Intel memory ordering, fence instructions, and atomic operations.

Originally appeared at:
http://sites.google.com/site/peeterjoot/math2009/dekker.pdf?revision=2

Peeter Joot — peeter.joot@gmail.com
Dec 4, 2009 RCSfile : dekker.txt,v Last Revision : 1.5 *Date* : 2009 − 12 − 0504 : 08 : 17

Contents

1. Motivation.

In my recent post about memory barriers and atomics (or the PDF version here), I stated a number of times that x86 (or x86-64) was among the platforms that provided strongly ordered memory access.

I have an old friend, who we'll call Joe here, uses a Dekker's algorithm program to demonstrate to his computer architecture class that this is in fact not the case. This flies in the face of all my experience with DB2, so it becomes immediately interesting from both a theoretical point of view and a correctness issue for the mutual exclusion code that I maintain for our product, much of which I have written or rewritten myself over the years.

When talking to Joe, I had stated that I thought the MFENCE, LFENCE, and SFENCE instructions were only required for SSE. I said the Intel memory model already provided an ordering guarantee, and that no fence instructions were required. My mistaken recollection was that these were effectively no-ops for non-SSE code and special memory configurations used in privileged mode. The reasons for my misunderstanding is explored here, including why DB2 mutex implementation does not require these fences. I knew this once back in the days when the P6 was new, but I had forgotten all this processor-ordering business, demonstrated by an implementation of the Dekker algorithm on a recent Intel multiprocessor machine.

The fact that DB2 mutex and atomic code does not use memory fence instructions can likely be extrapolated to any other Intel targeting mutex implementation regardless of the products or library that providing the mutex. We all are restricted to the same tricks and toolbox, the Intel

instruction set specification itself, and we will see that the easiest and fastest way to implement a mutex does not require these fence instructions and why.

Dekker's algorithm uses a pair of flags and a turn variable to control access to a critical section. While this algorithm is of historical interest, and the subject of classroom discussions, I doubt that anybody uses it for anything real. There are a number of flaws, the most obvious of which is that it allows only the synchronization of a pair of threads. A less obvious limitation is that it assumes strongly ordered memory access. These strongly ordered memory access assumptions can be subdivided into two cases. The first of these ordering assumptions is that we require acquire and release fences to ensure the data guarded by the critical section is not accessed outside of the critical section. The second of the ordering assumptions is the trickier part, since there are memory ordering access assumptions in the algorithm itself. I would expect that these have been studied in depth by others, but even without such study, running the code with and without a few types of barrier instructions demonstrates conclusively that the algorithm is flawed as is, at least on Intel hardware.

Trolling the internet I found references stating that Dekker's algorithm can be faster than mutex methods build using the more modern hardware specific atomic instructions. Experiment shows that this is not necessarily true. To demonstrate this requires some low level knowledge. One cannot compare Dekker's to pre-canned blackbox library methods like pthread_mutex_lock, where the implementation details are unknown. With Dekker's algorithm doing brute force spinning, a good comparison requires that the mutex implementation also do so. Results comparing the Dekker's method and some alternatives will be supplied below. These are of general interest since the question of how expensive is an atomic compared to a mutex occurs frequently (at least in DB2 development).

2. What the Intel documentation says about ordering.

The Intel architecture documentation and instruction set reference do detail memory fence instructions. There are actually three such instructions in current CPUs, very much like PowerPC, which is intrinsically weakly ordered, and (except for the Power6 blip) always has been. Intel provides a bidirectional fence instruction MFENCE, an acquire fence LFENCE, and a release fence SFENCE. The MFENCE instruction dates way back to the P6 where Intel first weakened their memory ordering (defining their unique processor ordering).

Here's what the instruction set reference says about these fence instructions

- SFENCE [1]

  Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.

- LFENCE

  Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. This serializing operation guarantees that every load instruction that precedes in program order the LFENCE instruction is globally visible before

any load instruction that follows the LFENCE instruction is globally visible. The LFENCE instruction is ordered with respect to load instructions, other LFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to store instructions or the SFENCE instruction.

- MFENCE

  Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any SFENCE and LFENCE instructions, and any serializing instructions (such as the CPUID instruction).

A condensed and somewhat more human friendly version of these statements can be found in the system programming guide.

The functions of these instructions are as follows:

- SFENCE

  Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.

- LFENCE

  Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.

- MFENCE

  Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Now, we do not use MFENCE anywhere in DB2 code. It was once tried with the hope of trying to fix some code that had a mysterious bug, but it did not help. We replaced the problematic code with simpler more trustworthy stuff. Reading the Intel docs where there is no immediate sign that one would not have to use fence instructions for mutual exclusion code, one may think we avoid MFENCE since we prereq later hardware that supplies the LFENCE and SFENCE instructions and we use those instead. While the prereq statement is now probably true, but that is not why we do not use MFENCE. I will return to this later. For now lets instead explore some code.

3. The Dekker algorithm, implementation, and a non-correctness test case.

The Wikipedia Dekker's algorithm page [2] describes the algorithm. An implementation of the code described in that article, pretty much line for line except for some comment addition, is the following

```
1 struct FLAG_STATE
2 {
3     volatile int flag ;
4
5     char dummy[120] ; // cacheline separator.
```

```
6  } ;
7
8  FLAG_STATE g[2] ;
9  volatile int turn = 0 ;
10
11 void my_lock( int tid )
12 {
13    int other = 1 - tid  ;
14
15    g[tid].flag = true ; // I want in the CS.
16
17    FENCE ;
18
19    while ( true == g[other].flag )
20    {
21       //
22       // The other guy is in the CS or wants in.
23       //
24       if ( turn != tid )
25       {
26          // he said he wants it, but hasn't given me my turn yet.
27          //
28          // let him have a chance.
29          //
30          g[tid].flag = false ;
31
32          // wait till he gives me my turn
33          while ( turn != tid )
34          {
35          }
36
37          // he's about to give up his turn or has, so I can say I want it again.
38          //
39          // I'll now spin on his flag changing (though I could miss it if
40          // unlucky and have to start all over if he tries again fast).
41          //
42          g[tid].flag = true ;
43       }
44    }
45 }
46
47 void my_unlock( int tid )
48 {
49    int other = 1 - tid  ;
50
51    turn = other ;
52
53    g[tid].flag = false ;
54 }
```

This is called with tid values of 0 or 1 depending on which of the two contenders are trying to get into the mutual exclusion battle. Only two such contenders are allowed.

This algorithm is costly. Compared to a serially executing the increment instructions without locking or contention this took 76 times more time on the system I tried it on. This isn't exactly a fair comparison since executing the Dekker code without contention would also be faster.

Observe that there is no release barrier and the beginning of my_unlock and no acquire barrier at the end of my_lock, so one would expect this to fail on a weakly ordered platform like PowerPC, even if the algorithm itself guarantees mutual exclusion, and it is implemented properly. Failure here means allowing access of the data protected by the mutex outside of the mutual exclusion region.

If one presumes that the compiler has dumbly emitted the code in program order, and if one also presumes that this algorithm and the Wikipedia description of it is sound, and that it was implemented correctly, then one would expect this to work on a platform with strong memory ordering. On a platform with weak ordering acquire and release fences should be required for correctness, and possibly more.

Does this code work? This code was compiled without optimization, hoping that the compiler will not reorder anything, but I did not check the assembly listings to be sure. Unoptimized code is usually spectacularly bad, dumb and straightforward, so it is not entirely unreasonable to presume that the instructions are all generated in program order. Of a set of three runs with FENCE=MFENCE,SFENCE,LFENCE,NOFENCE where two threads looped incrementing an integer variable ten million times each, this code failed the consistency checking with LFENCE and NOFENCE. With SFENCE this code ran correctly twice, but in one of the iterations produced a count 7 less than the required target value for a mutex protected counter incremented a well known number of times. This does not demonstrate that this algorithm is correct with FENCE=MFENCE, but it is decisively busted otherwise.

Now, Joe had a different implementation of Dekker's that did not do the turn hand off. His looked like so (after tweaking it a bit to make it look like mine for consistency)

```
1  void my_lock( int tid )
2  {
3     g[tid].flag = 1 ;
4     turn = 1-tid ;
5
6     FENCE ;
7
8     while( g[1-tid].flag )
9     {
10       if ( turn == tid )
11       {
12          break ;
13       }
14    }
15 }
16
17 void my_unlock( int tid )
18 {
19    g[tid].flag = 0 ;
20 }
```

This code was also run with FENCE=MFENCE,SFENCE,LFENCE,NOFENCE. It passes with MFENCE and SFENCE, and fails with LFENCE and NOFENCE like the implementation of the Wikipedia listing.

Does this mean that his code is correct with SFENCE or even correct with MFENCE or the implementation of the wikipedia listing is correct with MFENCE? That is harder to say. The correctness of either set of code even with FENCE=MFENCE is not prove that the code is correct by a few successful test runs. What is interesting about all this is that Joe's assertion that memory

ordering is weakened enough to require fences for correctness now seems very plausible.

Note that this code is also not cheap. It is better than the listing originating with the vanilla wikipedia article, but was also clocked at 47 to 67 times slower than contended serial access. Again not entirely a fair comparison since this code would also do better serially.

We still have to understand why this is true here and not for atomic instruction based mutual exclusion. Additionally in many years of running DB2 on all varieties of Intel and amd64 hardware we haven't seen signs of requiring fence instructions for our non-mutex atomic code. That doesn't mean we do not have such problems, since it could be very hard to identify this sort of error. Keeping suspense high, the reasons for this a bit more, and instead show what is required to implement a mutex a more natural way on Intel.

4. Using an lock xchg mutex

Here is a dumb busy spinlock implementation on Intel using the GCC compiler

```
1  volatile char lock_word = 0 ;
2
3  void my_lock( int tid_ignored )
4  {
5     char old_value ;
6     char new_value = 1 ;
7
8     do {
9        __asm__ __volatile__( "lock; xchgb  %0,%1\n\t" :
10                             "=r" (old_value), "+m" (lock_word) :
11                             "0" (new_value) :
12                             "cc" ) ;
13
14    } while ( 1 == old_value ) ;
15 }
16
17 void my_unlock( int tid_ignored )
18 {
19    lock_word = 0 ;
20 }
```

Compared to unlocked serial uncontended increment, using this as a replacement for the Dekker mutex (ie. variations of that with enough fencing inserted to make the code appear functional), this runs about 16 times slower. This is already much better than the Dekker code (either the wikipedia based version or Joe's), but it would again be interesting to check how much worse that gets without contention. How much of this bus locked atomic exchange cost is aggravated by the contention itself.

5. Using an lock xadd for increment

For this simple task of doing thread safe increment, we can do much better easily. Here's some code to do the increment without using a mutex

```
1  volatile unsigned foo = 0 ;
2
3  inline void my_increment()
4  {
5     unsigned old_value ;
```

```
6     unsigned add_value = 1 ;
7
8     __asm__ __volatile__( "lock ; xaddl %0,%1\n\t" :
9                           "=r"(old_value), "+m" (foo) :
10                          "0" (add_value) :
11                          "cc" ) ;
12 }
13
14 void my_lock( int tid_ignored )
15 {
16 }
17
18 void my_unlock( int tid_ignored )
19 {
20 }
```

Executed serially a loop of such increments was measured at about 3 times slower than the plain old increment. That baseline timing actually used this code but with the bus locking signal flag removed. It costs us to do the bus locking even if we do not need it. Adding contention into the mix, the loop of these increments now clocks in at about 7-9 times slower than the unlocked uncontended code. This provides some rough idea of how much the correct code minimally costs us. All of these times will be summarized below.

6. Performance comparison.

Here's a summary of the costs of the various increment methods in order of added cost. This is with a plain old xadd as a baseline (no lock prefix).

- xadd

  This was the baseline time.

- lock xadd (ie. atomic add. No fences. No contention).

  3x baseline time.

- lock xadd (ie. atomic add. w/ contention)

  7-9x baseline time.

- lock xchg (spinlock mutex w/contention)

  16x baseline time.

- Joe's implementation of Dekker w/ [SM]FENCE (mutex w/ contention)

  47-67x baseline time.

- My implementation of the wikipedia Dekker listing w/ MFENCE (mutex w/ contention)

  76x baseline time.

7. Why no fences are required for the lock prefixed atomic or spinlock mutex code.

I have held off talking about what makes the lock xchg based code work without requiring fences. The reason for this can be found in volume 3, and is somewhat anticlimactic

8.2.2 "Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions."

The DB2 mutex code (and likely many other similar implementations) is fine as is. Our "atomic" library which also uses lock flagged atomic instructions of various sorts (xadd, cmpxchg, ...) also requires no additional or specific barriers as it does on ia64 or PowerPC. This atomic code MUST also use the lock prefix or else it is demonstrably busted when there is contention, so it imposes a minimum costs of approximately 3x in order to be safe when there is contention.

8. Conclusions.

I think that it is fair to say that the processor ordering model of modern Intel CPUs is demonstrably unordered requiring barriers in some cases. It would be interesting to identify exactly where in the Dekker algorithm there are ordering dependencies. I suspect such a study is already in the literature, although it is of little practical value since bus locked Intel instructions are superior.

While some timing differences were presented, these may just provide a ballpark performance comparison. Doing a performance comparison without optimization is of dubious value. It may possible that since only relative times were compared there may be some validity to the comparison otherwise, but that really requires doing it to be sure. It would be interesting to redo at least the xchg vs xadd vs baseline cases with optimization and see the differences.

The weaker aspects of Intel memory ordering are hidden from code that exploits their bus locked instructions for atomic and mutex library implementations. Requiring that any dependence on ordering is explicitly guarded by critical sections bounded by bus locked instructions is the key to correctness. As I write this it occurs to me that since not all of the DB2 mutex types use an atomic store for the lock release we may actually have an issue. That implicitly depends on ordering, so there may actually be a correctness issue to be investigated and this will have to be explored. Yet another reason not to write this sort of code oneself if it can be avoided. It is so easy to get it wrong and finding a demonstration that this one time in 20 million has blown up can be very hard.

References

[1] Intel. Intel architectures software developer's manuals.htm [online]. 2009. [Online; accessed 4-December-2009]. Available from: `http://www.intel.com/products/processor/index.htm`.

[2] Wikipedia. Dekker's algorithm — wikipedia, the free encyclopedia [online]. 2009. [Online; accessed 4-December-2009]. Available from: `http://en.wikipedia.org/w/index.php?title=Dekker%27s_algorithm&oldid=329540788`.