Script generated by TTT

Title: Simon: Programmiersprachen (26.10.2012)

Date: Fri Oct 26 10:19:14 CEST 2012

Duration: 73:04 min

Pages: 70

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Memory Consistency

Dr. Axel Simon and Dr. Michael Petter
Winter term 2012

Need for Concurrency

Consider two processors:

- in 1997 the *Pentium P55C* had 4.5M transistors
- in 2006 the *Itanium 2* had 1700M transistors

⇝ 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

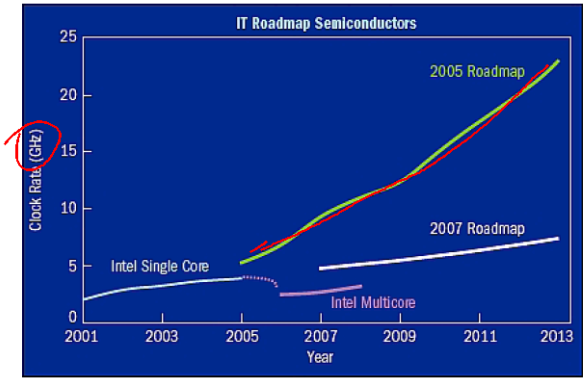
⇝ Intel could have built a processor with 256 Pentium cores in 2006

However:

- most programs are not inherently parallel
  - ⇝ parallelizing a program is between difficult and impossible
- correctly communicating between different cores is challenging
  - ⇝ 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
  - ⇝ all effort is voided if program runs no faster
  - distributing work load is application specific

The free lunch is over

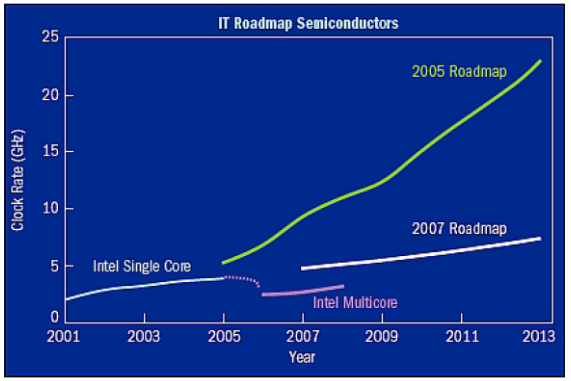
Single processors cannot be made much faster due to physical limitations.



Source: D. Patterson, UC–Berkeley

The free lunch is over

Single processors cannot be made much faster due to physical limitations.



Source: D. Patterson, UC–Berkeley

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 | 3 / 35

Concurrency for the Programmer

How is concurrency exposed in a programming language?

1. spawning of new concurrent computations
2. communication between threads

Concurrency for the Programmer

How is concurrency exposed in a programming language?

1. spawning of new concurrent computations
2. communication between threads

Communication can happen in many ways:

- communication via shared memory *(this lecture)*
- atomic transactions on shared memory
- message passing

Learning Outcomes

1. Happened-before Partial Order
2. Sequential Consistency
3. The MESI Cache Model
4. Weak Consistency
5. Memory Barriers

Memory Consistency | Motivation | 4 / 35

Communication between Cores

We consider the concurrent execution of these functions:

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- initial state of `a` and `b` is 0

Communication between Cores

We consider the concurrent execution of these functions:

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- initial state of `a` and `b` is 0
- A writes `a` before it writes `b`

Communication between Cores

We consider the concurrent execution of these functions:

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- 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:

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- 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

Communication between Cores

We consider the concurrent execution of these functions:

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

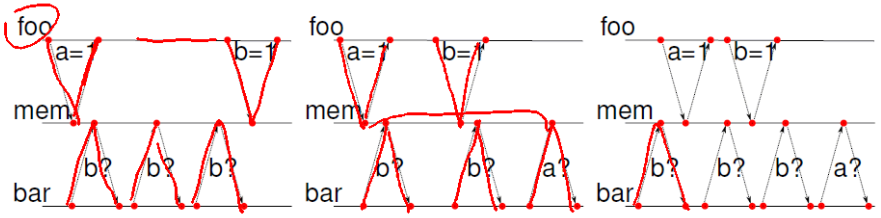
```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- 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`

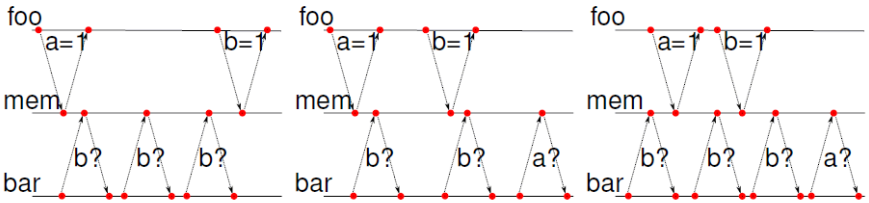
Strict Consistency

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



Strict Consistency

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

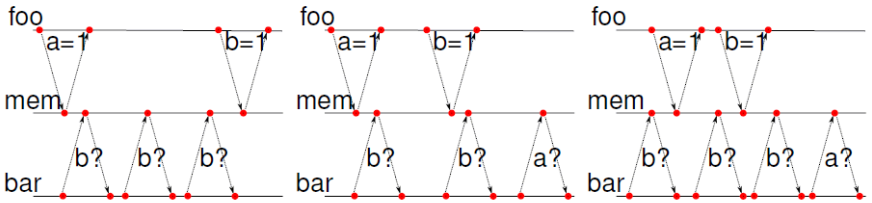


Unrealistic to assume that there is only one order between memory accesses:

- each conditional (and loop iteration) doubles the number of possible lock-step executions
- processors use caches ⇝ lock-step assumption is violated since cache behaviour depends on data

Strict Consistency

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



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 behaviour depends on data

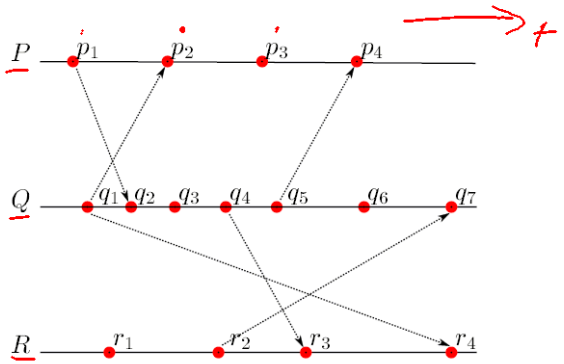
⇝ strict consistency is too strong to be realistic
⇝ state correctness in terms of what event *may* happen before another one

Events in a Distributed System

A process as a series of events [Lamport(1978)]: Given a distributed system of processes $P, \ldots$, each process P consists of events $p_1, p_2, \ldots$.

Events in a Distributed System

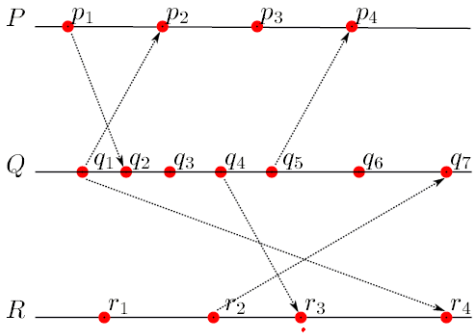
A process as a series of events [Lamport(1978)]: Given a distributed system of processes $P, \ldots$, each process P consists of events $p_1, p_2, \ldots$.
Example:



- event p_i in process P *happened before* p_{i+1}

Events in a Distributed System

A process as a series of events [Lamport(1978)]: Given a distributed system of processes $P, \ldots$, each process P consists of events $p_1, p_2, \ldots$.
Example:



- 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

The Happened-Before Relation

Definition

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

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is irreflexive ($p \to p$ never holds)

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is irreflexive ($p \to p$ never holds)
- $\to$ is transitive ($p \to q \wedge q \to r$ then $p \to r$)

Memory Consistency | Happened-Before Relation | 8 / 35

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is irreflexive ($p \to p$ never holds)
- $\to$ is transitive ($p \to q \wedge q \to r$ then $p \to r$)
- $\to$ is asymmetric ($p \to q$ if and only if $\neg(q \to p)$)

Memory Consistency | Happened-Before Relation | 8 / 35

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is irreflexive ($p \to p$ never holds)
- $\to$ is transitive ($p \to q \wedge q \to r$ then $p \to r$)
- $\to$ is asymmetric ($p \to q$ if and only if $\neg(q \to p)$)

$\rightsquigarrow$ the $\to$ relation is a *strict partial order*

Memory Consistency | Happened-Before Relation | 8 / 35

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is irreflexive ($p \to p$ never holds)
- $\to$ is transitive ($p \to q \wedge q \to r$ then $p \to r$)
- $\to$ is asymmetric ($p \to q$ if and only if $\neg(q \to p)$)

$\rightsquigarrow$ the $\to$ relation is a *strict partial order*

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

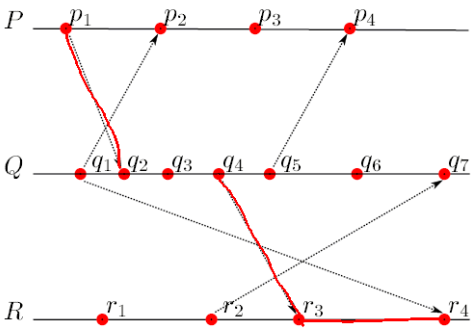
strict partial order	non-strict partial order
irreflexive $\neg(p \prec p)$	reflexive $p \preceq p$
asymmetric $p \prec q \iff \neg(q \prec p)$	antisymmetric $p \preceq q \wedge q \preceq p$ implies $p = q$

Concurrency

Let $a \not\to b$ abbreviate $\neg(a \to b)$.

Definition

Two distinct events p and q are said to be *concurrent* if $p \not\to q$ and $q \not\to p$.



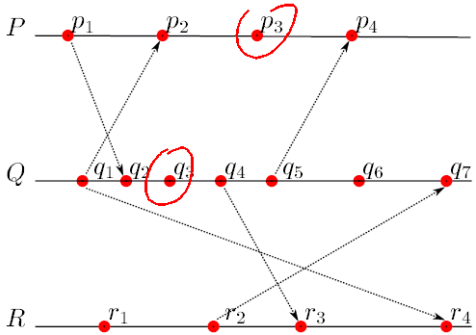
- $p_1 \to r_4$ in the example

Concurrency

Let $a \not\to b$ abbreviate $\neg(a \to b)$.

Definition

Two distinct events p and q are said to be *concurrent* if $p \not\to b$ and $b \not\to a$.



- $p_1 \to r_4$ in the example
- p_3 and q_3 are, in fact, concurrent since $p_3 \not\to q_3$ and $q_3 \not\to p_3$

Ordering

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 $p \to q$ then $C(p) < C(q)$.

Ordering

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 distributes 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 $C(p) < C(q)$

Ordering

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 distributes 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 $C(p) < C(q)$

$\rightsquigarrow$ 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 $\rightarrow$

Ordering

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 distributes 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 $C(p) < C(q)$

$\rightsquigarrow$ 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 $\rightarrow$

The *set* of C that satisfy the clock condition are exactly the *set* of executions possible in the system.
$\rightsquigarrow$ use the process model and $\rightarrow$ to define better consistency model

Sequential Consistency

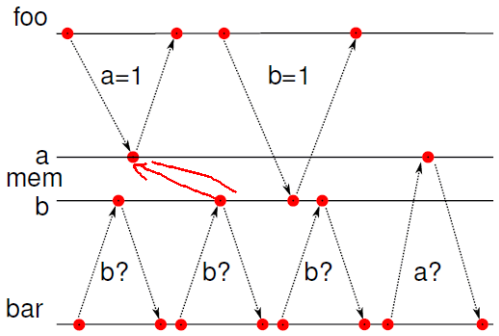
Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process

Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

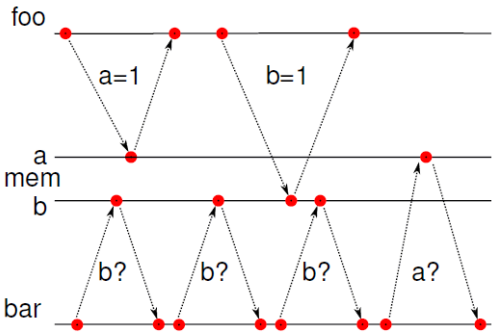
- model each memory location as different process



Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process



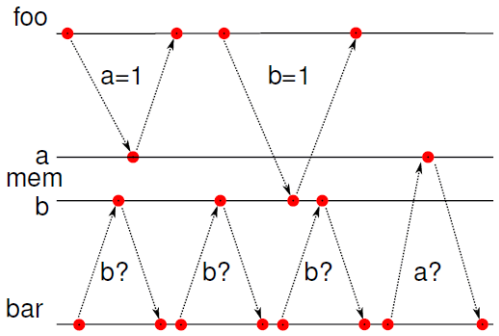
Some observations:

- the accesses of `foo` to `a` occurs before `b`

Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process



Some observations:

- the accesses of `foo` to `a` occurs before `b`
- the first two read accesses to `b` are in parallel to `a=1`

Benefits of Sequential Consistency

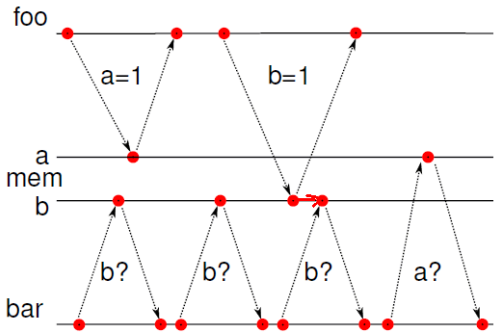
Benefits of the sequential consistency model:

- it is a realistic model for well-behaved concurrency
- . . . since it allows to concisely represent *some* interleavings

Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process



Some observations:

- the accesses of `foo` to `a` occurs before `b`
- the first two read accesses to `b` are in parallel to `a=1`

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- it is a realistic model for well-behaved concurrency
- . . . since it allows to concisely represent *some* interleavings

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- it is a realistic model for well-behaved concurrency
- . . . since it allows to concisely represent *some* interleavings

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* that avoid the exclusive access to memory as much as possible

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- it is a realistic model for well-behaved concurrency
- . . . since it allows to concisely represent *some* interleavings

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* that avoid the exclusive access to memory as much as possible

Not a realistic model for modern hardware with out-of-order execution:

- memory accesses are only sequentially consistent from the point of view of the processor executing them
- what other processors see is determined by the complex protocol of the caches

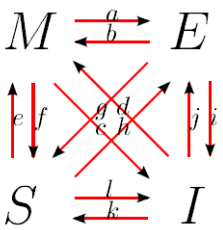
⇝ need to understand how caches work

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:

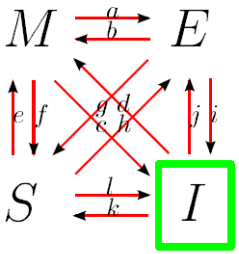


The MESI Protocol: States

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:



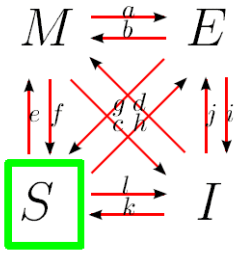
I: it is *invalid* and is ready for re-use

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

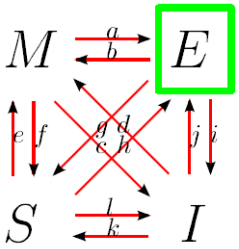
S: other caches have an identical copy of this cache line, it is *shared*

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

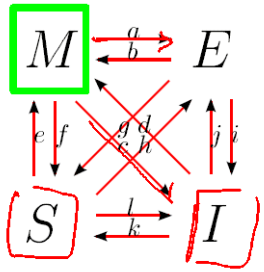
S: other caches have an identical copy of this cache line, it is *shared*

E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches

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



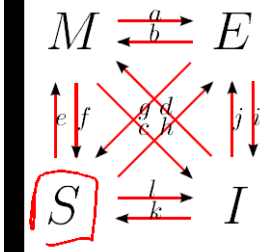
Each cache line is in one of the states M, E, S, I:

- 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: 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:

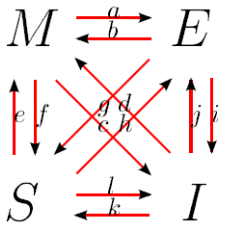
- I: it is *invalid* and is ready for re-use
- S: other caches have an identical copy of this cache line, it is *shared*
- E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore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

$\rightsquigarrow$ the state of cache lines is kept consistent by sending *messages*

The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

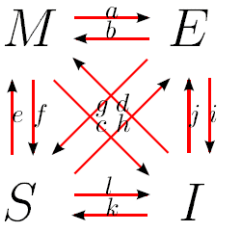
- *Read:* sent if CPU needs to read from an address



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

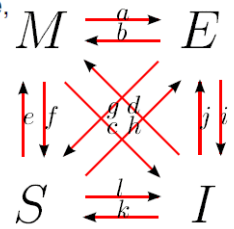
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

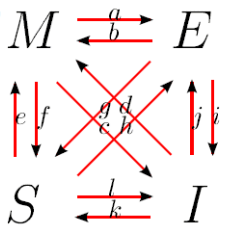
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

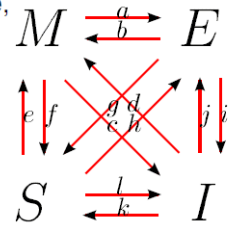
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

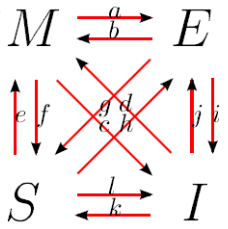
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted
- *Read Invalidate:* like *Read*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



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

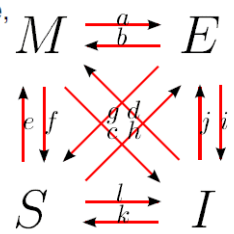
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted
- *Read Invalidate:* like *Read*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



The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

- *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



Additional *store* and *read* messages are transmitted to main memory.

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
```

state-ment	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
A.1	- I	- I	- I	- I	0	0	
	- I	- I	- I	- I	0	0	*read invalidate* of a from CPU A
	- I	- I	- I	- I	0	0	*invalidate ack.* of a from CPU B
	1 M	- I	- I	- I	0	0	*read response* of a=0 from RAM
B.1	1 M	- I	- I	- I	0	0	*read* of b from CPU B
	1 M	- I	- I	- I	0	0	*read response* with b=0 from RAM
	1 M	- I	- I	0 E	0	0	
B.1	1 M	- I	- I	0 E	0	0	
A.2	1 M	- I	- I	0 E	0	0	*read invalidate* of b from CPU A
	1 M	- I	- I	- I	0	0	*invalidate ack.* of b from CPU B
	1 M	- I	- I	- I	0	0	*read response* of b=0 from CPU B
	1 M	1 M	- I	- I	0	0	

MESI Example (II)

Thread A

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

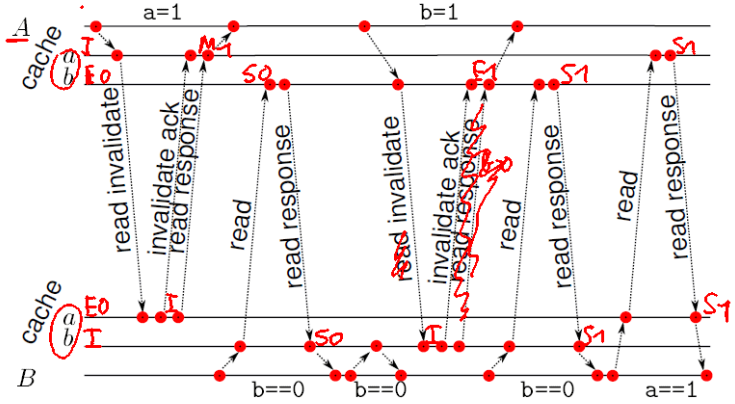
Thread B

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

state-ment	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
B.1	1 M	1 M	- I	- I	0	0	
	1 M	1 M	- I	- I	0	0	*read* of b from CPU B
B.2	1 M	1 S	- I	1 S	0	0	*read response* of b=1 from CPU A
	1 M	1 S	- I	1 S	0	0	*read* of a from CPU B
	1 S	1 S	1 S	1 S	0	0	*read response* of a=1 from CPU A
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	1 S	1 S	1 S	1 S	0	0	
	1 S	1 S	- I	1 S	0	0	*invalidate* of a from CPU A
	1 M	1 S	- I	1 S	0	0	*invalidate ack.* of a from CPU B

MESI Example: Happened Before Model

Idea: each cache line one process, model each message

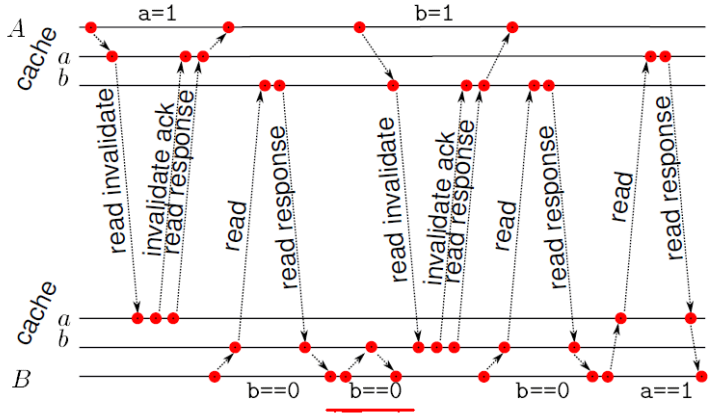


Observations:

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

MESI Example: Happened Before Model

Idea: each cache line one process, model each message

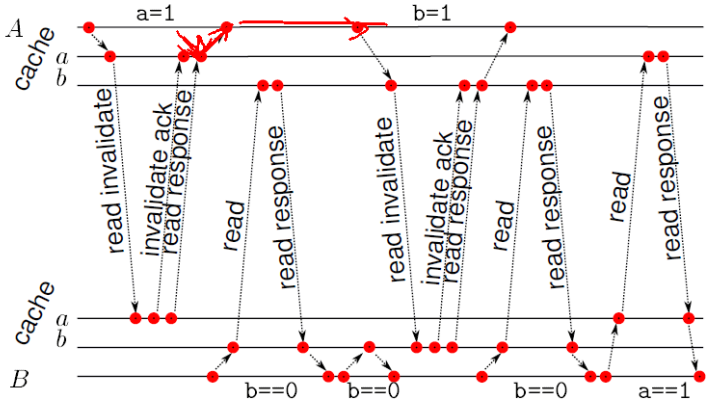


Observations:

- each memory access must complete before executing next instruction
  ⇝ add edge
- second execution of test `b==0` stays within cache ⇝ no traffic

MESI Example: Happened Before Model

Idea: each cache line one process, model each message

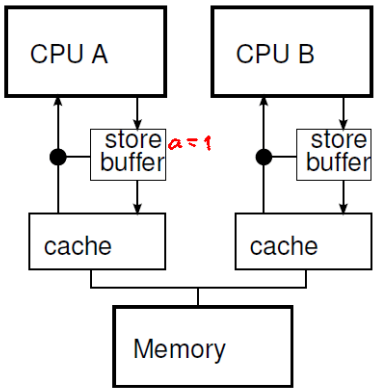


Observations:

- each memory access must complete before executing next instruction
  ⇝ add edge
- second execution of test `b==0` stays within cache ⇝ no traffic
- writes always stall ⇝ CPU A should continue executing after $a = 1$

Store Buffers

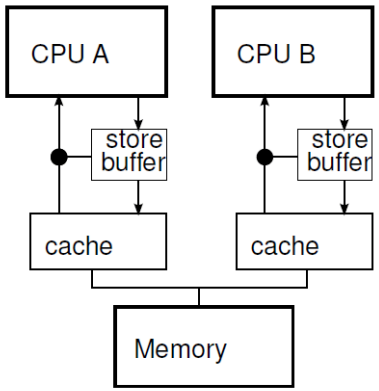
Goal: continue execution after write operation



- put each write into a *store buffer*
  and trigger reception of cache line

Store Buffers

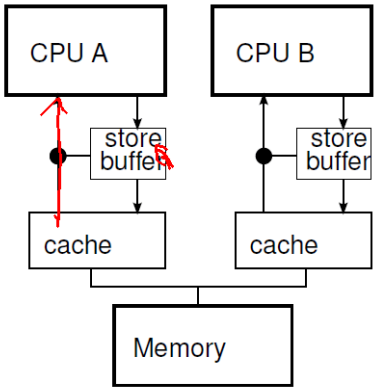
Goal: continue execution after write operation



- put each write into a *store buffer*
  and trigger reception of cache line
- once a cache line has arrived,
  apply relevant writes

Store Buffers

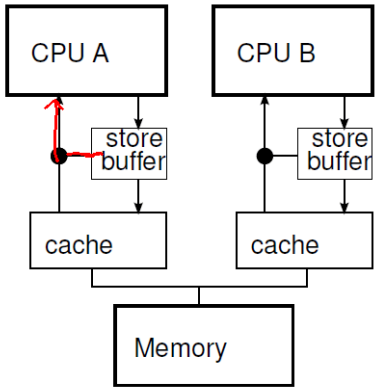
Goal: continue execution after write operation



- put each write into a *store buffer* and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a *set*
- ⚠ sequential consistency per CPU is violated unless

Store Buffers

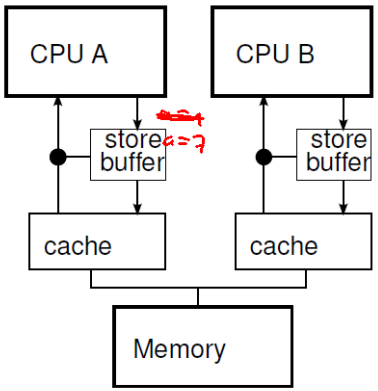
Goal: continue execution after write operation



- put each write into a *store buffer* and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a *set*
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache

Store Buffers

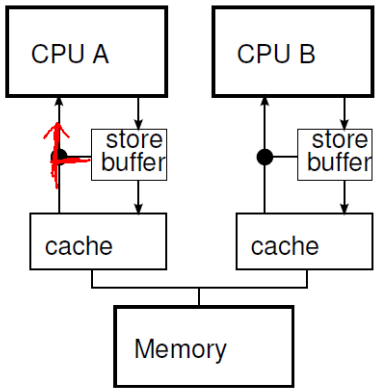
Goal: continue execution after write operation



- put each write into a *store buffer* and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a *set*
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the value that is waiting to be written
  - a write to the same location is combined with an existing write

Store Buffers

Goal: continue execution after write operation



- put each write into a *store buffer* and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a *set*
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the value that is waiting to be written
  - a write to the same location is combined with an existing write

What about sequential consistency for the whole system?

Happened-Before Model for Store Buffers

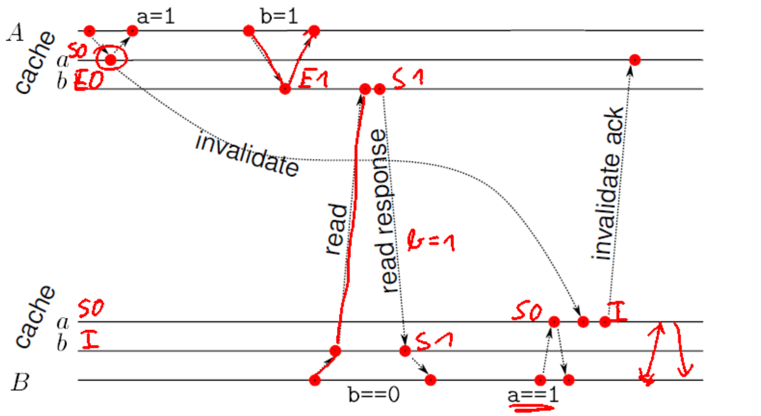
Thread A

```
a = 1;
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Happened-Before Model for Store Buffers

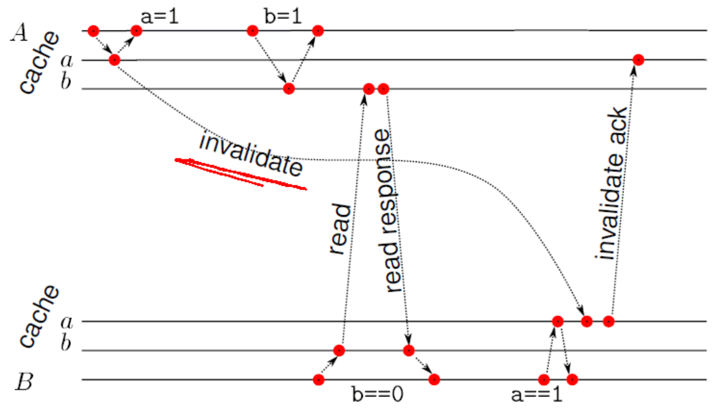
Thread A

```
a = 1;
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Happened-Before Model for Store Buffers

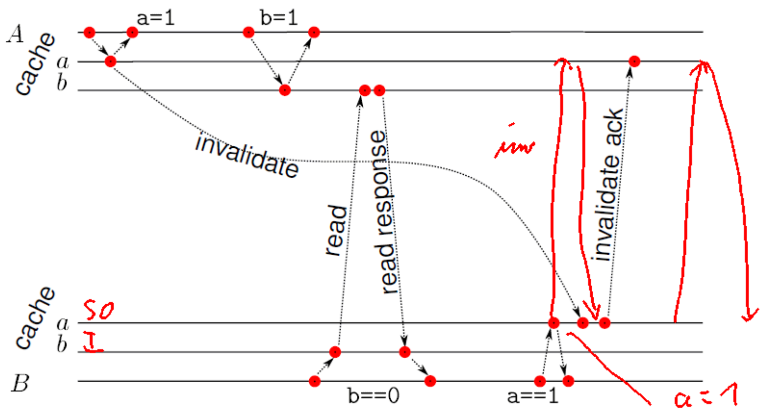
Thread A

```
a = 1;
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Explicit Synchronisation: Write Barrier

Overtaking of messages *is desirable* and should not be prohibited in general.

- store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit *write barrier* has to be inserted

Explicit Synchronisation: Write Barrier

Overtaking of messages *is desirable* and should not be prohibited in general.

- store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit *write barrier* has to be inserted
- Intel x86 CPUs provide the `sfence` instruction

Explicit Synchronisation: Write Barrier

Overtaking of messages *is desirable* and should not be prohibited in general.

- store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever two stores in one CPU must appear in sequence at a different CPU, an explicit *write barrier* has to be inserted
- Intel x86 CPUs provide the `sfence` instruction
- a write barrier marks all current store operations in the store buffer
- the next store operation is only executed when all marked stores in the buffer have completed

Happened-Before Model for Write Fences

Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I