EE 660: Computer Architecture
Parallel Programming
and Small Multiprocessors
Yao Zheng
Department of Electrical Engineering
University of Hawai‘i at Mānoa

Based on the slides of Prof. David Wentzlaff
Trends in Computation

Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and D. We^ntzlaff
Trends in Computation

Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and D. We3ntzlaff
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (even in a uniprocessor system)

**Producer-Consumer**: A consumer process must wait until the producer process has produced data

**Mutual Exclusion**: Ensure that only one process uses a resource at a given time
A Producer-Consumer Example

Producer posting Item x:
- Load $R_{tail}$, (tail)
- Store x, ($R_{tail}$)
- $R_{tail} = R_{tail} + 1$
- Store $R_{tail}$, (tail)

Consumer:
- Load $R_{head}$, (head)
- spin:
  - Load $R_{tail}$, (tail)
  - if $R_{head} == R_{tail}$ goto spin
  - Load R, ($R_{head}$)
  - $R_{head} = R_{head} + 1$
  - Store $R_{head}$, (head)
- process(R)

The program is written assuming instructions are executed in order.

Problems?
A Producer-Consumer Example

continued

Producer posting Item x:
1. Load $R_{tail}$, (tail)
2. Store $x$, $(R_{tail})$
   \[ R_{tail} = R_{tail} + 1 \]

Consumer:
1. Load $R_{head}$, (head)
2. Load $R_{tail}$, (tail)
3. if $R_{head} == R_{tail}$ goto spin
4. Load $R$, $(R_{head})$
   \[ R_{head} = R_{head} + 1 \]
   Store $R_{head}$, (head)
   process(R)

Can the tail pointer get updated before the item $x$ is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:
1. 2, 3, 4, 1
2. 4, 1, 2, 3
"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 the order specified by the program"

*Leslie Lamport*

Sequential Consistency =

arbitrary *order-preserving interleaving* of memory references of sequential programs
Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
Store 1, (X) (X = 1)
Store 11, (Y) (Y = 11)

T2:
Load R₁, (Y)
Store R₁, (Y') (Y' = Y)
Load R₂, (X)
Store R₂, (X') (X' = X)

what are the legitimate answers for X' and Y'?

(X', Y') ∈ {(1,11), (0,10), (1,10), (0,11)}

If Y is 11 then X cannot be 0
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

What are these in our example?

T1:
- Store 1, (X) (X = 1)
- Store 11, (Y) (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y’), R₁ (Y’ = Y)
- Load R₂, (X)
- Store (X’), R₂ (X’ = X)

additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory?
Multiple Consumer Example

Producer posting Item x:
- Load $R_{\text{tail}}$, (tail)
- Store x, (R\_tail)
- $R_{\text{tail}} = R_{\text{tail}} + 1$
- Store $R_{\text{tail}}$, (tail)

Critical section:
Needs to be executed atomically by one consumer $\Rightarrow$ locks

Consumer:
- Load $R_{\text{head}}$, (head)
- spin:
  - Load $R_{\text{tail}}$, (tail)
  - if $R_{\text{head}} == R_{\text{tail}}$ goto spin
  - Load R, (R\_head)
  - $R_{\text{head}} = R_{\text{head}} + 1$
  - Store $R_{\text{head}}$, (head)
  - process(R)

What is wrong with this code?
A semaphore is a non-negative integer, with the following operations:

**P(s):** *if s > 0, decrement s by 1, otherwise wait*

*probeer te verlagen*, literally ("try to reduce")

**V(s):** *increment s by 1 and wake up one of the waiting processes*

*verhogen* ("increase")

P’s and V’s must be executed atomically, i.e., without

- *interruptions* or
- *interleaved accesses to s* by other processors

---

**Process i**

- **P(s)**
  - <critical section>
- **V(s)**

*initial value of s determines the maximum no. of processes in the critical section*
Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:  

*atomic read-modify-write instructions*

Examples: \( m \) is a memory location, \( R \) is a register

Test&Set (m), R:
\[
\begin{align*}
R & \leftarrow M[m]; \\
if \ R == 0 \ then \\
M[m] & \leftarrow 1;
\end{align*}
\]

Fetch&Add (m), \( R_v, R \):
\[
\begin{align*}
R & \leftarrow M[m]; \\
M[m] & \leftarrow R + R_v;
\end{align*}
\]

Swap (m), R:
\[
\begin{align*}
R_t & \leftarrow M[m]; \\
M[m] & \leftarrow R_t; \\
R & \leftarrow R_t;
\end{align*}
\]
Multiple Consumers Example

*using the Test&Set Instruction*

```
P: Test&Set (mutex), R_{temp}
    if (R_{temp} != 0) goto P

spin:
    Load R_{head}, (head)
    Load R_{tail}, (tail)
    if R_{head} == R_{tail} goto spin
    Load R, (R_{head})
    R_{head} = R_{head} + 1
    Store R_{head}, (head)

V: Store 0, (mutex)
    process(R)
```

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P’s and V’s

*What if the process stops or is swapped out while in the critical section?*
Nonblocking Synchronization

Compare&Swap(m), R_t, R_s:
  if (R_t==M[m])
    then M[m]=R_s;
    R_s=R_t;
    status ← success;
  else status ← fail;

status is an implicit argument

try: Load R_{head}, (head)
spin: Load R_{tail}, (tail)
      if R_{head}==R_{tail} goto spin
      Load R, (R_{head})
      R_{newhead} = R_{head}+1
      Compare&Swap(head), R_{head}, R_{newhead}
      if (status==fail) goto try
      process(R)
Load-link & Store-conditional
aka Load-reserve, Load-Locked

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-link R, (m):  
  <flag, adr> ← <1, m>;  
  R ← M[m];

Store-conditional (m), R:  
  if <flag, adr> == <1, m>  
    then cancel other procs' reservation on m;  
    M[m] ← R;  
    status ← succeed;  
  else status ← fail;

try:  
spin:  
  Load-link R_{head}, (head)  
  Load R_{tail}, (tail)  
  if R_{head} == R_{tail} goto spin  
  Load R, (R_{head})  
  R_{head} = R_{head} + 1  
  Store-conditional R_{head}, (head)  
  if (status == fail) goto try  
  process(R)
Performance of Locks

Blocking atomic read-modify-write instructions
  e.g., Test&Set, Fetch&Add, Swap
vs
Non-blocking atomic read-modify-write instructions
  e.g., Compare&Swap,
  Load-link/Store-conditional
vs
Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
  degree of contention,
caches,
out-of-order execution of Loads and Stores
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

• *Out-of-order execution capability*
  
  Load(a); Load(b)  
  Load(a); Store(b)  
  Store(a); Load(b)  
  Store(a); Store(b) 
  
  *yes*  
  *yes if a ≠ b*  
  *yes if a ≠ b*  
  *yes if a ≠ b*

• *Caches*
  
  Caches can prevent the effect of a store from being seen by other processors

SC complications motivate architects to consider *weak or relaxed* memory models
Memory Fences

Instructions to sequentialize memory accesses

Processors with relaxed or weak memory models permit Loads and Stores to different addresses to be reordered, remove some/all extra dependencies imposed by SC

- LL, LS, SL, SS

Need to provide memory fence instructions to force the serialization of memory accesses

Examples of relaxed memory models:

- Total Store Order: LL, LS, SS, enforce SL with fence
- Partial Store Order: LL, LS, enforce SL, SS with fences
- Weak Ordering: enforce LL, LS, SL, SS with fences

Memory fences are expensive operations – mem instructions wait for all relevant instructions in-flight to complete (including stores to retire – need store acks)

However, cost of serialization only when it is required!
Using Memory Fences

Producer posting Item x:
- Load $R_{tail}$, (tail)
- Store x, ($R_{tail}$)
- MFence_{SS}
  - $R_{tail} = R_{tail} + 1$
  - Store $R_{tail}$, (tail)

ensures that tail ptr is not updated before x has been stored

Consumer:
- Load $R_{head}$, (head)
- spin:
  - Load $R_{tail}$, (tail)
  - if $R_{head} == R_{tail}$ goto spin
  - MFence_{LL}
  - Load R, ($R_{head}$)
  - $R_{head} = R_{head} + 1$
  - Store $R_{head}$, (head)
  - process(R)

ensures that R is not loaded before equality check
Mutual Exclusion Using Load/Store

A protocol based on two shared variables $c_1$ and $c_2$. Initially, both $c_1$ and $c_2$ are 0 (*not busy*)

**Process 1**

...  
c1=1;  
L: \( \text{if } c_2 == 1 \text{ then go to L} \)  
< critical section>  
c1=0;

**Process 2**

...  
c2=1;  
L: \( \text{if } c_1 == 1 \text{ then go to L} \)  
< critical section>  
c2=0;

What is wrong? **Deadlock!**
Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets $c_1$ to 0) while waiting.

- **Process 1**
  ```plaintext
  ... 
  L: c1=1;  
  if c2==1 then  
    { c1=0; go to L}  
  < critical section>  
  c1=0
  ```

- **Process 2**
  ```plaintext
  ... 
  L: c2=1;  
  if c1==1 then  
    { c2=0; go to L}  
  < critical section>  
  c2=0
  ```

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section ⇒ *starvation*
A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (not busy)

Process 1

...  
c1=1;
turn = 1;
L: if c2==1 && turn==1  
then go to L  
< critical section>
c1=0;

Process 2

...  
c2=1;
turn = 2;
L: if c1==1 && turn==2  
then go to L  
< critical section>
c2=0;

- turn == i ensures that only process i can wait
- variables c1 and c2 ensure mutual exclusion

Solution for n processes was given by Dijkstra and is quite tricky!
N-process Mutual Exclusion

Lamport’s Bakery Algorithm

Process $i$

Entry Code

Initially $num[j] = 0$, for all $j$

```
choosing[i] = 1;
num[i] = max(num[0], ..., num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++) {
    while( choosing[j] );
    while( num[j] &&
        ( ( num[j] < num[i] ) ||
          ( num[j] == num[i] && j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)
Multidrop Memory Bus
Pipelined Memory Bus

Arbitration
Control
Address
Data
Clock

Processor 1
Processor 2
Main

LD
0x1234abcd
0xDA7E0000
Memory Coherence in SMPs

Suppose CPU-1 updates A to 200.
- **write-back**: memory and cache-2 have stale values
- **write-through**: cache-2 has a stale value

Do these stale values matter?
What is the view of shared memory for programming?
Write-back Caches & SC

• **T1 is executed**
  - prog T1
    - ST 1, X
    - ST 11, Y
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 10
    - X’ =
    - Y’ =
  - cache-2
    - Y =
    - Y’ =
    - X =
    - X’ =

• **cache-1 writes back Y**
  - prog T2
    - LD Y, R1
    - ST R1, Y’
    - LD X, R2
    - ST R2, X’

• **T2 executed**
  - cache-1
    - X = 1
    - Y = 11
    - X’ =
    - Y’ =
  - memory
    - X = 0
    - Y = 11
    - X’ =
    - Y’ =
  - cache-2
    - Y = 11
    - Y’ = 11
    - X = 0
    - X’ = 0

• **cache-1 writes back X**
  - prog T2
    - LD Y, R1
    - ST R1, Y’
    - LD X, R2
    - ST R2, X’

• **cache-2 writes back X’ & Y’**
  - prog T2
    - LD Y, R1
    - ST R1, Y’
    - LD X, R2
    - ST R2, X’

*inconsistent*
Write-through Caches & SC

- T1 executed
  - prog T1
    - ST 1, x
    - ST 11, Y
    - cache-1
      - X = 0
      - Y = 10
    - memory
      - X = 0
      - Y = 10
      - X' =
      - Y' =
    - cache-2
      - Y =
      - Y' =
      - X = 0
      - X' =
  - prog T1
    - ST 1, x
    - ST 11, Y
    - cache-1
      - X = 1
      - Y = 11
    - memory
      - X = 1
      - Y = 11
      - X' =
      - Y' =
    - cache-2
      - Y =
      - Y' =
      - X = 0
      - X' =

- T2 executed
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 1
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - Y' = 11
    - X = 0
    - X' = 0

Write-through caches don’t preserve sequential consistency either
Cache Coherence vs. Memory Consistency

• A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  – i.e., updates are not lost
• A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  – Equivalently, what values can be seen by a load
• A cache coherence protocol is not enough to ensure sequential consistency
  – But if sequentially consistent, then caches must be coherent
• Combination of cache coherence protocol plus processor memory reorder buffer implements a given machine’s memory consistency model
Warmup: Parallel I/O

Either Cache or DMA can be the Bus Master and effect transfers

(DMA stands for “Direct Memory Access”, means the I/O device can read/write memory autonomous from the CPU)
Problems with Parallel I/O

- **Memory → Disk:** Physical memory may be stale if cache copy is dirty.
- **Disk → Memory:** Cache may hold stale data and not see memory writes.

Diagrams show the relationship between PROC, Cache, Memory Bus, Physical Memory, DMA, and DISK in the context of parallel I/O operations.
Snoopy Cache *Goodman & Ravishankar 1983*

- Idea: Have cache watch (or snoop upon) DMA transfers, and then “do the right thing”
- Snoopy cache tags are dual-ported
Use snoopy mechanism to keep all processors’ view of memory coherent
Update (Broadcast) vs. Invalidate

Snoopy Cache Coherence Protocols

• Write Update (Broadcast)
  – Writes are broadcast and update all other cache copies

• Write Invalidate
  – Writes invalidate all other cache copies
Write Update (Broadcast) Protocols

**write miss:**
Broadcast on bus, other processors update copies (in place)

**read miss:**
Memory is always up to date
Write Invalidate Protocols

write miss:
the address is invalidated in all other caches before the write is performed

read miss:
if a dirty copy is found in some cache, a write-back is performed before the memory is read
Each cache line has state bits:

- **M**: Modified
- **S**: Shared
- **I**: Invalid

```
State bits

<table>
<thead>
<tr>
<th>Address tag</th>
</tr>
</thead>
</table>
```

- **Write miss** (P1 gets line from memory)
- **Other processor reads** (P1 writes back)
- **Read miss** (P1 gets line from memory)
- **Read by any processor**
- **Other processor intent to write**
- **P1 intent to write**

Cache state in processor P1:

- **P1 reads or writes**
- **Other processor intent to write** (P1 writes back)
Two Processor Example
(Reading and writing the same cache line)

P1 reads
P1 writes
P2 reads
P2 writes
P1 reads
P1 writes
P2 writes
P1 writes

P1

P2 reads,
P1 writes back

P2 reads,        P1 reads or writes
P2 writes back   Write miss

P1 intent to write

P1 reads

P1 writes

P2 reads

P2 writes

P2 reads

P2 writes

P1 reads

P1 writes

P2 writes

P1 writes

P2 writes

P1 writes

P2 reads

P2 writes

P1 reads or writes

Write miss

P1 intent to write

P2 reads

P2 writes

P1 reads or writes

Write miss

P1 intent to write

P2 reads

P2 writes

P1 reads or writes

Write miss

P1 intent to write
Observation

- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist
MESI: An Enhanced MSI protocol

increased performance for private data (Illinois Protocol)

Each cache line has a tag

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Modified Exclusive</td>
</tr>
<tr>
<td>E</td>
<td>Exclusive but unmodified</td>
</tr>
<tr>
<td>S</td>
<td>Shared</td>
</tr>
<tr>
<td>I</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Address tag

- Write miss
- P₀ write or read
- Other processor reads
- P₀ writes back
- Read miss, shared
- Read by any processor
- Other processor intent to write
- Other processorwrites back
- Other processor intent to write, P₁ writes back

Cache state in processor P₁
Each cache line has a tag

- **M**: Modified Exclusive
- **O**: Owned
- **E**: Exclusive but unmodified
- **S**: Shared
- **I**: Invalid

Address tag

- **Write miss**: P₁ write or read
- **Other processor reads**: P₁ tracks write back
- **Read miss, shared**: Other processor intent to write
- **Read by any processor**: P₁ writes back
- **P₁ intent to write**: Other processor reads
- **Other processor intent to write**: P₁ writes back
Each cache line has a tag

- **M**: Modified Exclusive
- **E**: Exclusive but unmodified
- **S**: Shared
- **I**: Invalid
- **F**: Forward

Address tag

- State bits
- Write miss
- P₁ write or read
- Other processor reads
- P₁ writes back
- Read miss, shared
- Read by any processor

Cache state in processor P₁

45
Scalability Limitations of Snooping

• Caches
  – Bandwidth into caches
  – Tags need to be dual ported or steal cycles for snoops
  – Need to invalidate all the way to L1 cache

• Bus
  – Bandwidth
  – Occupancy (As number of cores grows, atomically utilizing bus becomes a challenge)
False Sharing

| state | blk addr | data0 | data1 | ... | dataN |

A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose $M_1$ writes $\text{word}_i$ and $M_2$ writes $\text{word}_k$ and both words have the same block address.

What can happen?
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)
  – Christopher Batten (Cornell)

• MIT material derived from course 6.823
• UCB material derived from course CS252 & CS152
• Cornell material derived from course ECE 4750
## Blackboard Example: Sequential Consistency

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>Valid</th>
<th>Not Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td>1 1 5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2 2 6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5 3 7</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3 4 1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>6 5 2</td>
<td>4</td>
</tr>
</tbody>
</table>

7 6 3 6
8 7 4 7
4 8 8 8
Analysis of Dekker’s Algorithm

Scenario 1

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
     then go to L
     < critical section>
  c1=0;

Scenario 2

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
     then go to L
     < critical section>
  c1=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
      then go to L
      < critical section>
  c2=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
      then go to L
      < critical section>
  c2=0;
Copyright © 2013 David Wentzlaff