



Reduce False Sharing in .NET*

**by Dean Chandler, Senior Software Engineer,
Intel Corporation**

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation

Introduction

The Microsoft .NET* Framework offers a fast and easy way to design and implement your software in a hardware agnostic environment. This allows you to write software with the expectation that it will run optimally for the current platform (single or multiprocessor). However, this expectation--like most expectations for technology--is not always true.

As many long-time developers for Windows* know, software that performs well in a single threaded environment does not always perform as expected in multiprocessor/multithreaded environments. There are many non-platform specific issues with multithreaded applications running on multiprocessor systems. One of the most well known issues is that the processor cache line reloads due to what is known as false sharing. These misses effectively undo the scaling benefits of adding additional processors to systems. They force the memory subsystem to reacquire data into the processor cache for every memory load/store a worker thread performs on certain types of global data.

Unfortunately, the .NET Framework hides most of the logic that would let you overcome false sharing in a native environment. Memory management, both allocation and garbage collection, is handled inside the .NET Framework with very little exposure for you. While this attempts to support a *write once, run on many platforms* approach, the tradeoff is a significant scaling and performance impact on your multithreaded code. This paper discusses methods to reduce the impact of false sharing by using standard techniques in C# code and how to detect false sharing using the VTune™ Performance Analyzer.

Multiprocessor Cache Coherency

For systems with multiple processors, hardware vendors need to ensure data coherency across all processors. Data coherency ensures that the data in all processor caches are updated when a single processor writes to the back up memory. This implies that there is a memory protocol that prevents one processor from overwriting data that another processor is using in its cache. Intel multiprocessor systems use the **Modified - Exclusive - Shared - Invalid (MESI)** protocol. This protocol ensures cache coherency of the internal cache lines of each processor.

When you first load the program data into a processor's cache line, MESI requires the processor to mark the cache line with *Exclusive (E)* access, which gives that processor unlimited load/store access to the data on the cache line. However, if another processor requires access to the same data that already resides in another processor's cache, MESI requires the processor to mark this line as *Shared(S)*. For all subsequent stores to that cache line from any processor, the cache line is marked as *Modified (M)*, which causes all processors with that cache line to *Invalidate (I)* the cache line and reload it from system memory. Since modern cache architecture loads the complete cache line from system memory the line can contain many variables. Figure 1 displays how two distinct variables that reside next to each other in system memory can be loaded into adjacent locations on a processors cache line causing the processor to mark the whole line as shared and invalidate the line for each load/store.

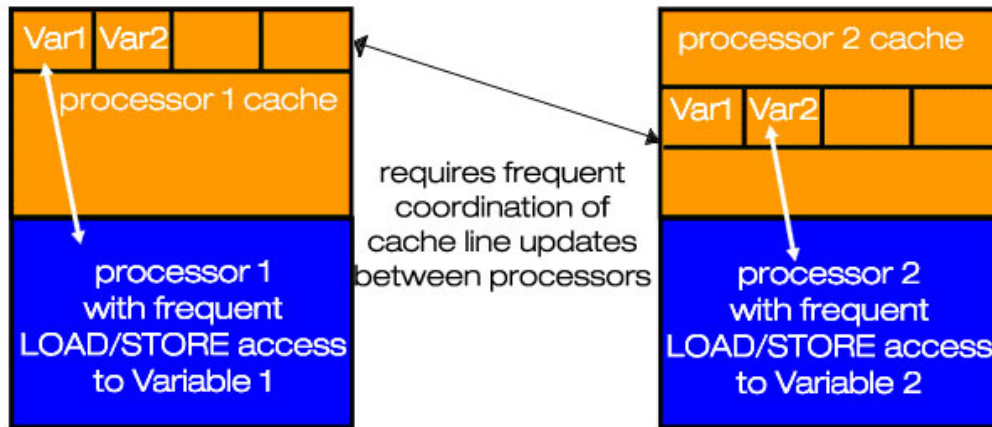


Figure 1

False Sharing

False sharing is a well known affect of systems with multiple processors, and is caused by multiple threads of the same program trying to load/store data from the same cache line. This occurs most often when different elements of a shared global structure occur inside a function that is used concurrently on multiple threads on different processors. A common data type that causes this problem is the *point* type supported by the .NET Framework. This type is simply a structure with two members, usually integer types. If you create a public global array of the point structure to be used in a function that will be threaded across multiple processors, adjacent members will have a high likelihood of causing false sharing as they are updated inside a loop. This updating will force the invalidation of the cache line for each processor every time a point element is modified from the same cache line.

This occurs because the MESI protocol will invalidate shared cache lines on different processors for each load/store of an element in a shared cache line. Figure 2 graphically depicts how this occurs. Upon first usage of the point array, each CPU loads the point array into one of its cache lines and starts modifying the point array element that it was assigned to use. If the thread running on CPU 0 is assigned element 0 of the point array and the thread running on CPU 1 is assigned element 1 of the array, false sharing penalties occur upon each CPU's modification of the element. As far as the system is concerned, all elements of the point array are on the same logical cache line and must follow the MESI protocol to invalidate the whole cache line every time the CPU modifies its point element. This invalidation of the cache line occurs for every load/store from separate CPU cache lines for any variable that happens to reside on that cache line, which is why the term false sharing arises: it is not sharing the same variable that would require locking semantics to protect the data. This affect can cause a large amount of system memory load/stores, which cause large performance/scaling degradation.

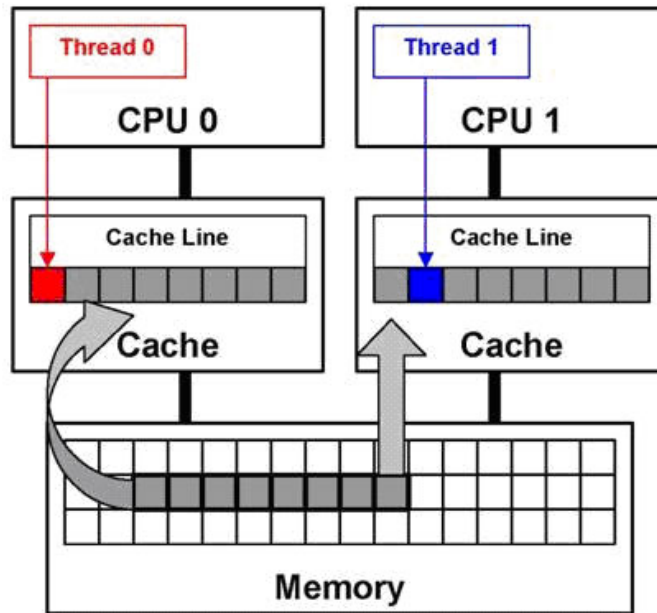


Figure 2

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. This is illustrated in the diagram (top). 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. This forces a memory update to maintain cache coherency.

Detecting False Sharing in Managed Code

There are several ways to detect false sharing inside managed source code. First ensure that the code does not have instances where threads access globally or dynamically allocated shared structures residing on the same cache line. In order to perform this check, research how the cache lines are created for the processor and platform. For instance, cache lines for Intel® Xeon™ processor are laid out in 128 byte lines that always start at an address that is a multiple of 0x80 or with seven LSB of the address as zeros. So, by determining where the .NET runtime places the variable by inspecting memory addresses in the debugger, you can ensure that the variables do not share the same address range (such as 0x0c123440 through 0x0c123480). Be aware that the .NET runtime has full control of the variables that are created in memory, so seemingly unrelated global variables could end up on the same cache line that a threaded function is modifying. The .NET garbage collector can also move global variables around that could cause false sharing. In general, all local variables inside the threaded function can be ruled out as a cause of false sharing.

A more efficient way to determine if false sharing is present is to use the VTune Performance Analyzer. For multiprocessor systems, configure the VTune analyzer to sample the "2nd Level Cache Load Misses Retired" event. For a Pentium® 4 processor with Hyper-Threading Technology (HT Technology), configure VTune analyzer to sample the "Memory Order Machine Clear" event. If there

is a high occurrence and concentration of these events at or near load/store instructions within the threaded function, there is likely false sharing. Inspect the code to determine the likelihood that the memory locations reside on the same cache line and that other CPUs are using the same cache line. The next section discusses the VTune analyzer data in detail.

Fixing False Sharing in Managed Code

The goal for fixing any false sharing-related issue is to ensure that the global variables that are modified inside threaded functions are created on separate cache lines. This can be difficult when working inside a managed runtime like the .NET Framework, as there are no obvious ways to align the code to a cache line to ensure that the variables do not reside on the same cache line. One of the most common techniques to solve false sharing issue is to copy the global variables to a local function variable and copy the data back before the function exits. This ensures that the variables are created on separate cache lines because they are part of the thread local storage and will be assigned addresses offset by the threads stack address. Another method to avoid false sharing is to ensure that data structures are large enough that members of the structure take up a whole cache line. Use this technique sparingly since it can quickly consume the memory resources.

The source code in Figure 3 displays a sample of C# .NET source code that creates false sharing issues (see *Appendix A* for a full source listing). The code shows the creation of a class with a simple line function and a point array structure to contain the point generated along this line. The problem with this for multithreaded programs is that as CalcPnt is called by more than one thread, the plData array elements are modified on the same cache line. This is guaranteed to cause false sharing as CalcPnt modifies plData's elements X and Y.

```
class FSharCls
{
    public Point[] plData;

    public void CalcPnt( )
    {
        int x;
        plData[locThreadNum].X = numIter/2*-1;

        for ( x=1; x<numIter; x++ )
        {
            plData[locThreadNum].Y = lnSlp*plData[locThreadNum].X++
                + locLnOffSt;
        }
    }
}
```

Figure 3

VTune analyzer data for this code has been collected and summarized by the thread as displayed in Figure 4. The data shows a large number of memory order machine clears, which is indicative of a problem with false sharing. As the test was run on a Pentium 4 processor with HT Technology, VTune was configured to collect data on the memory counter instead of the second level cache counter, which will show a high sample count for multiprocessor or multi-core platforms. Figure 5 displays a view of the source code of the data taken for this sample, giving you a preview of which lines are causing the false sharing issues.

Thread	Clockticks samples (118)	Instructions Retired samples (118)	Memory Order Machine Clear samples (118)	2nd Level Cache Load Misses Retired samples (118)	CPI
False Sharing Not Fixed					
thread6	27410	2824	8275	0	3.31
thread5	27198	2789	8421	0	3.23

Figure 4

	False Sharing Source	Clockticks (118)	Instructions Retired (118)	Memory Order Machine Clear (118)
False Sharing Not Fixed				
	29 public void CalcPrint()			
	30 {			
	31 int x;			
0x1404	32 int locLnOffst = lnOffst;			
0x1412	33 int locThreadNum = threadNum;			
0x1419	34 pData[locThreadNum].X = numIter/2*-1;			
	35			
0x1437	36 for (x=1; x<numIter; x++)	17,858	1,711	7,450
	37 {			
0x1445	38 pData[locThreadNum].Y = lnSlp*pData[locThreadNum].X++	9,465	1,106	971
	39 + locLnOffst;			
	40 }			
0x1472	41 }			
	42 }			
	43			

Figure 5

One solution is to create a unique global data structure and use .NET's StructLayout Attribute class, which can both pad the individual elements using the LayoutKind.Explicit member and then define the location of the structure's elements and overall size of the structure as shown in Figure 6. Since the structure is 256 bytes in size, each element of the array will be at least 128 bytes separated in memory, thus getting rid of any false sharing issues for this class member. For the Intel Xeon processor, 256 byte structure size works best because .NET aligns data structures on 8 bytes; this generally causes a data structure of 128 bytes to span cache lines, which leads to the possibility of false sharing for some elements of the structure. While this works well for the Intel Xeon processor with a 128 byte cache, ensure that the size is appropriate for the targeted multiprocessor platforms.

```

[StructLayout(LayoutKind.Explicit, Size=256)]
public struct _pDataFX
{
    [FieldOffset(0)] public int X;
    [FieldOffset(4)] public int Y;
}

class FSharCls
{
    public _pDataFX[] pData;

    public void CalcPnt ( )
    {
        int x;
        pData[locThreadNum].X = numIter/2*-1;

        for ( x=1; x<numIter; x++ )
        {
            pData[locThreadNum].Y = lnSlp*pData[locThreadNum].X++
                + locLnOffSt;
        }
    }
}

```

Figure 6

The thread and source sample data for the VTune analyzer are shown in Figure 7 and Figure 8. As expected when you get rid of false sharing data structures, the sample counts for the memory counter go to zero. Also, the reduction of the clock ticks samples and the increase in instructions retired means a decrease in CPI, which means the processor is running more efficiently.

Thread	Clockticks samples (118)	Instructions Retired samples (118)	Memory Order Machine Clear samples(118)	2nd Level Cache Load Misses Retired samples (118)	CPI
False Sharing Fixed					
thread6	5619	3410	0	0	0.61
thread5	5612	3412	0	0	0.61

Figure 7

	False Sharing Source	Clockticks (118)	Instructions Retired (118)	Memory Order Machine Clear (118)
	False Sharing Fixed			
	29 public void CalcPnt()			
	30 {			
	31 int x;			
0x1404	32 int locLnOffSt = lnOffst;			
0x1412	33 int locThreadNum = threadNum;			
0x1419	34 pData[locThreadNum].X = numiter/2 ² -1;			
	35			
0x1437	36 for (x=1; x<numiter; x++)	1,321	810	
	37 {			
0x1445	38 pData[locThreadNum].Y = lnSlp*pData[locThreadNum].X++	4,282	2,539	
	39 + locLnOffSt;			
	40 }			
0x1472	41 }			
	42 }			

Figure 8

Implementing this change to the source code in Appendix A allowed an improvement in the performance of this function by 10x using the new data structure. Figure 9 shows how the scaling improves from 1 to 8 processors.

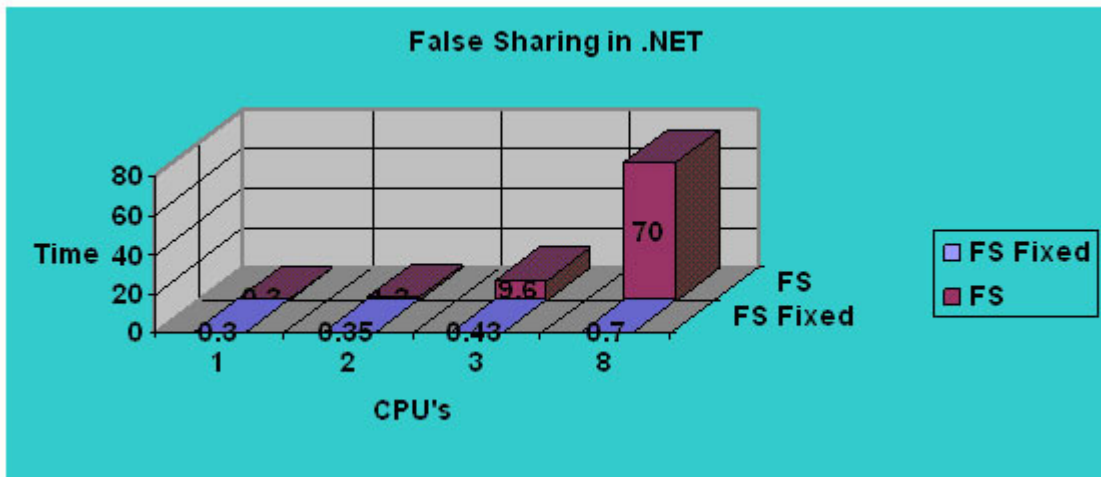


Figure 9

Summary

The .NET Framework improves the development process in many ways. However, be aware of how data structures can affect platform performance. Knowing how false sharing affects threaded application performance and how to fix these problems will reap large benefits as the application is scaled up from dual-processor to multi-processor systems. Developing for the .NET Framework gives you different ways to fix the same problems. For instance, to align a structure for a native C++ compiler, use a Declspec align preprocessor. In the .NET Framework, use the Structure Layout class to do the same.

References

[How to Avoid False Sharing](#)

[Developing Platform Consistent Multithreaded Applications](#)

[Intel Developer Services Threading Community Forum](#)

About the Author

Dean Chandler is a Senior Software Engineer in the Intel Software and Solutions Group. His current assignment is on-site at Microsoft's Redmond campus working with Microsoft's CRM product group to help with scalability and performance of the upcoming CRM 2.0 release. He has also been involved with performance tuning of the Microsoft Office System product. Before joining Intel, Dean was a Software Engineer at Compaq.

Appendix A

```
using System;
using System.Drawing;
using System.Threading;
using System.Runtime.InteropServices;

namespace flsShar
{
    [StructLayout(LayoutKind.Explicit, Size=256)]
    public struct _plDataFX
    {
        [FieldOffset(0)] public int X;
        [FieldOffset(4)] public int Y;
    }

    class FSharCls
    {
        public int numIter;
        public int lnSlp;
        public int lnOffst;
        public int numThreads;
        public int threadNum;

#if prfFx
        public _plDataFX[] plData;
#else
        public Point[] plData;
#endif

        public void CalcPnt()
        {
            int x;
            int locLnOffSt = lnOffst;
            int locThreadNum = threadNum;
            plData[locThreadNum].X = numIter/2*-1;

            for ( x=1; x<numIter; x++ )
            {
                plData[locThreadNum].Y = lnSlp*plData[locThreadNum].X++
                    + locLnOffSt;
            }
        }
    }

    class mnFSCls : unMngFunc
    {
        [STAThread]
        static void Main(string[] args)
        {
            long myTime=0;
            long prcFreq=0, prcTime0=0, prcTime1=0;
            string perfFnc="NF";
            Thread [] pntThrd;
            FSharCls FS = new FSharCls();
            FS.lnSlp = 1;
        }
    }
}
```

```

        FS.threadNum = 0;
        FS.numIter = 60000000;
        FS.numThreads = Convert.ToInt32 (
System.Environment.GetEnvironmentVariable
("NUMBER_OF_PROCESSORS"));
        QueryPerformanceFrequency( ref prcFreq );

#if prfFx
        FS.plData = new _plDataFX[FS.numThreads];
        perfFnc="FX";
#else
        FS.plData = new Point[FS.numThreads];
#endif

        pntThrd = new Thread[FS.numThreads];

        for ( int x=0; x<FS.numThreads; x++ )
            pntThrd[x] = new Thread( new ThreadStart( FS.CalcPnt ));

        Console.WriteLine ( " Running FS with {0} iterations \n", FS.numIter );
        Console.WriteLine ( " Using {0} threads on {1} Procs\n", FS.numThreads,
            FS.numThreads );
        Console.WriteLine ( " Using Perf function: {0}\n", perfFnc );

        QueryPerformanceCounter( ref prcTime0 );

        VTResume();

        for ( int x=0; x<FS.numThreads; x++ )
        {
            FS.InOffst = FS.threadNum;
            pntThrd[x].Start();
            Sleep( 20 );
            FS.threadNum++;
        }

        for ( int x=0; x<FS.numThreads; x++ )
            pntThrd[x].Join();

        VTPause();

        QueryPerformanceCounter( ref prcTime1 );
        myTime=((prcTime1-prcTime0)*1000)/prcFreq;

        Console.WriteLine( "
Time to run fshar is {0}ms \n Start Clk {1} \n End clk {2}\n",
            myTime.ToString(), prcTime0.ToString(), prcTime1.ToString() );
    }
}

```

```

public class unMngFunc
{
    [DllImport("D:\\Program Files\\Intel\\VTune\\Analyzer\\Bin\\vtuneapi.dll",
        EntryPoint="VTPause")]
    public static extern void VTPause();

    [DllImport("D:\\Program Files\\Intel\\VTune\\Analyzer\\Bin\\vtuneapi.dll",
        EntryPoint="VTResume")]
    public static extern void VTResume();

    [DllImport("kernel32", EntryPoint="Sleep")]
    public static extern void Sleep(long dwMilliseconds);

    [DllImport("kernel32", EntryPoint="GetTickCount")]
    public static extern long GetTickCount();

    [DllImport("kernel32.dll")]
    public extern static short QueryPerformanceCounter(ref long x);

    [DllImport("kernel32.dll")]
    public extern static short QueryPerformanceFrequency(ref long x);
}
}

```

