The Cache-Coherence Problem

Intro to Chapter 5

Shared Memory vs. No Shared Memory

- **Advantages of shared-memory machines**
  - Naturally support shared-memory programs
    - Clusters can also support them via *software virtual shared memory*, but with much coarser granularity and higher overheads
  - Single OS image
  - Can fit larger problems: total mem = mem\(_n\) = n \times mem\(_1\)
    - Large programs cause thrashing in clusters
  - Allow fine-grained sharing
    - Messages are expensive, so large messages work, but small messages are dominated by overheads
    - Fine-grained synchronization fits better

- **Disadvantages of shared-memory machines**
  - Cost of providing shared-memory abstraction
**Multiprocessor Usage**

- On-chip: shared memory (+ shared cache)
- Small scale (up to 2–30 processors): shared memory
  - Most processors have MP support out of the box
  - Most of these systems are bus-based
  - Popular in commercial as well as HPC markets
- Medium scale (64–256): shared memory and clusters
  - Clusters are cheaper
  - Often, clusters of SMPs
- Large scale (> 256): few shared memory and many clusters
  - SGI Altix 3300: 512-processor shared memory
  - Large variety on custom/off-the-shelf components such as interconnection networks.
    - Beowulf clusters: fast Ethernet
    - Myrinet: fiber optics
    - IBM SP2: custom

**Goals**

So far, we have learned:
- How to write parallel programs
- How caches and buses work

Let’s now construct a bus-based multiprocessor

Will it work?
The Cache-Coherence Problem

- Illustration:
  - "A joint checking account owned by A, B, and C.
  - Each has a checkbook, and withdraws and deposits funds several times every day.
  - The bank imposes large penalty if a check is issued without sufficient fund at the account.
  - The only mode of communication is by sending the current account balance by mail.
  - A, B, and C work in the same building, but the bank is on a different building. It takes only 1 min. to send mail within the building, but 1 hour to another building."

- Goal 1: A, B, and C need an up-to-date and coherent view of their account balance.
- Goal 2: To achieve goal 1 with minimum number of messages.

At the Minimum

- At the minimum, they need a protocol to support ...
  - Write propagation: passing on the information that the balance has been updated.
- But they also need ...
  - Write serialization: Updates by \(X,Y\) in that order are seen in the same order by everybody
- Reaching goal 2 depends on the access patterns
  - Mostly reads and a few writes?
  - Many successive writes?
Will Parallel Code Work Correctly?

Two issues:
- Will it print sum = 10?
- How can it support locking correctly?

Example 1: The Cache-Coherence Problem

- Will it print sum = 10?
Cache-Coherence Problem Illustration

Cache

P1

P2

P3

Bus

Main Memory

Mem Ctrl

sum=0

Lecture 7

ECE/CSC 506 - Summer 2006 - E. F. Gehringer, based on slides by Yan Solihin
Cache-Coherence Problem Illustration

P1

P2

rd & sum

P3

Mem Ctrl

sum = 0

Mem Ctrl

sum = 0

Cache-Coherence Problem Illustration

P1

P2

wr sum

(sum = 3)

P3

Mem Ctrl

sum = 0

sum = 0

sum = 0
Cache-Coherence Problem Illustration

P1
sum=3
D

P2
sum=7
D

P3

Mem Ctrl
sum=0

wr & sum (sum = 7)

rd & sum
print sum=3

Cache-Coherence Problem Illustration

P1
sum=3
D

P2
sum=7
D

P3

Mem Ctrl
sum=0
Cache-Coherence Problem

- Do P1 and P2 see the same sum?
- Does it matter if we use a WT cache?
- What if we do not have caches, or sum is uncacheable. Will it work?

Write-Through Cache Does Not Work
Software Lock Using a Flag

```c
void lock (int process, int lvar) { // process is 0 or 1
    while (lvar == 1) {} ;
    lvar = 1;
}

void unlock (int process, int lvar) {
    lvar = 0;
}
```

- Will this guarantee mutual exclusion?

Peterson’s Algorithm

```c
int turn;
int interested[n]; // initialized to 0

void lock (int process, int lvar) { // process is 0 or 1
    int other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE) {} ;
} // Post: turn != process or interested[other] == FALSE

void unlock (int process, int lvar) {
    interested[process] = FALSE;
}
```

- Exit from lock() happens only if:
  - `interested[other] == FALSE`: either the other process has not competed for the lock, or it has just called `unlock()`
  - `turn != process`: the other process is competing, has set the turn to itself, and will be blocked in the `while()` loop
Will it Work?

- Correctness depends on the global order of

  ```
  A: interested[process] = TRUE;
  B: turn = process;
  ```

- Thus, it will not work if:
  - Compiler reorders the operations
    - No data dependence, so unless the compiler is notified, it may well reorder the operations
    - This prevents compiler from using aggressive optimizations used in serial programs
  - The architecture reorders the operations
    - Write buffers, memory controller
    - Network delay for statement A
    - If `turn` and `interested[]` are cacheable, A may result in cache miss, but B in cache hit
  - This is called the memory-consistency problem.

No Race

// Proc 0
interested[0] = TRUE;
turn = 0;
while (turn==0 && interested[1]==TRUE)
{
  // since interested[1] == FALSE,
  // Proc 0 enters critical section

  // Proc 1
  interested[1] = TRUE;
turn = 1;
  while (turn==1 && interested[0]==TRUE)
  {
    // since turn==1 && interested[0]==TRUE
    // Proc 1 waits in the loop until Proc 0
    // releases the lock

  // unlock
  Interested[0] = FALSE;
  // now can exit the loop and acquire the
  // lock

  // Proc 0
  interested[0] = TRUE;
turn = 0;
  while (turn==0 && interested[1]==TRUE)
  {
    // since interested[1] == FALSE,
    // Proc 0 enters critical section
  }"
Race

// Proc 0
interested[0] = TRUE;
turn = 0;

while (turn==0 && interested[1]==TRUE) {
} // since turn==1, // Proc 0 enters critical section

unlock
Interested[0] = FALSE;

// now can exit the loop and acquire the lock

Race on a Non-Sequential Consistent Machine

// Proc 0
interested[0] = TRUE;
turn = 0;

while (turn==0 && interested[1]==TRUE) {
} // since interested[1] == FALSE, // Proc 0 enters critical section

// Proc 1
interested[1] = TRUE;
turn = 1;

while (turn==1 && interested[0]==TRUE) {
} // since turn==0, // Proc 1 enters critical section
Two Fundamental Problems

- Cache Coherence problem
  - Tackled in hardware with cache coherence protocols
  - Correctness guaranteed by the protocols, but with varying performance

- Memory Consistency problem
  - Tackled by various memory consistency models, which differ by
    - what operations can be reordered, and what cannot be reordered
    - Guarantee of completeness of a write
  - Compilers and programs have to conform to the model for correctness!
  - 2 approaches:
    - Sequential Consistency:
      - Multi-threaded codes for uniprocessors automatically run correctly
      - How? Every shared rd/wr completes globally in program order
      - Most intuitive but worst performance
    - Others (Relaxed Consistency Models):
      - Multi-threaded codes for uniproc need to be ported to run correctly
      - Additional instruction (memory fence) to ensure global order between 2 operations

Caches and Cache Coherence

- Caches are important
  - Reducing average data access time
  - Reducing bandwidth requirement on bus/interconnect

- So let’s use caches, but solve the coherence problem

- Sufficient conditions for coherence:
  - Notation: Request_{proc}(data)
  - **Write propagation**:
    - Rd_i(X) must return the “latest” Wr_j(X)
  - **Write serialization**:
    - Wr_i(X) and Wr_j(X) are seen in the same order by everybody
      - i.e., if I see w1 after w2, you should not see w2 before w1
      - In essence, global ordering of memory operations to a single location
    - no need for read serialization since it’s local
A Coherent Memory System: Intuition

- **Uniprocessors**
  - Coherence between I/O devices and processors
  - Infrequent, so software solutions work
    - Uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches
- **But coherence problem much more critical in multiprocessors**
  - Pervasive
  - Performance-critical
  - Necessitates a hardware solution

* Note that “latest” is ambiguous. Ultimately, what we care is any write is propagated, as a building block of synchronization mechanism. Synchronization defines what “latest” means

Natural Extensions of a Memory System

(a) Shared cache

(b) Bus-based shared memory

(c) Dancehall

(d) Distributed-memory
Focus: Bus-Based, Centralized Memory

- Shared cache: popular in CMP (Power4, Sparc, Pentium, Itanium)
  - Low-latency sharing and prefetching across processors
  - Sharing of working sets
  - No coherence problem (if sharing L1 cache)
  - But contention and negative interference
    - Thread starvation, priority inversion, thread-mix dependent throughput
  - Higher hit and miss latency due to extra ports and cache size
  - Mid 80s: to connect couple of processors on a board (Encore, Sequent)
  - Today: CMP (e.g., Power4)

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

- Distributed memory
  - Popular way to build large size systems (32-128 procs), discussed later

Assume a Bus-Based SMP

- Built on top of two fundamentals of uniprocessor system
  - Bus transactions
  - Cache-line finite-state machine

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

- Uniprocessor cache states:
  - Every cache line has a finite state machine
  - In WT-write no-allocate: Valid, Invalid states
  - WB: Valid, Invalid, Modified (“Dirty”)

- Multiprocessors extend both these somewhat to implement coherence
Snoop-Based Coherence

- **Basic Idea**
  - Assign a snooper to each processor so that all bus transactions are visible to all processors *(Snooping)*
  - Processors (via cache controllers) change line states on relevant events

- **Implementing a Protocol**
  - Each cache controller reacts to processor and bus events:
    - Takes actions when necessary
      - Updates state, responds with data, generates new bus transactions
    - Memory controller also snoops bus transactions and returns data only when needed
  - Granularity of coherence is typically cache line/block
    - Same granularity as in transfer to/from cache

---

Coherence with Write-Through Caches

```plaintext
sum = 0;
begin parallel
for (i=0; i<2; i++) {
  lock(id, myLock);
  sum = sum + a[i];
  unlock(id, myLock);
} end parallel
Print sum;
```

Suppose $a[0] = 3$ and $a[1] = 7$

- Bus-based SMP implementation choice in the mid 80s
- What happens when we snoop a write? invalidating/update the block
  - no new states or bus transactions in this case
  - write update protocols: write is immediately propagated
  - write invalidation protocols: causes miss on later access, and memory up-to-date via write-through
Snooper Assumption

- Atomic bus
- Writes in program order

Transactions

- Processor transactions:
  - PrRd
  - PrWr
- Snooped bus transactions
  - BusRd
  - BusWr
Write-Through State-Transition Diagram

- Key: Write invalidates all other caches
- Therefore, we have:
  - Modified line: exists as V in only 1 cache
  - Clean line: exists as V in at least 1 cache
  - Invalid state represents invalidated line or not present in the cache

Is It Coherent?

- Write propagation:
  - through invalidation
  - then a cache miss, loading a new value
- Write serialization:
  - Assume invalidation happens instantaneously
  - Assuming atomic bus (all through Chap 5, relaxed on Chap 6)
  - Writes serialized by order in which they appear on bus (bus order)
  - So do invalidations
- Do reads see the latest writes?
  - Read misses generate bus transactions, so will get the last write
  - Read hits: do not appear on bus, but are preceded by
    - most recent write by this processor (self), or
    - most recent read miss by this processor
  - Thus, reads hits see latest written values (according to bus order)
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 xaction that follows that for W.
- Write is subsequent to read or write M if M generates bus xaction and the xaction for the write follows that for M.
- Write is subsequent to read if read does not generate a bus xaction and is not already separated from the write by another bus xaction.

- Writes establish a partial order
- Doesn’t constrain ordering of reads, though bus will order read misses too
  – any order among reads between writes is fine, as long as in program order

Problem with Write-Through

- High bandwidth requirements
  - Every write goes to the shared bus and memory
  - Example:
    - 200MHz, 1 CPI 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
  - Thus, unpopular for SMPs
- Write-back caches
  - Write hits do not go to the bus => reduce most write bus transactions
  - But now how do we ensure write propagation and serialization?
Summary

- Shared memory with caches raises the problem of cache coherence.
  - Writes to the same location must be seen in the same order everywhere.
- But this is not the only problem
  - Writes to different locations must also be kept in order if they are being depended upon for synchronizing tasks.
  - This is called the memory-consistency problem
- One solution for small-scale multiprocessors is a shared bus.
- State-transition diagrams can be used to show how a cache-coherence protocol operates.
- The simplest protocol is write-through, but it has performance problems.