The MEBI 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:

M \rightarrow b \rightarrow E

S \rightarrow I

Introducing Caches: The MEBI Protocol

The MEBI 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:

M \rightarrow q \rightarrow E

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

S \rightarrow 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
- **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

---

---

---
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.

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
  - E: exclusive, with value x
  - S: shared, with value x
  - I: invalid

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

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>CPU A</th>
<th>CPU B</th>
<th>Read Invalidate</th>
<th>Invalidate Ack.</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>B.1</td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>B.1</td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>A.2</td>
<td>E0</td>
<td>E0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>E0</td>
<td>E0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>CPU A</th>
<th>CPU B</th>
<th>Writeback</th>
<th>Read of b</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>B.2</td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>A.1</td>
<td>S1</td>
<td>S1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>CPU A</th>
<th>CPU B</th>
<th>Invalidate</th>
<th>Assert a = 1</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>S1</td>
<td>S1</td>
<td>I</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>I</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A.1</td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>M1</td>
<td>I</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
MESI Example (II)

Thread A

\[ a = 1; \quad // \text{A.1} \]
\[ b - 1; \quad // \text{A.2} \]

Thread B

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

<table>
<thead>
<tr>
<th>statement</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>M1</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>S1</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>S1</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>A.1</td>
<td>S1</td>
<td>S1</td>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S1</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>M1</td>
<td>S1</td>
<td>I</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

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.
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**.

---

Summary: MESI cc-Protocol

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
Introducing Store Buffers: Out-Of-Order-Writes

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
```

---

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

Goal: continue execution after cache-miss write operation

- put each write into a store buffer and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes
  - today, a store buffer is always a queue [OSS09]
  - two writes to the same location are not merged
- !! sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the youngest value that is waiting to be written

What about sequential consistency for the whole system?
**Happened-Before Model for Store Buffers**

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

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

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

**Explicit Synchronization: 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
- x86 CPUs provide the \texttt{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
- a write barrier after each write gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)
- use (write) barriers only when necessary

**Happened-Before Model for Write Fences**

Thread A
\[
\begin{align*}
    a &= 1; \\
    \text{sfence}(); \\
    b &= 1;
\end{align*}
\]

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

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

**Introducing Invalidate Queues: O-O-O Reads**
Happened-Before Model for Write Fences

Thread A
\[
\begin{align*}
  a &= 1; \\
  s&fence(); \\
  b &= 1;
\end{align*}
\]

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

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

Introducing Invalidate Queues: O-O-O Reads

Invalidate Queue

Invalidation of cache lines is costly:
- all CPUs in the system need to send an acknowledge
- invalidating a cache line competes with CPU accesses
- a cache-intensive computation can fill up store buffers in other CPUs

Happened-Before Model for Invalidate Queues

Thread A
\[
\begin{align*}
  a &= 1; \\
  s&fence(); \\
  b &= 1;
\end{align*}
\]

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

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I
Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
- Intel x86 CPUs provide the mfence instruction
- a read barrier marks all entries in the invalidate queue
- the next read operation is only executed once all marked invalidations have completed
- a read barrier before each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)

match each write barrier in one process with a read barrier in another process

Happened-Before Model for Read Barriers

Thread A

\[
\begin{align*}
a &= 1; \\
sfence(); \\
b &= 1;
\end{align*}
\]

Thread B

\[
\begin{align*}
\text{while} \ (b == 0) \ {} & \\
\lfence(); \\
\text{assert}(a == 1);
\end{align*}
\]

Summary: Weakly-Ordered Memory Models

Modern CPUs use a weakly-ordered memory model:
- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
- most systems provide a barrier that is both, read and write (e.g. mfence on x86)
- ahead-of-time imperative languages can use memory barriers, but compiler optimizations may render programs incorrect
- use the volatile keyword in C/C++
- in the latest C++ standard, an access to a volatile variable will automatically insert a memory barrier
- otherwise, inline assembler has to be used

memory barriers are the “lowest-level” of synchronization

Example: The Dekker Algorithm on SMP Systems