What is the purpose of the "PAUSE" instruction in x86?

Asked 8 years, 11 months ago Active 3 years, 1 month ago Viewed 12k times

66

25

I am trying to create a dumb version of a spin lock. Browsing the web, I came across a assembly instruction called "PAUSE" in x86 which is used to give hint to a processor that a spin-lock is currently running on this CPU. The intel manual and other information available state that

> The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops. The documentation also mentions that "wait(some delay)" is the pseudo implementation of the instruction.

The last line of the above paragraph is intuitive. If I am unsuccessful in grabbing the lock, I must wait for some time before grabbing the lock again.

However, what do we mean by memory order violation in case of a spin lock? Does "memory order violation" mean the incorrect *speculative* load/store of the instructions after spin-lock?

The spin-lock question has been asked on Stack overflow before but the memory order violation question remains unanswered (at-least for my understanding).

parallel-processing x86 x86-64 intel critical-section

Share Improve this question

Follow

edited Jan 8 '18 at 12:14

Flow

22.3k 13 91 148

asked Oct 15 '12 at 10:52

prathmesh.kallurkar

5,010 8 34 48

Link to the Intel docs: software.intel.com/en-us/download/... – ahcox Apr 23 '19 at 16:19

2 Answers

Active | Oldest | Votes

101

Just imagine, how the processor would execute a typical spin-wait loop:

```
1 Spin_Lock:
2    CMP lockvar, 0   ; Check if lock is free
3    JE Get_Lock
4    JMP Spin_Lock
5 Get_Lock:
```

After a few iterations the branch predictor will predict that the conditional branch (3) will never be taken and the pipeline will fill with CMP instructions (2). This goes on until finally another

processor writes a zero to lockvar. At this point we have the pipeline full of speculative (i.e. not yet committed) CMP instructions some of which already read lockvar and reported an (incorrect) nonzero result to the following conditional branch (3) (also speculative). This is when the memory order violation happens. Whenever the processor "sees" an external write (a write from another processor), it searches in its pipeline for instructions which speculatively accessed the same memory location and did not yet commit. If any such instructions are found then the speculative state of the processor is invalid and is erased with a pipeline flush.

Unfortunately this scenario will (very likely) repeat each time a processor is waiting on a spin-lock and make these locks much slower than they ought to be.

Enter the PAUSE instruction:

```
1 Spin_Lock:
2    CMP lockvar, 0   ; Check if lock is free
3    JE Get_Lock
4    PAUSE            ; Wait for memory pipeline to become empty
5    JMP Spin_Lock
6 Get_Lock:
```

The PAUSE instruction will "de-pipeline" the memory reads, so that the pipeline is not filled with speculative CMP (2) instructions like in the first example. (I.e. it could block the pipeline until all older memory instructions are committed.) Because the CMP instructions (2) execute sequentially it is unlikely (i.e. the time window is much shorter) that an external write occurs after the CMP instruction (2) read lockvar but before the CMP is committed.

Of course "de-pipelining" will also waste less energy in the spin-lock and in case of hyperthreading it will not waste resources the other thread could use better. On the other hand there is still a branch mis-prediction waiting to occur before each loop exit. Intel's documentation does not suggest that PAUSE eliminates that pipeline flush, but who knows...

Share Improve this answer Follow edited Oct 16 '12 at 10:06 answered Oct 15 '12 at 21:56

Mackie Messer
6,683 3 32 39

6

As @Mackie says, the pipeline will fill with `cmp` s. Intel will have to flush those `cmp` s when another core writes, which is an expensive operation. If the CPU doesn't flush it, then you have a memory order violation. An example of such a violation would be the below:

(This starts with lock1 = lock2 = lock3 = var = 1)

Thread 1:

```
spin:
cmp lock1, 0
jne spin
cmp lock3, 0 # lock3 should be zero, Thread 2 already ran.
je end # Thus I take this path
mov var, 0 # And this is never run
end:
```

Thread 2:

```
mov lock3, 0
mov lock1, 0
mov ebx, var # I should know that var is 1 here.
```

First, consider Thread 1:

if `cmp lock1, 0; jne spin` branch predicts that lock1 isn't zero, it adds `cmp lock3, 0` to the pipeline.

In the pipeline, `cmp lock3, 0` reads lock3 and finds out that it equal to 1.

Now, assume Thread 1 is taking its sweet time, and Thread 2 begins running quickly:

```
lock3 = 0
lock1 = 0
```

Now, let's go back to Thread 1:

Let's say the `cmp lock1, 0` finally reads lock1, finds out that lock1 is 0, and is happy about its branch predicting ability.

This command commits, and nothing gets flushed. Correct branch predicting means nothing is flushed, even with out-of-order reads, since the processor deduced that there is no internal dependency. lock3 isn't dependent on lock1 in the eyes of the CPU, so this all is okay.

Now, the `cmp lock3, 0`, which correctly read that lock3 was equal to 1, commits.

`je end` is not taken, and `mov var, 0` executes.

In Thread 3, `ebx` is equal to 0. This should have been impossible. This is the memory order violation that Intel must compensate for.

Now, the solution that Intel takes to avoid that invalid behavior, is to flush. When `lock3 = 0` ran on Thread 2, it forces Thread 1 to flush instructions that use lock3. Flushing in this case means that Thread 1 will not add instructions to the pipeline until all instructions that use lock3 have been committed. Before the Thread 1's `cmp lock3` can commit, the `cmp lock1` must commit. When the `cmp lock1` tries to commit, it reads that lock1 is actually equal to 1, and that the branch prediction was a failure. This causes the `cmp` to get thrown out. Now that Thread 1 is flushed, `lock3` 's location in Thread 1's cache is set to `0`, and then Thread 1 continues execution (Waiting on `lock1`). Thread 2 now get notified that all other cores have flushed usage of `lock3` and updated their caches, so Thread 2 then continues execution (It will have executed independent statements in the meantime, but the next instruction was another write so it probably has to hang, unless the other cores have a queue to hold the pending `lock1 = 0` write).

This entire process is expensive, hence the PAUSE. The PAUSE helps out Thread 1, which can now recover from the impending branch mispredict instantly, and it doesn't have to flush its

pipeline before branching correctly. The PAUSE similarly helps out Thread 2, which doesn't have to wait on Thread 1's flushing (As said before, I'm unsure of this implementation detail, but if Thread 2 tries writing locks used by too many other cores, Thread 2 will eventually have to wait on flushes).

An important understanding is that while in my example, the flush is required, in Mackie's example, it is not. However, the CPU has no way to know (It doesn't analyze code at all, other than checking consecutive statement dependencies, and a branch prediction cache), so the CPU will flush instructions accessing `lockvar` in Mackie's example just as it does in mine, in order to guarantee correctness.

Share Improve this answer Follow edited Aug 1 '18 at 20:45

answered Aug 1 '18 at 18:44

Nicholas Pipitone
3,589 2 20 35

I think it would be better to expand `wait(lock1)` and fill in `Get_Lock` . See https://wiki.osdev.org/Spinlock. Perhaps you can include more discussion on the memory order rules regarding loads and stores and that performing loads out of order is an implementation detail that necessitates the check for maintaining the order. – Hadi Brais Aug 1 '18 at 19:43

1 The main issue (I think) with Mackie's answer is that all loads are to the same location and belong to the same instruction. So realistically there would be no reordering in the first place. Having two different loads is a realistic example. – Hadi Brais Aug 1 '18 at 19:45

Branch misses still have to re-steer the front-end, even if they don't have to discard any uops from the out-of-order part of the core. I think the key thing is that branch misses are a lot cheaper (because the CPU has branch-order buffers) than memory order mis-speculations or other pipeline nukes, that fully flush the pipeline like an exception. i.e. branch-mispredicts are expected and optimized for. – Peter Cordes Aug 1 '18 at 20:41

This is still a pretty good example, but it's not how I'd explain it. I'm not sure it's all totally correct, but it does at least give the right idea of the basic problem that's being solved. – Peter Cordes Aug 1 '18 at 20:43

@PeterCordes Yeah, I was just putting the mispredict as an additional benefit, but I added the flushing note in the 2nd to last paragraph to be more explicit that that's the main gain here. How would you explain it? – Nicholas Pipitone Aug 1 '18 at 20:48