What's up with `compare_exchange_weak` anyway?

Raymond C

March 29th, 2018

Last time, I left you with a homework assignment: Watch this video on `std::atomic`.

At time code 33:03, the presenter notes the weak version of compare-exchange (which is permitted to fail even if the value matches the expected value) and tries to reverse-engineer what kind of hardware would require this operation, eventually settling on a NUMA architecture where cross-node memory accesses can time out.

But there's no need to speculate about something that exotic, because the answer is all around us. In fact, it's probably happening right now on a computer in the presenter's pocket.

Most RISC processors do not have a compare-exchange instruction. Instead, they use a *load locked/store conditional* pattern. This pattern is employed by the ARM architecture, and we also saw it for Alpha AXP, and we'll see it later for MIPS and PowerPC.

The *load locked/store conditional* pattern goes like this:

- Issue a *load locked* instruction which reads a value from memory and instructs the processor to monitor that location for writes from other processors.
- Perform some computations.
- Issue a *store conditional* instruction which writes a value to the same memory location that was locked, provided the processor can prove that the memory has not been written to in the interim.

The conditional store can fail if another processor has written to the memory, or memory on the same cache line or other unit of monitoring granularity, or if the processor took an interrupt.

On an ARM, a strong compare-exchange contains a loop because the only way that `compare_exchange_strong` is permitted to fail is when the current value of the atomic variable does not match the expected value. If the failure reason was because of contention, then the strong version must perform an internal retry loop until the operation succeeds, or until the failure condition is met.

```
    ; r0 is the proposed new value
    ; r1 is the expected old value
    ; r2 is the address of the atomic variable

retry:
    DMB                     ; data memory barrier
    LDREX   r3, [r2]        ; load current value and lock it
    CMP     r3, r1          ; is it what we expected?
    BNE     fail            ; N: operation failed
                            ; actual current value is in r3

    STREX   r4, r0, [r2]    ; try to store new value
    CBNZ    r4, retry       ; lost the lock, try again
    DMB                     ; data memory barrier
```

Consider the compare-exchange loop in the code sample in the presentation:

```
    do { new_n->next = old_h; }
    while (!head.compare_exchange_strong(old_h, new_n));
```

The `compare_exchange_strong` has an embedded loop, and it’s part of another loop. So we have to generate two loops:

```
    ; r0 is new_n
    ; r1 is old_h
    ; r2 is the address of the atomic variable "head"

outer_loop:
    STR     r1, [r0]        ; new_n->next = old_h

retry:
    DMB                     ; data memory barrier
    LDREX   r3, [r2]        ; locked load of head
    CMP     r3, r1          ; is it what we expected?
    BNE     fail            ; N: operation failed

    STREX   r4, r0, [r2]    ; try to store new value
    CBNZ    r4, retry       ; lost the lock, try again

    DMB                     ; data memory barrier

    ; succeeded - continue with code that comes after

    ...

    ; This code goes at the end of the function because ARM
    ; statically predicts forward-jumps as not-taken.
fail:
    DMB                     ; data memory barrier
    MOV     r1, r3          ; old_h = current value of head
    B       outer_loop      ; restart the outer loop
```

The outer loop drives the loop written by the C++ programmer. The inner loop is the one required by `compare_exchange_strong`.

The weak version avoids this nested loop:

```
    do { new_n->next = old_h; }
    while (!head.compare_exchange_weak(old_h, new_n));
```

With this version, the compiler can simply bail out at the first sign of trouble. It avoids having to create a separate `fail` label and reduces register pressure because it doesn't need to carry the expected and actual values through the (no-longer present) inner loop.

```
    ; r0 is new_n
    ; r1 is old_h
    ; r2 is the address of the atomic variable "head"

outer_loop:
    STR     r1, [r0]        ; new_n->next = old_h

    MOV     r3, r1          ; save old_h before we overwrite
it
    DMB                     ; data memory barrier
    LDREX   r1, [r2]        ; locked load of head into old_h
    CMP     r3, r1          ; is it what we expected?
    BNE     outer_loop      ; N: retry with revised old_h

    STREX   r3, r0, [r2]    ; try to store new value
    CBNZ    r3, outer_loop  ; lost the lock, try again

    DMB                     ; data memory barrier

    ; succeeded - continue with code that comes after
```

When should you prefer the strong version of compare-exchange as opposed to the weak version? We'll take up that question next time.

Raymond Chen
Follow

Tagged Code

Read next

How do I choose between the strong and weak versions of compare-exchange?

It depends on how bad a spurious failure is for your algorithm.

Raymond C

March 30, 2018

0 comment

The MIPS R4000, part 1: Introduction

Here we go again.

Raymond C

April 2, 2018

0 comment

0 comments

Comments are closed. Login to edit/delete your existing comments