An attempt to illustrate differences between memory ordering and atomic access.

Originally appeared at:
http://sites.google.com/site/peeterjoot/math2009/atomic.pdf

Peeter Joot — peeter.joot@gmail.com
Oct 8, 2009 RCSfile : atomic.tex,v Last Revision : 1.12 *Date* : 2009/11/3002 : 59 : 03

Contents

Abstract. Discussion of problems and solutions associated with correct atomic and memory barrier usage has been a recurrent theme of many cubicle chats within DB2 development, and I will attempt to describe here part of what I've learned in some of those chats. As a software developer, not a hardware implementer, I cannot pretend to understand more than a fragment of this topic, but it is enough to be dangerous and perhaps help others be dangerous too.

1. Introduction

Use of atomic instructions for manipulating "word" (or smaller) size quantities, avoiding the use of mutual exclusion library functions is becoming increasingly easy for developers. This hasn't made the using atomic instructions or library methods correctly any less difficult or error prone.

Information on how to use these written for an average Joe developer is hard to come by. What can be easily found is good detailed low level information targeted at or written by operating system kernel extension writers or compiler developers [1], or language implementers and designers [2] [3] [4] [5] [6] [7] [8]. The driving reason for most atomic usage is lock avoidance, and a desire for performance drives

most use of atomic methods. When atomic operations are used to replace lock protected updates, this can change the semantics of the program in subtle ways that are difficult to understand or get right. In particular, additional use of memory barrier instructions or library methods may be required to correctly replace lock protected updates.

It is assumed here that most use of atomics is done for performance based lock (mutex) avoidance. An attempt to cheat, writing atomic based code that avoids a lock, unfortunately requires some understanding of what that lock provided in the first place. Here the meaning of, requirements for, and implementation of, a lock are discussed. The implementation of an atomic operation is then discussed, and we then move on to an enumeration of the types of memory barrier instructions and their effects. Finally, unless the assembly samples provided have not scared aways the reader, we cover a simple non-mutex example of barrier usage. This example uses an atomic to signal completion of a previous update and requires memory barriers for correctness on weakly ordered systems.

2. **Why do I need locking or atomics?**

Why would you want to use an atomic? If you are reading because you want to know a bit about memory barrier usage, you probably already know. However, for illustrative purposes, consider the very simplest example of non-thread-safe updates of an integer in shared memory.

```
1    pShared->x++ ;
```

Any such operation requires three steps. Read the value, change the value, update the value in memory. On some platforms this is explicit and obvious since the value will have to be read into a register before making the update. Here's a PowerPC example that illustrates this

```
1     lwz       r4=(sharedMem)(r3,0)     ; r3 == pShared ; (r3,0) == *pShared
2     addi      r0=r4,1
3     stw       (sharedMem)(r3,0)=r0
```

The value in the memory address contained in the register r3 (offset by zero bytes) is loaded into a register (r4). One is added to this (into r0), and the value is stored back into the original memory location. Running on a CISC system this may all appear as one instruction, but it still requires the same number of steps internally.

Suppose you had a threaded application where this counter variable is updated routinely by many threads as they complete some routine and commonplace action. What can now happen, interleaved (with time downwards) across threads may now look like this

```
1  T0: pShared->x = 0
2
3  T1: R1 = pShared->x (R1 = 0)
4      R1 = R1 + 1
5  T1: <timeslice expires>
6
7  T2: R2 = pShared->x
8      R2 = R2 + 1
9      pShared->x = R2
10
11 T3: R3 = pShared->x (assume T3 "sees" T2's update, a value of 1)
12     R3 = R3 + 1
13     pShared->x = R3
14
15 T4: R4 = pShared->x (assume T4 "sees" T3's update, a value of 2)
16     R4 = R4 + 1
17     pShared->x = R4
18
19 T1: <new timeslice.  Starts running again.>
```

```
20    pShared->x = R1 (T2 and T3's updates are now clobbered)
```

We had four updates to the counter after the initialization, but the counter value only goes up by one. Uncontrolled updates like this can oscillate in peculiar fashions, and something that may be expected to be monotonically increase perhaps only trends upwards on average. If the counter is not heavily contended you may even fluke out, running for a long time before getting unlucky with an inconsistent update of some sort.

This example leads us nicely to requirements for locking or atomic methods to protect such updates.

2.1. ***Use of locking to make the read, change update cycle safe.***

Mutual exclusion mechanisms go by many names, locks, mutexes, critical sections, and latches to name a few. System libraries usually provide implementations, a common example of which are the Unix pthread_mutex methods. The unsafe increment above can be made thread safe, protecting it with such a guard. With error handling omitted for clarity, a modified example could be as follows

```
1    pthread_mutex_lock( &m ) ;
2
3    pShared->x++ ;
4
5    pthread_mutex_unlock( &m ) ;
```

If all threads only ever updates this value we can use a similar bit of code to read and do something with the read value

```
1    pthread_mutex_lock( &m ) ;
2
3    v = pShared->x ;
4
5    pthread_mutex_unlock( &m ) ;
6
7    doSomethingWithIt( v ) ;
```

No two sequential samplings of the shared variable x would see the value go down was possible in the uncontrolled update case.

Now, you may ask why a mutex is required for the read. If that is just a single operation, why would it matter? That's a good question, but not easy to answer portably. If your variable is appropriately aligned and of an appropriate size (practically speaking this means less than or equal to the native register size) and you aren't compiling with funky compiler options that turn your accesses into byte accesses (such options do exist), then you may be able to do just that. However, if that variable, for example, is a 64-bit integer and you are running on a 32-bit platform, then this read may take two instructions and you have a risk of reading the two halves at different points in time. Similarly, suppose you were doing a 32-bit integer read, but that 32-bit integer was aligned improperly on a 16-bit boundary (on a platform that allows unaligned access), then your apparent single read may internally require multiple memory accesses (perhaps on different cache lines). This could cause the same sort of split read scenario.

You probably also need to declare your variable volatile and also have to be prepared to deal with a few other subtleties if lock avoidance on read is going to be attempted. By the end of these notes some of those subtleties will have been touched on.

To make the story short, it should be assumed that if you want portable correct results you need to take and release the mutex for both read and write.

2.2. ***What does correctness cost me?***

Now, having corrected the increment code, what does that cost us. Timing the following very simple single threaded program with an without the mutex code

```
1  #include <pthread.h>
2
3  int x = 0 ;
4  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ;
5
6  int inc() ;
7
8  int main()
9  {
10    for ( int i = 0 ; i < 1000000000 ; i++ )
11    {
12       inc() ;
13    }
14 }
15
16 int inc()
17 {
18 #if defined USE_THE_MUTEX
19    pthread_mutex_lock( &m ) ;
20 #endif
21
22    x++ ;
23
24    int v = x ;
25
26 #if defined USE_THE_MUTEX
27    pthread_mutex_unlock( &m ) ;
28 #endif
29
30    return x ;
31 }
```

I find that this billion sets of call a function, increment a variable and return it without the mutex takes about 2.5 seconds. With the mutex calls enabled, the total elapsed time required goes up to 8.5 seconds. It costs over three times the time to do it with a mutex, and that's without any contention on the mutex whatsoever! With threads all fighting over the mutex and blocking on kernel resources when they cannot get it, this is as good as it gets. Correctness doesn't come cheaply.

2.3. *How does the lock work?*

Any mutex implementation requires at least one platform specific method of performing a read, update, change cycle as if it is done without interruption. Your hardware will likely provide a Compare and swap or Load Link/Store Conditional (which can be used to construct higher level atomics like compare and swap). Implementing a mutex is intrinsically unportable territory. Older intel cpus did not provide a compare and swap, but one did have a bus locked exchange available (LOCK XCHG). HP's PARSIC, also easily argued to not be a modern architecture, only provides an atomic load and clear word instruction, also requiring 16 BYTE a aligned word. What can be assumed is that any multiple cpu machine provides some sort of low level atomic instruction suitable for building a mutual exclusion interface. If one were to imagine what a pthread_mutex_lock protected increment expands to, it would likely have the following form

```
1     T valueRead ;
2
3     while ( ATOMIC_OP_WAS_SUCCESSFUL != SOME_INSTRUCTIONS_FOR_LOCK_ACQUISION( &m ) )
4     {
5        // instructions (like PAUSE()) for hyper threaded cpus.
6
```

```
7         // code for wasting a bit of time hoping to get the lock
8         //   (when not running on a uniprocessor).
9
10        // sleep or block using an OS primitive
11    }
12
13    valueRead = pShared->myLockProtectedData ;
14    pShared->myLockProtectedData = valueRead + 1 ;
15
16    SOME_INSTRUCTIONS_FOR_LOCK_RELEASE( &m ) ;
```

Even before talking about the hardware, one has to assume that there are mechanisms available to the library writer that prevent the generated code from being reordered. Any instructions containing accesses of myLockProtectedData must NOT occur BEFORE the ACQUISION instructions, and must NOT occur AFTER the RELEASE instructions. If the compiler were to generate code that had the following effect

```
1    T valueRead = pShared->myLockProtectedData ;
2    pShared->myLockProtectedData++ ;
3
4    pthread_mutex_lock( &m ) ;
5    pthread_mutex_unlock( &m ) ;
```

Things would obviously be busted. In the case of the pthread functions we have non-inline method with unknown side effects requiring a call to an external shared library. This appears to be enough that the lock protected data does not have to be declared volatile and can be used while the mutex is held as if in a serial programming context. Understanding the murky language rules that give us the desired thread safe behavior is difficult (for an average Joe programmer like me). I came to the conclusion this is not a completely well defined area, motivating a lot of the recent C++ standards work into memory consistency and threaded behavior.

Presuming one assumes that the compiler is laying down the instructions for this code in program order (or something sufficiently close), or that there are mechanism available to ensure this, is this enough to guarantee that we have correct program behavior?

It may come as a rude surprise that, depending on the instructions used to acquire and release the mutex, having the all the instructions scheduled in program order is NOT actually sufficient. Not all hardware executes instructions in the order the compiler specifies. We also need mechanisms to prevent the hardware from starting a memory access too early or letting a memory access complete too late.

This requirement for memory ordering instructions is really the whole point of this discussion. We must have additional mechanism on top of the raw atomic instructions that only ensure read,change,update accesses on an isolated piece of memory (the lock word) can be made as if uninterpretable.

For illustrative purposes, suppose that one was using the gcc Atomic Builtins to attempt to implement a mutex. One (generally) wrong way to use these would be

```
1    // the mutex data
2    volatile int m = 0 ;
3    #define HELD_BIT 1
4    #define WAITER_BIT 2
5
6    // Get the mutex.
7    while ( true )
8    {
9        int prev = __sync_fetch_and_or( &m, HELD_BIT ) ;
10
11       if ( 0 == ( prev & HELD_BIT ) )
12       {
13           break ;
14       }
15       // sleep and other stuff, possibly blocking and setting WAITER_BIT.
```

```
16      }
17
18      // Got the mutex, access the protected data.
19      pShared->myLockProtectedData++ ;
20
21      // Release the mutex.
22      int preLockState = __sync_fetch_and_and( &m, ~HELD_BIT ) ;
23      assert( preLockState & HELD_BIT ) ;
24      if ( preLockState & WAITER_BIT )
25      {
26          WakeUpAnyWaiters() ;
27      }
```

Reading the GCC docs one sees that this has desired compiler behaviour for instruction scheduling. Namely, "these builtins are considered a full barrier. That is, no memory operand will be moved across the operation, either forward or backward". The compiler will not move instructions that access memory and move them into or out of the critical section bounded by the pair of atomic instructions.

Will the hardware retain that ordering? NO, not necessarily.

These compiler builtins are partially patterned after ones provided by intel, and intel designed them to match functionality provided by their ia64 architecture, a weakly ordered instruction set. While ia64 was effectively killed by AMD64 and Windows (now living on only in the guise of HP's IPF systems), there are other weakly ordered instruction sets predating ia64. Some PowerPC chips that you will find on big AIX systems, or in older macs, or in your childrens ps3 will be weakly ordered. The Sparc architecture allows for weak ordering, but the chips you find in sun machines implement their TSO (total store order) model which is mostly ordered.

This allowance for unordered memory is why, way down at the end of the GCC page, one finds a couple special _lock methods. The point of these is to ensure that the hardware does not reorder the relative order of memory accesses of the lock word and the lock data, even when the compiler lays down the code in "program order".

We can do a mutex implementation less wrongly using a memory barrier for release as in the following fragment.

```
1       // the mutex data
2       volatile int m = 0 ;
3       #define HELD_BIT 1
4
5       // Get the mutex.
6       while ( true )
7       {
8           int prev = __sync_fetch_and_or( &m, HELD_BIT ) ;
9
10          if ( 0 == ( prev & HELD_BIT ) )
11          {
12              break ;
13          }
14          // sleep and other stuff if you didn't get it
15      }
16
17      // Got the mutex, access the protected data.
18      pShared->myLockProtectedData++ ;
19
20      __sync_syncronize() ;
21
22      m = 0 ;
```

Observe that the use of the word barrier is overloaded on the GCC page, with some usage associated with compiler instruction scheduling, and other usage referring to memory ordering. Their syncronize

function appears to be both. Looking at the generated assembly for a weakly ordered system would verify this interpretation, and one would likely find an lwsync or sync instruction on PowerPC for example. This modification of the locking code ensures the compiler lays down the code with the clearing of the lock word after the protected data access, and the hardware memory barrier instruction should ensure that reads or writes to the protected data are complete before the store to the lock word is allowed to occur. Depending on the type of instruction that is emitted for the synchronize builtin, this may not actually prevent loads and stores that occur after the lock release from being started before the lock is released. Generally if that must be prevented a full barrier is required, and it would not surprise me if most implementations of library methods like pthread_mutex_unlock do not prevent memory accesses that occur after the lock release from being initiated before the lock release occurs. This sort of subtlety is not likely to be found in documentation for the mutex library functions.

With the sample code above we are still left with the possibility that the hardware will execute the memory access instructions for the lock protected data before the lock is acquired, making the lock useless. We can fix this with the _lock test and set and release builtins, and also remove the likely overzealous full memory barrier that synchronize provides (allowing post lock release memory access into the critical section). Sample code for this would may look like

```
1   // the mutex data
2   volatile int m = 0 ;
3   #define HELD_VALUE 1
4
5   // Get the mutex.
6   while ( true )
7   {
8      int wasHeld = __sync_lock_test_and_set( &m, HELD_VALUE ) ;
9
10     if ( !wasHeld )
11     {
12        break ;
13     }
14     // sleep and other stuff if you didn't get it
15  }
16
17  // Got the mutex, access the protected data.
18  pShared->myLockProtectedData++ ;
19
20  __sync_lock_release( &m ) ;
```

This now implements functional mutual exclusion code, even on weakly ordered systems. The compiler has provided methods that ensure it does not move the lock protected data accesses out of the critical section, we are using atomic instructions that guarantee other threads cannot also be modifying the data (presuming they all use the same mutex), and also have instructions that ensure the hardware does not inappropriately reorder the memory accesses. Inappropriate reordering means that accesses to the lock protected data remain after the lock acquisition and before the lock release. It may not mean that memory accesses from before the lock acquisition are done by the time the lock is acquired nor that memory accesses that occur after the lock is released occur after the lock is released (the compiler says it does that, but the hardware may be a sneaky bastard behind your back).

So, that's a mutex. We can use something like this to avoid some of the excessive cost observed when using the pthread library mutex functions, and would expect that the uncontended cost of the back to back increment code could be lessened.

Presuming you do not mind the unportability of such code, and manage to get it right, and want to live with maintaining your own mutex implementation, if you do a poor job dealing with contention, you may very well get worse performance than the system provided methods in the contended codepath.

You really really have to want the performance to go down the path of implementing your own mutual exclusion code. DB2 is such a product. DB2 is also a product where first born sacrifices in the name of per-

formance are not considered abhorrent. In our defense our mutex implementation predates the availability of widely available library methods. We had SMP exploitation before pthreads existed using mutiple processes and shared memory. Our own implementation also has the advantage of providing framework to build problem determination functionality that cannot be found in generic library implementations. Performance driven coding is not the only reason we do it ourself even in this more modern age where library methods do exist.

3. What is an atomic, and how does an atomic method work?

At the core of every atomic library method or compiler intrinsic is one or more special purpose instructions. In many cases these are the same types of instructions that can be used to implement mutual exclusion code (when properly augmented by memory barrier instructions).

If you don't have an instruction for exactly what you want, it can often be built from some other lower level operation. Suppose for example you want to safely do the contended increment without using a lock, but only have a compare and swap. Something like the following can be used

```
1  int fetch_and_add( volatile int *ptr, int value )
2  {
3     int v = *ptr ;
4     while ( true )
5     {
6        int prev = __sync_val_compare_and_swap( ptr, v, v + value ) ;
7        if ( prev == v )
8        {
9           break ;
10       }
11
12       v = prev ;
13    }
14
15    return prev ;
16 }
```

Generally, one can assume at least a compare and swap operation, but this may also be simulated, built up out of more direct and primitive operations. On PowerPC for example, this is done with the very flexible load-linked/store-condition instructions (load with reservation and store conditional in PowerPC lingo). A fetch and add can be built with a small loop

```
1  fadd32_spin:
2     lwarx     r0,0,r3       ; load word and acquire a registration
3     add       r0,r0,r4      ; add in the desired amount to the possibly current value
4     stwcx.    r0,0,r3       ; try to store it
5     bne-      fadd32_spin   ; if loaded value old or somebody beat us repeat.
6     sub       r3,r0,r4      ; fake a return using normal function return register r3
```

Building a portable dependence on a load-linked/store-condition primitive can be tricky, but possible. In DB2's reader-writer mutex implementation, a data structure employing some nifty lock free algorithms ([9] [10] [11] [12] [13] [14]) we use this load-linked/store-conditional code to implement a slick compare-against-bitmask-and-increment-or-set-bit atomic operation. We can do the same thing with compare and swap, but the generated assembly code is particular pretty on AIX, where PowerPC gives us this more powerful primitive.

3.1. *A historical reflection aside.*

Even only a couple of years ago, it was much harder to exploit atomic instructions. Part of the reason is that we have tossed or stopped caring about older systems. Nobody cares anymore about older ia32

systems that did not have cmpxchg. Pre-PowerPC AIX implementations that did not have lwarx and stwcx. instructions are now dead and buried in landfill somewhere. PARISC systems that did not provide any sort of reasonable atomic instructions are now being replaced by HP IPF systems. We can now generally assume that some basic atomic building blocks are available and no longer have to tiptoe around avoiding breaking the old guys.

Another reason that makes atomic usage easier these days is the wide availability of compiler builtins and library infrastructure, rendering this previous area of horrendous unportability accessible. Use of the GCC builtins was illustrated above, showing some incorrect and correct lock implementations. On the same page can be found instructions for compare and swap, atomic read and increment, and others. Similar builtins are available from many other compilers. The boost library implementation of atomic types are a good example. The next revision of the C++ standard will likely include boost-like methods. It appears that C# implements Interchange methods very much like the msdev intrinsics, so this is even becoming a cross language portability area in a limited fashion.

It is fun to illustrate just how murky the portability waters were before the prevalance of compiler intrinsics. Atomic exploitation generally required writing assembly code, and that is never particularly fun. Here is a sample code showing how one would code an "inline assembly" language implementation of fetch and add using the IBM compiler targeting PowerPC

```
1 int PowerPcFetchAndAdd32Internal ( volatile int * pAtomic, int value ) ;
2
3 #pragma mc_func PowerPcFetchAndAdd32Internal { "7c001828" "7c002214" \
4    "7c00192d" "40a2fff4" "7c640050" }
5
6 #pragma reg_killed_by PowerPcFetchAndAdd32Internal cr0, gr0, gr3
```

What we have here is literally a mechanism to instruct the compiler to shove the raw instructions, specified by hexadecimal opcodes, directly into the instruction stream, with the writer of the inline assembly routine providing information about all the registers clobbered and side effects of the code. This inline assembly code is actually generated from the listing in PowerPC assembly fragment above, which was carefully constructed to have the same structure as a real function call on AIX (input params in r3,r4, and output in r3)

Before the GCC intrinsics were available, one could use the powerful GCC inline assembly syntax to instruct the compiler what to shove into the code. Here's an example for fetch and add

```
1 #define IA32_XADD32(pAtomic, addValue, oldValue)                    \
2 __asm__ __volatile__( "lock; xaddl %0,%1\n\t"                       \
3  /* outputs */     : "=r"(oldValue), /* %0 : register */           \
4                      "+m" (*pAtomic) /* %1 : memory location */     \
5  /* inputs */      : "0" (addValue)  /* %2 == %0 : register */      \
6  /* clobbers */    : "cc"            /* condition registers (ZF) */ \
7                       )
```

Looks kind of like ASCII barf, but does the job nicely enough if you can get it right.

On platforms that provided no good inline assembly methods was not available (which includes those platforms and compilers for which inline assembly makes the code worse than using out of line assembly) one had the option of coding standalone assembler files to create the function call methods desired. That can be just as difficult since one is then forced to learn the assembler syntax in more detail and understand the idiosyncrasies of all the platform calling conventions.

4. Types of barriers.

There are in general a few different types of barriers (or fences). Not all architectures that allow for unordered instructions necessarily implement all of these. Some of these may also only be available as variations of specific instructions, so when standalone use is required a more expensive instruction may be required.

To describe the barrier types I will borrow ia64 nomenclature and assembly fragments, which is nicely descriptive despite death of this platform. The ia64 architecture provides acquire, release and full barriers. Some of these are built into the load and store operations on these platforms.

4.1. Acquire barrier

If one sees assembly code containing

```
1 ld4       r3    = [r30]    ; "BEFORE"
2 st1       [r4] = r40       ; "BEFORE"
3
4 ld4.acq   r5    = [r42]    ; LOAD with ACQUIRE barrier
5
6 ld4       r6    = [r43]    ; AFTER
7 st1       [r7] = r44       ; AFTER
```

One has no guarantee about the relative ordering of the BEFORE-labeled memory accesses. Similarly one has no guarantee about the ordering of the pair of AFTER-labeled memory accesses, but because of the acquire barrier the load and acquire barrier must be complete before the AFTER accesses.

Note that the acquire barrier is one directional. It does not prevent the pair of BEFORE memory accesses from occurring after the barrier. Illustrating by example, the instructions could be executed as if the compiler had generated the following sequence of instructions

```
1 ld4.acq   r5    = [r42]
2
3 st1       [r7] = r44
4 ld4       r3    = [r30]
5 ld4       r6    = [r43]
6 st1       [r4] = r40
```

There may be additional architecture rules about dependent loads and stores effecting the types of code motion that is allowed by the compiler, or the way that the CPU implements the architecture, but I will ignore those here.

This sort of effective instruction reordering is often described saying that one instruction has passed another.

This acquire barrier, perhaps unsurprising, is exactly the type of barrier required when implementing code for mutex acquisition. In addition to ia64, the PowerPC platform requires such barriers, and provides the isync instruction to the programmer. An example in a higher level language that may not have the desired effect without an such an acquire barrier is

```
1 if ( pSharedMem->atomic.bitTest() ) // use a bit to flag that somethingElse is "ready"
2 {
3     foo( pSharedMem->somethingElse ) ;
4 }
```

If correct function of your program depends on somethingElse only being read after the atomic update flagging a condition is complete, then this code is not correct without an isync or other similar platform specific acquire barrier.

Your code should really look something like this

```
1 if ( pSharedMem->atomic.bitTestAcquire() )
2 {
3     foo( pSharedMem->somethingElse ) ;
4 }
```

Here it is assumed the atomic library provides acquire and release variations of all atomic operations. For PowerPC this could be as simple as adding an isync to the unordered atomic operation, but on a platform like ia64 one would probably use a different instruction completely (ie. cmpxchg.acq for example).

With such an acquire barrier the somethingElse access or another other memory operation that follows the load associated with the bitTest necessarily follows the load itself. For example, this can mean that a non-dependent load that has been executed ahead of the acquire barrier will have to be thrown out and reexecuted.

4.2. ***Release barrier***

A release barrier is perhaps a bit easier to understand. Illustrating again by example, suppose that we have the following set of load and store instructions, separated by a release barrier.

```
1 ld4       r3     = [r30]    ; BEFORE
2 st1       [r4]   = r40      ; BEFORE
3
4 st4.rel   [r42] = r5        ; STORE with release barrier.
5
6 ld4       r6     = [r43]    ; "AFTER"
7 st1       [r7]   = r44      ; "AFTER"
```

The release barrier ensures that the pair of BEFORE-labeled reads and write memory accesses are complete before the barrier instruction. There is nothing that prevents the BEFORE instructions from executing in an alternate order, but all of the effects of these instructions visible by any other CPU must be complete before the effects of the store barrier instruction is observable by that other CPU. Like the acquire barrier, the release barrier is a one way fence. Neither of the instructions labeled "AFTER" necessarily have to be executed after the release barrier.

On PowerPC this release barrier is implemented with a light weight sync (different than the isync that implements the acquire barrier). As a general rule of thumb, code that requires a release barrier will require an acquire barrier and vice-versa. These instructions are almost always required to be used in complementary pairs.

Similar but opposite to the acquire barrier, a release barrier has the semantics required for implementing the release portion of a mutex.

4.3. ***Full barrier.***

We have seen how use of release and acquire barriers can ensure that that memory access before or after the barrier can be forced to maintain that order relative to the barrier. There are however, some types of memory ordering that neither of these can ensure.

Using ia64 code to illustrate, suppose one has memory accesses before a release barrier, and memory accesses after an acquire barrier.

```
1 ld4       r3     = [r30]    ; BEFORE
2 st1       [r4]   = r40      ; BEFORE
3
4 st4.rel   [r42] = r5        ; STORE with release barrier.
5 ld4.acq   r8     = [r45]    ; LOAD with acquire barrier.
6
7 ld4       r6     = [r43]    ; AFTER
8 st1       [r7]   = r44      ; AFTER
```

The barriers here ensure that the BEFORE-labeled instructions all complete before the release barrier effects become visible to other cpus, and also ensure that none of the AFTER-labeled instructions may start before the acquire barrier instruction is complete. This barrier cannot prevent the acquire barrier LOAD instruction from completing before the STORE barrier. Illustrating by example, the CPU could execute things like so

```
1 ld4.acq   r8     = [r45]    ; LOAD with acquire barrier.
2
```

```
3 ld4       r3     = [r30]    ; BEFORE
4 st1       [r4]   = r40      ; BEFORE
5
6 ld4       r6     = [r43]    ; AFTER
7 st1       [r7]   = r44      ; AFTER
8
9 st4.rel   [r42] = r5        ; STORE with release barrier.
```

This alternate ordering also meets the requirements of the original code. All of the BEFORE-labeled instructions are still before the release barrier and all the AFTER-labeled instructions occur after the acquire barrier. In order to enforce this type of ordering we require a special standalone ordering construct. On ia64 this is the mf (memory fence) instruction, and on that platform it acts as both a standalone acquire and release barrier, but also prevents store load reordering. Of the weakly ordered platforms that are widely available (PowerPC, Sparc, ia64), all of these architectures require a special standalone instruction to enforce this special ordering.

The Sparc architecture has a number of possible ordering models, but the one implemented in all the CPUs that you would find in a Solaris Sparc based system is designated total store order (TSO). For all practical purposes this means that all instruction pairs are ordered except a store load sequence, and in order to enforce such an ordering one must interject a 'membar #storeload' instruction.

On PowerPC, in order to guarantee store load instructions maintain their order, one must interject a heavy-weight sync instruction between the two (sync).

One gets the impression that whatever hardware black magic is required to implement store load ordering, it has the appearance of being harder to enforce this particular type of instruction ordering than any other.

One of the interesting things about these store load barrier instructions, is not that they do enforce this ordering, but that the release and acquire barrier instructions do not. In particular a mutex implemented with acquire and release barriers and not with a full barrier instruction can let exterior memory accesses into the critical section.

Suppose for example we have the following code

```
1  pSharedMem->x = 3 ;   // BEFORE
2  v = pSharedMem->y ;   // BEFORE
3
4  mutex.acquire() ; // assumed to contain an acquire barrier.
5
6  pSharedMem->z = 4 ;
7  foo( pSharedMem->w ) ;
8
9  mutex.release() ; // assumed to contain a release barrier.
10
11 pSharedMem->a = 2 ;   // AFTER
12 q = pSharedMem->b ;   // AFTER
```

Unless the mutex uses very heavy handed memory barrier instructions one has the possibility that either the AFTER or BEFORE labeled memory accesses may sneak into the critical section that the mutex provides.

This subtlety may not be documented by the mutex implementation, but one should not assume that a mutex does any more than keep stuff within the critical section from getting out.

5. **The flag pattern illustrated.**

The mutex is by far the most important example of barrier use that is required for correctness in a weakly ordered system. It is however, perhaps not the easiest to understand. A slightly simpler pattern is the use of an atomic to flag that another memory update is complete.

As mentioned it is common for correct memory barrier usage to require paired conugate acquire and release barriers. This is very much like what is in the implementation of the critical section that you are probably trying to avoid by using an atomic and the associated barriers. It will however, be split, with the release and acquire barrier usage divided into portions for the producer and consumer. As an example your critical section implementation would typically be of the form:

```
1  while (!pShared->lock.testAndSet_Acquire()) ;
2  // (this loop should include all the normal critical section stuff like
3  // spin, waste,
4  // pause() instructions, and last-resort-give-up-and-blocking on a resource
5  // until the lock is made available.)
6
7  // Access to shared memory.
8
9  pShared->foo = 1
10 v = pShared-> goo
11
12 pShared->lock.clear_Release()
```

Acquire memory barrier above makes sure that any loads (pShared-¿goo) that may have been started before the successful lock modification are tossed, to be restarted if necessary.

The release memory barrier ensures that the load from goo into the (local say) variable v is complete before the lock word protecting the shared memory is cleared.

You have a similar pattern in the typical producer and consumer atomic flag scenario (it is difficult to tell by your sample if that is what you are doing but should illustrate the idea).

Suppose your producer used an atomic variable to indicate that some other state is ready to use. You'll want something like this:

```
1  pShared->goo = 14
2
3  pShared->atomic.setBit_Release()
```

Without a release ("write") barrier here in the producer you have no guarantee that the hardware isn't going to get to the atomic store before the goo store has made it through the CPU store queues, and up through the memory hierarchy where it is visible (even if you have a mechanism that ensures the compiler orders things the way you want).

In the consumer

```
1  if ( pShared->atomic.compareAndSwap_Acquire(1,1) )
2  {
3      v = pShared->goo
4  }
```

Without a "read" barrier here you won't know that the hardware hasn't gone and fetched goo for you before the atomic access is complete. The atomic (ie: memory manipulated with the Interlocked functions doing stuff like lock cmpxchg), is only "atomic" with respect to itself, not other memory.

That's really all there is to this example, so it is perhaps anticlimatic, as well as including a redundant discussion of a mutex implementation. However, it is a good first step and these notes are already too long. A more and better example will perhaps follow on a different day.

6. **Conclusion.**

Some of the issues associated with using atomics and barriers to replace the simpler lock mechanism have been discussed. It is unfortunate to have to provide examples that utilize assembly code to illustrate these ideas, but it is harder to discuss these topics without doing so. Hopefully the inclusion of some of the low level details and assembly listings has not rendered this on paper discussion too incomprehensible.

The fact that some low level systems details must be understood for correct use of atomics to avoid locking should perhaps be a warning to the programmer. It can be very hard to implement lock avoidance algorithms correctly, harder still to test them. There is also a significant maintenance impact to going down this road, so be good and sure that you really need the performance provided before taking this path. Premature optimization in this area can be particularly dangerous and costly.

Do you really have a good reason for going down the atomic path instead of just using a lock? Doing all this correctly can be very difficult. Be prepared for a lot of self doubt and insecurity in code reviews and make sure you have a lot of high concurrency testing with all sorts of random timing scenarios. Use a critical section unless you have a very very good reason to avoid it, and don't write that critical section yourself.

The correctness aspects of memory barrier utilization makes your code intrinsically unportable. Because many correct uses of atomics also require barrier instructions, this can in turn imply that the "innocent and simple" atomic class library implementation you have access to can be correspondingly unportable. This may be surprising to the developer, but then again, many aspects of weakly ordered system behavior can be surprising.

On a weakly ordered system, your compiler probably provides _acquire and _release variations for most of the atomic manipulation methods. On a strongly ordered platform like ia32 or amd64, _acquire() or _release() suffixed atomic intrinsics will not be any different than the vanilla operations. With ia64 effectively dead (except on HP where its still twitching slightly) PowerPC is probably the most pervasive surviving weakly ordered architecture around, and even it has toyed with non-weak ordering (power5 was weakly ordered, power6 was not, and power7 is again weakly ordered). It will be interesting to what path the hardware folks take, and whether these details are of any importance in 10 or 15 years.

References

[1] M. Lyons, E. Silha, and B. Hay. PowerPC storage model and AIX programming, 2002. Available from: `http://www.ibm.com/developerworks/eserver/articles/powerpc.html`.

[2] D. Lea. The JSR-133 cookbook for compiler writers, 2005. Available from: `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[3] L.C.H.J. Boehm. Threads and memory model for C++ [online]. Available from: `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/`.

[4] J. Manson and B. Goetz. Jsr 133 (java memory model) faq [online]. 2004. Available from: `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`.

[5] L.C.H.J. Boehm. C++ atomic types and operations. mailings N2427. *C++ standards committee*, 2007. Available from: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html`.

[6] H. Sutter. Prism-A Principle-Based Sequential Memory Model for Microsoft Native Code Platforms Draft Version 0.9. 1, 2006. Available from: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2197.pdf`.

[7] R. Silvera, M. Wong, P. McKenney, and B. Blainey. A simple and efficient memory model for weakly-ordered architectures, 2007. Available from: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2153.pdf`.

[8] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996. Available from: `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf`.

[9] J.D. Valois. Lock-free linked lists using compare-and-swap. Available from: `ftp://ftp.cs.rpi.edu/pub/valoisj/podc95.ps.gz`.

[10] T.L. Harris, K. Fraser, and I.A. Pratt. A practical multi-word compare-and-swap operation. *Lecture Notes in Computer Science*, 2508:265–279, 2002. Available from: `http://www.cl.cam.ac.uk/research/srg/netos/papers/2002-casn.pdf`.

[11] T.L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001. Available from: `http://www.cl.cam.ac.uk/research/srg/netos/papers/2001-caslists.pdf`.

[12] T.L. Harris. Harris Papers [online]. Available from: `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`.

[13] R. Bencina. lock free papers and notes. [online]. 2006. Available from: `http://www.audiomulch.com/~rossb/code/lockfree/`.

[14] M.M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM New York, NY, USA, 2002. Available from: `http://www.research.ibm.com/people/m/michael/podc-2002.pdf`.