Progress Towards Multiprocessors

+ Rate of speed growth in uniprocessors is saturating

+ Modern multiple issue processors are becoming very complex $\rightarrow$ multicores

+ Steady progress in parallel software: the major obstacle to parallel processing
Flynn’s Classification of Parallel Architectures

According to the parallelism in I and D stream

- Single I stream, single D stream (SISD): uniprocessor
- Single I stream, multiple D streams (SIMD): same I executed by multiple processors using diff D
  - each processor has its own data memory
  - there is a single control processor that sends the same I to all processors
  - these processors are usually special purpose
• Multiple I streams, single D stream (MISD) : no commercial machine

• Multiple I streams, multiple D streams (MIMD)
  – each processor fetches its own instructions and operates on its own data
  – usually off the shelf µprocessors
  – architecture of choice for general purpose use
  – Flexible: can be used in single user mode or multiprogrammed
MIMD Machines

1. Centralized shared memory architectures
   - Small number of processors (≈ up to 16-32)
   - Processors share a centralized memory
   - Usually connected in a bus
   - Also called UMA machines (Uniform Memory Access)
Multiprocessors and Thread-Level Parallelism

Sharing a centralized memory becomes less attractive as the number of processors sharing it increases. Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are most often called symmetric (shared-memory) multiprocessors (SMPs), and this style of architecture is sometimes called uniform memory access (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks. Figure 4.1 shows what these multiprocessors look like. This type of symmetric shared-memory architecture is currently by far the most popular organization. The architecture of such multiprocessors is the topic of Section 4.2.

The second group consists of multiprocessors with physically distributed memory. Figure 4.2 shows what these multiprocessors look like. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix E.
2. Machines w/physically distributed memory
   - Support many processors
   - Memory distributed among processors
   - Scales the mem bandwidth if most of the accesses are to local mem
   - Also reduces the memory latency
   - Of course inter processor communication is more costly and complex
   - Often each node is a cluster (bus based multiprocessor)
   - 2 types, depending on method used for inter processor communication:
     1. Distributed shared memory (DSM) or scalable shared memory
     2. Message passing machines or multicomputers
Both direction networks (i.e., switches) and indirect networks (typically multi-dimensional meshes) are used. Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantages for a distributed-memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. As we will see shortly, the use of distributed memory also leads to two different paradigms for interprocessor communication.

Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors.

In the first method, communication occurs through a shared address space, as it does in a symmetric shared-memory architecture. The physically separate memories can be addressed as one logically shared address space, meaning that a
DSMs:
• Memories addressed as one shared address space: processor P1 writes address X, processor P2 reads address X
• Shared memory means that same address in 2 processors refers to same mem location
• Also called NUMA (Non Uniform Memory Access)
• Processors communicate implicitly via loads and stores

Multicomputers:
• Each processor has its own address space, disjoint to other processors, cannot be addressed by other processors
• The same physical address on 2 diff processors refers to 2 diff locations in 2 diff memories
• Each proc-mem is a diff computer
• Processes communicate explicitly via passing of messages among them
e.g. messages to request / send data
to perform some operation on remote data
→ Synchronous msg passing : initializing processor sends a request and waits for a reply before continuing
→ Asynchronous msg passing … does not wait
→ Processors are notified of the arrival of a msg
    → polling
    → interrupt
→ Standard message passing libraries: message passing interface (MPI)
Shared memory communication (DSM)

+ Compatibility w/well understood mechanisms in centralized multiprocessors
+ Easy of programming /compiler design for pgms w/ irregular communication patterns
+ Lower overhead of communication:
  better use of bandwidth when using small communications
+ Reduced remote communication by using automatic caching of data
Msg-passing Communication

+ Simpler hardware (no support for cache coherence in HW)
± communication is explicit → painful
    → forces programmers and compilers to pay attention/ optimize communication

Challenges in Parallel Processing
1) Serial sections
    e.g. To have a speedup of 80 w/100 processor, what fraction of original computation can be sequential?
Amdahl’s law:

\[
\text{Speedup} = \frac{1}{(1 - f_{\text{enh}}) + \frac{F_{\text{enh}}}{S_{\text{enh}}}} = \frac{1}{(1 - f_{\text{parallel}}) + \frac{f_{\text{parallel}}}{100}} = 80
\]

\[f_{\text{parallel}} = 99.75\% \ , \ f_{\text{serial}} = 0.25\%\]

2) Large latency of remote accesses (50-1,000 clock cycles)

Example: 0.5 ns machine has a round trip latency of 200 ns. 0.2% of instructions cause a cache miss (processor stall). Base CPI without misses is 0.5.

What’s new CPI?

\[\text{CPI} = 0.5 + 0.2\% \times \frac{200}{0.5} = 1.3\]
The Cache Coherence Problem

- Caches are critical to modern high-speed processors
- Multiple copies of a block can easily get inconsistent
  - processor writes, I/O writes,...
Hardware Solutions

• The schemes can be classified based on:
  – Shared caches vs Snoopy schemes vs. Directory schemes
  – Write through vs. write-back (ownership-based) protocols
  – Update vs. invalidation protocols
  – Dirty-sharing vs. no-dirty-sharing protocols
Snoopy Cache Coherence Schemes

• A cache coherence scheme based on the notion of a snoop that watches all activity on a global bus, or is informed about such activity by some global broadcast mechanism.

• Most commonly used method in commercial multiprocessors
MSI Scheme

Invalid → Bus Write Miss
Bus invalidate → Shared
P-read → Shared
P-Read → Shared

Invalid → Dirty
P-write \\

Dirty → P-write

P-write → Shared
P-Read \\

Bus-read → Shared

P-read → Dirty

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
</tbody>
</table>
Cache

Bus

Memory

P1: rd x
P1: wr x
P2: wr x
P2: rd x
P1: rd x
P3: rd x
P2: wr x
P1: rd x
<table>
<thead>
<tr>
<th>Access</th>
<th>P1’s Cache</th>
<th>P2’s Cache</th>
<th>P3’s Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P1: wr x</td>
<td>D</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P2: rd x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>P3: rd x</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
</tbody>
</table>
P1: rd x
P1: wr x
P2: wr y
P2: rd x
P1: rd y
P3: rd y
P2: wr x
P1: rd x
<table>
<thead>
<tr>
<th>Access</th>
<th>P1’s Cache</th>
<th>P2’s Cache</th>
<th>P3’s Cache</th>
<th>State for</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>I</td>
<td>I</td>
<td>x</td>
</tr>
<tr>
<td>P1: wr x</td>
<td>D</td>
<td>I</td>
<td>I</td>
<td>x</td>
</tr>
<tr>
<td>P2: wr y</td>
<td>I</td>
<td>D</td>
<td>I</td>
<td>y</td>
</tr>
<tr>
<td>P2: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>x</td>
</tr>
<tr>
<td>P1: rd y</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>y</td>
</tr>
<tr>
<td>P3: rd y</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>y</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>I</td>
<td>D</td>
<td>I</td>
<td>x</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>x</td>
</tr>
</tbody>
</table>
Write Through Schemes

• All processor writes result in:
  – update of local cache and a global bus write that:
    • updates main memory
    • invalidates/updates all other caches with that item

• Advantage: Simple to implement
• Disadvantages: Since ~15% of references are writes, this scheme consumes tremendous bus bandwidth. Thus only a few processors can be supported.

⇒ Need for dual tagging caches in some cases
Write-Back/Ownership Schemes

- When a single cache has ownership of a block, processor writes do not result in bus writes thus conserving bandwidth.
- Most bus-based multiprocessors nowadays use such schemes.
- Many variants of ownership-based protocols exist:
  - Goodman’s write-once scheme
  - Berkley ownership scheme
  - Firefly update protocol
  - ...
Invalidation vs. Update Strategies

1. Invalidation: On a write, all other caches with a copy are invalidated.
2. Update: On a write, all other caches with a copy are updated.
   • Invalidation is bad when:
     – single producer and many consumers of data.
   • Update is bad when:
     – multiple writes by one PE before data is read by another PE.
     – Junk data accumulates in large caches (e.g. process migration).

• Overall, invalidation schemes are more popular as the default.
Illinois Scheme (MESI)

- States: I, VE (valid-exclusive), VS (valid-shared), D (dirty)
- Two features:
  - The cache knows if it has a valid-exclusive (VE) copy. In VE state no invalidation traffic on write-hits.
  - If some cache has a copy, cache-cache transfer is used.
- Advantages:
  - closely approximates traffic on a uniprocessor for sequential programs.
  - In large cluster-based machines, cuts down latency
- Disadvantages:
  - complexity of mechanism that determines exclusiveness
  - memory needs to wait before sharing status is determined
Dirty → Invalid
Invalid → Shared
Shared → Invalid
Invalid → Dirty
Dirty → Shared
Shared → Dirty

Bus Write Miss → Invalid
Valid → Tools
Exclusive → P-read
P-read → Exclusive
Exclusive → Valid
Valid → Exclusive

P-read [someone has it]
P-write
P-read [no one else has it]
P-write
P-read [no one else has it]
P-write
P-read [someone has it]
P-write

Bus-read
P-read
P-write
P-read
P-write
P-read
P-write
P-read
P-write
P-read
P1: rd x
P1: wr x
P2: wr x
P2: rd x
P1: rd x
P3: rd x
P2: wr x
P1: rd x

<table>
<thead>
<tr>
<th>Access</th>
<th>Cache 1</th>
<th>Cache 2</th>
<th>Cache 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: rd x</td>
<td>E</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P1: wr x</td>
<td>D</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P2: rd x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>P3: rd x</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>I</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
</tbody>
</table>
Directory Based Cache Coherence

Key idea: keep track in a global directory (in main memory) which processors are caching a location and the state.
Basic Scheme (Censier and Feautrier)

Read from main-memory by PE_i

- If dirty bit is off then \{read from main memory; turn p[i] ON; \}
- If dirty bit is ON then \{recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE_i (from memory);\}

• Assume K processors
• With each cache-block in memory: K presence bits and 1 dirty bit
• With each cache-block in cache: 1 valid bit and 1 dirty (owner) bit
If dirty-bit OFF then {supply data to PE_i; send invalidations to all PE’s caching that block and clear their P[k] bits; turn dirty bit ON; turn P[i] ON; .. }

If dirty bit ON then {recall the data from owner PE which invalidates itself; (update memory); clear bit of previous owner; forward data to PE_i (from memory); turn bit PE[I] on; (dirty bit ON all the time) }

Write- hit to data valid (not owned ) in cache: {access memory-directory; send invalidations to all PE’s caching block; clear their P[k] bits; turn dirty bit ON ; turn PE[i] ON }

Write 
Miss
Write 
Hit
Non-owned data
P1: rd x
P1: wr x
P2: wr x
P2: rd x
P3: wr x
P1: rd x
<table>
<thead>
<tr>
<th>Access</th>
<th>Directory</th>
<th>Cache 1</th>
<th>Cache 2</th>
<th>Cache 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D  P1  P2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: rd x</td>
<td>0  1  0</td>
<td>V</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P1: wr x</td>
<td>1  1  0</td>
<td>V, D</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>P2: wr x</td>
<td>1  0  1</td>
<td>I</td>
<td>V, D</td>
<td>I</td>
</tr>
<tr>
<td>P2: rd x</td>
<td>1  0  1</td>
<td>I</td>
<td>V, D</td>
<td>I</td>
</tr>
<tr>
<td>P3: wr x</td>
<td>1  0  0</td>
<td>I</td>
<td>I</td>
<td>V, D</td>
</tr>
<tr>
<td>P1: rd x</td>
<td>0  1  0</td>
<td>V</td>
<td>I</td>
<td>V</td>
</tr>
</tbody>
</table>
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.

Each memory block has a home processor and directory.
Home Directory

- Home directory maintains the state bits of the memory block
- Every request (miss/invalidate request) goes through home directory
- If line is dirty, home directory fetches the dirty data (from the appropriate cache), updates home memory and supplies the data back to the requestor and update the state bits
- If line is not dirty, home directory supplies the data to the requestor and updates the state bits
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The \textit{local node} is the node where a request originates. The \textit{home node} is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.

![Diagram showing cache coherence in a distributed-memory multiprocessor.](image-url)
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
Read Miss, Data Dirty

In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.

Figure 4.19
A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intranode and internode communications pass.
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
Write Miss, Data Clean & Shared

In addition to tracking the state of each potentially shared memory block, we must track which processors have copies of that block, since those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The process of invalidating or locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 4.20 shows the type of messages sent among nodes. The local node is the node where a request originates. The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number.
Do It Yourself

- Write miss, data dirty
- Write hit on non-owned block
Synchronization

• Typically → built w/ user level software routines → that rely on hardware-based synch primitives

• Small machines: uninterruptible instruction that atomically retrieves & changes a value.
  Software synchronization mechanisms are then constructed on top of it.

• Large scale machines: powerful hardware - supported synchronization primitives
  Key: ability to atomically read and modify a mem-location
users are not expected to use the hardware mechanisms directly
instead: systems programmers build a synchronization library: locks, etc.

Examples of Primitives

1) Atomic exchange: interchanges a value in a reg for a value in memory
e.g. lock = 0 free
    1 taken
processor tries to get a lock by exchanging a 1 in a register with the lock memory location
• If value returned = 1 : some other processor had grabbed it
• If value returned = 0 : you got it
  no one else can since already 1

2) Test-and-set : return the value in memory & set it to 1

3) Fetch-and increment : return the value in memory & increment it
A Second Approach: 2 instructions

- Having 1 atomic read-write may be complex
- Have 2 instructions: the 2nd instruction returns a value from which it can be deduced whether the pair was executed as if atomic
- e.g. MIPS “load linked” and “store conditional’
  - if the contents of the location read by the LL change before the SC to the same address → SC fails
  - if the processor context switches between the two → SC also fails
– The SC returns a value indicating whether it failed / succeeded
– the LL returns the initial value

atomic exchange :

try : mov R3, R4
    ll   R2,0(R1)
    sc   R3,0(R1)
    beqz R3,try
    mov R4,R2

/* at end : R4 and 0(R1) have been atomically exchanged */
• If another proc intervenes and modifies the value between \( l1, sc \rightarrow sc \) returns 0 (and fails)

• can also be used for atomic fetch-and -increment

\[
\begin{align*}
\text{try : } & l1 & R2,0(R1) \\
& \text{addi } & R3,R2,#1 \\
& \text{sc } & R3,0(R1) \\
& \text{beqz } & R3,\text{try}
\end{align*}
\]
Implementing Locks

• Given an atomic operation → use the coherence mechanisms implement spin locks

• Spin Locks: locks that a processor continuously tries to acquire, spinning in a loop
  + grabs the lock immediately after it is freed
  - tie up the processor

1) If no cache coherence:
   – Keep the lock in memory
   – to get lock: continuously exchange in a loop
• To release the lock: write a 0 to it
daddui R2,R0,#1
lockit: exch R2,0(R1)
    bnez R2,lockit

2) If cache coherence
• try to cache the lock → no need to access memory; can spin in the cache
• “locality’ in lock accesses: processor that last acquired it will acquire it next → will reside in the cache of that processor
• However: cannot keep spinning w/a write → invalidate everyone → bus traffic to reload lock

• Need to do only reads until it sees that the lock is available → then an exchange

  lockit :  
  
  ld    R2,0(R1)  
  bnez  R2,lockit
  daddui R2,R0,#1
  exch  R2,0(R1)
  bnez  R2,lockit
This example shows another advantage of the load linked/store conditional primitives: The read and write operations are explicitly separated. The load linked need not cause any bus traffic. This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock, the `LL` has replaced the `LD`, and the `SC` has replaced the `EXCH`):

\[
\text{lockit: LL R2,0(R1) ;load linked} \\
BNEZ R2,lockit ;not available-spin \\
DADDUI R2,R0,#1 ;locked value \\
SC R2,0(R1) ;store \\
BEQZ R2,lockit ;branch if store fails
\]

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released. We address this issue and other issues for larger processor counts in Appendix H.

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor P0</th>
<th>Processor P1</th>
<th>Processor P2</th>
<th>Coherence state of lock</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Spins, testing if lock = 0</td>
<td>Spins, testing if lock = 0</td>
<td>Shared</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Shared</td>
<td>Bus/directory services P2 cache miss; write back from P0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(Waits while bus/directory busy)</td>
<td>Lock = 0</td>
<td>Shared</td>
<td>Cache miss for P2 satisfied</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td>Shared</td>
<td>Cache miss for P1 satisfied</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Executes swap, gets cache miss</td>
<td>Completes swap: returns 0 and sets Lock = 1</td>
<td>Exclusive (P2)</td>
<td>Bus/directory services P2 cache miss; generates invalidate</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Swap completes and returns 1, and sets Lock = 1</td>
<td>Enter critical section</td>
<td>Exclusive (P1)</td>
<td>Bus/directory services P1 cache miss; generates write back</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Spins, testing if lock = 0</td>
<td></td>
<td></td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>