Avoiding and Identifying False Sharing Among Threads

<keywords: threading, cache coherence, data alignment, profiler, programming tools>

Abstract

In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance. This article covers methods to detect and correct false sharing.

This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in Figure 1. This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

The source line shown in red in the following example code causes false sharing:

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;

  #pragma omp for
  for (i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me];
}
```

There is a potential for false sharing on array `sum_local`. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of `sum_local` (the source line shown in red), which invalidates the cache line for all processors.

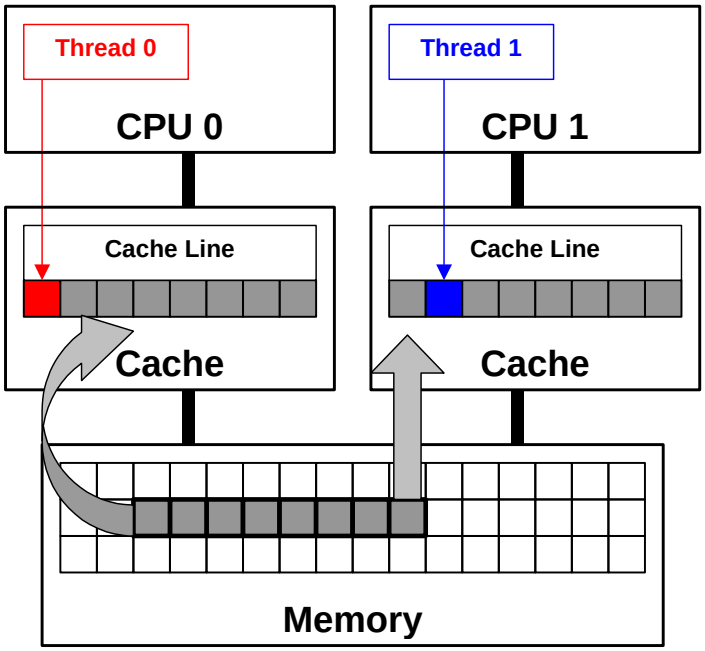

Figure 1. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency.

In Figure 1, threads 0 and 1 require variables that are adjacent in memory and reside on the same cache line. The cache line is loaded into the caches of CPU 0 and CPU 1 (gray arrows). Even though the threads modify different variables (red and blue arrows), the cache line is invalidated, forcing a memory update to maintain cache coherency.

To ensure data consistency across multiple caches, multiprocessor-capable Intel® processors follow the MESI (Modified/Exclusive/Shared/Invalid) protocol. On first load of a cache line, the processor will mark the cache line as 'Exclusive' access. As long as the cache line is marked exclusive, subsequent loads are free to use the existing data in cache. If the processor sees the same cache line loaded by another processor on the bus, it marks the cache line with 'Shared' access. If the processor stores a cache line marked as 'S', the cache line is marked as 'Modified' and all other processors are sent an 'Invalid' cache line message. If the processor sees the same cache line which is now marked 'M' being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as 'Shared'. The other processor that is accessing the same cache line incurs a cache miss.

The frequent coordination required between processors when cache lines are marked 'Invalid' requires cache lines to be written to memory and subsequently loaded. False sharing increases this coordination and can significantly degrade application performance.

Since compilers are aware of false sharing, they do a good job of eliminating instances where it could occur. For example, when the above code is compiled with optimization options, the compiler eliminates false sharing using thread-private temporal variables. Run-time false sharing from the above code will be only an issue if the code is compiled with optimization disabled.

Advice

The primary means of avoiding false sharing is through code inspection. Instances where threads access global or dynamically allocated shared data structures are potential sources of false sharing. Note that false sharing can be obscured by the fact that threads may be accessing completely different global variables that happen to be relatively close together in memory. Thread-local storage or local variables can be ruled out as sources of false sharing.

The run-time detection method is to use the Intel® VTune™ Performance Analyzer or Intel® Performance Tuning Utility (Intel PTU, available at http://software.intel.com/en-us/articles/intel-performance-tuning-utility/). This method relies on event-based sampling that discovers places where cacheline sharing exposes performance visible effects. However, such effects don't distinguish between true and false sharing.

For systems based on the Intel® Core™ 2 processor, configure VTune analyzer or Intel PTU to sample the `MEM_LOAD_RETIRED.L2_LINE_MISS` and `EXT_SNOOP.ALL_AGENTS.HITM` events. For systems based on the Intel® Core i7 processor, configure to sample `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM`. If you see a high occurrence of `EXT_SNOOP.ALL_AGENTS.HITM` events, such that it is a fraction of percent or more of `INST_RETIRED.ANY` events at some code regions on Intel® Core™ 2 processor family CPUs, or a high occurrence of `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM` events on Intel® Core i7 processor family CPU, you have true or false sharing. Inspect the code of concentration of `MEM_LOAD_RETIRED.L2_LINE_MISS` and `MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM` events at the corresponding system at or near load/store instructions within threads to determine the likelihood that the memory locations reside on the same cache line and causing false sharing.

Intel PTU comes with predefined profile configurations to collect events that will help to locate false sharing. These configurations are "Intel® Core™ 2 processor family – Contested Usage" and "Intel® Core™ i7 processor family – False-True Sharing." Intel PTU Data Access analysis identifies false sharing candidates by monitoring different offsets of the same cacheline accessed by different threads. When you open the profiling results in Data Access View, the Memory Hotspot pane will have hints about false sharing at the cacheline granularity, as illustrated in Figure 2.

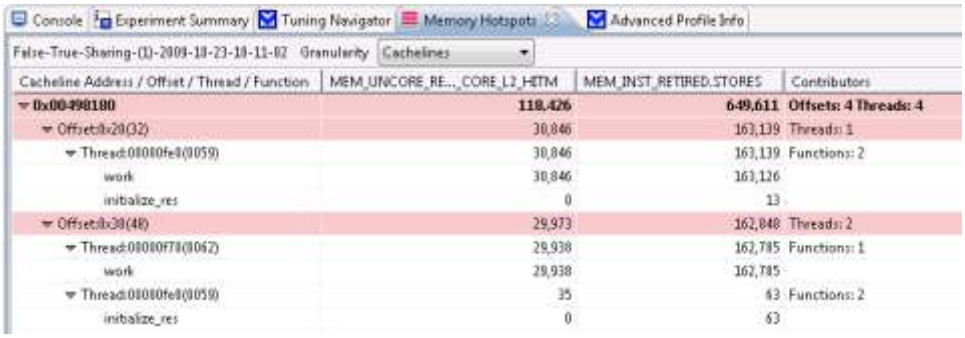

Figure 2. False sharing shown in Intel PTU Memory Hotspots pane.

In Figure 2, memory offsets 32 and 48 (of the cacheline at address 0x00498180) were accessed by the ID=59 thread and the ID=62 thread at the work function. There is also some true sharing due to array initialization done by the ID=59 thread.

The pink color is used to hint about false sharing at a cacheline. Note the high figures for MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM associated with the cacheline and its corresponding offsets.

Once detected, there are several techniques to correct false sharing. The goal is to ensure that variables causing false sharing are spaced far enough apart in memory that they cannot reside on the same cache line. While the following is not an exhaustive list three possible methods are discussed below.

One technique is to use compiler directives to force individual variable alignment. The following source code demonstrates the compiler technique using __declspec (align(n)) where n equals 64 (64 byte boundary) to align the individual variables on cache line boundaries.

```
__declspec (align(64)) int thread1_global_variable;
__declspec (align(64)) int thread2_global_variable;
```

When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary. If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line. The following source code demonstrates padding a data structure to a cache line boundary and ensuring the array is also aligned using the compiler __declspec (align(n)) statement where n equals 64 (64 byte boundary). If the array is dynamically allocated, you can increase the allocation size and adjust the pointer to align with a cache line boundary.

```
struct ThreadParams
{
  // For the following 4 variables: 4*4 = 16 bytes
  unsigned long thread_id;
  unsigned long v; // Frequent read/write access variable
  unsigned long start;
  unsigned long end;

  // expand to 64 bytes to avoid false-sharing
  // (4 unsigned long variables + 12 padding)*4 = 64
  int padding[12];
};

__declspec (align(64)) struct ThreadParams Array[10];
```

It is also possible to reduce the frequency of false sharing by using thread-local copies of data. The thread-local copy can be read and modified frequently and only when complete, copy the result back to the data structure. The following source code demonstrates using a local copy to avoid false sharing.

```
struct ThreadParams
{
  // For the following 4 variables: 4*4 = 16 bytes
  unsigned long thread_id;
  unsigned long v; //Frequent read/write access variable
  unsigned long start;
  unsigned long end;
};

void threadFunc(void *parameter)
{
```

```
  ThreadParams *p = (ThreadParams*) parameter;
  // local copy for read/write access variable
  unsigned long local_v = p->v;

  for(local_v = p->start; local_v < p->end; local_v++)
  {
    // Functional computation
  }

  p->v = local_v;  // Update shared data structure only once
}
```

Usage Guidelines

Avoid false sharing but use these techniques sparingly. Overuse can hinder the effective use of the processor's available cache. Even with multiprocessor shared-cache designs, avoiding false sharing is recommended. The small potential gain for trying to maximize cache utilization on multi-processor shared cache designs does not generally outweigh the software maintenance costs required to support multiple code paths for different cache architectures.

Additional Resources

Intel® Software Network Parallel Programming Community

Intel® VTune™ Performance Analyzer

Intel® Performance Tuning Utility