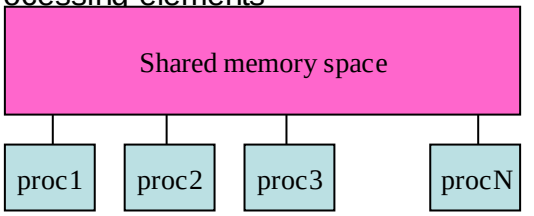
Shared Memory: More on Coherence and Consistency

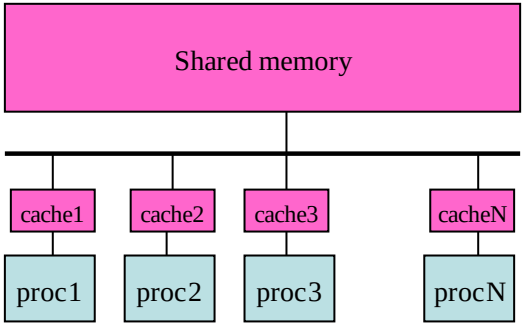
Sandhya Dwarkadas
University of Rochester

What is Shared Memory?

Shared memory: Memory that may be simultaneously accessed by two or more processing elements



Coherent 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

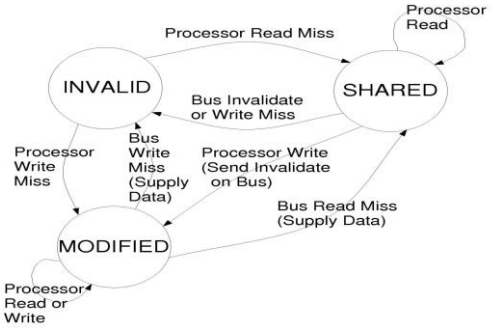
A multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the result of the execution and ensure that

- modifications made by a processor propagate to all copies of the data (write propagation)
- operations by each process occur in the order in which they were issued to the memory system by the process
- writes to the same location by all processors are serialized (write serialization) and the value returned by each read is the value written by the last write in the hypothetical order
-

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



Correctness Requirements

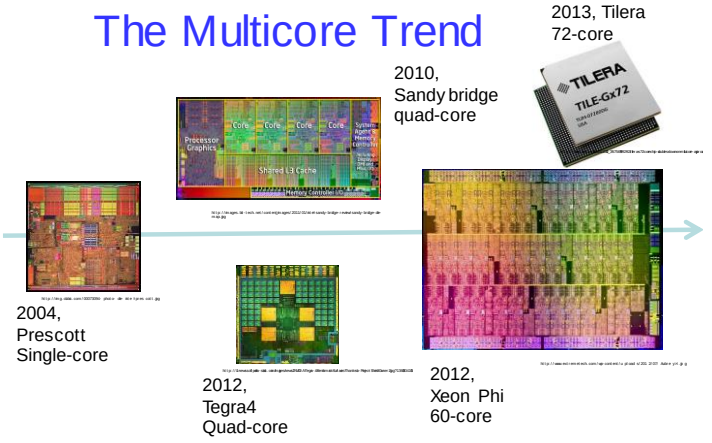
- Need to avoid
  - Deadlock – caused by a cycle of resource dependencies
  - Livelock – activity without forward progress
  - Starvation – extreme form of unfairness where one or more processes do not make forward progress while others do

Design Challenges

- Cache controller and tag design
- Non-atomic state transitions
- Serialization
- Cache hierarchies
- Split-transaction buses

Multicore Processors Everywhere

The Multicore Trend

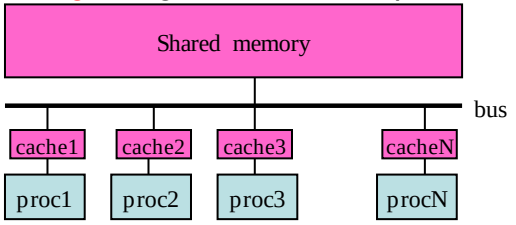


Scaling Challenges

- Bandwidth and shared resource contention
- Non-uniform memory access
- Power/energy efficiency
- Scalable coherence implementation

Snoop-Based or Broadcast Coherence

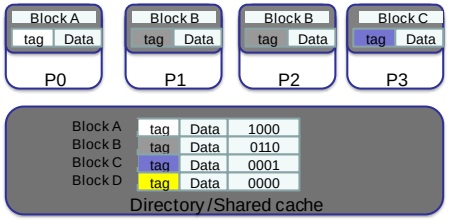
- Make use of a broadcast medium to manage replicas
- Benefit: Low metadata requirements
- Challenge: High bandwidth requirements



Solution: Directory-based Cache Coherence

Directory: maintain per-core sharer information to save bandwidth

Full map: associate sharing vector with tags of shared L2



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)

Multiprocessor Interconnects

- Topology
- Routing algorithm
- Switching strategy (circuit vs. packet)
- Flow control mechanism

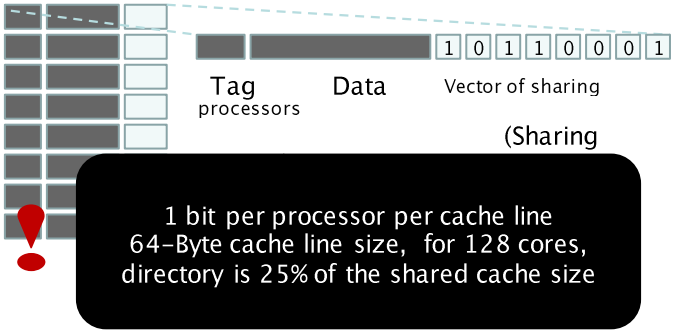
Interconnect Topologies

- Fully connected
  - Single large switch
  - Bus
- Linear arrays and rings
- Multi-dimensional meshes and tori
- Trees
- Butterflies
- Hypercube

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

Conventional Full Map Directory



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)

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

Memory Consistency Model

- Specifies constraints on the order in which memory operations to different locations must appear to be performed with respect to one another

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

Implications

- Program order

P1	P2
A = 1;	while (flag == 0);
flag = 1;	print A;

- Write atomicity

P1	P2	P3
A = 1;	while (A == 0);	
	B = 1;	while (B == 0);
		print A;

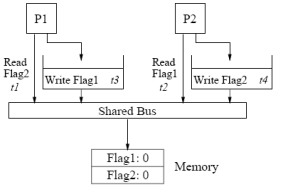
Dekker's Algorithm

```
P1: A = 0;                  P2: B = 0;
    …                           …
    A = 1;                      B = 1;
L1: while (B == 1) {..}     L2: while (A == 1) {…}
    …                           …
```

Can B = 0 at P1 and A = 0 at P2 at the corresponding if statements?

Write Buffers [Bypassing Capability]

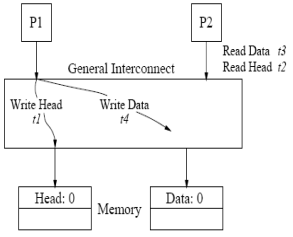
- Reads bypass writes, reads are blocking



P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Overlapping Write Operations

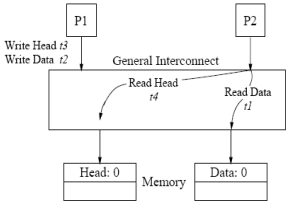
- Writes may bypass other writes in write buffer



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Non-blocking Reads

- Reads are allowed to bypass reads and writes



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(c) non−blocking reads

Drawbacks of Sequential Consistency

- SC restricts any compiler optimization that can result in reordering memory operations
  - Code motion, register allocation, common sub-expression elimination, loop blocking, software pipelining
- SC restricts hardware generated memory re-orderings because of program-order and write-atomicity requirements
  - Write Buffers, OOO instruction issue, pipelining of memory operations, lock-up free caches, non-atomic memory operations
- Potential performance penalties from the above

Memory Model Relaxations

- Possible relaxations
  - Write → Read
  - Write → Write
  - Read → Read, Write
  - Read other's write early
  - Read own write early
- All Models provide some Safety net (memory fence/ordering instructions or prefixes)
- All models maintain uni-processor data and control dependencies
- Write serialization is maintained by all the models except PC, RCpc, PowerPC *[for most practical purposes where all processors observe all write operations in the same order (write serialization), is indistinguishable from a system where all writes are executed atomically]*

Categorization of Relaxed Models

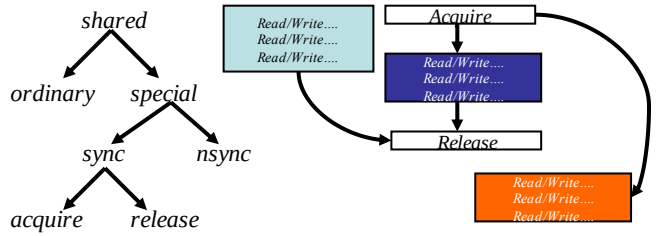
Relaxation:	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

Relaxing All Program Orders

- Read or a Write operation may be reordered w.r.t following read or write to a different location
  - Weak Ordering Model
    - Release Consistency Model (RCsc / RCpc)
    - Digital Alpha, Sparc V9 RMO, IBM Power PC
- Except Alpha, the above models allow reordering of two reads to the same location.
- RCpc and PowerPC allow a read to return the value of another processors write early.

Release Consistency

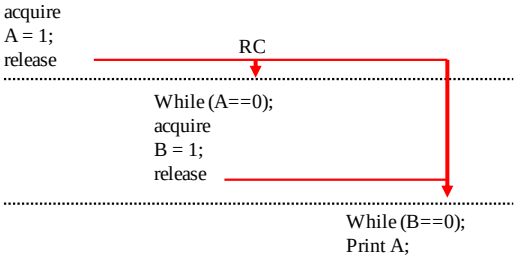
- Distinguishes between ordinary and special memory accesses
- Ordinary accesses are completely unordered with respect to each other
- Divides synchronization into acquires and releases



RC Example

Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully

Before a release is allowed to be performed, all previous reads and writes by the process must have completed (therefore, example below should always print A=1)



Threads cannot be implemented as a library [Boehm, PLDI'05]

- E.g., C/C++ with pthreads library

  "Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads."
- How is this enforced?
  - Programs must use library-provided synchronization
  - Memory barrier instructions used within the library synchronization to prevent hardware memory access reordering
  - Synchronization calls treated as opaque to avoid compiler memory access reordering

Example: Concurrent Modification

- Does this program contain a race?

	Initially, x=y=0	Compiler transformation
T1:	if(x==1) ++y;	++y; if(x!=1) –-y;
T2:	if (y==1) ++x;	++x; if (y!=1) --x;

- No race in original program since no variable can become non-zero
- … if the compiler makes the modifications on the right, there is a race!

Summary

- Non-atomic state transitions complicate coherence implementation
- Directory protocols used to scale processors to large core counts
- Relaxed consistency models allow read/write reorderings
  - Implications for the hardware
  - Implications for the compiler