Lecture 13
Directory & Optimizations

Winter 2018
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch.
Readings

For Monday, March 5:

- Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 3 & 4)


Announcements

- Midterm Wednesday
Design Principles

• Think of sending and receiving messages as separate events

• At each “step”, consider what new requests can occur
  ▰ E.g., can a new writeback overtake an older one?

• Two messages traversing same direction implies a race
  ▰ Need to consider both delivery orders
    ○ Usually results in a “branch” in coherence FSM to handle both orderings
  ▰ Need to make sure messages can’t stick around “lost”
    ○ Every request needs an ack; extra states to clean up messages
  ▰ Often, only one node knows how a race resolves
    ○ Might need to send messages to tell others what to do
CC Protocol Scorecard

- Does the protocol use negative acknowledgments (retries)?
- Is the number of active messages (sent but unprocessed) for one transaction bounded?
- Does the protocol require clean eviction notifications?
- How/when is the directory accessed during transaction?
- How many lanes are needed to avoid deadlocks?
NACKs in a CC Protocol

• Issues: Livelock, Starvation, Fairness
• NACKs as a flow control method (“home node is busy”)  
  ❏ Really bad idea...
• NACKs as a consequence of protocol interaction...

Race!  
Put-M & Fwd-Get-S

Race!  
Final State: S  
No need to Ack

1: Put-M
2: Get-S
3: Fwd-Get-S
4: 
5: Get-S NACK
6:
Bounded # Msgs / Transaction

- Scalability issue: how much queue space is needed
- Coarse-vector vs. cruise-missle invalidation
Frequency of Directory Updates

• How to deal with transient states?
  - Keep it in the directory: unlimited concurrency
  - Keep it in a pending transaction buffer (e.g., transaction state register file): faster, but limits pending transactions

• Occupancy free: Upon receiving an unsolicited request, can directory determine final state solely from current state?
Required # of lanes

• Need at least 2:
  
  1: Get-M

  2: Inv

  3: Inv-Ack

  4: Data

• More may be needed by I/O, complex forwarding

• How to assign lane to message type?
  
  ❖ Secondary (forced) requests must not be blocked by new requests
  
  ❖ Replies (completing a pending transaction) must not be blocked by new requests
Some more guidelines

• All messages should be ack’d (requests elicit replies)
• Maximum number of potential concurrent messages for one transaction should be small and constant (i.e., independent of number of nodes in system)
• Anticipate *ships passing in the night* effect
• Use context information to avoid NACKs
Optimizing coherence protocols

Read $A$ (miss)

Readlatency

Get-$S$ $A$

Data $A$

Recall $A$

Data $A$
Prefetching

Prefetch A
Read A (miss)
Read latency

Get-S A
Data A
Recall A
Data A
3-hop reads

Read A (miss)

Get-S A

Fwd-Get-S A

Data A

Read latency
3-hop writes

L (Store A (miss)) → H
- Get-M A
- Data [ack=x]
- Inv-Ack A
- Invalidate A

R

Store latency
Migratory Sharing

- Each Read/Write pair results in read miss + upgrade miss
- Coherence FSM can detect this pattern
  - Detect via back-to-back read-upgrade sequences
  - Transition to “migratory M” state
  - Upon a read, invalidate current copy, pass in “mig E” state
Producer Consumer Sharing

Node 1
Read X
Write X

Node 2
Read X

Node 3
Read X

• Upon read miss, downgrade instead of invalidate
  - Detect because there are 2+ readers between writes

• More sophisticated optimizations
  - Keep track of prior readers
  - Forward data to all readers upon downgrade
Shortcomings of Protocol Optimizations

• Optimizations built directly into coherence state machine
  ❑ Complex! Adds more transitions, races
  ❑ Hard to verify even basic protocols
  ❑ Each optimization contributes to state explosion
  ❑ Can target only simple sharing patterns
  ❑ Can learn only one pattern per address at a time
Table-based protocol predictors

• Decouple predictor from protocol
  ❑ Learn multiple sharing patterns simultaneously
  ❑ Protocol hints \(\rightarrow\) no impact on state machine
  ❑ But, may require significant storage
Memory Sharing Predictor [ISCA'99]:

- 2-level table-based predictor at each dir.
- Keeps history of prior messages
- For each history, keeps a sharing outcome
- E.g., an upgrade by P3 leads to reads by P1, P2

History Table:
- (upgrade, P3)

Pattern Table:
- (upgrade, P3)
- (read, [P1, P2])
- (read, [P1, P2])
- (upgrade, P3)
- (read, [P1, P2])

Prediction for block 0x100
Last Touch Predictors [ISCA '00]

- Predict last access
- Release block
- 3-hop misses → 2-hop

Self-Invalidations are
+ Timely
  early as possible
+ Accurate
  only last-touched block
+ No protocol changes
- Requires more storage

Remote read latency

reader directory writer

predict self-invalidate

send read read send send
An LTP per processor

- collects trace per block
- upon invalidation
  - records trace
- upon every rd/wr
  - compares trace
- e.g. \{PC_i, PC_j, PC_k\}
  - is a last-touch trace

PC_i: rd/wr X  miss on X
PC_j: rd/wr X
PC_k: rd/wr X  last touch

Dynamic Instruction Stream

How Does an LTP Work?
Memory Consistency
Memory Consistency Model

A memory (consistency) model specifies the order in which memory accesses performed by one thread become visible to other threads in the program.

It is a contract between the hardware of a shared memory multiprocessor and the successive programming abstractions (instruction set architecture, programming languages) built on top of it.

- Loosely, the memory model specifies:
  - the set of legal values a load operation can return
  - the set of legal final memory states for a program
Who cares about memory models?

- Programmers want:
  - A framework for writing correct parallel programs
  - Simple reasoning - “principle of least astonishment”
  - The ability to express as much concurrency as possible

- Compiler/Language designers want:
  - To allow as many compiler optimizations as possible
  - To allow as much implementation flexibility as possible
  - To leave the behavior of “bad” programs undefined

- Hardware/System designers want:
  - To allow as many HW optimizations as possible
  - To minimize hardware requirements / overhead
  - Implementation simplicity (for verification)

We will consider all three perspectives
Uniprocessor memory model

• Loads return value of nearest preceding matched store in <p
  ❑ Need to make partial overlaps work
  ❑ Probably need some special cases for I/O
  ❑ Otherwise, any sort of reordering goes!

• Programmer’s perspective:
  ❑ Generally, no way to tell what order things actually happened

• Compiler’s perspective
  ❑ “as-if” rule – any optimization is legal as long as it produces
    the same output “as-if” executed in program order
  ❑ No “random” changes to memory values (except volatile)

• HW perspective
  ❑ Out-of-order, store buffers, speculation all ok
  ❑ Order only needed per-address, enforced via LSQ
Language-Level
DRF-0 Vs SC
Memory Model
Program Order

A ; B

a thread

Execute A and then B
Atomic Shared Memory

Memory is a map from address to values with reads/writes taking effect immediately.

<table>
<thead>
<tr>
<th>Address</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEADBEE</td>
<td>0xDEADBEE</td>
</tr>
<tr>
<td>0xDEADBEE</td>
<td>0x0000002a</td>
</tr>
<tr>
<td>0xDEADBEE</td>
<td>0x00000042</td>
</tr>
<tr>
<td>0xDEADBEE</td>
<td>0xDEADBEEF</td>
</tr>
</tbody>
</table>
Intuitive Concurrency Semantics

Memory model that guarantees this is called **sequential consistency**
Sequential Consistency

X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;

sequential consistency (SC)
[Lamport 1979]
memory operations appear to occur in some global order consistent with the program order
Intuitive reasoning fails in C++/Java

X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;

In C++ model this can crash!
Intuitive reasoning fails in C++/Java

```c
X* x = null;
bool flag = false;

// Producer          // Consumer
A: x = new X();       C: while(!flag);
B: flag = true;       D: x->f++;
```

B doesn’t depend on A. It might be faster to reorder them!

Null Dereference!
Why are accesses reordered?

- **Programming Language**
  - sequentially valid optimizations can reorder memory accesses
  - e.g. common subexpression elimination, register promotion, instruction scheduling

- **Data-Race-Free-0 Model**
  - Java Memory Model
  - C++ Memory Model

- **Hardware**
  - sequentially valid
  - buffers

- **Weak Semantics**
A program has a **data race** if it has an execution in which two **conflicting accesses** to memory are simultaneously ready to execute.

```c
// Thread t
A: x = new Data();
B: flag = true;
C: while(!flag);
D: x->f++;
```

**access the same memory location at least one is a write**
Useful Data Races

- Data races are essential for implementing shared-memory synchronization

```c
AcquireLock()
{
    while (lock == 1) {}  
    t = CAS (lock, 0, 1);  
    if (!t) retry;  
}
```

```c
ReleaseLock()  
{  
    lock = 0;  
}
```
A program is data-race-free if all data races are appropriately annotated (volatile/atomic).

DRFO
[Adve & Hill 1990]
SC behavior for data-race-free programs, weak or no semantics otherwise

Java Memory Model (JMM)
[Manson et al. 2005]

C++0x Memory Model
[Boehm & Adve 2008]
DRF0-compliant Program

X* x = null;
atomic bool flag = false;

A: x = new X();
B: flag = true;
C: while(!flag);
D: x->f++;

- DRF0 guarantees SC
  .... only if data-race-free (all unsafe accesses are annotated)
- What if there is one data-race?
  .... all bets are off (e.g., compiler can output an empty binary!)
Data-Races are Common

• Unintentional data-races
  ❑ Easy to accidentally introduce a data race
    ○ forget to grab a lock
    ○ grab the wrong lock
    ○ forget an atomic annotation
    ○ ...

• Intentional data-races
  ❑ 100s of “benign” data-races in legacy code
Data Races with no Race Condition (assuming SC)

• Single writer multiple readers

```plaintext
// Thread t
A:  time++;  

// Thread u
B:  l = time;
```
Data Races with no Race Condition (assuming SC)

- Lazy initialization

```c
// Thread t
if ( p == 0 )
    p = init();

// Thread u
if ( p == 0 )
    p = init();
```
Intentional Data Races

• ~97% of data races are not errors under SC
  □ Experience from one Microsoft internal data-race detection study [Narayanasamy et al. PLDI’07]

• The main reason to annotate data races is to protect against compiler/hardware optimizations
Data Race Detection is Not a Solution

• Current static data-race detectors are not sound and precise
  ❑ typically only handle locks, conservative due to aliasing, ...

• Dynamic analysis is costly
  ❑ DRFx: throw exception on a data-race [Marino’10]
  ❑ Either slow (8x) or requires complex hardware

• Legacy issues
Deficiencies of DRF0

- weak or no semantics for data-racy programs
- no easy way to identify & reject data-racy programs

problematic for

DEBUGGABILITY

Analogous to unsafe languages: relying on programmer infallibility

- optimizing jump to arbitrary code! [Boehm et al., PLDI 2008]
- on safety at the cost of complexity [Ševčík&Aspinall, ECOOP 2008]
Languages, compilers, processors are adopting DRF0

Not a strong foundation to build our future systems
Language-level SC: A Safety-First Approach

Program order and shared memory are important abstractions

Modern languages should protect them

All programs, buggy or otherwise, should have SC semantics
What is the Cost of SC?

SC prevents essentially all compiler and hardware optimizations.

And thus SC is impractical.