Everything you know is wrong: The amazing time traveling CPU, and other horrors of concurrency*

Ethel Bardsley

Imperial College London, UK
emb2009@ic.ac.uk

Abstract

In this paper, we shall explore *weak memory models*, their insidious effects, and how it could happen to you! It shall explained how and why both compilers and CPUs rewrite your program to make it faster, the inevitable fallout of this, and what you can do to protect your code. We shall craft a lock, building from a naïve and broken implementation up to a safe and correct form, and study the underlying model that requires these modifications as we go.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Concurrency, weak memory, compilers

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.11

1 A tale of things to come

Once upon a time, life was simple. Compilers were straightforward, CPUs comprehensible. And things were good, for a time. But then the users' desire for more speed overwhelmed them. Deals were made, souls were sold, and so came about the pipeline, and with it speculation, and instruction level parallelism, and out of order execution. But while the processor's hunger grew, memories and buses could not keep up enough to sate the beast. And so came caching and buffering, more and more levels, towering above the masses. At this time, compilers also resorted to trickery, performing fiendish transformations, twisting a program's natural form to make it a more palatable meal. However, these terrible secrets below the surface were hidden from the software above, which could go about its day, blissfully unaware of the madness below the streets.

But then, in our hubris, we wanted multiple threads. And we wanted multiple CPUs to run our new threads. We even started to shed the locks placed to keep us safe, proudly declaring ourselves "lock free and scalable". And so the insanity started to leak.

While many, even most, of the plains and roads above were the same as ever, sometimes a stray program might stumble into things it should not see. A crunch, a scream. Some are killed instantly, others stumble on before falling some time later. Yet others, worse, survive but are "changed", returning to their daily lives, occasionally finding themselves somewhere unexpected with no knowledge of how they got there, or why they're standing, dazed, over the corpse of a now hideously corrupted file system.

But there are ways to fight back! Barriers, used to help forge the same locks we abandoned, could save us. With careful understanding and placement, we could keep the unspeakable out, while still reaping many of the benefits they provided us. And so it comes to me to warn of the danger, and pass the rites of protection on to you!

* This paper was written in the course of work funded by Intel Corporation.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 11–18

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 In the beginning

Below is a C implementation of Peterson's mutual exclusion algorithm for two threads, 0 and 1 [10]:

```
1  unsigned flag [2] = {0,0};
2  unsigned turn = 0;
3
4  void lock (unsigned thread) {
5          unsigned other_thread = (thread+1)%2;
6
7          flag[thread] = 1;
8          turn = other_thread;
9          do {
10                 //spin
11         } while (flag[other_thread] && turn == other_thread);
12 }
13
14 void unlock (unsigned thread) {
15         flag[thread] = 0;
16 }
```

Each thread has a flag with which to state its intention to lock (line 1), and there is a variable to indicate whose turn it is in the event they both want the lock at once (line 2). Upon entering `lock()`, the thread first sets its associated flag and lets it be the other thread's turn (lines 7–8). Then they wait until either the other thread's flag is unset, or it becomes their turn (lines 9–11). To unlock, the calling thread unsets its flag (line 15).

This locking mechanism has been proven correct [3], and indeed it is under the assumption that the code as written is what will be executed. However, as Knuth would remind us, simply proving code correct can be insufficient, and we shall see what is necessary to maintain correctness in the face of an actual implementation.

3 The very hungry compiler

Optimizing compilers are very clever machines. They take often-inefficient human-readable code and transform it into an equivalent program that is much faster. Alas, much of this is performed under the assumption that the code is either single-threaded, or that nothing will be shared between threads. Consider the core of the above lock implementation:

```
1          flag[thread] = 1;
2          turn = other_thread;
3          do {
4                  //spin
5          } while (flag[other_thread] && turn == other_thread);
```

The compiler "knows" that `turn` is equal to `other_thread` because we set it that way before the loop, and so can remove that "redundant" part of the while condition:

```
1          flag[thread] = 1;
2          turn = other_thread;
3          do {
4                  //spin
5          } while (flag[other_thread]);
```

Similarly, `flag[other_thread]` is never modified by the loop, so is only checked once, before the loop:

```
1         flag[thread] = 1;
2         turn = other_thread;
3         if (flag[other_thread]) {
4           while (1) {
5                 //spin forever
6           }
7         }
```

And the compiler is totally "safe" to make these transformations — if this code were single threaded, these are optimizations you would even want your compiler to make. In fact, here is the control flow graph of the code generated by the Clang C compiler for the naïve implementation from Section 2 at optimization level O3:

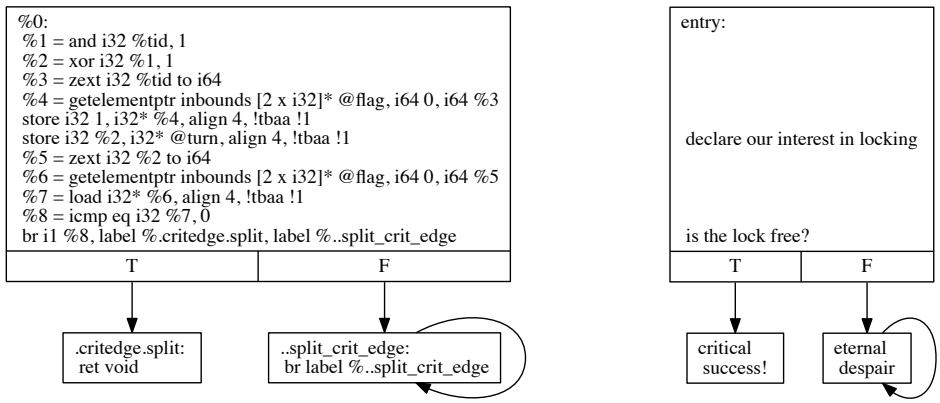


Figure 1 Control flow graph of LLVM bitcode for `lock()`, with English translation on the right

This obviously does not work. So what can we do to fix our lock? Some would suggest declaring the lock variables as `volatile`, as this provides two guarantees in C:

- The compiler may not omit a `volatile` access
- The compiler may not reorder `volatile` stores

However, we only require the first guarantee; reading `flag` or `turn` first is unimportant, and enforcing the second restricts acceptable optimization. Furthermore, `volatile` *does not* prevent the compiler reordering other accesses [2, 6], including whatever the lock may have been guarding, rendering the lock useless. Instead of `volatile`, we should insert a *compiler barrier*, which forces the compiler to not omit any reads or re-order accesses around it:

```
1         flag[thread] = 1;
2         turn = other_thread;
3         do {
4                 asm volatile ("":::"memory");
5         } while (flag[other_thread] && turn == other_thread);
```

The inserted line 4 adds a blank assembly instruction, tells the compiler not to remove it (`volatile`), with the `"memory"` parameter declaring that it can arbitrarily change, or "clobber" all of memory. This is sufficient for compiler to load `flag` and `turn` every iteration,

while allowing it to optimize the condition itself. It also provides a stronger ordering guarantee than `volatile` variables, ensuring that any other code after the barrier stays there.

And to check the control flow graph:

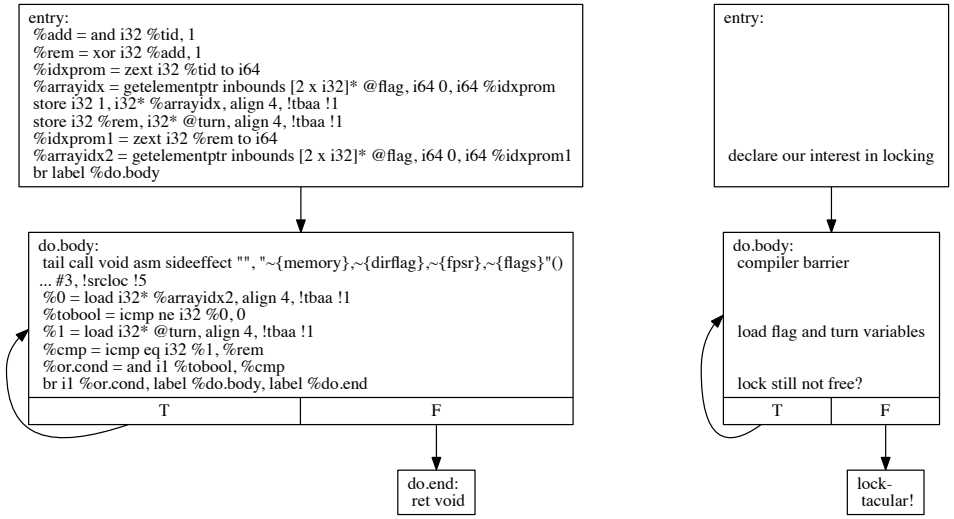


Figure 2 Control flow graph for `lock()` with compiler barrier inserted, with translation on right

Sweet!

4 The Call of Cthoncurrency

Alas, despite bending the compiler to our will, the hardware itself, the very fabric of execution, will conspire to undo us. This is where the true madness lies.

The increasing gap between CPU and RAM performance has required a number of solutions to keep the processor fed. As multiprocessor systems become the norm, the overhead required to maintain coherency between the many component parts spirals. And so rules were relaxed. Guarantees were loosened. Causality was called into question. And this is usually OK — like the very hungry compiler, this only matters when threads are sharing data, and only for those shared locations. For everything else, this speeds execution, simplifies CPU design, lowers power consumption, and generally makes everyone happy.

However, for those cases of shared data between threads, confusing and seemingly "impossible" bugs can arise. We shall describe the culprits, how they lead us here, and what we can do about it.

Sequential consistency The intuitive notion of multi-threaded programming [8], where instructions execute in the order you wrote them and all processors see memory the same way at the same time. Under this notion, executing a multi-threaded program on multiple processors is the same as doing so on a single one, and concurrency is easy to reason about. A simpler time, when the world made sense.

Instruction reordering Many modern CPUs implement *out-of-order execution*, meaning that instructions may not actually happen in program order. This allows many important performance optimizations, e.g. if a processor encounters a sequence `X;Y`, has the data for `Y` but not `X`, and `Y` does not depend upon `X`, then it can perform `Y` while waiting. The CPU

will have a *reorder window*, within which it can perform dependence analysis and rearrange instructions for optimal throughput. While some designs (e.g. x86) will ensure external effects such as writes, are put back in the original order, many do not (e.g. ARM).

Memory buffering In modern computers, memory is very far away and very slow; reading from cache can be an order of magnitude slower than executing an instruction, with a full trip to main memory being an order slower again. Rather than stalling waiting for a write to finish, writes go into a per-core buffer to be resolved later while the core continues executing. Similarly, rather than waiting for a read to traverse the cache hierarchy, the reorder window allows the processor to see what locations will be required next and start reading them in ahead of time to be ready for when the instruction is actually executed.

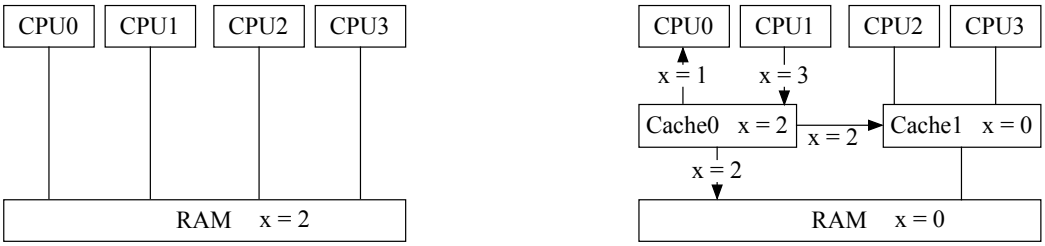


Figure 3 Intuitive view of memory in a multiprocessor system (left) vs harsh reality (right)

On the right,

- CPU_0 read an old value `x = 1`.
- CPU_1 just wrote `x = 3`, and will see that.
- $Cache_0$ contains the `x = 2` that CPU_1 wrote earlier in the program, and is propagating that to RAM and $Cache_1$.
- RAM and $Cache_1$ contain the original x = 0.

As a result of reordering and buffering, each CPU conceptually works in its own disconnected time bubble, reading from the distant past, writing to the distant future. Sometimes their timelines will cross, but, like the time traveller's wife, they will never know when, or how far the other has progressed.

Example: x86-TSO A (comparatively) simple example of weak memory is the model describing Intel and AMD's x86 and x86-64 processors, known as *x86-TSO* (Total Store Order) [9]. It differs from sequential consistency in only two ways: a processor might read old data, and writes take an unspecified amount of time to be globally visible. In all other ways, it is sequentially consistent:

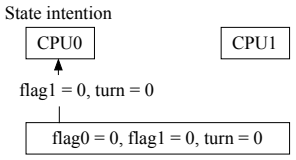
- Writes from a given processor, while arbitrarily delayed, are always in program order (e.g. Sparc's PSO (Partial Store Order) will not ensure this)
- Once a value is globally visible, it is immediately globally visible to all others (e.g. ARM does not provide this, allowing 'write atomicity relaxation')
- A read cannot have an older value than a prior read to the same location (e.g. DEC Alpha can do this, known as 'read-after-read reordering')
- A read cannot violate causality, reading data that no thread wrote (e.g. C++11 and Java allow 'out-of-thin-air reads')

But it still sufficiently deviates from sequential consistency to be problematic.

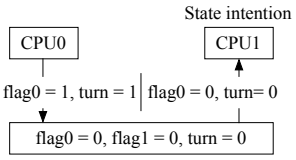
For example, while we prevented the compiler from discarding memory accesses, the hardware may yet reorder them. Due to the effects of read and write buffering, it may appear as though our lock was actually:

```
1       do {
2               asm volatile ("":::"memory");
3       } while (flag[other_thread] && turn == other_thread);
4       flag[thread] = 1;
5       turn = other_thread;
```

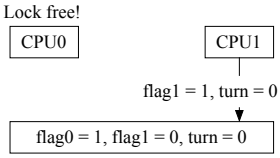
The reads will have started ahead of time, and the writes will take a while to propagate down (see Figure 4 for a step-by-step of how this may happen). As a result, both threads can read stale data, incorrectly see the lock as free, and both proceed into the critical section, wreaking havoc as they do so.



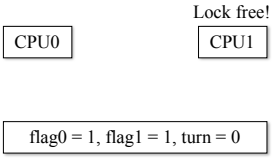
(a) CPU_0 states its intention to lock, and starts loading $flag_1$.



(b) CPU_0's intention write is in transit. CPU_1 also states intent, and starts loading $flag_0$.



(c) CPU_0 checks the copy of $flag_1$ it read in (a), sees it clear, progressing into the critical section. CPU_1's write is in transit.



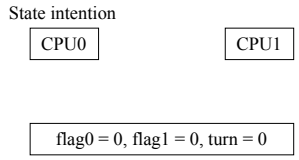
(d) CPU_1 checks the copy of $flag_0$ it read in (b), sees it clear. Both CPUs are now in the critical section.

Figure 4 Two CPUs entering the lock simultaneously under x86-TSO

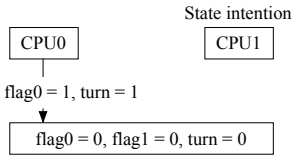
So what can we do? Much like the compiler barrier, we can insert a *barrier instruction* to tell the processor to leave our program alone. An `MFENCE` instruction on x86 CPUs will stall until its write buffer has been fully flushed to memory, and forces it to perform reads after, rather than loading them up ahead of time. This allows us to regain just enough sequential consistency to ensure correctness. We can insert one between declaring intent to lock and checking for lock freedom (see Figure 5 for the step-by-step):

```
1       flag[thread] = 1;
2       turn = other_thread;
3       asm volatile ("mfence")
4       do {
5               asm volatile ("":::"memory");
6       } while (flag[other_thread] && turn == other_thread);
```

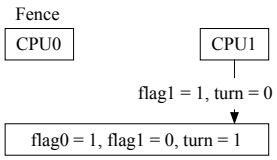
x86 also offers `SFENCE` and `LFENCE` to only enforce write or read ordering, respectively, and other architectures with weaker models often provide many more for fine-grained control. In "hot" or high performance code, it may be preferable to use the weakest possible (barriers force the CPU to slow down, and stronger means slower), but a full barrier like `MFENCE` will always be safe.



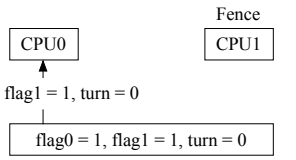
(a) CPU_0 states its intention to lock.



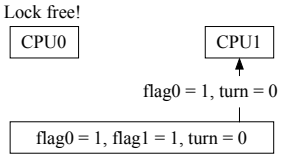
(b) CPU_0's write is in transit. CPU_1 also states intent.



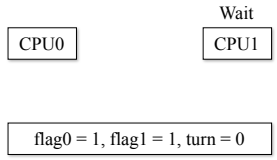
(c) CPU_0 fences, waiting for its write to finish. CPU_1's write is in transit.



(d) CPU_0 is forced to read fresh copies of the flags by the prior fence. CPU_1 also fences.



(e) CPU_0 sees that it is its turn, proceeds into the critical section. CPU_1 reads fresh copies of the flags.



(f) CPU_1 sees it is not its turn, and waits.

Figure 5 Successful mutual exclusion!

5 The moral of the story

In this paper we have seen the effects of some compiler and hardware optimizations, and how these can cause unexpected behaviors in concurrent code. The simple take-home message is to avoid writing your own low-level concurrency primitives or "lock-free" concurrent data structures where possible. If you must, let paranoia be your guide: check compiler output for sensitive regions, and read in-depth descriptions of what your compilers and architectures will do [4]. Tools that assist in detecting where fences are required may also be of help [1, 5].

For some managed platforms, such as Java, the model is consistent between environments regardless of host architecture, simplifying the conceptual overheadfor managing concurrency without locks. Additionally, in the case of Java and C#, `volatile` does behave more like a barrier without the unintuitive potential for reordering of other accesses [7].

Alas, for C and C++ this is unfortunately both compiler and architecture specific. While the `asm` lines used in this paper are specific to GCC and Clang, equivalents exist for other compilers, such as `_ReadWriteBarrier()` compiler barrier and `MemoryBarrier()` hardware barrier in MSVC. To achieve more portable code, you should wrap these in compile-time macros, e.g. `compiler_barrier` and `full_barrier` [4]. There is also some support for atomic accesses and barriers introduced in the C11 and C++11 standards, however not all compilers support these yet, and some never will, and so they are also effectively non-portable.

On the other hand, if you write single-threaded code, or multi-threaded code without shared data structures, or use the locking mechanisms provided by your environment (e.g. pthreads), none of the above matters. You can go about your ways, unconcerned and carefree.

References

1 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *CoRR*, abs/1312.1411, 2013.

2 Jonathan Corbet. Volatile considered harmful. `https://www.kernel.org/doc/Documentation/volatile-considered-harmful.txt`. Retreived from Linux 3.15-rc7.

3 Micha Hofri. Proof of a mutual exclusion algorithm—a classic example. *ACM SIGOPS Operating Systems Review*, 24(1):18–22, 1990.

4 David Howells and Paul E. McKenney. Linux kernel memory barriers. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`. Retreived from Linux 3.15-rc7.

5 Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 141–152, New York, NY, USA, 2013. ACM.

6 ISO. Programming Language C (C11). Standard ISO/IEC 9899:2011, International Organization for Standardization, 2011.

7 JCP. JSR 133: JavaTM Memory Model and Thread Specification Revision. Standard, Java Community Process, 2004.

8 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

9 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009.

10 Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.