Multiprocessors

Why Multiprocessors?

- **A growth in data-intensive applications**
- **A growing interest in server markets**
- **Complexity of current microprocessors**
- **Collecting several much easier than redesigning one**
- **Steady improvement in parallel software (scientific apps, databases, OS)**

Long term goal of the field: scale number processors to size of budget, desired performance

Categories

- SISD (Single Instruction stream Single Data stream)
  - Uni-processor
- SIMD (Single Instruction stream Multiple Data stream)
  - Data-level parallelism
  - Each processor has its own data memory, but there is a single instruction memory and control processor, which fetches and dispatches instructions
  - Multimedia, graphics performance
- MISD (Multiple Instruction stream Single Data stream)
  - No commercial product of this type
- MIMD (Multiple Instruction stream Multiple Data stream)
  - Each processor has its own instructions and operates on its own data
  - *Thread level parallelism*

Data Level Parallelism: SIMD

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

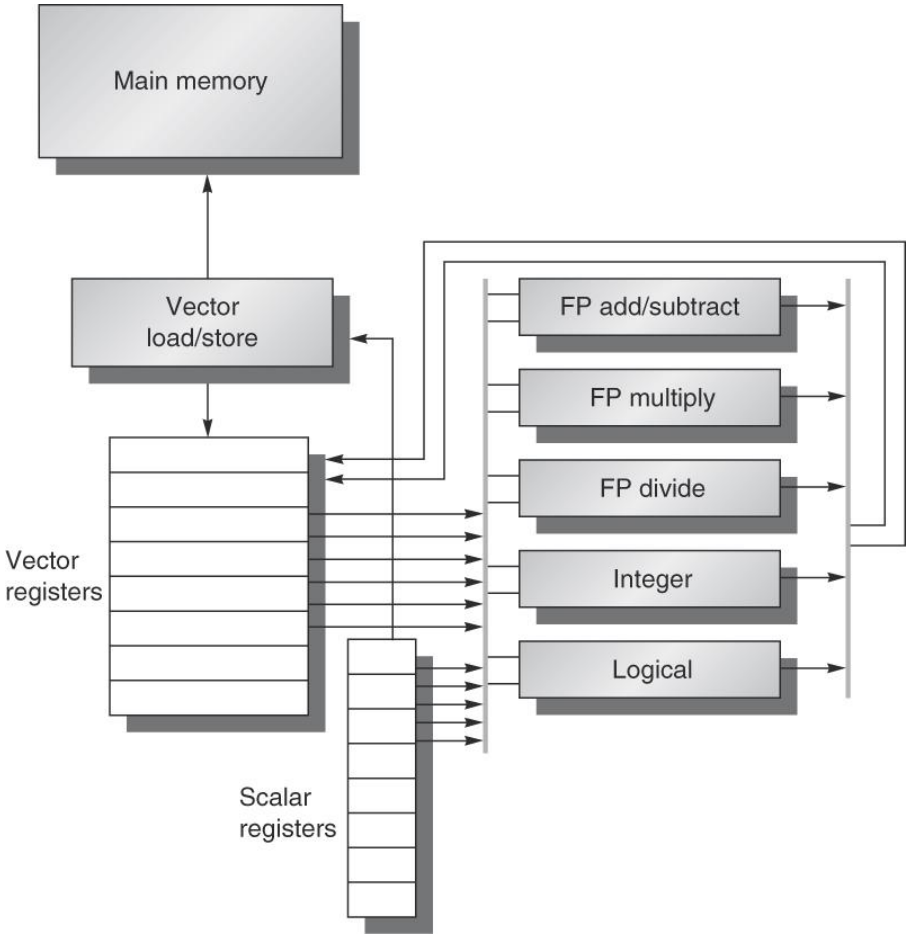
Data Level Parallelism: SIMD

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

The basic structure for a vector architecture



•This processor has a scalar architecture just like MIPS.

•There are also eight 64-element vector registers, and all the functional units are vector functional units

• vector instructions for both arithmetic and memory accesses

•The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations

Example architecture: VMIPS

- Vector registers
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers

VMIPS Instructions

- ADDVV.D V1, V2, V3
  - add two vectors V2 and V3, then put each result in V1
- ADDVS.D V1, V2, F0
  - add scalar F0 to each element of vector V2, then put each result in V1

- SUBVV.D V1, V2, V3
- SUBVS.D V1, V2, F0
- SUBSV.D V1, F0, V2

- MULVV.D, MULVS.D, DIVVV.D, DIVVS.D, DIVSV.D

- LV V1, R1
- SV R1, V1
  - vector load and vector store from address

How vector Processors work: An Example

- Y= a * X + Y

- X and Y are vectors, initially resident in memory
- a is a scalar
- This problem is so-called SAXPY or DAXPY loop in benchmark (Single/Double precision a*X Plus Y)

MIPS code

```
        L.D     F0, a           ; load scalar a
        DADDIU  R4,Rx,#512      ; last address to load
Loop:   L.D     F2, 0(Rx)       ; load X[i]
        MUL.D   F2, F2, F0      ; a*X[i]
        L.D     F4, 0(Ry)       ; load Y[i]
        ADD.D   F4,F4,F2        ; a * X + Y
        S.D     F4, 0(Ry)       ; store into Y[i]
        DADDIU  Rx,Rx,#8        ; increment index to X
        DADDIU  Ry,Ry,#8        ; increment index to Y
        DSUBU   R20,R4,Rx       ; compute bound
        BNEZ    R20, Loop       ; check if done
```

VMIPS code for DAXPY

```
L.D       F0,a       ; load scalar a
LV        V1,Rx      ; load vector X
MULVS.D   V2,V1,F0   ; vector-scalar multiply
LV        V3,Ry      ; load vector Y
ADDVV     V4,V2,V3   ; add
SV        Ry,V4      ; store the result
```

- Requires 6 instructions vs. almost 600 for MIPS

Comparison of MIPS and VMIPS code

- The overhead instructions are not present in the VMIPS code
- The compiler produces vector instructions for such a sequence: The code is **vectorized** or **vectorizable**
- Loops can be vectorized if there are no loop-carried dependences
- VMIPS has fewer pipeline load interlocks
- Vector architects call forwarding of element-dependent operations “chaining”

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together

Chimes

- Sequences with RAW dependency hazards can be in the same convoy via *chaining*

- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

- *Chime*
  - Unit of time to execute one convoy
  - *m* convoys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

Show how the following sequence lays out in convoys, assuming a single copy of each vector functional unit

```
LV        V1,Rx      ;load vector X
MULVS.D   V2,V1,F0   ;vector-scalar multiply
LV        V3,Ry      ;load vector Y
ADDVV.D   V4,V2,V3   ;add two vectors
SV        Ry,V4      ;store the sum
```

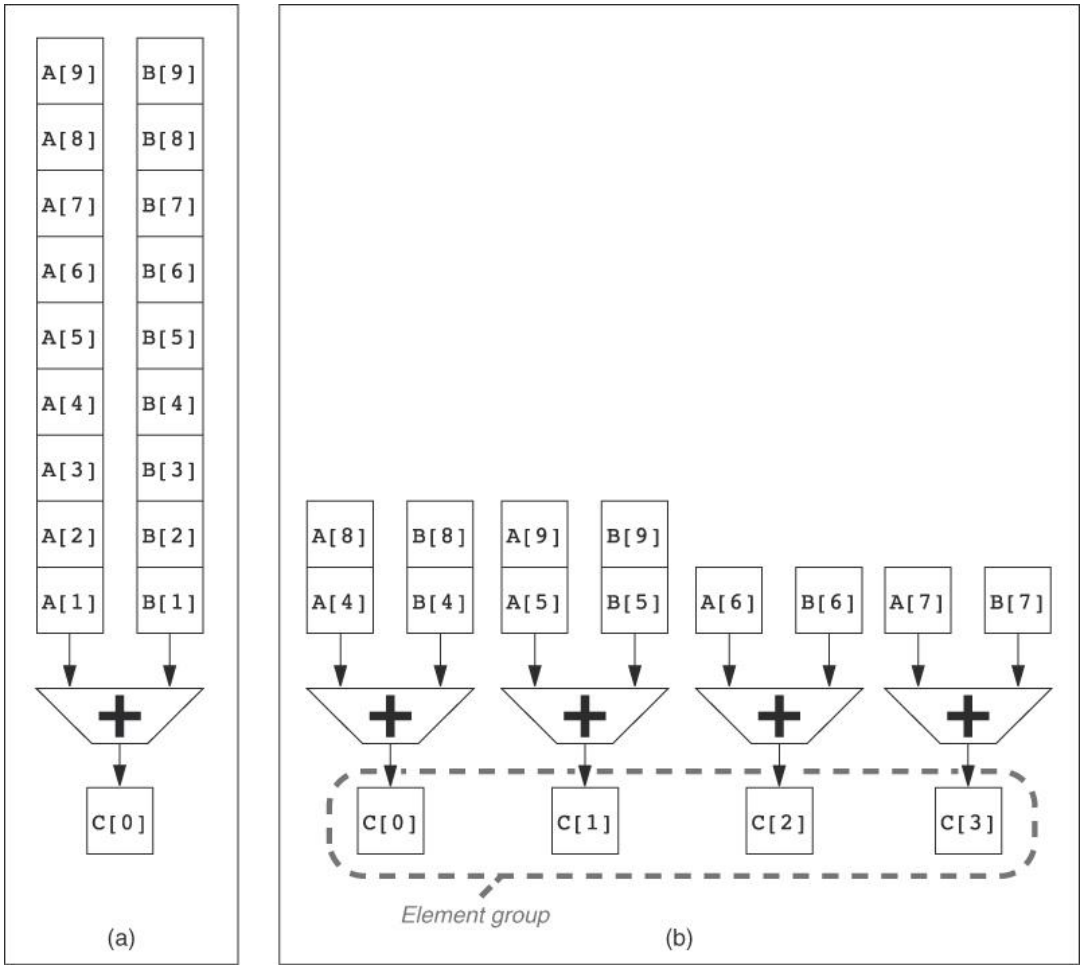
Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

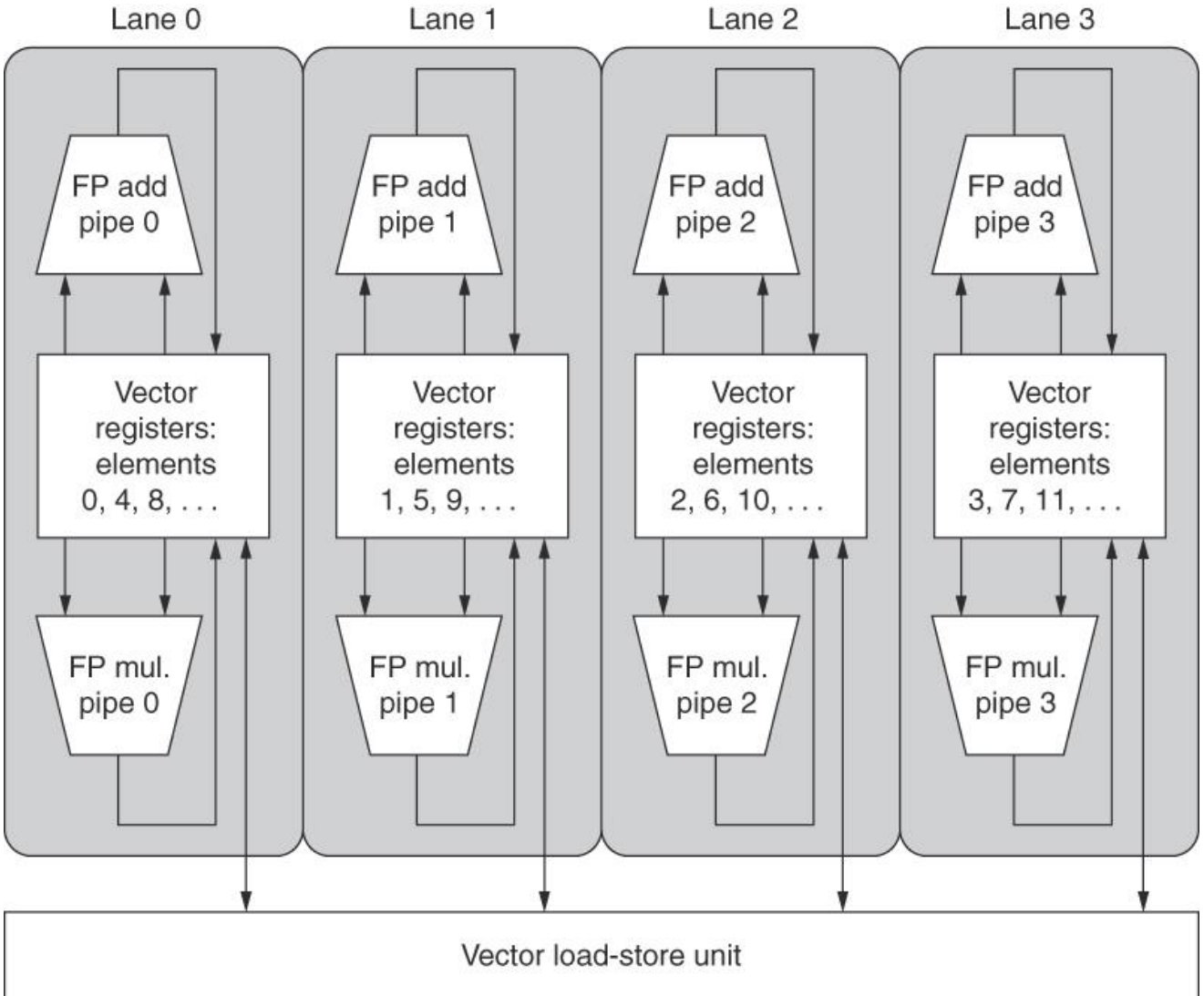
- 3 chimes
- For 64 element vectors, requires 64 x 3 = 192 clock cycles

Multiple Lanes

- Using parallel pipelines
- Allows for multiple hardware lanes

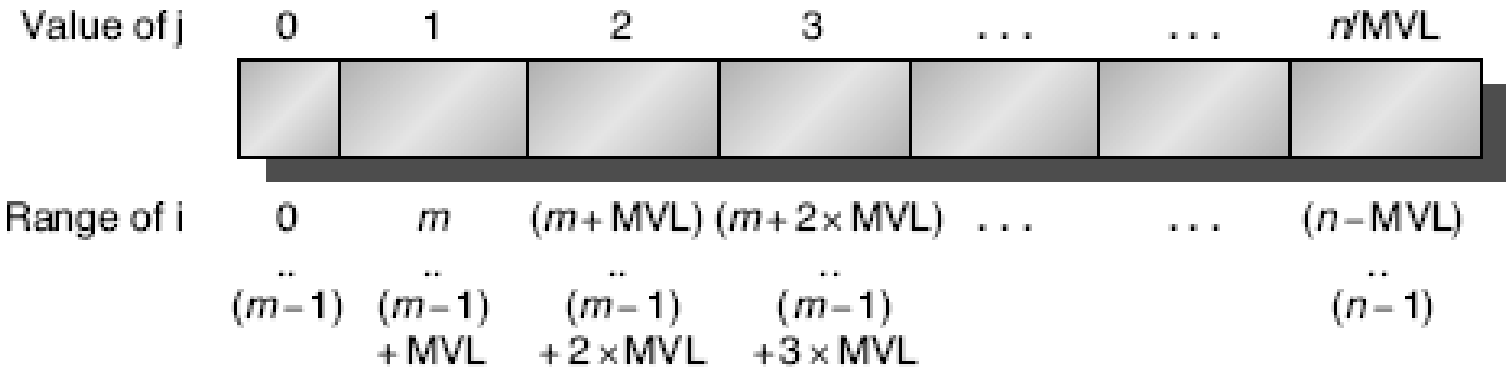


Structure of a vector unit containing four lanes



Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum vector length (MVL):



Strip mining

- Generate code such that each vector operation is done for a size less than or equal to the MVL
- In the figure, m=n%MVL
- The first segment's length=n%MVL, all subsequent segments are of length MVL

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
        for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
                Y[i] = a * X[i] + Y[i] ; /*main operation*/
        low = low + VL; /*start of next vector*/
        VL = MVL; /*reset the length to maximum vector length*/
}
```

Vector Mask Registers: Handling "If" statements in vector loops

- Consider:

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

- Use vector mask register to "disable" elements:

```
LV          V1,Rx       ;load vector X into V1
LV          V2,Ry       ;load vector Y
L.D         F0,#0       ;load FP zero into F0
SNEVS.D     V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2    ;subtract under vector mask
SV          Rx,V1       ;store the result in X
```

MIMD

- MIMD offers flexibility
- With correct hardware and software support, MIMDs can function as
  - Single-user multiprocessors focusing on high performance for one application
  - Multi-programmed multiprocessors running many tasks simultaneously
  - Or some combination of these functions
- MIMD can build on the cost-performance advantages of off-the-shelf processors
  - Nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.
  - Multi-core chips leverage the design investment in a single processor core by replicating it

MIMD - Clusters

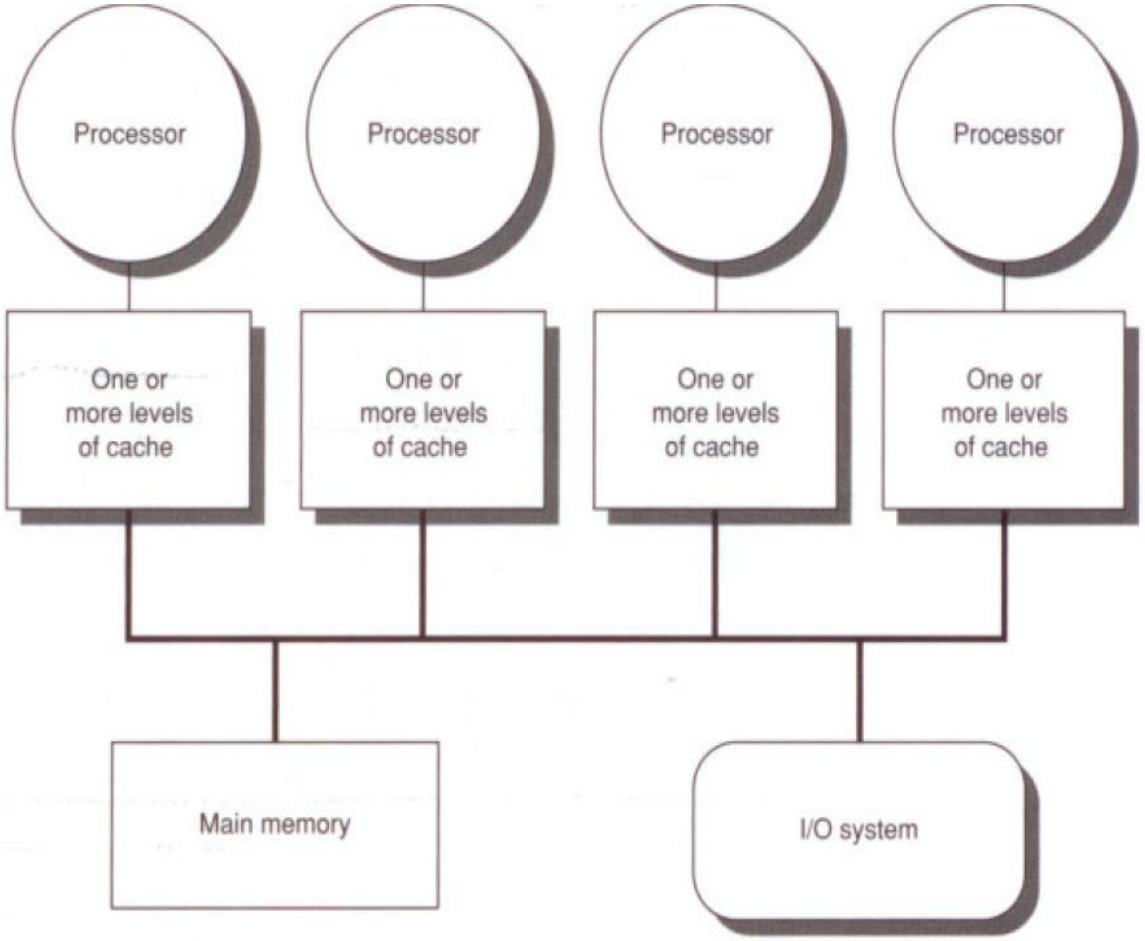
- One popular class of MIMD computers are clusters, which often use standard components and standard network technology
- Clusters
  - Commodity clusters
    - Often assembled by users or computer center directors, rather than vendors
    - For applications focusing on throughput and requiring almost no communication among threads
    - Web serving, multiprogramming, transaction-processing applications
    - Inexpensive
  - Custom clusters
    - A designer customizes the detailed node design or the interconnection network
    - For parallel applications that can exploit large amounts of parallelism on a single problem
    - Such applications require significant amount of communication during computation

MIMD - Multicore

- On-chip multi-processing (single-chip multiprocessing, multi-core)
  - Place multiple processors on a single die, typically share L2 cache, memory or I/O bus
- Usually, each processor execute a different process
  - Process: (usually) Independent of each other
  - Thread: May share code and address space

Major MIMD Styles

1. Centralized shared memory (Symmetric (shared-memory) multiprocessors) ("Uniform Memory Access" time) or ("Shared Memory Processor")

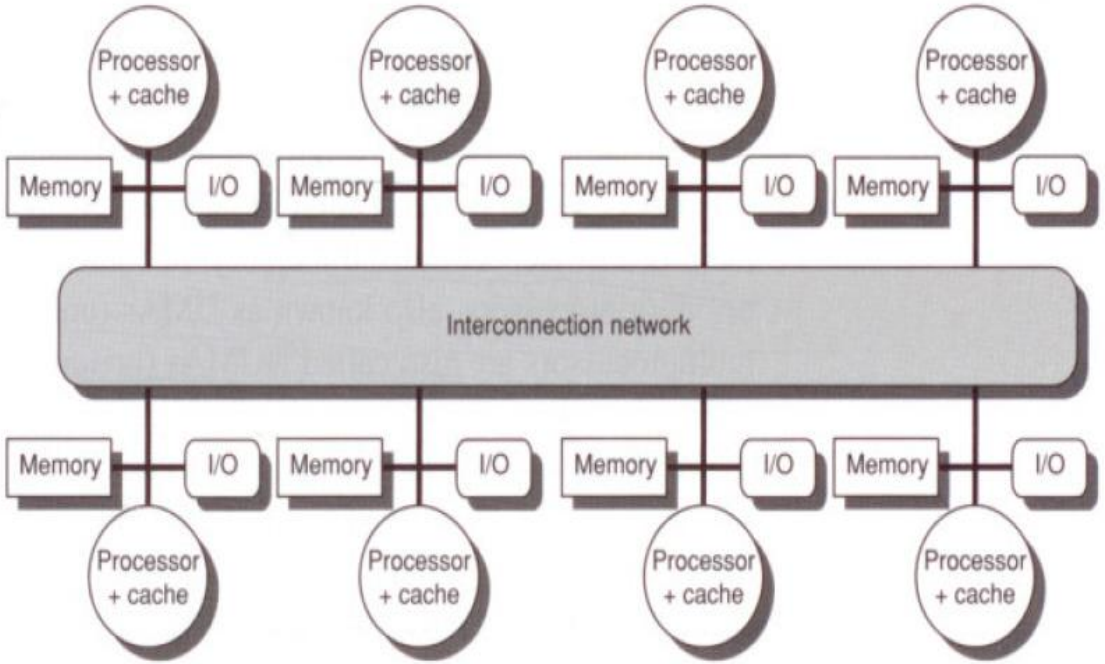


- Uniform latency from memory

Major MIMD Styles: 2. Decentralized memory

memory module with CPU

- Get more memory bandwidth
- Lower memory latency
- Major Drawback: Longer communication latency
- Drawback: Software model more complex



Models for communication

- 1. Communication through a shared address space
  - The physically separate memories can be addressed as one logically shared address space
  - Called Distributed shared-memory
  - For non-uniform memory access – access time depends on location of data
- 2. Communication through message passing
  - Address spaces are logically disjoint and cannot be addressed by a remote processor

Symmetric Multi-Processors

- The large, multilevel caches reduce the memory demand of a processor
- More than one processor to share the bus to the memory
- Different organizations:
  - Processor and cache on an extension board
  - Integrated on the main-board (most popular)
  - Integrated on the same chip
- Private data
  - Used by single processor
- Shared data
  - Used by multiple processor
  - Provide communication among processors through read/write the shared data
- The new problem: cache coherence

The cache coherence problem for a single memory location (X), read and written by two processors (A and B)

Time	Event	Cache Content for CPU A	Cache Content for CPU B	Memory content for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"

- Formally: A memory system is coherent if
  - P writes x and P reads x. If no other writes of x by another processor between the write and the read, the read always return the value written by P.

  - If P1 writes x and P2 reads it, P1's write will be seen by P2 if the read and write are sufficiently far apart and no other writes to x between the two accesses

  - Writes to a single location are serialized:
    seen in one order (Latest write will be seen)

Coherence and Consistency

- Coherence: defines what values can be returned by a read

- Consistency: defines when a written value will be returned by a read
  - We cannot require that a read of X instantaneously see the value written for X by some other processor
  - The issue of exactly when a written value must be seen by a reader is defined by a memory consistency model

Coherence Solutions

- **Snooping:**
  - **No centralized state is kept.**
  - **Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block**
  - **Caches are on a shared-memory bus**
  - **All cache controllers monitor or snoop on the bus to determine if they have a copy of a block that is requested on the bus**
  - **Dominates for small scale machines (most of the market)**

- **Directory-Based Schemes**
  - **The sharing status of a block is kept in one location called directory**
  - **Keep track of what is being shared in one place (centralized logically, but can be distributed physically)**
  - **Has slightly higher implementation overhead than snooping**
  - **Can scale to larger processor counts**

Basic Snooping Protocols

- **Write Invalidate Protocol:**
  - **Multiple readers, single writer**
  - **Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies**
  - **Read Miss:**
    - » **Write-through: memory is always up-to-date**
    - » **Write-back: snoop in caches to find the most recent copy**

- **Write Broadcast Protocol**
  - **Also called Write Update Protocol**
  - **Write to shared data: broadcast on bus, processors snoop, and *update* any copies**
  - **Read miss: memory is always up-to-date**
  - **typically write through**

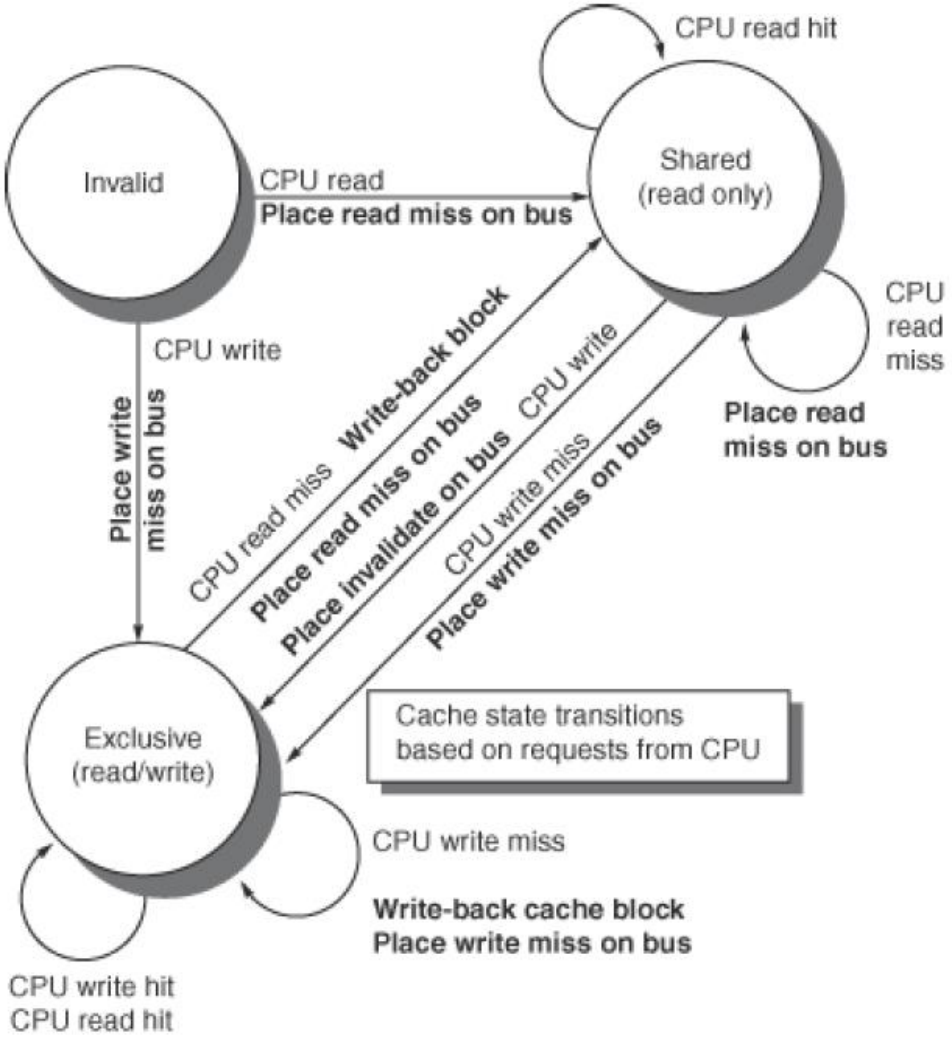
Example of Write Invalidation on a snooping bus

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

An Example Snooping Protocol

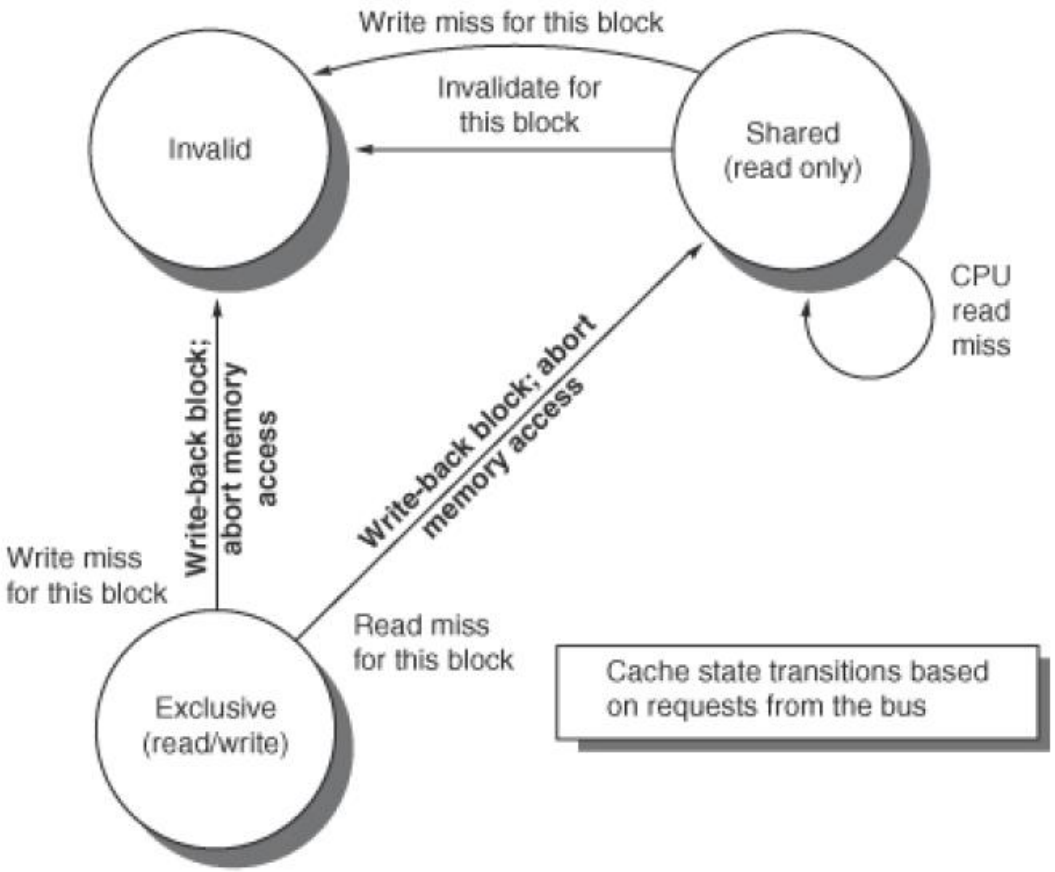
- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
  - **Clean in all caches and up-to-date in memory (Shared)**
  - **Dirty in exactly one cache (Exclusive)**
  - **Not in any caches**
- **Each cache block is in one state (track these):**
  - **Shared : block can be read**
  - **Exclusive : cache has the only copy, it's writeable, and dirty**
  - **Invalid : block contains no data**

Cache state transitions based on requests from CPU

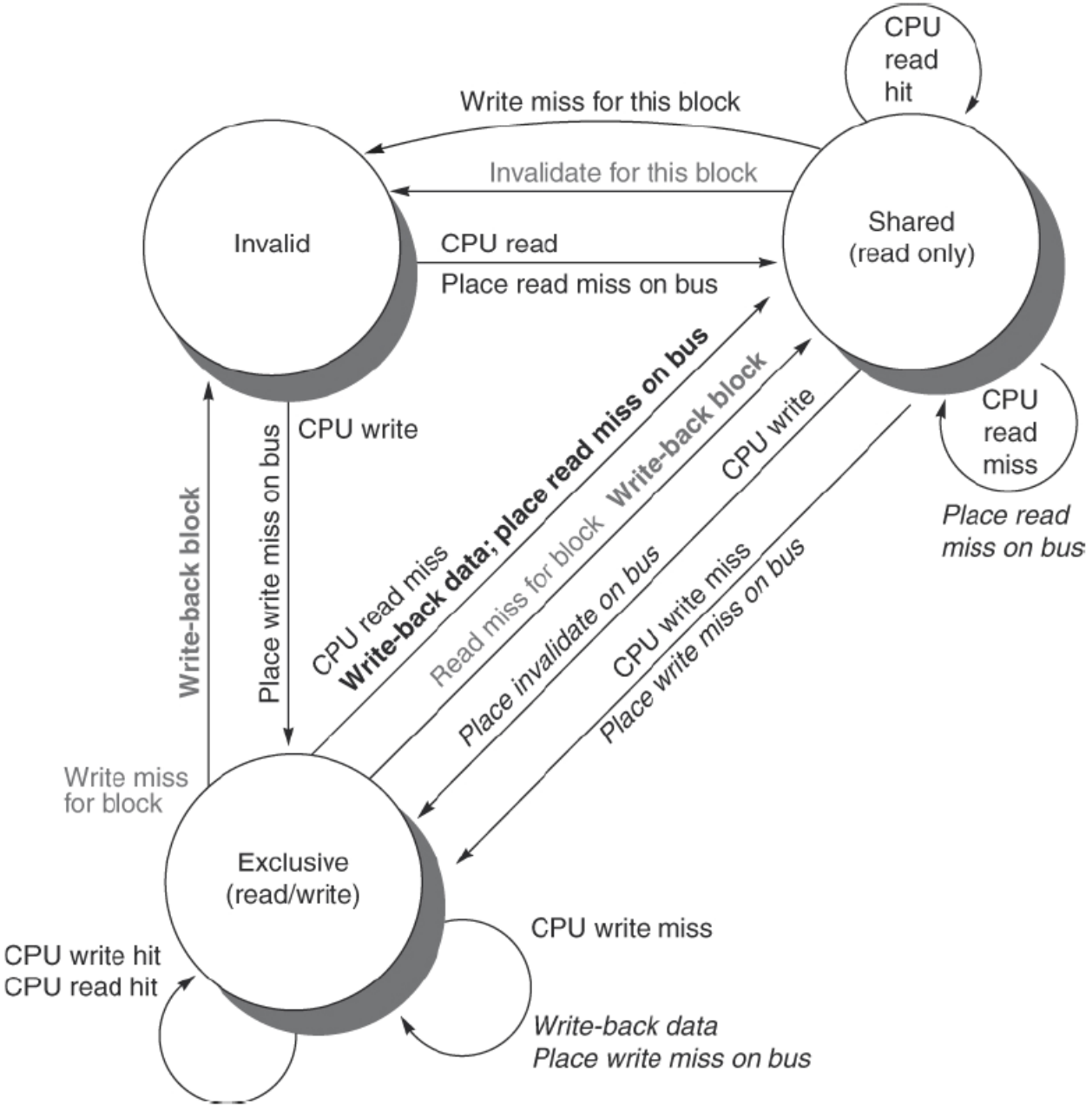


Boldface: bus actions generated as part of the state transition

Cache state transitions based on requests from bus



Cache Coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray



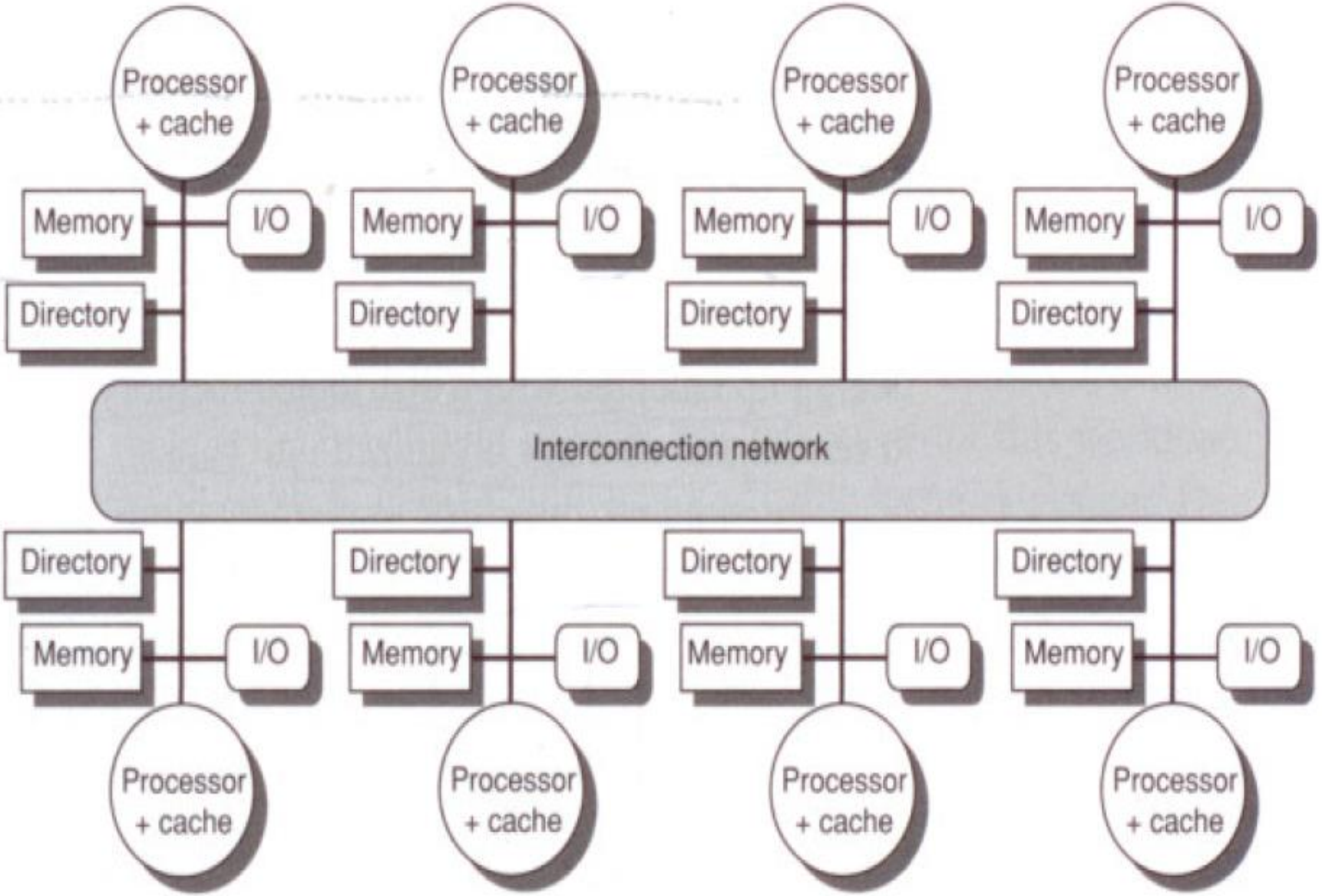
Exercise

- Suppose two processors P1 and P2 are both in Shared state for a data block X.
- P2 writes X at time T1.
- Then, P1 writes X at time T2.
- Afterwards, P2 reads X at time T3.
- Please write down and explain the state transitions, actions on the bus, and requests of P1 and P2 at time T1, T2, and T3.

Time	CPU request	Bus Activity	State Transitions of P1	State Transitions of P2
			Action of P1	Action of P2
T1: P2 writes X	CPU Write	Invalidate for X	Share -> Invalid	Shared -> Exclusive
				Place Invalidate on Bus
T2: P1 writes X	CPU Write	Write Miss for X	Invalid -> Exclusive	Exclusive -> Invalid
			Place Write Miss on Bus	Write Back Block; Abort Mem Access
T3: P2 reads X	CPU Read	Read Miss for X	Exclusive -> Shared	Invalid -> Shared
			Write Back Block; Abort Mem Access	Place Read Miss on Bus

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information


Processor + cache
Memory
I/O
Directory
Interconnection network

Directory-based coherence

- Local node: requesting node
- Home node: where the memory location and the directory entry of an address reside
- Remote node: have a copy of the cache block, whether exclusive or shared

Directory based Cache Coherence Protocol

- Similar to Snooping Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has a copy of the cache block; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)

A Popular Middle Ground

- **Two-level “hierarchy”**
- **Individual nodes are multiprocessors, connected non-hierarchically**
- **Coherence across nodes is directory-based**
- **Coherence within nodes is snooping or directory**

Challenges of Parallel Processing

- **1. Limited parallelism available in programs**
  - **Running independent tasks**
- **2. High cost of communications**
  - **Threads must communicate to complete the task**
  - **Large latency of remote access**
  - **In existing shared-memory multiprocessors, communication cost: from 50 clock cycles (multicore) to 1000 clock cycles (large-scale multiprocessors)**

Example 1

- **Suppose you want to achieve a speedup of 80 with 100 processors.**
- **What fraction of the original computation can be sequential?**

Answer to Example 1

- **Suppose you want to achieve a speedup of 80 with 100 processors.**
- **What fraction of the original computation can be sequential?**

$$speedup = \frac{1}{\frac{Fraction_{enhanced}}{speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

$$80 = \frac{1}{\frac{Fraction_{parallel}}{100} + (1 - Fraction_{parallel})}$$

$$80 Fraction_{parallel} + 8000 - 8000 Fraction_{parallel} = 100$$

$$7920 Fraction_{parallel} = 7900$$

$$Fraction_{parallel} = 99.75\%$$

$$Fraction_{sequential} = 0.25\%$$

Example 2

- Suppose we have an application running on a 32-processor multiprocessor
- Takes 200ns to handle reference to a remote memory
- Assume all local references hit in the local memory hierarchy (except those involving communication)
- Processors are stalled on a remote request
- Processor clock rate 2GHz
- Base CPI=0.5 (the case when all references hit in the cache)
- How much faster for (A) versus (B)
  - (A) no communication is needed
  - (B) 0.2% of the instructions involved a remote communication reference

Answer for Example 2

(A) no communication is needed

(B) 0.2% of the instructions involved a remote communication reference

- **CPI(A) = 0.5**

- **Effective CPI(B)**

  = Base CPI + Remote request rate x Remote request cost

  = 0.5 + 0.2% x (200ns/0.5ns) = 0.5 + 0.2%*400 = 1.3

  - **Processor clock rate 2GHz => 1 clock cycle: 0.5 ns**
  - **Remote request cost = 200ns/0.5ns = 400 cycles**

- **How much faster for (A) versus (B)?**

  1.3/0.5 = 2.6 times faster

  (only 0.2% of instructions involved a remote access would still cause a big impact)

Conclusion

- **Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)**

- **Snooping cache over shared medium for smaller number of Multi-Processor**
- **Snooping => uniform memory access; bus makes snooping easier because of broadcast**

- **Directory has extra data structure to keep track of state of all cache blocks**
- **Distributing directory => scalable shared address multiprocessor**
  =>Non uniform memory access