line and make changes in cache

- If a line with wr-bit set is evicted, the transaction must be aborted (or must rely on some software mechanism to handle saving overflowed data)

“Lazy” Implementation

- When a transaction reaches its end, it must now make its writes permanent

- A central arbiter is contacted (easy on a bus-based system), the winning transaction holds on to the bus until all written cache line addresses are broadcasted (this is the commit) (need not do a writeback until the line is evicted – must simply invalidate other readers of these cache lines)

- When another transaction (that has not yet begun to commit) sees an invalidation for a line in its rd-set, it realizes its lack of atomicity and aborts (clears its rd- and wr-bits and re-starts)

“Lazy” Implementation

- Lazy versioning: changes are made locally – the “master copy” is updated only at the end of the transaction
- Lazy conflict detection: we are checking for conflicts only when one of the transactions reaches its end
- Aborts are quick (must just clear bits in cache, flush pipeline and reinstate a register checkpoint)
- Commit is slow (must check for conflicts, all the coherence operations for writes are deferred until transaction end)
- No fear of deadlock/livelock – the first transaction to acquire the bus will commit successfully
- Starvation is possible – need additional mechanisms

“Lazy” Implementation – Parallel Commits

- Writes cannot be rolled back – hence, before allowing two transactions to commit in parallel, we must ensure that they do not conflict with each other

- One possible implementation: the central arbiter can collect signatures from each committing transaction (a compressed representation of all touched addresses)

- Arbiter does not grant commit permissions if it detects a possible conflict with the rd-wr-sets of transactions that are in the process of committing

- The “lazy” design can also work with directory protocols

“Eager” Implementation

- 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

“Eager” Implementation

- 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

“Eager” Implementation

- 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

“Eager” Implementation

- Note that if Tr-B is doing a write, it may be forced to stall because Tr-A may have done a read and does not want to invalidate its cache line just yet

- If new reading transactions keep emerging, Tr-B may be starved – again, need other sw/hw mechanisms to handle starvation

- Commits are inexpensive (no additional step required); Aborts are expensive, but rare (must reinstate data from logs)

Other Issues

- Nesting: when one transaction calls another
  - flat nesting: collapse all nested transactions into one large transaction
  - closed nesting: inner transaction's rd-wr set are included in outer transaction's rd-wr set on inner commit; on an inner conflict, only the inner transaction is re-started
  - open nesting: on inner commit, its writes are committed and not merged with outer transaction's commit set

- What if a transaction performs I/O?
- What if a transaction overflows out of cache?

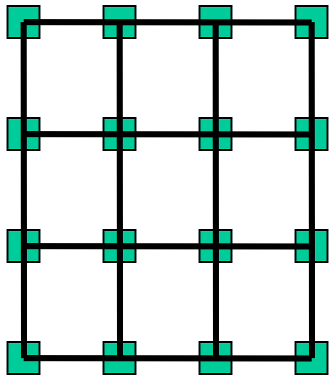
Useful Rules of Thumb

- Transactions are often short – more than 95% of them will fit in cache
- Transactions often commit successfully – less than 10% are aborted
- 99.9% of transactions don't perform I/O
- Transaction nesting is not common
- Amdahl's Law again: optimize the common case!

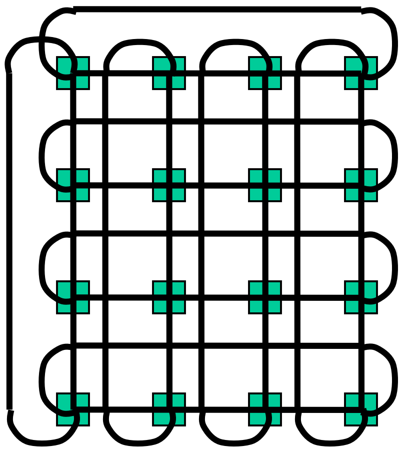
Discussion

- “Eager” optimizes the common case and does not waste energy when there’s a potential conflict
- TM implementations require relatively low hardware support
- Multiple commercial examples: Sun Rock, AMD ASF, IBM BG/Q, Intel Haswell

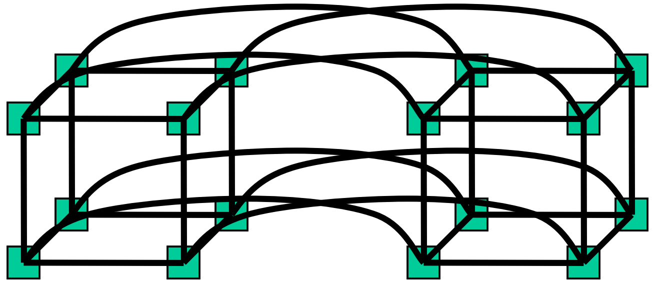
Network Topology Examples



Grid



Torus



Hypercube

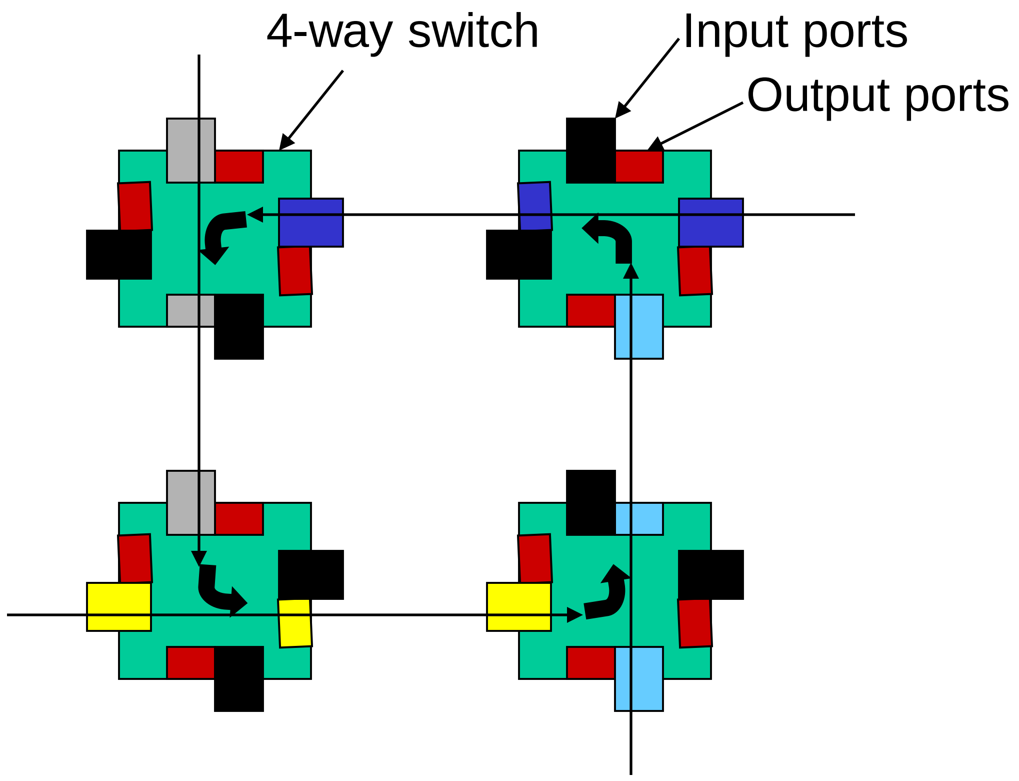
Routing

- Deterministic routing: given the source and destination, there exists a unique route
- Adaptive routing: a switch may alter the route in order to deal with unexpected events (faults, congestion) – more complexity in the router vs. potentially better performance
- Example of deterministic routing: dimension order routing: send packet along first dimension until destination co-ord (in that dimension) is reached, then next dimension, etc.

Deadlock

- Deadlock happens when there is a cycle of resource dependencies – a process holds on to a resource (A) and attempts to acquire another resource (B) – A is not relinquished until B is acquired

Deadlock Example



🟦 Packets of message 1

⬜ Packets of message 2

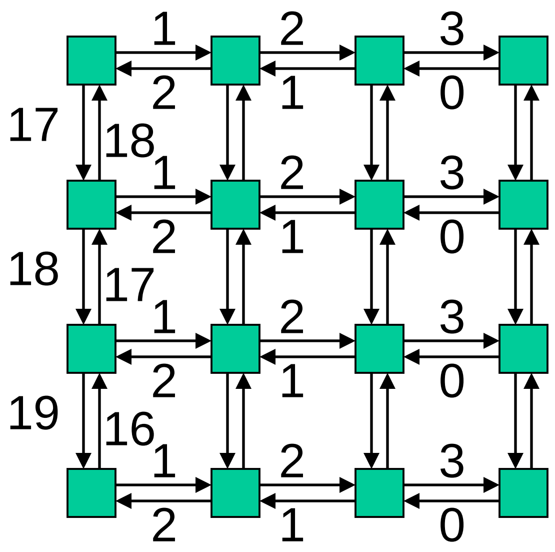
🟨 Packets of message 3

🟦 Packets of message 4

Each message is attempting to make a left turn – it must acquire an output port, while still holding on to a series of input and output ports

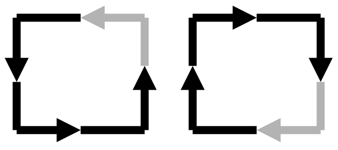
Deadlock-Free Proofs

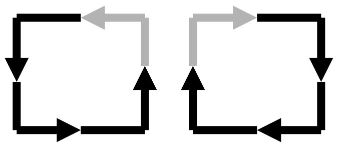
- Number edges and show that all routes will traverse edges in increasing (or decreasing) order – therefore, it will be impossible to have cyclic dependencies
- Example: k-ary 2-d array with dimension routing: first route along x-dimension, then along y

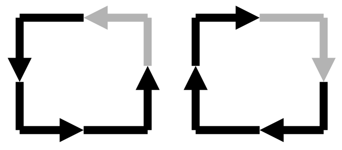


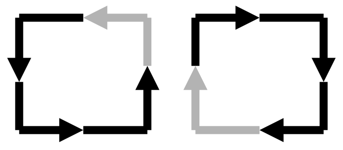
Breaking Deadlock

- Consider the eight possible turns in a 2-d array (note that turns lead to cycles)
- By preventing just two turns, cycles can be eliminated
- Dimension-order routing disallows four turns
- Helps avoid deadlock even in adaptive routing


West-First


North-Last


Negative-First


Can allow deadlocks

Packets/Flits

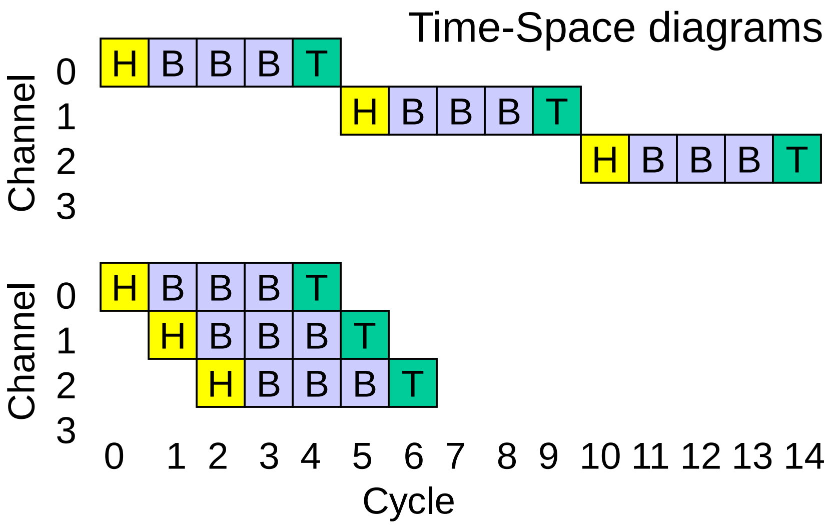
- A message is broken into multiple packets (each packet has header information that allows the receiver to re-construct the original message)
- A packet may itself be broken into flits – flits do not contain additional headers
- Two packets can follow different paths to the destination Flits are always ordered and follow the same path
- Such an architecture allows the use of a large packet size (low header overhead) and yet allows fine-grained resource allocation on a per-flit basis

Flow Control

- The routing of a message requires allocation of various resources: the channel (or link), buffers, control state
- Bufferless: flits are dropped if there is contention for a link, NACKs are sent back, and the original sender has to re-transmit the packet
- Circuit switching: a request is first sent to reserve the channels, the request may be held at an intermediate router until the channel is available (hence, not truly bufferless), ACKs are sent back, and subsequent packets/flits are routed with little effort (good for bulk transfers)

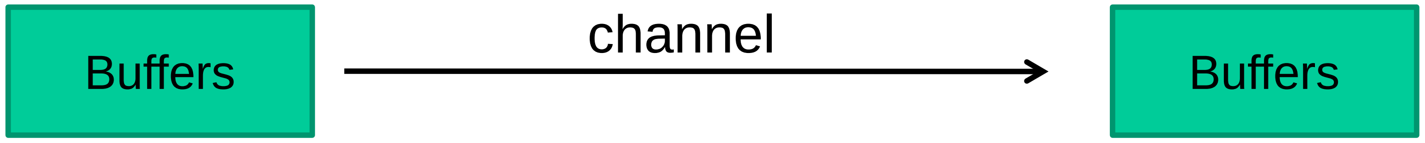
Buffered Flow Control

- A buffer between two channels decouples the resource allocation for each channel
- Packet-buffer flow control: channels and buffers are allocated per packet
  - Store-and-forward
  - Cut-through



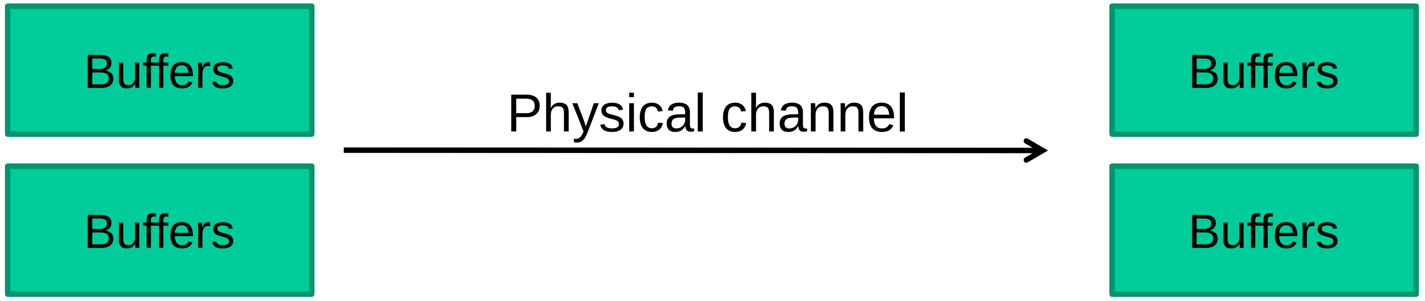
- Wormhole routing: same as cut-through, but buffers in each router are allocated on a per-flit basis, not per-packet

Virtual Channels



Flits do not carry headers. Once a packet starts going over a channel, another packet cannot cut in (else, the receiving buffer will confuse the flits of the two packets). If the packet is stalled, other packets can't use the channel.

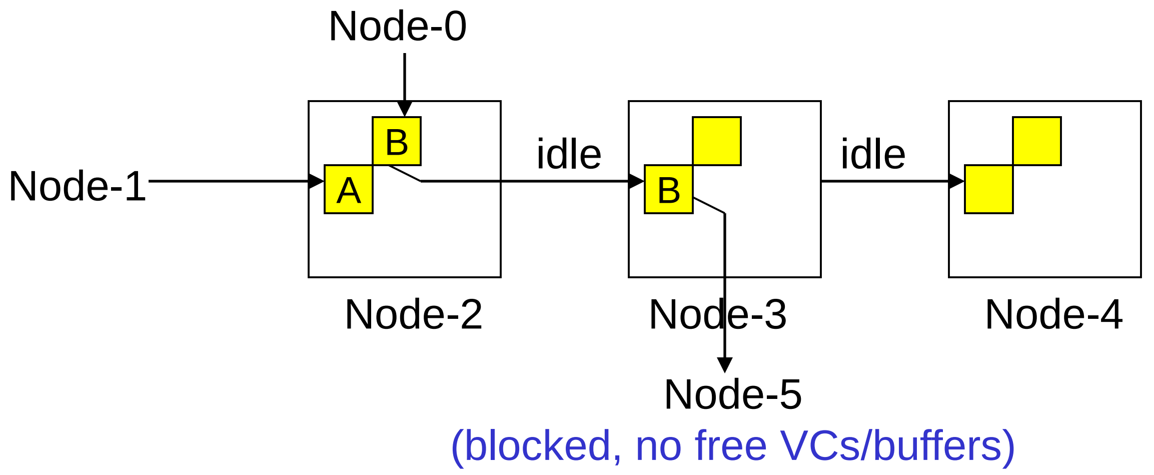
With virtual channels, the flit can be received into one of N buffers. This allows N packets to be in transit over a given physical channel. The packet must carry an ID to indicate its virtual channel.



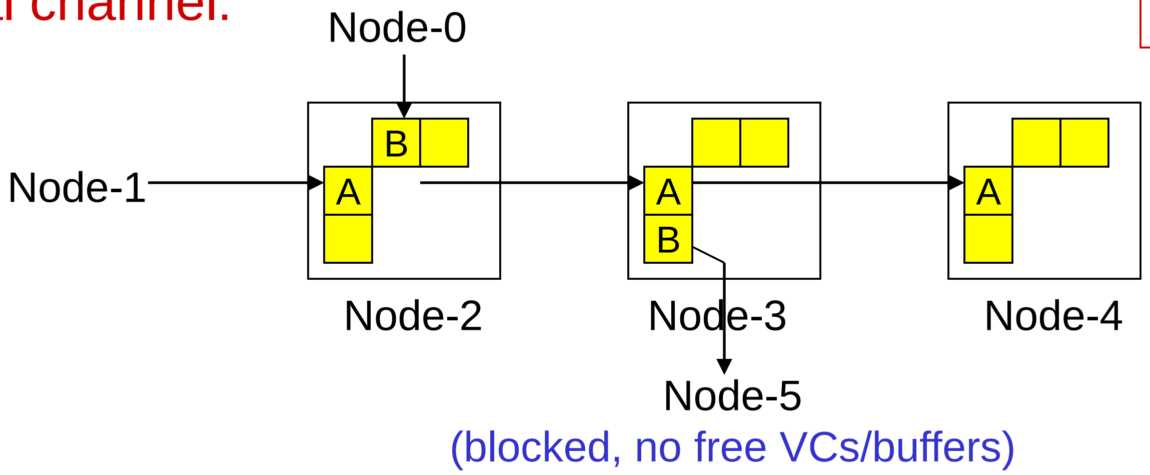
Example

- Wormhole:

A is going from Node-1 to Node-4; B is going from Node-0 to Node-5



- Virtual channel:



Traffic Analogy:
B is trying to make a left turn; A is trying to go straight; there is no left-only lane with wormhole, but there is one with VC

Virtual Channel Flow Control

- Incoming flits are placed in buffers
- For this flit to jump to the next router, it must acquire three resources:
  - A free virtual channel on its intended hop
    - We know that a virtual channel is free when the tail flit goes through
  - Free buffer entries for that virtual channel
    - This is determined with credit or on/off management
  - A free cycle on the physical channel
    - Competition among the packets that share a physical channel

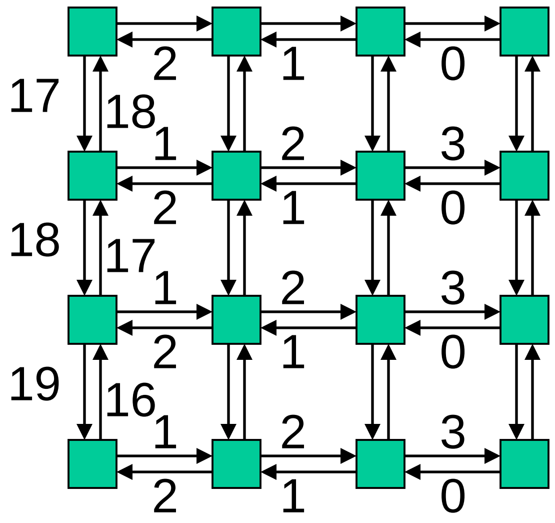
Buffer Management

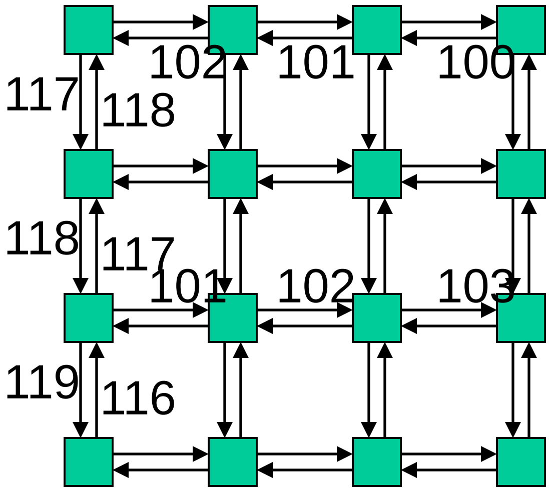
- Credit-based: keep track of the number of free buffers in the downstream node; the downstream node sends back signals to increment the count when a buffer is freed; need enough buffers to hide the round-trip latency

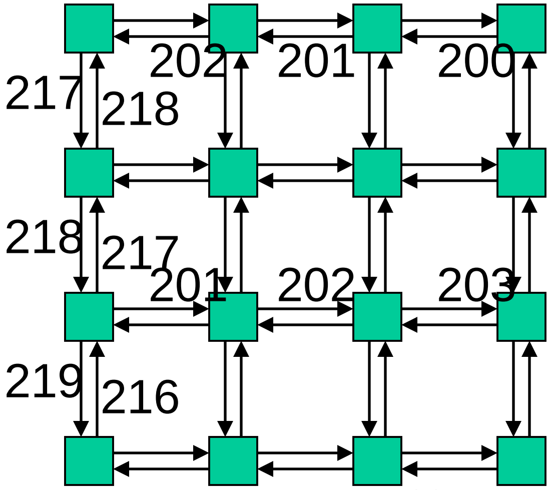
- On/Off: the upstream node sends back a signal when its buffers are close to being full – reduces upstream signaling and counters, but can waste buffer space

Deadlock Avoidance with VCs

- VCs provide another way to number the links such that a route always uses ascending link numbers







- Alternatively, use West-first routing on the 1st plane and cross over to the 2nd plane in case you need to go West again (the 2nd plane uses North-last, for example)

Title

- Bullet