Chapter 5

Thread-Level Parallelism

Instructor: Josep Torrellas

CS433

Copyright Josep Torrellas 1999, 2001, 2002, 2013

Progress Towards Multiprocessors

+ Rate of speed growth in uniprocessors saturated
+ Wide-issue processors are very complex
+ Wide-issue processors consume a lot of power
+ 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
  - Architecture of choice for general purpose mps
  - Flexible: can be used in single user mode or multiprogrammed
  - Use of the shelf μprocessors

MIMD Machines

1. Centralized shared memory architectures
   - Small #'s of processors ($\approx$ up to 16-32)
   - Processors share a centralized memory
   - Usually connected in a bus
   - Also called UMA machines (Uniform Memory Access)
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

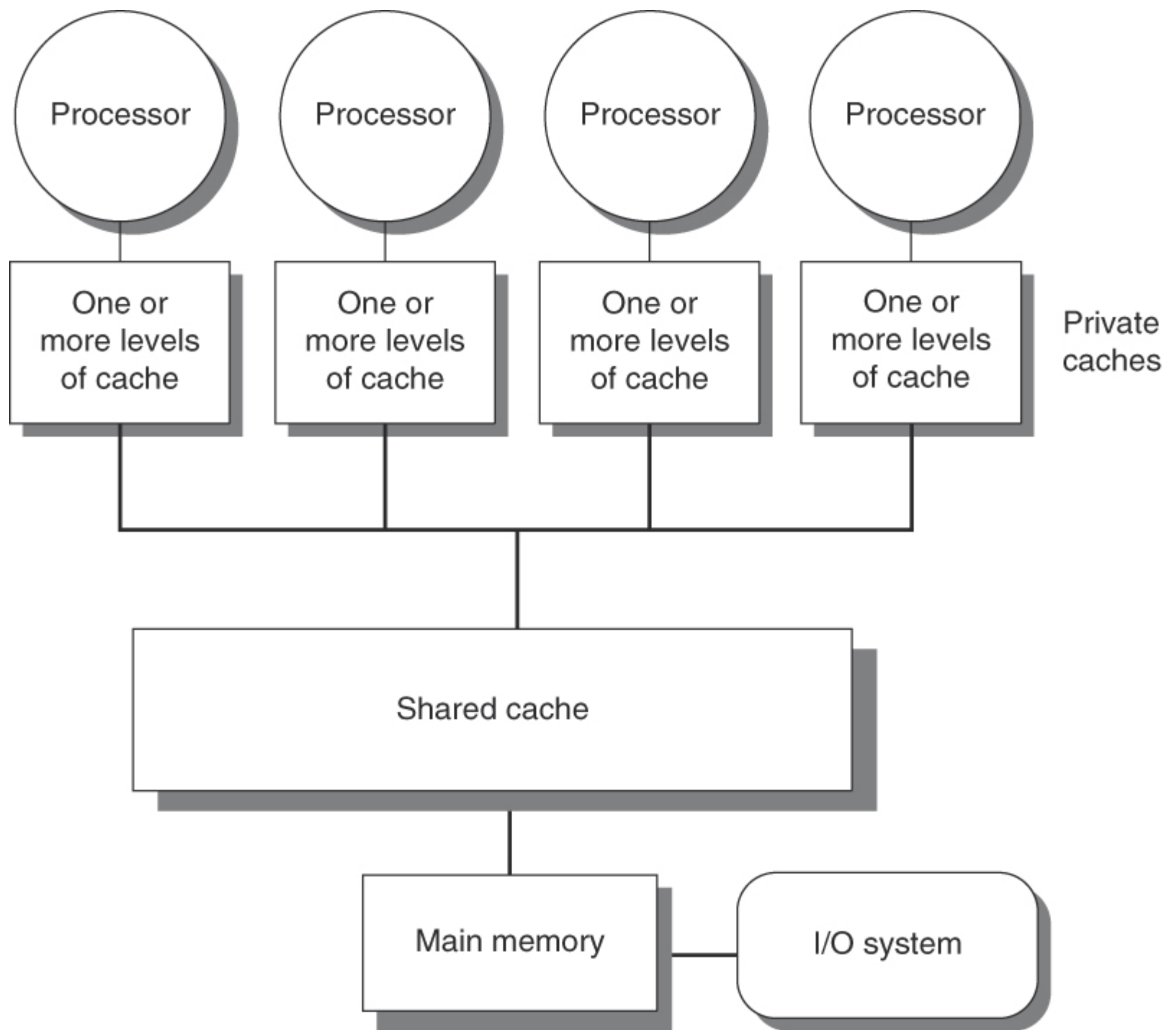


Figure 5.1

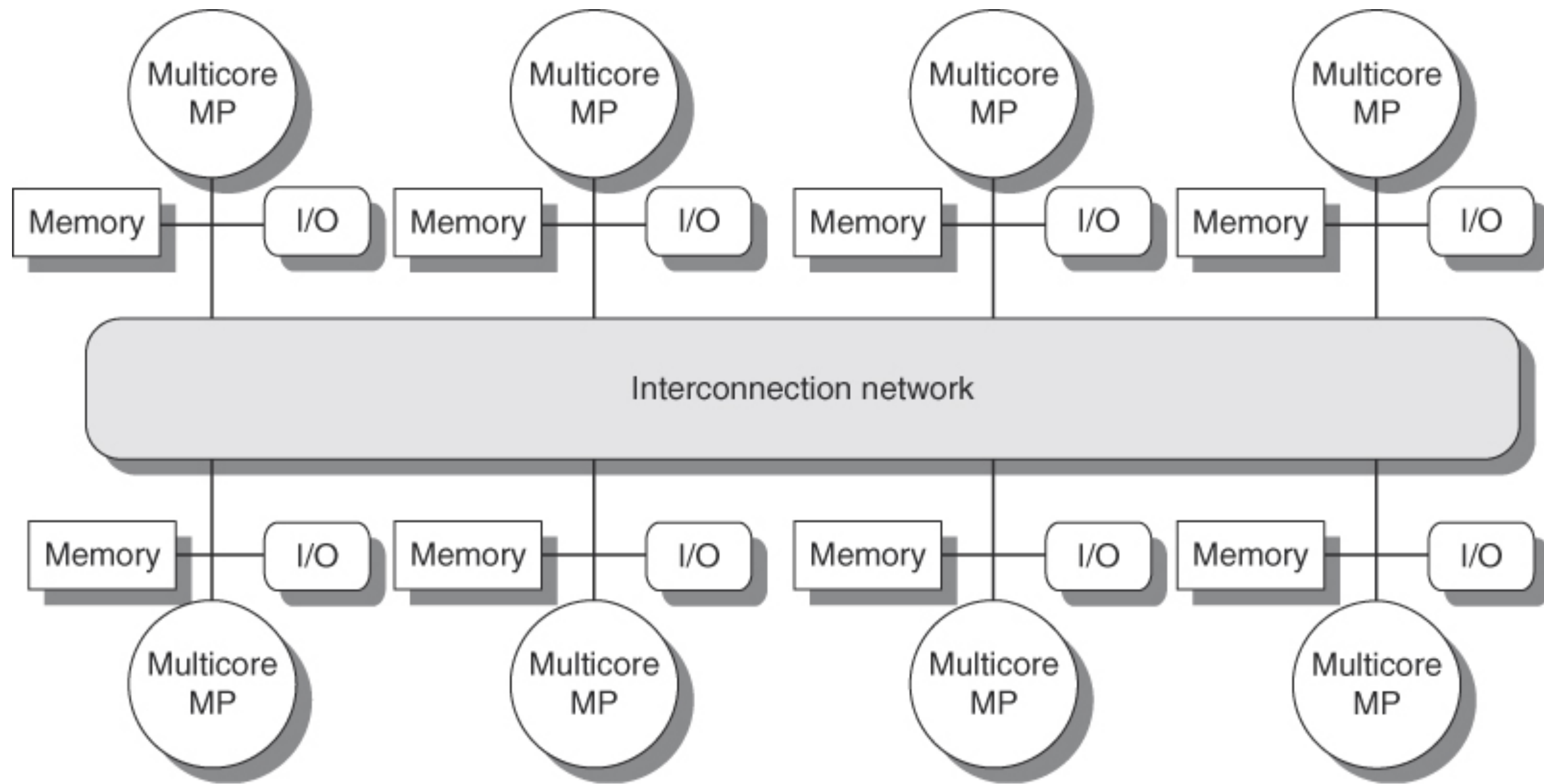


Figure 5.2

2. Machines w/physically distributed memory (cont)

- Also reduces the memory latency
- Of course interprocessor communication is more costly and complex
- Often each node is a cluster (bus based multiprocessor)
- 2 types, depending on method used for interprocessor communication:
  1. Distributed shared memory (DSM) or scalable shared memory
  2. Message passing machines or multicomputers

DSMs :

- Memories addressed as one shared address space: processor P1 writes address X, processor P2 reads address X
- Shared memory means that some address in 2 processors refers to same mem location; not that mem is centralized
- 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

+ Compatibility w/well understood mechanisms in centralized mps
+ 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:

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

$f_{parallel} = 99.75\%$ $f_{parallel} = 0.25\%$

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

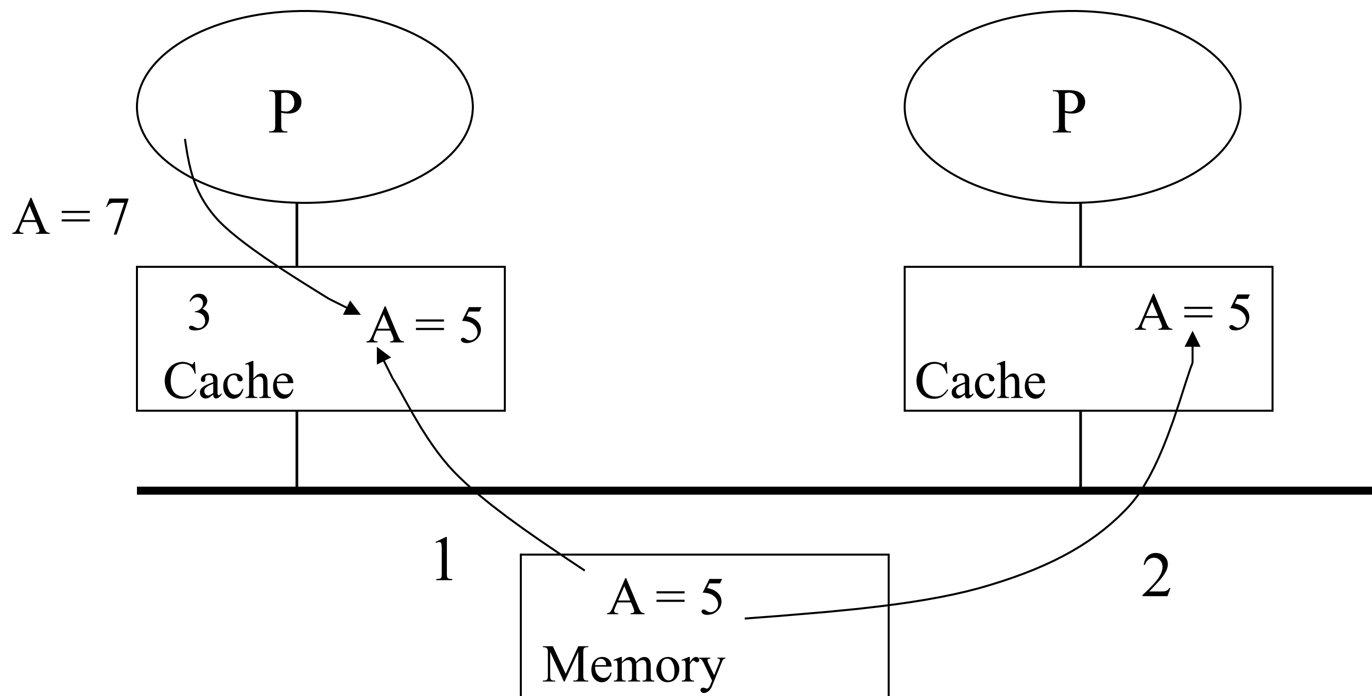
Example : 0.5 ns machine has a round trip latency of 200 ns. 0.2% of instructions cause a cache miss (processor stall). Base CPI without misses is 0.5.

Whats new CPI ?

CPI = 0.5 + 0.2% * 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 :
  - Snoopy schemes vs. Directory schemes
  - Write through vs. write-back (ownership-based) protocols
  - Update vs. invalidation protocols

Snoopy Cache Coherence Schemes

- A distributed 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.

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.

  $\Rightarrow$ Need for dual tagging caches in some cases

Write-Back/Ownership Schemes

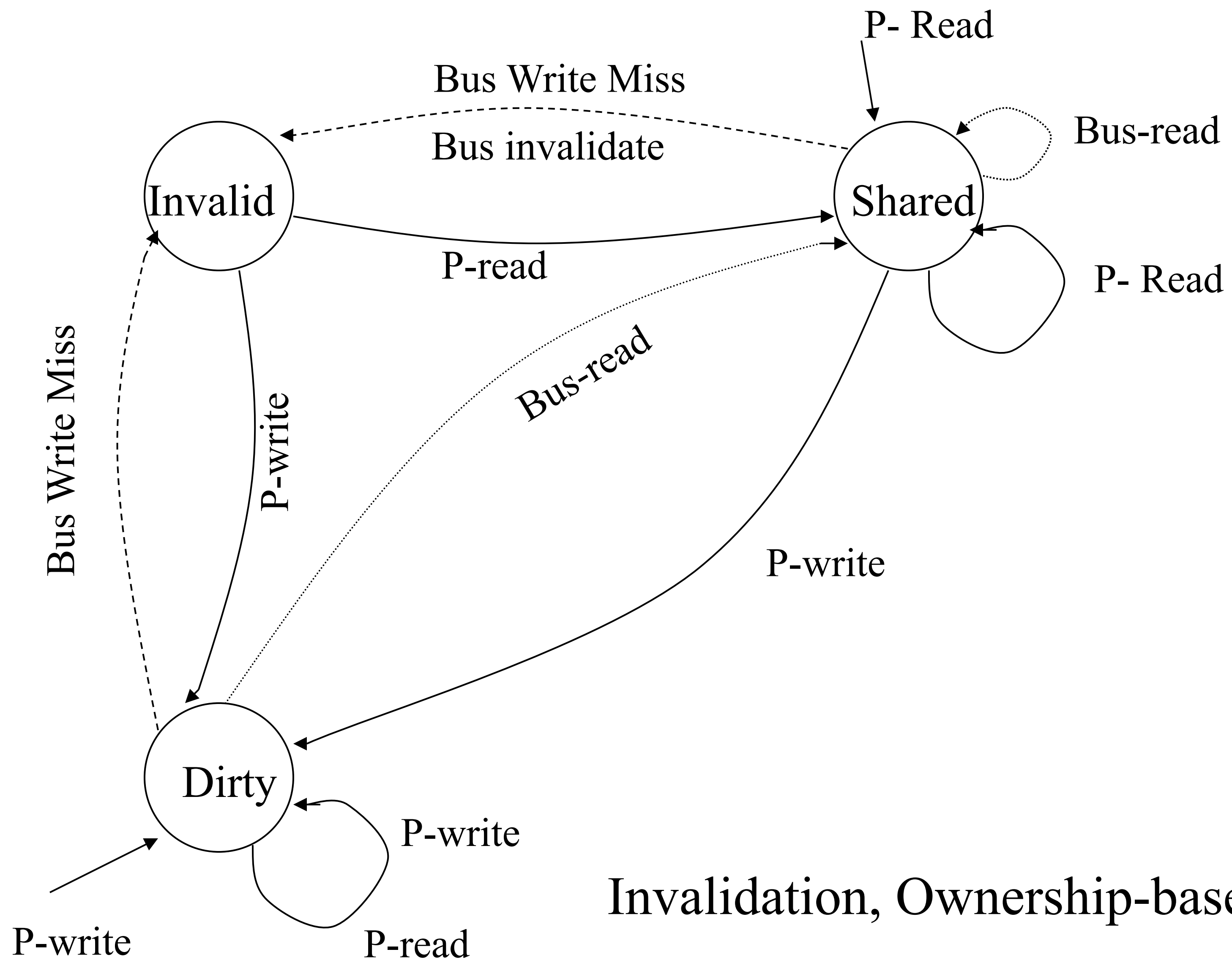
- When a single cache has ownership of a block, processor writes do not result in bus writes thus conserving bandwidth.
- Most bus-based multiprocessors nowadays use such schemes.
- Many variants of ownership-based protocols exist:
  - Goodman's write -once scheme
  - Berkley ownership scheme
  - Firefly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

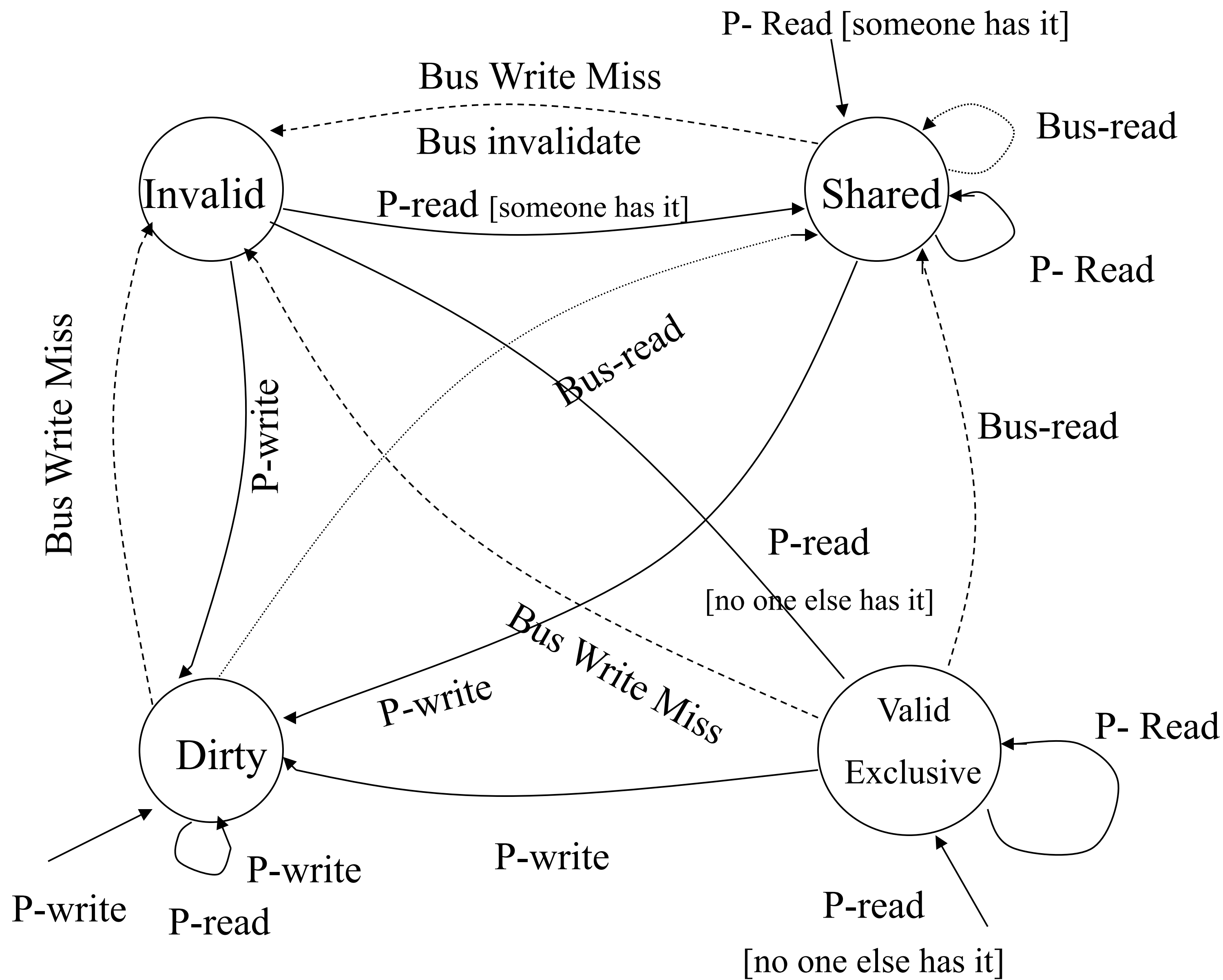
- Overall, invalidation schemes are more popular as the default.



Invalidation, Ownership-based

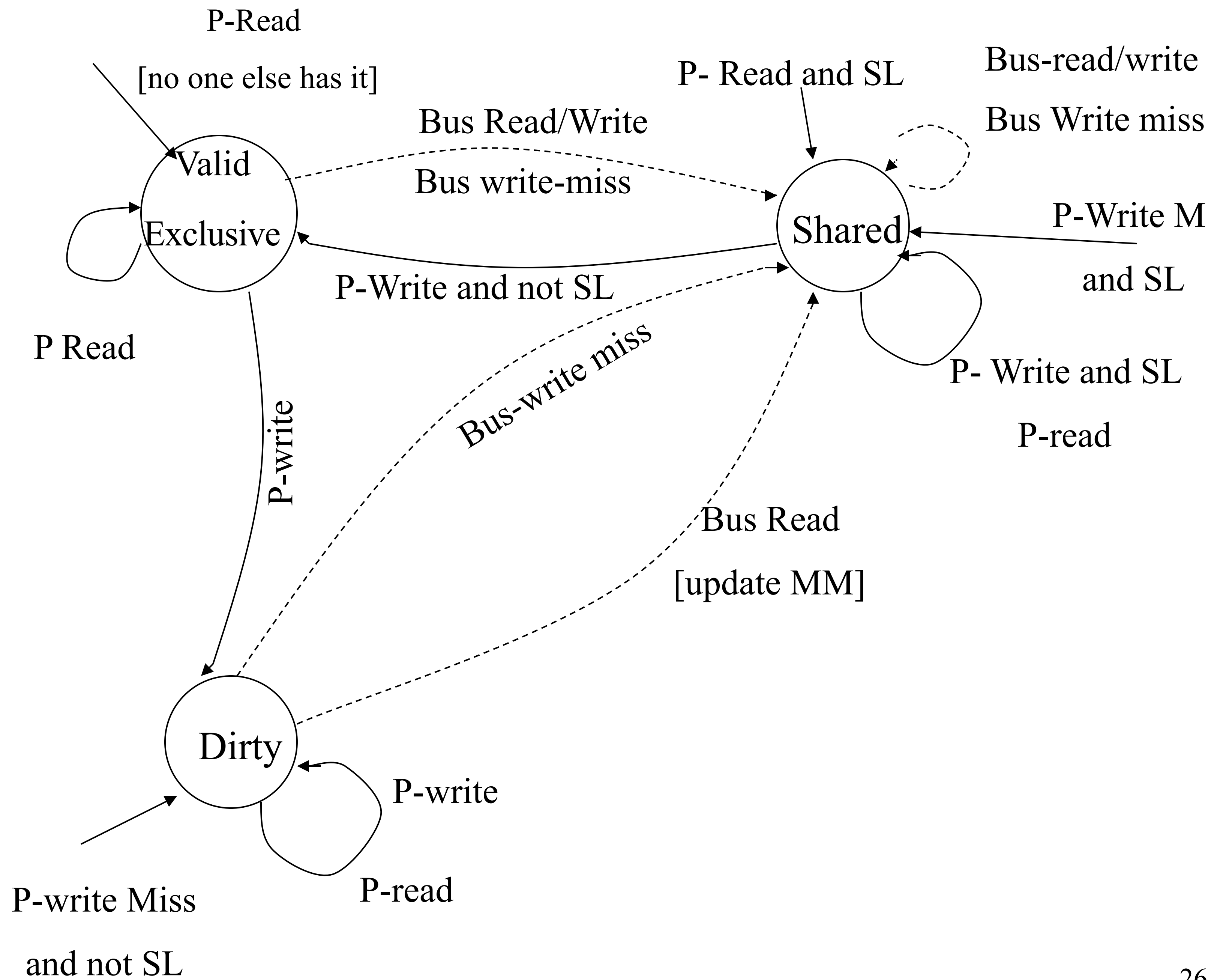
Illinois Scheme

- States: I, VE (valid-exclusive), VS (valid-shared), D (dirty)
- Two features :
  - The cache knows if it has an 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 pgms.
  - In large cluster-based machines, cuts down latency
- Disadvantages:
  - complexity of mechanism that determines exclusiveness
  - memory needs to wait before sharing status is determined


P- Read [someone has it]
Bus Write Miss
Bus invalidate
Bus-read
Invalid
Shared
P-read [someone has it]
P- Read
Bus-read
Bus Write Miss
P-write
Bus-read
P-read
[no one else has it]
Bus Write Miss
P-write
Valid
Exclusive
P- Read
Dirty
P-write
P-write
P-write
P-read
P-read
[no one else has it]

DEC Firefly Scheme

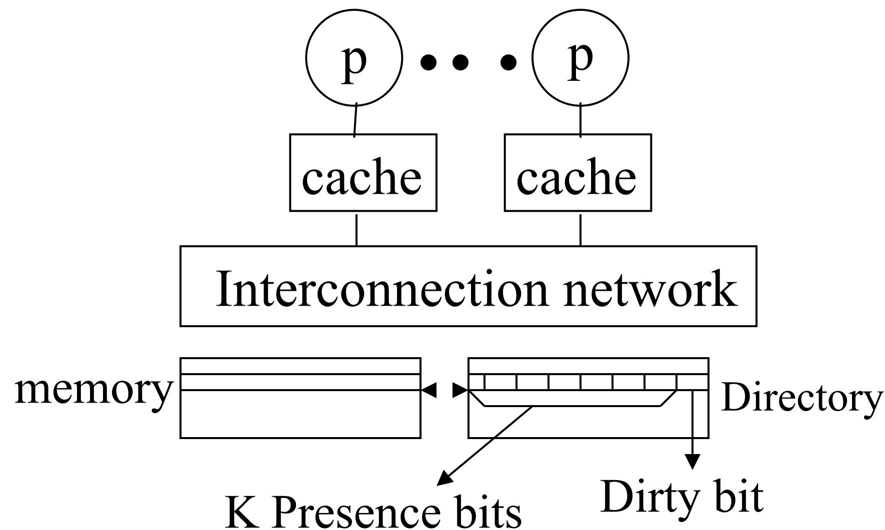
- Classification: Write-back, update, no-dirty-sharing.
- States :
  - VE (valid exclusive): only copy and clean
  - VS (valid shared) : shared -clean copy. Write hits result in updates to memory and other caches and entry remains in this state
  - D(dirty): dirty exclusive (only copy)
- Used special “shared line” on bus to detect sharing status of cache line
- Supports producer-consumer model well
- What about sequential processes migrating between CPU’s?


P-Read
[no one else has it]
Valid
Exclusive
P Read
Bus Read/Write
Bus write-miss
P-Write and not SL
P- Read and SL
Bus-read/write
Bus Write miss
Shared
P-Write M
and SL
P- Write and SL
P-read
P-write
Bus-write miss
Bus Read
[update MM]
Dirty
P-write
P-read
P-write Miss
and not SL

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)



- 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

Read Miss

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;}

Write Miss

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; turn bit PE[I] on; (dirty bit ON all the time) }

Write Hit Non - owned data

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; supply data to PE i ; turn dirty bit ON ; turn PE[i] ON }

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
- Can there be any races ? (e.g. both get 1)
- Consider the read - write was 2 instructions

2) Test-and-set : test a value & set it

   e.g. test for a zero and set to 1

3) Fetch-and increment : return the value & increment it

   0 means that the synch var is unclaimed

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 ll, sc → sc returns 0 (and fails)

- Can also be used for atomic fetch-and -increment

```
try :   ll      R2,0(R1)
        addi    R3,R2,#1
        sc      R3,0(R1)
        beqz    R3,try
```

Implementing Locks

- Given an atomic operation → use the coherence mechanisms of an MP to 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
- Since “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

test and test and set

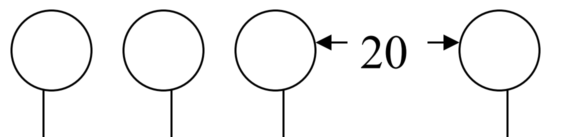
```
lockit :  ld      R2,0(R1)
          bnez    R2,lockit
          daddui  R2,R0,#1
          exch    R2,0(R1)
          bnez    R2,lockit
```

See Figure 5.24

- Can do same thing w/ll-sc

```
lockit :    ll      R2,0(R1)
            bnez    R2,lockit
            daddui  R2,R0,#1
            sc      R2,0(R1)
            beqz    R2,lockit ; 0 means SC failed
```

Problem of spin locks : not scalable → lots of traffic when the lock is released if many processes waiting



Each proc tries to

lock a var

- One processor release lock → invalidates everybody — 1 bus transaction
- All processors read miss — 19 bus trans
- All processors do an exchange — many bus trans

exchanges invalidate other processors