Topics

• Network-Based Systems
  – Coherence based on directories

• Maintaining Consistency
  – Relaxed consistency models
Conceptual View

- All processors access single memory
  - Physical address space
  - Use virtual address mapping to partition among processes
- If one processor updates location, then all will see it
  - Memory consistency
Network-Based Realization

Memory
- Partitioned Among Processors

Network
- Transmit messages to perform accesses to remote memories

Caches
- Local copies of heavily used data
- Must avoid stale data
  - Harder than with bus-based system
  - Lots of things happening simultaneously

Considerations
- Scales well
  - 1024 processor systems have been built
- Nonuniform memory access
  - 100’s of cycles for remote access
Network-Based Cache Coherency

Home-Based Protocol

- Each block has “home”
  - Memory controller tracking its status
  - Based on physical address

- Home maintains
  - Block status
  - Identity of copy holders
    - 1 bit flag / processor
  - Only need entry when block has remote copy

### Block Status Values

- **Shared**
  - 1 or more remote, read-only copies

- **Exclusive**
  - Single, writeable copy in remote cache

- **Uncached**
  - No remote copies
Network-Based Consistency

To Obtain Copy of Block

• Processor sends message to its home
• Home retrieves remote copy if status is exclusive
• Sends copy to requester
• If exclusive copy requested, send invalidate message to all other copy holders

Tricky Details

• Lots of possible sources of deadlock & errors
• Don’t have serialization of events imposed by bus
• Transactions only “seen” by sender & receiver

Example Protocol

• Complete functionality
• Omits optimizations
• Conservative synchronization
Implementation Details

- **p**: Processor identifier
- **a**: Block address
- **d**: Block data

### Messages from Cache to Block Home

<table>
<thead>
<tr>
<th>Type</th>
<th>Contents</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>p, a</td>
<td>Processor p wants to read a</td>
</tr>
<tr>
<td>XREAD</td>
<td>p, a</td>
<td>Processor p wants to write a</td>
</tr>
<tr>
<td>WRITEBACK</td>
<td>p, a, d</td>
<td>Proc. p flushes exclusive block</td>
</tr>
<tr>
<td>WRITEHOLD</td>
<td>p, a, d</td>
<td>Proc. p shares exclusive block</td>
</tr>
<tr>
<td>RELEASE</td>
<td>p, a</td>
<td>Proc. p flushes clean block</td>
</tr>
</tbody>
</table>

### Messages from Block Home to Cache

<table>
<thead>
<tr>
<th>Type</th>
<th>Contents</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALIDATE</td>
<td>p, a</td>
<td>Give up clean copy</td>
</tr>
<tr>
<td>DATA</td>
<td>p, a, d</td>
<td>Reply to read / write request</td>
</tr>
<tr>
<td>FETCH</td>
<td>p, a</td>
<td>Get readable copy from remote</td>
</tr>
<tr>
<td>XFETCH</td>
<td>p, a</td>
<td>Get writeable copy from remote</td>
</tr>
<tr>
<td>NACK</td>
<td>p, a</td>
<td>Cannot fulfill request</td>
</tr>
</tbody>
</table>
Tricky Issues

Independent Actions between Cache & Home

- Cache evicts exclusive block while home in process of fetching
  - Home will get WRITEBACK when expecting WRITEHOLD
  - Cache will receive FETCH for invalid block
- Cache evicts exclusive block while home in process of xfetching
  - Cache will receive XFETCH for invalid block

Independent Actions between Two Processor Caches

- Processor r attempts read or write while home handling read or write request by processor p
  - Home sends NACK to processor r
  - Processor r retries read or write operation
- Potential for livelock
  - Processor r keeps getting NACK’ed
Performing Processor Operations

Processor requests cache to perform load or store
– On word in cache block i

Cache line currently holds block t
– May or may not have i = t

Cache can either:
– Perform operation using local copy
– Issue request message to network (*italicized*)
  » Wait for response (*bold*)
  » Perhaps stall until completed
Cache Handling of Processor Read

- **Invalid**
  - **Read Miss** \(READ(p,i)\)
  - **NACK\((p,i)\)**
  - **Data** \(DATA(p,i,d)\)

- **Shared**
  - **Read Miss** \(RELEASE(p,t)\)
  - **Data** \(DATA(p,i,d)\)

- **Exclusive**
  - **Read Miss** \(WRITEBACK(p,t,d)\)
  - **Read Hit**

Legend:
- \(i\) Requested Block
- \(t\) Current Block
Processing by Home

Home Memory Controller

- Receives requests from caches
  - To handle processor reads & writes
  - Unsolicited evictions
- Sends commands to copy holders
- Determines when all outstanding copies invalidated
- Responds to requestor
- Send NACK if unable to satisfy request

<table>
<thead>
<tr>
<th>Block</th>
<th>Status</th>
<th>Copy Holders</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>shared</td>
<td>0101010000</td>
</tr>
<tr>
<td>25</td>
<td>Exclusive</td>
<td>0100000000</td>
</tr>
<tr>
<td>26</td>
<td>uncached</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Network
Home Handling of Processor Read

- **p**: Requesting Processor
- **q**: Copy-holding Processor
- **r**: Competing Processor(s)

**S**: Set of copy holders
- `+=` Add to set
- `–=` Remove from set

**Variables**:
- `READ(r,i)`
- `NACK(r,i)`
- `XREAD(r,i)`
- `NACK(r,i)`
- `READ(p,i)`
- `FETCH(q,i)`
- `WRITEHOLD(q,i,d)`
- `DATA(p,i,d)`
- `WRITEBACK(q,i,d)`
- `DATA(p,i,d)`
- `READ(p,i)`
- `DATA(p,i,d)`
- `WRITEHOLD(q,i,d)`
- `DATA(p,i,d)`
- `WRITEBACK(q,i,d)`
- `DATA(p,i,d)`
- `READ(p,i)`
- `DATA(p,i,d)`
- `WRITEHOLD(q,i,d)`
- `DATA(p,i,d)`
- `WRITEBACK(q,i,d)`
- `DATA(p,i,d)`

**States**:
- **Read Pending**
- **Exclusive**
- **Uncached**
- **Shared**

**Transitions**:
- `READ(r,i) → NACK(r,i)`
- `XREAD(r,i) → NACK(r,i)`
- `READ(p,i) → FETCH(q,i)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`
- `READ(p,i) → DATA(p,i,d)`
- `WRITEHOLD(q,i,d) → DATA(p,i,d)`
- `WRITEBACK(q,i,d) → DATA(p,i,d)`

**Unsolicited eviction**
Cache Handling of Processor Write

Invalid

- NACK(p,i)
- Write Miss
- XREAD(p,i)

Write Pending

- Write Miss
- RELEASE(p,t)

Shared

Exclusive

- DATA(p,i,d)
- Write Hit

Write Hit

Write Miss

WRITEBACK(p,t,d)

Requested Block

Current Block
Home Handling of Processor Write

- **Uncached**
  - \( \text{XREAD}(p,i) \)
  - \( \text{DATA}(p,i,d) \)
  - \( S \leftarrow p \)

- **Exclusive**
  - \( \text{XREAD}(p,i) \)
  - \( \text{XFETCH}(q,i) \)
  - \( \text{WRITEBACK}(q,i,d) \)
  - \( \text{DATA}(p,i,d) \)
  - \( S \leftarrow q \)
  - \( S \leftarrow p \)

- **Dirty Write Pending**
  - \( \text{READ}(r,i) \)
  - \( \text{ACK}(r,i) \)
  - \( \text{XREAD}(r,i) \)
  - \( \text{NACK}(r,i) \)

- **Clean Write Pending**
  - \( \text{RELEASE}(q,i) \)
  - \( S \leftarrow q \)
  - \( S \) nonempty

- **Shared**
  - \( \text{XREAD}(p,i) \)
  - \( \forall q \in S: \text{INVALIDATE}(q,i) \)
  - \( S \) nonempty

- **Set of copy holders**
  - \( S \)
  - \( += \) Add to set
  - \( -= \) Remove from set

- **Requesting Processor**
  - \( p \)

- **Copy-holding Processor(s)**
  - \( q \)

- **Competing Processor(s)**
  - \( r \)

- **CS 740 F’97**
Network Monitoring by Cache

Cache receives commands from network
– Unlike snooping, only get messages regarding currently-held blocks

Possible actions
– Invalidate entry and release copy
– Allow sharing of exclusively-held block
– Surrender copy of block
Cache Handling of Network Commands

Blocks that have already been written back or released

Invalid

Valid

Shared

Exclusive

Invalidate\((p,t)\)

XFetch\((p,t)\)

Invalidate\((p,t)\)

Release\((p,t)\)

XFetch\((p,t)\)

Fetch\((p,t)\)

Writehold\((p,t,d)\)

Writeback\((p,t,d)\)

\(t\) Current Block
Home Handling of Unsolicited Evictions

**WRITEBACK**(q,i,d)

- Uncached
- Shared
- Exclusive

**RELEASE**(q,i)

- S empty
- S nonempty

S Set of copy holders

+= Add to set

-= Remove from set

q Copy-Holding Processor
Additional Optimizations

Data Forwarding
• Copy holding processor sends data directly to requestor
  – But still must synchronize with home
• Currently pass through home

Quick Acknowledgement
• Supply data to requestor before all outstanding copies invalidated
  – Copy holders may continue reading stale data

Processor wants to write to currently-held clean block
• Currently treat as write miss
  – Home will invalidate held copy
• Request home to invalidate all other copies
• May want quick acknowledgement
  – Implications examined in Asst. 6, Part II
Memory Consistency Revisited

View By Individual Processor

• Reads and writes to given address occur in program order
  – Guaranteed by hazard-preventing mechanisms of processor

Sequential Consistency

• Reads & Writes by processor to distinct address
• Overall effect should match that of some interleaving of the individual process steps

When is a Write “Performed”

• When any later read by any processor “sees” new data
• All outstanding copies invalidated

Initially:  \( x = y = 0 \)

<table>
<thead>
<tr>
<th>Process A</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1: ( x = 1 )</td>
</tr>
<tr>
<td>a2: if (( y == 0 )) ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1: ( y = 1 )</td>
</tr>
<tr>
<td>b2: if (( x == 0 )) ...</td>
</tr>
</tbody>
</table>
Sequential Inconsistency

- **Cannot have both tests yield T**
  - b2 must precede a1
  - a2 must precede b1
  - Cannot satisfy these plus program order constraints

Real Life Scenario

- **Process A**
  - Remote write x
    - » Put into write buffer
  - Local read y
- **Process B**
  - Remote write y
    - » Put into write buffer
  - Local read x
- **Could have both reads yield 0**
Sources of Sequential Inconsistency

Processor Write Buffer
• Allows processor to continue without waiting for write to complete
  – W -> R
  » write may not complete before later read initiated
• If > 1 entry, also allows multiple outstanding writes
  – W -> W
  » writes may not complete in order

Nonblocking Read
• Processor doesn’t stall waiting for read to return result
  – R -> W
• If > 1 entry load buffer, also allows multiple outstanding reads
  – R -> R

Fast Acknowledgements in Network-Based Protocol
• Respond to \textit{XREAD} request before all outstanding copies invalidated

Important Mechanisms for Tolerating Memory Latency
Relaxed Consistency Models

Goals

• Enable latency tolerating mechanisms as much as possible
• Provide concurrent programmer with some mechanisms for program synchronization

Examples

• Discussed in H&P Section 8.6
Supporting Program Synchronization

Typical Scenario
- Processes need access to shared resources
  - Tables, shared program state
- Use software locks to prevent simultaneous access
  - Shared variables + synchronization conventions

Processor Requirement
- Use acquire primitives to obtain lock
  - E.g., MIPS LL/SC
- Read & write shared data only in critical section
- Use release primitives to release lock
  - Synchronized store
Release Consistency

• Weakest model discussed in book
• Still allows effective synchronization

Requirements
• Acquire operation must complete before any succeeding reads or writes
  – Don’t enter critical section too early
• Release operation may not begin until all pending reads & writes completed
  – Don’t read or write shared state once lock released
Memory Consistency in CILK

DAG Consistency

- View control structure as directed acyclic graph
  - Series-parallel structure formed by spawn’s & synch’s
- Guarantee sequential consistency for reads & writes along any path
  - But no guarantees for potentially concurrent threads

Implications

- Deterministic outcome as long as no interference among potentially concurrent threads
  - write-read or write-write
- Good enough for writing parallel applications
  - Usually want deterministic results
- Not adequate for supporting OS functions
Implementing DAG Consistency

- DAG dependency between threads i & j on separate processors p & q
- Processor p writes back any dirty blocks before passing control from thread i
  - Home copies of all data produced up through i valid
- Processor q flushes all blocks before executing thread j
  - All thread data by j and successors will be retrieved from homes

Comparison to Other Protocols
- No need to forcibly retrieve data from processor
- Well suited to software implementation