Chapter 5: Multiprocessors (Thread-Level Parallelism) - Part 2

Introduction
  - What is a parallel or multiprocessor system?
  - Why parallel architecture?
  - Performance potential
  - Flynn classification

Communication models
Architectures
Centralized shared memory
Distributed shared memory
Parallel programming
Synchronization
Memory consistency models

Memory Consistency Model - Motivation

Example shared-memory program

Initially all locations = 0

Processor 1     Processor 2
Data = 23       while (Flag != 1) {;}
Flag = 1        ... = Data

Execution (only shared-memory operations)

Processor 1     Processor 2
Write, Data, 23 Write, Flag, 1
Write, Flag, 1 Read, Flag, 1
Read, Data, ___

Memory Consistency Model: Definition

Memory consistency model
Order in which memory operations will appear to execute
  ⇒ What value can a read return?
Affects ease-of-programming and performance

The Uniprocessor Model

Program text defines total order = program order
Uniprocessor model
  Memory operations appear to execute one-at-a-time in program order
  ⇒ Read returns value of last write
BUT uniprocessor hardware
  Overlap, reorder operations
Model maintained as long as
  maintain control and data dependences
⇒ Easy to use + high performance
Implicit Memory Model

Sequential consistency (SC) [Lamport]
Result of an execution appears as if
- All operations executed in some sequential order (i.e., atomically)
- Memory operations of each process in program order

Understanding Program Order – Example 1
Initially Flag1 = Flag2 = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag1 = 1</td>
<td>Flag2 = 1</td>
</tr>
<tr>
<td>if (Flag2 == 0)</td>
<td>if (Flag1 == 0)</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
</tbody>
</table>

Execution:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write, Flag1, 1</td>
<td>Write, Flag2, 1</td>
</tr>
<tr>
<td>Read, Flag2, 0</td>
<td>Read, Flag1, ___</td>
</tr>
</tbody>
</table>

Can happen if
- Write buffers with read bypassing
- Overlap, reorder write followed by read in h/w or compiler
- Allocate Flag1 or Flag2 in registers

Understanding Program Order - Example 2
Initially A = Flag = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 23;</td>
<td>P2</td>
</tr>
<tr>
<td>while (Flag != 1) {;}</td>
<td>while (Flag != 1) {;}</td>
</tr>
<tr>
<td>... = A;</td>
<td>... = A;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write, A, 23</td>
<td>Write, Flag, 1</td>
</tr>
<tr>
<td>Write, Flag, 1</td>
<td>Read, Flag, 0</td>
</tr>
<tr>
<td>Read, Flag, 1</td>
<td>Read, A, ___</td>
</tr>
</tbody>
</table>
Understanding Program Order: Summary

SC limits program order relaxation:
- Write → Read
- Write → Write
- Read → Read, Write

Understanding Atomicity

Cache Coherence Protocols

How to propagate write?
- Invalidate -- Remove old copies from other caches
- Update -- Update old copies in other caches to new values

Understanding Atomicity - Example 1

Initially A = B = C = 0

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
</tr>
<tr>
<td>A = 1;</td>
<td>A = 2;</td>
<td>while (B != 1) {}</td>
<td>while (B != 1) {}</td>
</tr>
<tr>
<td>B = 1;</td>
<td>C = 1;</td>
<td>while (C != 1) {}</td>
<td>while (C != 1) {}</td>
</tr>
<tr>
<td>tmp1 = A;</td>
<td>tmp2 = A;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Understanding Atomicity - Example 1

<table>
<thead>
<tr>
<th>Initially A = B = C = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
</tr>
<tr>
<td>A = 1;</td>
</tr>
<tr>
<td>B = 1;</td>
</tr>
</tbody>
</table>

Can happen if updates of A reach P3 and P4 in different order

Coherence protocol must serialize writes to same location

(Writes to same location should be seen in same order by all)

### Understanding Atomicity - Example 2

<table>
<thead>
<tr>
<th>Initially A = B = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
</tr>
<tr>
<td>A = 1</td>
</tr>
<tr>
<td>B = 1;</td>
</tr>
</tbody>
</table>

Can happen if read returns new value before all copies see it

### SC Summary

**SC limits**

- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- When a processor can read the value of a write
- Unserialized writes to the same location

**Alternative**

1. Aggressive hardware techniques proposed to get SC w/o penalty using speculation and prefetching
   - But compilers still limited by SC
2. Give up sequential consistency
   - Use relaxed models

### Classification for Relaxed Models

Typically described as system optimizations - system-centric

**Optimizations**

- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read, Write
  - Read others’ write early
  - Read own write early

All models provide safety net

- All models maintain uniprocessor data and control dependences, write serialization
### Some System-Centric Models

<table>
<thead>
<tr>
<th>Relaxation:</th>
<th>W→R Order</th>
<th>W→W Order</th>
<th>R→R/W Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 370</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PSO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCsc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCpc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RMO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PowerPC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### System-Centric Models: Assessment

System-centric models provide higher performance than SC

BUT 3P criteria

- **Programmability?**
  - Lost intuitive interface of SC

- **Portability?**
  - Many different models

- **Performance?**
  - Can we do better?

Need a higher level of abstraction

### An Alternate Programmer-Centric View

One source of consensus
- Programmers need SC to reason about programs
But SC not practical today
  - How about the next best thing…

### A Programmer-Centric View

Specify memory model as a contract
- System gives sequential consistency
- IF programmer obeys certain rules

+ Programmability
+ Performance
+ Portability
The Data-Race-Free-0 Model: Motivation

Different operations have different semantics

\[
\begin{align*}
P_1 & \quad P_2 \\
\text{A} & = 23; \quad \text{while} \ (\text{Flag} \neq 1) \ \{} \\
\text{B} & = 37; \quad \ldots = B; \\
\text{Flag} & = 1; \\
\end{align*}
\]

\(\text{Flag} = \) Synchronization; \(\text{A, B} = \) Data

Can reorder data operations

Distinguish data and synchronization

Need to

- Characterize data / synchronization
- Prove characterization allows optimizations w/o violating SC

Data-Race-Free-0: Some Definitions

Two operations conflict if

- Access same location
- At least one is a write

Data-Race-Free-0: Some Definitions (Cont.)

(Consider SC executions \(\Rightarrow\) global total order)

Two conflicting operations race if

- From different processors
- Execute one after another (consecutively)

\[
\begin{align*}
P_1 & \quad P_2 \\
\text{Write, A, 23} & \quad \text{Read, Flag, 0} \\
\text{Write, B, 37} & \quad \text{Read, B, } \ldots \\
\text{Write, Flag, 1} & \quad \text{Read, Flag, 1} \\
\end{align*}
\]

Races usually "synchronization," others "data"

Can optimize operations that never race

Data-Race-Free-0 (DRF0) Definition

Data-Race-Free-0 Program

All accesses distinguished as either synchronization or data

All races distinguished as synchronization

(in any SC execution)

Data-Race-Free-0 Model

Guarantees SC to data-race-free-0 programs

It is widely accepted that data races make programs hard to debug independent of memory model (even with SC)
### Distinguishing/Labeling Memory Operations

Need to distinguish/label operations at all levels
- High-level language
- Hardware
  
  Compiler must translate language label to hardware label

Java: volatiles, synchronized
C++: atomics
Hardware: fences inserted before/after synchronization

### Data-Race-Free Summary

The idea
- Programmer writes data-race-free programs
- System gives SC

For programmer
- Reason with SC
- Enhanced portability

For hardware and compiler
- More flexibility

Finally, convergence on hardware and software sides
  (BUT still many problems…)