The AArch64 processor (aka arm64), part 14: Barriers

devblogs.microsoft.com/oldnewthing/20220812-00

August 12, 2022

Raymond Chen

Barriers are important on ARM-family systems because it has a weak memory model compared to the x86 series that most people are familiar with.

We start with the explicit barrier instructions:

```
    dmb     ish     ; data memory barrier
    dsb     ish     ; data synchronization barrier
    isb     sy      ; instruction synchronization barrier
```

The data memory barrier ensures that all preceding writes are issued before any subsequent memory operations (including speculative memory access). In acquire/release terms, it is a full barrier. The instruction does not stall execution; it just tells the memory controller to preserve externally-visible ordering. This is probably the only barrier you will ever seen in user-mode code.

The data synchronization barrier is a data memory barrier, but with the additional behavior of stalling until all outstanding writes have completed. This is typically used before changing memory mappings, such as during context switches, to ensure that any outstanding writes complete to the original memory before it gets unmapped.

The instruction synchronization barrier flushes instruction prefetch. This is typically used if you have generated new code, say by jitting it or paging it in from disk.

All of these barrier instructions take a parameter known as the *synchronization domain*. In practice, they will be the values I gave in the examples above.

There are some other niche barriers like the "consumption of speculative data barrier" (`CSDB`) and "physical speculative store bypass barrier" (`PSSBB`), which I won't bother going into because you're not going to see them.

By default, the memory access instructions do not impose any special ordering. But there are variations that let you request acquire or release semantics. We saw the general pattern in the bonus chatter last time:

- `A` – perform the load with acquire semantics
- `L` – perform the store with release semantics
- `AL` – perform the load with acquire semantics and the store with release semantics

The `AL` version applies only to load-modify-store instructions, which are all optional. But the acquire load and release store are supported by all processors.

```
; load acquire
ldarb   Wt/zr, [Xn/sp]          ; byte
ldarh   Wt/zr, [Xn/sp]          ; halfword
ldar    Rt/zr, [Xn/sp]          ; word or doubleword
; no register-pair version

; load acquire exclusive
ldaxrb  Wt/zr, [Xn/sp]          ; byte
ldaxrh  Wt/zr, [Xn/sp]          ; halfword
ldaxr   Wt/zr, [Xn/sp]          ; word or doubleword
ldaxp   Rt/zr, [Xn/sp]          ; pair

; store release
stlrb   Wt/zr, [Xn/sp]          ; byte
stlrh   Wt/zr, [Xn/sp]          ; halfword
stlr    Wt/zr, [Xn/sp]          ; word or doubleword
; no register-pair version

; store release exclusive
stlxrb  Ws/zr, Wt/zr, [Xn/sp]   ; byte
stlxrh  Ws/zr, Wt/zr, [Xn/sp]   ; halfword
stlxr   Ws/zr, Wt/zr, [Xn/sp]   ; word or doubleword
stlxp   Rs/zr, Wt/zr, [Xn/sp]   ; pair
```

These special acquire and release versions are handy in the load-locked/store-conditional pattern because they reduce the need for issue explicit barriers.

Here's how the gcc compiler generates the code:

```
    ; sequential consistency interlocked increment and
    ; acquire-release interlocked increment
@@: ldaxr   w8, [x0]                ; load acquire from x0
    add     w8, w8, 1               ; increment
    stlxr   w9, w8, [x0]            ; store it back with release
    cbnz    @B                      ; if failed, try again

    ; acquire-only interlocked increment
@@: ldaxr   w8, [x0]                ; load acquire from x0
    add     w8, w8, 1               ; increment
    stxr   w9, w8, [x0]             ; store it back (no release)
    cbnz    @B                      ; if failed, try again

    ; release-only interlocked increment
@@: ldxr    w8, [x0]                ; load (no acquire) from x0
    add     w8, w8, 1               ; increment
    stlxr   w9, w8, [x0]            ; store it back with release
    cbnz    @B                      ; if failed, try again

    ; relaxed interlocked increment
@@: ldxr    w8, [x0]                ; load from x0
    add     w8, w8, 1               ; increment
    stxr    w9, w8, [x0]            ; store it back
    cbnz    @B                      ; if failed, try again
```

On the other hand, the Microsoft compiler adds additional barriers:

```
    ; sequential consistency interlocked increment and
    ; acquire-release interlocked increment
@@: ldaxr   w8, [x0]                ; load acquire from x0
    add     w8, w8, 1               ; increment
    stlxr   w9, w8, [x0]            ; store it back with release
    cbnz    @B                      ; if failed, try again
    dmb     ish                     ; memory barrier (?)

    ; acquire-only interlocked increment
@@: ldaxr   w8, [x0]                ; load acquire from x0
    add     w8, w8, 1               ; increment
    stxr    w9, w8, [x0]            ; store it back
    cbnz    @B                      ; if failed, try again
    dmb     ish                     ; memory barrier (?)

    ; release-only interlocked increment
@@: ldaxr   w8, [x0]                ; load acquire from x0 (?)
    add     w8, w8, 1               ; increment
    stlxr   w9, w8, [x0]            ; store it back with release
    cbnz    @B                      ; if failed, try again

    ; no-fence interlocked increment
@@: ldxr    w8, [x0]                ; load from x0
    add     w8, w8, 1               ; increment
    stxr    w9, w8, [x0]            ; store it back
    cbnz    @B                      ; if failed, try again
```

Older versions of the Microsoft compiler used a spurious release on the `stlxr` when generating an acquire-only interlocked increment, but it appears to be fixed in 19.14. The spurious acquire on the release-only interlocked increment, and the mystery memory barrier instructions, are still there in 19.32.

Not sure what the extra barriers are for. Maybe there's something special about the Windows ABI that requires them? Maybe there's some subtlety in the architecture that I'm not aware of? I don't know.

While I'm here, I may as well mention this other instruction that isn't a barrier, but it's closely related:

```
    ; prefetch memory
    prfm    kind, [...]
    prfum   kind, [...]             ; force unscaled offset
```

The addressing mode can include pre- and post-increment.

The *kind* is a concatenation of a Type, Target, and Policy.

Category	Value	Meaning
Type	`PLD`	Prefetch for load
	`PLI`	Prefetch instruction
	`PLS`	Prefetch for store
Target	`L1`	L1 cache
	`L2`	L2 cache
	`L3`	L3 cache
Policy	`KEEP`	Temporal (load into cache normally)
	`STRM`	Streaming, non-temporal (data will be used only once)

For example, `PLDL3STRM` means “Prefetch for load into L3 cache for one-time use.”

Raymond Chen

Follow