COMA: Cache-Only Memory Arch.

- Organized like NUMA shared memory MP
  - Except, main memories are replaced with attraction memories
- Principle: Data is free to migrate to where it is being used.
  - Data placement problem is reduced
  - Data can adapt to dynamic behavior of programs
  - BUT, there is no longer a "backing store" for data
COMA, contd.

- **No Backing Store**
  - => cannot delete last copy during replacement
  - => must be able to find data

- **Possible Solution:**
  - Have a "directory home" for data that is in a static location
  - Consulting directory home can avoid deleting last copy
  - Directory home can be consulted to find location of data

- **DDM Solution:**
  - Use hierarchical snooping buses
  - Allow directories to be dynamically distributed in hierarchy.

---

**Minimal COMA**

```
Arbitration, Selection

OA

CONTROLER

Above Protocol

STATE + DATA MEMORY

PROCESSOR

Top Protocol

ATTRACTION MEMORY

PROCESSOR

ATTRACTION MEMORY

DDM bus
```

(C) 2003 J. E. Smith
Minimal COMA

- Cache lines are referred to as "items"
- Single bus
- Above protocol: to bus (and other processors)
- Below protocol: to local processor
- Invalidate Protocol
- Protocol States
  - invalid: this subsystem does not have the item
  - exclusive: only this subsystem has item
  - shared: this subsystem and others may have item
  - reading: waiting for a data value after read (split transaction bus)
  - waiting: waiting to become exclusive after issuing erase
  - reading and waiting: waiting for data value, to become exclusive
  - answering: this subsystem has promised to answer a read request

Minimal COMA, contd.

- Bus transactions
  - Erase: erase all copies of this item
  - Exclusive: acknowledge an erase request
  - Read: read a copy of the item
  - Data: data in response to an earlier read
  - Inject: carry the only copy of an item and look for subsystem to place it (used on replacement)
  - Out: carry the item out of the subsystem (used on replacement)
Minimal COMA Protocol

Operations
1) Read in Exclusive or Shared, or Write in exclusive proceeds without interruption
2) Read in Invalid results in a Read request broadcast on bus and transition to state Reading
   Eventually an attraction memory will respond with a Data bus transaction. Requesting attraction memory will capture data from bus, change to Shared state and read will proceed.
3) Write in shared state results in an Erase request to be broadcast on bus and transition to state Waiting.
   Eventually, the “top” protocol on bus responds with an Exclusive acknowledgement and Waiting attraction memory can proceed (write and set state to Exclusive).
   If a transaction in an output buffer matches an Erase address on the bus, it is removed from the buffer and state is changed to Reading-and-Waiting and one puts a read request on bus.
4) Write in Invalid state results in a Read request and state Reading-and-Waiting; Upon a Data reply, the state changes to Waiting and an Erase request is sent.
Minimal COMA, contd.

- **Replacement**
  - Basic Rule: can't throw away only remaining copy
  - One might select oldest Shared entry.
    » This generates an Out bus transaction
  - If an Out transaction finds an attraction memory item Shared, Reading, Waiting, or Reading-and-Waiting, it no-ops;
  - Otherwise, it is converted into an Inject transaction.
  - An Inject can also be produced by replacing an item that is Exclusive
  - The Inject copy is the last copy of an item and must find a new home.
    » Find 1st empty space (Invalid state)
    » Replace a shared item
    » If physical address space < or = sum of attraction memories, then space will eventually be found.
    » Otherwise, paging may be needed

Hierarchical DDM

- For larger systems: build hierarchy of busses
Hierarchical DDM

• A directory at any level keeps track of data that resides below in hierarchy
• Transactions only pass through a directory from above if the request must be serviced by a subsystem below
• Transactions only pass through a directory from below if the request cannot be serviced by a subsystem below

Hierarchical DDM, Contd.

• Multilevel Read
  – If Read cannot be satisfied at a given level, it is sent to the next higher bus
  – The directory changes the item’s state to Reading
  – Eventually, a directory is found that has the item.
  – Its state is changed to Answering and re-transmits on lower bus; This directory changes to Answering and re-transmits below, etc. until the actual data item is found.
  – Then the Data response reverses the path, setting states to Shared along the way
Hierarchical DDM, Contd.

- **Multilevel Write**
  - An Erase works its way up through directories changing directory states to Waiting along the way.
  - Erase from below to a directory with item results in exclusive acknowledgement being sent below.
  - Propagation of erase ends when an Exclusive entry is found or it reaches the top of the tree.

- **Replacement**
  - Replacement of a Shared item causes an Out transaction to go up hierarchy until it finds a Shared, Reading, Waiting, or Answering copy.
  - Otherwise when it reaches an Exclusive directory, the Out turns into an Inject.
  - Replacement of an Exclusive item generates an Inject.
  - Injects look for space on local bus, then goes to home bus (function of address)

- **Directory Growth**
  - Directories in hierarchy satisfy inclusion property
  - Directories in higher levels of hierarchy get successively larger (in size and associativity) by factors of $B_i$; where $B_i$ is the "branch factor" at level i.
    
    _Ouch._
  - On could limit associativity to help a little; but this will require evictions to maintain inclusion.
Interconnection Network

- Bandwidth may need to be higher at higher levels
  - Use Directory Splitting
    \( \Rightarrow \) form of fat tree

Virtual Memory

- Page-In
  - Allocate space and lock into attraction memory
  - Do I/O from disk
  - Unlock attraction memory locations
- Page-out
  - Collect items and lock into attraction memory
  - Output to disk
  - Purge and unlock attraction memory locations
- Memory Ovhd
  - tags are quite short
  - 8 attraction memories, each 2-way assoc \( \Rightarrow \) 4 tag bits plus 4 bits of state \( \Rightarrow \) 6 percent ovhd
- A potential Problem? relative data placement
Alternatives

- Use a home directory location
  - => avoid hunt for directory; replicated directory info could add to total access time if home is far away
  - cache some directory entries locally?
- Shared attraction memories (as in prototype)

Virtual Shared Memory

- VM: Brief Review
  - virtual memory -- not just to make a small memory look large.
- In general: allow data "objects" to reside anywhere in a system or network
  (main memory, disk, across a network, etc.)
  - data may not even exist, but may be created when needed e.g. copy on write
- Basic function: access object via loads/stores
  - if not in main memory: fault
  - OS gets object and sets up map
  - return to user and re-try
- Implementation:
  - page tables in memory -- location of page, access privileges, valid
  - TLB -- performance enhancement in processor
Clustered Systems

- A number of computers connected via a (often high speed) network
- Separate logical address spaces in each computer for same user program
- Access to network may require normal I/O system
  => usually done through OS to preserve protection/security
Virtual Shared Memory

- Program model: shared memory
- Memory mapping managers
  - part of the OS
  - responsible for mapping portions logical memory
  - into physical memories of the multiple computers
- Memory mapping managers must keep logical memory coherent -- similar to cache coherence
- Primary architecture mechanism(s) for implementing shared memory
  - access bits in the page tables (and TLBs)
    => memory access and page faults to OS
  - Inter-processor interrupts

Differences from Cache Coherence

- Speed
  - Work is done by traps to OS and System calls
    - Much slower than cache hardware
- Large blocks of data can be moved to amortize the overhead
  - Compatible with page sizes used
    - (4K to 16K are common)
- Backing store is very far away
  => COMA - like implementations
- "false sharing" problem can be much more severe
  - unrelated data may share a page
Differences from Cache Coherence, contd.

- More flexible algorithm choice
  - Because of software implementation,
  - and because of the time scale involved,
  - Methods can be more flexible and dynamic
  - Algorithms are not wired into hardware
    \[ \Rightarrow \text{can be changed/optimized over time via software upgrades} \]

Memory Coherence Strategies

- Invalidate Protocol
  - on write fault to a page:
    » move a copy to faulting processor
    » invalidate all other copies
    » mark page with write permission
    » return from fault
  - on a read fault to a page:
    » get a copy of the page
    » mark copy as "read only"
    » change access to other copies to "read only"
    » return from fault
Memory Coherence Strategies, contd.

• **Update Protocol**
  – on a write fault to a page:
    » write to all copies of the page
    » return from fault
  – on a read fault to a page:
    » just like invalidate case
  – Because of the high overhead of doing a broadcast, write update method is generally impractical

Page Ownership

• **Fixed**: same processor always owns a given page
  – other processors must negotiate with manager
  – prior to all writes
  – generally impractical due to high overheads
• **Dynamic**: ownership is shifted among processors
• **Strategies for maintaining dynamic ownership**
  – centralized and distributed
• **Distribution of ownership data**
  – fixed or dynamic
Basic Software Mechanisms

• Page Tables: key info
  – Access: access permissions to page
  – Copy set: list of processors that have "read" copies
  – Lock: to lock page entry and ensure atomic update
  – Software queues processes behind lock

Invalidation Methods

• Broadcast
  – do not need copy set
    => an interrupt of every processor?
  – can cause big overheads in large systems

• Individual invalidation
  – use copy set
  – interrupt only those processors with a copy
  – each message requires a request and reply
    => 2m messages for m copies

• Multicast
  – simultaneous messages to all copies
  – can be most efficient method
  – depends on whether there is hardware support
Methods for Holding Copy Sets

- Bit vector
- Linked bit vector
  - segments linked together
- neighbor bit vector
  - only contains bit vectors for neighbors
    => invalidations must be propagated

Centralized Managers

- Use a "protected subsystem" or "monitor"
  => data structure plus associated procedures to provide access
- Resides in a single processor (computer) contains a complete table of all pages
- Central manager Table entries:
  - Owner
  - Copy Set
  - Lock
- Tables in each processor
  - Access
  - Lock
Centralized Manager: Read Page Fault

requester: lock local page table
requester: *message* to manager
manager: lock manager page table entry
manager: adds to copy set
manager: *message* to owner to send copy to requester
owner: lock owner page table
owner: *message* copy to requester
owner: set access = read only
owner: unlock owner page table
requester: confirmation *message* to manager
requester: set access to read
requester: unlock local page table
manager: receive confirmation
manager: unlock manager page table entry

total: 4 messages

Centralized Manager: Write Page Fault

requester: lock local page table
requester: *message* to manager
manager: lock manager page table
manager: send invalidate *message* to copy set
manager: set owner = requester; clear copy set
manager: *message* to owner to send copy to requester
owner: lock owner page table
owner: *message* copy to requester
owner: set access = null
owner: unlock owner page table
requester: confirmation *message* to manager
requester: set access to write
requester: unlock local page table
manager: receive confirmation
manager: unlock manager page table entry
Centralized Manager, contd

• **Summary**
  - typically 4 messages (+ invalidations)
  - at most two messages to locate owner
  - confirmation message can be removed by adding
  - copy set in owner tables
    » synchronization becomes distributed, but manager still must respond to every page fault

Fixed Distributed Manager

• **Attempt to remove bottleneck**
• **A manager per processor**
• **Fixed locations for page info**
  - distribute pages to be managed in some fixed way
  - e.g. interleave on lower order bits
  - processor’s manager is responsible for its pages
• **Algorithms similar to central manager**
• **Claim in paper: difficult to find good distribution function that fits all applications well**
Broadcast Distributed Manager

- A type of dynamic distributed manager
- A processor manages the pages it owns
- Faulting processors broadcast to find manager/owner
- No owner table -- each processor has owner flag
- Conceptually simple,
- Broadcast on almost every fault is miserable – except for a very small number of processors

A Dynamic Distributed Manager

- Keep track of "probable" owners
  - Add field to each processor’s local page table
  - probable owner is the processor to which this processor most recently passed ownership
    => list of processors leading to true owner
- Initially: owner is a default processor that owns all pages
- On a fault: send message to probable owner indicated in local table
  - If this is not owner, then \flits\P probable owner field is used, etc.
A Dynamic Distributed Manager: Operation

- Read fault:
  - owner: send a copy
  - owner: set probable owner field to requester

- Write fault:
  - owner: send a copy
  - owner: invalidate copies
  - owner: set probable owner field to requester

- Receive invalidation:
  - holder: invalidate copy
  - holder: set probable owner to requester

- Performance degrades *logarithmically* as the number of processors that *content* for the page

---

A Dynamic Distributed Manager: Optimizations

- Do not change owner on a read fault
  - requester sets probable owner to current owner
  - (reduces one message for each read page fault)

- Broadcast owner’s identity after M page faults to a page

- Optimization: distribute copy sets
  - keep partial copy sets with probable owners
  => faster invalidation when there is no broadcast
  => read fault only needs to find a copy... not the owner

---

Table II. Lookup Time for Faults

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Average number of messages/fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = N/4</td>
<td>M = N/2</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
</tr>
<tr>
<td>16</td>
<td>2.10</td>
</tr>
<tr>
<td>32</td>
<td>3.10</td>
</tr>
<tr>
<td>64</td>
<td>4.80</td>
</tr>
<tr>
<td>128</td>
<td>6.60</td>
</tr>
<tr>
<td>256</td>
<td>9.30</td>
</tr>
</tbody>
</table>

---

Table III. Broadcasts Faults

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Average number of messages/fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = N/4</td>
<td>M = N/2</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
</tr>
<tr>
<td>16</td>
<td>2.00</td>
</tr>
<tr>
<td>32</td>
<td>3.00</td>
</tr>
<tr>
<td>64</td>
<td>5.00</td>
</tr>
<tr>
<td>128</td>
<td>6.00</td>
</tr>
<tr>
<td>256</td>
<td>9.00</td>
</tr>
<tr>
<td>512</td>
<td>1.10</td>
</tr>
</tbody>
</table>

---

(C) 2003 J. E. Smith  ECE/CS 757  37
Performance

• Parallel programs on a ring of Apollo workstations
• Note small number of processors
• Programs:
  – Parallel Jacobi alg for 3D PDEs; sparse matrices
  – Parallel sort
  – Matrix multiply
  – Dot product

Performance

• Jacobi PDE: 50 x 50 x 50
  – superlinear speedup
  – (more overall physical memory)
  – this is also observed with cache systems sometimes
  – disk accesses very much reduced

Fig. 6: Speedups of a 3-D PDE where $n = 50^3$. 
Performance

- **Jacobi PDE: 40 x 40 x 40**
  - If problem size is smaller, speedup is no longer superlinear

![Graph showing speedup vs. number of processors for Jacobi PDE](image)

Benchmarks, contd.

- **Parallel Sort**
  - 200K elements sorted
  - Speedup is less than linear (solid)
  - Even with zero communication delay (dashed)

![Graph showing speedup vs. number of processors for Parallel Sort](image)
Benchmarks, contd.

- Parallel dot product
  - 128K elements
  - speedup not good at all

Benchmarks, contd.

- Matrix multiplication
  - 128 by 128
  - speedup close to linear
Memory Coherence Algorithms

- PDE Benchmark
- Compare improved centralized, fixed dist, dyn dist algorithms
- Similar numbers of page faults
  forwarding requests tends to distinguish algorithms
  (i.e. finding owner)
- Forwarding requests similar for fixed dist. and improved centralized
- Forwarding requests for dynamic distributed is much better
  (probable owner is usually correct)

Application Performance on SVM Systems

- Modern SVM protocols
- Sharing patterns
  - mapping inherent sharing onto
  - induced sharing
- Performance results
  - consider different sharing patterns
  - analysis of where the time goes
Release Consistency

• A form of weak ordering
• Assumes sharing is in the form:
  – Acquire exclusive access
  – Modify shared variables
  – Release exclusive access
• At acquire point --
  – initiate acquire immediately
  – accesses following acquire must wait for acquire to complete
• At release point --
  – wait for accesses to complete
  – initiate release
  – accesses following release do not wait for release to complete

Lazy Release Consistency SVM

• All-software multiple writer protocol
  – Multiple writers record changes locally
  – Merge changes at synchronization points
• To detect changes:
  – Each processor saves a twin
  – Compare current with twin => changes
  – Record changed locations in diff
• At an acquire event, make coherent
  – Invalidate pages modified by others
  – before previous release event
  – On first access after invalidation,
  – collect diffs & merge to form coherent page
Lazy Release Consistency Benefits

- Delaying coherence reduces data and coherence messages  
  - and reduces software protocol overhead
- Fragmentation is reduced  
  - by combining multiple mods  
  - by sending only diffs
- Example: multiprocess a vectorizable do loop  
  - doesn't really follow acquire-release model...  
  - each process writes to different array elements in same page  
  - terminate loop with barrier synchronize  
  - make coherent at barrier synch.

Lazy Release Consistency Costs

- Diff processing  
  - Creation of twins  
  - Creation of diffs  
  - Merging diffs  
  - The above cost software time and pollute caches
- Acquiring processor visits multiple processors to collect diffs
- Consecutive diffs of same page have to be obtained separately for merging  
  (from multiple acquires?)