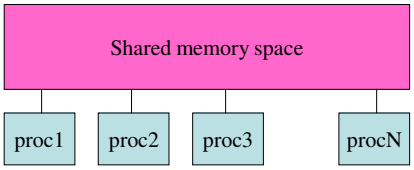
Shared Memory: Synchronization, Coherence, and Consistency

Sandhya Dwarkadas
University of Rochester

Shared Memory: A Look Underneath



Shared Memory Implementation

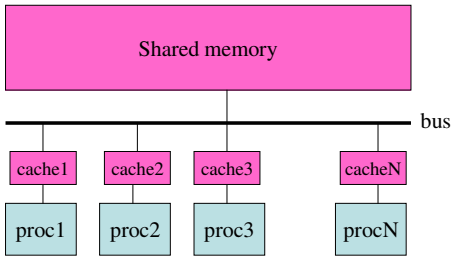
- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Coherence

A multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the result of the execution and

- it ensures that modifications made by a processor propagate to all copies of the data
- program order is preserved for each process in this hypothetical order
- writes to the same location by different processors are serialized and the value returned by each read is the value written by the last write in the hypothetical order

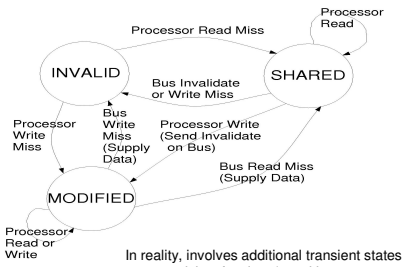
Physical Implementation



Snoop-Based Coherence

- Makes use of a shared broadcast medium to serialize events (all transactions visible to all controllers and in the same order)
  - Write update-based protocol
  - Write invalidate-based (e.g., basic MSI, MESI protocols)
- Cache controller uses a finite state machine (FSM) with a handful of stable states to track the status of each cache line
- Consists of a distributed algorithm represented by a collection of cooperating FSMs

A Simple Invalidate-Based Protocol - State Transition Diagram



In reality, involves additional transient states to deal with non-atomicity of actions/transitions

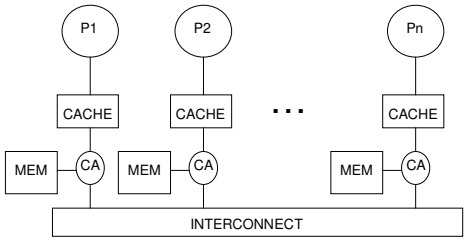
Correctness Requirements

- Need to avoid
  - Deadlock – caused by a cycle of resource dependencies
  - Livelock – activity without forward progress
  - Starvation – extreme form of unfairness where one or more processes do not make forward progress while others do

Directory-Based Coherence

- Distribute memory, use point-to-point interconnect for scalability
- Need to manage coherence for each memory line – state stored in directory
  - Simple memory-based (e.g., DASH, FLASH, SGI Origin, MIT Alewife, HAL)
  - Cache-based (linked list (e.g., Sequent NUMA-Q, IEEE SCI)

Cache-Coherent - NUMA



Simple Memory-based Directory Coherence

- Advantage
  - Precise sharing information
- Disadvantage
  - Space/storage proportional to PxM
- Work-around for either width or height
  - Increase cache block size
  - 2-level protocol
  - Limited pointer scheme
  - Directory cache

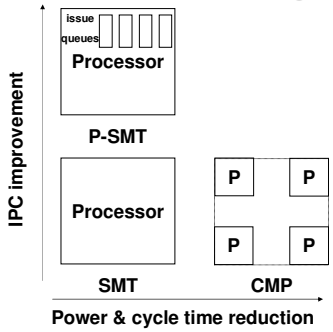
Cache-Based Directory Coherence

- Home main memory contains a pointer to the first sharer + state bits
- Pointers at each cache line to maintain a doubly-linked list
- Advantage – reduced space overhead
- Disadvantage – serialized invalidates (latency and occupancy)

The Case for CMPs

- Examples: Power4/5, Intel P4, Piranha, Niagara, Hydra, Rock
- Why are single-chip multiprocessors a good idea?
  - Parallelism – ILP and TLP
  - Wire delays
    - Wires connecting processors more infrequently used and therefore less critical
  - Design time
  - Energy reduction

Multi-Threaded Processor Organizations



SMT : Simultaneous Multi-Threading CMP : Chip Multi-Processor
P-SMT : Partitioned SMT

Motivating Applications

- Commercial applications (web servers) with high levels of TLP but low ILP due to
  - Large working sets
  - Poor locality
  - Data dependent branches
- Power density of data centers
  - Google [IEEE Micro '03]: 400-700 W/sq. ft. for racked server clusters (currently support 70-150 W/sq. ft.)

Niagara [IEEE Micro'05]

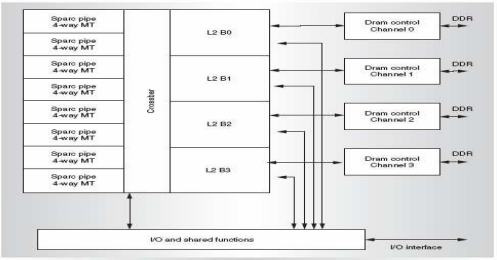


Figure 2. Niagara block diagram.

Implications: Recognizing Sharing Patterns, Exploiting Locality

- Predictable vs. unpredictable
- Regular vs. irregular
- Coarse vs. fine-grain (contiguous vs. non-contiguous in the address space)
- Near-neighbor vs. long range in an interconnection topology
- In terms of invalidation patterns
  - Read-only
  - Producer-consumer
  - Broadcast/multicast
  - Migratory
  - Irregular read-write

Synchronization

- Basic types
  - Mutual exclusion
  - Events
- Components of a synchronization operation
  - Acquire method (enter critical section, proceed past event)
  - Waiting algorithm (busy waiting, blocking)
  - Release method (enable others to proceed)

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```


Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4      ; mov exchange value
     ll    R2, 0(R1) ; load linked
     sc R3, 0(R1)    ; store conditional
     beqz R3, try    ; branch if store fails
     mov  R4, R2     ; put load value in R4
```


Spinlock Algorithms

- Test&test&set (w, w/o exponential backoff)
- Ticket lock (w, w/o proportional backoff)
- Array based queue locks
- MCS linked-list based queue locks

MCS Lock [TOCS'91]

- Features
  - Linked-list based queue
  - O(1) network transactions required
  - Spins on local variables
- Needs two atomic instructions
  - Fetch_and_store(L, I)
  - Compare_and_swap(L, I, NIL)

MCS Lock in detail

```
type qnode = record
    next : ^qnode locked : Boolean
type lock = ^qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil                    // queue was non-empty
        I->locked := true predecessor->next := I
        repeat while I->locked               // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil                         // no known successor
        if compare_and_store (L, I, nil)
            return                           // compare_and_store returns true iff it stored
        repeat while I->next = nil   // spin
    I->next->locked := false
```

Performance Goals

- Low latency, short critical path
- Low traffic
- Scalability
- Low storage cost
- Fairness

Coordination Properties

- Lock-free data structures
  - Operations defined on it do not require mutual exclusion over multiple instructions (use atomic primitives)
- Non-blocking algorithms
  - Operations guarantee that some process will complete its operation in a finite amount of time, even if other processes halt
- Wait-free algorithms
  - Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

Memory Consistency Models

When must a processor see a value that has been updated by another processor?

P1: A = 0;	P2: B = 0;
…	…
A = 1;	B = 1;
L1: if (B == 0) …	L2: if (A == 0) …

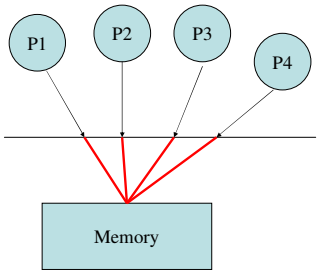
Consistency

- Defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Sequential Consistency

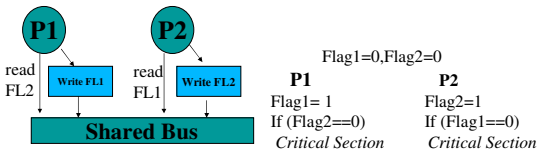
- ``A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport '79]
  - In practice, this means that every write must be seen on all processors before any succeeding read or write can be issued

Programmer's View of Sequential Consistency



Write Buffers [Computer'96]

Write Buffer with bypassing
- Reads bypass writes, Reads are blocking
- Write placed into buffer and processor continues



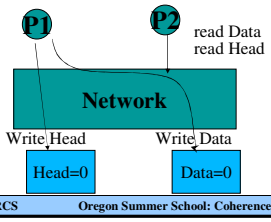
Flag1=0,Flag2=0

P1	P2
Flag1= 1	Flag2=1
If (Flag2==0)	If (Flag1==0)
Critical Section	*Critical Section*

Bypass violates sequential consistency

Pipelined Writes [Computer'96]

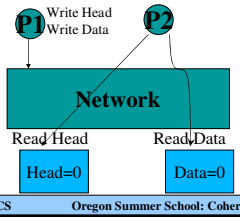
Writes may bypass other writes in write buffer
- Improves buffer utilization
- Utilizes bandwidth of multiple memory banks.
- Point-Point interconnection network alleviates bus bottlenecks.



P1	P2
Data=4000	While(head==0){}
Head=1	A=Data

Non-Blocking Reads [Computer'96]

Reads are allowed to bypass reads and writes
- Dynamic Scheduling, Lockup-free caches, Speculative execution
- Point-Point interconnection network alleviates bus bottlenecks.



P1	P2
Data=4000	While(head==0){}
Head=1	A=Data

Orderings Among Reads and Writes

- Orderings among reads and writes performed by a single processor
  - R->R : a read followed by a read
  - R->W: a read followed by a write
  - W->W: a write followed by a write
  - W->R: a write followed by a read
- If there is a dependence between the read and the write, then uniprocessor program semantics demand that the operations be ordered.
- If there is no dependence, the memory consistency model determines what orders must be preserved
- Relaxing an order implies that we allow an operation executed later by the processor to complete first.

Relaxed Memory Consistency Models

- Characteristics
  - How they relax the program order requirement
    - W->R, W->R, R->R, R->W operations to different memory locations
  - How they relax the write atomicity requirement
    - Whether they allow a read to return the value of a processor's write before the write is made visible to all other processors

Relaxation Categories

- Relax write to read program order
- Relax write to write program order
- Relax read to read and read to write program orders
- Read other's write early
- Read own write early

Hardware-Provided Consistency Models [from Computer'96]

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

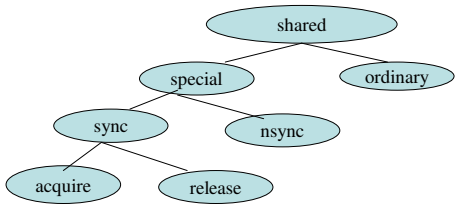
Example

P1 *P2* *P3*
A=1 *if (A=1)* *if (B=1)*
B=1 *R1=A*
Result R1=0 B=1

Which Model permits the above result ?
370 – No
TSO – No
PC - Yes

Release Consistency

- acquire→all, all→release, special→special



Safety net

- Relaxed models also typically provide programmers with mechanisms for overriding such relaxations. Such mechanisms are called a safety net. (e.g serialization instructions, RMW, release, acquire)

Memory Model Requirements

- Balance between
  - Ease of use
  - Transformations and optimizations in
    - Compilers
    - Hardware

Library-Based Approach to Concurrency

E.g., pthreads
- Programs must be data-race-free by inserting calls to pthreads synchronization routines (which in turn contain memory barriers)
- Calls to pthreads functions are treated as opaque (may read or write any global variable), implying that memory references cannot be moved across the call

Problems: Correctness Issues [PLDI'05]

- Does not sufficiently define when a data race exists in the original program, or when the compiler may introduce one
  - Concurrent modifications
  - Rewriting of adjacent data
  - Register promotion

Example: Concurrent Modification

- A correctly synchronized program is one in which there are no data races when executed in a sequentially consistent manner

P1: if (x == 1) ++y;

P2: if (y == 1) ++x;

Problems: Performance Issues

- Practical benefits of parallel algorithms involving races
  - E.g., gauss-seidel iteration, sieve of Eratosthenes, mark-sweep garbage collection
- What reorderings are permitted?
- Once we allow data races, language specification and compiler implementation must clearly specify semantics

The Java Memory Model [PoPL'05]

- Guarantees sequential consistency to data-race-free programs
- Bounds the behavior of incorrectly synchronized programs by a well-defined notion of causality
  - Respects safety and security properties

Sequential Consistency Restricts Reordering

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

Re-ordering not permitted regardless of data or control dependence

Self-Justifying Write Speculation

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

Figure 2: An Out Of Thin Air Result

Ensuring Ordering

Initially, x == 0, ready == false. ready is a volatile variable.

Thread 1	Thread 2
x = 1;	if (ready)
ready = true	r1 = x;

If r1 = x; executes, it will read 1.

Figure 3: Use of Happens-Before Memory Model with Volatiles

Ensure Causality

Correctly synchronized programs must have sequentially consistent semantics

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

Control Dependence Breaking Optimization (allowable)

Before compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

After compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: r2 = r1;	
3: if (true) ;	

r1 == r2 == r3 == 2 is sequentially consistent

Figure 5: Effects of Redundant Read Elimination

Data Dependence Breaking Optimization (allowable)

Initially, x = y = 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

Is r1 == r2 == r3 == 1 possible?

Figure 6: Using Global Analysis

Unallowed Behavior

Initially, x == y == 0

Thread 1	Thread 2
do { r1 = x;	do { r2 = y;
} while (r1 == 0);	} while (r2 == 0);
y = 42;	x = 42;

Correctly synchronized, so non-termination is the only legal behavior

Figure 13: Correctly Synchronized Program

Impact of the Model

- Considerations for implementors
  - Control dependences
  - Allowable reorderings
- Considerations for programmers
  - Correctly synchronized programs do not need to worry about reorderings
    - Exhibit sequentially consistent behavior