Shared Memory Multiprocessors

Multiple processors harmoniously use same memory.
Most prevalent case, symmetric multiprocessors (SMP):
- processors have identical access to main memory
- dominates server market; migrating to workstation/desktop

SMPs are attractive for several reasons:
- uniform access via loads/stores
- automatic data movement and coherent replication in caches
- easy to build from uniprocessors

We will focus on modifications to memory hierarchy to support shared memory.

Supporting Programming Models on Shared Memory Multiprocessors

<table>
<thead>
<tr>
<th>Shared address space</th>
<th>Message passing</th>
<th>Multiprogramming</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compilation</td>
<td>Operating systems support</td>
</tr>
<tr>
<td></td>
<td>or library</td>
<td>Physical communication medium</td>
</tr>
<tr>
<td>Supports both programming models well.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shared Address Space
- Works magically.
- Address translation and protection in hardware (hardware SAS).

Message Passing
- Use shared memory buffers.
- Can be very high performance since no OS involvement necessary.
Natural Extensions of Memory System

Caches and Cache Coherence

Caches play key role in all cases:
- Reduce average data access time.
- Reduce bandwidth demands placed on shared interconnect.

But private processor caches create a problem (cache coherence):
- Copies of a variable can be present in multiple caches,
- A write by one processor may not become visible to others (stale values).

Focus: Bus-based, Centralized Memory

Shared cache
- Low-latency sharing and prefetching across processors
- Sharing of working sets
- No coherence problem (and hence no false sharing either)
- But high bandwidth needs and negative interference (e.g., conflicts)
- Hit and miss latency increased due to intervening switch and cache size
- Mid 80s: to connect couple of processors on a board (Encore, Sequent)
- Today: for multiprocessor on a chip (for small-scale systems or nodes)

Dancehall
- No longer popular: everything is uniformly far away...

Distributed memory
- Most popular way to build scalable systems, discussed later.

Coherent Memory System

Basic tenant: Read of location A should return the latest value written to location A (by any process).

Could happen on uniprocessor systems, too if not careful (due to out-of-order issue).

But obviously coherence problem much more critical in multiprocessors:
- It is pervasive,
- It is performance-critical.
- Consequently, must be treated as a basic hardware design issue,
Example Cache Coherence Problem

Processors see different values for u after event 3.
With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when (processes accessing main memory may see very stale values).
Unacceptable to programs, and frequent!

Basic Definitions

Extend from definitions in uniprocessors to those in multiprocessors:

*memory operation*: a single read (load), write (store) or read-modify-write access to a memory location (assumed to execute atomically)

*issue*: a memory operation issues when it leaves processor's internal environment and is presented to memory system

*perform*: operation appears to have taken place, as far as processor can tell from other memory operations it issues

*complete*: perform with respect to all processors

What Does “last” Mean?

Recall: Value returned by read should be last value written,
But “last” is not well-defined.

Even in sequential case, last defined in terms of program order, not time:

- Order of operations in the machine language presented to processor,
- “Subsequent” defined in analogous way, and well defined.

In parallel case, program order defined within a process, but need to make sense of orders across processes.

Must define meaningful semantics.

More on Write and Reads

Write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write.

Read performs w.r.t. the processor when subsequent writes issued by the processor cannot affect the value returned by the read.
Serial Order

Imagine a single shared memory and no caches:

- Every read and write to a location, accesses the same physical location
- Operation completes when it does so.

Memory imposes a serial or total order on operations to the location:

- Operations to the location from a given processor are in program order.
- The order of operations to the location from different processors is some interleaving that preserves the individual program orders.

Formal Definition of Coherence

Results of a program: values returned by its read operations

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

1. Operations issued by any particular process occur in the order issued by that process, and
2. Value returned by a read is the value written by the last write to that location in the serial order.

Now, what does “last” mean?

“Last” now means most recent in a hypothetical serial order that maintains these properties.

For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look).

Note that the total order is never really constructed in real systems (don’t even want memory, or any hardware, to see all operations).

But program should behave as if some serial order is enforced.

This is an order in which things appear to happen, not actually happen.

Implicit Properties

Two important properties concerning writes:

Write propagation: value written must become visible to others

Write serialization: writes to location seen in same order by all
- if I see w2 after w1, you should not see w2 before w1
- no need for analogous read serialization since reads not visible to others
Bus-based Coherency

Implement coherency using two fundamentals of uniprocessor systems (bus transitions and cache states).

Uniprocessor bus transaction:
- Three phases: arbitration, command/address, data transfer
- All devices observe addresses, one is responsible.

Uniprocessor cache states:
- Effectively, every block is a finite state machine.
- Write-through, write no-allocate has two states: valid, invalid.
- Writeback caches have one more state: modified (“dirty” = more on this later)

Snoopy Cache - Implementation

Local cache controller now receives inputs from both its processor and snooper.

In either case, takes zero or more actions:
- update state
- respond with data
- generate new bus transaction

Protocol is distributed algorithm (via cooperating state machines).
Granularity of coherence is typically cache block (same as allocation and transfer granularity).

Snoopy Cache - Basic Idea

Central Concept: Transactions on bus are visible to all processors.
Processors or their representatives can snoop (monitor) bus and take action on relevant events.

Snoopy Coherency Protocols

So, this is what you have all been waiting for...
Coherency protocols are near the center of the multiprocessor universe.
- crucial for high-performance
- rich design space/many choices
- numerous design tradeoffs

Next time we will look at a number of protocols, but today we will introduce one simple example: write-through/write-invalidate

But first we need to review some cache design choices,
**Write Policy**
Design Question: What happens on a store (write to cache)?

write through - store operations sent immediately to next level
- pro: system memory is always up to date
- con: increases traffic ratio

write (copy) back - stores are written into the cache; modified (“dirty”) data is passed on to next level eventually
- pro: reduces traffic ratio
- con: need to keep track of dirty state

---

**Write Allocation Policy**
Design Question: What happens on a store miss?

write allocate - stores deposit copy of data in cache
- minimize latency if read-after-write (RAW) is common (e.g. heap and stack allocation)
- can complicate hardware design
- used commonly with small block sizes

write no allocate - stores bypass the cache entirely (next level)
- good for typical programs which perform write-after-read (WAR)
- leads to simple hardware (usually)
- used commonly with large block sizes

---

**Coherence with Write-through Caches**

Key extensions to uniprocessor: snooping, invalidating/updating caches
- no new states or bus transactions in this case
- invalidation- versus update-based protocols

Write propagation: even in invalidation case, later reads will see new value
- invalidation causes miss on later access, and memory up-to-date via write-through

---

**Write-through State Transition Diagram**

Two states per block in each cache, as in uniprocessor
- state of a block can be seen as p-vector
- Hardware state bits associated with only blocks that are in the cache
- other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
- can have multiple simultaneous readers of block, but write invalidates them
Protocol Coherency

It seems to work in our small example, but we need to prove correctness.

Does this protocol satisfy requirements 1 (program order) and 2 (serial order)?

We need some more definitions to be able to answer this...

Atomic Bus and Memory

Simple assumptions for now, later on we will relax these...

All phases of one bus transaction complete before next one starts.
Processor waits for memory operation to complete before issuing next.
With one-level cache, assume invalidations applied during bus transaction.

Write-Through/Write-Invalidate Example

Initial State: A = 5, B = 2

What is the final state of the caches and memory under the following sequence?

P1

Read A  Write A=101  Write B=20

P2

Read A  Read B  Read A  Write A=0

Protocol Coherency: Writes Look Good

Does this protocol satisfy requirements 1 (program order) and 2 (serial order)?

All writes go to atomic bus:
  - Writes are serialized by order in which they appear on bus (bus order).
  - Hence invalidations are applied to caches in bus order.

Writes are fine, but what about reads?
  - Read hits may be independent
  - Read hits do not appear on bus
Read Ordering

Read misses: appear on bus, and will see last write in bus order, no problem.
Read hits: do not appear on bus...
  - But value read was placed in cache by either
    - most recent write by this processor, or
    - most recent read miss by this processor
  - Both these transactions appear on the bus
  - So reads hits also see values as being produced in consistent bus order
So the protocol checks out.

Problem with Write-Through

High bandwidth requirements
  - Every write from every processor goes to shared bus and memory
  - Consider 200MHz, 1CPI processor, and 15% instrs, are 8-byte stores
  - Each processor generates 30M stores or 240MB data per second
  - 1GB/s bus can support only about 4 processors without saturating
  - Write-through especially unpopular for SMPs
Write-back caches absorb most writes as cache hits
  - Write hits don’t go on bus
  - But now how do we ensure write propagation and serialization?
  - Need more sophisticated protocols: large design space
But first, let’s understand other ordering issues.

Determining Orders More Generally

A memory operation M2 is subsequent to a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.
Read is subsequent to write W if read generates bus transaction that follows that for W.
Write is subsequent to read or write M if M generates bus transaction and the transaction for the write follows that for M.
Write is subsequent to read if read does not generate a bus transaction and is not already separated from the write by another bus transaction.

Limitations of Coherence

---

P1

/*Assume initial value of A and flag is 0*/
A = 1;
while (flag == 0); /*spin idly*/
flag = 1;
print A;

Writes to a location become visible to all in the same order.
But when does a write become visible?
How to establish orders between a write and a read by different processors?
  - Typically use event synchronization, by using more than one location.
  - Intuition not guaranteed by coherence.
Sometimes expect memory to respect order between accesses to different locations issued by a given process.
Would like to preserve orders among accesses to same location by different processes.
Coherence doesn’t help: pertains only to single location,
Another Example of Orders

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Assume initial values of $A$ and $B$ are 0 */</td>
<td></td>
</tr>
<tr>
<td>(1a) $A = 1$;</td>
<td>(2a) print $B$;</td>
</tr>
<tr>
<td>(1b) $B = 2$;</td>
<td>(2b) print $A$;</td>
</tr>
</tbody>
</table>

What’s the intuition?

Whatever it is, we need an ordering model for clear semantics:

- across different locations as well
- so programmers can reason about what results are possible

This is the memory consistency model.

Sequential Consistency

Total order achieved by interleaving accesses from different processes.

Maintains program order, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others.

Programmer’s intuition is maintained:

“A multiprocessor 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, 1979]

Memory Consistency Model

Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another

- What orders are preserved?
- Given a load, constrains the possible values returned by it.

Without it, can’t tell much about an SAP program’s execution.

Implications for both programmer and system designer.

- Programmer uses to reason about correctness and possible results,
- System designer can use to constrain how much accesses can be reordered by compiler or hardware.

Contract between programmer and system.

What Really is Program Order?

Intuitively, order in which operations appear in source code

- Straightforward translation of source code to assembly.
- At most one memory operation per instruction.

But not the same as order presented to hardware by compiler

So which is program order? It depends...

We assume order as seen by programmer.
Sequential Consistency Example

What matters is order in which appears to execute, not executes.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1a) A = 1;</td>
<td>(2a) print B;</td>
</tr>
<tr>
<td>(1b) B = 2;</td>
<td>(2b) print A;</td>
</tr>
</tbody>
</table>

"Assume initial values of A and B are 0"/

- possible outcomes for (A,B): (0,0), (1,0), (1,2); impossible under SC: (0,2)
- we know 1a->1b and 2a->2b by program order
- A = 0 implies 2b->1a, which implies 2a->1a, which implies 2a->1b
- B = 2 implies 1b->2a, which leads to a contradiction

- BUT, actual execution 1b->1a->2b->2a is SC, despite not program order
  - appears just like 1a->1b->2a->2b as visible from results
- actual execution 1b->2a->2b->1a is not SC

Implementing Sequential Consistency

Two kinds of requirements for memory operations:

Must appear to become visible (to local process and others) in program order.

Should appear to complete atomically with respect to all processes in overall total order before next is issued.

- Need to guarantee that total order is consistent across processes.
- Tricky part is making writes atomic.

Write Atomicity

Write Atomicity: Position in total order at which a write appears to perform should be the same for all processes.

- Nothing a process does after it has seen the new value produced by a write W should be visible to other processes until they too have seen W.
- In effect, extends write serialization to writes from multiple processes.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1; while (A==0);</td>
<td>B=1; while (B==0);</td>
<td>print A;</td>
</tr>
</tbody>
</table>

Transitivity implies A should print as 1 under SC.
Problem if P2 leaves loop, writes B, and P3 sees new B but old A (from its cache, say).

More Formally

Each process’s program order imposes partial order on set of all operations.
Interleaving of these partial orders defines a total order on all operations.
Many total orders may be SC (SC does not define particular interleaving).

SC Execution: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving).

SC System: A system is SC if any possible execution on that system is an SC execution.
Sufficient Conditions for Sequential Consistency

Every process issues memory operations in program order.

After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.

After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity).

Sufficient, not necessary, conditions...

Clearly, compilers should not reorder for SC, but they do!

Even if issued in order, hardware may violate for better performance.

Sequential Consistency in Write-through Example

Provides SC, not just coherence.

Extend arguments used for coherence:

- Writes and read misses to all locations serialized by bus into bus order.
- If read obtains value of write W, W guaranteed to have completed,
- When write W is performed w.r.t. any processor, all previous writes in bus order have completed,

Our Treatment of Ordering

Assume for now that compiler does not reorder.

Hardware needs mechanisms to detect:

- Detect write completion (read completion is easy).
- Ensure write atomicity.

For all protocols and implementations, we will see

- How they satisfy coherence, particularly write serialization.
- How they satisfy sufficient conditions for SC (write completion and write atomicity).
- How they can ensure SC but not through sufficient conditions.

Will see that centralized bus interconnect makes it easier.

Summary

Today we saw some challenges to the design of shared memory systems:

- coherence
- sequential consistency
- memory consistency

And we took a look at a simple coherency protocol (write-through).

Next time we will learn more protocols and assess their tradeoffs.