

Re: Volatile + multithreading

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2005-04/msg01116.html>

- *From:* "Doug Harrison [MVP]" <dsh@xxxxxxxx>
 - *Date:* Thu, 21 Apr 2005 17:09:17 -0500
-

On Thu, 21 Apr 2005 17:30:09 -0400, wxs wrote:

> Is volatile really of any use with the current compilers VS 2003?

The volatile keyword has little use in multithreaded programming. See this message for a list of things volatile is useful for:

<http://groups-beta.google.com/group/microsoft.public.vc.language/msg/8003bec1ea2199d2>

> The only time I think volatile may do something on code is if it is used on
> a parameter value or on a method variable. Using volatile on any class
> field or global seems to have no impact because it seems the compiler will
> never not read or write the variable directly from memory for any variable
> that is not a parameter variable or a method variable.

To the extent that it's semantically correct, and to the extent the compiler optimizes things, the compiler certainly will keep variables in registers as long as it can.

> Thinking more closely on this it kind of makes sense otherwise all
> multithreaded code would potentially be in jeopardy. Think what would
> happen if you really did have to apply volatile to avoid a variable being
> put into a register. In multithreaded code that would many virtually any
> variable you use that you access from another thread would require having
> volatile applied. locks, and locking do not protect you, as the compiler
> doesn't know anything about lock calls (even if it did, it wouldn't matter
> because if you called a locking/synchronization object indirectly through a
> virtual members or just multiple calls there is no way it would know there
> could be an issue until runtime which is too late.)

After you've acquired a mutex, a compiler is free to load a variable into a register and keep it there until the mutex is released. The interesting events are the acquisition and release of the mutex, across which variables cannot be cached. If these are function calls, and the compiler doesn't perform interprocedural analysis, it has to be pessimistic and assume all variables are reachable from these mutex functions, and it will suppress optimizations. If the compiler does perform this analysis and can see into these functions, or they aren't functions at all, it has to recognize these operations as "special" and suppress optimizations around them. Otherwise,

Re: Volatile + multithreading

everything would have to be volatile, and that's impractical for many reasons, not the least of which is that no one writes volatile member functions, so you would have to forget about sharing classes like `std::vector` between threads.

About the interprocedural optimizations, here's an excerpt from a message I wrote a couple of years ago to illustrate this:

Here's a simplistic explanation which is probably not far off the mark for VC. The mutex lock/unlock operations are function calls, and the compiler knows nothing about these functions, so it can't do any interprocedural optimization. Global variables are reachable through functions called by the current function, including lock/unlock. The compiler can't see into the lock/unlock functions to determine that they don't access the globals or call other functions which ultimately do access them. Thus, when you have the sequence below, for non-volatile, global variables `x` and `y`:

```
m.lock();  
y = x;  
x = 2;  
m.unlock();
```

The compiler cannot optimize the assignment to `x` out of existence, because it can't tell that `unlock()` won't refer to `x`. It can't move the `y` and `x` assignments before or after the lock/unlock calls, because that can change the values those functions observe. It can't cache the value of `x`, call `lock()`, and assign the cached value to `y`, because `lock()` may have modified `x`. Before calling `unlock()`, it must flush `x` and `y` out of registers to memory, so that `unlock()` will observe their current values. And so on. The only way I know to screw this up is to write to the variables outside of the critical section, but that's a violation of the locking protocol. So at the compiler level, the variables don't need to be volatile.

In addition to providing mutual exclusion, the mutex lock/unlock operations issue whatever memory barrier instructions are necessary, so that the writes are visible to other threads observing the locking protocol. So at the hardware level, there's no need for the variables to be volatile, assuming volatile implies MB instructions, because they're implicit in the mutex lock/unlock operations.

As already mentioned, all I see is volatile slowing down execution here, while making you cast away volatile to use member functions of classes like `CString`, which now that I'm thinking about it, is undefined per the C++ Standard, 7.1.5.1/7. So unless a class `X` provides volatile member functions, you can't declare a volatile `X` and call member functions on it, because casting away volatile and referring to non-volatile members is undefined. (And I defy you to name a class which defines volatile member functions.)

Re: Volatile + multithreading

(NB: A compiler which can see into the locking operations would have to mark them somehow to suppress optimizations which can violate the expected semantics. There's no other reasonable choice.)

Note that in Windows, `EnterCriticalSection`, `WaitForSingleObject`, etc are opaque functions that live in DLLs, so the correct behavior comes pretty much for free.

> think of a case where I do the following.

> class MyClass

> {

> bool m_bDone;

>

> MyMethod()

> {

> //What if compiler put m_bDone in register AX here

> AcquireCriticalSection()

> while(!m_bDone) //What happens if m_bDone was preloaded in a register

> earlier and now this is while(!AX)

>

> {

> //Do work here

> }

> ReleaseCriticalSection();

> }

>

> //.. Other methods here

>

> }

>

> This means that locks would not protect you, only the keyword volatile

> would. This would require you to put volatile on every variable used in a

> way that mattered.

>

> It seems the compiler writers must have known this and instead imposed the

> rule that only parameter variables and stack variables (method variables)

> can be optimized into registers since no one is likely to pass the address

> to one of those to be used by another thread.

Compiler writers don't implement optimizations on the basis of "likely". So the provenance of the variables doesn't matter. If a variable is potentially reachable by other functions, it has to be pessimistic. For example:

```
void g(int* p);
```

```
void h();
```

```
int f()
```

Re: Volatile + multithreading

```
{  
int x = 0;  
g(&x);  
h();  
return x;  
}
```

After `g` is called, the compiler must assume `h()` can modify `x` through some pointer `g` squirreled away, unless it can prove otherwise. Note this is true even for single-threaded code.

- > This I believe prevents a
- > majority of the multithreading issues that could have resulted from
- > optimization. I thought there might be a compiler option that would
- > optimize this so you would require this. It seems there is a compiler option
- > for aliasing that seems to suggest enabling it will require putting volatile
- > everywhere, but when I tried it, it still did not optimize any of the
- > non-stack method/parameter variables into registers that I could see.

The `/Oa` option isn't safe even for single-threaded code, and it's going away in the next version of the compiler.

- > I'm sure some compiler or platform does support it, and I guess it's
- > possible Microsoft compiler might support it too eventually, but if they
- > enabled it by default it would break a whole heck of a lot of code. So it
- > seems that volatile on Intel platforms with the Microsoft compiler by
- > default really don't require volatile in almost any circumstance.
- >
- > Or am I missing something?

The only formal specification for multithreaded programming I know of is Posix. However, the things I've been talking about represent the only reasonable choice for a compiler intended to be used for multithreaded programming, and they agree with the Posix spec. For such a compiler, synchronization is sufficient, while volatile is neither sufficient nor necessary.

--

Doug Harrison
Microsoft MVP – Visual C++

• **References:**

◆ **[Volatile + multithreading](#)**

◇ From: wxs

- Prev by Date: **[Re: Volatile + multithreading](#)**
- Next by Date: **[Re: CDialog – Cannot set check mark to menu Item](#)**
- Previous by thread: **[Re: Volatile + multithreading](#)**

Re: Volatile + multithreading

- Next by thread: **Re: Volatile + multithreading**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**