The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenney(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
- **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

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

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
- second execution of test b=0 stays within cache: no traffic
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

---

Store Buffers

Goal: continue execution after cache-miss write operation

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

---

CPU A
store buffer
cache

CPU B
store buffer
cache

---

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
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
  - *store buffer* is a *queue*
- ▼ sequential consistency per CPU is violated unless

---

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
  - *store buffer* is a *queue*
  - two writes to the same location are not merged

---

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
  - *store buffer* is a *queue*
  - two writes to the same location are not merged
- ▼ sequential consistency per CPU is violated unless
  - each read checks *store buffer* before cache
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
  - store buffer is a *queue*
  - 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

****

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*
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
- Intel x86 CPUs provide the `sfence` instruction

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

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
- a write barrier after each write gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)
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
- Intel 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)

Invalidation Queue

Invalidate cache lines is costly:
- all CPUs in the system need to send an acknowledge
- invalidating a cache line competes with CPU accesses

Invalidation 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-intense computation can fill up store buffers in other CPUs
Invalidate Queue

Invalidate 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

\[\sim\] 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

Invalidate Queue

Invalidate 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

\[\sim\] immediately acknowledge an invalidation and apply them later

- put each invalidate message into an invalidate queue

Invalidate Queue

Invalidate 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

\[\sim\] immediately acknowledge an invalidation and apply them later

- put each invalidate message into an invalidate queue

- local read and writes do not consult the invalidate queue
Invalidation 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-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?

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

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
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

---

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)
Happened-Before Model for Read Fences

Thread A
\[ a = 1; \]
\[ sfence(); \]
\[ b = 1; \]

Thread B
\[ \textbf{while} (b == 0) \{ \} \]
\[ lfence(); \]
\[ assert(a == 1); \]

Summary: Weakly-Ordered Memory Models

Modern CPUs use a \textit{weakly-ordered memory model}:

- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences

Summary: Weakly-Ordered Memory Models

Modern CPUs use a \textit{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 on barrier that is both, read and write (e.g. \texttt{mfence} on x86)
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 on 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
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 on barrier that is both read and write (e.g. `mfence` on x86)
- many threads 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

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

In process P_i:
- if `P_{i-1}` does not want to enter, proceed immediately to the critical section
- `~flag[i]` is a lock and may be implemented as such

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

In process P_i:
- if `P_{i-1}` 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-1}` also wants to enter, wait for `turn` to be set to i
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;

A Note on Dekker’s Algorithm

Dekker’s algorithm has the three desirable properties:
- **ensure mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will

applications for Dekker: implement a \((map \circ fold)\) operation concurrently

\[
\begin{align*}
\text{T acc = init();} \\
\text{for (int i = 0; i < c; i++) {} } \\
\text{<T,U> (acc,tmp) = f(acc,i);} \\
\text{g(tmp, i);} \\
\end{align*}
\]

A Note on Dekker’s Algorithm

Dekker’s algorithm has the three desirable properties:
- **ensure mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will

applications for Dekker: implement a \((map \circ fold)\) operation concurrently

\[
\begin{align*}
\text{T acc = init();} \\
\text{for (int i = 0; i < c; i++) {} } \\
\text{<T,U> (acc,tmp) = f(acc,i);} \\
\text{g(tmp, i);} \\
\end{align*}
\]

- accumulating a value by performing two operations \(f\) and \(g\) in sequence
A Note on Dekker’s Algorithm

Dekker’s algorithm has the three desirable properties:

- **mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will

applications for Dekker: implement a \((\text{map} \circ \text{fold})\) operation concurrently

\[
\begin{align*}
\text{T acc} &= \text{init}(); \\
\text{for} \ (\text{int} \ i = 0; \ i < c; \ i++) \{ \\
&\qquad \langle T, U \rangle \ (\text{acc}, \text{tmp}) = f(\text{acc}, i); \\
&\qquad g(\text{tmp}, i); \\
\}
\end{align*}
\]

- accumulating a value by performing two operations \(f\) and \(g\) in sequence
- the calculation in \(f\) of the \(i\)th iteration depends on iteration \(i - 1\)
- non-trivial program to parallelize

Concurrent Fold

Create an \(n\)-place buffer for communication between processes \(P_f\) and \(P_g\).

\[
\begin{align*}
\text{T acc} &= \text{init}(); \\
\text{Buffer}<\text{U}> \text{ buf} &= \text{buffer}<\text{T}>(n); \ // \text{some locked buffer} \\
\text{Pf}: \ &\text{for} \ (\text{int} \ i = 0; \ i < c; \ i++) \{ \\
&\qquad \langle T, U \rangle \ (\text{acc}, \text{tmp}) = f(\text{acc}, i); \\
&\qquad \text{buf.put}(`\text{tmp}); \\
\}
\end{align*}
\]

If \(f\) and \(g\) are similarly expensive, the parallel version might run twice as fast.

A Note on Dekker’s Algorithm

Dekker’s algorithm has the three desirable properties:

- **mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
- **free of starvation**: if a process wants to enter, it eventually will

applications for Dekker: implement a \((\text{map} \circ \text{fold})\) operation concurrently

\[
\begin{align*}
\text{T acc} &= \text{init}(); \\
\text{for} \ (\text{int} \ i = 0; \ i < c; \ i++) \{ \\
&\qquad \langle T, U \rangle \ (\text{acc}, \text{tmp}) = f(\text{acc}, i); \\
&\qquad g(\text{tmp}, i); \\
\}
\end{align*}
\]

- accumulating a value by performing two operations \(f\) and \(g\) in sequence
- the calculation in \(f\) of the \(i\)th iteration depends on iteration \(i - 1\)
- non-trivial program to parallelize
- idea: use two threads, one for \(f\) and one for \(g\)

Concurrent Fold

Create an \(n\)-place buffer for communication between processes \(P_f\) and \(P_g\).

\[
\begin{align*}
\text{T acc} &= \text{init}(); \\
\text{Buffer}<\text{U}> \text{ buf} &= \text{buffer}<\text{T}>(n); \ // \text{some locked buffer} \\
\text{Pf}: \ &\text{for} \ (\text{int} \ i = 0; \ i < c; \ i++) \{ \\
&\qquad \langle T, U \rangle \ (\text{acc}, \text{tmp}) = f(\text{acc}, i); \\
&\qquad \text{buf.put}(`\text{tmp}); \\
\}
\end{align*}
\]

If \(f\) and \(g\) are similarly expensive, the parallel version might run twice as fast.

But busy waiting is bad:

- the cores might be idle anyway: no harm done (but: energy efficiency?)
- \(f\) can generate more elements while busy waiting
Concurrent Fold

Create an \(n\)-place buffer for communication between processes \(P_f\) and \(P_g\).
\[
\text{T acc = init();}
\text{Buffer<U> buf = buffer<T>(n); // some locked buffer}
\]

\[
\text{Pf:}
\text{for (int i = 0; i < c; i++)}
\text{\{}
\text{\quad (acc, tmp) = f(acc, i);}
\text{\quad buf.put(tmp);} \\
\text{\}
\]

\[
\text{Pg:}
\text{for (int i = 0; i < c; i++)}
\text{\{}
\text{\quad T tmp = buf.get();}
\text{\quad g(tmp, i);} \\
\text{\}
\]

If \(f\) and \(g\) are similarly expensive, the parallel version might run twice as fast.

But busy waiting is bad:
- the cores might be idle anyway: no harm done (but: energy efficiency?)
- \(f\) can generate more elements while busy waiting
- \(g\) might remove items in advance, thereby keeping busy if \(f\) is slow

Generalization to \(\text{fold} \circ \text{fold}\)

Observation: \(g\) might also manipulate a state, just like \(f\).

\(\rightsquigarrow\) stream processing
- general setup in signal/data processing
- data is manipulated in several stages
- each stage has an internal state
- an item completed in one stage is passed on to the next stage

Use of Dekker’s algorithm:
- could be used to pass information between stages
- but: fairness of algorithm is superfluous

Generalization to \(\text{fold} \circ \text{fold}\)

Observation: \(g\) might also manipulate a state, just like \(f\).

\(\rightsquigarrow\) stream processing
- general setup in signal/data processing
- data is manipulated in several stages
- each stage has an internal state
- an item completed in one stage is passed on to the next stage

Use of Dekker’s algorithm:
- could be used to pass information between stages
- but: fairness of algorithm is superfluous
  - producer does not need access if buffer is full
  - consumer does not need access if buffer is empty
Generalization to $\mathit{fold} \circ \mathit{fold}$

Observation: $k$ might also manipulate a state, just like $f$.

- stream processing
- general setup in signal/data processing
- data is manipulated in several stages
- each stage has an internal state
- an item completed in one stage is passed on to the next stage

Use of Dekker's algorithm:
- could be used to pass information between stages
- but: fairness of algorithm is superfluous
  - producer does not need access if buffer is full
  - consumer does not need access if buffer is empty
- $\mathit{specialize}$ algorithm?

Dekker's Algorithm and Weakly-Ordered

Problem: Dekker's algorithm requires sequentially consistency.

Idea: insert memory barriers between all variables common to both threads.

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

A Note on Dekker's Algorithm

Dekker's algorithm has the three desirable properties:
- **ensure mutual exclusion**: at most one process executes the critical section
- **deadlock free**: the process will never wait for each other
Dekker's Algorithm and Weakly-Ordered

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

\[
P0: \quad \text{flag[0]} = \text{true}; \\
\text{sfence()}; \\
\text{while (lfence(), \text{flag[1]} == \text{true})} \\
\quad \text{if (lfence(), \text{turn} \neq 0)} \\
\qquad \text{flag[0]} = \text{false}; \\
\qquad \text{sfence()}; \\
\quad \text{while (lfence(), \text{turn} \neq 0)} \\
\qquad \quad \text{// busy wait} \\
\quad \text{flag[0]} = \text{true}; \\
\quad \text{sfence()}; \\
\text{// critical section} \\
\text{turn = 1;} \\
\text{sfence();} \\
\text{flag[0]} = \text{false};
\]

Discussion

Memory barriers lie at the lowest level of synchronization primitives.
Where are they useful?

- when several processes implement an automaton and ... 
- synchronization means coordinating transitions of these automata

- insert a read memory barrier `lfence()` in front of every write to common variables
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread

Discussion

Memory barriers lie at the lowest level of synchronization primitives.
Where are they useful?

- when several processes implement an automaton 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?