Lecture 9: Directory Protocol, TM

- Topics: corner cases in directory protocols, coherence vs. message-passing, TM intro

Handling Write Requests

- The home node must invalidate all sharers and all invalidations must be acked (to the requestor), the requestor is informed of the number of invalidates to expect

- Actions taken for each state:
  - shared: invalidates are sent, state is changed to excl, data and num-sharers are sent to requestor, the requestor cannot continue until it receives all acks (Note: the directory does not maintain busy state, subsequent requests will be fwded to new owner and they must be buffered until the previous write has completed)

Handling Writes II

- Actions taken for each state:
  - unowned: if the request was an upgrade and not a read-exclusive, is there a problem?
  - exclusive: is there a problem if the request was an upgrade? In case of a read-exclusive: directory is set to busy, speculative reply is sent to requestor, invalidate is sent to owner, owner sends data to requestor (if dirty), and a “transfer of ownership” message (no data) to home to change out of busy
  - busy: the request is NACKed and the requestor must try again

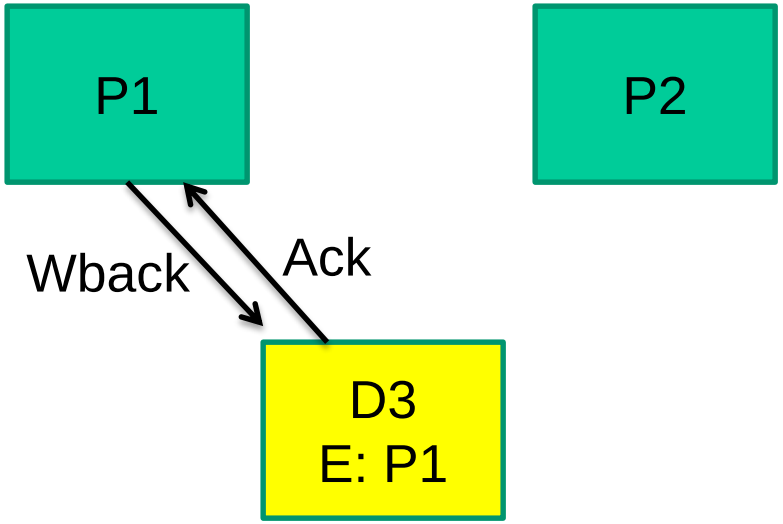
Handling Write-Back

- When a dirty block is replaced, a writeback is generated and the home sends back an ack

- Can the directory state be shared when a writeback is received by the directory?

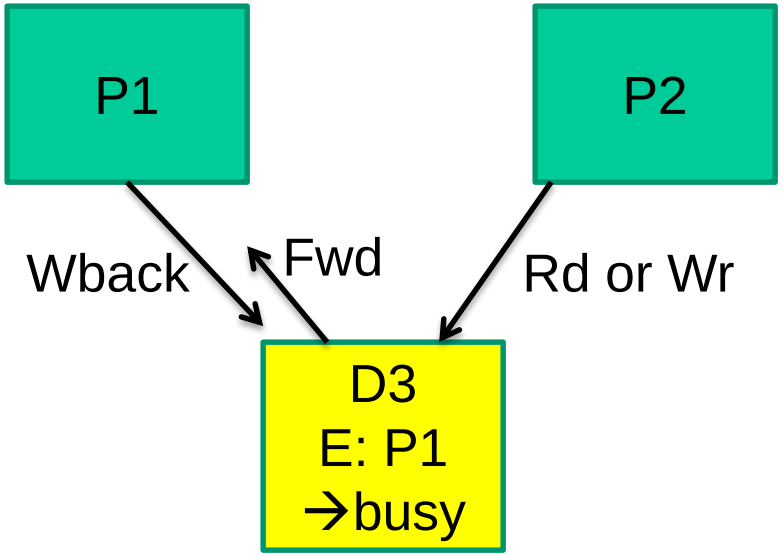
- Actions taken for each directory state:
  - exclusive: change directory state to unowned and send an ack
  - busy: a request and the writeback have crossed paths: the writeback changes directory state to shared or excl (depending on the busy state), memory is updated, and home sends data to requestor, the intervention request is dropped

Writeback Cases



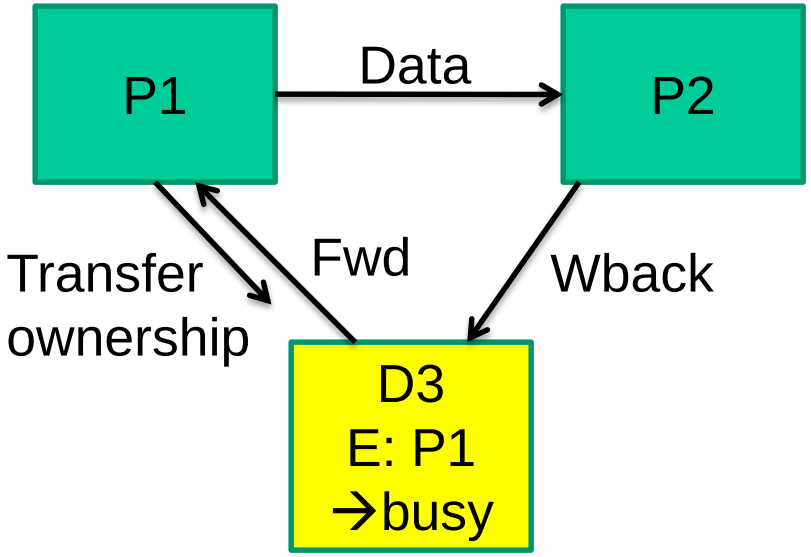
This is the “normal” case
D3 sends back an Ack

Writeback Cases



If someone else has the block in exclusive, D3 moves to busy
If Wback is received, D3 serves the requester
If we didn't use busy state when transitioning from E:P1 to E:P2,
D3 may not have known who to service
(since ownership may have been passed on to P3 and P4…)
(although, this problem can be solved by NACKing the Wback and having P1 buffer its "strange" intervention requests… this could lead to other corner cases…)

Writeback Cases



If Wback is from new requester, D3 sends back a NACK
Floating unresolved messages are a problem
Alternatively, can accept the Wback and put D3 in some new busy state

Conclusion: could have got rid of busy state between E:P1 → E:P2, but with Wback ACK/NACK and other buffering
could have kept the busy state between E:P1 → E:P2, could have got rid of ACK/NACK, but need one new busy state

Future Scalable Designs

- Intel's Single Cloud Computer (SCC): an example prototype
- No support for hardware cache coherence
- Programmer can write shared-memory apps by marking pages as uncacheable or L1-cacheable, but forcing memory flushes to propagate results
- Primarily intended for message-passing apps
- Each core runs a version of Linux

Scalable Cache Coherence

- Will future many-core chips forego hardware cache coherence in favor of message-passing or sw-managed cache coherence?

- It's the classic programmer-effort vs. hw-effort trade-off … traditionally, hardware has won (e.g. ILP extraction)

- Two questions worth answering: will motivated programmers prefer message-passing?, is scalable hw cache coherence do-able?

Message Passing

- Message passing can be faster and more energy-efficient
- Only required data is communicated: good for energy and reduces network contention
- Data can be sent before it is required (push semantics; cache coherence is pull semantics and frequently requires indirection to get data)
- Downsides: more software stack layers and more memory hierarchy layers must be traversed, and.. more programming effort

Scalable Directory Coherence

- Note that the protocol itself need not be changed
- If an application randomly accesses data with zero locality:
  - long latencies for data communication
  - also true for message-passing apps
- If there is locality and page coloring is employed, the directory and data-sharers will often be in close proximity
- Does hardware overhead increase? See examples in last class… the overhead is ~2-10% and sharing can be tracked at coarse granularity… hierarchy can also be employed, with snooping-based coherence among a group of nodes

Transactions

- Access to shared variables is encapsulated within transactions – the system gives the illusion that the transaction executes atomically – hence, the programmer need not reason about other threads that may be running in parallel with the transaction

Conventional model:

```
…
lock(L1);
   access shared vars
unlock(L1);
…
```

TM model:

```
…
trans_begin();
   access shared vars
trans_end();
…
```

Transactions

- Transactional semantics:
  - when a transaction executes, it is as if the rest of the system is suspended and the transaction is in isolation
  - the reads and writes of a transaction happen as if they are all a single atomic operation
  - if the above conditions are not met, the transaction fails to commit (abort) and tries again

```
transaction begin
    read shared variables
    arithmetic
    write shared variables
transaction end
```

Why are Transactions Better?

- High performance with little programming effort
  - Transactions proceed in parallel most of the time if the probability of conflict is low (programmers need not precisely identify such conflicts and find work-arounds with say fine-grained locks)
  - No resources being acquired on transaction start; lesser fear of deadlocks in code
  - Composability

Example

Producer-consumer relationships – producers place tasks at the tail of a work-queue and consumers pull tasks out of the head

```
Enqueue
  transaction begin
     if (tail == NULL)
       update head and tail
     else
       update tail
  transaction end
```

```
Dequeue
     transaction begin
        if (head->next == NULL)
          update head and tail
        else
          update head
     transaction end
```

With locks, neither thread can proceed in parallel since head/tail may be updated – with transactions, enqueue and dequeue can proceed in parallel – transactions will be aborted only if the queue is nearly empty

Example

Is it possible to have a transactional program that deadlocks, but the program does not deadlock when using locks?

```
flagA = flagB = false;
   thr-1                          thr-2
lock(L1)                       lock(L2)
   while (!flagA) {};             flagA = true;
   flagB = true;                  while (!flagB) {};
     *                              *
unlock(L1)                     unlock(L2)
```

- Somewhat contrived
- The code implements a barrier before getting to *
- Note that we are using different lock variables

Atomicity

- Blindly replacing locks-unlocks with tr-begin-end may occasionally result in unexpected behavior

- The primary difference is that:
  - transactions provide atomicity with every other transaction
  - locks provide atomicity with every other code segment that locks the same variable

- Hence, transactions provide a “stronger” notion of atomicity – not necessarily worse for performance or correctness, but certainly better for programming ease

Other Constructs

- Retry: abandon transaction and start again
- OrElse: Execute the other transaction if one aborts
- Weak isolation: transactional semantics enforced only between transactions
- Strong isolation: transactional semantics enforced beween transactions and non-transactional code

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!
  → fast commits, can have slightly slow aborts, can have slightly slow overflow mechanisms

Design Space

- Data Versioning
  - Eager: based on an undo log
  - Lazy: based on a write buffer
    Typically, versioning is done in cache;
    The above two are variants that handle overflow

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

Title

- Bullet