Contents

- Overview
- Uniprocessor Review
- Sequential Consistency
- Relaxed Memory Models
- Program Abstractions
- Conclusions
Overview

- Correct & Efficient Shmem Programs
  - Require precise notion of behavior w.r.t. read (R) and write (W) operations between processor memories.

Example 1, Figure 1
Initial, all ptrs = NULL; all ints = 0;

P₁
While (no more tasks) {
  Task = GetFromFreeList();
  Task->Data = ...;
  insert Task in task queue
}
Head = head of task queue;

P₂, P₃, ..., Pₙ
While (MyTask == null) {
  Begin Critical Section
  if (Head != null) {
    MyTask = Head;
    Head = Head->Next;
  }
  End Critical Section
}
... = MyTask->Data;

Q: What will Data be?  A: Could be old Data
Definitions

- **Memory Consistency Model**
  - Formal Specification of Mem System Behavior to Programmer

- **Program Order**
  - The order in which memory operations appear in program

- **Sequential Consistency** (SC): An MP is SC if
  - Exec Result is same as if all Procs were in some sequence.
  - Operations of each Proc appear in this sequence in order specified by its program. (Lamport [16])

- **Relaxed Memory Consistency Models** (RxM)
  - An RxM less restrictive than SC. Valuable for efficient shmem.

- **System Centric**: HW/SW mechanism enabling Mem Model

- **Programmer-centric**: Observation of Program behavior a memory model from programmer’s viewpoint.

- **Cache-Coherence**:
  1. A write is eventually made visible to all MPs.
  2. Writes to same loc appear as **serialized (same order)** by MPs

  **NOTE**: not equivalent to **Sequential Consistency (SC)**
UniProcessor Review

- Only needs to maintain control and data dependencies.
  - Compiler can perform extreme Optz: (reg alloc, code motion, value propagation, loop transformations, vectorizing, SW pipelining, prefetching, ...)
- A multi-threaded program will look like:

```
T1 ----> T2 ----> T3 ----> T4 ----> ... ----> Tn
```

All of memory will appear to have the same values to the threads in a UniProcessor System. You still have to deal with the normal multi-threaded problems by one processor, but you don’t have to deal with issues such as Write Buffer problems or Cache Coherence.

Conceptually, SC wants the one program memory w/ switch that connects procs to memory + Program Order on a per-Processor basis.
**Seq. Consist. Examples**

**Dekker’s Algorithm:**
What if Flag1 set to 1 then Flag2 set to 1 then **ifs**? Or F2 Read bypasses F1 Write?

A: **Sequential Consistency** (program order & Proc seq)

What if P2 gets Read of A but P3 gets old value of A?

A: **Atomicity of memops** (All procs see instant and identical view of memops.)

**NOTE:** UniProcessor system doesn’t have to deal with old values or R/W bypasses.
Architectures

Will visit:

Architectures w/o Cache
- Write Buffers w/ Bypass Capability
- Overlapping Write Operations
- Non-Blocking Read Operations

Architectures w/ Cache
- Cache Coherence & SC
- Detecting Completion of Write Operations
- Illusion of Write Atomicity
Write Buffer w/ Bypass Capability

Bus-based Mem System w/o Cache

• Bypass can hide Write latency
• Violates Sequential Consistency

Q: What happens if Read of Flag1 & Flag2 bypass Writes?

A: Both enter critical section

P1 // init: all = 0
Flag1 = 1
If (Flag2 == 0) critical section

P2 // init: all = 0
Flag2 = 1
If (Flag1 == 0) critical section

NOTE: Write Buffer not a problem on UniProcessor Programs
Overlapping Writes

- Interconnection network alleviates the serialization bottleneck of a bus-based design. Also, Writes can be coalesced.

Q: What happens if Write of Head bypasses Write of Data?

A: Data Read returns 0

P1 // init: all = 0
Data = 2000
Head = 1

P2 // init: all = 0
While (Head == 0);
... = Data
Non-Blocking Reads

Non-Blocking Reads Enable
- non-blocking caches
- speculative execution
- dynamic scheduling

Q: What happens if Read of Data bypasses Read of Head?

A: Data Read returns 0

P₁ // init: all = 0
Data = 2000
Head = 1

P₂ // init: all = 0
While (Head == 0) ;
... = Data
Cache-Coherence & SC

- Write buffer w/o cache similar to Write-thru cache
  - Reads can proceed before Write completes (on other MPs)

- Cache-Coherence: not equiv to Sequential Consistency (SC)
  1. A write is eventually made visible to all MPs.
  2. Writes to same loc appear as serialized (same order) by MPs
  3. Propagate value via invalidating or updating cache-copy(ies)

- Detecting Completion of Write Operation
  - What if P2 gets new Head but old Data?
  - Avoided if invalidate/update before 2\textsuperscript{nd} Write
  - Write ACK needed
    - Or at least Invalidate ACK
Illusion of Write Atomicity

Cache-coherence Problems:
1. Cache-coherence (cc) Protocol must propagate value to all copies.
2. Detecting Write completion takes multi ops w/ multiple replications.
3. Hard to create “Illusion of Atomicity” w/ non-atomic writes.

Q: What if P1 & P2 updates reach P3 & P4 differently?

A: Reg1 & Reg2 might have different results (& violates SC)
Solution: Can serialize writes to same location
Alternative: Delay updates until ACK of previous to same loc
Still not equiv to Sequential Consistency.

<table>
<thead>
<tr>
<th>P1: A=B=C=0</th>
<th>P2 = 0</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>A = 2</td>
<td>While (B != 1) ; While (C != 1) ; Reg1 = A</td>
<td></td>
</tr>
<tr>
<td>B = 1</td>
<td>C = 1</td>
<td>While (B != 1) ; While (C != 1) ; Reg2 = A</td>
<td></td>
</tr>
</tbody>
</table>
Ex2: Illusion of Wr Atomicity

Q: What if $P_2$ reads new $A$ before $P_3$ gets updated w/ $A$; AND $P_2$ update of $B$ reaches $P_3$ before its update of $A$ AND $P_3$ reads new $B$ & old $A$?

A: Prohibit read from new value until all have ACK’ed.

Update Protocol (2-phase scheme):
1. Send update,Recv ACK from each MP
2. Updated MPs get ACK of all ACKs.
   (Note: Writing proc can consider Write complete after #1.)
Compilers do many optz w.r.t. mem reorderings:
- CSE, Code motion, reg alloc, SW pipe, vect temps, const prop,…
- All done from uni-processor perspective. Violates shmem SC
- e.g. Would never exit from many of our while loops.

Compiler needs to know shmem objects and/or Sync points or must forego many optz.
SC imposes many HW and Compiler constraints

Requirements:
1. Complete of all mem ops before next (or Illusion thereof)
2. Writes to same loc need be serialized (cache-based).
3. Write Atomicity (or illusion thereof)

Discuss HW Techniques useful for SC & Efficiency:
- Pre-Exclusive Rd (Delays due to Program Order); cc invalid mems
- Read Rolebacks (Due to speculative exec or dyn sched).
- Global shmem data dep analysis (Shasha & Snir)

Relaxed Memory Models (RxM) next
Relaxed Memory Models

- Characterization (3 models, 5 specific types)

1. Relaxation
   - 1a. Relax Write to Read program order (PO)
   - 1b. Relax Write to Write PO
   - 1c. Relax Read to Read & Read to Write POs

2. Read others’ Write early
   (assume different locations)
   (cache-based only)

3. Read own Write early
   (most allow & usually safe; but what if two writers to same loc?)

- Some RxMs can be detected by programmer, others not.
- Various Systems use different fence techniques to provide safety nets.
  - AlphaServer, Cray T3D, SparcCenter, Sequent, IBM 370, PowerPC
**Relaxed Write to Read PO**

- Relax constraint of **Write** then **Read** to a diff loc.
  - Reorder Reads w.r.t. previous Writes w/ **memory disambiguation**.
  - 3 Models handle it differently. All do it to **hide Write Latency**
    - Only IBM 370 provides **serialization instr** as safety net between W&R
    - TSO can use Read-Modify-Write (RMW) of either Read or Write
    - PC must use RMW of Read since it uses less stringent RMW requirements.

---

### Example 1

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>F₁ = 1</td>
<td>F₂ = 1</td>
</tr>
<tr>
<td>A = 1</td>
<td>A = 2</td>
</tr>
<tr>
<td>Rg₁ = A</td>
<td>Rg₃ = A</td>
</tr>
<tr>
<td>Rg₂ = F₂</td>
<td>Rg₄ = F₁</td>
</tr>
</tbody>
</table>

**Rslt:** Rg₁ = 1, Rg₃ = 2
Rg₂ = Rg₄ = 0

- TSO & PC since they allow Read of F₁/F₂ before Write of F₁/F₂ on each proc

### Example 2

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- if (A == 1)
  - B = 1
- if (B == 1)
  - Rg₁ = A

**Rslt:** Rg₁ = 0, B = 1

- IBM 370 since it allows P₂ Read of new A while P₃ Reads old A
Relaxing **W2R** and **W2W**

- **SPARC Partial Store Order (PSO)**
  - Writes to diff locs from same proc can be pipelined or overlapped and are allowed to reach memory or other caches out of program order.
  - PSO == TSO w.r.t. letting itself read own write early and prohibiting others from reading value of another proc’s write before ACK from all.

**W2R:** Decker’s Algorithm – PSO will still allow non-SC rslts

```
P1 // init: all = 0
Flag1 = 1
If (Flag2 == 0)
  critical section
P2 // init: all = 0
Flag2 = 1
If (Flag1 == 0)
  critical section
```

**W2W:** PSO Safety net is to provide STBAR (store barrier)

```
P1 // init: all = 0
Data = 2000
STBAR   // Write Barrier
Head = 1
```

```
P2 // init: all = 0
While (Head == 0) ;
... = Data
```
Relaxing All Program Order

- R or W may be reordered w.r.t. R or W to diff location
  - Can hide latency of Reads in the context of either static or dynamic (out-of-order) scheduling processors (can use spec exec & non-blocking caches).

- Alpha, SPARC V9 RMO, PowerPC
  - SPARC & PPC allow reorder of reads to same location.

- Violate SC for previous codes (let’s get dangerous!)

- All allow a proc to read own write early but:
  - RCpc and PPC allow a read to get value of other MP Wr early (complex)

Two categories of Parallel program semantics:
1. Distinguish MemOps based on type (WO, RCsc, RCpc)
2. Provide explicit fence capabilities (Alpha, RMO, PPC)
Weak Ordering (WO)

- Two MemOp Categories: 1) Data Ops   2) Sync Ops

- Program order enforced between these
  - Programmer must ID Sync Op (safety net) – counter utilized (inc/dec)
  - Data regions between Sync Ops can be reordered/optimized.
  - Writes appear atomic to programmer.
Release Consistency (RCsc/RCpc)

• **RCsc**: Maintains SC among special ops
  • Constraints: acquire → all; all → release; special → special

• **RCpc**: Maintains Processor Consistency among special ops; W → R program order among special ops eliminated.
  • Constraints: acquire → all; all → release; special → special
    • except special WR → special RD

• RMW used to maintain Program Order
  • subtle details
Alpha, RMO, PPC

- **Alpha**: fences provided
  - **MB** – Memory Barrier
  - **WMB** - Write Memory Barrier
  - Write atomicity fence not needed.

- **SPARC V9 RMO**: **MEMBAR** fence
  - bits used for any of $R \rightarrow W; W \rightarrow W; R \rightarrow R; W \rightarrow R$
  - No need for RMW.
  - Write atomicity fence not needed.

- **PowerPC**: **SYNC** fence
  - Similar to **MB** except:
    - $R \rightarrow R$ to same location can still be OoO Exec (use RMW)
  - Write Atomicity may require RMW
    - Allows write to be seen early by another processor’s read
Abstractions

- Generally, compiler optz can go full bore between \textit{sync/special/fence} or \textit{sync IDs}.
  - Some optz can be done w.r.t. global shmobj objects.
- Programmer supplied, standardized safety nets.
  - “Don’t know; Assume worst” – Starting method?
    - Over-marking SYNCs is overly-conservative
- Programming Model Support
  - \texttt{doall} – no deps between iterations –(HPF/F95 – \texttt{forall, where})
  - SIMD (CUDA) – Implied multithread access w/o sync or IF cond
  - Data type - \texttt{volatile} - C/C++
  - Directives – \texttt{OpenMP}: \texttt{#pragma omp parallel} Sync Region
  - \texttt{#pragma omp shared(A)} Data Type
  - Library – (eg, MPI, OpenCL, CUDA)
Using the HW & Conclusions

- Compilers can
  - Protect memory ranges [low...high]
  - Assign data segments to protected page locations
  - Use high-order bits for addr of VM
  - Extra opcode usage (eg, GPU sync)
  - Modify internal memory disambiguation methods
  - Perform Inter-Procedural Optz for shmem optz.

Relaxed Memory Consistency Models

- + Puts more performance, power & responsibilities into hands of programmers and compilers.
- - Puts more performance, power & responsibilities into hands of programmers and compilers.
The End