Shared Memory Architecture

Shared Memory Multiprocessor

Single global memory space 0, … , A – 1

Physically partitioned into M physical devices

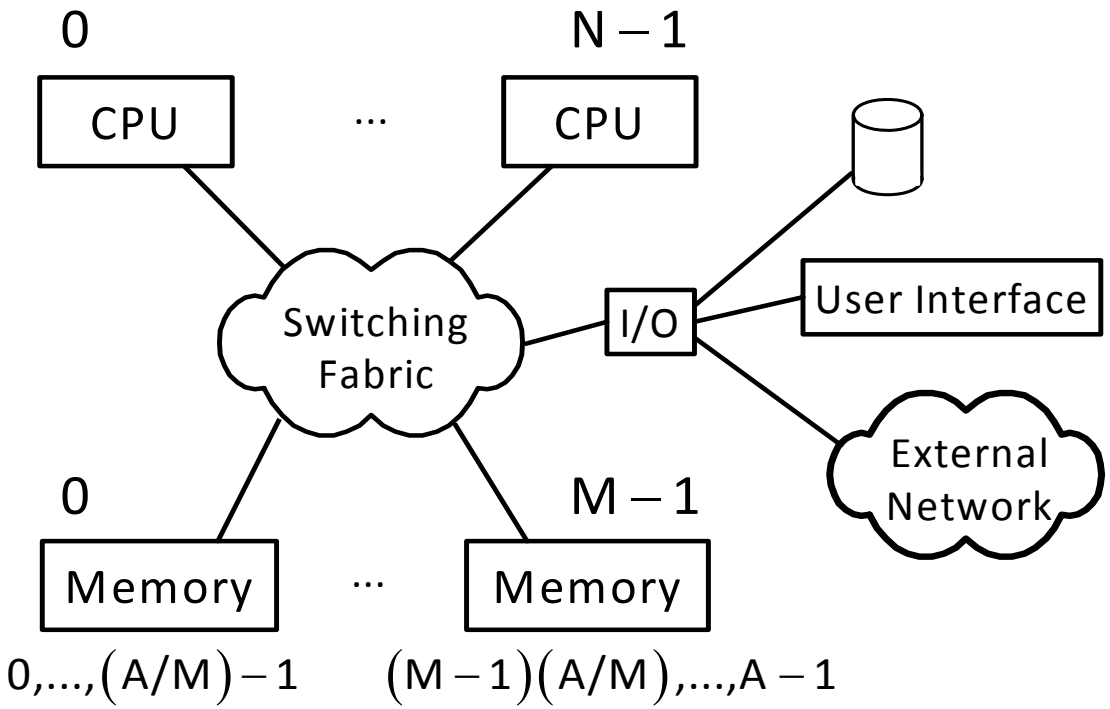
All CPUs access full memory space via interconnection network

CPUs communicate via shared memory addresses

Standard write / read transactions to shared addresses

Synchronize memory access

Prevent data hazards



Shared Memory System Subtypes

Uniform memory access (UMA)

- Shared memory accessible by all processors
- Access time to any memory location uniform for all processors
- Symmetric multiprocessor
  - All processors are of equivalent type
  - All processors have symmetric relationship to memory
  - Typically fast bus-based network (SCSI or PCI)

Non-uniform memory access (NUMA)

- Memory distributed among processors
  - Each processor physically attached to part of shared memory
  - Cost-effective scaling to large systems
- Access time to memory address depends on distance to processor
  - Most memory accesses are local and fast

Cache-only memory architecture (COMA)

- Each processor has part of shared memory as private cache
- Requires that data be migrated to processor requesting it

Contemporary Trend

Symmetric Multiprocessor (SMP)

N equivalent microprocessors

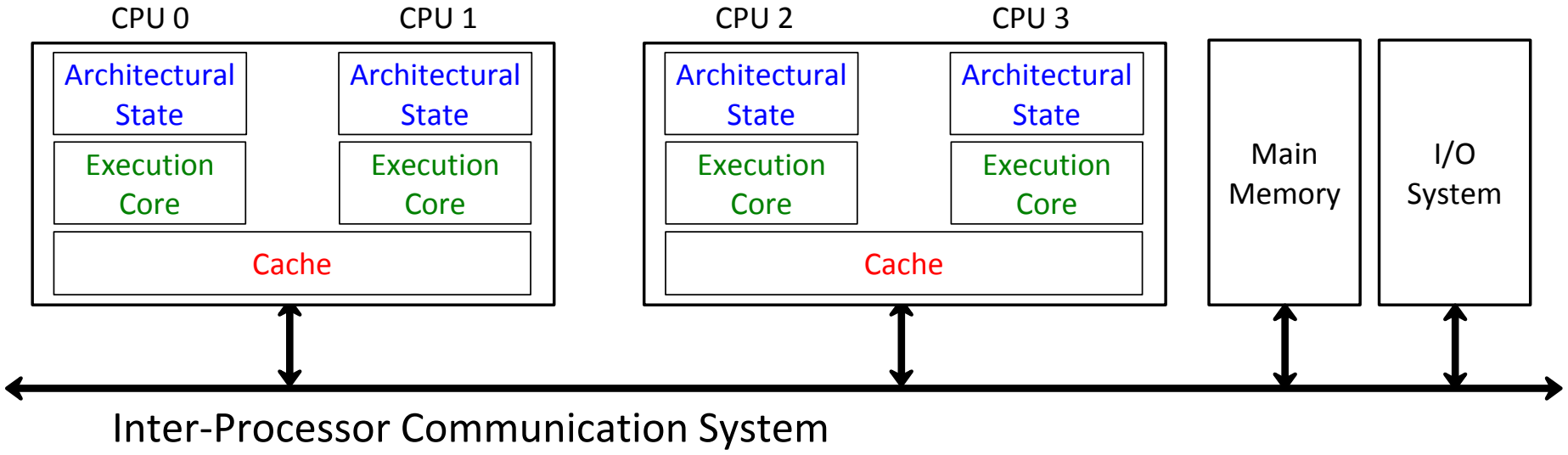
Multiple processor cores on single integrated circuit

Communication network between processors

Thread Level Parallelism (TLP)

Operating system runs in one processor

OS assigns threads to processors by some scheduling algorithm



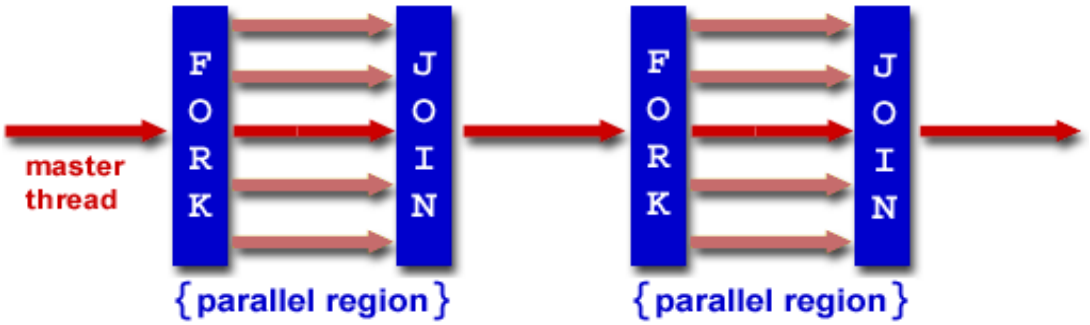
Open MP for Shared Memory Systems

Application Program Interface (API) for multiprocessing

- Supports shared memory applications in C/C++ and Fortran
- Directives for explicit thread-based parallelization
- Simple programming models on shared memory machines

Fork — Join Model

- Master thread (consumer thread)
  - Programs initiate as single thread
  - Executes sequentially until parallel construct is encountered
- Fork producer threads
  - Master thread creates team of parallel threads
  - Program statements in parallel construct execute in parallel
- Join producer threads
  - Producer threads complete
  - Synchronize and terminate
  - Master thread continues
- Nesting
  - Forks can be defined within parallel sections



Ref: https://computing.llnl.gov/tutorials/openMP/

General Code Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;
/* Serial code */
    ...
#pragma omp parallel private(var1, var2) shared(var3)
  {
 /* Parallel section executed by all threads */
    ...
 /* All threads join master thread and disband */
  }
/* Resume serial code */
    ...
}
```

Variables private to each thread
Each thread has private copy

Variables shared among all threads
One copy accessed by all threads

Selected Options for Parallel Pragma

if (scalar_expression)

Performs conditional multithreading

private (list)

Creates copy of variable types for each thread

shared (list)

Single shared copy of variables in list

Requires synchronization of memory accesses to avoid hazards

reduction (operator: list)

Performs join operation on list of private variables

num_threads (integer-expression)

Specifies number of threads to create

"Hello Worlds" Program

```
#include <omp.h>
main ()  {
int nthreads, tid;
/* Fork team of threads with private variables */
#pragma omp parallel private(tid)
  {
  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

Parallel For Directive

```
#pragma omp for
```

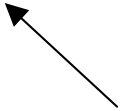
Options

- schedule (type [,chunk])
  - static
    - Loop iterations divided into chunk pieces size
    - Statically assigned to threads
    - Chunk is not specified — iterations are evenly divided
  - dynamic
    - Loop iterations dynamically scheduled
    - When thread finishes chunk it is dynamically assigned another
- ordered
  - iterations executed as in a serial program
- nowait
  - No synchronization at end of parallel loop
  - Threads do not wait for other threads to finish at end of private code

Parallel For Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
main ()
{
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

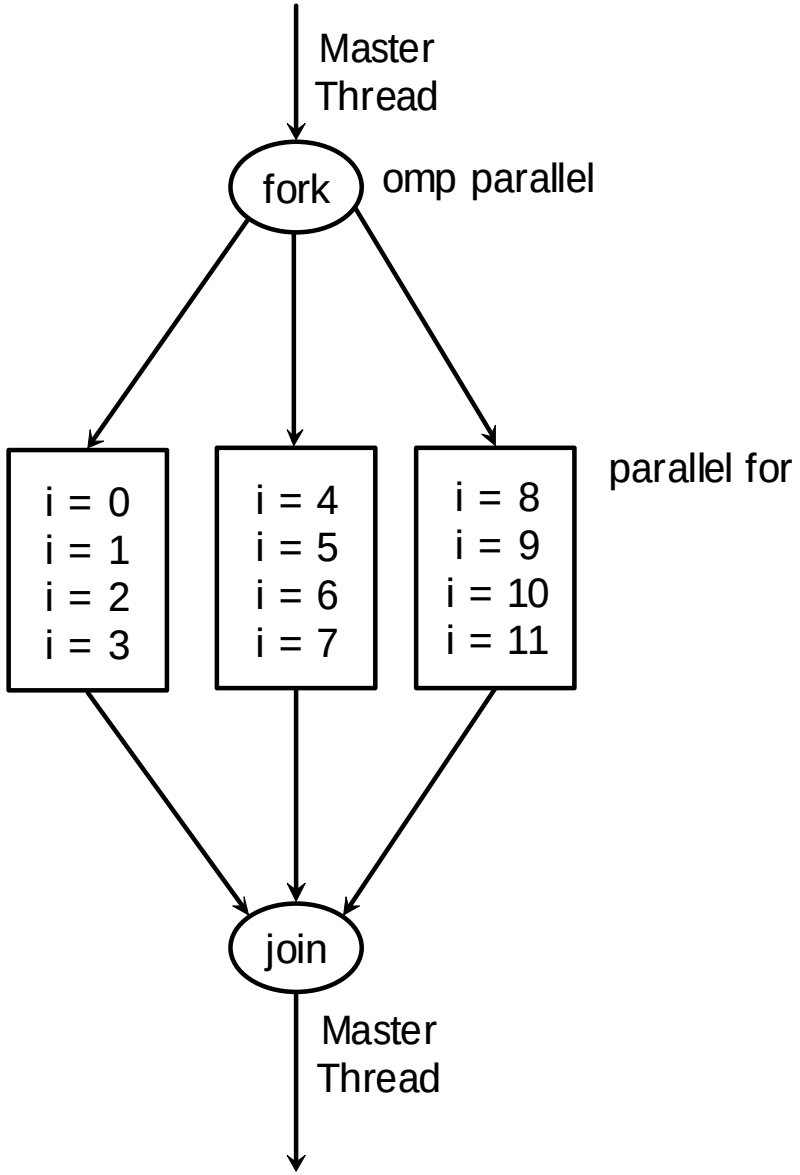
Data decomposition
- Arrays A, B, C, and variable N shared
- Variable i private
  - Each thread has unique copy
- Each loop iterates on chunk sized piece
- Threads do not synchronize (NOWAIT)



threads = N / chunk = 10

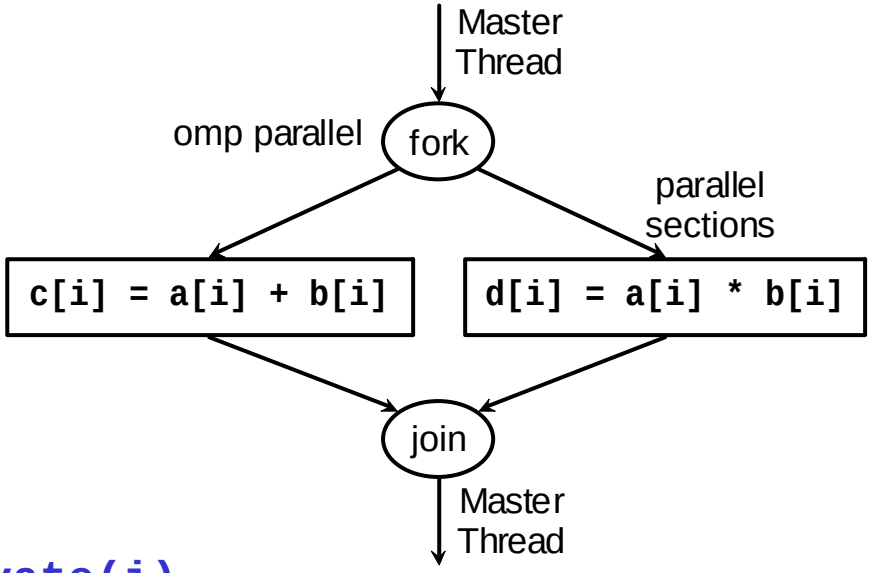
Running Parallel For

```
#pragma omp parallel
{
#pragma omp for
        for (i=0; i < 12; i++)
                c[i] = a[i] + b[i];
}
```



Sections Directive

```
#include <omp.h>
#define N     1000
main ()
{
int i;
float a[N], b[N], c[N], d[N];
for (i=0; i < N; i++){
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
    }  /* end of sections */
  }  /* end of parallel section */
}
```



Functional decomposition

Enclosed sections of code divided among threads in team

Race Conditions

Race condition

Data hazard caused by parallel access to shared memory

Example

```
#pragma omp parallel shared(x) num_threads(2)
    {
      x = x + 1;
    }
```

Two threads should increment x independently: **x ← x + 2**

Interleaved execution sequence for **x = 5**

```
Thread 1 – R1 ← x          ; CPU1 loads copy of x = 5
Thread 2 – R1 ← x          ; CPU2 loads copy of x = 5
Thread 1 – R1 ← R1 + 1     ; CPU1 updates R1 ← 6
Thread 2 – R1 ← R1 + 1     ; CPU2 updates R1 ← 6
Thread 1 – x ← R1          ; CPU1 writes x ← 6
Thread 2 – x ← R1          ; CPU2 writes x ← 6
```

Program completes with result **x ← 6** instead of **x ← 7**

Synchronization

Directives to control access to shared data among threads

`#pragma omp master`

Only master thread (thread 0) performs following **`{block}`**

`#pragma omp critical`

Only one thread can execute following **`{block}`** at a time

Other threads wait for thread to leave critical section before entering

`#pragma omp barrier`

Each thread reaching barrier waits until all threads reach barrier

`#pragma omp atomic`

Memory update in next statement must be completed atomically

Mini-critical section for memory write

Preventing Race Condition

Example

```
#pragma omp parallel shared(x) num_threads(2)
    {
      #pragma omp critical
      x = x + 1;
    }
```

Execution sequence with critical section for **x = 5**

```
Thread 1 – R1 ← x            ; CPU1 loads copy of x = 5
 ; Thread 2 blocks until thread 1 completes
Thread 1 – R1 ← R1 + 1       ; CPU1 updates R1 ← 6
Thread 1 – x ← R1            ; CPU1 writes x ← 6
 ; Thread 1 completes ⇒ thread 2 unblocked
Thread 2 – R1 ← x            ; CPU2 loads copy of x = 6
Thread 2 – R1 ← R1 + 1       ; CPU2 updates R1 ← 7
Thread 2 – x ← R1            ; CPU2 writes x ← 7
```

Program completes with result **x ← x + 2**

Performance implications — critical section runs sequentially

Reduction

Reduction (operator: list)

Performs join operation on list of private variables

Operation can be **`+ * - & ^ | && ||`**

Example

```
#pragma omp parallel for reduction(+:sum)
for (i=0; i < N; i++)
{
     sum+ = a[i] * b[i];
}
```

Each thread has private copy of variable sum

On join (end of parallel construct)

- Private copies of sum combined by addition (+)
- Result copied into master thread copy of sum

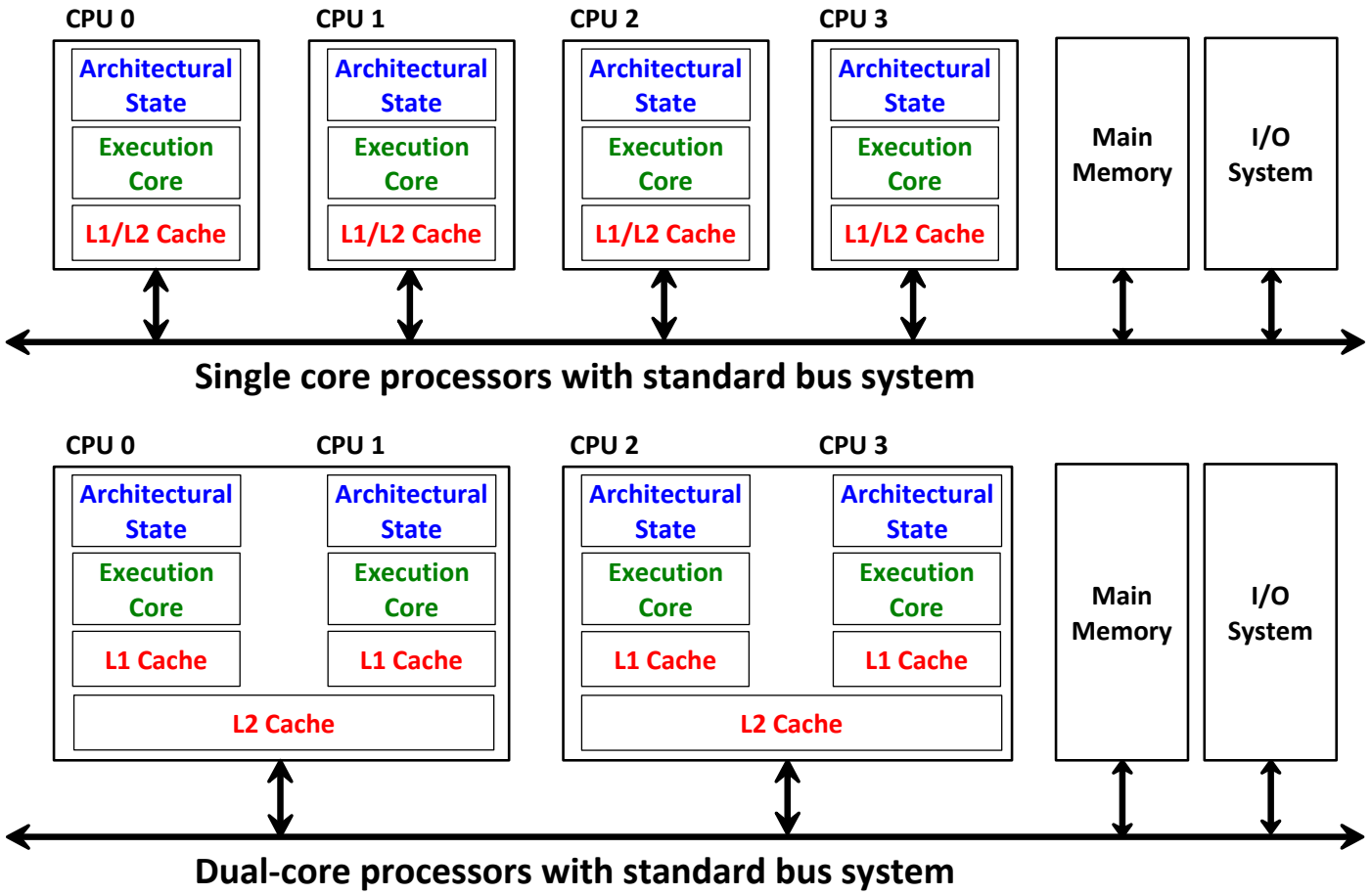
Bus-Based Symmetric Multiprocessor (SMP)

Bootstrap Processor (BSP)

Chosen by BIOS to perform system startup and run OS

Application Processors (AP)

In halt state unless allocated application threads by OS



Cache Organization for Dual Core Processors

Pentium D

Dual core processor

Each core has private L1 D+I cache and L2 D+I cache

Core Duo, Core 2, i3, i5, i7, ...

Dual core processor

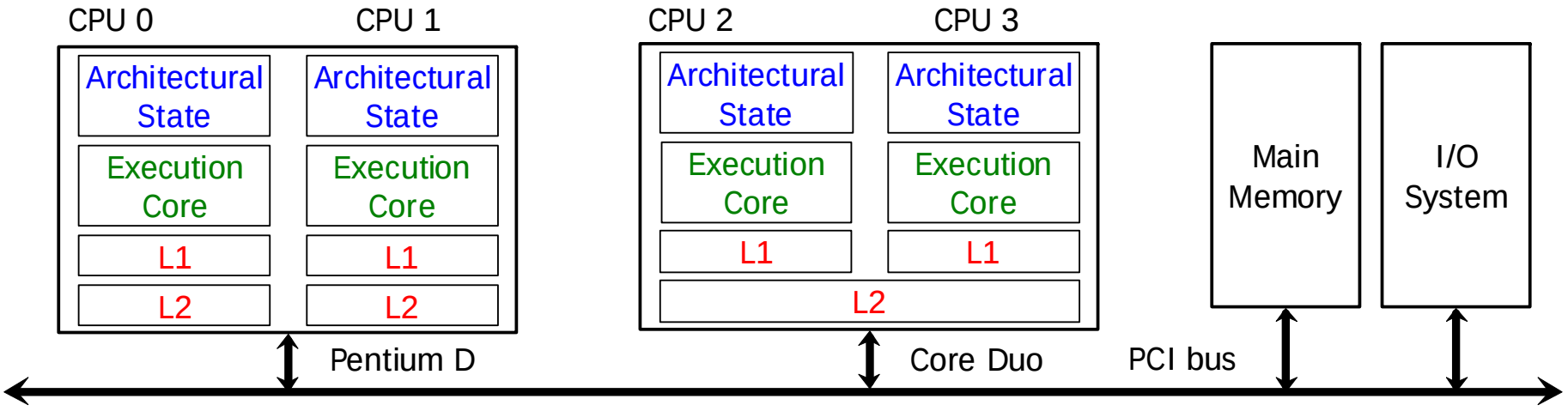
Each core has private L1 data cache

Cores share L2 cache

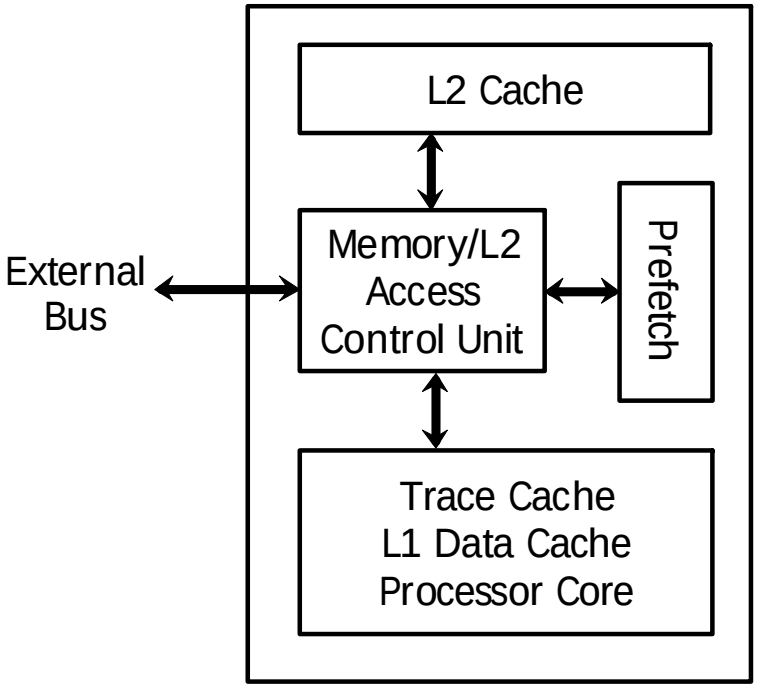
No L1 instruction cache

Instructions fetched directly from L2

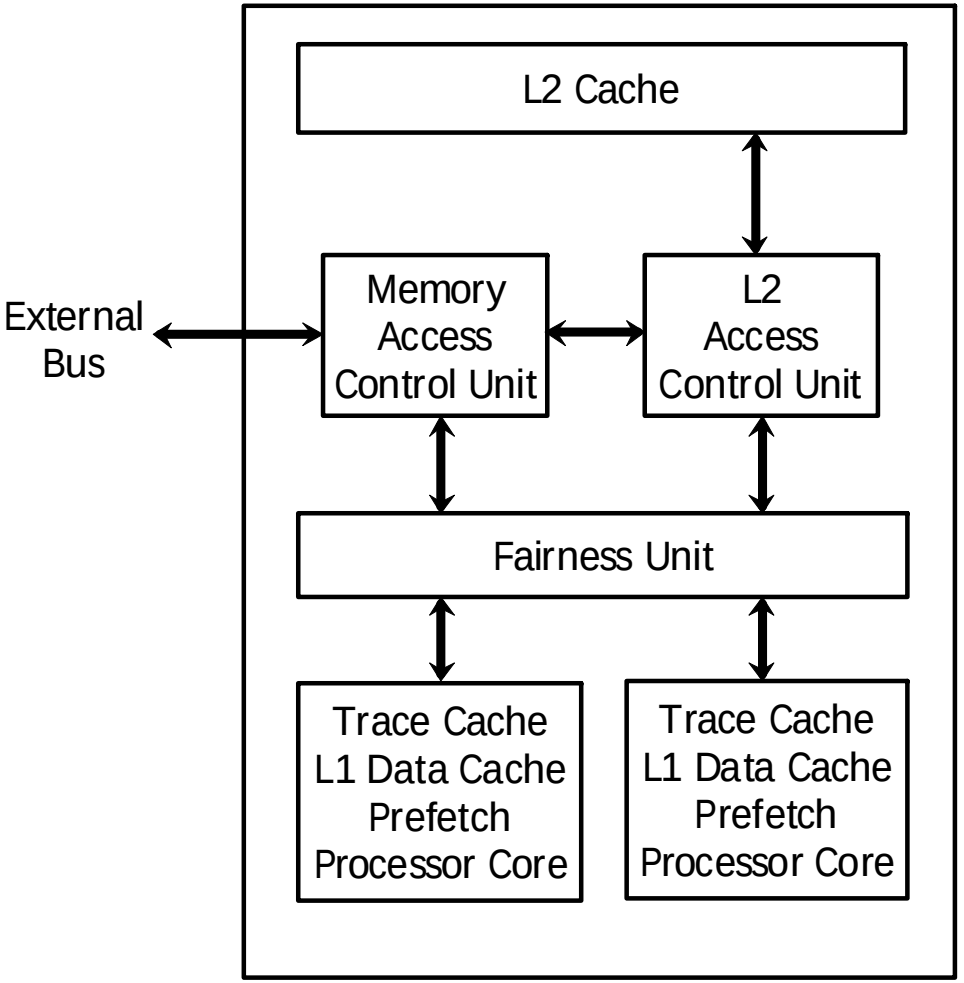
Employ trace caching instead



Memory/L2 Access Control Unit



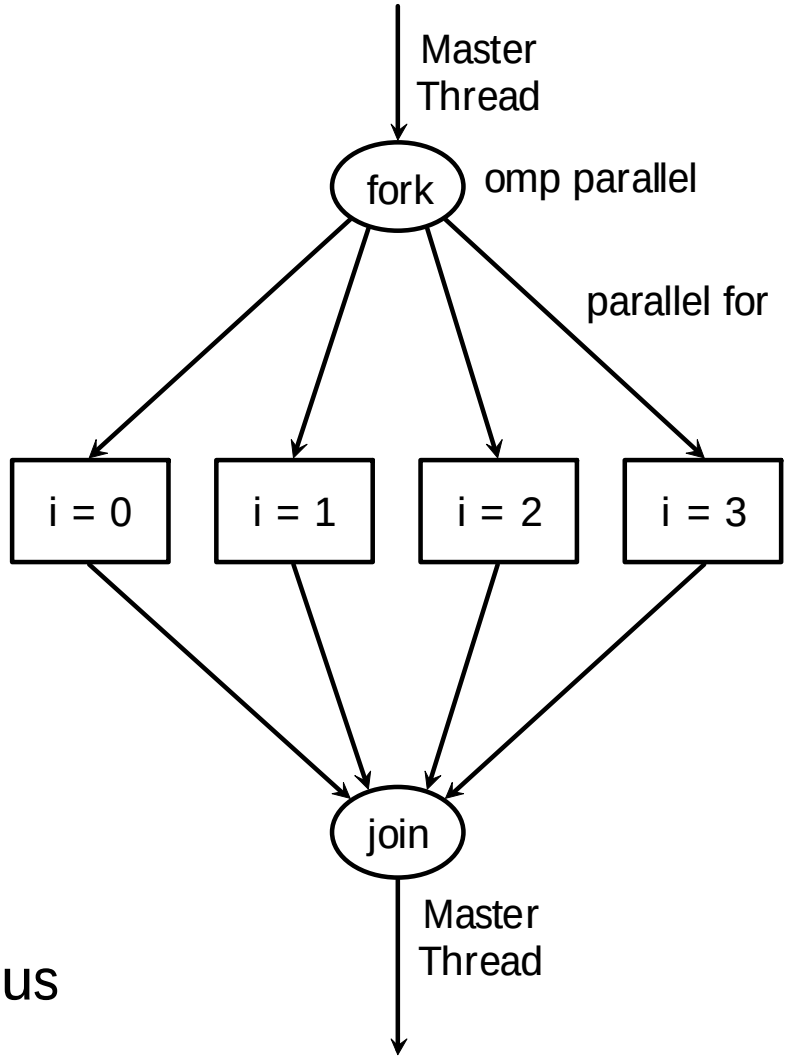
Pentium-M



Core Duo

Communication Overhead in SMP

```
#pragma omp parallel
{
#pragma omp for
        for (i=0; i<4; i++)
                c[i] = a[i] + b[i];
}
```



Sequential program requires

4 × load-add-store

Parallel program requires

4 × update instruction cache over serial bus

1 × load-add-store

In most architectures cache updates more expensive than arithmetic

Shared Memory Program — Vector Product

Compute $\sum_{i=0}^{3} \mathbf{a[i]} * \mathbf{b[i]}$ from data in shared memory

Sequential Code

```
for (i=0; i<4; i++){
  load Ra, a[i]
  load Rb, b[i]
  Ra ← Ra * Rb
  add Racc, Ra
}
store p, Racc
```

CC ~ overhead + 4 × 4 + 1

Neglecting overhead

$$S \sim \frac{17}{12} = 1.42$$

Parallel Code

```
fork 4 threads with private i=0,1,2,3
  load Ra, a[i]          …   load Ra, a[i]
  load Rb, b[i]              load Rb, b[i]
  Ra ← Ra * Rb               Ra ← Ra * Rb
  store p[i], Ra             store p[i], Ra
fork 2 threads with private i=0,2
  load Ra, p[i]          …   load Ra, p[i]
  load Rb, p[i+1]            load Rb, p[i+1]
  Ra ← Ra + Rb               Ra ← Ra + Rb
  store p[i], Ra             store p[i], Ra
load Ra, p[0]
load Rb, p[2]
Ra ← Ra + Rb
store p, Ra
```

CC ~ overhead + 1 × 4 + 1 × 4 + 4

Shared Memory Multiprocessor Capacity

Capacity limitation

CPUs operate independently on data cache

Access shared memory to exchange data when required

Data exchange required $\Rightarrow$ cache miss on at least one CPU

Capacity definition

Interconnection network capacity $\geq$ volume of exchanged data

Exchange demand volume $= N \times D_x \times M$

$N =$ CPUs

$D_x =$ average data access rate (bytes per second)

$M =$ cache miss rate $=$ inter-CPU access rate

Interconnection capacity $= R_n \times W$

$R_n =$ transfer rate (transfer cycles per second)

$W =$ transfer width (bytes per transfer)

$$N \leq \frac{R_n W}{D_x M}$$

Capacity Example

Standard PCI express bus — 16 bytes per cycle at 1 GHz

Average data access rate depends on

- Integer width
- Loads per instruction
- Instructions per second = 1/[(seconds per CC) × (CC per instruction)]

Miss rate depends on number of data reads between cache updates

- Compute dominated M ~ 0.01
- Communication dominated M ~ 0.1

$$N \le \frac{R_n W}{D_x M}$$

$$N \le \frac{10^9 \text{ transfers/sec} \times 16 \text{ bytes/transfer}}{10^9 \text{ bytes/sec} \times 0.1} = 160$$

$R_n = 1$ GHz

$W = 16$ bytes/transfer

$$D_x = (4 \text{ bytes/load}) \times (0.25 \text{ data loads/instruction}) \times (10^9 \text{ instruction/sec})$$

$$= 10^9 \text{ bytes/sec}$$

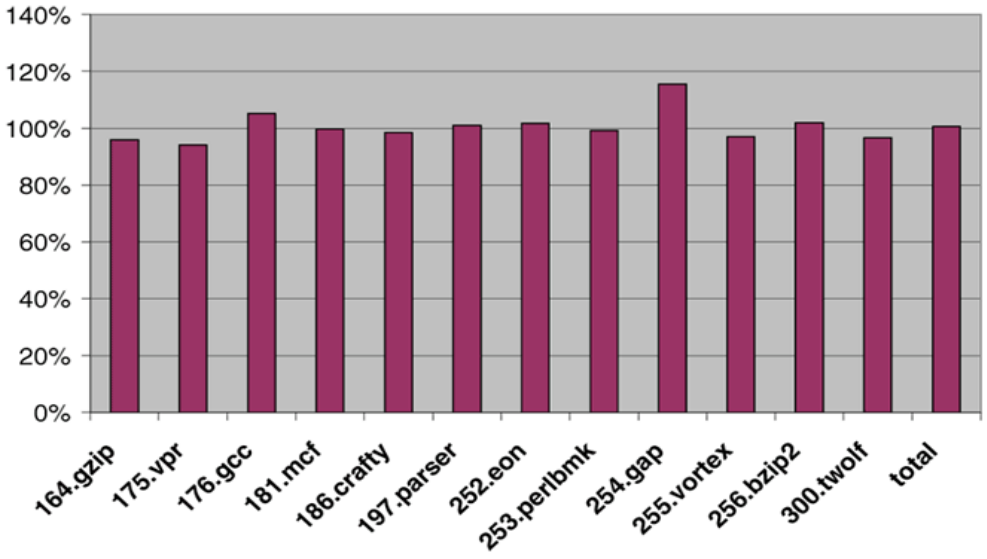
$M = 0.1$ (communication dominated miss rate)

Single-Threaded Performance

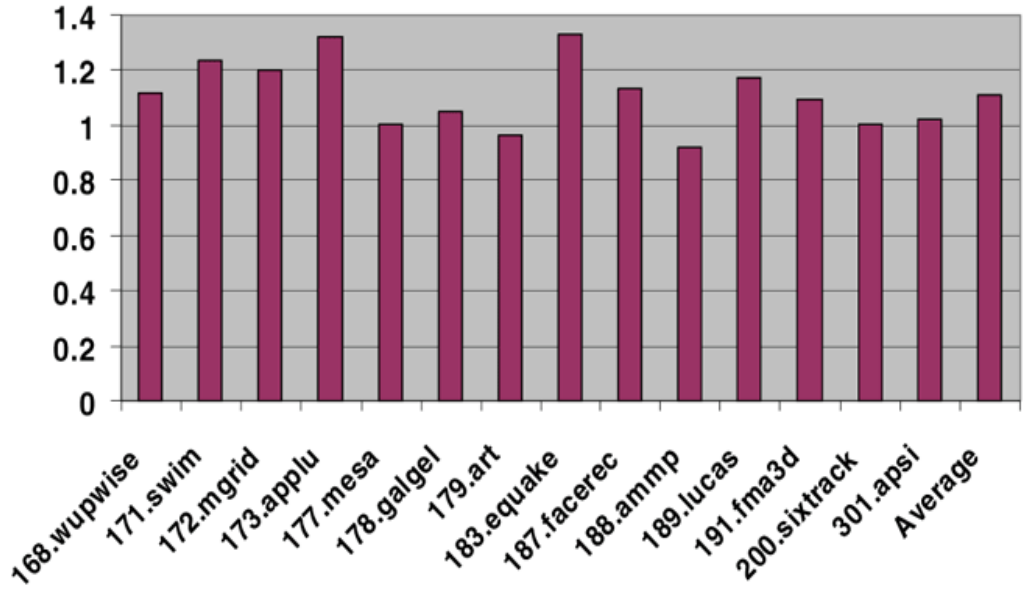
Core Duo vs. Pentium M

Single-threaded SPEC code performance

Cint



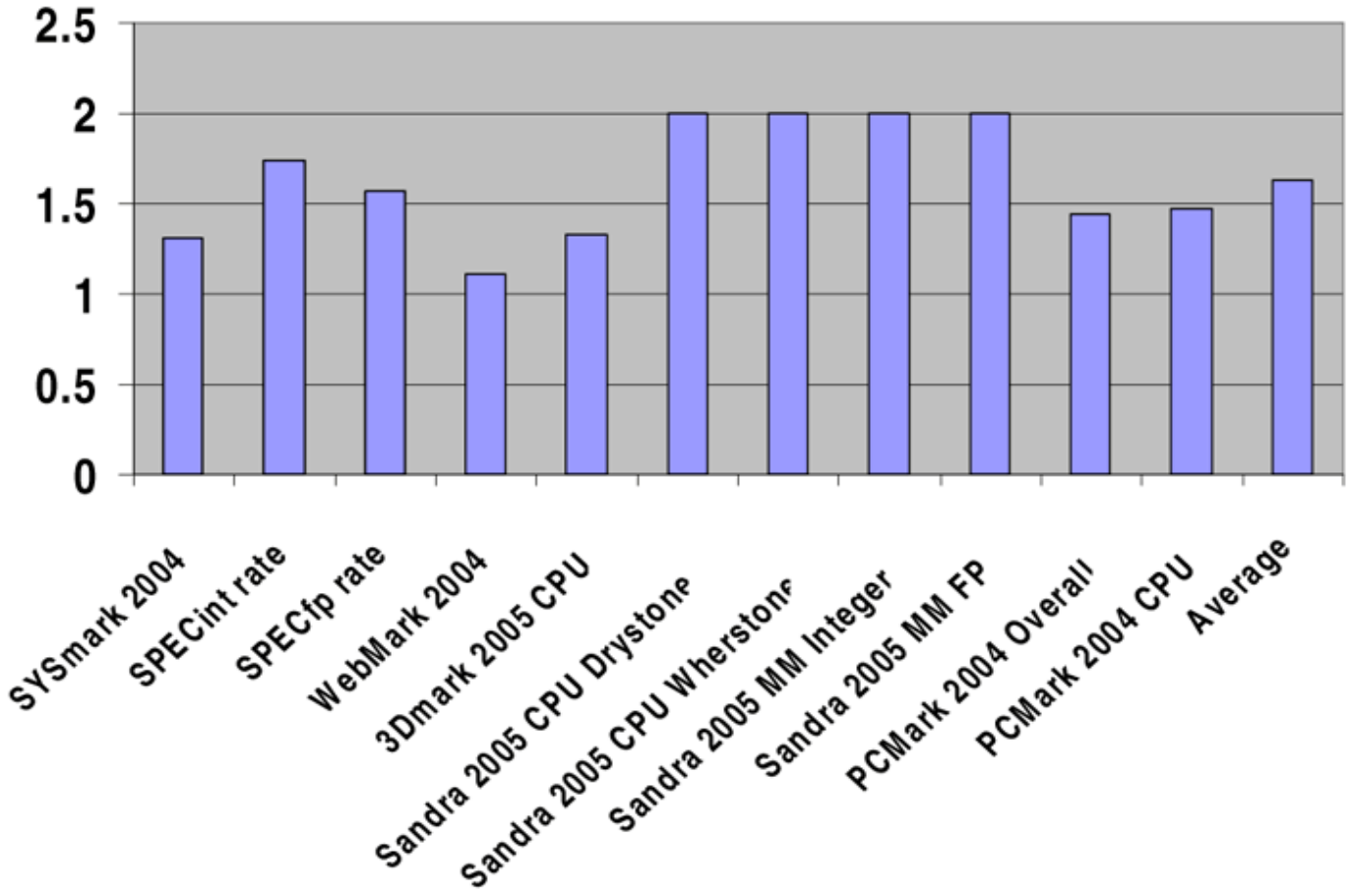
Cfp



Multi-Threaded Performance

Core Duo vs. Pentium M

Multithreaded code performance



Effect of Threading Model

Data domain decomposition

Divide large data arrays among processors

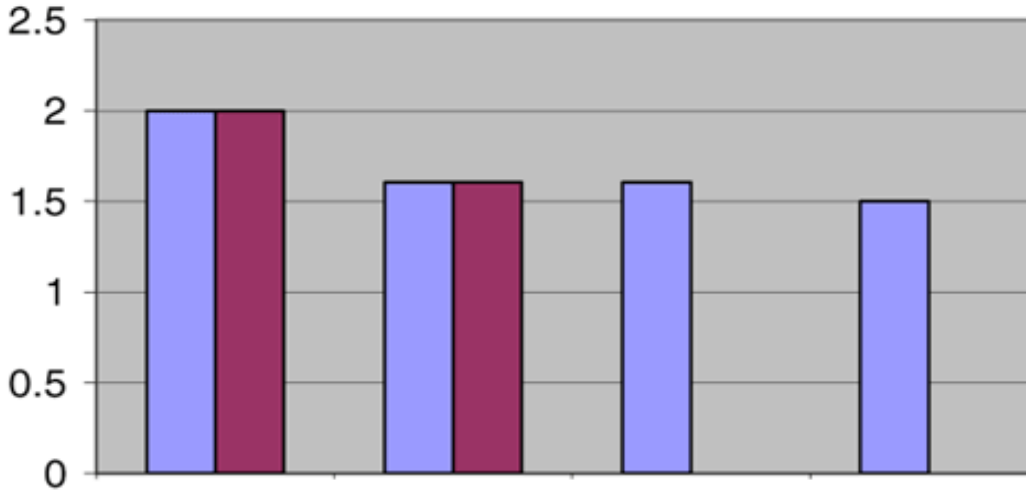
Works well with shared memory SMP

Functional domain decomposition

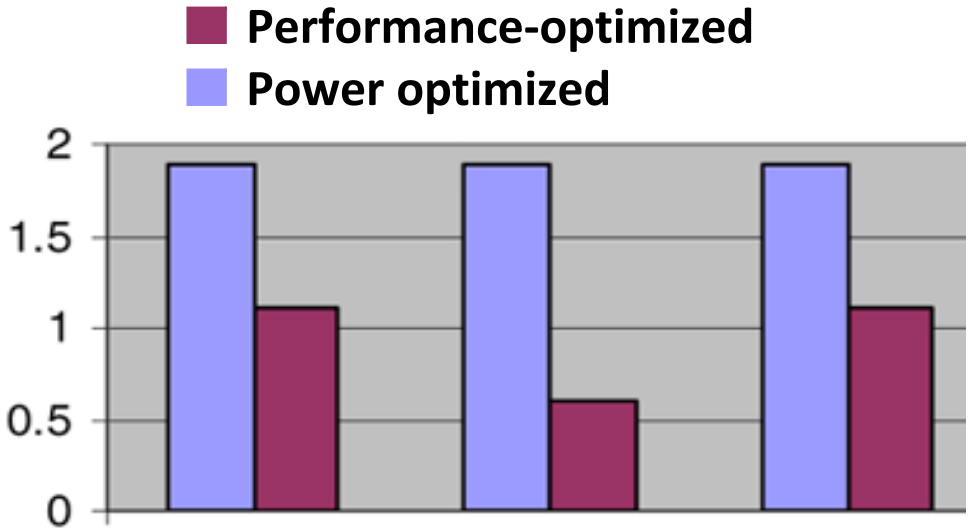
Divide tasks among processors

Tends toward load balancing problems on SMP

Works better with work queue systems on message passing systems



Data decomposition on Core Duo



Functional decomposition on Core Duo

Data Hazards in Shared Memory SMP

Three levels of data hazard in shared memory systems

- Program level
  - Concurrent programming of inherently sequential operations
  - Handled with programmed synchronization directives
    - Atomic read/write, critical sections, barrier
- Cache consistency
  - Multiple processors writing shared copies of data
  - Protocols for maintaining valid copies of data values
- Hardware level consistency
  - Instruction-level memory semantics are abstractions
  - Real hardware operates in more complex manner

General approach to handle hazards

- Enforce operational definition for consistency
- Enables unambiguous program validation

Ref: Adve, Gharachorloo, "Shared Memory Consistency Models: A Tutorial", http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

Sequential Consistency

Strict consistency

- Memory operations performed in order intended by programmer
- Possible to implement only on single processor systems

Sequential consistency (Lamport 1979)

- Clear, consistent, repeatable definition of execution order
- Result of any execution identical to result of execution in which:
  - Operations on all processors executed in specified sequential order
  - Operations on each processor appear in order specified by program
- Specified sequential order
  - Any well-defined interleaving
  - Example — round robin

Implications for programmer

- Unsynchronized threads assumed to execute interleaved
- Cache assumed to enforce write-order consistency
- Hardware assumed to enforce read/write-order consistency

Implementing Critical Section with Semaphore

Semaphore

Unsigned number with 2 operations

$$\mathbf{P(s)}: \begin{cases} \mathbf{s} \leftarrow \mathbf{s}-\mathbf{1}, & \mathbf{s} > \mathbf{0} \\ \text{wait}, & \mathbf{s} = \mathbf{0} \end{cases}$$

$$\mathbf{V(s)}: \quad \mathbf{s} \leftarrow \mathbf{s}+\mathbf{1}$$

Binary semaphore

Mutual exclusion (mutex) or lock

s initialized to **1**

Critical section

```
P(s)      /* section begins (s=1) or blocks (s=0) */
    x = x + 1;
V(s)      /* s ← 1 permits another thread to operate */
```

Difficulty

Requires system-wide atomic semaphore operation

Impractical to disable all system interrupts and interleaving during P and V

Shared Variable Lock

Shared variables `Flag1` and `Flag2` initialized to zero

Thread 1	Thread 2

`Flag1 = 1;`
`loop: if (Flag2 == 1) loop`
`<critical section>`
`Flag1 = 0;`
	`Flag2 = 1;`
	`loop: if (Flag1 == 1) loop`
	`<critical section>`
	`Flag2 = 0;`

Spin-loop

Loop instruction repeats until condition clears

Interleaved execution

Actual execution order `t1, t2, t3, t4`

Creates deadlock

Thread 1	Thread 2

`Flag1 = 1;` `t1`
`loop: if (Flag2 == 1) loop` `t3`
`<critical section>`
`Flag1 = 0;`
	`Flag2 = 1;` `t2`
	`loop: if (Flag1 == 1) loop` `t4`
	`<critical section>`
	`Flag2 = 0;`

Modified Shared Variable Lock

Shared variables `Flag1` and `Flag2` initialized to zero

Thread 1	Thread 2

`loop: Flag1 = 1;`
`if (Flag2 == 1)`
`{Flag1 = 0 ; loop}`
`<critical section>`
`Flag1 = 0;`
	`loop: Flag2 = 1;`
	`if (Flag1 == 1)`
	`{Flag2 = 0 ; loop}`
	`<critical section>`
	`Flag2 = 0;`

Interleaved execution

Order `t1, t2, t3, t4`

No deadlock but possible livelock if hardware writes not atomic

Thread 1	Thread 2

`loop: Flag1 = 1;` `t1`
`if (Flag2 == 1)`
`{Flag1 = 0 ; loop}` `t3`
`<critical section>`
`Flag1 = 0;`
	`loop: Flag2 = 1;` `t2`
	`if (Flag1 == 1)`
	`{Flag2 = 0 ; loop}` `t4`
	`<critical section>`
	`Flag2 = 0;`

Dekker Algorithm

Shared variables `Flag1, Flag2 = 0 ; turn = 1`

Thread 1	Thread 2

`Flag1 = 1;`
`turn = 1;`
`loop: if (Flag2 == 1`
`&& turn == 1) loop`
`<critical section>`
`Flag1 = 0;`
	`Flag2 = 1;`
	`turn = 2;`
	`loop: if (Flag1 == 1)`
	`&& turn == 2) loop`
	`<critical section>`
	`Flag2 = 0;`

Thread 1	Thread 2

`Flag1 = 1; t1`
`turn = 1; t3`
`loop: if (Flag2 == 1`
`&& turn == 1) loop t5`
`<critical section>`
`Flag1 = 0;`
	`Flag2 = 1; t2`
	`turn = 2; t4`
	`loop: if (Flag1 == 1)`
	`&& turn == 2) loop t6`
	`<critical section>`
	`Flag2 = 0;`

No deadlock or livelock

Generalization to n > 2 threads — Lamport bakery algorithm

Machine Language Support

Atomic instruction primitives in processor ISA

Provide hardware-level semaphore **M** for well-defined atomic memory access

Enable implementation of atomic constructs at compiler level

Instruction primitive	Application
Test_and_Set M, R **Regs[R] ← Mem[M]** **if Regs[R] == 0 Mem[M] ← 1**	**L1: Test_and_Set M, R1 ; spinlock** **BNEZ R1, L1** **<critical section>** **Swap M, R1**
Fetch_and_Add M, R1, R2 **Regs[R1] ← Mem[M]** **Mem[M] ← Regs[R1] + Regs[R2]**	**ADDI R2, R0, #1** **L1: Fetch_and_Add M, R1, R2** **BNEZ R1, L1** **<critical section>** **Swap M, R1**
Swap M, R: **Regs[R_{temp}] ← Mem[M]** **Mem[M] ← Regs[R]** **Regs[R] ← Regs[R_{temp}]**	

Compare and Swap (CAS)

Swaps M and R2 if M = R1

```
Compare_and_Swap M, R1, R2
    if (Regs[R1] == Mem[M]){
        Mem[M] = Regs[R2]
        Regs[R2] = Regs[R1]
        Cflag ← 1
     }
    else Cflag ← 0
```

No lock

Non-blocking atomic operation more efficient (M. Herlihy — 1991)

Critical Section	`#pragma omp critical` `  x = x + 1;`

Assembly Code	`L1:  LW R1, x  ; load x`
	`  ADDI R2, R1, #1  ; prepare new value for x`
	`  CAS x, R1, R2  ; if no change in stored x, update x`
	`  BEQZ Cflag, L1  ; else start again`

Load Reserve and Store Conditional

Load-reserve

Returns current value of memory location

Associates reservation flag with address

Flag can be reset by subsequent load-reserve

Store-conditional

Performs write if reservation flag still set

Stronger than compare and swap

Prevents store in location that was written after read

Even if original value was restored

```
load-reserve R, M
    <flag, adr> ← <1, M>;
    Regs[R] ← Mem[M];
```

```
store-conditional M, R
    if <flag, adr> == <1, M>
      clear <flag, adr>;
      Mem[M] ← Regs[R];
      status ← 1;
    else  status ← 0;
```

```
L1: load-reserve R1, x
    ADDI R1, R1, #1
    store-conditional x, R1
    BEQZ status, L1
```

Hardware Level Consistency without Cache

Instruction-level memory semantics

ISA defines unambiguous sequential actions

`LW R1, 32(R2)` ⇒ `Regs[R1]` ← `Mem[Regs[R2] + 32]`

`SW 32(R2), R1` ⇒ `Mem[Regs[R2] + 32]` ← `Regs[R1]`

Hardware-level implementation

Hardware optimizations alter presumed order of instruction actions

Instruction behavior meets definition on single processor machines

Can create hazards in multiprocessors

Typical hazards

Single processor may reorder data reads and writes

Multiple processors may reorder reads and writes among themselves

Multiple processors write to multiple memory devices simultaneously

Example — Write Buffer with Bypass

Write buffer

Processor writes updates to memory through FIFO buffer

Prevents memory write latency from stalling pipeline

Memory reordering

Write buffer permits reads to bypass previous writes

Prevents memory write queue from stalling subsequent reads

Forwarding

Read hit to address in write buffer updates directly from FIFO

No data hazards for single processor system

Violates sequential coherency in multiprocessor system

Example — Dekker algorithm with bypassing write buffer

Load **`other_flag`** bypasses Store **`my_flag`** in both processors

Load **`other_flag`** always returns **`0`**

Example — Write Overlapping

Multiprocessor system with general interconnect network

- Crossbar or multiple bus network
- Memory partitioned into multiple physical devices
- Permits multiple memory writes per transfer cycle

Violates sequential coherency in multiprocessor system

- Example

```
#pragma omp critical
  x = x + 1;
```

- Critical section implemented with Dekker algorithm (not CAS)
- Thread 1
  - Issues write instructions to **`x`** and **`Flag1`** releasing lock
  - Write to **`x`** delayed in network
  - Write to **`Flag1`** completes
- Thread 2 completes
- Thread 1 write to **`x`** completes after Thread 2 write to **`x`**

Example — Read/Write Reordering

Multiprocessor system with general interconnect network

Permits multiple memory writes per transfer cycle

Violates sequential coherency in multiprocessor system

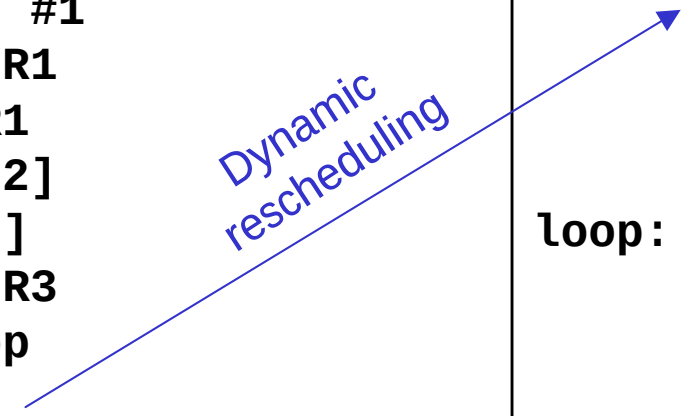
Example
```
#pragma omp critical
    x = x + 1;
```

Critical section implemented with Dekker algorithm (not CAS)

Thread 1 runs out-of-order but Thread 2 runs in-order

Thread 1 Listing	Thread 1 Out-of-Order Execution

`ADDI R1, R0, #1`
`SW [Flag1], R1`
`SW [turn], R1`
`loop: LW R2, [Flag2]`
`LW R3, [turn]`
`AND R4, R2, R3`
`BNEZ R4, loop`
`LW R5, [x]`
`ADDI R5, R5, #1`
`SW [x], R5`
`SW [Flag1], R0`
	`LW R5, [x]`
	`ADDI R1, R0, #1`
	`SW [Flag1], R1`
	`SW [turn], R1`
	`loop: LW R2, [Flag2]`
	`LW R3, [turn]`
	`AND R4, R2, R3`
	`BNEZ R4, loop`
	`ADDI R5, R5, #1`
	`SW [x], R5`
	`SW [Flag1], R0`



Memory Fences

Memory barrier (membar)

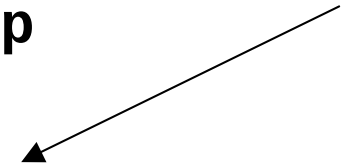
Machine-level instruction inserted by programmer or compiler

Enforces sequential consistency in rescheduling

Instructions are not moved past a membar

Example

```
        ...
loop:   BNEZ R4, loop
        MEMBAR
        LW R5, [x]
        ADDI R5, R5, #1
        ...
```



Processor will not execute load before memory barrier

Cache Policies in Single CPU System

Read miss

- Update cache (copy block from memory)

Write miss

- Write allocate policy
  - Update cache
- No write allocate
  - No cache update
  - Write data directly to memory

Write to cache

- Write through
  - Update memory (copy block from cache) on every cache write
  - Maintains cache-memory consistency
- Write back
  - Update memory when block swapped out
  - Requires fewer memory bus operations

Cache Consistency and Coherency

Multiple processors access shared data in main memory

- Each processor copies data blocks to private cache
- Caches and main memory become inconsistent

Example

- CPU-0 reads **X** to cache from main memory
- CPU-1 reads **X** to cache from main memory
- CPU-0 writes to **X** in cache
  - Copies of **X** in CPU-1 cache and main memory not valid

Strict consistency

- All copies of any data location are identical
- Almost impossible to implement

Sequential consistency

- Operations on all processors executed in specified sequential order
- Operations on each processor appear in order specified by program

Cache coherency

- Implementation of sequential consistency for cache and main memory
- Requires processor hardware support for cache coherency management

Cache Inconsistency in Write Back Policy

Concurrent threads

```
#pragma omp critical
 x = x + 1;
```

Each thread enforces sequential consistency with its private cache

Criticality not enforced on all copies of **x**

Example

CPU-0	Cache 0	Memory	Cache 1	CPU-1
LW R5, [x]	x = 2	x = 2 y = 10		
ADDI R5, R5, #1	x = 2	x = 2 y = 10	x = 2	LW R5, [x]
SW [x], R5	x = 3	x = 2 y = 10	x = 2	ADDI R5, R5, #1
LW R6, [y] ; write back x	y = 10	x = 3 y = 10	x = 3	SW [x], R5
		x = 3 y = 10	y = 10	LW R6, [y] ; write back x

Cache Inconsistency in Write Through Policy

Concurrent threads

```
#pragma omp critical
 x = x + 1;
```

Each thread enforces sequential consistency with its private cache

Criticality not enforced on all copies of **x**

Example

CPU-0	Cache 0	Memory	Cache 1	CPU-1
LW R5, [x]	x = 2	x = 2		
ADDI R5, R5, #1	x = 2	x = 2	x = 2	LW R5, [x]
SW [x], R5	x = 3	x = 3	x = 2	ADDI R5, R5, #1
		x = 3	x = 3	SW [x], R5

Cache Coherence Protocols

Cache coherence

Enforce sequential consistency for all caches and main memory

Enables system-wide atomic and critical constructs

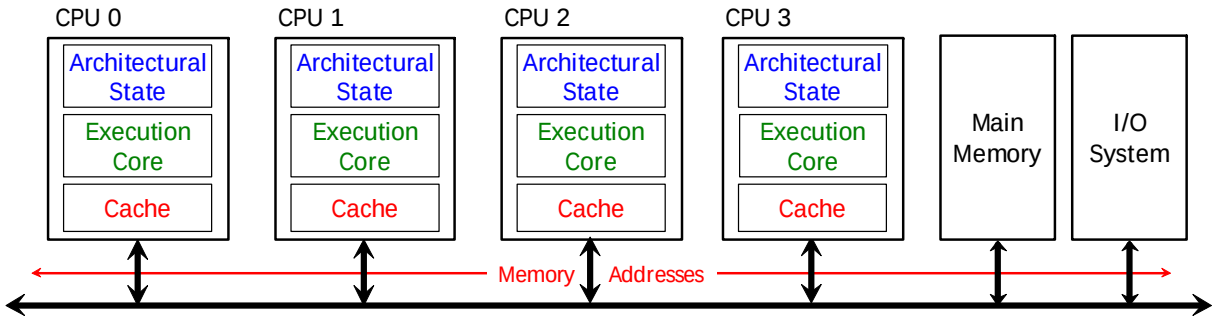
Snoopy cache

Cache blocks tags with status bits depending on coherence policy

- Status depends on access history to data in block
- Change in status can initiate write back before usual block eviction

Cache manager

- Monitors all addresses written on system bus by all processors
- Compares addresses with blocks in cache
- Updates state of cached block on address hit



Cache Coherence Policy Types

Write update

- Processor W places write address on bus
- Snooping processors S_i mark copy of block invalid on address hit
- W writes to cache and to bus
- S_i update local copies of block and mark updated blocks valid
- Creates large bus overhead

Write invalidate with inquire

- Processor W places write address on bus
- Snooping processors S_i mark copy of block invalid on address hit
- W writes to local cache — creates unique valid copy of data block
- Inquire — S_i place address on bus to access copy of block in W
- Lower bus overhead

Typical implementation in shared memory system

- Write allocate — W loads to cache on write miss
- Write invalidate — S_i invalidate cache block on write by W

State Definitions

Possible states of data block

Indicated by status bits in block tag

Modified

Unique valid copy of block

Block has been modified since loading

Owned

Device is cache is owner of block

Device services requests by other processors for block

Exclusive

Unique valid copy of block

Block has not been modified since loading

Shared

Block held by multiple caches in system

Invalid

Block must be reloaded before next access

Processor Behavior — Modified

Processor W cache holds unique valid copy of data block

- Block is dirty
  - Memory copy is different from W copy
- Cache copies in other processors S_i marked Invalid
  - S_i inquire (request update from this copy) before accessing block

W can continue to update this copy

- No memory update on cache writes
- Memory update on cache swap or inquire

W responds to inquire

- Snoops memory address from S_i on bus
- W updates memory
- Marks block invalid on write inquire

Processor Behavior — Shared

Processor W cache holds one of many valid copies of data line

- Block is clean
  - Memory copy is same as W copy
- Other processors S_i may have copies of block

W can update this copy

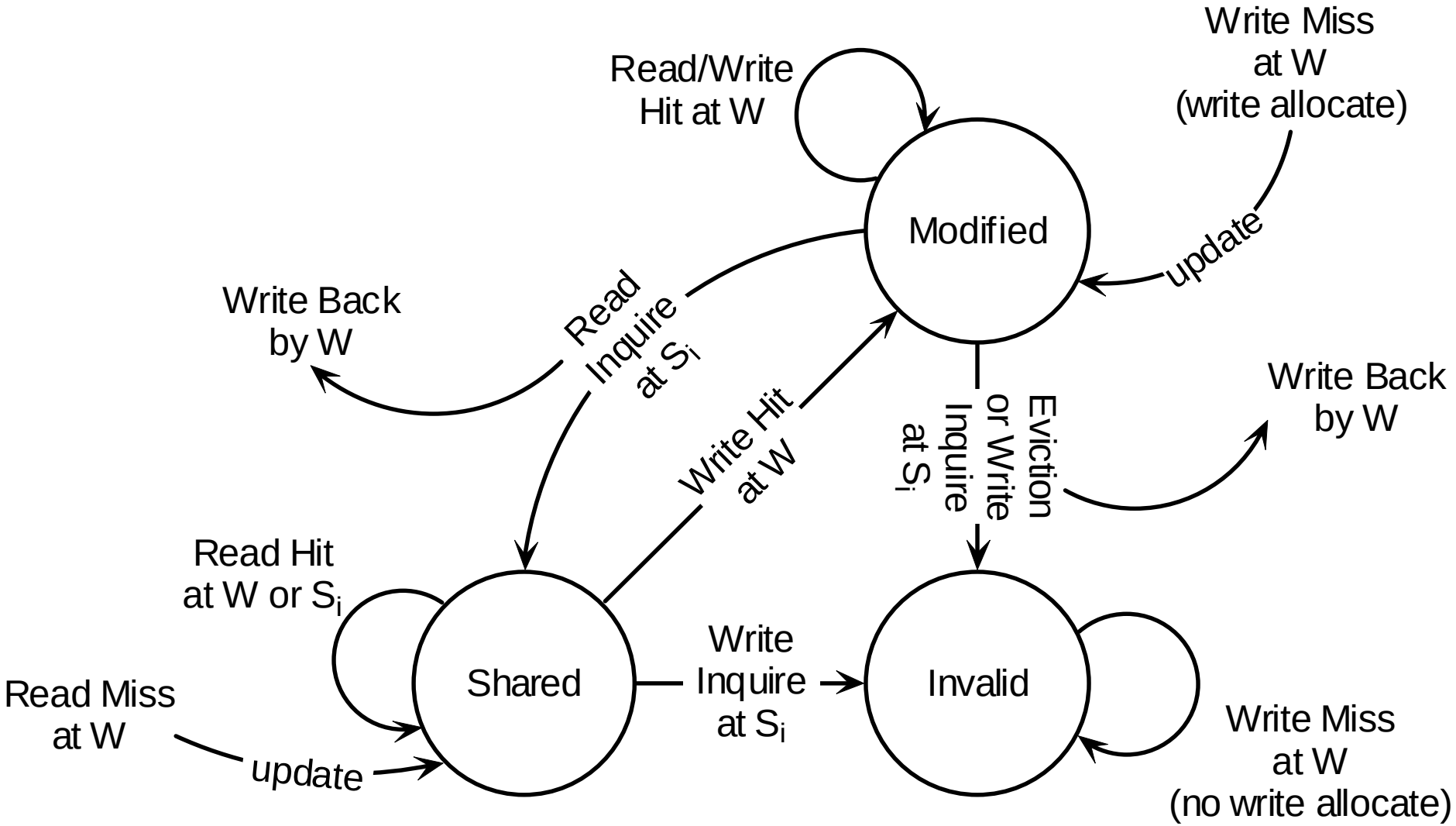
- Writes update address to memory bus
  - Processors S_i mark block invalid
- Updates cache
  - W marks block Modified

Processor Behavior — Invalid

Invalid

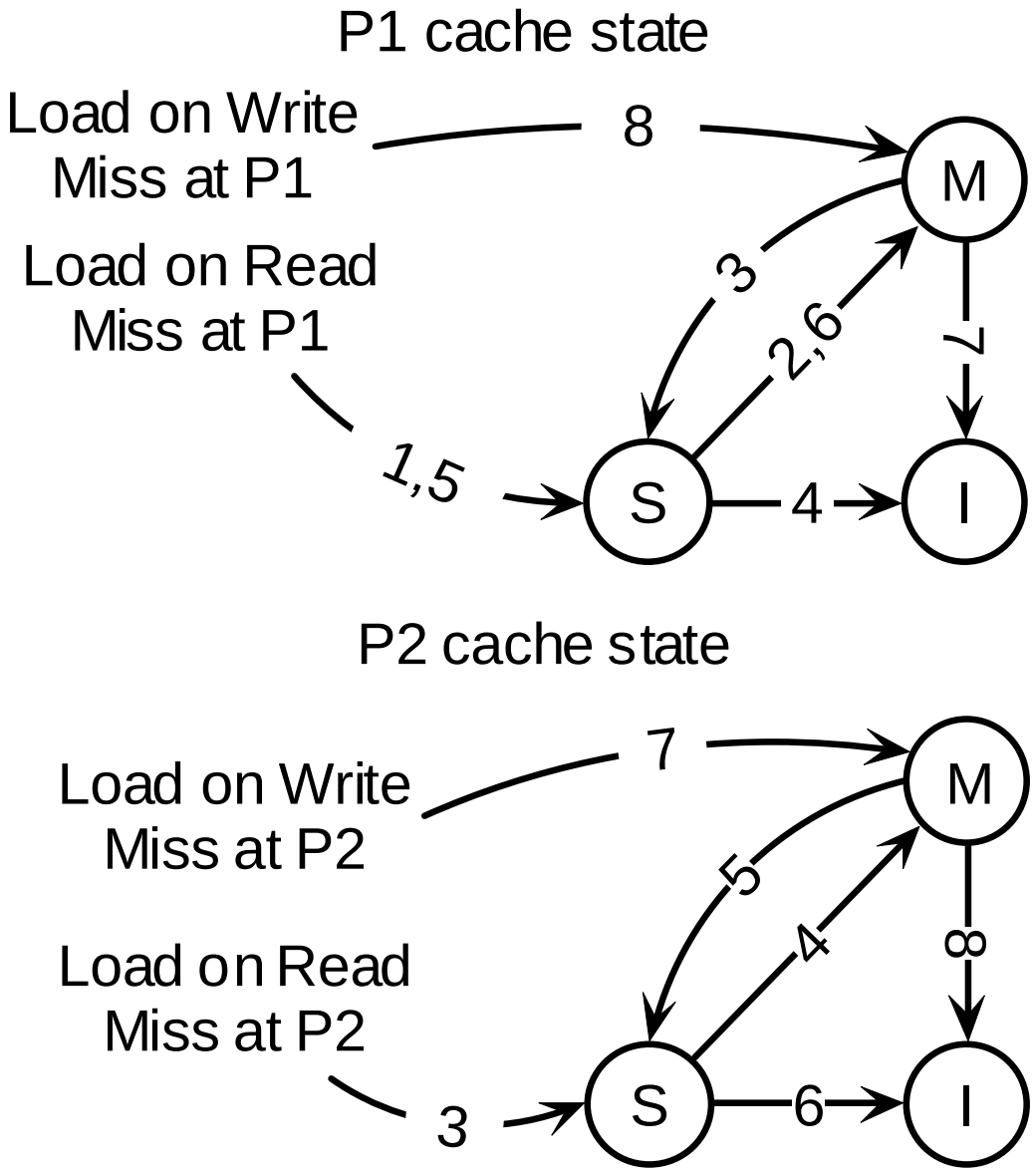
- Block must be reloaded before next read
  - State includes blocks tagged invalid and blocks not in cache
- Read and write accesses are cache misses
- Write allocate
  - Write miss at W initiates cache update
- No write allocate
  - Write miss at W does not initiate cache update
  - Write directly to memory

MSI Protocol State Diagram



MSI Example

	P1	P2
1	Read	
2	Write	
3		Read
4		Write
5	Read	
6	Write	
7		Write
8	Write	



Processor Behavior — Exclusive

Processor W cache holds only valid copy of data line

- Line is clean
  - Memory copy is same as W copy
- Cache copies in other processors S_i marked invalid
  - S_i inquire (request update from this copy) before accessing block

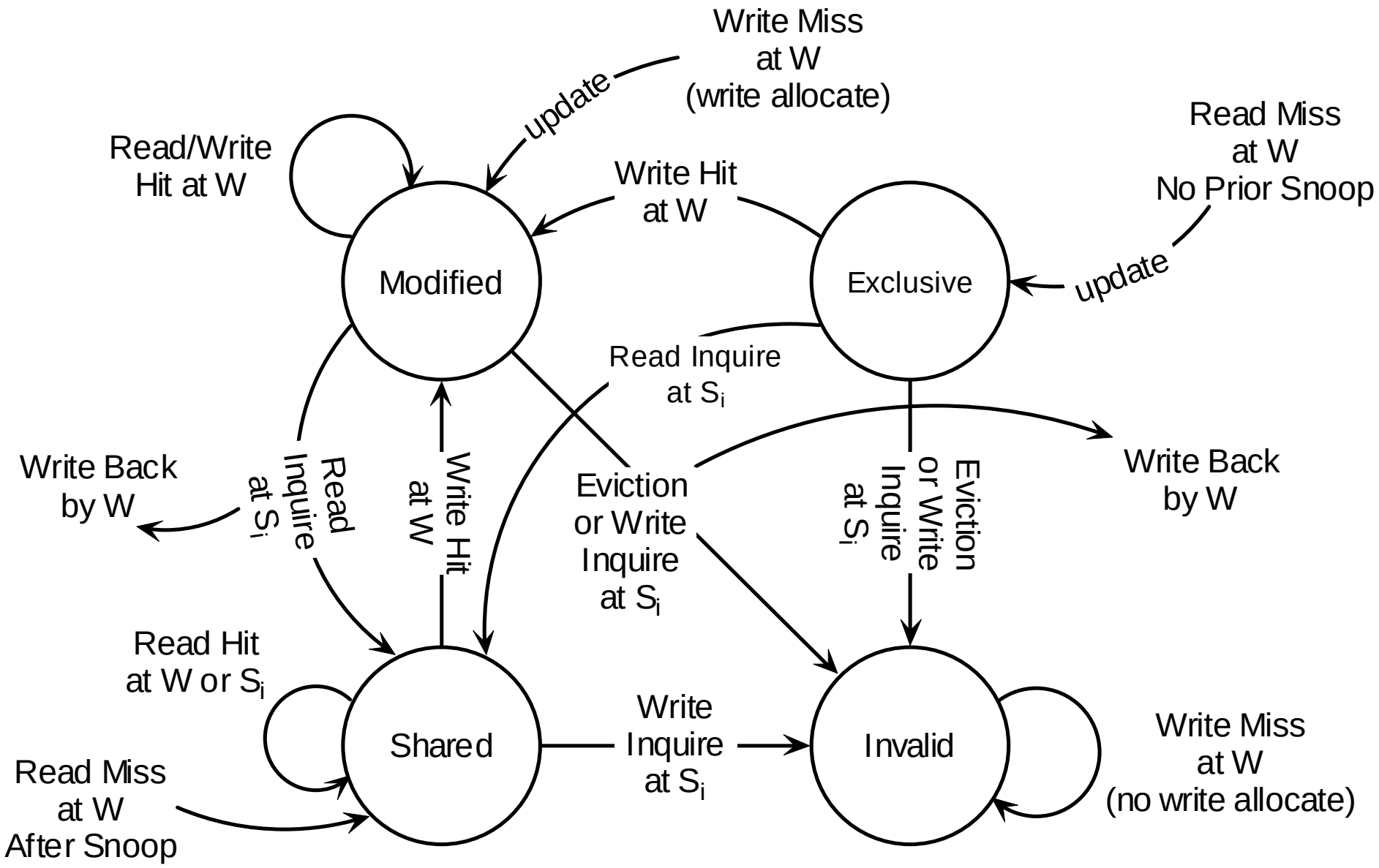
W responds to inquire

- Snoops memory address from S_i on bus
- W updates memory
- Marks block invalid on write inquire

Improved performance on private data

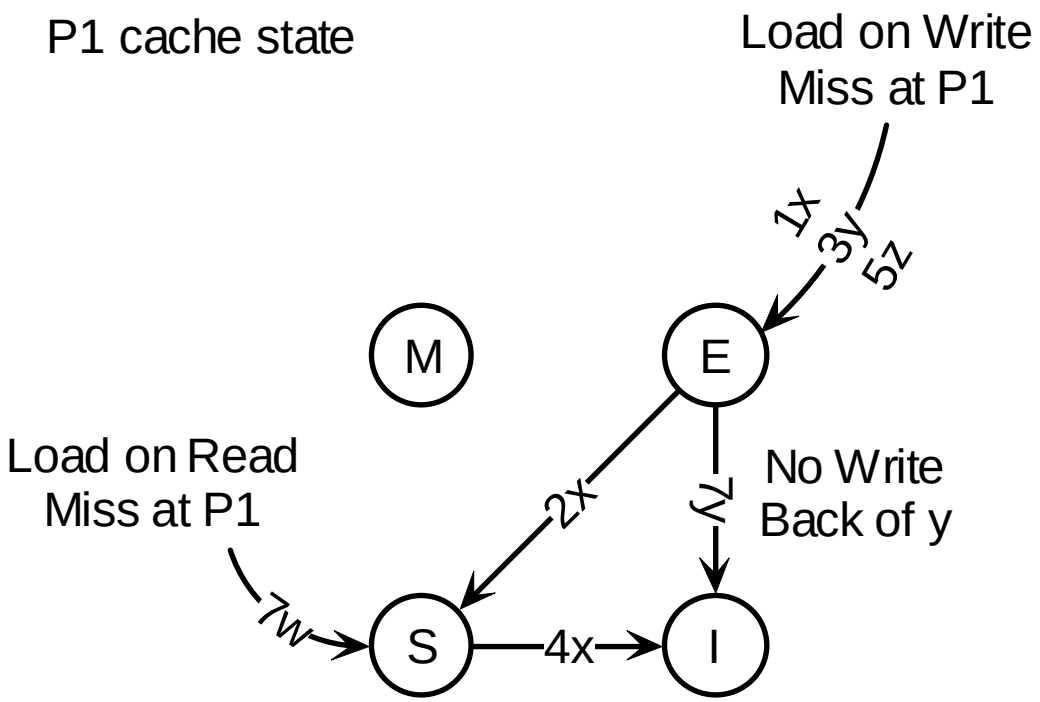
- No write back to memory on eviction of exclusive block

MESI Protocol State Diagram

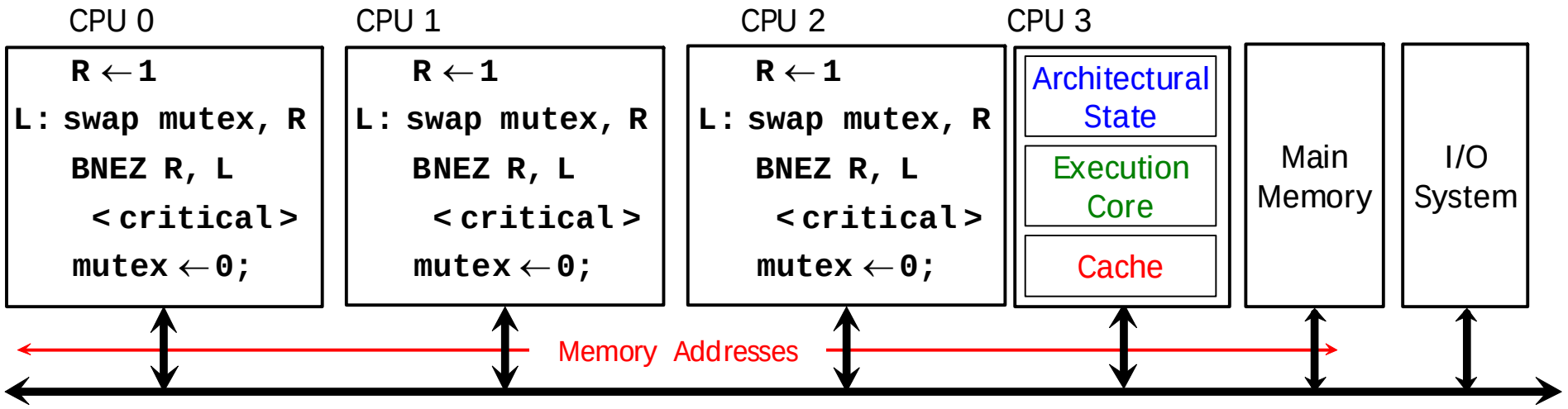


MESI Example

	P1	P2
1	Read x	
2		Read x
3	Read y	
4		Write x
5	Read z Replaces x in cache	
6		Read w
7	Evict y without write back Read w	
8		Evict x with write back Read y



Memory Thrashing with MSI



Value of shared variable mutex

- Required to enforce criticality
- Each swap causes
  - Read miss and cache update
  - Write to one local cache (write allocate)
  - Invalidation all other caches (write invalidate)

Inefficient movement of data across bus

- Multiple mutex cache misses/loads creates large overhead
- Overhead much larger than read/writes for critical section

Load-Reserve and Store-Conditional with MSI

Implement critical section

Use load-reserve/store-conditional

No shared mutex variable

```
   R ← 1
L: swap mutex, R;
   BNEZ R, L;
     <critical(x)>
   mutex ← 0;
```

```
L: load-reserve R, x
   <critical>
   store-conditional x, R
   BEQZ status, L
```

Each processor has private reservation flag

Flag set on load-reserve

Flag checked on store-conditional attempt

Snooping cache manager clears flag on write to memory location

Requires restart of load and critical section

Improved overall efficiency

Overhead created by mutex → multiple reads of critical variable **x**

Load-reserve/store-conditional eliminates stores to mutex variables

False Sharing

False sharing

Multiple threads access distinct address ranges in same cache block

Write to block causes invalidation in other processors

Creates memory thrashing

Example

```
int counter[THREAD_NUM];
int inc_counter ()
{
counter[my_tid]++;
return counter[my_tid];
}
```

Solution

Avoid small array accessed by multiple threads

Avoid global data or static data variables in same cache block

Cache Coherence and Hardware Optimizations

Memory hardware optimizations

- Equivalent to memory semantics within processor

Cache coherence

- Enforces sequential consistency between processors

Write buffer with read-bypass

- Cache/memory updates follow cache invalidation
  - Load cannot bypass store on two processors

Write overlapping

- Writes invalidate other cache copies
- Physical writes occur in correct order

Read / write reordering

- Writes invalidate other cache copies
- Rescheduled read invalidated when other CPU writes to variable

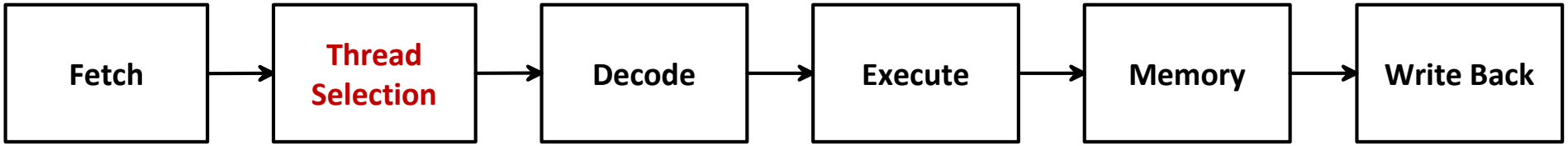
Sun T1 Multiprocessor

Server processor

- Introduced by Sun Microsystems in 2005
- Based on UltraSPARC architecture
- Optimized for TLP for online transaction processing (OLTP)

Core

- 8 processor cores
  - Single-issue — not superscalar
  - In-order execution — no dynamic scheduling
  - 6 stage pipeline — 5 standard stages plus thread selection



Threads

- 4 threads per core
  - Fine grained multithreading
  - Switches thread on each clock cycle
  - Skips over stalled threads

Sun T1 Organization

8 processor cores

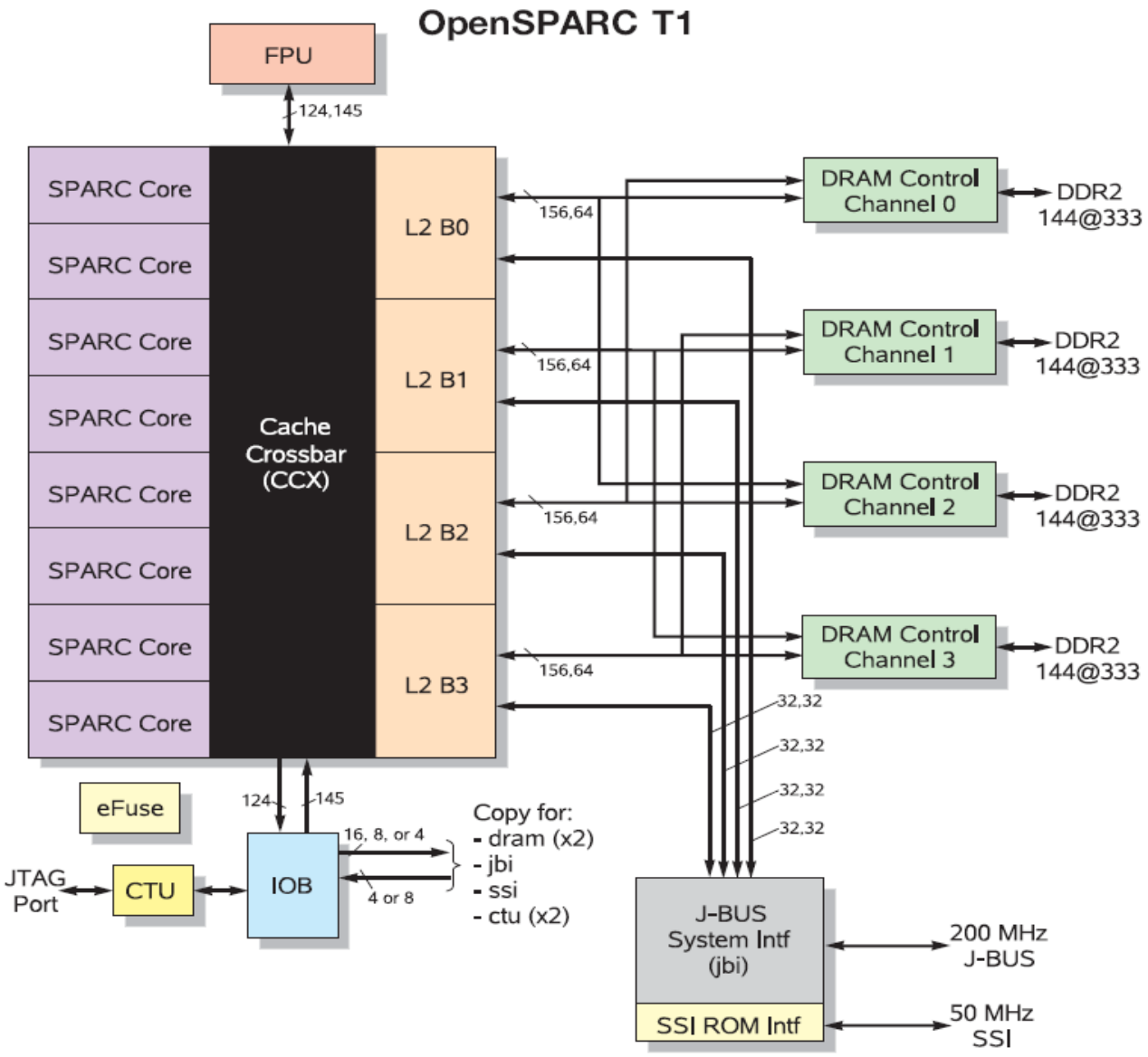
- 1 shared FPU
- Fine grain multithreading
  - 4 threads per core
- 1.4 GHz clock

Private L1 cache

- 16 KB instruction
- 8 KB data
- 64-byte block
- Every L1 block in L2
- L1 miss penalty = 23 CC

Shared L2 cache

- 3 MB in 4 banks
- L2 miss penalty = 110 CC
- Distributed directory in L2
  - Maintains coherency among L1 caches



Oracle (Sun) T4 Multiprocessor

T1 (2005) → T2 (2007) → T3 (2010) → T4 (2011)

	T1	T4
Threads / core	4	8
Threads / chip	32	64
Pipeline model	In-order	Dynamic scheduling
Pipeline stages	6	16
Shared EUs	1 FPU	1 FPU + 1 Cryptographic
Clock rate	1.4 GHz	3.0 GHz
L1 cache	8 × 16 KB I + 8 × 8 KB D	8 × 16 KB I + 8 × 16 KB D
L2 cache	3 MB	8 × 128 KB
L3 cache	—	4 MB

Oracle T4 server

4 SPARC T4 CPUs (256 threads) + 1 TB RAM

25% faster than HP 4 Xeon server on OLTP benchmark TPC-H

http://www.oracle.com/us/solutions/performance-scalability/sparc-t4-4-tpc-113011-bmark-1385055.html