Programming Languages

Concurrency: Memory Consistency

Dr. Axel Simon and Dr. Michael Petter
Winter term 2014

Need for Concurrency

Consider two processors:
- in 1997 the Pentium P55C had 4.5M transistors
- in 2006 the Itanium 2 had 1700M transistors

\[ \therefore \text{Intel could have built a processor with 256 Pentium cores in 2006} \]

Need for Concurrency

Consider two processors:
- in 1997 the Pentium P55C had 4.5M transistors
- in 2006 the Itanium 2 had 1700M transistors

\[ \therefore \text{Intel could have built a processor with 256 Pentium cores in 2006} \]

However:
- most programs are not inherently parallel
  - \[ \therefore \text{parallelizing a program is between difficult and impossible} \]
- correctly communicating between different cores is challenging
  - \[ \therefore \text{correctness of concurrent communication is very hard} \]
  - low-level aspects: locking algorithms must be correct
  - high-level aspects: program may deadlock
- a program on \( n \) cores runs \( m \ll n \) times faster
  - \[ \therefore \text{all effort is voided if program runs no faster} \]
  - distributing work load is application specific
The free lunch is over
Single processors cannot be made much faster due to physical limitations.

Concurrency for the Programmer
How is concurrency exposed in a programming language?
- spawning of new concurrent computations
- communication between threads

Concurrency for the Programmer
How is concurrency exposed in a programming language?
- spawning of new concurrent computations
- communication between threads

But Moore's law still holds for the number of transistors:
- they double every 18 months for the foreseeable future
- may translate into doubling the number of cores
- programs have to become parallel

Memory Consistency | Motivation
---|---

Concurrent communication can happen in many ways:
- communication via shared memory (this lecture)
- atomic transactions on shared memory
- message passing

Learning Outcomes
- Happened-before Partial Order
- Sequential Consistency
- The MESI Cache Model
- Weak Consistency
- Memory Barriers
Communication between Cores

We consider the concurrent execution of these functions:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>`void foo(void) {</td>
<td>`void bar(void) {</td>
</tr>
<tr>
<td>a = 1;</td>
<td>while (b == 0) {</td>
</tr>
<tr>
<td>b = 1;</td>
<td>a = 1;</td>
</tr>
<tr>
<td>}</td>
<td>b = 1;</td>
</tr>
<tr>
<td></td>
<td>assert(a == 1);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

- initial state of `a` and `b` is 0
- `A` writes `a` before it writes `b`
- `B` should see `b` go to one before executing the `assert` statement

Communication between Cores

We consider the concurrent execution of these functions:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>`void foo(void) {</td>
<td>`void bar(void) {</td>
</tr>
<tr>
<td>a = 1;</td>
<td>while (b == 0) {</td>
</tr>
<tr>
<td>b = 1;</td>
<td>a = 1;</td>
</tr>
<tr>
<td>}</td>
<td>b = 1;</td>
</tr>
<tr>
<td></td>
<td>assert(a == 1);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

- initial state of `a` and `b` is 0
- `A` writes `a` before it writes `b`
- `B` should see `b` go to one before executing the `assert` statement
- the `assert` statement should always hold
- here the code is **correct** if the `assert` holds
Communication between Cores

We consider the concurrent execution of these functions:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void foo(void) {</code></td>
<td><code>void bar(void) {</code></td>
</tr>
<tr>
<td><code>  a = 1;</code></td>
<td><code>  while (b == 0) {</code></td>
</tr>
<tr>
<td><code>  b = 1;</code></td>
<td><code>  assert(a == 1);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

- initial state of `a` and `b` is 0
- `A` writes `a` before it writes `b`
- `B` should see `b` go to one before executing the `assert` statement
- the `assert` statement should always hold
- here the code is correct if the `assert` holds

`<` correctness means: writing a one to `a` happens before reading a one in `b`

Strict Consistency

Assuming `foo` and `bar` are started on two cores operating in lock-step. Then one of the following may happen:

```
foo
mem
bar
```

Unrealistic to assume that there is only one order between memory accesses:

- each conditional (and loop iteration) doubles the number of possible lock-step executions
- processors use caches: lock-step assumption is violated since cache behavior depends on data
Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P_1, \ldots$, each process $P$ consists of events $p_1, p_2, \ldots$.

Example:

$P$ $p_1$ $p_2$ $p_3$ $p_4$

$Q$ $q_1$ $q_2$ $q_3$ $q_4$ $q_5$ $q_6$ $q_7$

$R$ $r_1$ $r_2$ $r_3$ $r_4$

- event $p_i$ in process $P$ happened before $p_{i+1}$
- if $p_i$ is an event that sends a message to $Q$ then there is some event $q_j$ in $Q$ that receives this message and $p_i$ happened before $q_j$

Wand Law (I)

Events in time are like power of wands:
Wand Law (I)

Events in time are like power of wands:

- beats

hence:

- the "beats" relation is transitive

Wand Law (II)

More wand laws:

- "beats" is transitive
- "beats" is irreflexive

implies that "beats" is asymmetric: if

then

¬ "beats" is a strict partial order
**The Happened-Before Relation**

**Definition**
If an event $p$ happened before an event $q$ then $p \rightarrow q$. 

$\in \mathbb{E} \times \mathbb{E}$

Observe:
- $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)

**The Happened-Before Relation**

**Definition**
If an event $p$ happened before an event $q$ then $p \rightarrow q$.

Observe:
- $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q \land q \rightarrow r$ then $p \rightarrow r$)
The Happened-Before Relation

Definition
If an event \( p \) happened before an event \( q \) then \( p \rightarrow q \).

Observe:
- \( \rightarrow \) is partial (neither \( p \rightarrow q \) or \( q \rightarrow p \) may hold)
- \( \rightarrow \) is irreflexive (\( p \rightarrow p \) never holds)
- \( \rightarrow \) is transitive (\( p \rightarrow q \land q \rightarrow r \) then \( p \rightarrow r \))
- \( \rightarrow \) is asymmetric (if \( p \rightarrow q \) then \( \neg(q \rightarrow p) \))

\( \rightarrow \) the \( \rightarrow \) relation is a strict partial order

Note: a strict partial order \( \prec \) differs from a (non-strict) partial order \( \preceq \) due to:

<table>
<thead>
<tr>
<th>strict partial order</th>
<th>non-strict partial order</th>
</tr>
</thead>
<tbody>
<tr>
<td>irreflexive ( \neg(p \prec p) )</td>
<td>reflexive ( p \preceq p )</td>
</tr>
<tr>
<td>asymmetric ( p \prec q ) implies ( \neg(q \prec p) )</td>
<td>antisymmetric ( p \preceq q \land q \preceq p ) implies ( p = q )</td>
</tr>
</tbody>
</table>

Concurrency

Let \( a \not\rightarrow b \) abbreviate \( \neg(a \rightarrow b) \).

Definition
Two distinct events \( p \) and \( q \) are said to be concurrent if \( p \not\rightarrow q \) and \( q \not\rightarrow p \).

\( p \not\rightarrow q \) in the example

\( p_1 \rightarrow r_4 \) in the example

\( p_3 \) and \( q_3 \) are, in fact, concurrent since \( p_3 \not\rightarrow q_3 \) and \( q_3 \not\rightarrow p_3 \)
Let $C$ be a **logical clock** that assigns a time-stamp $C(p)$ to each event $p$.

**Definition (Clock Condition)**

$C$ satisfies the clock condition if for any events $p \rightarrow q$ then $C(p) < C(q)$.

For a distributed system the clock condition holds iff:

1. if $p_i$ and $p_j$ are events of $P$ and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. if $p$ is the sending of a message by process $P$ and $q$ is the reception of this message by process $Q$ then $C(p) < C(q)$

→ a logical clock $C$ that satisfies the clock condition describes a **total order** $a < b$ (with $C(a) < C(b)$) that is compatible with the strict partial order $\rightarrow$
Defining $C$ Satisfying the Clock Condition

Given:

Summary

We can model concurrency using processes and events:
- there is a happened-before relation between the events of each process
- there is a happened-before relation between communicating events
- happened-before is a strict partial order
- a clock is a total strict order that embeds the happened-before partial order

Moving Away from Strong Consistency

Idea: use process diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:
- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
- $\sim$ concisely represent some interleavings

We obtain a model for sequential consistency.
### Definition: Sequential Consistency

**Definition (Sequential Consistency Condition for Multi-Processors)**

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 this sequence in the order specified by its program.

Given a result of a program with $n$ threads on a SC system,

- with operations $p_0^i, p_1^i, \ldots$ and $p_0^k, p_1^k, \ldots$ and $\ldots p_0^l, p_1^l, \ldots$,

- there exists a total order $\exists C \cdot C(p_j^i) < C(p_j^k)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,

- such that this execution has the same result.
**Definition: Sequential Consistency**

**Definition (Sequential Consistency Condition for Multi-Processors)**

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 this sequence in the order specified by its program.

Given a result of a program with \( n \) threads on a SC system,
1. with operations \( p_0^1, p_1^1, \ldots \) and \( p_0^2, p_1^2, \ldots \) and \( p_0^l, p_1^l, \ldots \)
2. there exists a total order \( \exists C : C(p_i^j) < C(p_i^l) \) for all \( i, j, k, l \ldots \) where \( j = l \) implies \( i < k \),
3. such that this execution has the same result.

Yet, in other words:
- \( \bullet \) defines the **execution path** of each thread

**Disproving Sequential Consistency**

Given a result of a program with \( n \) threads on a SC system,
1. with operations \( p_0^1, p_1^1, \ldots \) and \( p_0^2, p_1^2, \ldots \) and \( p_0^l, p_1^l, \ldots \)
2. there exists a total order \( \exists C : C(p_i^j) < C(p_i^l) \) for all \( i, j, k, l \ldots \) where \( j = l \) implies \( i < k \),
3. such that this execution has the same result.

Yet, in other words:
- \( \bullet \) defines the **execution path** of each thread
- the total order defined in \( \bullet \) is one **interleaving** of the execution paths

**Defining the execution path**

\[ \bullet \] defines the **execution path** of each thread

**Disproving Sequential Consistency**

Given a result of a program with \( n \) threads on a SC system,
1. with operations \( p_0^1, p_1^1, \ldots \) and \( p_0^2, p_1^2, \ldots \) and \( p_0^l, p_1^l, \ldots \)
2. there exists a total order \( \exists C : C(p_i^j) < C(p_i^l) \) for all \( i, j, k, l \ldots \) where \( j = l \) implies \( i < k \),
3. such that this execution has the same result.

Idea for showing that a system is **not** sequentially consistent:
- pick a result obtained from a program run on a SC system
Disproving Sequential Consistency

Given a result of a program with \( n \) threads on a SC system,

- with operations \( p_0^1, p_1^1, \ldots \) and \( p_0^2, p_1^2, \ldots \) and \( \ldots p_0^n, p_1^n, \ldots \)
- there exists a total order \( \exists C : C(p_i^j) < C(p_l^k) \) for all \( i, j, k, l \ldots \) where \( j = l \) implies \( i < k \),
- such that this execution has the same result.

Idea for showing that a system is not sequentially consistent:
- pick a result obtained from a program run on a SC system
- pick an execution \( \ddagger \) and a total ordering of all operations \( \ddagger \)
- add extra processes to model other system components
- the original order \( \ddagger \) becomes a partial order \( \rightarrow \)

Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.
- idea: model each memory location as a different process
Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location as a different process

Sequential consistency still obeyed:
- the accesses of foo to a occurs before b

Benefits of Sequential Consistency

Benefits of the sequential consistency model:
- concisely represent all interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- ~a good model for correct behaviors of concurrent programs
- ~programs results besides SC results are undesirable (they contain races)

It is a realistic model for older hardware:
- sequential consistency model suitable for concurrent processors that acquire exclusive access to memory
- processors can speed up computation by using caches and still maintain sequential consistency
Benefits of Sequential Consistency

Benefits of the sequential consistency model:
- concisely represent all interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- a good model for correct behaviors of concurrent programs
- programs results besides SC results are undesirable (they contain races)

It is a realistic model for older hardware:
- sequential consistency model suitable for concurrent processors that acquire exclusive access to memory
- processors can speed up computation by using caches and still maintain sequential consistency

Not a realistic model for modern hardware with out-of-order execution:
- what other processors see is determined by complex optimizations to caching
  need to understand how caches work

The Mesi Protocol: States

Procesors (and also: GPUs, intelligent I/O devices) use caches to avoid a costly round-trip to RAM for every memory access.
- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

Each cache line is in one of the states $M, E, S, I$:

- $M$: it is invalid and is ready for re-use
- $S$: other caches have an identical copy of this cache line, it is shared
- $E$: it is invalid and is ready for re-use
The Mesi Protocol: States

Processors (and also: GPUs, intelligent I/O devices) use caches to avoid a costly round-trip to RAM for every memory access.
- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

Each cache line is in one of the states M, E, S, I:
- I: it is invalid and is ready for re-use
- S: other caches have an identical copy of this cache line, it is shared
- E: the content is in no other cache; it is exclusive to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore been modified

The Mesi Protocol: Messages

Moving data between caches is coordinated by sending messages [McK10]:
- Read: sent if CPU needs to read from an address
The Mesi Protocol: Messages

Moving data between caches is coordinated by sending messages [McK10]:

- **Read**: sent if CPU needs to read from an address
- **Read Response**: response to a read message, carries the data at the requested address
- **Invalidate**: asks others to evict a cache line
- **Invalidation Acknowledge**: reply indicating that an address has been evicted

```
M  \arrow[dr]\arrow[ur] \quad E
\downarrow \quad \downarrow
S  \arrow[dr]\arrow[ur] \quad I
```

The Mesi Protocol: Messages

Moving data between caches is coordinated by sending messages [McK10]:

- **Read**: sent if CPU needs to read from an address
- **Read Response**: response to a read message, carries the data at the requested address
- **Invalidate**: asks others to evict a cache line
- **Invalidation Acknowledge**: reply indicating that an address has been evicted
- **Read Invalidate**: like **Read** + **Invalidate** (also called "read with intend to modify")
The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McK10]:

- **Read**: sent if CPU needs to read from an address
- **Read Response**: response to a *read* message, carries the data at the requested address
- **Invalidate**: asks others to evict a cache line
- **Invalidate Acknowledge**: reply indicating that an address has been evicted
- **Read Invalidate**: like *Read + Invalidate* (also called “read with intend to modify”)
- **Writeback**: info on what data has been sent to main memory

---

We mostly consider messages between processors. Upon *(Read) Invalidate*, a processor replies with *Read Response/Writeback* before the *Invalidate Acknowledge* is sent.

---

MESA Example

Consider how the following code might execute:

**Thread A**

```
 thread A
 a = 1; // A.1
 b = 1; // A.2
```

**Thread B**

```
 while (b == 0) {} // B.1
 assert(a == 1); // B.2
```

- In all examples, the initial values of variables are assumed to be 0

---

MESA Example

Consider how the following code might execute:

**Thread A**

```
 thread A
 a = 1; // A.1
 b = 1; // A.2
 assert(a == 1); // B.2
```

**Thread B**

```
 while (b == 0) {} // B.1
 assert(a == 1); // B.2
```

- In all examples, the initial values of variables are assumed to be 0
- Suppose that a and b reside in different cache lines
- Assume that a cache line is larger than the variable itself
MESI Example

Consider how the following code might execute:

Thread A

```
a = 1;  // A.1
b = 1;  // A.2
```

Thread B

```
while (b == 0) {};  // B.1
assert(a == 1);    // B.2
```

- In all examples, the initial values of variables are assumed to be 0
- Suppose that a and b reside in different cache lines
- Assume that a cache line is larger than the variable itself
- We write the content of a cache line as
  - M: modified, with value x

---

MESI Example (I)

Thread A

```
a = 1;  // A.1
b = 1;  // A.2
```

Thread B

```
while (b == 0) {};  // B.1
assert(a == 1);    // B.2
```

---

MESI Example (II)

Thread A

```
a = 1;  // A.1
b = 1;  // A.2
```

Thread B

```
while (b == 0) {};  // B.1
assert(a == 1);    // B.2
```

---

Memory Consistency

The MESI Protocol

<table>
<thead>
<tr>
<th>State</th>
<th>CPU A</th>
<th>CPU B</th>
<th>RAM</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>I</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B.1</td>
<td>M1</td>
<td>M1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B.1</td>
<td>M1</td>
<td>M1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A.2</td>
<td>M1</td>
<td>M1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(read invalidate of a from CPU A)

(invalidate ack. of a from CPU B)

(read response of a=0 from RAM)

(read of b from CPU B)

(read response of b=0 from RAM)

(read invalidate of b from CPU A)

(read response of b=0 from CPU B)

(invalidate ack. of b from CPU B)

<table>
<thead>
<tr>
<th>State</th>
<th>CPU A</th>
<th>CPU B</th>
<th>RAM</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>M1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B.2</td>
<td>S1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A.1</td>
<td>S1</td>
<td>S1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(read of b from CPU B)

(write back of b=1 from CPU A)

(read of a from CPU B)

(write back of a=1 from CPU A)

(invalidate of a from CPU A)

(invalidate of b from CPU B)
**MESI Example: Happened Before Model**

**Idea:** each cache line one process, A caches b=0 as E, B caches a=0 as E

Observations:
- each memory access must complete before executing next instruction
- add edge

**Can MESI Messages Collide?**

If two processors emit a message at the same time, the protocol might break. Access to common bus is coordinated by an *arbiter*.

**Source:** YouTube “The Mysterious Ticking Noise”
Can MESI Messages Collide?

If two processors emit a message at the same time, the protocol might break. Access to common bus is coordinated by an arbiter.

Sequential consistency:
- a characterization of well-behaved programs
- a model for different speed of execution
- for fixed paths through the threads and a total order between accesses to the same variable: executions can be illustrated by happened before diagram with one process per variable
- MESI cache coherence protocol ensures SC for processors with caches

Out-of-Order Execution

performance problem: writes always stall

Thread A
\[
\begin{align*}
    a &= 1; \quad // A.1 \\
    b &= 1; \quad // A.2
\end{align*}
\]

Thread B
\[
\begin{align*}
    \text{while } (b == 0) \{ ; \} \quad // B.1 \\
    \text{assert}(a == 1); \quad // B.2
\end{align*}
\]