ORACLE
SOLARIS

An Oracle White Paper
September 2010

Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2: Part 1, Compiler Barriers

ORACLE

Introduction

Oracle Solaris Studio 12 Update 2 introduces a new header file `<mbarrier.h>`. This header file provides a set of memory ordering intrinsics that are useful for enforcing memory ordering constraints in multithreaded applications.

This article is part 1 of a two-part series. This part discusses how compiler barriers can be used to stop the compiler from generating code that is incorrect due to reordered memory accesses. Part 2 discusses how memory barriers or memory fences can be used to ensure that the processor does not reorder memory operations.

What Is Memory Ordering?

Memory ordering is the order in which memory operations (loads and stores) are performed. There are two ways that memory ordering might be changed:

- The compiler might change memory ordering as a result of some compile-time optimizations.
- The processor might change memory ordering at run time.

Compiler Memory Ordering

For performance reasons, a compiler performs various optimizations. It might hold variable values in registers (rather than in memory), reorder expressions, avoid reloading values, eliminate the use of variables altogether, and so on.

There are two constraints to what the compiler can do. First of all, it must produce code that is functionally equivalent to the code as written. This is the "as-if" rule, which basically says that the output of the optimized program should be as if the program were executed exactly as written.

The second constraint is the user-specified flags passed to the compiler. If these flags say, for example, that "pointers never alias," then the compiler can use that assumption in producing more optimal code.

The code shown in Listing 1 reflects a situation in which a compiler could produce a more-optimal code sequence by reordering the memory operations.

The code starts a team of threads. Each thread waits until it is signaled before starting work and then signals as soon as it completes work. The main thread is responsible for creating the team of threads, starting them, and then waiting until their work is complete.

Listing 1. Multithreaded Code in which the Compiler Might Perform Undesirable Optimizations

```
#include <stdio.h>
#include <pthread.h>

volatile int   start[10];
volatile int   ended[10];
pthread_t    threads[10];

void * work( void * param)
{
   int id = (int) param;
   while( start[ id ] == 0 ){} // Wait until work started
   printf( "Thread %i started\n", id );
   ended[ id ] = 1;            // Indicate that work completed
   printf( "Thread %i finished\n", id );
}
```

```
void dowork()
{
  for (int count = 0; count < 10; count++ )
  {
    start[ count ] = 1;  // Start thread working
  }
  for (int count = 0; count < 10; count++ )
  {
    while( ended[ count ] == 0 ) {} // Wait until thread completes work
  }
}

int main()
{
  // Create team of threads
  for( int count = 0; count < 10; count++ )
  {
    start[ count ] = 0;
    ended[ count ] = 0;
    pthread_create( &threads[count], 0, work, (void*)count );
  }

  dowork();

  // Join team of threads
  for( int count = 0; count < 10; count++ )
  {
    pthread_join( threads[count], 0 );
  }
}
```

Compiling and running the code in Listing 1 without optimization produces the results shown in Listing 2. The threads start and finish in an unspecified order.

Listing 2. Compiling and Running Code Without Optimization

```
% cc -mt listing1.c
% ./a.out
Thread 1 started
Thread 1 finished
Thread 9 started
Thread 9 finished
Thread 5 started
Thread 5 finished
Thread 0 started
Thread 0 finished
Thread 3 started
Thread 3 finished
Thread 4 started
Thread 4 finished
Thread 6 started
Thread 6 finished
Thread 7 started
Thread 7 finished
Thread 2 started
Thread 2 finished
Thread 8 started
Thread 8 finished
```

We can compare the results in Listing 2 with the results from when the code is compiled with optimization (shown in Listing 3).

Listing 3. Compiling and Running Code with Optimizations

```
% cc -g -O -mt listing1.c
% ./a.out
Thread 0 started
Thread 0 finished
Thread 1 started
Thread 1 finished
Thread 2 started
Thread 2 finished
Thread 3 started
Thread 3 finished
Thread 4 started
Thread 4 finished
Thread 5 started
Thread 5 finished
Thread 6 started
Thread 6 finished
Thread 7 started
Thread 7 finished
Thread 8 started
Thread 8 finished
Thread 9 started
Thread 9 finished
```

When the code is compiled with optimization, the threads start and complete in order. The reason for this is an optimization that the compiler performed on the routine `dowork()`. The optimized version of this routine is shown in Listing 4.

The compiler combined the two loops in the `dowork()` routine. This resulted in the main thread starting each child thread and then waiting until that child thread completes. This is in contrast to the desired behavior of starting all the child threads in parallel.

Listing 4. Compiler Optimized Version of the Routine dowork()

```
void dowork()
{
  for (int count = 0; count < 10; count++ )
  {
    start[ count ] = 1;  // Start thread working
    while( ended[ count ] == 0 ) {} // Wait until thread completes work
  }
}
```

Volatile Variables Are not Necessarily the Best Solution

It is worth discussing the use of the keyword `volatile` in this code. The keyword `volatile` means that a value needs to be loaded from memory before use and stored back to memory immediately after use. This behavior is useful because it ensures that loops, such as the one in Listing 5, are not converted to infinite loops during optimization.

Listing 5. Loop that Could Be Converted to an Infinite Loop During Optimization

```
while( ended[ count ] == 0 ) {} // Wait until thread completes work
```

However, using volatile variables is not without costs. If the variable is used as part of a computation, its value cannot be carried in a register; it needs to be refreshed at each use. So there is some performance impact.

The use of volatile variables does not make code "correct." It essentially reduces the optimization of code that uses the variables. For example, the code in Listing 6 contains a data-race condition regardless of whether the variable is declared as volatile.

Listing 6. Volatile Variables Do not Protect Against Data Races

```
void increment(int * variable)
{
   (*variable)++;
}
```

Finally, as we saw in the example from Listing 1, volatile variables do not protect against side effects of optimization. The conclusion we should draw from this is that volatile variables are not a panacea; they might solve a subset of problems, but they do not solve all of them.

At this point it is appropriate to ask "What is required in order for the code to function correctly?" The answer to that is we need points in the code that the compiler cannot optimize around.

If we take the code in Listing 5, we want the compiler to have to reload the variable `ended[]` at every iteration of the loop, and we do not want the compiler to assume that the variable is invariant. Similarly, we want the compiler to avoid merging the two loops, as shown in the optimization in Listing 4. So what we need is a compiler barrier.

Complier Barriers

A compiler barrier is a sequence point. At such a point, we want all previous operations to have stored their results to memory, and we want all future operations to not have been started yet.

The most common sequence point is a function call. We can recode Listing 5 to use a function call to achieve the desired effect. This method is shown in Listing 7.

The call to `DoNothing()` can immediately return having, as its name suggests, done nothing. However, the compiler does not "know" that nothing was done; the compiler has to assume that the function call might have changed global data. Consequently, the compiler needs to reload the value of the variable `ended[]`.

Listing 7. Using a Function Call as a Compiler Barrier

```
while( ended[ count ] == 0 ) {DoNothing()} // Wait until thread completes work
```

There are two problems with this approach. The first problem is that it uses an unnecessary function call, and the code might be more efficient if the function call were not performed.

The second, more serious, problem is that under optimization, the call to `DoNothing()` might be in-lined. If this happens, the sequencing point is removed, and the loop is converted once again to an infinite loop.

The header file `<mbarrier.h>` introduces the `__compiler_barrier()` intrinsic, which is designed to provide the required functionality.

This intrinsic has two advantages. The first is that the barrier remains during optimization, and the second is that it does not generate any instructions. The `__compiler_barrier()` intrinsic can be used to provide the desired reloading of the variable `ended[]`, as shown in Listing 8.

Listing 8. Using the compiler_barrier() Intrinsic to Avoid Generating Infinite Loops During Optimization

```
// Wait until thread completes work
          while( ended[ count ] == 0 ) {__compiler_barrier()}
```

The `__compiler_barrier()` intrinsic can be used to avoid the requirement of declaring the variables volatile, and it can be placed between the two loops in the `dowork()` routine to ensure that the compiler does not merge them.

The code modified to use the compiler barrier is shown in Listing 9. The changes are indicated in bold.

Listing 9. Full Listing Modified to use the __compiler_barrier() Intrinsic

```
#include <stdio.h>
#include <pthread.h>
#include <mbarrier.h>

int   start[10]; // volatile removed
int   ended[10]; // volatile removed
pthread_t    threads[10];

void * work( void * param)
{
   int id = (int) param;
   // Use a compiler barrier to reload start[id] every iteration
   while( start[ id ] == 0 ){ __compiler_barrier(); } // Wait until work started
   printf( "Thread %i started\n", id );
   ended[ id ] = 1;            // Indicate that work completed
   printf( "Thread %i finished\n", id );
}
```

```
void dowork()
{
  for (int count = 0; count < 10; count++ )
  {
    start[ count ] = 1;  // Start thread working
  }
  // Use a compiler barrier to stop the two loops being merged
  __compiler_barrier();
  for (int count = 0; count < 10; count++ )
  {
   // Wait until thread completes work
   // Use a compiler barrier to reload ended[count] every iteration
    while( ended[ count ] == 0 ) { __compiler_barrier(); }
  }
}

int main()
{
  // Create team of threads
  for( int count = 0; count < 10; count++ )
  {
    start[ count ] = 0;
    ended[ count ] = 0;
    pthread_create( &threads[count], 0, work, (void*)count );
  }

  dowork();

  // Join team of threads
  for( int count = 0; count < 10; count++ )
  {
    pthread_join( threads[count], 0 );
  }
}
```

Conclusion

This article, which is part 1 of a two-part series, described how to use compiler barriers to stop the compiler from generating code that is incorrect due to reordered memory accesses.

Part 2 of the series discusses how memory barriers or memory fences can be used to ensure that the processor does not reorder memory operations.

Oracle is committed to developing practices and products that help protect the environment

Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2: Part 1, Compiler Barriers
September 2010
Author: Darryl Gove

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

SOFTWARE. HARDWARE. COMPLETE.