INTERESTING CASE OF RULE 2

Based on the presented data, one may conclude that using -O3 is always a good idea. The presented data set contains only a subset of the Mantevo benchmark suite. The incompleteness of data may lead to wrong conclusions. Sometimes -O3 may not be a good idea for a code: e.g., vectorization (enabled by -O3) may segfault on a loop which does unaligned memory access on x86. But this is not demonstrated by the presented dataset.

Review of last lecture

- Architecture case studies
  - Memory performance is often the bottleneck
  - Parallelism grows with compute performance
  - Caching is important
  - Several issues to address for parallel systems
- Cache Coherence
  - Hardware support to aid programmers
  - Two guarantees:
    - Write propagation (updates are eventually visible to all readers)
    - Write serialization (writes to the same location are observed in global order)
  - Two major mechanisms:
    - Snooping
    - Directory-based
  - Protocols: MESI (MOESI, MESIF)

DPHPC Overview

- Don't forget the projects!
  - Groups to be defined by Thursday (send email to Salvatore)
  - Project progress presentations on 11/6 (<1 month from now)!
- Cache-coherence is not enough!
  - Many more subtle issues for parallel programs!
- Memory Models
  - Sequential consistency
  - Why threads cannot be implemented as a library
  - Relaxed consistency models
- Linearizability
  - More complex objects

Goals of this lecture

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)
  
  ```
  X = 2
  Y = 10
  while (X==0):
    Z = Y
  ```

  Class question: what value will Z on P2 have?
  - Y=10 does not need to have completed before X=2 is visible to P2!
  - This allows P2 to exit the loop and read Y=0!
  - This may not be the intent of the programmer!
  - This may be due to congestion (imagine X is pushed to a remote cache while Y misses to main memory) and or due to write buffering, or …
  - Bonus class question: what happens when Y and X are on the same cache line (assume simple MESI and no write buffer)?
Memory Models

- Need to define what it means to “read a location” and “to write a location” and the respective ordering!
  - What values should be seen by a processor
- First thought: extend the abstractions seen by a sequential processor:
  - Compiler and hardware maintain data and control dependencies at all levels:
    - Compiler
    - Hardware

Sequential Processor

- Correctness condition:
  - The result of the execution is the same as if the operations had been executed in the order specified by the program “program order”
  - A read returns the value last written to the same location
  - “last” is determined by program order!
- Consider only memory operations (e.g., a trace)

  - N Processors
  - P1, P2, …., PN
  - Operations
    - Read, Write on shared variables (initial state: most often all 0)
  - Notation:
    - P1: R(x):3
      - P1 reads x and observes the value 3
    - P2: W(x,5)
      - P2 writes 5 to variable x

Terminology

- Program order
  - Deals with a single processor
  - Per-processor order of memory accesses, determined by program’s control flow
  - Often represented as trace
- Visibility order
  - Deals with operations on all processors
  - Order of memory accesses observed by one or more processors
  - E.g., “every read of a memory location returns the value that was written last”
    - Defined by memory model

Memory Models

- Contract at each level between programmer and processor
- Optimization transformations
  - High-level language (API/PL)
  - Compiler frontend
  - Intermediate language (IR)
  - Compiler backend/JIT
  - Machine code (ISA)
  - Processor
  - Optimizing transformations
  - Reordering
  - Operation overlap
  - OOO Execution
  - VLIW, Vector ISA

Sequential Consistency

- Extension of sequential processor model
- The execution happens as if:
  - The operations of all processors were executed in some sequential order (atomicity requirement), and
  - The operations of each individual processor appear in this sequence in the order specified by the program (program order requirement)
- Applies to all layers:
  - Disallows many compiler optimizations (e.g., reordering of any memory instruction)
  - Disallows many hardware optimizations (e.g., store buffers, nonblocking reads, invalidation buffers)

Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order
Original SC Definition

“The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program”

(Lamport, 1979)

Alternative SC Definition

- Textbook: Hennessy/Patterson Computer Architecture
- A sequentially consistent system maintains three invariants:
  1. A load L from memory location A issued by processor P, obtains the value of the previous store to A by P, unless another processor has to store a value to A in between.
  2. A load L from memory location A obtains the value of a store S to A by another processor P, if S and L are “sufficiently separated in time” and if no other store occurred between S and L.
  3. Stores to the same location are serialized (defined as in [2]).
- “Sufficiently separated in time” not precise
  - Works but is not formal (a formalization must include all possibilities)

Example Operation Reordering

- Recap: “normal” sequential assumption:
  - Compiler and hardware can reorder instructions as long as control and data dependencies are met
- Examples:
  - Register allocation
  - Code motion
  - Common subexpression elimination
  - Loop transformations
  - Pipelining
  - Multiple issue (OOO)
  - Write buffer bypassing
  - Nonblocking reads

Sequential Consistency Examples

- Relying on program order: Dekker’s algorithm
  - Initially, all zero
  - What can happen at compiler and hardware level?

```
P1
a = 1
if (b == 0)
  critical section
  a = 0

P2
b = 1
if (a == 0)
  critical section
  b = 0
```

```
P1: R(b): 1
P1: W(a,5)
P2: R(a): 1
P2: W(b,1)
P3: R(a): 5
P3: R(b): 1
P1: W(a,1)
PRINT(5)
P3 has not seen yet!
```

Simple compiler optimization

- Initially, all values are zero
- Assume P1 and P2 are compiled separately!
- What optimizations can a compiler perform for P1?
  - Register allocation or even replace with constant, or
  - Switch statements
  - What happens?
  - P2 may never terminate, or
  - Compute with wrong input

```
P1
input = 23
ready = 1
P2
while (ready == 0) {}
compute(input)
```

Sequential Consistency Examples

What each P thinks the order is:

```
P3
P1: W(a,1)
P2: W(b,1)
```

```
P3 thinks the order is:
P1: W(a,1)
P3: W(b,5)
P2: W(b,1)
```

```
P3: W(a,5)
P1: W(a,1)
P3: R(a)
P2: W(b,1)
P3: R(b)
PRINT(5)
```

Alternatively, both writes may have went to a write buffer, in which case both P1 would read 0 and enter the critical section together.
Optimizations violating program order

- Analyzing P1 and P2 in isolation!
  - Compiler can reorder
  - Hardware can reorder, assume writes of a, b go to write buffer or speculation

Considerations

- Define partial order on memory requests A \rightarrow B
- If P_i issues two requests A and B and A is issued before B in program order, then A \rightarrow B
- A and B are issued to the same variable, and A is issued first, then A \rightarrow B (on all processors)
- These partial orders can be interleaved, define a total order
- Many total orders are sequentially consistent!
- Example:
  - P1: W(a), R(b), W(c)
  - P2: R(a), W(a), R(b)
  - Are the following schedules (total orders) sequentially consistent?
    1. P1: W(a), P2: R(a), P2: W(a), P1: R(b), P2: R(b), P1: W(c)
    2. P1: W(a), P2: R(a), P1: R(b), P2: R(b), P1: W(c), P2: W(a)
    3. P2: R(a), P2: W(a), P1: R(b), P1: W(a), P1: W(c), P2: R(b)

Discussion

- Programmer's view:
  - Prefer sequential consistency
  - Easiest to reason about

- Compiler/hardware designer's view:
  - Sequential consistency disallows many optimizations!
  - Substantial speed difference

- Most architectures and compilers don’t adhere to sequential consistency!

- Solution: synchronized programming
  - Access to shared data (aka. “racing accesses”) are ordered by synchronization operations
  - Synchronization operations guarantee memory ordering (aka. fence)
  - More later!
Cache Coherence vs. Memory Model

- Varying definitions!

- Cache coherence: a mechanism that propagates writes to other processors/caches if needed, recap:
  - Writes are eventually visible to all processors
  - Writes to the same location are observed in (one) order

- Memory models: define the bounds on when the value is propagated to other processors
  - E.g., sequential consistency requires all reads and writes to be ordered in program order

The fun begins: Relaxed Memory Models

- Sequential consistency
  - $R \rightarrow R, W \rightarrow W$ (all order guaranteed)

- Relaxed consistency (varying terminology):
  - Processor consistency (aka. TSO)
    - Releases $R \rightarrow W, W \rightarrow W$ (with aliasing)
  - Weak consistency and release consistency (aka. RM0)
    - Releases $R \rightarrow W, R \rightarrow M, W \rightarrow W$
  - Other combinations/variants possible
    - There are even more types of orders (above is a simplification)

Architectures

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha IA-64</th>
<th>PA RISC</th>
<th>POWER</th>
<th>SPARC</th>
<th>MIPS</th>
<th>SPARC</th>
<th>TSO</th>
<th>x86</th>
<th>IA-64/32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Loads reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Stores reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Stores reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic loads reordered</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic stores reordered</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Dependent loads reordered</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Dependent stores reordered</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Some older 68k and 386 systems have weaker memory ordering.

**Case Study: Memory ordering on Intel (x86)**

- Intel® 64 and IA-32 Architectures Software Developer’s Manual
  - Chapter 8.2 Memory Ordering

- Google Tech Talk: IA Memory Ordering
  - Richard L. Hudson
  - http://www.youtube.com/watch?v=WUfvvFD5tAA

x86 Memory model: TLO + CC

- Total lock order (TLO)
  - Instructions with "lock" prefix enforce total order across all processors
  - Implicit locking: xchg (locked compare and exchange)

- Causal consistency (CC)
  - Write visibility is transitive

- Eight principles
  - After some revisions ☺️

**The Eight x86 Principles**

1. "Reads are not reordered with other reads." ($R \rightarrow R$)
2. "Writes are not reordered with other writes." ($W \rightarrow W$)
3. "Writes are not reordered with older reads." ($R \rightarrow W$)
4. "Reads may be reordered with older writes to different locations but not with older writes to the same location." (NO $W \rightarrow W$)
5. "In a multiprocessor system, memory ordering obeys causality." (memory ordering respects transitive visibility)
6. "In a multiprocessor system, writes to the same location have a total order." (implied by cache coherence)
7. "In a multiprocessor system, locked instructions have a total order." (enables synchronized programming!)
8. "Reads and writes are not reordered with locked instructions." (enables synchronized programming!)
**Principle 1 and 2**

Reads are not reordered with other reads. (R→R)
Writes are not reordered with other writes. (W→W)

**Principle 3**

Writes are not reordered with other writes. (W→W)

**Principle 4**

Reads may be reordered with older writes to different locations but not with other writes to the same location. (NO W→W)

**Principle 5**

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

**Principle 6**

In a multiprocessor system, writes to the same location have a total order. (enables synchronized programming!)

**Principle 7**

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

Reads and writes observed in program order. Cannot be reordered!

If r1 == 2, then r2 must be 1!
Not allowed: r1 == 2, r2 == 0

Question: is r1=0, r2=1 allowed?

If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(a) → P1:R(a), thus r1 must be 0!

All values zero initially

All values zero initially

All values zero initially

All values zero initially

All values zero initially

All values zero initially

Question: is r1=0 and r2=0 allowed?

Question: is r1=1 and r2=1 allowed?

Question: is r1=1, r2=0, r3=1 allowed?

Question: is r1=0, r2=2, r3=0, r4=1 allowed?

Question: is r3=1, r4=0, r5=0, r6=1 allowed?

* Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 == 0
  * If P3 observes ordering P1:xchg P2:xchg, then P4 observes the same ordering
  * xchg has implicit lock

OK

OK

OK
Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

**Principle 8**

Not allowed:
- r2 == 0, r4 == 0
-Locked instructions have total order, so P1 and P2 agree on the same order

If volatile variables use locked instructions → practical sequential consistency (more later)

---

**Notions of Correctness**

We discussed so far:
- Read/write of the same location
- Cache coherence (write serialisation and atomicity)
- Read/write of multiple locations
  - Memory models (visibility order of updates by cores)
- Now: objects (variables/fields with invariants defined on them)
  - Invariants "tie" variables together
  - Sequential objects
  - Concurrent objects

---

**Sequential Objects**

- Each object has a type
- A type is defined by a class
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state
- Remark
  - An interface is an abstract type that defines behavior
  - A class implementing an interface defines several types

---

**Sequential Queue**

```cpp
class Queue {
private:
    int head, tail;
    std::vector<Item> items;
public:
    Queue(int capacity) {
        head = tail = 0;
        items.resize(capacity);
    }
    // ...
};
```
Sequential Queue

```cpp
class Queue {
public:
void enq(Item x) {
    // ...
    return item;
}
};
```

**Preconditions:**
- Specify conditions that must hold before method executes
- Involve state and arguments passed
- Specify obligations a client must meet before calling a method
- Example: `enq()`

**Invariants:**
- Queue must not be full!

**Postconditions:**
- Specify conditions that must hold after method executed
- Involve old state and arguments passed
- Example: `enq()`

**Advantages of sequential specification**
- One process executes operations one at a time
- Sequential semantics
- Semantics of operation defined by specification of the class
- Preconditions and postconditions
- Design by Contract!

Sequential Execution

Time

[Diagrams illustrating sequential execution with operations: `enq(x)`, `enq(y)`, `deq()`]
Concurrent threads invoke methods on possibly shared objects

- At overlapping time intervals

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
</table>
| State    | Meaningful only between method executions | Overlapping method executions → object may never be "between method executions."

Reasoning must now include all possible interleavings

- Of changes caused by methods themselves

Add Method

- Consider adding a method that returns the last item enqueued

```cpp
Item deq() {
    Item item = items[head];
    head = (head+1) % items.size();
}
```

```cpp
void enq(Item x) {
    Item x = items[tail]
    tail = (tail+1) % items.size();
}
```

```cpp
Item peek() {
    if (tail == head) throw EmptyException;
    return items[tail];
}
```

- Consider if `peek()` and `enq()` run concurrently: what if tail has not yet been incremented?
- Consider if `peek()` and `deq()` run concurrently: what if last item is being dequeued?

Lock-based queue

```cpp
class Queue {
public:
    std::lock_guard<std::mutex> lock;
    int head, tail;
    std::vector<Item*> items;
    std::mutex lock;
    Queue(int capacity) {
        items.resize(capacity);
    }
    // ...
}
```

We can use the lock to protect Queue's fields.

One of C++'s ways of implementing a critical section
C++ Resource Acquisition is Initialization

- RAII – suboptimal name
- Can be used for locks (or any other resource acquisition)
  - Constructor grabs resource
  - Destructor frees resource
- Behaves as if
  - Implicit unlock at end of block!
- Main advantages
  - Always unlock/free lock at exit
  - No "lost" locks due to exceptions or strange control flow (e.g., goto)
  - Very easy to use

Example execution

```cpp
void enq(Item x) {
  std::lock_guard<std::mutex> l; // lock
  if (tail == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail + 1) % items.size();
}

Item deq() {
  std::lock_guard<std::mutex> l; // lock
  if (tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head + 1) % items.size();
  return item;
}
```

Methods effectively execute one after another, sequentially.

Correctness

- Is the locked queue correct?
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?
- Class question: What is the problem with this approach?
  - Same as for SC

<table>
<thead>
<tr>
<th>enq(x)</th>
<th>deq()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It does not scale!
What is the solution here?