Script generated by TTT

Title: Petter: Programmiersprachen (23.10.2019)

Date: Wed Oct 23 12:30:52 CEST 2019

Duration: 76:13 min

Pages: 27

MESI Example

Consider how the following code might execute:

Thread A	Thread B

`a = 1;    // A.1`
`b = 1;    // A.2`
	`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
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

MESI Example

Consider how the following code might execute:

Thread A	Thread B

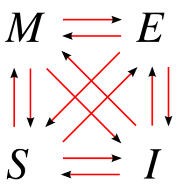
`a = 1;    // A.1`
`b = 1;    // A.2`
	`while (b == 0) {};  // B.1`
	`assert(a == 1);     // B.2`

- 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
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

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:* when in state E or S, response to a *Read* message, carries the data for the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that a cache line has been evicted
- *Read Invalidate:* like *Read* + *Invalidate* (also called "read with intend to modify")
- *Writeback: Read Response* when in state M, as a side effect noticing main memory about modifications to the cacheline, changing sender's state to S



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

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
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

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
```

statement	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	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M1	I	I	I	0	0	read of b from CPU B
	M1	I	I	I	0	0	read response with b=0 from RAM
B.1	M1	I	I	E0	0	0	
A.2	M1	I	I	E0	0	0	read invalidate of b from CPU A
	M1	I	I	E0	0	0	read response of b=0 from CPU B
	M1	S0	I	S0	0	0	invalidate ack. of b from CPU B
	M1	M1	I	I	0	0	

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
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

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
```

statement	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	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M1	I	I	I	0	0	read of b from CPU B
	M1	I	I	I	0	0	read response with b=0 from RAM
B.1	M1	I	I	E0	0	0	
A.2	M1	I	I	E0	0	0	read invalidate of b from CPU A
	M1	I	I	E0	0	0	read response of b=0 from CPU B
	M1	S0	I	S0	0	0	invalidate ack. of b from CPU B
	M1	M1	I	I	0	0	

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
```

statement	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	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M 1	I	I	I	0	0	read of b from CPU B
	M 1	I	I	I	0	0	read response with b=0 from RAM
B.1	M 1	I	I	E 0	0	0	
A.2	M 1	I	I	E 0	0	0	read invalidate of b from CPU A
	M 1	I	I	E 0	0	0	read response of b=0 from CPU B
	M 1	S 0	I	S 0	0	0	invalidate ack. of b from CPU B
	M 1	M 1	I	I	0	0	

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
```

statement	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	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M 1	I	I	I	0	0	read of b from CPU B
	M 1	I	I	I	0	0	read response with b=0 from RAM
B.1	M 1	I	I	E 0	0	0	
A.2	M 1	I	I	E 0	0	0	read invalidate of b from CPU A
	M 1	I	I	E 0	0	0	read response of b=0 from CPU B
	M 1	S 0	I	S 0	0	0	invalidate ack. of b from CPU B
	M 1	M 1	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
```

statement	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
B.1	M 1	M 1	I	I	0	0	read of b from CPU B
	M 1	M 1	I	I	0	0	write back of b=1 from CPU A
B.2	M 1	S 1	I	S 1	0	1	read of a from CPU B
	M 1	S 1	I	S 1	0	1	write back of a=1 from CPU A
	S 1	S 1	S 1	S 1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	S 1	S 1	S 1	S 1	1	1	invalidate of a from CPU A
	S 1	S 1	I	S 1	1	1	invalidate ack. of a from CPU B
	M 1	S 1	I	S 1	1	1	

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
```

statement	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
B.1	M 1	M 1	I	I	0	0	read of b from CPU B
	M 1	M 1	I	I	0	0	write back of b=1 from CPU A
B.2	M 1	S 1	I	S 1	0	1	read of a from CPU B
	M 1	S 1	I	S 1	0	1	write back of a=1 from CPU A
	S 1	S 1	S 1	S 1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	S 1	S 1	S 1	S 1	1	1	invalidate of a from CPU A
	S 1	S 1	I	S 1	1	1	invalidate ack. of a from CPU B
	M 1	S 1	I	S 1	1	1	

MESI Example (II)

Thread A

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

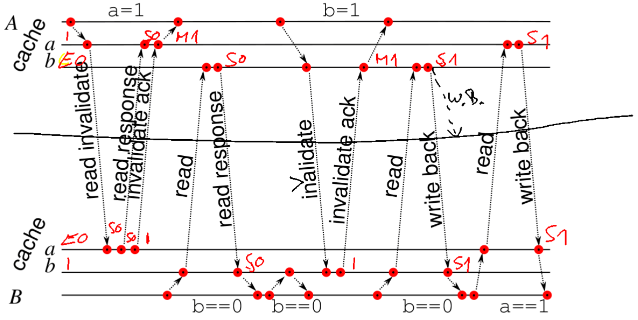
Thread B

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

statement	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
B.1	M1	M1	I	I	0	0	read of b from CPU B
	M1	M1	I	I	0	0	write back of b=1 from CPU A
B.2	M1	S1	I	S1	0	1	read of a from CPU B
	M1	S1	I	S1	0	1	write back of a=1 from CPU A
	S1	S1	S1	S1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	S1	S1	S1	S1	1	1	invalidate of a from CPU A
	S1	S1	I	S1	1	1	invalidate ack. of a from CPU B
	M1	S1	I	S1	1	1	

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

Introducing Store Buffers: Out-Of-Order Stores

Out-of-Order Execution

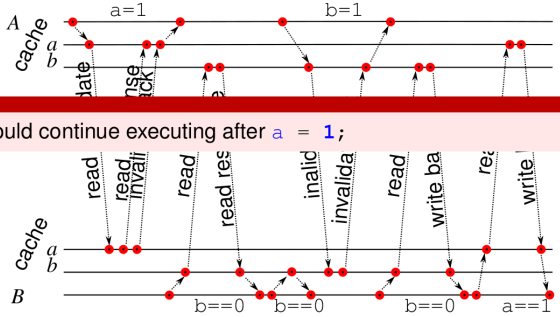
⚠ performance problem: writes always stall

Thread A

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

Thread B

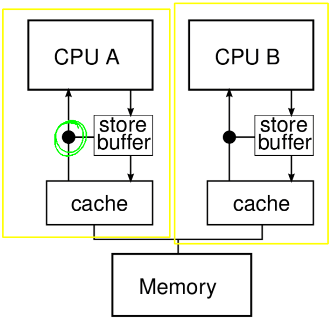
```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```



⇝ CPU **A** should continue executing after `a = 1;`

Store Buffers

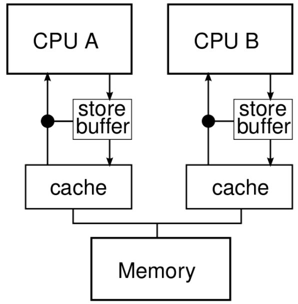
⚠ *Abstract Machine Model:* defines semantics of memory accesses



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

Store Buffers

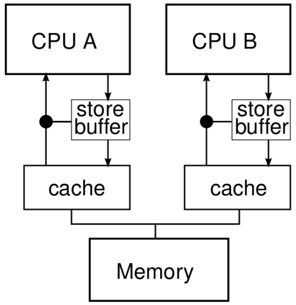
⚠ *Abstract Machine Model:* defines semantics of memory accesses



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

Store Buffers

⚠ *Abstract Machine Model:* defines semantics of memory accesses



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

TSO Model: Formal Spec [Sl92]

Definition (Total Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total

$$\forall_{a,b\,\in\,addr\ i,j\,\in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$

2. Stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \leq \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

3. Loads preceding an other operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Ld}_i[a] \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

4. A load's value is determined by the latest write as observed by the local CPU

$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = max_{\sqsubseteq}\,(\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

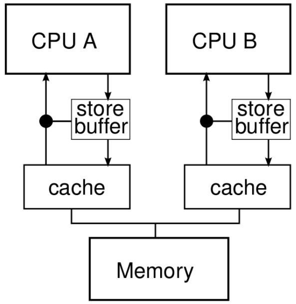
Particularly, one ordering property is not guaranteed:

$$\mathtt{St}_i[a] \leq \mathtt{Ld}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

⚠ Local stores may be observed earlier by local loads then from somewhere else!

Store Buffers

⚠ *Abstract Machine Model:* defines semantics of memory accesses



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

TSO Model: Formal Spec [SI92]

Definition (Total Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total

$$\forall_{a,b\,\in\, addr\ i,j\,\in CPU}\quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$

2. Stores in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

3. Loads preceding an other operation (wrt. program order $\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Ld}_i[a] \le \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

4. A load's value is determined by the latest write as observed by the local CPU

$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq}(\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \le \mathtt{Ld}_i[a]\}))$$

Particularly, one ordering property is not guaranteed:

$$\mathtt{St}_i[a] \le \mathtt{Ld}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

⚠ Local stores may be observed earlier by local loads then from somewhere else!

Happened-Before Model for TSO

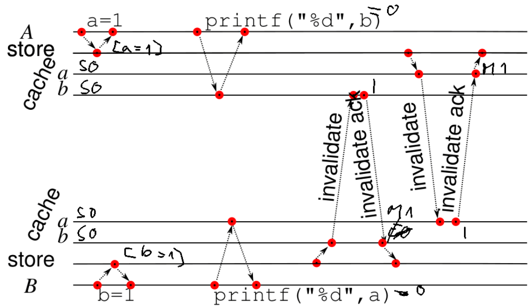
Thread A

```
a = 1;
printf("%d",b);
```

Thread B

```
b = 1;
printf("%d",a);
```

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



TSO in the Wild: x86

The x86 CPU, powering desktops and servers around the world is a common representative of a TSO Memory Model based CPU.

- FIFO store buffers keep quite strong consistency properties
- The major obstacle to Sequential Consistency is

$$\mathtt{St}_i[a] \le \mathtt{Ld}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

  - modern x86 CPUs provide the `mfence` instruction
  - `mfence` orders all memory instructions:

$$\mathtt{Op}_i \le mfence() \le \mathtt{Op}_i' \Rightarrow \mathtt{Op}_i \sqsubseteq \mathtt{Op}_i'$$

- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use fences only when necessary

Happened-Before Model for TSO

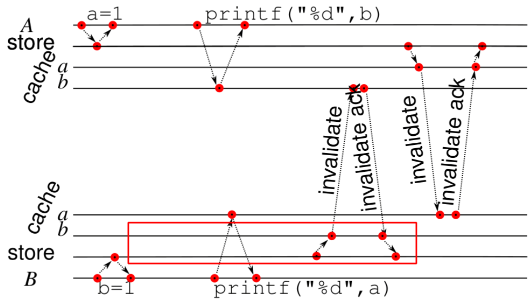
Thread A

```
a = 1;
printf("%d",b);
```

Thread B

```
b = 1;
printf("%d",a);
```

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



TSO in the Wild: x86

The x86 CPU, powering desktops and servers around the world is a common representative of a TSO Memory Model based CPU.

- FIFO store buffers keep quite strong consistency properties
- The major obstacle to Sequential Consistency is

$$\mathtt{St}_i[a] \le \mathtt{Ld}_i[b] \quad \not\Rightarrow \quad \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

  - modern x86 CPUs provide the `mfence` instruction
  - `mfence` orders all memory instructions:

$$\mathtt{Op}_i \le \mathit{mfence}() \le \mathtt{Op}_i{}' \quad \Rightarrow \quad \mathtt{Op}_i \sqsubseteq \mathtt{Op}_i{}'$$

- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use fences only when necessary

PSO Model: Formal Spec [Sl92]

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total

$$\forall_{a,b \in addr\ i,j \in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$

2. Fenced stores in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{sfence()} \le \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

3. Stores to the same address in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{St}_i[a]' \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[a]'$$

4. Loads preceding another operation (wrt. program order $\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Ld}_i[a] \le \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

5. A load's value is determined by the latest write as observed by the local CPU

$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \le \mathtt{Ld}_i[a]\}))$$

⚠ Now also stores are not guaranteed to be in order any more:

$$\mathtt{St}_i[a] \le \mathtt{St}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

⇝ What about sequential consistency for the whole system?

Explicit Synchronization: Write Barrier

Overtaking of messages *may be desirable* and does not need to be prohibited in general.

- generalized store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever a store in front of another operation in one CPU must be observable in this order *by a different CPU*, an explicit *write barrier* has to be inserted
  - 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