Why is volatile not considered useful in multithreaded C or C++ programming?

As demonstrated in this answer I recently posted, I seem to be confused about the utility (or lack thereof) of `volatile` in multi-threaded programming contexts.

My understanding is this: any time a variable may be changed outside the flow of control of a piece of code accessing it, that variable should be declared to be `volatile` . Signal handlers, I/O registers, and variables modified by another thread all constitute such situations.

So, if you have a global int `foo` , and `foo` is read by one thread and set atomically by another thread (probably using an appropriate machine instruction), the reading thread sees this situation in the same way it sees a variable tweaked by a signal handler or modified by an external hardware condition and thus `foo` should be declared `volatile` (or, for multithreaded situations, accessed with memory-fenced load, which is probably a better a solution).

How and where am I wrong?

c++ c multithreading volatile c++-faq

edited Oct 21 '11 at 9:35

R. Martinho Fernandes
106k 32 266 373

asked Mar 20 '10 at 22:10

Michael Ekstrand
13.7k 2 36 62

5 All volatile does is say that the compiler should not cache the access to a volatile variable. It says nothing about serialising such access. This has been discussed here I don't know how many times, and I don't think this question is going to add anything to those discussions. – anon Mar 20 '10 at 22:22

4 And yet again, a question which doesn't deserve it, and has been asked here many times before gets upvoted. Will you please stop doing this. – anon Mar 20 '10 at 22:26

11 @neil grumpy today? ;) – jalf Mar 20 '10 at 23:07

11 @neil I searched for other questions, and found one, but any existing explanation I saw somehow didn't trigger what I needed to really understand why I was wrong. This question has elicited such an answer. – Michael Ekstrand Mar 21 '10 at 18:17

1 For a great in-depth study on what CPUs do with data (via their caches) check out: rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf – KenK May 4 '12 at 17:24

9 Answers

The problem with `volatile` in a multithreaded context is that it doesn't provide *all* the guarantees we need. It does have a few properties we need, but not all of them, so we can't rely on `volatile` *alone*.

However, the primitives we'd have to use for the *remaining* properties also provide the ones that `volatile` does, so it is effectively unnecessary.

For thread-safe accesses to shared data, we need a guarantee that:

- the read/write actually happens (that the compiler won't just store the value in a register instead and defer updating main memory until much later)
- that no reordering takes place. Assume that we use a `volatile` variable as a flag to indicate whether or not some data is ready to be read. In our code, we simply set the flag after preparing the data, so all *looks* fine. But what if the instructions are reordered so the flag is set *first*?

`volatile` does guarantee the first point. It also guarantees that no reordering occurs *between different volatile reads/writes*. All `volatile` memory accesses will occur in the order in which

they're specified. That is all we need for what `volatile` is intended for: manipulating I/O registers or memory-mapped hardware, but it doesn't help us in multithreaded code where the `volatile` object is often only used to synchronize access to non-volatile data. Those accesses can still be reordered relative to the `volatile` ones.

The solution to preventing reordering is to use a *memory barrier*, which indicates both to the compiler and the CPU that *no memory access may be reordered across this point*. Placing such barriers around our volatile variable access ensures that even non-volatile accesses won't be reordered across the volatile one, allowing us to write thread-safe code.

However, memory barriers *also* ensure that all pending reads/writes are executed when the barrier is reached, so it effectively gives us everything we need by itself, making `volatile` unnecessary. We can just remove the `volatile` qualifier entirely.

edited Aug 2 '14 at 2:54
Andrew Medico
13.6k 8 27 51

answered Mar 20 '10 at 23:17
jalf
152k 29 218 452

3 Does it really guarantee order and forbid the use of cached values or only does so at the compiler level? If the former was true, you could possibly write portable thread synchronization, but all the code I have seen uses CPU-specific instructions and assumes the CPU will reorder everything. – jbcreix Mar 21 '10 at 8:37

2 @jbcreix: Which "it" are you asking about? Volatile or memory barriers? In any case, the answer is pretty much the same. They both have to work both at compiler and CPU level, since they describe the observable behavior of the program --- so they have to ensure that the CPU doesn't reorder everything, changing the behavior they guarantee. But you currently can't write portable thread synchronization, because memory barriers are not part of standard C++ (so they're not portable), and `volatile` isn't strong enough to be useful. – jalf Mar 21 '10 at 14:07

3 An MSDN example does this, and claims that instructions can't be reordered past a volatile access: msdn.microsoft.com/en-us/library/12a04hfd(v=vs.80).aspx – OJW Sep 16 '11 at 15:06

16 @OJW: But Microsoft's compiler redefines `volatile` to be a full memory barrier (preventing reordering). That's not part of the standard, so you can't rely on this behavior in portable code. – jalf Oct 4 '11 at 7:00

2 @Skizz: no, that's where the "compiler magic" part of the equation comes in. A memory barrier has to be understood by *both* the CPU and the compiler. If the compiler understands the semantics of a memory barrier, it knows to avoid tricks like that (as well as reordering reads/writes across the barrier). And luckily, the compiler *does* understand the semantics of a memory barrier, so in the end, it all works out. :) – jalf Jul 10 '12 at 8:16

Understand that C programmers have often taken volatile to mean that the variable could be changed outside of the current thread of execution; as a result, they are sometimes tempted to use it in kernel code when shared data structures are being used. In other words, they have been known to treat volatile types as a sort of easy atomic variable, which they are not. The use of volatile in kernel code is almost never correct; this document describes why.

The key point to understand with regard to volatile is that its purpose is to suppress optimization, which is almost never what one really wants to do. In the kernel, one must protect shared data structures against unwanted concurrent access, which is very much a different task. The process of protecting against unwanted concurrency will also avoid almost all optimization-related problems in a more efficient way.

Like volatile, the kernel primitives which make concurrent access to data safe (spinlocks, mutexes, memory barriers, etc.) are designed to prevent unwanted optimization. If they are being used properly, there will be no need to use volatile as well. If volatile is still necessary, there is almost certainly a bug in the code somewhere. In properly-written kernel code, volatile can only serve to slow things down.

Consider a typical block of kernel code:

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

If all the code follows the locking rules, the value of shared_data cannot change unexpectedly while the_lock is held. Any other code which might want to play with that data will be waiting

on the lock. The spinlock primitives act as memory barriers - they are explicitly written to do so - meaning that data accesses will not be optimized across them. So the compiler might think it knows what will be in shared_data, but the spin_lock() call, since it acts as a memory barrier, will force it to forget anything it knows. There will be no optimization problems with accesses to that data.

If shared_data were declared volatile, the locking would still be necessary. But the compiler would also be prevented from optimizing access to shared_data *within* the critical section, when we know that nobody else can be working with it. While the lock is held, shared_data is not volatile. When dealing with shared data, proper locking makes volatile unnecessary - and potentially harmful.

The volatile storage class was originally meant for memory-mapped I/O registers. Within the kernel, register accesses, too, should be protected by locks, but one also does not want the compiler "optimizing" register accesses within a critical section. But, within the kernel, I/O memory accesses are always done through accessor functions; accessing I/O memory directly through pointers is frowned upon and does not work on all architectures. Those accessors are written to prevent unwanted optimization, so, once again, volatile is unnecessary.

Another situation where one might be tempted to use volatile is when the processor is busy-waiting on the value of a variable. The right way to perform a busy wait is:

```
while (my_variable != what_i_want)
    cpu_relax();
```

The cpu_relax() call can lower CPU power consumption or yield to a hyperthreaded twin processor; it also happens to serve as a memory barrier, so, once again, volatile is unnecessary. Of course, busy-waiting is generally an anti-social act to begin with.

There are still a few rare situations where volatile makes sense in the kernel:

- The above-mentioned accessor functions might use volatile on architectures where direct I/O memory access does work. Essentially, each accessor call becomes a little critical section on its own and ensures that the access happens as expected by the programmer.
- Inline assembly code which changes memory, but which has no other visible side effects, risks being deleted by GCC. Adding the volatile keyword to asm statements will prevent this removal.
- The jiffies variable is special in that it can have a different value every time it is referenced, but it can be read without any special locking. So jiffies can be volatile, but the addition of other variables of this type is strongly frowned upon. Jiffies is considered to be a "stupid legacy" issue (Linus's words) in this regard; fixing it would be more trouble than it is worth.
- Pointers to data structures in coherent memory which might be modified by I/O devices can, sometimes, legitimately be volatile. A ring buffer used by a network adapter, where that adapter changes pointers to indicate which descriptors have been processed, is an example of this type of situation.

For most code, none of the above justifications for volatile apply. As a result, the use of volatile is likely to be seen as a bug and will bring additional scrutiny to the code. Developers who are tempted to use volatile should take a step back and think about what they are truly trying to accomplish.

edited Oct 14 '14 at 5:34

answered Mar 21 '10 at 2:59

Mustapha Abiola
787 4 11

"Adding the volatile keyword to asm statements will prevent this removal." Really? – curiousguy Oct 4 '11 at 0:24

1 @curiousguy: Yes. See also gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/Extended-Asm.html . – phresnel Dec 22 '11 at 11:56

@phresnel I see. That's kind of surprising. – curiousguy Dec 22 '11 at 19:29

I don't think you're wrong -- volatile is necessary to guarantee that thread A will see the value change, if the value is changed by something other than thread A. As I understand it, volatile is basically a way to tell the compiler "don't cache this variable in a register, instead be sure

to always read/write it from RAM memory on every access".

The confusion is because volatile isn't sufficient for implementing a number of things. In particular, modern systems use multiple levels of caching, modern multi-core CPUs do some fancy optimizations at run-time, and modern compilers do some fancy optimizations at compile time, and these all can result in various side effects showing up in a different order from the order you would expect if you just looked at the source code.

So volatile is fine, as long as you keep in mind that the 'observed' changes in the volatile variable may not occur at the exact time you think they will. Specifically, don't try to use volatile variables as a way to synchronize or order operations across threads, because it won't work reliably.

Personally, my main (only?) use for the volatile flag is as a "pleaseGoAwayNow" boolean. If I have a worker thread that loops continuously, I'll have it check the volatile boolean on each iteration of the loop, and exit if the boolean is ever true. The main thread can then safely clean up the worker thread by setting the boolean to true, and then calling pthread_join() to wait until the worker thread is gone.

answered Mar 20 '10 at 22:19

Jeremy Friesner
18.7k 2 38 80

2 Your Boolean flag is probably unsafe. How do you guarantee that the worker completes its task, and that the flag remains in scope until it is read (if it is read)? That is a job for signals. Volatile is good for implementing simple spinlocks *if* no mutex is involved, since alias safety means the compiler assumes `mutex_lock` (and every other library function) may alter the state of the flag variable. – Potatoswatter Mar 20 '10 at 22:57

6 Obviously it only works if the nature of the worker thread's routine is such that it is guaranteed to check the boolean periodically. The volatile-bool-flag is guaranteed to remain in scope because the thread-shutdown sequence always occurs before the object that holds the volatile-boolean is destroyed, and the thread-shutdown sequence calls pthread_join() after setting the bool. pthread_join() will block until the worker thread has gone away. Signals have their own problems, particularly when used in conjunction with multithreading. – Jeremy Friesner Mar 20 '10 at 23:26

1 The worker thread *isn't* guaranteed to complete its work before the boolean is true -- in fact, it almost certainly will be in the middle of a work unit when the bool is set to true. But it doesn't matter when the worker thread completes its work unit, because the main thread is not going to be doing anything except blocking inside pthread_join() until the worker thread exits, in any case. So the shutdown sequence is well-ordered -- the volatile bool (and any other shared data) won't be freed until after pthread_join() returns, and pthread_join() won't return until the worker thread is gone. – Jeremy Friesner Mar 21 '10 at 3:24

6 @Jeremy, you are correct in practice but theoretically it could still break. On a two core system one core is constantly executing your worker thread. The other core sets the bool to true. However there is not guarantee the the worker thread's core will ever see that change, ie it may never stop even though it repeated checks the bool. This behavior is allowed by the c++0x, java, and c# memory models. In practice this would never occur as the busy thread most likely insert a memory barrier somewhere, after which it will see the change to the bool. – deft_code Mar 22 '10 at 14:54

2 Take a POSIX system, use real time scheduling policy `SCHED_FIFO`, higher static priority than other processes/threads in the system, enough cores, should be perfectly possible. In Linux you can specify that real-time process can use 100% of the CPU time. They will never context switch if there is no higher priority thread/process and never block by I/O. But the point is that C/C++ `volatile` is not meant for enforcing proper data sharing/synchronization semantics. I find searching for special cases to prove that incorrect code maybe sometimes could work is useless exercise. – FooF Aug 1 '13 at 2:44

Your understanding really is wrong.

The property, that the volatile variables have, is "reads from and writes to this variable are part of perceivable behaviour of the program". That means this program works (given appropriate hardware):

```
int volatile* reg=IO_MAPPED_REGISTER_ADDRESS;
*reg=1; // turn the fuel on
*reg=2; // ignition
*reg=3; // release
int x=*reg; // fire missiles
```

The problem is, this is not the property we want from thread-safe anything.

For example, a thread-safe counter would be just (linux-kernel-like code, don't know the c++0x equivalent):

```
atomic_t counter;
```

```
...
atomic_inc(&counter);
```

This is atomic, without a memory barrier. You should add them if necessary. Adding volatile would probably not help, because it wouldn't relate the access to the nearby code (eg. to appending of an element to the list the counter is counting). Certainly, you don't need to see the counter incremented outside your program, and optimisations are still desirable, eg.

```
atomic_inc(&counter);
atomic_inc(&counter);
```

can still be optimised to

```
atomically {
  counter+=2;
}
```

if the optimizer is smart enough (it doesn't change the semantics of the code).

answered Mar 20 '10 at 22:43

jpalecek
32.8k 3 51 94

For your data to be consistent in a concurrent environment you need two conditions to apply:

1) Atomicity i.e if I read or write some data to memory then that data gets read/written in one pass and cannot be interrupted or contended due to e.g a context switch

2) Consistency i.e the order of read/write ops must be *seen* to be the same between multiple concurrent environments - be that threads, machines etc

volatile fits neither of the above - or more particularly, the c or c++ standard as to how volatile should behave includes neither of the above.

It's even worse in practice as some compilers (such as the intel Itanium compiler) do attempt to implement some element of concurrent access safe behaviour (i.e by ensuring memory fences) however there is no consistency across compiler implementations and moreover the standard does not require this of the implementation in the first place.

Marking a variable as volatile will just mean that you are forcing the value to be flushed to and from memory each time which in many cases just slows down your code as you've basically blown your cache performance.

c# and java AFAIK do redress this by making volatile adhere to 1) and 2) however the same cannot be said for c/c++ compilers so basically do with it as you see fit.

For some more in depth (though not unbiased) discussion on the subject read this

answered Mar 21 '10 at 1:28

zebrabox
4,055 13 28

2 +1 - guaranteed atomicity was another piece of what I was missing. I was assuming that loading an int is atomic, so that volatile preventing the re-ordering provided the full solution on the read side. I think it's a decent assumption on most architectures, but it is not a guarantee. – Michael Ekstrand Mar 21 '10 at 18:18

`volatile` is useful (albeit insufficient) for implementing the basic construct of a spinlock mutex, but once you have that (or something superior), you don't need another `volatile`.

The typical way of multithreaded programming is not to protect every shared variable at the machine level, but rather to introduce guard variables which guide program flow. Instead of `volatile bool my_shared_flag;` you should have

```
pthread_mutex_t flag_guard_mutex; // contains something volatile
bool my_shared_flag;
```

Not only does this encapsulate the "hard part," it's fundamentally necessary: C does not include **atomic operations** necessary to implement a mutex; it only has `volatile` to make extra guarantees about *ordinary* operations.

Now you have something like this:

```
pthread_mutex_lock( &flag_guard_mutex );
my_local_state = my_shared_flag; // critical section
pthread_mutex_unlock( &flag_guard_mutex );

pthread_mutex_lock( &flag_guard_mutex ); // may alter my_shared_flag
my_shared_flag = ! my_shared_flag; // critical section
pthread_mutex_unlock( &flag_guard_mutex );
```

`my_shared_flag` does not need to be volatile, despite being uncacheable, because

1. Another thread has access to it.
2. Meaning a reference to it must have been taken sometime (with the `&` operator).
   - (Or a reference was taken to a containing structure)
3. `pthread_mutex_lock` is a library function.
4. Meaning the compiler can't tell if `pthread_mutex_lock` somehow acquires that reference.
5. Meaning the compiler must *assume* that *`pthread_mutex_lock` modifes the shared flag*!
6. So the variable must be reloaded from memory. `volatile`, while meaningful in this context, is extraneous.

edited Mar 20 '10 at 23:23

answered Mar 20 '10 at 23:18

Potatoswatter
75.6k 4 116 235

The comp.programming.threads FAQ quotes Dave Butenhof:
http://www.lambdacs.com/cpt/FAQ.html#Q56

All that's equally applicable to C++.

edited Feb 29 '12 at 10:07

answered Oct 5 '10 at 8:05

Tony D
50.6k 6 54 107

According to my old C standard, *"What constitutes an access to an object that has volatile-qualified type is implementation-defined"*. So C compiler writers *could* have choosen to have "volatile" mean "*thread safe access in a multi-process environment*". But they didn't.

Instead, the operations required to make a critical section thread safe in a multi-core multi-process shared memory environment were added as new implementation-defined features. And, freed from the requirement that "volatile" would provide atomic access and access ordering in a multi-process environment, the compiler writers prioritised code-reduction over historical implemention-dependant "volatile" semantics.

This means that things like "volatile" semaphores around critical code sections, which do not work on new hardware with new compilers, might once have worked with old compilers on old hardware, and old examples are sometimes not wrong, just old.

answered Nov 14 '14 at 11:34

david
444 3 9

This is all that "volatile" is doing: "Hey compiler, this variable could change AT ANY MOMENT (on any clock tick) even if there are NO LOCAL INSTRUCTIONS acting on it. Do NOT cache this value in a register."

That is IT. It tells the compiler that your value is, well, volatile- this value may be altered at any moment by external logic (another thread, another process, the Kernel, etc.). It exists more or less solely to suppress compiler optimizations that will silently cache a value in a register that it is inherently unsafe to EVER cache.

You may encounter articles like "Dr. Dobbs" that pitch volatile as some panacea for multi-threaded programming. His approach isn't totally devoid of merit, but it has the fundamental flaw of making an object's users responsible for its thread-safety, which tends to have the same issues as other violations of encapsulation.

answered Aug 2 '14 at 1:54

Zack Yezek
222 1 3