Distributed Systems

Distributed Shared Memory
Introduction

• Recall that there are two different kinds of multiple processor systems
  – Multiprocessors
  – Multicomputers

• In multiprocessors, two or more processors share a common memory
  – Any processor can read and write to any word in shared memory by moving data to or from the desired location

• Designing computers where multiple processors share a single memory is difficult
  – Hard to scale
  – Complex to implement
  – Difficult to maintain
  – Expensive
Introduction

• Multicomputers are architect with many single-board computers that have a private memory, a network interface and connect together using network cabling and communication protocols

• Hardware designers generally prefer multicomputers over multiprocessors

• From a software perspective significant work has been performed on how to program multiprocessors

• The reverse is true for multicomputers
  – Communication is generally through message passing which makes input/output the central abstraction
Problems with Message Passing

- Message passing enables inter-processor communication by sending messages over a communication network
- Complex issue 1: Flow control
- Complex issue 2: Buffering
- Complex issue 3: Lost messages and network reliability
  - A network is generally less reliable than a bus or switch in a multiprocessor
- Complex issue 4: Blocking
- Complex issue 5: Providing a single view of memory across the distributed system
- Complex issue 6: No central view of time
- **RESULT:** Programming distributed systems with message passing remains tricky
Dilemma

- Multiprocessors are easier to program but harder to design from a hardware perspective

- Multicomputers are easier to design from a hardware perspective, but writing distributed programs is tedious and difficult

- We need systems that are easy to build and easy to program!

- We will look at some issues and problems that can be centrally solved to address building distributed systems that are both easy to build and easy to program
Distributed Shared Memory (DSM)

- Proposed by Li and Hudak in 1989
- Goal of DSM
  - A collection of workstations connected by a network
  - All computers share a single, paged, virtual address space
- Each memory page is present on exactly one machine
- A reference to a local page is done in hardware by the local machine at full memory speed
- A reference to a remote page causes a hardware page fault
  - The OS then sends a message to the proper remote machine that, in turn, sends the page back to the requesting computer
Distributed Shared Memory

- DSM scheme is similar to traditional virtual memory systems
- When a process touches a nonresident page, a trap occurs and the OS fetches the page and maps it in.
- The difference is instead of getting a page from disk, the OS gets it from another machine over the network
- From a user process perspective, DSM makes a multicomputer look like, and behave like a multiprocessor
- All communication and synchronization can be done via memory
  - The DSM environment handles the complexities of the multicomputer environment
DSM Problems

- Performance - many pages are shipped across the network
- Swapping involves network I/O not which is much slower then disk I/O
- Solutions
  - Do not share the entire memory space, only selection portions of it (data and variables)
    - Provides a higher level of abstraction
  - Replicate the shared variables on multiple machines
    - Must keep redundant copies of the same data consistent
  - Share encapsulated data types - Objects
Shared Memory

• There are many types of shared memory
  – Single chip, single bus
  – Advanced configurations with sophisticated caching schemes
Bus Based

- Bus organizations consist of CPU’s that might contain a cache
- Cache improves performance but may lead to consistency problems in the main memory
- Bus must be managed because only one CPU at a time can access the bus
Cache’s and Cache Coherence

• Because the bus may lead to a bottleneck, CPU’s are equipped with caches to reduce bus contention
• Caches improve performance
• Problem, caches are redundant copies of information, thus, how do we keep all copies of the data synchronized
  – Worst case, each processor can have a copy and the memory has a copy
• No problem with caching with read operations
• Write operations must be handled to ensure memory consistency
• Performed with cache coherence protocols
• Popular types of cache coherence protocols:
  – Bus-based
  – Directory-based
Write-Through Cache Consistency Protocol

- Need snoopy cache
  - A snoopy cache is a cache that examines all activity on the bus
- Cache coherence protocols can be modeled by changing state in a state machine based on
  - The local event (read miss, read hit, write miss, write hit)
  - Interception of a remote event by snooping on the bus

<table>
<thead>
<tr>
<th>Event</th>
<th>Local Action</th>
<th>Local Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Miss</td>
<td>Fetch data from memory and store it in cache</td>
<td>No action</td>
</tr>
<tr>
<td>Read Hit</td>
<td>Fetch data from local cache</td>
<td>No action</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Update data in memory and store in cache</td>
<td>No action</td>
</tr>
<tr>
<td>Write Hit</td>
<td>Update memory and cache</td>
<td>Invalidate cache entry</td>
</tr>
</tbody>
</table>
The Write-Once Cache Coherence Protocol

• The write through protocol is not efficient
  – Each write hit causes all writes to use the bus
• Other approaches involve updating the other cached copies instead of invalidating the copies
  – Still very inefficient
• Principle of locality suggests that cached words will probably be needed again soon
• Enable the CPU that is updating the word to “own” the word.
  – Avoid having to notify other CPU’s about updates to the word until another CPU registers interest in the word
• One solution was proposed by Goodman (1983)
  – The Write-Once protocol
Write-Once Cache Coherence Protocols

• Memory words that are being read may be in multiple caches

• A word that is being heavily written by one machine is kept in its local cache
  – No update to main memory or other caches is necessary
  – Eventually another CPU will register interest in the word for reading or writing
    • The local machine delivers the word to the CPU requesting the word

• If no other process registers interest in the word, it remains in the local cache and main memory is inconsistent
  – Eventually the word will be purged from the local cache and written back to main memory

• Write-once can be modeled by enabling memory words to be in one of three states
The Write-Once Cache Coherence Protocol

• Every word kept in cache is “tagged” with a state (may keep the state (tag) in the TLB)
• The write-once protocol defines the following states:
  – **INVALID**: The cache block does not contain valid data - can not use it
  – **CLEAN**: The cache block contains a valid copy of the data, there may be other copies in other caches, and the memory reflects the value in the cache, but it is safe to use
  – **DIRTY**: The cache block contains the only copy of valid data in the system. No other caches contain a valid or clean copy, and the value in memory is incorrect. However from the perspective of the local CPU, the value is safe to use.
• The Write-Once protocol manages state transitions in the caches
The Write-Once Cache Coherence Protocol
Problems with Bus-Based Cache Coherence

- Bus-based cache consistency requires that all caches monitor the bus and “snoop” for consistency commands
- This approach does not scale well
  - Bus becomes a bottleneck
  - Does not work with switched buses or multicomputers because there is no facility to monitor the bus
  - Requires that all caches get involved with managing the enterprise cache coherence protocol
    - Most of the time a cache is not interested in cache activities of other caches
    - Only interested if other caches are performing operations on blocks that are cached by the local processor
- An alternative approach is needed
Directory-Based Cache Coherence

• Directory based cache coherence solves some of the scalability issues associated with bus-based cache coherence techniques

• Directory based cache coherence protocols work by tracking all memory blocks that have copies residing in remote cache blocks

• This information is used to send cache consistency commands to only those caches that have a copy of the block
  – Eliminates the need for global broadcast

• Categories of directory based cache coherence protocols
  – Full-map directories, Limited-map directories, Chained directories

• Every memory page has an individual directory entry
Full Map Directories

- Full map directories store enough information with each global block of memory so that every processor in the system can simultaneously contain the same block of data.

- Key data structure elements used in full-map directories:
  - `clean_dirty_flag`: used to determine if a remote cache had write permissions.
  - `presence_indicator_array`: used to determine if the block resides in a remote cache - this array has one entry for each processor in the system.
  - `write_enable_flag`: used to lock and unlock a local block for write operations.
  - `valid_invalid_indicator`: used to track the validity of a local block. Is the block OK to use?
Full Map Directory Architecture

Directory Block Entry

- Data(X)
- Clean/Dirty Flag
- P1, P2, P3, P4

Cache Block 1
- Valid/Invalid Indicator
- Write Enable Flag
- DATA

Cache Block 2
- Valid/Invalid Indicator
- Write Enable Flag
- DATA

Cache Block 3
- Valid/Invalid Indicator
- Write Enable Flag
- DATA

Cache Block 4
- Valid/Invalid Indicator
- Write Enable Flag
- DATA
Modeling Directory-Based Cache Coherence Protocols

- Cache coherence protocols can be studied by investigating the following events:
  - Read hit
  - Read miss
  - Write hit
  - Write miss
- Only the read miss and write hit are interesting
- Read hits can always provide the desired block from the local cache
- Write misses can be modeled by executing a read miss immediately prior to a write hit
  - The read miss operation loads the block into the local cache
  - The operation then becomes a write hit operation
The Censier and Feautier Protocol

• The Censier and Feautier protocol used two data structures
  – One for each page in the physical memory
  – One for each page in the local caches

structure directory_block
{
  bit clean_dirty_flag;
  bit presence_indicator
    [NUMBER_OF_PROCESSORS];
  byte data[CACHE_BLOCK_SIZE];
}

structure cache_block
{
  bit valid_invalid_indicator;
  bit write_enable_flag;
  byte data[CACHE_BLOCK_SIZE];
}
The Censier and Feautier Protocol

Lighter Line: Read Miss
Darker Line: Write hit (Read Miss & Write Miss)
The Censier and Feautier Protocol

- **Read hit** - the block is in the local cache, thus it may be directly read without any performance overhead

- **Read miss** - the block is not in the local cache, thus the directory for the block is referenced
  - If the clean/dirty flag is set, a remote cache is the owner of the block
    - The remote block might have been updated leaving the memory block inconsistent
    - The directory redirects the request to the block owner
      - The block owner is the only cache entry in the presence indicator array
    - When the remote cache receives the request it clears the write_enable flag
    - The remote cache sends the updated block back to the directory
    - The directory then updates the block, the presence_indicator array, and returns the block to the requesting cache
The Censier and Feautier Protocol

• **Read miss** - Continued
  – If the clean/dirty flag is not set, then the data value in the block is valid and multiple other caches might have copies of the block
  – The directory updates the presence_indicator array and returns the data block back to the requesting cache
  – The requesting cache ensures that the write_enable flag is cleared

• **Write hit** - the data block is in the local cache
  – If the write_enable flag is set then the write can directly be applied to the local cache block
  – If the write_enable flag is not set then the following operations happen:
    • The local cache notifies the directory of its intent to update the block
    • The directory traverses the presence_indicator array and sends an invalidate signal to all caches that have a copy of the memory page
The Censier and Feautier Protocol

• **Write hit** - Continued.
  – If the write_enable flag is not set then the following operations happen: (continued)
    • Upon receiving an invalidate signal all caches mark their copies as invalid
    • The directory sets the clean_dirty flag (we can no longer trust the value in the directory)
    • The directory notifies the requesting cache that it is OK to perform write operations
    • The local cache sets the write_enable flag
    • The local cache performs the actual write

• The Censier and Feautier protocol requires significant (and variable) storage for each directory entry because of the presence_indicator array

• Reconfiguration of directory entries is required if we add additional processors
  – Problems can be solved by an alternative directory-cache coherence protocol
The Stenström Protocol

- The Stenström Protocol is a slight modification of the Censier and Feautier protocol.
- The presence_indicator array is relocated from the directory block to the cache data structure.
- Much better because the number of caches in the system is always much less than the number of directory entries.
- The Stenström protocol requires that if one or more caches have a copy of the directory block then a cache will be marked as the owner of the block.
- Directory block ownership is dynamic and may transition between the caches in the system.
- Every directory block must be owned.
The Stenström Protocol

- The Stenström protocol used two data structures
  - One for each page in the physical memory
  - One for each page in the local caches

```
structure stenstrom_directory_block
{
  bit cache_owner[Log2(NUM_CACHES)];
  byte data[CACHE_BLOCK_SIZE];
}

structure stenstrom_cache_block
{
  bit valid_invalid_indicator;
  bit write_enable_flag;
  bit block_owner;
  bit presence_indicator_array [NUMBER_OF_PROCESSORS];
  byte data[CACHE_BLOCK_SIZE];
}
```
The Stenström Protocol

Lighter Line:  Read Miss
Darker Line:  Write hit (Read Miss & Write Miss)
The Stenström Protocol

- The Stenström protocol contains the following additional fields in its data structures
  - cache_owner: This value in the directory indicates the current owner of the directory block
  - presence_indicator_array: This array is located in the cache data structure rather than the directory data structure
- The directory in the Stenström protocol defers all requests to the current block owner
- The caches may transition block ownership by notifying the directory
- The presence_indicator array is only valid in the current cache that is tagged as the cache owner
The Stenström Protocol

- **Read hit** - the block is in the local cache, thus it may be directly read without any performance overhead

- **Read miss** - the block is not in the local cache, thus the directory for the block is referenced
  - If the directory indicates that the current block is unwound (no cache owner set) then the directory contains a valid copy of the data block
    - The block is provided to the requesting cache
    - The directory updates its cache owner field to the id of the requesting cache
    - The data block is returned to the requesting cache
    - The block_owner flag is set in the requesting cache
    - The presence_indicator_array in the requesting cache is properly initialized
The Stenström Protocol

- **Read miss** - Continued.
  - If the directory indicates that the current block is owned by another cache then the data in the directory block may be inconsistent with the real value of the block
  - The directory block then defers the request to the cache that is the current block owner
  - When the request is received by the block owner the write_enable field is checked
  - If the write_enable flag is set then the cache will clear this flag (prevent future write operations)
    - The presence_indicator array is then updated
    - A copy of the block is then sent by the block owner directly to the requesting cache
  - If the write_enable flag is not set the cache will update the presence_indicator array and send a copy of the block directly to the requesting cache
  - The requesting cache ensures that the write_enable and cache_owner flags are not set
The Stenström Protocol

- **Write hit** - the data block is in the local cache
  - If the write_enable flag is set then the current cache is the block owner and the write can directly be applied to the local cache block
  - If the write_enable flag is not set then the following operations happen:
    - The block_owner field is checked:
      - If the current cache is the block owner then the presence_indicator array is traversed
      - Each cache in the presence_indicator array is sent an invalidate signal
      - Once all remote caches are invalidated, the write_enable flag is set
      - The presence_indicator array is updated
      - The write is applied to the local cache block
The Stenström Protocol

• **Write hit** - the data block is in the local cache - Continued
  
  – If the write_enable flag is not set then the following operations happen: (Continued)
  
  • The block_owner field is checked:
    
    – If the current cache is the not the block owner then local cache notifies the directory of its intent to update the block
    
    – The directory redirects the request to the actual block owner (using the block owner field in the directory data structure)
    
    – The current block owner clears the block owner flag and transmits the presence_indicator array directly to the requesting cache
    
    – The requesting cache now sets the block owner flag and notifies the directory that it is the new block owner
    
    – The requesting cache is now the block owner and can follow the policy of a write hit by the block owning cache
Limited Map Directory Strategies

• Full map directory strategies requires that the cache coherence protocol maintain a presence_indicator array which has one entry for each cache in the system
• Because the size of the presence_indicator array is based on the number of caches, reconfiguring the system to add additional processors may be difficult
• An alternative approach might be to fix the size of the presence_indicator array
• Thus there may a maximum number of cached copies of a directory entry that is less then the total number of caches in the system
  – This constrains the maximum number of cached copies of a directory block
  – Most of the time only few copies (at max) are present in multiple caches
Limited Map Directory Strategies

- Because the size of the presence_indicator array is fixed we need a mechanism to handle a request for a directory block when the presence_indicator array is full
- Can be handled by a replacement strategy such as FIFO or LRU
- If the presence_indicator array is full a cache is invalidated using a replacement strategy
- Once a cache is invalidated the presence_indicator array entry for the invalidated cache can be reused to satisfy the current request
- Because a directory block is unlikely to be shared by many caches at the same time, a limited-map directory strategy has several desirable features
Limited Map Directory Strategies Example

Before Cache 3 requests a copy of Data(X)

After Cache 3 requests a copy of Data(X)
Chained Map Directory Strategies

- Like the limited directory approach, the chained map directory approach attacks the problem of having a large presence_indicator array.
- With chained map directories we use a linked list to track all caches that have a copy of a directory block.
- A linked list approach is very dynamic and does not suffer from the restrictions of a limited directory approach.
  - No need for a replacement strategy.
- Unfortunately a chained map directory approach may be slower than a full map directory and limited map directory because pointers must be traversed to find all cached copies of a directory block.
Chained Map Directory Strategies
SCI - IEEE Standard - P1596

- SCI (Scalable Coherent Interface), IEEE Standard - P1596

Note that a double linked list (forward and backward pointers) are used to speed insert and delete operations on the linked list. The SCI calls the link list a “Sharing List”
Cache Coherence - Summary

- Bus-based techniques are simple and are based on the use of a “snooping” cache that eavesdrops on the bus for events that impact the local cache

- Directory based cache coherence techniques are generally more efficient than bus-based techniques
  - Directory-based techniques scale much better than bus-based techniques
  - Only interested caches get involved in servicing the cache coherence protocol

- Directory based cache coherence protocols work on both multicomputer and multiprocessor architectures

- Enables a single, paged, virtual address space

- Many different optimizations can be made to tradeoff space versus performance
Another Approach

• The distributed memory approaches that we discussed so far attempt to globally optimize a processors access to a memory block in distributed memory
  – Known as Uniform Memory Access (UMA) architectures
  – Need complex cache coherence protocols to correctly operate
• Another approach is based on recognizing the shortcomings of a distributed memory
  – Known as Nonuniform Memory Access (NUMA) architectures
  – No attempt is made to make a local memory read as fast as a remote memory read
  – No need for a complex cache coherence protocol
NUMA Architectures

CPU’s access local memory or remote memory by using a message

Goal is to have programs execute using mostly local memory
How NUMA Works

• KEY: CPU is entirely responsible for its local memory
• When a program makes a memory reference the MMU inspects the upper bits of the memory address
  – If the address is a local address the MMU retrieves the word which might be in memory or in the local cache
  – If the address is a remote address the MMU builds a request packet
    • Write (Address, Value)
    • Read (Address)
  – The local MMU sends the request packet to the CPU that is responsible for that portion of the virtual address space
  – Upon receiving a request the MMU executes the request on behalf of the requesting CPU and sends back appropriate feedback
Benefits of NUMA

- No cache consistency protocol is needed
- Each word in the virtual address space is managed by 1 CPU, 1 MMU, and 1 cache
- Example systems:
  - Cm*
  - BBN Butterfly
- NUMA architectures provide the same capabilities as UMA architectures, but they might be slower
  - One global virtual address space
  - Physical address space may be distributed across many private memories
  - Any process on any CPU can access any page of physical memory, regardless of its location
NUMA Operating System

- Because remote memory reads are much slower than local memory access in NUMA architectures, several things can be done to improve overall performance.
- The OS needs to decide where to allocate physical memory pages for a particular process to maximize performance:
  - Try to get as many needed pages as possible mapped into the local CPU’s memory space.
- When a page fault occurs, the following choices are available:
  - Replicate the page in a local page if the referenced page is read-only.
  - If the page is read-write:
    - Relocate the page to the local processor and invalidate the remote page.
    - Map the address to the remote page.
NUMA Operating Systems

- NUMA operating systems can use an adaptive approach to improve performance over time by using a daemon process
- Daemon gathers statistics about remote and local page usage and relocates pages if warranted
- If a page is moved too often it might result in thrashing
  - May mark the page as frozen for a period of time to stabilize the system
  - It is undesirable to constantly relocate remote pages in a NUMA OS
UMA versus NUMA

Objectives

• Properties of UMA architectures
  – Performance is achieved by having replicated copies of memory pages local to each CPU that is using the memory page
  – Requires a cache coherence protocol to keep all replicated copies of a memory page consistent
  – Cache coherence protocol needs to be fast (typically hardware based)

• Properties of NUMA architectures
  – Accessing remote memory is possible
  – Accessing remote memory is slower than accessing remote memory
  – Remote access times are not hidden by caching
  – No cache coherence protocol is needed
  – Performance can be improved by using an adaptive daemon to relocate pages