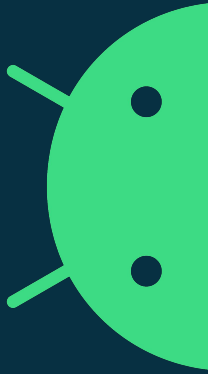


The never-ending saga of...

Control Dependencies

Linux Plumbers Conference, 2021

Will Deacon <will@kernel.org>



Mega thread alert!

<https://lore.kernel.org/r/YLn8dzbNwvqrqqp5@hirez.programming.kicks-ass.net>

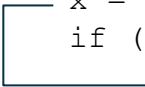
Thread overview: 122+ messages / `expand[flat|nested]` mbox.gz
2021-06-04 10:12 Peter Zijlstra [this message]

Let's see if we can make any sense of it...
(also see my LPC session last year)

Recap: What *is* a control dependency?

- The result of a *read* is used as input to a condition guarding a *write*
 - Ensures the write is ordered after the read (i.e. the write cannot be made visible to other CPUs until the condition has been resolved by the read)
 - Not all of the writes are annotated in practice
 - i.e. if there isn't a data race
- Used instead of (stronger) *acquire* memory barriers on some fast paths in the Linux Kernel
- Can be broken by the compiler
- Can be broken by the CPU

```
x = READ_ONCE(*foo);  
if (x > 42)  
    WRITE_ONCE(*bar, 1);
```



```
LDR  X0, [Xfoo]  
CMP  X0, #42  
B.LE 1f  
MOV  X1, #1  
STR  X1, [Xbar]
```

1:

- Read \Rightarrow write generally ordered by all CPU architectures
- Read \Rightarrow read control dependencies can often be reordered by hardware!

“Nice control dependency you got here. Be a shame if anything happened to it.” -- Al Capone

Breaking control dependencies: Mob boss #1

Compiler transformations

- Condition optimised away (evaluates to constant)
- Write occurs regardless of condition
- Conditional instructions
 - See later slide
- Speculative stores
 - Prevented by `-fno-allow-store-data-races?`
- Don't really feel like "real" code examples...
 - But *if* this goes wrong, it will be subtle and un-debuggable
 - Syntactic vs semantic dependencies
- See `memory-barriers.txt` for more examples

```
#define MAX      1

x = READ_ONCE(*foo);
if (x % MAX == 0)
    WRITE_ONCE(*bar, 1);
```

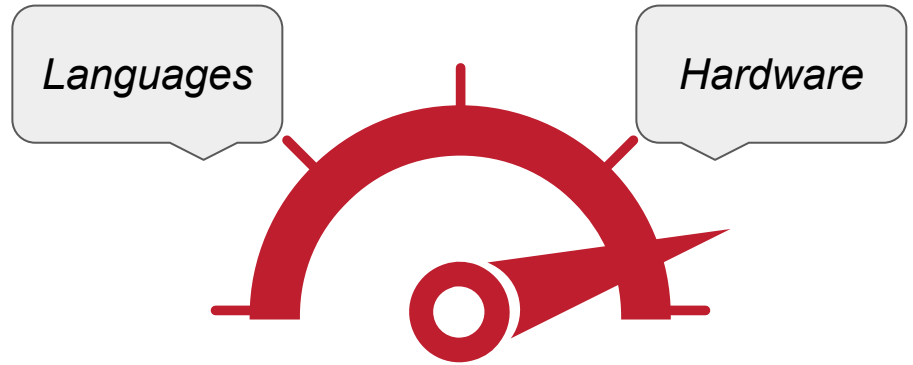
--->8

```
x = READ_ONCE(*foo);
if (x > 42) {
    WRITE_ONCE(*bar, 1);
    frob();
} else {
    WRITE_ONCE(*bar, 1);
    twiddle();
}
```

Breaking control dependencies: Mob boss #2

CPU reordering

- **Speculative stores**
 - Gives rise to “thin-air” values!
 - Value prediction?
- **Write occurs regardless of condition**
- **Conditional instructions**
- **Retrospective relaxation/clarification**
 - Treading on thin ice



Breaking control dependencies

CPU reordering on arm64

- **Speculative stores**
 - Thankfully doesn't happen yet!
- Write occurs regardless of condition
- Conditional instructions
 - Look, no conditional branch!
- Retrospective relaxation/clarification
 - "Pointed dependencies"
 - <https://lore.kernel.org/lkml/20210730172020.GA32396@knuckles.cs.ucl.ac.uk/>

```
x = READ_ONCE(*foo);
if (x > 42) {
    WRITE_ONCE(*bar, 1);
} else {
    WRITE_ONCE(*bar, 2);
}
WRITE_ONCE(*baz, 3);
```

--->8

```
LDR  X0, [Xfoo]
MOV  X1, #1
MOV  X2, #2
MOV  X3, #3

// X4 = X0 > 42 ? X1 : X2
CMP  X0, #42
CSEL X4, X1, X2, GT

STR  X4, [Xbar]
STR  X3, [Xbaz]
```

“You've got to ask yourself one question: 'Do I feel lucky about the compiler's instruction selection pass?’” -- Dirty Harry

Solution #1: `volatile_if()`

```
#define barrier()          asm volatile("" ::: "memory")

#define volatile_if(x) if ({
    _Bool __x = (x);
    BUILD_BUG_ON(__builtin_constant_p(__x));
    __x;
}) && ({ barrier(); 1; })
```

- Force the compiler to emit a conditional branch
 - Is it robust? 'x' can still be optimised and relies (at least) on `barrier()` being opaque.
 - Better-off as a compiler `__builtin`?
 - Not amenable to barrier-based (i.e. `smp_load_acquire()`) implementation
 - Disallow 'else' clause to solve *"Write occurs regardless of condition"* case?
- Unclear impact on codegen

Solution #2: Do nothing?

```
#define volatile_if(x)  if (x)
```

"I'd much rather have that kind of documentation, than have barriers that are magical for theoretical compiler issues that aren't real, and don't have any grounding in reality.

Without a real and valid example of how this could matter, this is just voodoo programming."

-- Linus Torvalds

Q: Will the issues remain theoretical forever?

Solution #3: Nuclear option

- **Barrier instructions exist *exactly for this purpose!***
 - An easy way out of the problem?
- **Per-architecture implementation**
- **Potential performance hit**
 - Requires annotation of the *load* instruction heading the dependency
 - Allow the condition to be optimised however the compiler likes
 - Applies to *all* relaxed accesses, even when dependencies are unused
- ***This is currently my preference for arm64***
 - Decreasing trust in robustness of dependency ordering
 - Further benchmarking on recent CPUs would provide an interesting data point



Aside: A better barrier() macro

- Prevent CSE from eliminating barrier() statements
 - GCC performs string comparison on the asm volatile block?
- Allow finer-grained control of access types (load/store) ordered by the barrier()
 - Load \Rightarrow Load/Store (*acquire-like*)
 - Load \Rightarrow Load (*rmb()*)
 - Load/Store \Rightarrow Store (*release-like*)
 - Store \Rightarrow Store (*wmb()*)



Thoughts?

Is this a real problem?

Is it worth solving?

Where/when/how should we solve it?

Thank you.

Recap: The sorry state of dependency ordering (LPC 2020)

- Hardware** CPU architectures **guarantee** that some dependencies enforce externally-visible ordering between memory accesses
- Performance** Dependency ordering is generally **cheaper** than using explicit fences, particularly where the dependency exists naturally as part of the algorithm.
- Linux** The **kernel relies on address/data dependency ordering** as a basis for RCU, but also control-dependency ordering to implement ring buffers and parts of the scheduler using volatile casts (`READ_ONCE/WRITE_ONCE`)
- C Compiler** No high-performance implementations exist of `memory_order_consume` and the **kernel does not follow the C11 memory model** anyway.