Analysis of False Cache Line Sharing Effects on Multicore CPUs

Suntorn Sae-eung, M.S.
suntorn.saeeung@students.sjsu.edu
(408) 775-9883

Robert Chun, Ph.D.
Computer Science Department
San Jose State University
San Jose, CA, 95192-0249 USA
robert.chun@sjsu.edu
(408) 924-5137

ABSTRACT

False Sharing (FS) is a notorious problem occurring in multiprocessor systems. It results in a performance degradation for multi-threaded programs. Since the architecture of a multicore processor is very similar to that of a multiprocessor system, the presence of the false sharing problem is speculated. Its effects should be measurable in terms of efficiency degradation in a concurrent environment on multicore systems.

This article discusses the causes of the false cache line sharing problem in dual-core CPUs, and demonstrates how it lessens the system performance by measuring speed-ups and efficiency of the experiments in sequential compared to parallel executions. Thus, demonstration programs are developed to collect the execution results of the test program with and without false sharing on the specific system hardware. Certain techniques are implemented to eliminate false sharing. These techniques are described, and their effectiveness in mitigating the speed-up and efficiency lost from false sharing is analyzed.

KEY WORDS

False Sharing, Cache Memory, Spacing, Padding

1.0 Introduction

The current trend of processor design is towards multicore CPUs. Recently, eight-core and twelve-core CPUs have been in the manufacturing process for both AMD and Intel [1]. Processor manufacturers overcome the heat-wall constraint by packing more than one computing module, so-called cores, into a package. Sometimes the chip is simply referred to as a Chip Multiprocessor (CMP); however, a processor can also be coined by the number of its cores. For example, a two core processor is called as a "dual core" CPU. Having many processing cores working together increases complexity in hardware design and software production. The hardware manufacturer is not the only party involved in taking advantage of the multiple core processors. Programmers must also understand how to make use of additional cores, and design the application by dividing processes into several sub-tasks, and assign them to several threads to utilize all available computing cores.

A potential problem in multiprocessor systems that can cause poor performance by mistakenly updating data in a shared cache line is the "*false sharing*" problem. Previous research on multiprocessor systems demonstrated huge impacts of the *false sharing* problem [5][6][8][9][12]. It can cause performance degradations of 20x on a four-processor system, and 100x on an 8-processor system.

Because multiprocessor and multicore architectures are similar, we hypothesize that FS can occur on multicore systems too. This paper demonstrates the existence of *false sharing* on systems with dual core CPUs, measures the impact of the false sharing issue, and compares the performance drops caused by *false sharing* between a dual core processor to a multiprocessor system.

1.1. Memory hierarchy and cache elements

Levels and types of memories are distinguished by their access time, capacities and complexities. Certain types of CPUs, along with their cache and main memory are selected as representatives to illustrate the memory hierarchy of multiprocessor and multicore systems. As *false sharing* is previously notorious in multiprocessor systems, the memory architecture of a Symmetric Multiprocessor (SMP) is compared with that of a Chip Multiprocessor (CMP).

1.1.1 Memory architecture in Symmetric Multiprocessor

The Symmetric Multiprocessor (SMP) is a classical configuration for a multiprocessor system. The memory hierarchy of SMP is categorized in two levels: cache memory and main memory. CPU access time, or latency, on the cache is far less than that from the main memory. Processors use the cache memory as a local memory, and consider the main memory to be a remote memory. CPUs need to request data through a shared network, bus, or crossbar in order to read from and write to the main memory. A simple diagram of a SMP system is shown in figure 1.

1.1.2 Memory architecture in Chip Multiprocessor

Chip Multiprocessor (CMP) is a way to name multicore processors. The cache in a CMP system is divided into tiers similar to a SMP, but a CMP's structure adds more layers of caches, e.g. a cache level 2, interleaving the L1 cache and the main memory so as to reduce the latency gap between the upper and the lower memory layers as shown in figure 2.

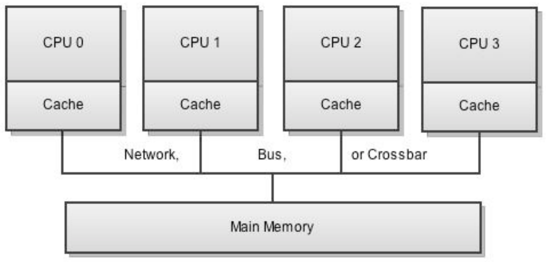

Figure 1. Memory hierarchy in SMP [4]

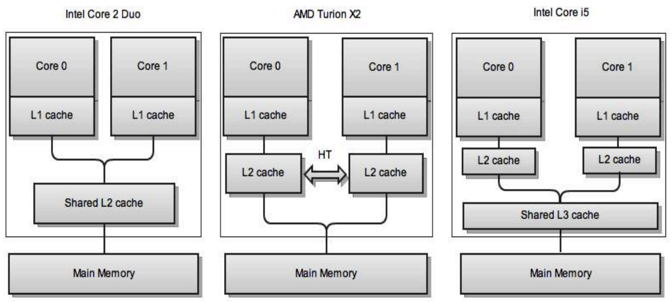

Figure 2. Memory hierarchy in CMP

The diagram shows three distinct layouts of caches. The Intel processor (left) implements a shared L2 and L3 cache enabling all cores to access shared data. AMD CPUs (middle) make use of a special dedicated hardware and protocol, Hyper Transport technology, to synchronize shared data between each core's L2 cache. Recently, a more advanced CPU, such as the Intel Core i5, a processor is composed of two levels of separate caches, and a shared L3 cache (right).

1.1.3 Cache lines

A cache line is the smallest unit of data that can be transferred between the main memory and the cache. The cache line size varies by processor makers; the size can be directly obtained from the processor's specification sheets, or retrived by executing some manufacturer-provided instruction sets. All processors in our test hardware have a 64-byte cache line size.

1.2 Multiprocessor/multicore cache coherency

For a multiprocessor system, all processors typically have their own caches, and machine vendors must ensure that data across processors are coherent. A protocol must be used to enforce data consistency among all the cores' caches so that the system correctly processes valid data; this protocol is called a "cache coherency" protocol. The protocol manages data to be updated appropriately using a write-back policy, resulting in decent overall performance by reducing the number of main memory updates.

1.3 False cache line sharing

False cache line sharing or *false sharing* in short is a form of cache trashing caused by a mismatch between the memory layout of write-shared data across processors and the reference pattern to the data. It occurs when two or more threads in parallel programs are assigned to work with different data elements in the same cache line. In other words, *false sharing* is a side effect in a multiprocessor system due to cache coherency.

Although the multiprocessor's system scale seems quite different from that of a small personal computer, the internal architecture of a multiprocessor is comparable to a multicore microprocessor chip in terms of the number of processors and memory hierarchy. A computer with dual-core, quad-core, or octal-core processors is now considered as a type of multiprocessor system. Thus, it would seem to be susceptible to a *false sharing* problem as well.

A multiprocessor system must maintain data coherency across CPUs. When a processor makes a change on its cache, other processors must be aware of the change, and determine whether its copy of data in cache needs to be reloaded or not. The cache coherency protocol defines rules to maintain data updates among processor groups with a minimal number of requests to the main memory, thereby optimizing system performance.

False sharing occurs when threads from different processors modify variables which reside on the same cache line. In case of Intel's processors, when the processor invalidates a cache line with an outdated value, it fetches an updated value from the main memory into its cache line to maintain data validity. Figure 4 and 5 demonstrate two threads with *false sharing* on SMP and CMP systems respectively. Threads 0 and 1 update variables that are adjacent to each other located on the same cache line. Although each thread modifies different variables, the cache line keeps being invalidated every iteration. As a result, the number of the main memory access increases considerably, and causes great delays due to the high latency in data transfers between levels of the memory hierarchy. The *false sharing* problem is ocassionally referred to as "cache line ping-pong [9]."

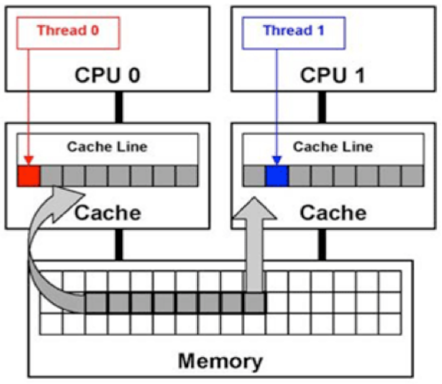

Figure 4. False cache line sharing on SMP [5]

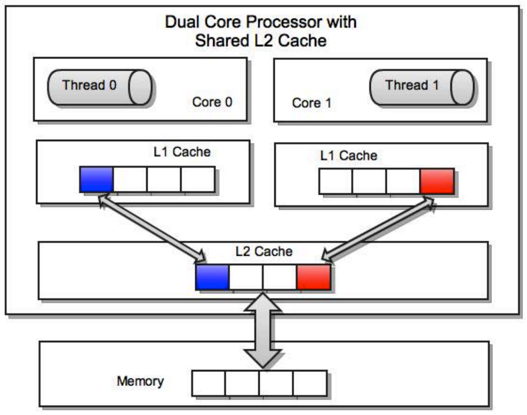

Figure 5. False cache line sharing on CMP [5]

2.0 Prior Work

Many researchers point out the great performance degradation caused by the *false sharing* problem on multiprocessor environments. Fewer papers performed tests on multicore CPUs since they are a relatively new architecture. The hypothesis in this paper is that *false sharing* would happen in a multicore system as it does in a multiprocessor one because it has many common components, yet the degree of impact may be different. More details will be discussed in the experiment and the result section.

2.1 Concurrent Hazards: False sharing

Butler did an experiment on a multiprocessor system to measure *false sharing* effects in [6]. The test system is composed of four dual core CPUs, total 8 processing units. The best case speed-up at the eight-threaded execution shows a 100x difference compared to the worst case. The paper employed certain techniques to eliminate *false sharing* effects.

2.2 Latency of conflict writes on Multicore Architecture

Dr. Josef discussed the latency penalty caused by FS [7]. The work shows that the amount of latency declines when the array allocated is between 128 Kbytes and 2Mbytes in size, which fits on cache level two. At this threshold of the array size, the high latency that would have been caused by the *false sharing* problem disappears. It is because shared L2 cache is a "true" sharing cache, and both cores can access data without cache invalidation, thereby eliminating *false sharing*. In brief, the experiment proved that shared cache between cores can eliminate the adverse impact stemming from *false sharing*.

3.0 Experiment Design

The experiment results are obtained from the execution time of a designated program onto three systems with different types of dual core CPUs. There are five test cases which have the same goal of completing an equally specified workload; however, different running schemes are set up to reveal the existence of the *false sharing* problem.

3.1 False sharing avoidance techniques

Since *false sharing* results from two or more cores using data in the same cache line, one way to get rid of it is to eliminate any sharing in the same cache line. Hence, certain techniques are proposed in order to avoid data sharing by modifying the data arrangement in the cache line.

3.1.1 Spacing technique

The Spacing technique is an approach used to split a contiguous allocated space. In an array, a set of variables is typically reserved in a chunk to take advantage of locality of reference. For instance, when four variables are declared in an array, an allocation consisting of four integer-sized adjoining memory blocks is made. Using the Spacing technique splits the shared data among the reserved array by shifting the offset between each contiguous array element so that each element resides on a separate, different cache line.

In figure 6a, integers D1, D2, D3 and D4 reside in the same cache line. With the implementation of the Spacing technique, *false sharing* on array data can be avoided as shown in figure 6b.

3.1.2 Padding technique

Besides the Spacing technique, Padding is another technique to reduce *false sharing* effects by filling a cache line with a pad.

A variable declaration requires an extra piece of information to manage memory space for the variable. When an array is declared, the operating system needs to define metadata that contains the array information. Metadata uses space just right before actual data, and consists of pointers and header information. For example, every array in .NET require metadata such as SZARRAY, which stores size information of the array. Whenever a thread read from or writes to an element, there is a read of the metadata happening before that of the actual array. Using Spacing technique does not separate metadata from the array; they still reside on the same cache line as in figure 6b. Therefore, *false sharing* is happening between the metadata and the first array element. To eliminate sharing on metadata, the cache line is padded so that the first element is shifted to the next cache line. Figure 6c illustrates the cache line structure using Padding.

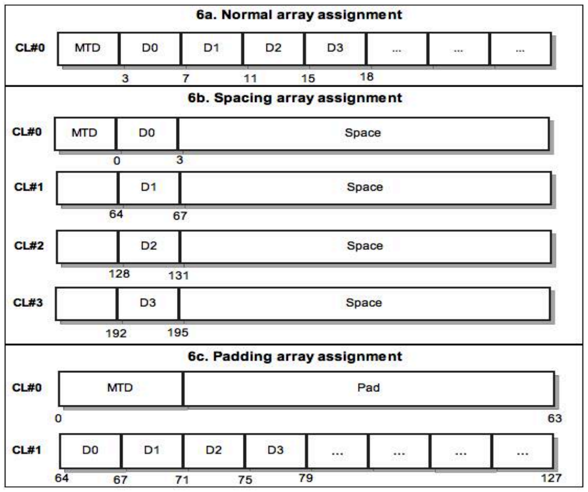

Figure 6. Cache line structure of Spacing, and Padding arrays

3.1.3. Combined Spacing and Padding technique

Using a Spacing-only or a Padding-only technique would not overcome the *false sharing* problem [6]. Therefore, the combination of both techniques is the best way to completely avoid *false sharing* by isolating each elements onto a single cache line. Figure 7, for example, shows a cache line layout of four array elements, including metadata.

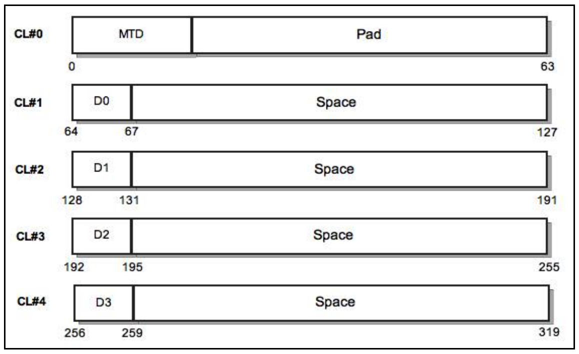

Figure 7. Cache line structure of combined Padding and Spacing technique

3.2 Testing code

The testing code, which is adapted from [6], demonstrates existence of the *false sharing* problem. The processing time of the program with the *false sharing* problem is compared to the program without the problem. The identical experiment is executed on three hardware configurations to compare the performance loss among different systems. The size of the Padding and Spacing variables are defined to be 64 bytes, which is equal to a size of one cache line, to ensure that every element is shifted off onto a separate cache line. The data arrangement is the crucial focus in order to avoid *false sharing* with five testing cases: SEQ, PAR, PAR_SPC, PAR_PAD and PAR_SPC_PAD.

The following code fragments show how each testing case declares the data array, sets an offset, and executes the workload.

```
var data = new int[ _Padding + ( _ThreadCount * _Spacing ) ];
...
var offset = _Padding + ( iThread * _Spacing );
...
for ( int x = 0 ; x < iters ; x++ ) data[ offset ]++;
```

For example, suppose that a system consists of a four core processor working on four integer elements; each core works on an array element. In the Parallel FS case (PAR), all four threads work on the contiguous array elements as shown in figure 11a. The array data is arbitrarily defined to start at the memory address 156. Generally an integer requires four bytes of memory space; therefore, four integers can be allocated in one cache line. Figure 8a show cache diagram of PAR case that *false sharing* occurs on cache lines. Meanwhile, figure 8b illustrates how PAR_SPC_PAD case avoids *false sharing* effects by isolating each array element onto separate cache lines.

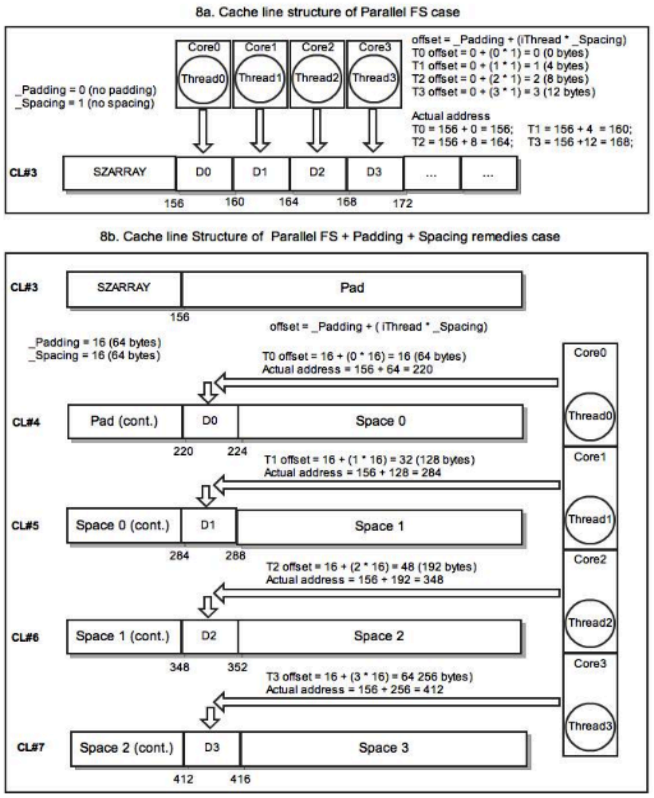
8a. Cache line structure of Parallel FS case
_Padding = 0 (no padding)
_Spacing = 1 (no spacing)
Core0
Core1
Core2
Core3
Thread0
Thread1
Thread2
Thread3
offset = _Padding + (iThread * _Spacing)
T0 offset = 0 + (0 * 1) = 0 (0 bytes)
T1 offset = 0 + (1 * 1) = 1 (4 bytes)
T2 offset = 0 + (2 * 1) = 2 (8 bytes)
T3 offset = 0 + (3 * 1) = 3 (12 bytes)
Actual address
T0 = 156 + 0 = 156; T1 = 156 + 4 = 160;
T2 = 156 + 8 = 164; T3 = 156 +12 = 168;
CL#3
SZARRAY
D0
D1
D2
D3
...
...
156
160
164
168
172
8b. Cache line Structure of Parallel FS + Padding + Spacing remedies case
CL#3
SZARRAY
Pad
156
offset = _Padding + (iThread * _Spacing)
_Padding = 16 (64 bytes)
_Spacing = 16 (64 bytes)
T0 offset = 16 + (0 * 16) = 16 (64 bytes)
Actual address = 156 + 64 = 220
CL#4
Pad (cont.)
D0
Space 0
220
224
T1 offset = 16 + (1 * 16) = 32 (128 bytes)
Actual address = 156 + 128 = 284
CL#5
Space 0 (cont.)
D1
Space 1
284
288
T2 offset = 16 + (2 * 16) = 48 (192 bytes)
Actual address = 156 + 192 = 348
CL#6
Space 1 (cont.)
D2
Space 2
348
352
T3 offset = 16 + (3 * 16) = 64 256 bytes)
Actual address = 156 + 256 = 412
CL#7
Space 2 (cont.)
D3
Space 3
412
416
Core0
Thread0
Core1
Thread1
Core2
Thread2
Core3
Thread3

Figure 8. Cache line structures of Parallel FS (PAR) and Parallel FS + Spacing and Padding remedies (PAR_SPC_PAD)

4.0 Hardware specifications

The experiment performs on three specified types of multicore processors: Intel Core2 Duo T5270, AMD Turion64 X2 TL-58, and Intel Core i5 520M.

Table 1. Processors' specifications of testing systems

Hardware	CPU	L1 Cache	L2 Cache	L3 Cache
Dell Vostro 1400	Intel Core2 Duo T5270 1.4GHz	32kbytes/core (Inst and Data)	Shared 2Mbytes	-
HP DV6000	AMD Turion64 X2 TL-58 1.9GHz	64kbytes/core (Inst and Data)	512kbytes/core (Unified)	-
Macbook Pro MC371LL/A	Interl Core i5 520M 2.4GHz	32kbytes/core (Inst and Data)	256kbytes/core (Unified)	Shared 3Mbytes

5.0 Experiment Results

The experiments results are collect and analyzed to understand how *false sharing* happens, and how much performance degradation it causes. The runtime values of five different test cases with varied data layouts and running schemes are collected. All five cases are assigned to complete the same amount of workload so that they can be compared in terms of performance. The details of data arrangement in each case are as follows.

1. Sequential (SEQ)—a sequential execution of the assigned workload on one core.
2. Parallel FS (PAR)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. There will be data contention in cache lines. The runtime on this case is expected to be influenced by *false sharing*.
3. Parallel FS + Spacing remedy (PAR_SPC)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. Additionally, this case applies the Spacing technique to avoid *false sharing* effects on the array elements.
4. Parallel FS + Padding remedy (PAR_PAD)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. This case implements the Padding technique to prevent *false sharing* occurring on the array metadata.
5. Parallel FS + Spacing and Padding remedies (PAR_SPC_PAD)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. Moreover, this case combines Spacing and Padding techniques so as to completely eliminate *false sharing* effects on the array elements and metadata.

5.1 Experiment results

Speed-up and efficiency are calculated from the runtime. Both numbers are computed as relative parallel performance based upon the sequential runtime by following equations [3].

$$\textit{Speed-up}(x) = \textit{sequential runtime} / \textit{parallel runtime} \qquad (1)$$

$$\textit{Efficiency}\ (\%) = (\textit{speed-up} / \textit{number of cores}) *100 \qquad (2)$$

Table 2 amasses all experiment results which are speed-ups, efficiency as well as performance degradation computed in terms of loss efficiency. Values of each hardware configurations are relatively compared based upon sequential execution figures.

Table 2. Experiment results summary

System \ Test cases	SEQ	PAR	PAR_SPC	PAR_PAD	PAR_SPC_PAD
Intel Core2 Duo T5270					
Speed-ups (x)	1.00	0.51	0.76	0.99	1.75
Efficiency (%)	100	25.48	37.85	49.43	87.58
Loss (%)	-	74.52	62.15	50.57	12.48
AMD Turion 64 X2					
Speed-ups (x)	1.00	0.50	0.73	0.63	1.97
Efficiency (%)	100	25.12	36.28	31.3	98.34
Loss (%)	-	74.88	63.72	68.7	1.66
Intel Core i5 520M					
Speed-ups (x)	1.00	0.57	1.06	0.98	2.17
Efficiency (%)	100	28.48	52.89	49.21	108.57
Loss (%)	-	71.52	47.11	50.79	0 (+8.57)

Intel Core2 Duo T5270

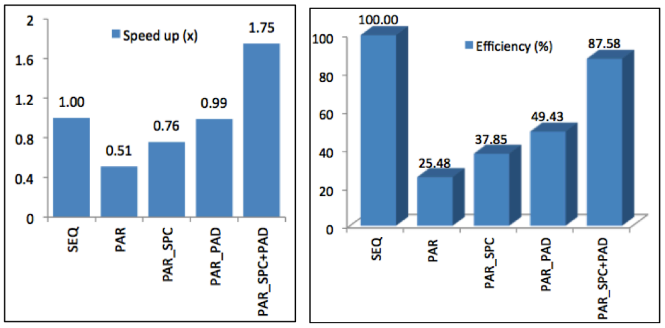

Figure 9. Intel Core2 Duo T5270 speed-ups and efficiency

AMD Turion64 X2 TL-58

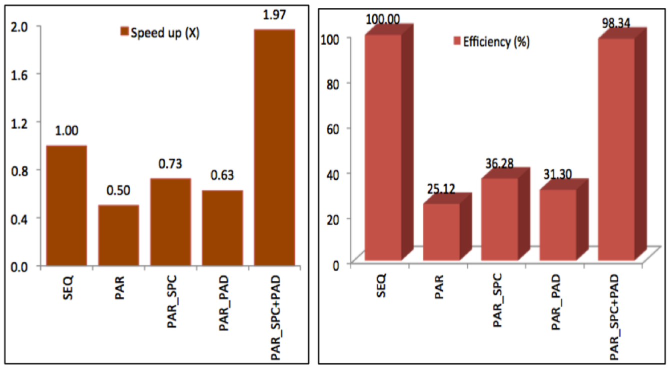

Figure 10. AMD Turion64 X2 TL-58 speed-ups and efficiency

Intel Core i5 520M

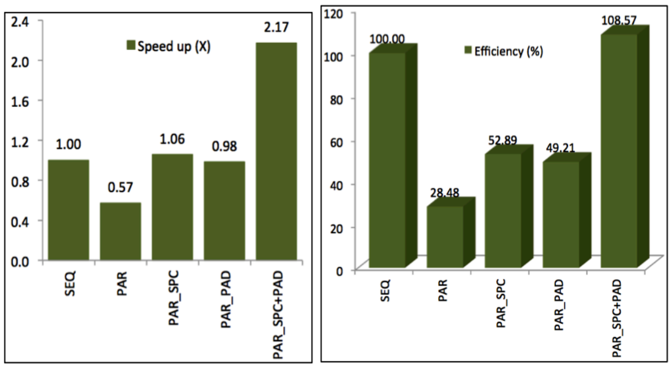

Figure 11. Intel Core i5 520M speed-ups and efficiency

5.1.1 Intel Core2 Duo T5270

Figure 9 (left) show speed-up ratios of the four parallel cases calculated based upon the Sequential case (SEQ) speed-up (1.0x).

The speed-up ratios demonstrate that *false sharing* has the most influences on PAR case execution (0.51x), and less impacts on the two cases with remedial techniques, PAR_SPC (0.76x) and PAR_PAD (0.99x). The PAR_SPC_PAD case obtains a practical value at 1.75x in speed.

Theoretically, two cores should accelerate system performance for two times (2x). However, the speed-up ratio in practical does not reach the theoretical value since some system resources are used to fork working threads, and synchronize data among those threads. A speed-up ratio range of 1.5x to 1.9x is considered practical in the level of parallelism with two processing cores [2].

Efficiency is a fairly good indicator to measure performance per processing unit. The Sequential case is a base value with 100% efficiency. For two cores working in parallel, the system must run two times faster than single core to achieve full efficiency. Figure 9 (right) shows the efficiency with a similar pattern to speed-up ratios, PAR at 25.48%, PAR_SPC at 37.85%, PAR_PAD 49.43%, and PAR_SPC_PAD at 87.58%. The amount of lost efficiency results from the different degrees of *false sharing* impact. The more false cache line sharing occurs, the lower performance it obtains.

5.1.2 AMD Turion64 X2 TL-58

Consider the speed-up of the PAR case, it does not scale well (0.5x) compared to the sequential case (1.0x). When the PAR case is employed with the Spacing technique to become the PAR_SPC, the speed-up augments to be 0.73x. The PAR_PAD also produces a greater speed-up (0.63x) compared to the PAR case as shown in figure 10 (left).

False sharing turns down speed-ups of the three mentioned cases in different degrees. However, the Parallel FS + Spacing and Padding remedies case (PAR_SPC_PAD) gains a promising speed-up at 1.97x, which is virtually close to an ideal value at 2.0x.

Among all parallel cases, only the PAD_SPC_PAD gains high efficiency at 98.34% as shown in figure 10 (right). The efficiency in any other cases reflects the different performance degradation by different degrees of *false sharing* effects.

5.1.3 Intel Core i5 520M

Figure 11 (left) shows speed-up ratios on the Intel Core i5 520M test system. The Parallel FS (PAR) case represents the poor performance execution with 0.57x in speed, or around two times slower than the sequential case. An improvement takes place on the PAR_SPC case (0.98x) and the PAR_PAD case (1.06x). The PAR_SPC_PAD case gains the highest speed-up at 2.17x.

Intel Core i5's efficiency has a similar pattern to two previous test systems. The efficiency of the PAR_SPC_PAD is noticeable with a "superlinear" number (108.57%), which efficiency exceeds 100% [3]. When a program which makes use of data stored in a share cache is repeatedly executed, its performance will substantially boost up because of memory locality. Another factor to reach a superlinear value is capablility of executing many concurrent threads. Intel Core i5 520M processor comes up with Hyper-Threading technology; each core can execute two threads at a time. Therefore, it increases probability for threads to take advantage of memory locality.

5.2 Performance drops caused by false sharing

From table 1, performance drops caused by *false sharing* is observed by efficiency losses.

The PAR case suffers from *false sharing* the most. The system performance drops by three fourth of the speculated efficiency, which caused efficiency loss 70-75%. The PAR_SPC case and the PAR_PAD case also have significant performance degradation approximate 50-70%, but less deficit compared to the PAR case. The PAR_SPC_PAD case performs efficiently, especially on the Core i5 520M processor. The case has a small number of losses on all three systems: Intel Core2 Duo T5270 at 12.48%, AMD Turion 64 X2 at 1.66%, and no loss for Intel Core i5 520M (8.57% in surplus).

5.3 False sharing impacts comparison on multiprocessor and dual core systems

The previous research points out the severity of the *false sharing* impact on multiprocessor systems in two orders of magnitudes (-100x) [6]. However, the experiment results in this paper demonstrate the worst case of performance degradation by a factor of four (-4x). An important observation is the degree of impact on a multiprocessor system is far aggressive than that on a dual core system. The suspicious factor is memory hierarchy.

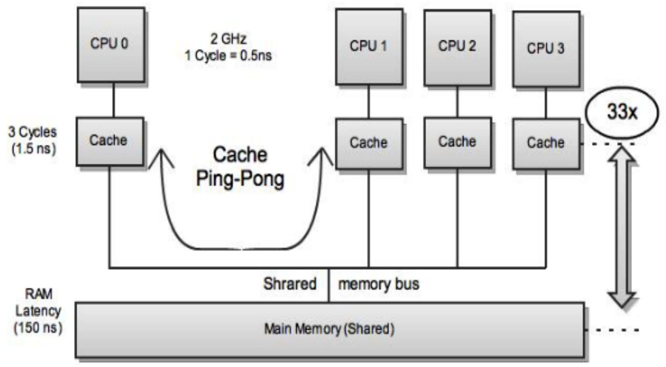

Figure 12. Cache Ping-ponging on multi-level memory in a multiprocessor system

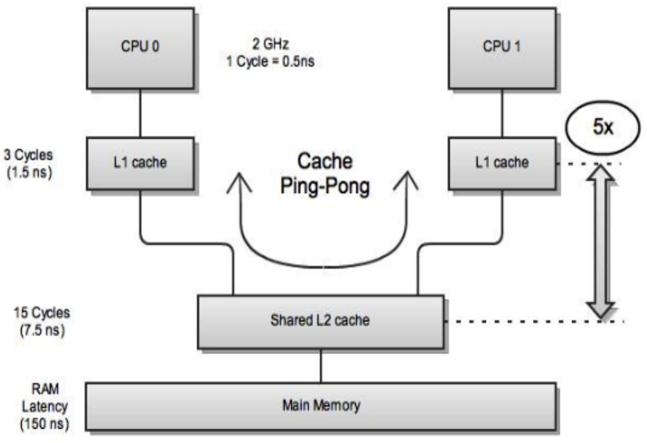

Figure 13. Cache Ping-ponging on multi-level memory in a dual core system

Figure 12 and 13 show multi-level memory hierarchies of a multiprocessor system and an Intel dual core processor system. Supposed that the program similar to the one that runs in the test experiment is executed in a multiprocessor system, *false sharing* occurs on the system. In the PAR case, the array elements in a cache line are updated by many processors; *false sharing* results in considerably numbers of cache line invalidation. When a processor writes a new value to its array elements, the whole cache line needs to be written back to the main memory, and reload to all processors' caches, known as cache Ping-Pong in figure 12. The CPUs' read and write operations befall between their caches and the (shared) main memory, in other words, between the cache and the main memory hierarchy. Since the processors need to access to the main memory through a shared bus, the system suffers from cache misses penalty. The amount of CPU waiting time substantially increases by the cache miss penalty as a following equation [10]:

Cache miss penalty (X bytes) = main memory access latency + X bytes/data receive rate ***(3)***

Cache miss penalty is computed by adding up a delay of main memory access and data transfer time from main memory to cache memory. The data transfer rate depends on the shared memory bus. Because the bus is used by all processors to access to main memory and peripheral devices, transfer time of the bus has much higher latency than that of an internal bus between caches and CPUs. Therefore, the substantial amount of increasing time caused by cache miss penalty results in significant performance reduction stemmed from the *false sharing* problem.

The similar scenario of *false sharing* occurs on a dual core system. Cache Ping-Ponging also happens in the system as shown in figure 13. Yet, the cache invalidation in the dual core system takes place in between the L1 cache and the shared L2 cache, instead in between the cache and the main memory in multiprocessor systems. The on-die caches are local memories having low latency. Data transfers among caches do not require bus transactions as data transfers between cache and main memory. Thus, the severity of *false sharing* on a dual core system does not cause significant performance degradation as it does on a multiprocessor system.

6.0 Conclusion

The study of *false sharing* effects on dual-core CPUs demonstrates the existence of *false sharing* on multicore CPUs. The issue apparently degrades overall performance in a concurrent execution.

(1) In the test case with false sharing occurs, PAR, on dual core processors, the efficiency degrades by approximately 70-75%. In other words, the test program works slower than speculated by four times; it runs at 25-30% efficiency instead of 100% efficiency.
(2) For the partially *false sharing* remedial cases, PAR_SPC and PAR_PAD, have certain runtime improvements to be 30-50% efficiency. However, the *false sharing* impact still stalls the two test cases, and leads to significant efficiency loss.
(3) On the best case, PAR_SPC_PAD, completely avoids *false sharing*, and obtains performance at nearly 100% efficiency.

All three test systems, Intel Core2 Duo T5270, AMD Turion64 X2, and Intel Core i5 520M processors, are consistently suffering from *false sharing* effects resulting in performance drops at 50%-75% efficiency.

On one hand, programmers can be optimistic for improvements on multicore CPUs since the ratio of performance drops caused by the *false sharing* problem on a dual core system is not as high as that on a multiprocessor system. The findings in this paper indicates performance of a dual core system drops approximately by a factor of four (-4x). Unlike the *false sharing* impact on a multiprocessor system, the previous research reported the performance loss as high numbers as one hundred times (-100x) on an eight processor system. The different degrees of the *false sharing* impacts stem from the different memory architectures between those two systems. The shared cache implementation on Intel dual core processors alleviates the adverse impact caused by *false sharing*. For AMD processors, although each core has a separate L2 cache which is subject to have *false sharing* problems, the processor handles the data synchronization among caches on all cores by using MOESI coherency protocol and dedicated data paths, the synergy of the two parts are named as Hyper Transport technology. In brief, both Intel and AMD have deliberately come up with the intelligent designs to cope with the data sharing issue across cores.

On the other hand, the programmers must still be aware of performance degradation caused by *false sharing*. For dual core processor system, a parallel version of the program working four times slower is considered unacceptable since it runs even slower than sequential version running on a single core processor. The *false sharing* problem, therefore, is a major potential issue in parallel programming on multicore CPUs.

We proposed and demonstrated implementation of Spacing and Padding techniques to avoid *false sharing*. These approaches remedy, or totally eliminate, the *false sharing* impact. Nevertheless, the implementation of Spacing and Padding techniques barters with memory space. For instance, on the dual core test systems, the amount of memory used in the PAR case is 8 bytes of the array plus the metadata size, which can be rounded up to be 16 bytes. The modified array size in the PAR_SPC_PAD case becomes three cache lines, or 192 bytes, two cache lines for two elements and one cache line for metadata. Thus, the cost to avoid *false sharing* is rather expensive.

7.0 Future Work

The processors with four cores, six cores, and eight cores will be a standard for personal computers in the foreseeable future. Also, the internal architecture of processors keeps changing to handle inter-core communication efficiently. For Intel Core-i7, data on each core is synchronized through inter-core connection paths known as Intel Quick Path technology [1]. AMD Phenom X4 Quad-core uses Hyper Transport 3.0 technology maximizing throughput to be 51.2Gbit/second [11]. All break-through technologies are invented to tackle data synchronization among cores. However, does the new cutting edge technology really work on all types of applications without the *false sharing* issue? If it does, that is good news for programmers. This paper shows the existence of *false sharing* on dual core CPUs, and it could imply that false sharing would still occur on a more-than-two-core processor. In case the problem does exist, how much is the impact on a quad core CPU? How much is the performance loss on an eight core or a sixteen core processor? The evaluation of the *false sharing* impact on such many cores CPUs will be subject to further research in the future.

8. References

[1] AMD, Intel ready 'many core' processors. Web site: http://news.cnet.com/8301-13924_3-10471333-64.html

[2] Pase, M. D., Eckl, M.A. 2005. A Comparison of Single-Core and Dual-Core Opteron Processor Performance for HPC. IBM Corporation. Web site: ftp://ftp.software.ibm.com/eserver/benchmarks/wp_Dual_Core_072505.pdf

[3] The Code Project. Butler, N. Superlinear: an investigation into concurrent speed-up. Web site: http://www.codeproject.com/KB/threads/Superlinear.aspx

[4] Loshin, D., Effective Memory Programming. McGraw-Hill.

[5] Chandler, D., Reduce False Sharing in .NET. Web site: http://software.intel.com/en-us/articles/reduce-false-sharing-in-net/

[6] The Code Project. Butler, N. Concurrent Hazards: False Sharing. Web site: http://www.codeproject.com/KB/threads/FalseSharing.aspx

[7] Weidendorfer, J., et al. 2007. Latencies of Conflicting Writes on Contemporary Multicore Architectures. *Springer Berlin Heidelberg*, vol. 4617, pp. 318-327.

[8] Bolosky, W. J., Scott, M. L. 1993. False sharing and its effect on shared memory performance. In USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4 (Sedms'93), Vol. 4. USENIX Association, Berkeley, CA, USA, 3-3.

[9] Cebix. Cache Line Ping-Pong. Web site: http://everything2.com/title/cache+line+ping-pong

[10] Adve, S. CS433g final exam Web site: http://www.cs.uiuc.edu/class/fa05/cs433g/assignments/Fall_2004_Final_Solution.pdf

[11] Hyper Transport Consortium. HyperTransport 3.1 Specification. Web site: http://www.hypertransport.org/default.cfm?page=HyperTransportSpecifications31

[12] Torrellas, J., Lam, H.S., Hennessy, J.L., False sharing and spatial locality in multiprocessor caches. In *Computers, IEEE Transactions*, vol.43, no.6, pp.651-663, Jun 1994. doi: 10.1109/12.286299