Memory Consistency

You can't ignore it, you should fake it

Rekai González Alberquilla
AMD UK

16th May 2024

AMD
together we advance_

Outline

Conventions
And disclaimers, and such

- `Bold monospace` is used for code or linux commands.
- The pairs *load instruction* - *read operation*, *store instruction* - *write operation* are used interchangeably throughout the presentation.

Recomended bibliography

- Bruce Jacob, *"The Memory System. You can't avoid it, You can't ignore it, you can't fake it"*.
- Daniel Sorin, Mark Hill, David Wood, *"A Primer on Memory Consistency and Cache Coherence"*.

Both on Springer Synthesis Lectures on Computer Architecture: Affordable, probably available in your library, and offered in open-access every so often.

- Sarita Adve, Kourosh Garachorloo, *"Shared Memory Consistency Models: A Tutorial"*. IEEE Computer'96, Vol. 29, Issue 12.
- Shen and Lipasti, *"Modern Processor Design: Fundamentals of Superscalar Processors"*. McGraw-Hill, 2003.

Preamble

This talk assumes

- You are familiar with Cache memories.
- You are familiar with Cache coherence.
- You are familiar with Out-of-order execution and superscalar processors.
  - Branch prediction is orthogonal.
- All data in C/C++ is `volatile`-qualified.
- All memory reads/writes are 8-bytes long, aligned.

Preamble

This talk assumes

- You are familiar with Cache memories.
- You are familiar with Cache coherence.
- You are familiar with Out-of-order execution and superscalar processors.
    - Branch prediction is orthogonal.
- All data in C/C++ is `volatile`-qualified.
- All memory reads/writes are 8-bytes long, aligned.

Consistency vs Coherence, often put together, two totally different beasts.

- Coherence
    - Transparent to the programmer.
    - Only HW designers and low level programmers care about it.
- Consistency
    - Part of the architecture (implication on instruction semantics).
    - Has an impact beyond HW and low level SW.

Definitions

- Parallel: Two events, A, and B, are parallel iff there is no order relationship between them.
- Static instruction: A sequence of bytes that adheres an element of the ISA. Defined by the value of the bytes.
- Dynamic instruction: An instance of a static instruction, defined by the static instruction as well as the execution context.

Definitions

- Parallel: Two events, A, and B, are parallel iff there is no order relationship between them.
- Static instruction: A sequence of bytes that adheres an element of the ISA. Defined by the value of the bytes.
- Dynamic instruction: An instance of a static instruction, defined by the static instruction as well as the execution context.

```
b: b.eq r0, r1, end
   fadd [r3], d0
   add r3, #8
   inc r0
   jmp b
end:
```

```
b.eq r0, r1, end # r0 = 3, r1 = 5
fadd [r3], d0    # r3 = 0x10018, d0 = 2.0
add r3, #8       # r3 = 0x10018 | r3 <- 0x10020
inc r0           # r0 = 3       | r0 <- 4
jmp b
b.eq r0, r1, end # r0 = 4, r1 = 5
fadd [r3], d0    # r3 = 0x10020, d0 = 2.0
add r3, #8       # r3 = 0x10020 | r3 <- 0x10028
inc r0           # r0 = 4       | r0 <- 5
jmp b
...
```

Definitions

- Parallel: Two events, A, and B, are parallel iff there is no order relationship between them.
- Static instruction: A sequence of bytes that adheres an element of the ISA. Defined by the value of the bytes.
- Dynamic instruction: An instance of a static instruction, defined by the static instruction as well as the execution context.

- Speculative: means executing without waiting for **memory order** guarantee, as opposed to **control order** guarantee.
- $L(A)$: Load/Read `Mem[A]`.
- $S(B)$: Store/Write `Mem[B]`.
- $A \neq B \Rightarrow Mem[A] \cap Mem[B] = \Phi$.

Definitions:Order relationships
All about not being one and only one order

- Program order: Is the order relationship between **dynamic instructions** implied by the trace (sequential execution) of the program, $<_p$.
- Coherence order: Is the order relationshipt between **dynamic instructions** implied by the cache coherence protocol.
- Memory order: Is the order in which (memory) operations arrive to the shared global memory, $<_m$.

The rest of the talk is about when, given two instructions I_i, I_j, $I_i <_p I_j \Rightarrow I_i <_m I_j$.

Definitions:Order relationships

All about not being one and only one order

- Program order: Is the order relationship between **dynamic instructions** implied by the trace (sequential execution) of the program, $<_p$.
- Coherence order: Is the order relationshipt between **dynamic instructions** implied by the cache coherence protocol.
- Memory order: Is the order in which (memory) operations arrive to the shared global memory, $<_m$.

The rest of the talk is about when, given two instructions I_i, I_j, $I_i <_p I_j \Rightarrow I_i <_m I_j$.

- Fence: the ultimate tool. A Fence is a special instruction such that: for any two instructions I_i, I_j, for any two fences F_u, F_v
  - $I_i <_p F_u \Rightarrow I_i <_m F_u$
  - $F_u <_p I_j \Rightarrow F_u <_m I_j$
  - $F_u <_p F_v \Rightarrow F_u <_m F_v$

i.e., fences enforce order.

- There can be fences that provide only a subset of the guarantees.

Motivating example
Why do we want a consistency model?

Assuming `P0` and `P1` are processors in a shared memory system.

```
Mem[X] := 0
Mem[Y] := 0
```

```
  [P0]
St [X], 0x1 # S(X)
St [Y], 0x1 # S(Y)
```

```
  [P1]
Ld r0, [Y] # L(Y)
Ld r1, [X] # L(X)
```

Which of the following are possible outcomes?

Knowing what is allowed and what is not is needed to validate HW and SW.

Motivating example

Why do we want a consistency model?

Assuming P0 and P1 are processors in a shared memory system.

```
Mem[X] := 0
Mem[Y] := 0
```

```
[P0]
St [X], 0x1 # S(X)
St [Y], 0x1 # S(Y)
```

```
[P1]
Ld r0, [Y] # L(Y)
Ld r1, [X] # L(X)
```

Which of the following are possible outcomes?

- `[P1] r0 = 1, r1 = 1`

Knowing what is allowed and what is not is needed to validate HW and SW.

AMD together we advance_

Motivating example

Why do we want a consistency model?

Assuming P0 and P1 are processors in a shared memory system.

```
Mem[X] := 0
Mem[Y] := 0
```

```
[P0]
St [X], 0x1 # S(X)
St [Y], 0x1 # S(Y)
```

```
[P1]
Ld r0, [Y] # L(Y)
Ld r1, [X] # L(X)
```

Which of the following are possible outcomes?

- `[P1] r0 = 1, r1 = 1`
- `[P1] r0 = 0, r1 = 1`

Knowing what is allowed and what is not is needed to validate HW and SW.

AMD together we advance_

Motivating example
Why do we want a consistency model?

Assuming `P0` and `P1` are processors in a shared memory system.

```
Mem[X] := 0
Mem[Y] := 0
```

```
 [P0]
St [X], 0x1 # S(X)
St [Y], 0x1 # S(Y)
```

```
 [P1]
Ld r0, [Y] # L(Y)
Ld r1, [X] # L(X)
```

Which of the following are possible outcomes?

- `[P1] r0 = 1, r1 = 1`
- `[P1] r0 = 0, r1 = 1`
- `[P1] r0 = 1, r1 = 0`

Knowing what is allowed and what is not is needed to validate HW and SW.

Motivating example

Why do we want a consistency model?

Assuming `P0` and `P1` are processors in a shared memory system.

```
Mem[X] := 0
Mem[Y] := 0
```

```
[P0]
St [X], 0x1 # S(X)
St [Y], 0x1 # S(Y)
```

```
[P1]
Ld r0, [Y] # L(Y)
Ld r1, [X] # L(X)
```

Which of the following are possible outcomes?

- `[P1] r0 = 1, r1 = 1`
- `[P1] r0 = 0, r1 = 1`
- `[P1] r0 = 1, r1 = 0`
- `[P1] r0 = 0, r1 = 0`

Knowing what is allowed and what is not is needed to validate HW and SW.

What is memory Consistency?

Memory consistency is a contract between a programmer and a system that describes which outcomes are possible for a particular program.

- **Strict consistency**: writes to memory are instantaneously observable by all processors in the system. *Purely theoretical*.
- **Sequential consistency** (Lamport, 1979): *"The result of any execution is the same as if the (read and write) operations of all processes on the data store were executed in some sequential order,and the operations of each individual processor appear in this sequence in the order specified by its program"* (MIPS R10000).
- **Release consistency**
  - **TSO**, strong: Younger loads may be reordered w.r.t. older stores. *I.e.* writes can be buffered locally (x86).
  - **RMO**, weak: A read or write may be reordered w.r.t. any other read or write to a different *location* (Power, Arm).

What does that mean?
The informal view

```
Mem[X] := 0
Mem[Y] := 0
Mem[Z] := 0
```

```
[P0]
St [X], 0x1
St [Y], 0x1
Ld r0, [Z]
```

```
[P1]
St[Z], 0x1
Ldr0, [Y]
Ldr1, [X]
```

P0.r0	P1.r0	P1.r1	SC	TSO	RMO
0	0	0	N	Y	Y
0	0	1	N	Y	Y
0	1	0	N	N	Y
0	1	1	Y	Y	Y
1	0	0	Y	Y	Y
1	0	1	N	Y	Y
1	1	0	N	N	Y
1	1	1	Y	Y	Y

Push towards stricter consistency

Or why did you say we care about this?

```cpp
class Peterson {
    bool _request[2]{false, false};
    int _yields{0};
  public:
    void lock(int me) {
        _request[me] = true;
        _yields = me;
        while (_request[1 - me] && _yields == me)
            continue;
    }
    void unlock(int me) {
        _request[me] = false;
    }
};
```

Push towards stricter consistency

Or why did you say we care about this?

```
class Peterson {
    bool _request[2]{false, false};
    int _yields{0};
  public:
    void lock(int me) {
        _request[me] = true;
        _yields = me;
        while (_request[1 - me] && _yields == me)
            continue;
    }
    void unlock(int me) {
        _request[me] = false;
    }
};
```

This algorithm is SC-sound but does not work in TSO or RMO

- Thinking SC is easier.

The (a bit more) formal view

What is the TSO guarantee?

No matter whether A and B are the same

- $L(A) <_p L(B) \Rightarrow L(A) <_m L(B)$: Loads cannot overtake loads.
- $L(A) <_p S(B) \Rightarrow L(A) <_m S(B)$: Stores cannot overtake loads.
- $S(A) <_p S(B) \Rightarrow S(A) <_m S(B)$: Stores cannot overtake stores.

AMD together we advance_

The (a bit more) formal view

What is the TSO guarantee?

No matter whether A and B are the same

- $L(A) <_p L(B) \Rightarrow L(A) <_m L(B)$: Loads cannot overtake loads.
- $L(A) <_p S(B) \Rightarrow L(A) <_m S(B)$: Stores cannot overtake loads.
- $S(A) <_p S(B) \Rightarrow S(A) <_m S(B)$: Stores cannot overtake stores.

- $L(A)$ yields the value written by

$$\max_{<_m} \{S(A) | S(A) <_m L(A) \text{ or } S(A) <_p L(A)\}$$

Loads take the value of

- If the $<_p$-latest store before the Load is $L(A) <_m S(A)$, then S(A) (Store-to-load-forwarding).
- Otherwise he $<_m$-latest store before the Load.

AMD
together we advance_

Back to the example

```
I1: W(_request[0], 1)
I2: W(_yields, 0)
I3: R(_request[1])
```

```
J1: W(_request[1], 1)
J2: W(_yields, 1)
J3: R(_request[0])
```

$\mathtt{I1} <_p \mathtt{I2} <_p \mathtt{I3}$

$\mathtt{J1} <_p \mathtt{J2} <_p \mathtt{J3}$

According to TSO

- $\mathtt{J3} <_m \mathtt{J1}$, and
- $\mathtt{I3} <_m \mathtt{I1}$ are valid outcomes

so $\mathtt{I3} <_m \mathtt{J3} <_m \mathtt{I1} <_m \mathtt{I2} <_m \mathtt{J1} <_m \mathtt{J2}$ is a valid outcome.

What about the good old flag based synchronisation?

```cpp
class Channel {
    bool _ready{0};
    int _data;
  public:
    int consume() {
        while (!_ready) {}
        int d = _data;
        _ready = false;
    }
    void produce(int v) {
        while (_ready) {}
        _data = v;
        _ready = true;
    }
};
```

```
C1: R0 <- R(_ready)
C2: BIfZero R0, C1
C3: R1 <- R(_data)
C4: W(_ready, 0)
```

```
P1: R0 <- R(_ready)
P2: BIfNotZero R0, P1
P3: W(_data, R1)
P4: W(_ready, 1)
```

Flags in TSO

Why does it work?

```
C1: R0 <- R(_ready)
C2: BIfZero R0, C1
C3: R1 <- R(_data)
C4: W(_ready, 0)
```

```
P1: R0 <- R(_ready)
P2: BIfNotZero R0, P1
P3: W(_data, R1)
P4: W(_ready, 1)
```

- $\mathtt{C1}_n <_p \mathtt{C3} \Rightarrow \mathtt{C1}_n <_m \mathtt{C3}$
- $\mathtt{C3} <_p \mathtt{C4} \Rightarrow \mathtt{C3} <_m \mathtt{C4}$
- $\mathtt{P1}_n <_p \mathtt{P3} \Rightarrow \mathtt{P1}_n <_m \mathtt{P3}$
- $\mathtt{P3} <_p \mathtt{P4} \Rightarrow \mathtt{P3} <_m \mathtt{P4}$

Putting all together +<2> If $\mathtt{C1}_i$ observes `P4`, *i.e.* $\mathtt{P4} <_m \mathtt{C1}_i$, then it is guaranteed by TSO that $\mathtt{P3} <_m \mathtt{C3}$, *i.e.* `_data` gets the expected `v`. +<3> If $\mathtt{P1}_i$ observes `C4`, *i.e.* $\mathtt{C4} <_m \mathtt{P1}_i$, then it is guaranteed by TSO that $\mathtt{C3} <_m \mathtt{P3}$, *i.e.* `_data` is not overwritten before being read.

High performance techniques

- We want to do as much Out-of-order as needed to maximise ILP/MLP extraction.
- Break name dependencies, honour real dependencies.
- Execute instructions as soon as operands are ready. Commit in order to preserve precise exceptions.

What does consistency have to do with all this?

- The moment there are more than one threads sharing the memory out-of-order execution needs to be done carefully.

Four techniques, as described in [Shen and Lipasti] and [Sorin *et al*]

- Load-load bypassing: Executing a younger load ahead of an older load waiting.
- Load-store bypassing: Executing a younger load ahead of an older store waiting.
- Store-Load Forwarding: Sending the data from an older store to a younger load.
- Store coalescing: Merging writes before sending them to memory.

Some code to reason on

Who does not love I-DAXPY?

```
void idaxpy(double** x, double** y, double** z, double A) {
    for (int i = 0; i < N; ++i) {
        *(z[i]) = A * *(x[i]) + *(y[i]);
    }
}
```

```
Loop:
I1  Rx  <= Ld [Rpx]          ; x[i]
I2  F2  <= Ld [Rx]           ; *x[i]
I3  F2  <= Mul F2, F0        ; * A
I4  Ry  <= Ld [Rpy]          ; y[i]
I5  F4  <= Ld [Ry]           ; *y[i]
I6  F4  <= Add F4, F0        ; + *y[i]
I7  Rz  <= Ld [Rpz]          ; z[i]
I8  St F4, [Rz]              ; *z[i] =
I9  Rpx <= Add Rpx, 8        ; ++i
I10 Rpy <= Add Rpy, 8        ; ++i
I11 Rpz <= Add Rpz, 8        ; ++i
I12 Cmp Rpx, End
I13 Bneq Loop
```

Load-Load bypassing

We want loads executed ASAP

Scenario, n-th iteration

- `I2`$_n$`: F2 <= Ld [Rx]` misses in DL1 `@Rx`.
- We do not want to wait for the miss to do `I4`$_n$`: Ry <= Ld [Rpy]` and `I5`$_n$`: F4 <= Ld [Ry]`.
- So we execute `I4`$_n$ as soon as `I10`$_{n-1}$ is done, and `I5`$_n$ as soon as `I4`$_n$ is done.

Thoughts?

AMD
together we advance_

Load-Load bypassing

We want loads executed ASAP

Scenario, n-th iteration

- $\texttt{I2}_n$`: F2 <= Ld [Rx]` misses in DL1 `@Rx`.
- We do not want to wait for the miss to do $\texttt{I4}_n$`: Ry <= Ld [Rpy]` and $\texttt{I5}_n$`: F4 <= Ld [Ry]`.
- So we execute $\texttt{I4}_n$ as soon as $\texttt{I10}_{n-1}$ is done, and $\texttt{I5}_n$ as soon as $\texttt{I4}_n$ is done.

Thoughts?

- Doing so would violate $\texttt{I2}_n <_p \texttt{I4}_n \Rightarrow \texttt{I2}_n <_m \texttt{I4}_n$.
  - If we are in relaxed consistency that is okay.
  - If we are in TSO or SC that is not okay!
- One way to fake it is as R10000 did: The eviction or invalidation of a cache block squashes any load to that block and all subsequent instructions.
  - If $S_1 : S(X) <_p S_2 : S(Y) <_m L_1 : L(Y) <_p L_2 : L(X)$, both TSO and SC have to guarantee that if L_1 reads the value written by S_2, L_2 needs to read the value from S_1 (or younger).
  - If L_2 gets a value older than S_1 it is guaranteed to observe an invalidation from S_1 before the data from S_2 arrives.

Load-Store bypassing

We want loads executed ASAP

Scenario, n-th iteration

- $\texttt{I3}_{n-1}$`: F2 <= Mul F2, F0` is a long latency operation/$\texttt{I7}_{n-1}$`: Rz <= Ld [Rpz]` misses in DL1 `@Rz`.
- Stores cannot write memory until they are the oldest instruction.
- We don't want $\texttt{I1}_n$ ($\texttt{I2}_n$, $\texttt{I4}_n$, $\texttt{I5}_n$) to have to wait until $\texttt{I8}_{n-1}$ is the oldest instruction/retires.
- So we execute $\texttt{I1}_n$as soon as as $\texttt{I9}_{n-1}$ is done.

Thoughts?

Load-Store bypassing
We want loads executed ASAP

Scenario, n-th iteration

- $\texttt{I3}_{n-1}$`: F2 <= Mul F2, F0` is a long latency operation/$\texttt{I7}_{n-1}$`: Rz <= Ld [Rpz]` misses in DL1 `@Rz`.
- Stores cannot write memory until they are the oldest instruction.
- We don't want $\texttt{I1}_n$ ($\texttt{I2}_n$, $\texttt{I4}_n$, $\texttt{I5}_n$) to have to wait until $\texttt{I8}_{n-1}$ is the oldest instruction/retires.
- So we execute $\texttt{I1}_n$as soon as as $\texttt{I9}_{n-1}$ is done.

Thoughts?

- If `z[n-1] == x[n]` we have to do Store-Load Forwarding to preserve program semantics.
- If `z[n-1] != x[n]`, doing so would violate $\texttt{I7}_{n-1} <_p \texttt{I1}_n \Rightarrow \texttt{I7}_{n-1} <_m \texttt{I1}_n$.
  - If we are in relaxed consistency or TSO that is okay.
  - If we are in SC that is not okay!
- The same solution as in the previous case would handle it.

Load-Store bypassing
We want loads executed ASAP

Scenario, n-th iteration

- `I3`$_{n-1}$`: F2 <= Mul F2, F0` is a long latency operation/`I7`$_{n-1}$`: Rz <= Ld [Rpz]` misses in DL1 `@Rz`.
- Stores cannot write memory until they are the oldest instruction.
- We don't want `I1`$_n$ (`I2`$_n$, `I4`$_n$, `I5`$_n$) to have to wait until `I8`$_{n-1}$ is the oldest instruction/retires.
- So we execute `I1`$_n$as soon as as `I9`$_{n-1}$ is done.

Thoughts?

- What if `I7`$_{n-1}$ misses in DL1 `@Rz`?
  - This is a program semantics issue =) nothing new as far as consistency is concerned.
  - Enforcing issue in order handles the situation.

AMD together we advance_

Store coalescing

If stores overlap, let's send them together

Scenario, n-th iteration

- `*z[n-2]` and `*z[n]` lie consecutively in the same cache line.
- `*z[n-1]` lies in a different cache line.
- We want to minimise the amount of cache writes, so we use a buffer to coalesce the data and perform two writes to DL1 instead of 3.

Thoughts?

Store coalescing

If stores overlap, let's send them together

Scenario, n-th iteration

- `*z[n-2]` and `*z[n]` lie consecutively in the same cache line.
- `*z[n-1]` lies in a different cache line.
- We want to minimise the amount of cache writes, so we use a buffer to coalesce the data and perform two writes to DL1 instead of 3.

Thoughts?

- Doing so would violate $\mathtt{I8}_{n-2} <_p \mathtt{I8}_{n-1} <_p \mathtt{I8}_n \Rightarrow \mathtt{I8}_{n-2} <_m \mathtt{I8}_{n-1} <_m \mathtt{I8}_n$.
  - If we are in relaxed consistency that is okay.
  - If we are in TSO or SC that is not okay!
- There is literature handling it by delaying coherence responses to still enforce TSO.

Push towards more relaxed consistency

- This should make it obvious why HW designers may want relaxed consistency.
- Why do we just not do all relaxed consistency then?
    - Code is not portable towards more restricted consistency model.
    - Ordering is a responsibility of the programmer (or compiler).
        - Harder to write (correct) software.
        - Achived through *fences*.
    - Fences are instructions such that impose order. If F is a *fence*, and I and J are any two instructions
        - $I <_p F \Rightarrow I <_m F$.
        - $F <_p J \Rightarrow F <_m J$.
- The flipside of fences is that
    - The most straigtforward way to enforce a fence is squashing all subsequent instructions upon retire.
    - Fences may have implications as they travel the memory hierarchy.

Conclusions

- You cannot ignore your memory model
  - If you are making SW, you need to know what outcomes are possible, so your memory sharers communicate the way you want them to.
  - If you are making HW, you do not need to implement it, faking it is just enough, and enables better performance.

Copyright and Disclaimer