Introducing Caches: The MESI Protocol

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 $M, E, S, I$:

- $M$: Modified
- $E$: Exclusive
- $S$: Shared
- $I$: Invalid

$M \rightarrow E$: 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 global state of cache lines is kept consistent by sending messages
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
- **Invalidation**: asks others to evict a cache line
- **Invalidation Acknowledge**: reply indicating that an address has been evicted
- **Read invalidation**: 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
assert(a == 1); // A.2
```
```
Thread B
while (b == 0) {}; // B.1
a = b; // B.2
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

MESI Example (I)

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

- `CPU A` column represents the CPU that has the data
- `CPU B` column represents the CPU that is requesting the data
- `RAM` column represents the RAM where the data is located
- `message` column represents the message that is sent between the CPUs

MESI Example (II)

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

- `CPU A` column represents the CPU that has the data
- `CPU B` column represents the CPU that is requesting the data
- `RAM` column represents the RAM where the data is located
- `message` column represents the message that is sent between the CPUs
MESI Example

Consider how the following code might execute:

```c
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

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**

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

**Thread B**

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

---

**Store Buffers and Total Store Ordering**

*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* [ROS9]
  - 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

---

**Happened-Before Model for Store Buffers**

**Thread A**

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

**Thread B**

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

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

---

What about sequential consistency for the whole system?
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 `fence` 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 Barriers

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

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-intense computation can fill up store buffers in other CPUs

≈ immediately acknowledge an invalidation and apply them later
- put each invalidate message into an invalidate queue
- if a MESI message needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated
- local read and writes do not consult the invalidate queue
- What about sequential consistency?
### Happened-Before Model for Invalidate Queues

**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

---

### 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 `lfence` 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)

\[\rightsquigarrow\] match each write barrier in one process with a read barrier in another process

---

### Happened-Before Model for Read Barriers

**Thread A**
```
a = 1;
sfence();
b = 1;
```

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

---

### 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

\[\rightsquigarrow\] memory barriers are the “lowest-level” of synchronization
Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```c
// flag[] is boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
turn = 0;  // or 1

P0:
flag[0] = true;
while (flag[1] == true)
    if (turn == 0) {
        flag[0] = false;
        while (turn != 0) {
            // busy wait
            flag[0] = true;
        }
    }
    // critical section
    turn = 1;
    flag[0] = false;

flag[1] = true;
while (flag[0] == true)
    if (turn == 1) {
        flag[1] = false;
        while (turn != 1) {
            // busy wait
            flag[1] = true;
        }
    }
    // critical section
    turn = 0;
    flag[1] = false;
```

The Idea Behind Dekker

Communication via three variables:
- flag[i]-true process P_i wants to enter its critical section
- turn=i process P_i has priority when both want to enter

P0:
flag[0] = true;
while (flag[1] == true)
    if (turn == 0) {
        flag[0] = false;
        while (turn != 0) {
            // busy wait
            flag[0] = true;
        }
    }
    // critical section
    turn = 1;
    flag[0] = false;

In process P_i:
- if P_i does not want to enter, proceed immediately to the critical section
- flag[i] is a lock and may be implemented as such
- if P_i also wants to enter, wait for turn to be set to i
- while waiting for turn, reset flag[i] to enable P_i to progress
- algorithm only works for two processes

Dekker's Algorithm and Weakly-Ordered MMs

Problem: Dekker's algorithm requires sequential consistency.
Idea: insert memory barriers between all variables common to both threads.

P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
    if (lfence(), turn != 0) {
        flag[0] = false;
sfence();
        while (lfence(), turn != 1){
            // busy wait
        }
        flag[0] = true;
sfence();
    }
    // critical section
    turn = 1;
sfence();
flag[0] = false; sfence();
```

- insert a load memory barrier lfence() in front of every read from common variables
Discussion

Memory barriers reside at the lowest level of synchronization primitives. Where are they useful?
- when several processes implement a automata and
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems
Why might they not be appropriate?
- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Summary

Memory consistency models:
- strict consistency
- sequential consistency
- weak consistency
Illustrating consistency:
- happened-before relation
- happened-before process diagrams
Intricacy of cache coherence protocols:
- the effect of store buffers
- the effect of invalidate buffers
- the use of memory barriers
Use of barriers in synchronization algorithms:
- Dekker’s algorithm
- stream processing, avoidance of busy waiting
- inserting fences
Future Many-Core Systems: NUMA

Many-Core Machines’ Read Responses congest the bus
In that case: Intel’s MESIF (Forward) to reduce communication overhead.

⚠️ But in general, Symmetric multi-processing (SMP) has its limits:
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

Overhead of NUMA Systems

Communication overhead in a NUMA system.
- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.
- A cache miss that cannot be satisfied by the local memory at A:
  - A sends a retrieve request to processor B owning the directory
  - B tells the processor C who holds the content
  - C sends data (or status) to A and sends acknowledge to B
  - B completes transmission by an acknowledge to A

References

Intel.
An introduction to the intel quickpath interconnect.

Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.

Paul E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.

Scott Owens, Susmit Sarkar, and Peter Sewell.
A better x86 memory model: x86-TSO.