Memory Barriers: A Hardware view for Software Hackers

Paul E. McKenney

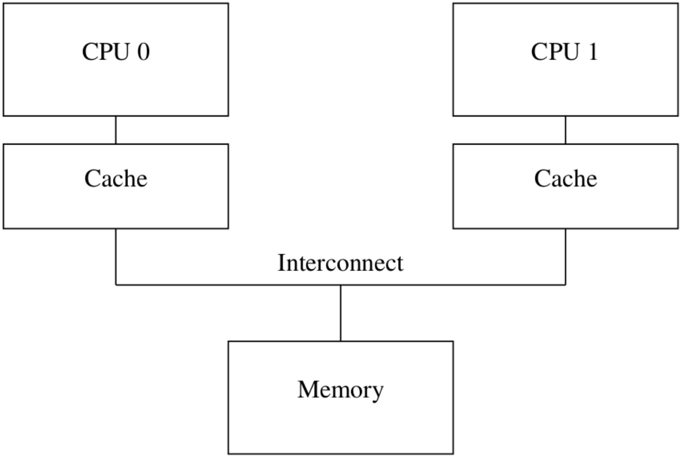
Presented by
Akshay Gopalakrishnan

December 4, 2021

Introduction

- This paper represents the inner working of hardware that results in several non-sequential behaviors of our concurrent programs
- The paper is rife with examples as well as showcasing the reasons for having such hardware features which in turn help in our programs performing better.
- Along with the positives the author also carefully cautions why such rampant changes for performance might result in highly non-trivial behaviors being showcased by the hardware running our programs.
- The paper concludes by discussing the then versions of several concurrent hardware that exhibit different non-sequential behaviors.

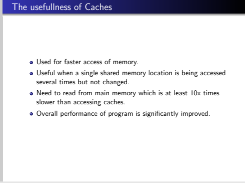
Cache structures



- An extra chunk of memory local to a given cpu (or multiple cpus in bigger systems).

The usefullness of Caches

- Used for faster access of memory.
- Useful when a single shared memory location is being accessed several times but not changed.
- Need to read from main memory which is at least 10x times slower than accessing caches.
- Overall performance of program is significantly improved.



The usefullness of Caches

- Used for faster access of memory.
- Useful when a single shared memory location is being accessed several times but not changed.
- Need to read from main memory which is at least 10x times slower than accessing caches.
- Overall performance of program is significantly improved.

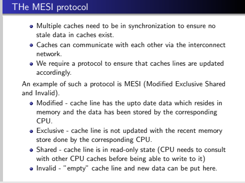
In modern systems there are multiple cache levels that exist, each of which is based on the scope. For instance, L1 cache is local to just one core. Whereas L2 is to multiple cores in the same processor (could also be others). All this layering is done for performance, part of the reason why we have such highly non-trivial behaviors of our concurrent programs.

THe MESI protocol

- Multiple caches need to be in synchronization to ensure no stale data in caches exist.
- Caches can communicate with each other via the interconnect network.
- We require a protocol to ensure that caches lines are updated accordingly.

An example of such a protocol is MESI (Modified Exclusive Shared and Invalid).

- Modified - cache line has the upto date data which resides in memory and the data has been stored by the corresponding CPU.
- Exclusive - cache line is not updated with the recent memory store done by the corresponding CPU.
- Shared - cache line is in read-only state (CPU needs to consult with other CPU caches before being able to write to it)
- Invalid - "empty" cache line and new data can be put here.


THe MESI protocol

- Multiple caches need to be in synchronization to ensure no stale data in caches exist.
- Caches can communicate with each other via the interconnect network.
- We require a protocol to ensure that caches lines are updated accordingly.

An example of such a protocol is MESI (Modified Exclusive Shared and Invalid).

- Modified - cache line has the upto date data which resides in memory and the data has been stored by the corresponding CPU.
- Exclusive - cache line is not updated with the recent memory store done by the corresponding CPU.
- Shared - cache line is in read-only state (CPU needs to consult with other CPU caches before being able to write to it)
- Invalid - "empty" cache line and new data can be put here.

Modified state is when my CPU is writiing to memory some data as well as duly updated the cache line. Now this line has the only copy of the latest data in memory. Exlcusive is one step behind modified state, wherein the CPU has updated memory, but not just its cache line. Shared state is when more than one cache line has the same data of memory. This only means, if I want to change one cache line, I must inform the others too. Invalid state can contain any stale data that is never going to be read by the CPU (as it is stale).

MESI Protocol Messages

The following messages are passed by a cache line to other caches in the system.

- Read - Contains the physical address of the cache line to be read.
- Read Response - Contains the data requested by a previous Read message.
- Invalidate - Contains the physical address of the cache line to be invalidated.
- Invalidate Acknowledge - CPU receiving an Invalidate message must respond with this message once the specfied cache line is invalidated.
- Read Invalidate - Does the action of both Read and Invalidate in one message.
- Writeback - Contains both the address and the data tobe written back to memory (could also implicitly update other cache lines with this).

Example of Cache Communication

The following table represents a sequence of actions done by CPU and the different states (MESI) of the caches after the action has been done.

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Writeback	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

Sequence 0 to 2

The 0th sequence represents the default state of cache before being used. Each cache line is set to "Invalid" state.
The 1st sequence represents a load done by CPU 0 to fetch data from address 0. The content in stored in CPU0 cache and the state changes to "Shared".
The 2nd sequences represents a load done by CPU 0 to fethc data from address 0. The content is stored in CPU1 cache and the state changes to "Shared".

Sequence 3 to 5

The 3rd sequence represents a load done by CPU 0 to fetch data from address 8. This implicity means to Invalidate one's own cache line by sending an invalidate message to it. This gives space to store data from address 0, which now is in the "Shared" state.
The 4th sequence represents an RMW done by CPU 2 on address 0. Before it can do this, it needs to invalidate other caches having data on this address by sending "Invalidate" message. Once that is done, it's own cache (CPU 2) is set to "Exclusive" state, while others which had data at address 0 to "Invalid".
The 5th sequence represents the actual store done by CPU 2 (as part of RMW). This changes its own cache to state "Modified" and sets the memory of address 0 to "Invalid".

Sequence 6 and 7

The 6th sequence represents CPU 1 performing an atomic write to data at Address 0. Since CPU 2 has its cache in modified state, the increment needs to change that value, while also invalidating their cache. So a "Read Invalidate" message is sent to each cache line, get the updated store value and increment it by 1. Now it sets its own cache line (CPU 1) to modified state. While other caches having data at address 0 is set ot invalidate.
The 7th sequence represents the actual commiting of the new data to memory. This can either be done by actually issuing a writeback or forcing the cache to make space for other address data (via a Load). Here, a Load is issued for address 0 by CPU 1, which forces a writeback to address 0. The value at memory address 0 is updated and the state becomes "Valid". Meanwhile the cache line of CPU 1 has a "Shared" state.

Example

Consider the example where CPU 0 wants to write to memory whose data is on cache line of CPU 1. This would require the following actions

- CPU 0 sends an Invalidate message to CPU 1.
- CPU 1 receives the message, sets the appropriate cache line to "Invalid" state.
- CPU 1 sends the Acknowledgement message along with the data on the cache line to CPU 0.
- CPU 0 receives the Acknowledgement and data.
- CPU 0 does the write to memory and updates its own cache.

Unnecessary stalls due to Writes

Notice that in the example above, CPU 0 needs to stall itself until the Acknowledgement message is received from CPU 1. This is showcased in the figure below.

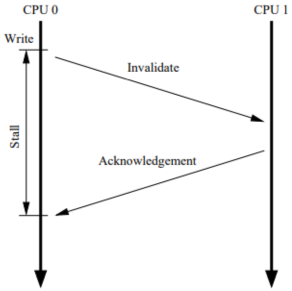


Figure 4: Writes See Unnecessary Stalls

CPU 0 need not stall as it will eventually do the write that it is supposed to do. Rather it can continue doing some other task.

Solution to write stalling: Write Buffers

One solution to this is to have a store buffer for each CPU. Writes to be done will be stored in this buffer and the CPU can continue doing future tasks. When the Acknowledgement messages are received by other CPUs, the write can be committed to its cache line (and eventually to memory).

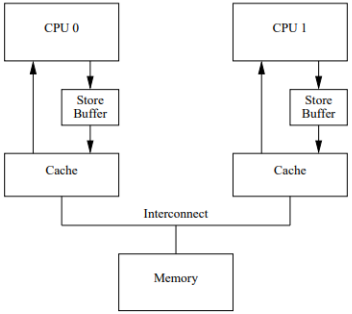


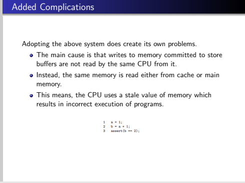
Figure 5: Caches With Store Buffers

Added Complications

Adopting the above system does create its own problems.

- The main cause is that writes to memory committed to store buffers are not read by the same CPU from it.
- Instead, the same memory is read either from cache or main memory.
- This means, the CPU uses a stale value of memory which results in incorrect execution of programs.

```
a = 1;
b = a + 1;
assert(b == 2);
```



Added Complications

Adopting the above system does create its own problems.

- The main cause is that writes to memory committed to store buffers are not read by the same CPU from it.
- Instead, the same memory is read either from cache or main memory.
- This means, the CPU uses a stale value of memory which results in incorrect execution of programs.

```
a = 1;
b = a + 1;
assert(b == 2);
```

Here is the reason why the example above fails. Consider only CPU 0 and 1 exist.

- CPU 0 wants to do the write $a = 1$. So it sends a Read Invalidate message to CPU 1 and commits the store to its Store buffer.
- CPU 1 receives the message and sends the invalidate with the current value of a from its cache.
- CPU 0 meanwhile wants to do $b = a+$, so starts reading a from its own cache line. It receives the value 0.
- Now CPU 0 receives the message from CPU 1 and the store buffer flushes the write to CPU 0 cache line.
- CPU 0 does $b = a + 1$ having read $a = 0$ from its cache before.
- Now $b = 1$.
- The assertion fails.

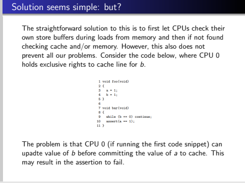
Solution seems simple: but?

The straightforward solution to this is to first let CPUs check their own store buffers during loads from memory and then if not found checking cache and/or memory. However, this also does not prevent all our problems. Consider the code below, where CPU 0 holds exclusive rights to cache line for *b*.

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

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

The problem is that CPU 0 (if running the first code snippet) can upadte value of *b* before committing the value of *a* to cache. This may result in the assertion to fail.



Solution seems simple: but?

The straightforward solution to this is to first let CPUs check their own store buffers during loads from memory and then if not found checking cache and/or memory. However, this also does not prevent all our problems. Consider the code below, where CPU 0 holds exclusive rights to cache line for *b*.

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

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

The problem is that CPU 0 (if running the first code snippet) can upadte value of *b* before committing the value of *a* to cache. This may result in the assertion to fail.

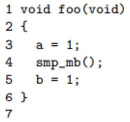
The reason why this would happen is the following steps

- CPU 0 wants to do $a = 1$, but a's cache line is not exclusive/modified, so it sends the write to store buffer and sends Read Invalidate to CPU 1.
- CPU 1 wants to do $while(b == 0)$ so tries to read value of b. It does not exist in its cache line, so sends a read to CPU 0.
- CPU 0 now wants to do $b = 1$ and since it owns this cache line, commits the write immedidately to cache.
- CPU 0 receives the read message from CPU 1 and sends $b = 1$ as a response to CPU 1.
- CPU 1 receives the read invalidate for a and sends Acknowledgement along with the value of a to CPU 0.
- CPU 1 then receives the value of b. Ends the loop (as $b = 1$) and moves to assertion.

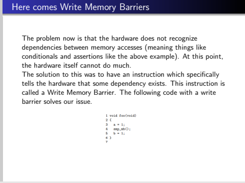
Here comes Write Memory Barriers

The problem now is that the hardware does not recognize dependencies between memory accesses (meaning things like conditionals and assertions like the above example). At this point, the hardware itself cannot do much.

The solution to this was to have an instruction which specifically tells the hardware that some dependency exists. This instruction is called a Write Memory Barrier. The following code with a write barrier solves our issue.



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



Here comes Write Memory Barriers

The problem now is that the hardware does not recognize dependencies between memory accesses (meaning things like conditionals and assertions like the above example). At this point, the hardware itself cannot do much.
The solution to this was to have an instruction which specifically tells the hardware that some dependency exists. This instruction is called a Write Memory Barrier. The following code with a write barrier solves our issue.

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

The reason inserting a write memory barrier there solves our problem is that CPU 0 will never do the wrtie $b = 1$ until the store buffer holding $a = 1$ is first committed to cache.

Need for Invalidate Queues

Added Complications

Example

Here comes Read Memory Barriers

Barrier instructions offered by Linux

Hardware Example: Alfa

Hardware Example: x86

Conclusion