Fall 12-2010

Analysis of False Cache Line Sharing Effects on Multicore CPUs

Suntorn Sae-eung
San Jose State University

ANALYSIS OF FALSE CACHE LINE SHARING EFFECTS ON MULTICORE CPUS

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

by

Suntorn Sae-eung

December 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Project Titled

ANALYSIS OF FALSE CACHE LINE SHARING EFFECTS ON MULTICORE CPUS

by
Suntorn Sae-eung

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Robert Chun,	Department of Computer Science	Date
Dr. Suneuy Kim,	Department of Computer Science	Date
Dr. Soon Tee Teoh,	Department of Computer Science	Date

ABSTRACT

False sharing (FS) is a well-known problem occurring in multiprocessor systems. It results in performance degradation on multi-threaded programs running on multiprocessor environments.

With the evolution of processor architecture over time, the multicore processor is a recent direction used by hardware designers to increase performance while avoiding heat and power walls. To fully exploit the processing power from these multicore hardware architectures, the software programmer needs to build applications using parallel programming concepts, which are based upon multi-threaded programming principles.

Since the architecture of a multicore processor is very similar to 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 project discusses the causes of the false sharing problem in dual-core CPUs, and demonstrates how it lessens the system performance by measuring efficiency of a test program in sequential compared to parallel versions. Thus, demonstration programs are developed to read a CPU cache line size, and 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.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Robert Chun, who always gives counsel professionally. His guidance and constant motivation forms firm accomplishment for this project. Moreover, his suggestions help me overcome obstacles, and make me energetic to complete this work while feeling that it is my wonderful learning experience.

I appreciate time and efforts that my committees, Dr. Kim and Dr.Teoh, spend on reviewing, and giving feedbacks on my report as well as scheduling for the presentation.

I would like to thank you Dr. Josef for his profound answers to my queries, and source code with helpful advice.

I would like to thank the faculty members for constantly supporting and making my tenure at San Jose State University a pleasant experience.

I appreciate the generosity of my friends, Top, for lending computers and accessories used in the experiment. Without the hardware lending, this project would not have been completed. Also Nikky and Ben for his relentlessly proofread my writing.

Also, I want to deeply thank my wife who organizes a well-living place, cooks delicious meals, and encourages me to work at best on the project. Her embraces propel me at any time fulfilling with love whenever I tired from work, or made a mistake.

Most importantly, thank you to my parents who are always cheerful, encouraging, and supporting with all their love.

Table of Contents

Lists of Figures

Lists of Tables

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 are another party that must also understand how to make use of additional cores. They have to build software that divides work into many sub-tasks, and assign the tasks on several threads working on the multiple 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*" (FS) issue. Since a multiprocessor architecture could be considered a precursor of a multicore processor, the problem has a tendency to occur on a multicore system too.

Previous research on multiprocessor systems demonstrated the huge impact of the *false sharing* problem [7][8][19][28][29]. The problem can cause performance degradation by 20x on a system with four processors, and by 100x on a system with eight processors.

This project demonstrates the existence of *false sharing* on systems with dual core CPUs, introduces how to observe the problem and measure 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.

To understand the root causes of *false sharing*, some facts and theories are introduced for background:

- Directions of CPU technology and programming techniques
- Needs of parallel programming
- Memory hierarchy and cache elements
- Multiprocessor/multicore cache coherency
- False cache line sharing

1.1 Directions of CPU technology and programming techniques

The first dual core CPU was released in 2001 by IBM [30]. Nowadays, multicore CPUs are ubiquitous [2]. To increase computing performance, the processor makers pack more than a single processing core in one package. The processor is generally called a multicore or a many-core CPU. The processors are able to gain higher performance by using the sum of the computing capability of multiple cores. In 2010, a personal computer with a quad-core CPU has become a standard specification in the market, e.g. Intel core i7 processors, and AMD Phenom II X4 processors. Increasing the number of cores in a processor is expected to be an industrial trend used to augment processing power for decades. For instance, Intel's roadmap announced that they are now developing an eighty-core CPU [3].

1.2 Needs of parallel programming

Section 1.1 shows that the current processor's trend is many-cored; however, most legacy applications were designed to work sequentially on a single processor. Though the applications are compatible with multicore processors, they cannot make use of the extra cores. In fact, the additional cores are not just rendered useless, they even contribute to waste due to their extra power consumption.

To take advantage of multicore processors, it is mandatory to transform the sequential software to a parallel version, or newly rebuild it as a concurrent application. Nonetheless, parallel programming knowledge is essential for both alternatives.

1.3 Memory hierarchy and cache elements

This section discusses concepts of 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 many-cored processor systems. As *false sharing* is previously notorious in multiprocessor systems, memory architecture of a Symmetric Multiprocessor (SMP) is compared with that of a Chip Multiprocessor (CMP).

1.3.1 Memory architecture in Symmetric Multiprocessor

Symmetric Multiprocessor or SMP is a classical configuration for a multiprocessor system. A simple diagram of an SMP is shown in figure 1.

In SMP configurations, the memory hierarchy 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.

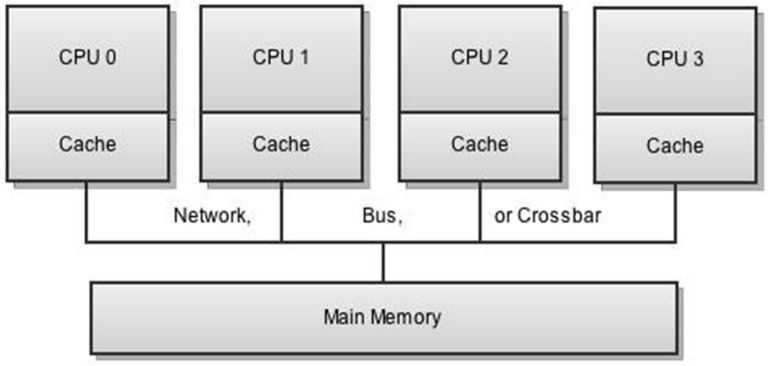


Figure 1. Memory hierarchy in SMP [4]

1.3.2 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 SMP, yet 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 layers as shown in figure 2.

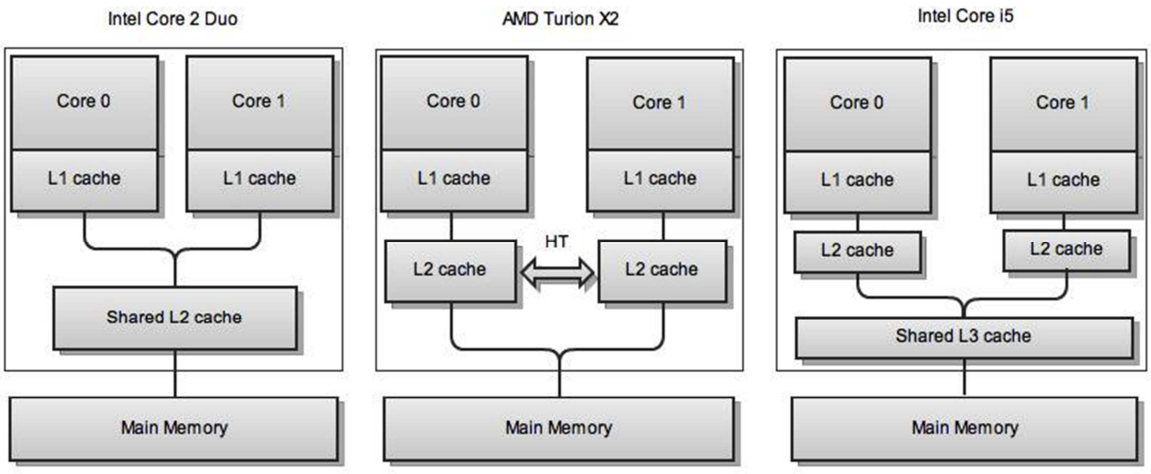


Figure 2. Memory hierarchy in CMP [5]

The diagram shows three distinct layouts of caches. The Intel processor implements a shared L2 and L3 cache enabling all cores to access to shared data (left). The AMD CPUs have a special dedicated hardware to synchronize shared data between each core's L2 caches (middle). This technology is AMD Hyper Transport technology. For 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).

1.3.3 Cache line

A cache line is the smallest unit that can be transferred between the main memory and the cache. The size of a cache line can be determined from the CPU specifications, or directly retrieved from the processor by using the manufacturer's instruction set. In this project, the cache line size of the Intel Core2 Duo T5270, the AMD Turion 64 X2 and Intel Core i5 520M is 64 bytes. Figure 3 magnifies how a cache line resides on the Intel Core2 T5270 processor. A program code to read the cache line size for the Intel processor is shown in appendix A.

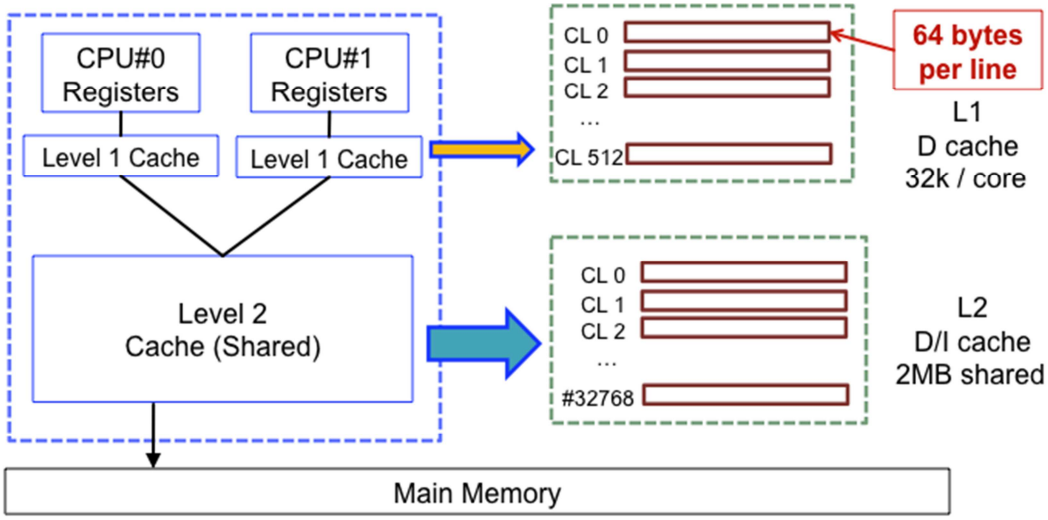


Figure 3. Cache line details of Intel Core2 T5270 processor [2]

1.4 Multiprocessor/multicore cache coherency

In systems consisting of two or more processors, each one typically has its own cache, 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.

Consider an example case of coherency. If CPU1 updates a variable named Z from 50 to 60, and CPU2 reads Z, what will happen to the cache of each CPU? At first, both CPUs have Z values as 50 in their caches. Then, CPU1 updates Z to be 60. Employed with the write-back policy, CPU1's cache does not need to immediately update the new value to the main memory. Therefore, the Z values in the main memory and CPU2's cache remains 50. In case CPU2 needs to read Z, it is mandatory for CPU1 to write the value 60 back to the main memory, and reload it to CPU2's cache before CPU2 starts a reading or writing process.

Intel uses MESI (**M**odified, **E**xclusive, **S**hared, **I**nvalid) cache coherency protocol [22], and AMD has the MOESI (MESI plus Owned) protocol [23]. From the previous example with the Intel protocol, when CPU1 updates the variable Z, it marks **Exclusive** to the cache line which Z resides, and allows load and store operations on the cache line. If CPU2 needs to read Z, it will mark the cache line as **Shared**. After CPU1 writes 60 as a new value into the cache line, the cache line status will become **Modified**, and force CPU2 to **Invalidate** its cache lines. Therefore, CPU1 needs to backup Z with value 60 to the main memory before CPU2 can reload 60 to its cache line, and finally read Z.

1.5 False cache line sharing

This section reviews more details on the causes and effects of false cache line sharing, or *false sharing* in short. *False sharing* 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 [25]. In other words, *false sharing* is a side effect in a multiprocessor system due to cache coherency.

Generally, a multiprocessor system is composed of hundreds of racks and processors in a huge computer room which supplies high performance computing power for special research or critical systems such as an airline reservation center, a financial enterprise, or NASA. Although the multiprocessor's system scale seems quite different to a personal computer, its 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 be susceptible to a *false sharing* problem as well.

One multiprocessor system must maintain data coherency across CPUs to enforce data validation. To take advantage of cache, the write back policy must be engaged. When a processor makes a change on its cache, other processors must be aware of the change, and determine whether its copies of data in cache needs to be reloaded or not. Therefore, the cache coherency protocol plays an important role at this point. It 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. Intel's processors adopt the MESI protocol. When a 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.

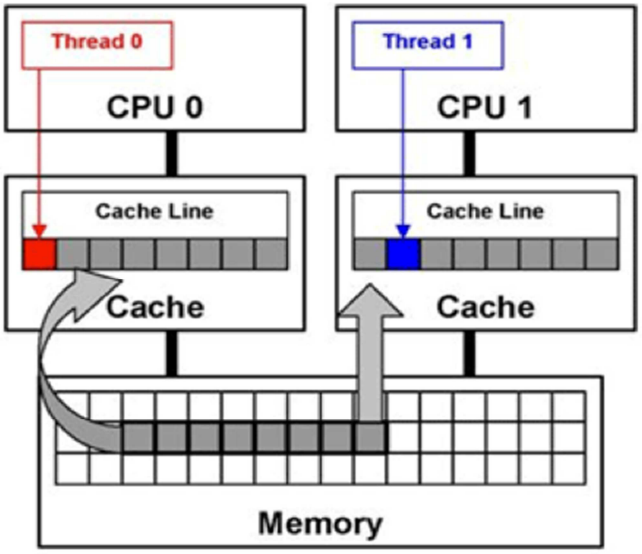


Figure 4. False cache line sharing on SMP [7]

In figure 4, when CPU1 writes a new value, it makes CPU0's cache invalidated, and causes a write back to the main memory. Consequently, if CPU0's updates its variable with another value, it results in CPU1's cache invalidation by backing up CPU1's cache line to the main memory. If both CPUs repeatedly write new values to their variables, invalidation will keep occurring between their caches and the main

memory. 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. Because of this, sometimes the false cache line sharing problem is called as "Cache Line Ping-Pong [19]."

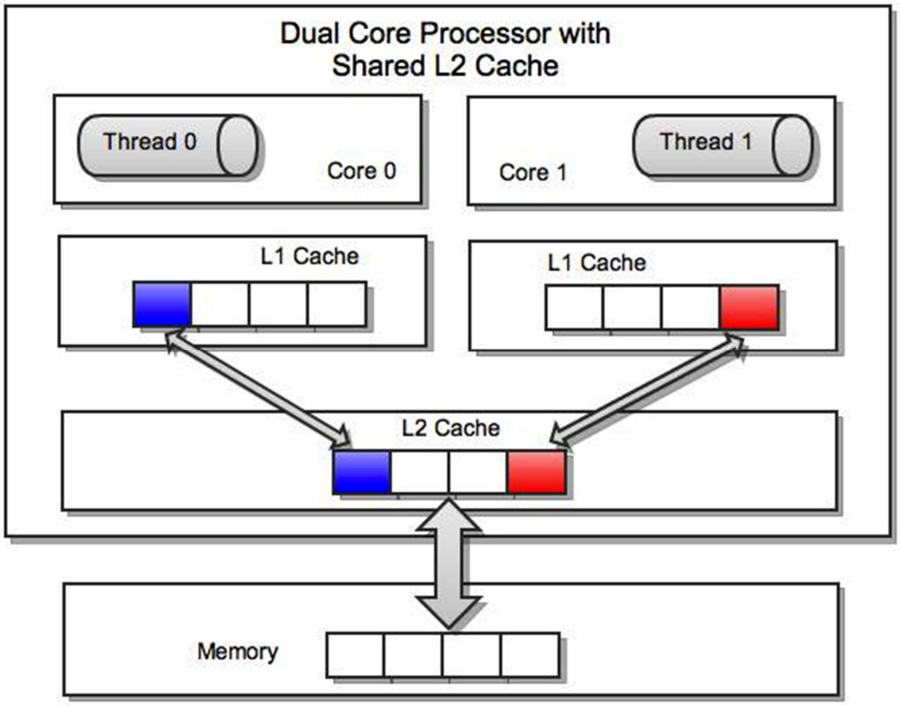


Figure 5. False cache line sharing on CMP [7]

2.0 Prior Work

This section reviews the prior research regarding *false sharing* effects on multiprocessor and multicore systems. To understand how the problem happens on the low level hardware, the detailed specifications of the test system must be described.

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 project is that *false sharing* would happen in a multicore architecture 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 result section.

2.1 Concurrent Hazards: False sharing

Butler did an experiment on a multiprocessor system to measure *false sharing* effects in [8]. His application was executed on a system with four packages of dual core CPUs, eight cores in total. The code drew a graph of speed-up in the cases of with and without *false sharing*.

The results of *false sharing* are shown in figure 6. The graph is plotted by speed-up ratios and the number of thread counts. The best speed-up at the eight-threaded execution shows a 100 times difference compared to the worst case. The gap could be bigger if the tests are run on 16-core, 32-core, or 64-core systems. Moreover, it can be observed from the graph that applying either a Spacing-only or Padding-only method does not significantly improve overall performance. Spacing and Padding will be described in section 3.2.

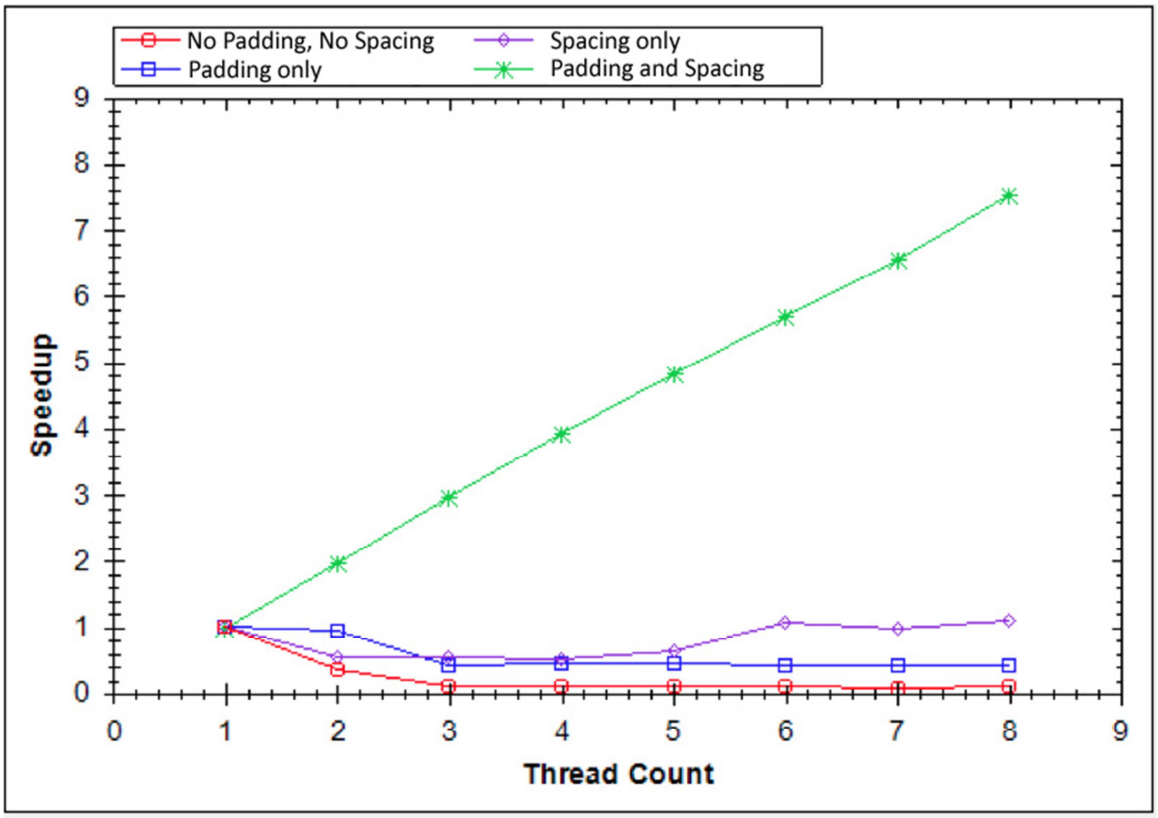


Figure 6. Speed-up results from Padding and Spacing employed on testing array [8]

2.2 Latency of conflict writes on Multicore Architecture

Dr. Josef discussed the latency penalty caused by *false sharing* [9]. The research evaluates write performance on both Intel and AMD processors.

The experiment was performed on the Intel Core Duo T2600 with a 32 Kbyte L1 cache per core, and a 2 Mbyte L2 shared cache. The result is plotted by values of the array size and latency cycles in figure 7. A higher number of cycles per iteration indicates lower performance.

Figure 7 shows that the amount of latency declines when the array is allocated 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 conclusion, the experiment proved that shared cache between cores can wipe out the adverse impact stemming from *false sharing*.

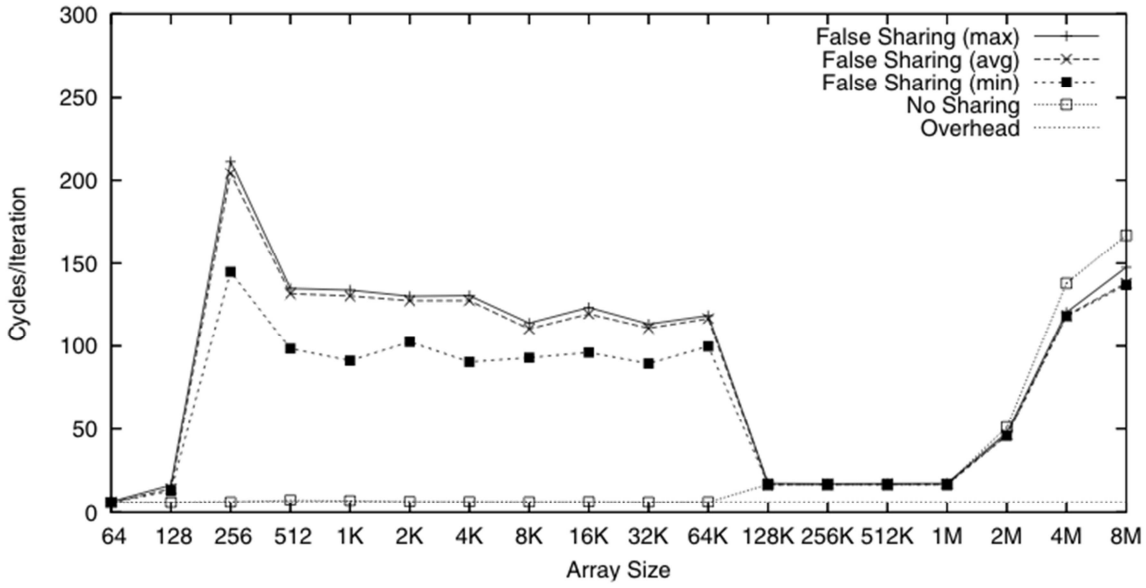


Figure 7. Number of latency cycles on varied array size, Intel Core Duo 2600 [16]

There are many approaches to abate *false sharing* effects. Tor and Susan introduced an approach to reduce *false sharing* on shared memory processors [10]. They developed compiler algorithms to analyze parallel programs by examining data structures susceptible to *false sharing.* They also employed the proper transformations to reduce *false sharing* effects. The results show a 2-58% improvement in the transformed versions. However, the work was performed on a simulator, and the actual code transformation is not revealed. For this reason, no further research could be performed.

3.0 Experiment Design

In this section, the experiment design in terms of hardware and software is discussed. The test application performs five experimental cases: Sequential, Parallel FS, Parallel FS + Spacing remedy, Parallel FS + Padding remedy, and Parallel FS + Padding and Spacing remedies. This section also describes techniques used to detect and avoid *false sharing* in this project.

3.1 False sharing detection

There are no tools to detect the occurrence of *false sharing* on a system in general. In other words, it is easy for the problem to be undetected since there are no indicators that any performance problems stem from *false sharing*. Whenever performance degrades, *false sharing* is just one suspect, and it is one that many programmers are not trained to look for.

Fortunately, there are certain profiling metrics that can indicate the existence of a *false sharing* issue [12]. It enables a way to narrow down the code by identifying the effects of *false sharing*, and helps the programmer become aware of the memory access pattern in a parallel program.

If the part of the program that is identified as a bottleneck is relatively CPU and memory bound, and that code rarely has I/O or blocking OS calls, then if both of the two following symptoms hold true then the existence of the *false sharing* problem is confirmed.

1. The code does not scale well when the concurrency level is increased by executing on more powerful hardware.

2. The code sometimes runs significantly slower for different input data that requires the same amount of processing and memory accesses, but a different pattern of data traversal. [12]

CPU performance counters are a set of the important indicators. The statistics from the low-level hardware can be used to determine the availability of CPU resources, including all other subsystems working with CPUs such as caches, branch prediction units, and so on. A profiler is able to retrieve these statistics so that they are analyzed in order to identify the type of the bottleneck thereby resolving the performance problems correctly. Two important parameters are used to show the occurrence of *false sharing*: L2 cache misses, and Cycles per Instruction (CPI).

L2 cache misses are a significant indicator to detect *false sharing* in a multiprocessor system. Many L2 cache misses would result in a large amount of data fetching from main memory into L2 cache. A root cause of L2 cache misses could be that (1) a processor requests data that does not reside in L2 cache, or (2) the corresponding cache lines are marked as invalid by data update operations from another processor. The latter mostly results from *false sharing*. Thus, a noticeable spike of L2 cache misses could indicate the existence of the *false sharing* problem.

CPI is a widely used indicator to diagnostic the overall performance. It demonstrates how many clock cycles are spent for each instruction. Therefore, CPI provides statistics on how efficient a program performs. CPI plays an important role to enumerate the amount of memory latency. Because the speed of a CPU is much faster than that of memory, the CPU needs to wait when fetching data from or writing data to memory. Thus an instruction takes more time to process.

The cache invalidation across processors causes L2 cache misses, and makes the CPU wait for data writing/reloading. Therefore, a great number of L2 cache misses combined with a high CPI indicates that *false sharing* is happening on a system.

3.2 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 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.

3.2.1 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 8a, integers D1, D2, D3 and D4 reside in the same cache line. If there are four assigned threads, one per core, updating those arrays, the cache coherence protocol will repeatedly cause data invalidation and force data to be written to, and reloaded from, the main memory. This cache Ping-Ponging greatly increases run time. With the implementation of the Spacing technique, *false sharing* on array data can be avoided as shown in figure 8b.

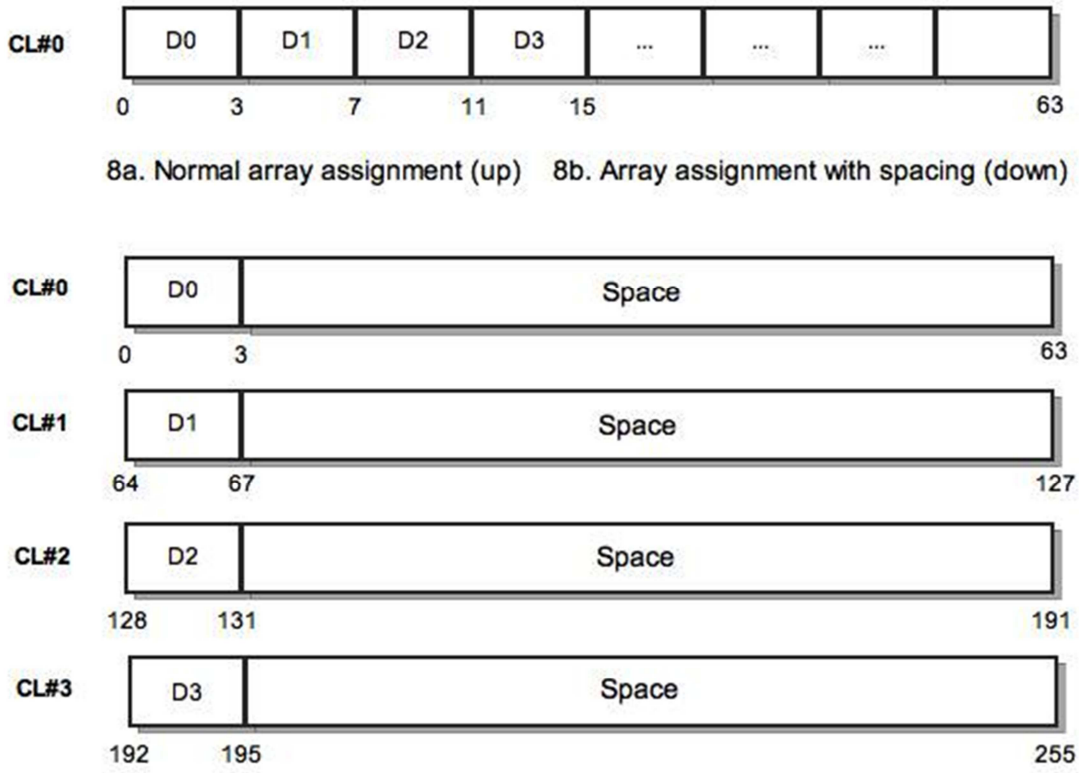


Figure 8. Cache line structure with Spacing technique

3.2.2 Padding technique

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

A variable declaration requires an additional piece of information to manage memory space for the variable. When a set of an array is declared, the operating system needs to define a piece of extra information that contains the array information which is called metadata. This metadata uses space just right before actual data, and consists of pointers and header information. For example, every array in .NET would require metadata consisting of SZARRAY, which stores size information of the array.

The existence and location of the metadata information in memory needs to be factored into account when using the padding remedy. This metadata is read before every

access to the array element. As a result, whenever a thread writes to an element, there is a read of the metadata, SZARRAY, happening just before the actual read on the data in the array. The Spacing technique does not separate the array metadata from the real array data, and the metadata still resides on the same cache line with the first array element as shown in figure 9a. Therefore, *false sharing* is still happening between the metadata and the first array element.

To eliminate sharing on metadata, the cache line where SZARRAY is located is padded so that the first array element is shifted to the next cache line. Figure 9b illustrates the cache line structure after the metadata SZARRAY is padded.

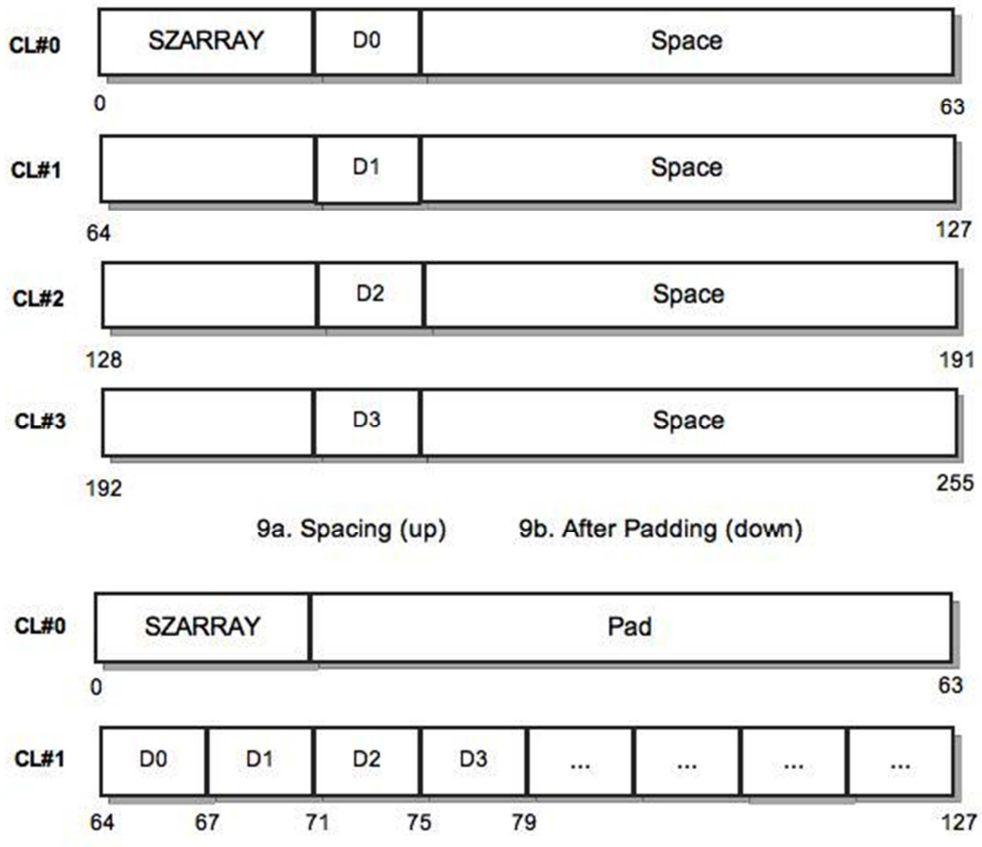


Figure 9. Cache line structure with Spacing only and Padding only techniques

3.2.3. Combined Spacing and Padding technique

According to Butler, using a Spacing-only or a Padding-only technique would not overcome the *false sharing* problem [8]. Therefore, the combination of both techniques is the best way to completely avoid *false sharing*. Figure 10b illustrates cache lines with Spacing and Padding applied. Each element is separated in a single cache line.

Obviously, Spacing and Padding each requires extra cache memory space. Programmers must estimate the memory sacrificed through the use of Spacing and Padding before building an actual application in order to maximize the performance (by mitigating false sharing) while minimizing the memory usage.

For example, an array is allocated 320 bytes in figure 10b instead of the originally reserved 24 bytes as in figure 10a to space consecutive elements onto separate cache lines.

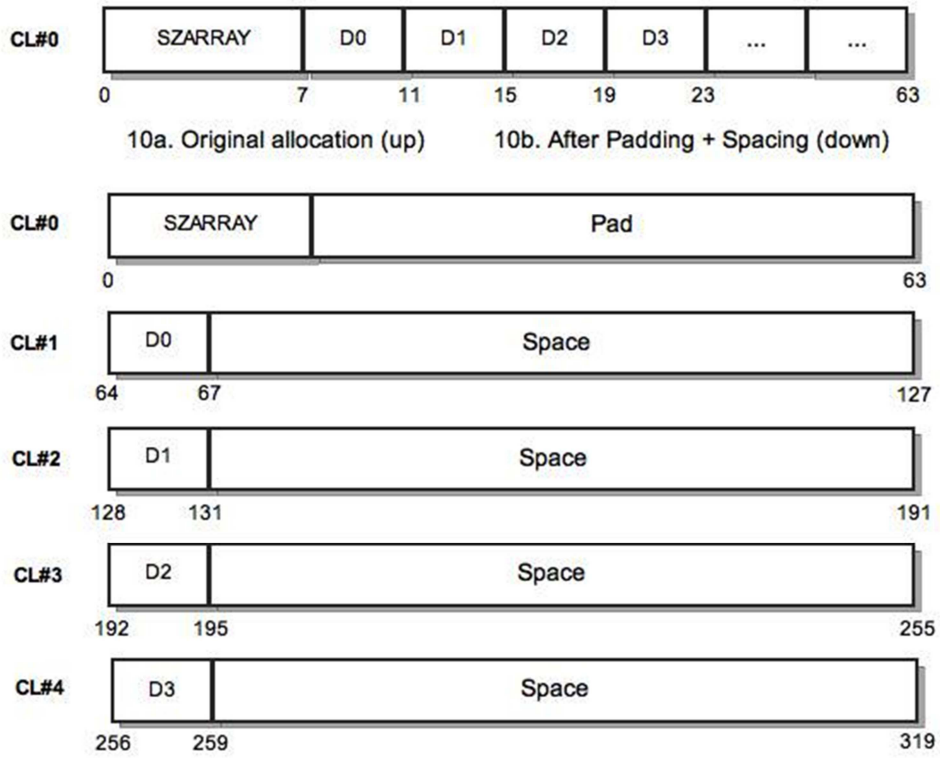


Figure 10. Cache line structure with combined Padding and Spacing technique

3.3 Testing code

The test programs are adapted from [8]. The testing code 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 test program begins with worker initialization. It reads the number of cores/processors from OS environment variables. The worker then forks one thread per core, and binds each thread to a processor. Next, the program divides the total workload into equal pieces, and assigns a piece to each thread. The workload in the test program is a simple operation that performs a memory access by writing a value to an array element.

Both false sharing remedies are applied. The size of the Padding and Spacing variables are defined to be 64 bytes, which is a size of one cache line, to ensure that every element is shifted onto a separate cache line.

There are five testing cases: Sequential, Parallel FS, Parallel FS + Spacing remedy, Parallel FS + Padding remedy, and Parallel FS + Spacing and Padding remedies. The data arrangement is the crucial focus in order to avoid *false sharing*. At first, the entire array is allocated, and each element is assigned to a thread. Each thread references to its own array offset, and repeatedly writes a value to its own element.

The following code fragments show how to declare the data array, set an offset, and execute the workload by writing a value to the array element.

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

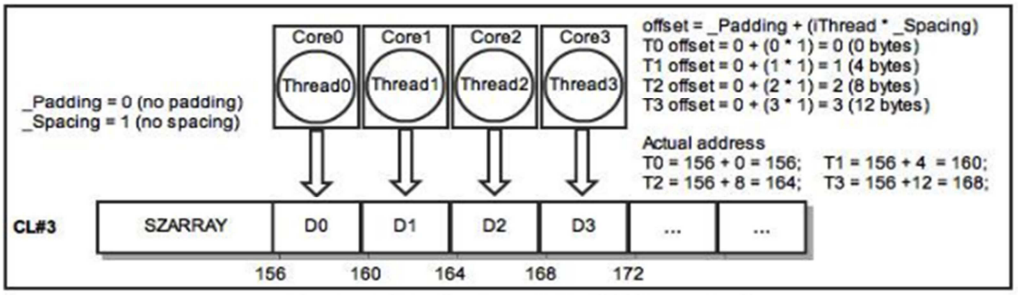
To avoid *false sharing*, the data layouts for all four FS parallel cases are differentiated. The Parallel FS case has all threads working on contiguous array elements. Offsets are used to define data layouts in all four parallel test cases. One offset space equals to a size of an integer or four bytes. The Parallel FS with Padding remedy case pads metadata by setting the offset variables ***_Padding*** to be 16 (64bytes), and ***_Spacing*** to be 1. The Parallel FS + Spacing remedy case splits off each array element by setting the offset variables ***_Spacing*** to be 16 (64bytes) and ***_Padding*** to be 0. The Parallel FS + Padding and Spacing remedies case sets both of the ***_Padding*** and ***_Spacing*** variables to be 16 (64bytes). The completed codes are listed in appendix A.

For example, suppose that a system consists of a four core processor, and there are only four integer elements; each core works on an array element. The array data is arbitrarily defined to start at the memory address 156. Generally an integer requires four bytes of memory space; therefore, all four integers can be allocated in one cache line. In the Parallel FS case, all four threads work on the contiguous array elements as shown in figure 11a. The case has false sharing happening on the cache line. The data layout of the Parallel FS + Spacing and Padding remedies case is designed to avoid *false sharing*. The layout separates those four integer elements and the metadata, and spreads them onto separate cache lines. Total 320 bytes of address space or five cache lines are required, as calculated below.

```
Data definition code fragment:
              int[_Padding + (_ThreadCount * _Spacing)]

Calculation:
              int[16 + (4 x 16)] ➔ int[80] ➔ 4 bytes X 80 offsets ➔ 320 bytes
```

The Parallel FS + Spacing and Padding remedies are performed with isolated cache lines as shown in figure 11b.



11a. Cache line structure of Parallel FS case (up)
11b. Cache line Structure of Parallel FS + Padding + Spacing remedies case (down)

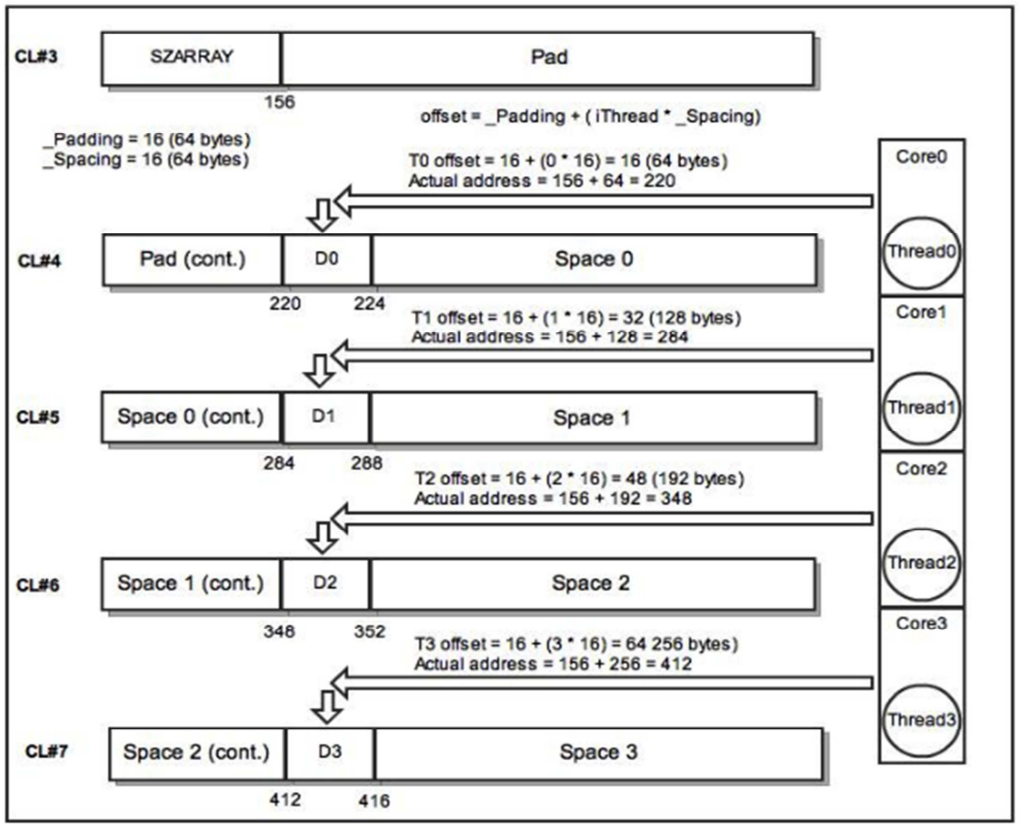


Figure 11. Cache line structures of the Parallel FS case and the Parallel FS + Spacing and Padding remedies case

4.0 Hardware, Software, and Development Kits Used

The experiments are executed on three different hardware systems. Hardware specifications, an operating system, software, and developing tools used in this project are enumerated in this section.

4.1 Hardware specifications

The experiments are performed on three specified types of multicore processors: Intel Core2 Duo, AMD Turion X2, and Intel Core i5.

4.1.1 Intel Core2 Duo test system

A Dell Vostro 1400 laptop represents a test system with an Intel Mobile Core2 Duo T5270 1.4GHz processor with a 32Kbyte L1 data cache and a 32Kbyte L1 instruction cache per core. The processor also has a shared 2Mbyte L2 cache on die. CPU-Z program displays the processor specifications and cache information in figure 12.

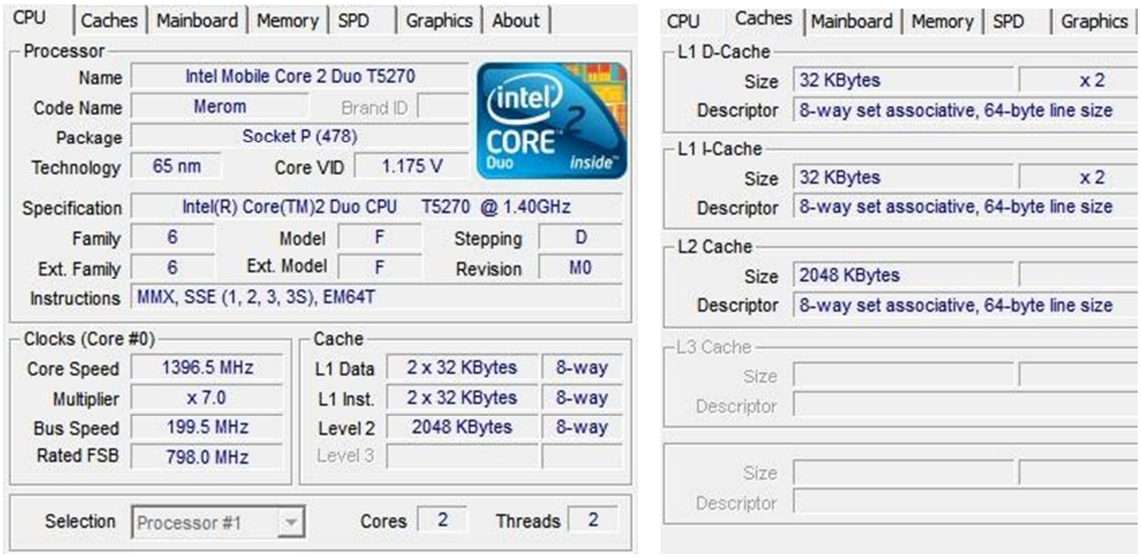


Figure 12. Intel Core2 Duo T5270 CPU and cache specifications [13]

4.1.2 AMD Turion X2 test system

Another test system is a HP DV6000 laptop embedded with an AMD Turion 64 X2 Mobile TL-58 1.9GHz CPU. The processor is composed of a 64Kbyte L1 data cache and a 64Kbyte instruction cache per core, and a 512Kbyte L2 cache per core. The AMD Turion 64 X2 processor specifications and cache information is exhibited by CPU-Z in figure 13.

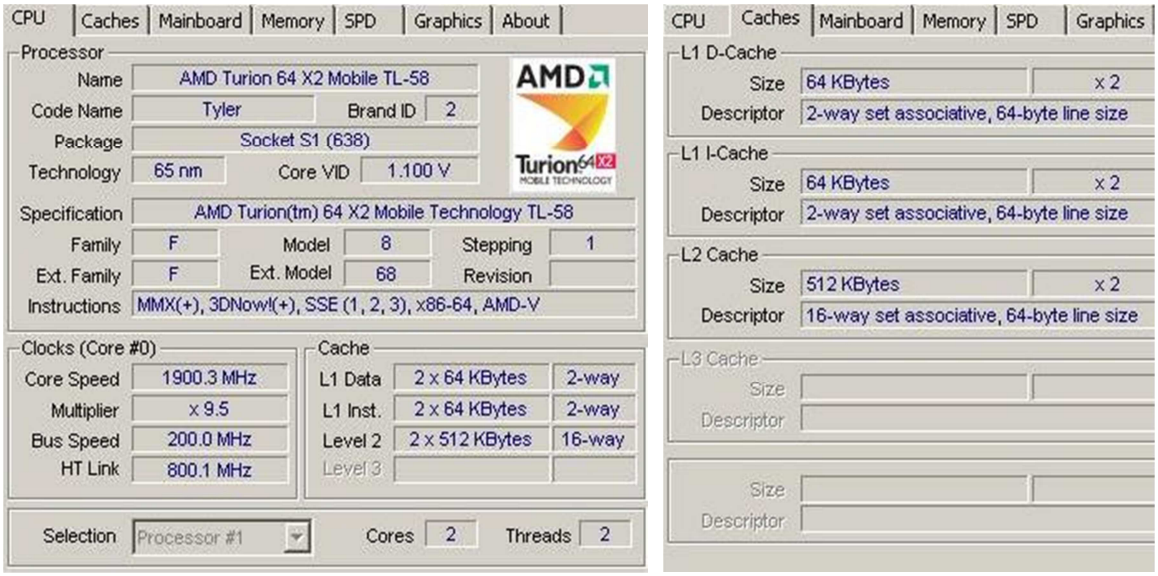


Figure 13. AMD Turion 64 X2 CPU and cache specifications [13]

4.1.3 Intel Core i5 test system

The last test system is a MacBook Pro laptop with an Intel Core i5 520M 2.4 GHz processor with Hyper-Threading (HT) technology. The CPU has three tier of caches: a 32Kbyte L1 data cache and a 32Kbyte L1 instruction cache per core, a 256Kbyte L2 cache per core, and a 3Mbyte L3 shared cache. Since the experiment must carry out on the Windows platform, Boot Camp, a utility on the Macintosh system, is used to install Windows XP SP3 prior to Visual Studio and other applications.

Figure 14 exhibits the Intel Core i5 520M processor specifications and cache information retrieved by CPU-Z on Windows XP SP3 with Boot Camp.

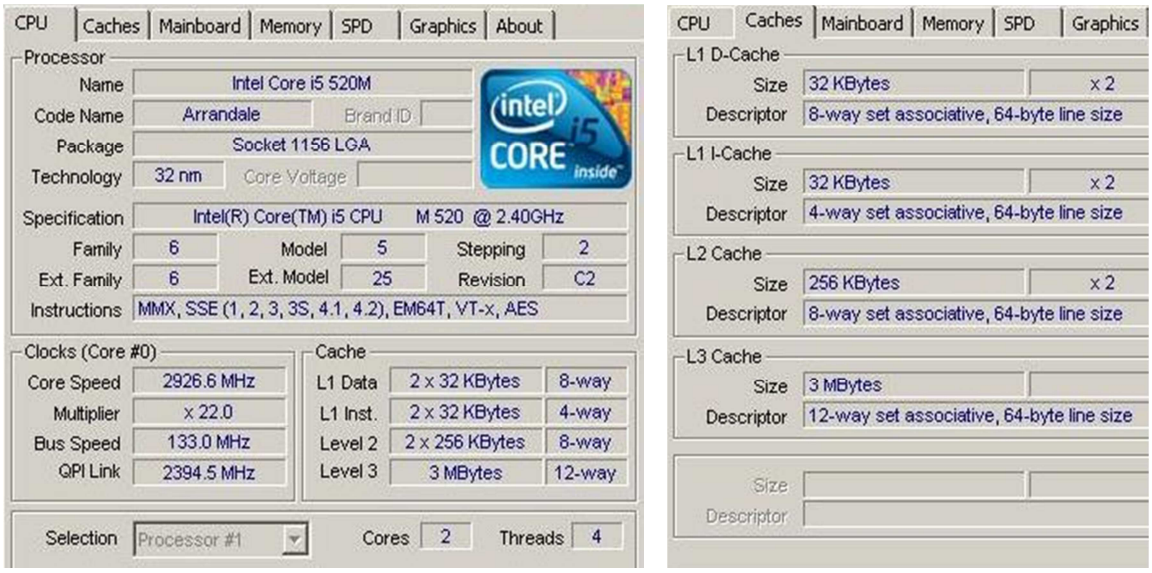


Figure 14. Intel Core i5 520M CPU and cache specifications [13]

4.2 Software

Software installed on the test systems is an operating system, utilities, and a software development tool. The system runs Windows XP service pack 3 as an operating system, and the program used in experiments are developed in C and C# languages on Visual Studio 2010. CPU-Z is a utility used to retrieve processor specifications and cache information.

5.0 Experiment Results

In this section, the results of the experiments are analyzed to understand how *false sharing* happens, and how to avoid it.

5.1 Gather cache line size

Before drawing a data layout diagram how *false sharing* occurs in a cache line, we need to gather cache specifications on the test system.

CL Reader is a program developed to read a cache line size of the Intel CPUs. It resorts to the Intel's manual which provides instruction sets for reading specific cache specifications from processor's registers. The utility shows a cache line size of the Intel test system equal to 64 bytes in figure 15. Nevertheless, the utility does not work on AMD processors because of compatibility between Intel and AMD instruction sets. Thus, the AMD processor's specifications are looked up by CPU-Z and manufacturers' specification manuals [14][15]. The program code of CL Reader is in appendix A.

Figure 15. Cache line size reported by CL Reader

5.2 Execution results

This experiment results are collected from the executions of five different test cases: Sequential, FS parallel, Parallel FS + Spacing remedy, Parallel FS + Padding remedy, and Parallel FS + Spacing and Padding remedies. These five cases are designed to execute the same amount of workload with different data layouts. The details of data arrangement in each case are:

- Sequential—a sequential execution of the assigned workload on one core.
- Parallel FS—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*.
- Parallel FS + Spacing remedy—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.
- Parallel FS + Padding remedy—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.
- Parallel FS + Spacing and Padding remedies—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.

The program execution is performed fifty iterations. The runtime is collected, and sorted in order. The five maximum and minimum figures are discarded to reduce data variation. The filtered data set of runtime is averaged to alleviate interferences from system environmental programs such as the anti-virus program, user applications, and system processes.

The performance comparison is measured by time to complete the workload. The workload is simply a write operation of a value to an array element, but repeatedly performed ten million times with a different value for each time. At the end of each execution, runtime results on the five different cases are printed, and speed-up ratios and efficiency are calculated from the runtime. Both numbers are computed as relative parallel performance based upon the sequential runtime as follows. The raw data table is listed in appendix B.

Speed-up(x) = sequential runtime / parallel runtime

Efficiency (%) = [(sequential runtime / parallel runtime) / number of cores]*100

Or

Efficiency (%) = (speed-up / number of cores) *100 **[11]**

5.2.1 Intel Core2 Duo T5270 results

The following table shows runtime, speed-up ratios, and efficiency percentage of the five test cases executed on the Intel Core2 Duo T5270 system.

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + Spacing & Padding
Runtime (millisecond)	115.75	227.11	152.90	117.08	66.08
Speed-up (X)	1	0.51	0.76	0.99	1.75
Efficiency (%)	100	25.48	37.85	49.43	87.58

Table 1. Intel Core2 Duo T5270 experiment results

The analysis compares the four parallel cases to the Sequential case, which is set as base performance. The Parallel FS case takes the greatest runtime (227.11ms) than any other cases. Usually, two processors working simultaneously on the same amount of

workload should take a half of time executed by one processor. However, the Parallel FS runtime is a doubled number of the Sequential one. The increased runtime is caused by *false sharing* which boosts the number of cache line invalidation and adds up the actual runtime with data reloading latency.

The Parallel FS + Spacing remedy shows a certain improvement when it is compared to the Parallel FS. Yet its runtime (152.90ms) is not satisfying since it is still greater than runtime in the Sequential case (115.75ms).

The Parallel FS + Padding remedy case spends less time (117.08ms) than the two prior cases. The number is even competitive to the Sequential case (115.75ms), but runtime with two cores would be a half of that on one core to gain equal efficiency. Therefore, performance degradation still shows up in this case because of the *false sharing* problem.

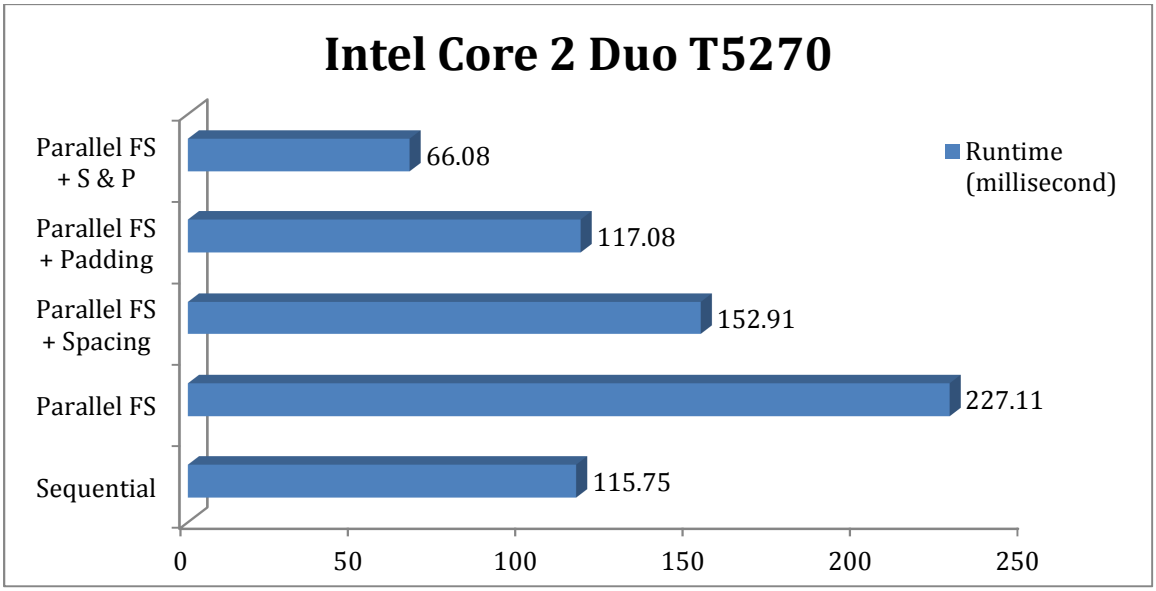


Figure 16. Average runtime on Intel Core2 Duo T5270 test system

According to a *false sharing* research, Butler proves that using solely either Spacing or Padding technique is unable to remove *false sharing* effects [8]. The theory is consistent to the experiment results.

Finally, Parallel FS + Padding and Spacing remedies case wins the best runtime (66.08ms). Since the data layout is deliberately defined to completely eliminate cache line sharing, it shows an outstanding performance compare with other cases. Figure 16 shows runtime of all test cases on the Intel Core 2 Duo T5270 system. The lower time indicates the better performance.

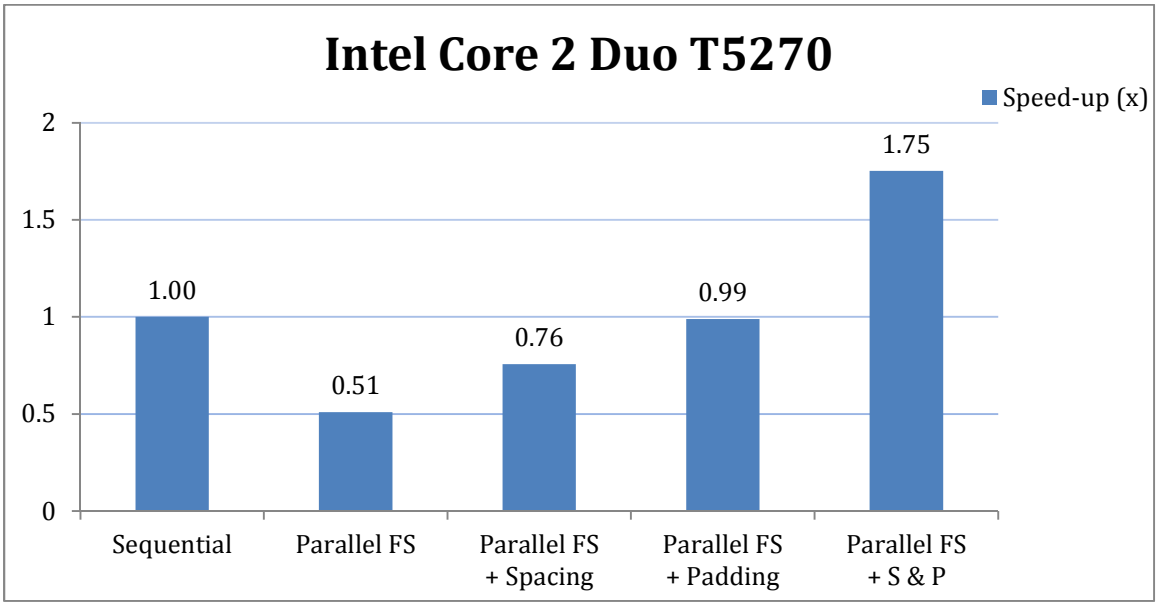


Figure 17. Speed-up ratios on Intel Core2 Duo T5270 test system

To further analyze the execution performance, the graph in figure 17 plots speed-up ratios of all cases calculated on the basis of Sequential case speed-up (1.0x).

The speed-up ratios demonstrate that *false sharing* has the most influences on the Parallel FS execution (0.51x), and less impacts on the two cases with remedial techniques, Parallel FS + Spacing remedy (0.76x) and Parallel FS + Padding remedy

(0.99x). The Parallel FS + Spacing and Padding remedies 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 because 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 [30].

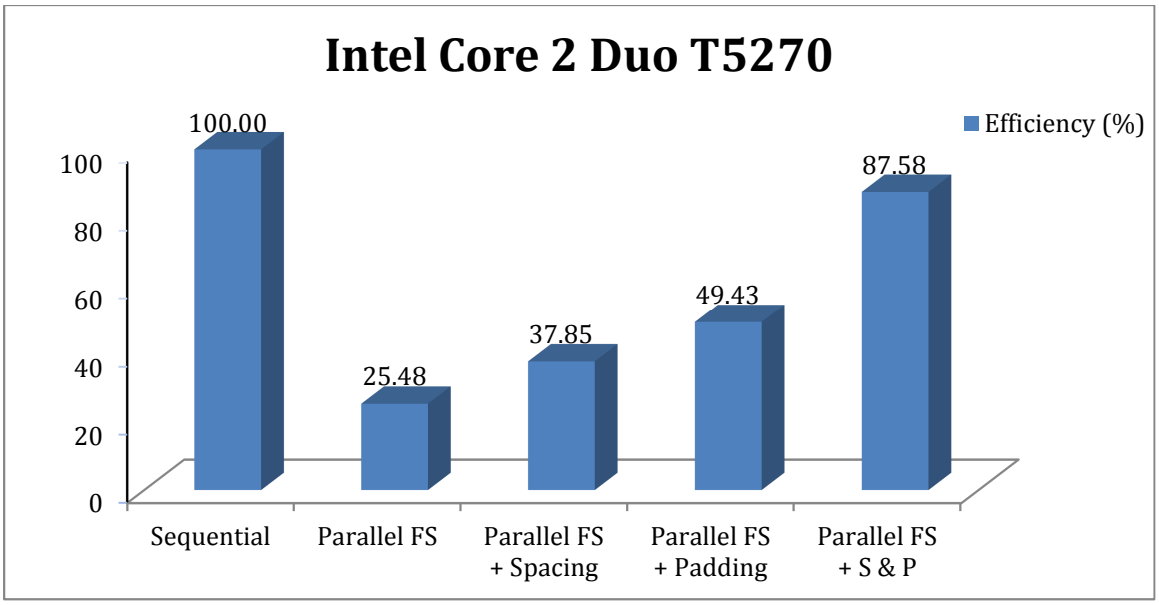


Figure 18. Efficiency percentage on Intel Core2 Duo T5270 test system

Efficiency is a fairly good indicator to measure performance per processing unit, or per core. 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 gain full efficiency. Figure 18 shows the efficiency that has a similar pattern to speed-up ratios: Parallel FS 25.48%, Parallel FS + Spacing remedy 37.85%, Parallel FS with Padding remedy 49.43%, and Parallel FS + Spacing and Padding remedies 87.58%. The amount

of lost efficiency results from the different degrees of *false sharing* impact. The more false cache line sharing occurs in a case, the lower performance it obtains.

5.2.2 AMD Turion 64 X2 Test Results

Table 2 shows the experiment results on the AMD Turion 64 X2. The average runtime, speed-up ratios, and efficiency percentage have similar characteristics to the Intel Core2 Duo T5270 experiment results.

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + Spacing & Padding
Runtime (millisecond)	147.00	292.64	202.59	234.80	74.73
Speed-up (X)	1	0.50	0.73	0.63	1.97
Efficiency (%)	100	25.12	36.28	31.30	98.34

Table 2. AMD Turion 64 X2 experiment results

The Parallel FS runtime (292.64ms) obtains the worst rank compared to all other cases. It takes approximated doubled runtime to the Sequential case.

The Parallel FS + Spacing remedy case (202.59ms) and the Parallel FS + Padding remedy (234.80ms) cases take less runtime than the Parallel FS, but not less than the sequential running. Unlike the Intel Core 2 Duo T5270 test, the Parallel FS + Spacing remedy outperforms the Parallel FS + Padding remedy.

The best runtime belongs to the Parallel FS + Spacing and Padding remedies. It is very close to ideal runtime of two processing cores which is the sequential runtime divided by two (73.5ms). Showing the differences among all cases, figure 19 displays a bar graph of the runtime. The lower runtime is the better performance.

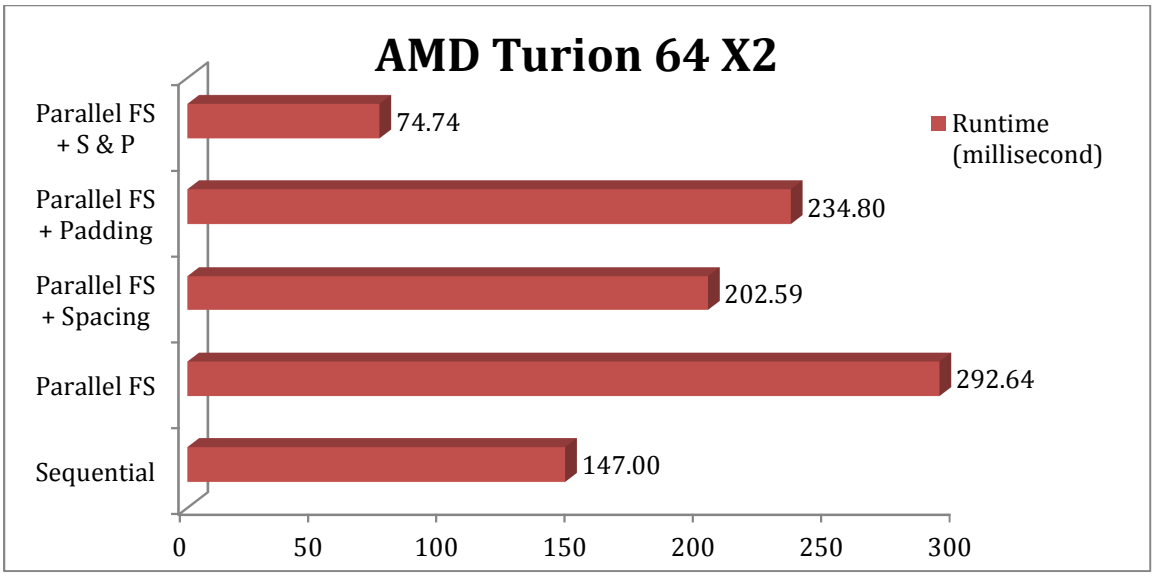


Figure 19. Average runtime on AMD Turion 64 X2 test system

Consider the speed-up ratios, the number of the Parallel FS case does not scale well (0.5x) compared to the sequential case (1.0x). When the Parallel FS case is employed with the Spacing technique to become the Parallel FS + Spacing remedy, the speed-up augments to be 0.73x. The Parallel FS + Padding remedy also reaches a greater speed-up (0.63x) compared to the Parallel FS case as shown in figure 20.

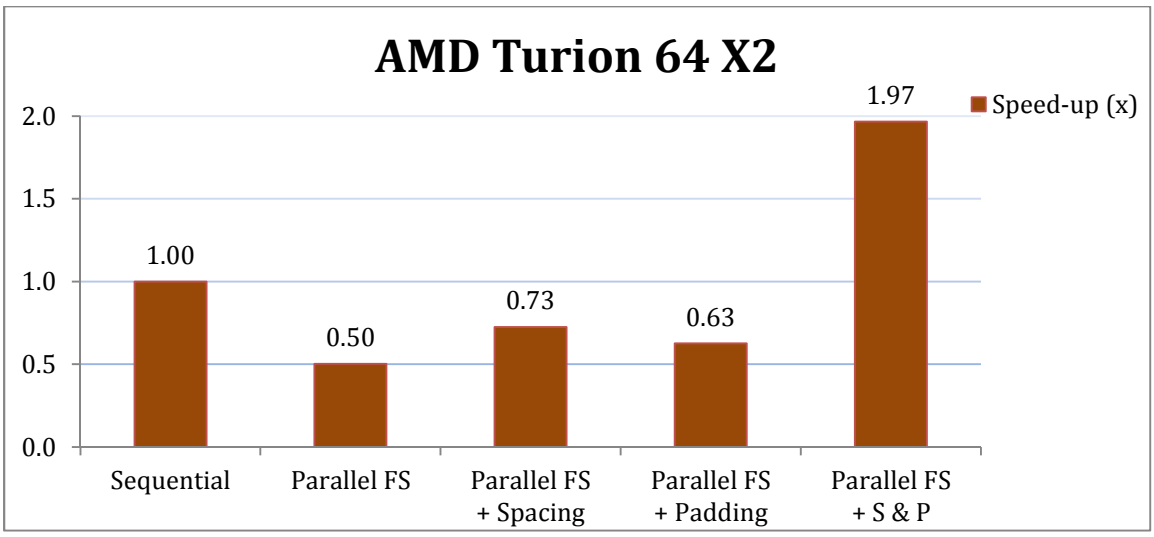


Figure 20. Speed-up ratio on AMD Turion 64 X2 test system

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

Among parallel cases, only the Parallel FS + Spacing and Padding (98.34%) can perform well in terms of efficiency as shown in figure 21. The efficiency in any other cases reflects the different performance degradation by different degrees of *false sharing* effects, Parallel FS (25.12%), Parallel FS + Padding remedy (31.30%), and Parallel FS + Spacing remedy (36.28%) respectively.

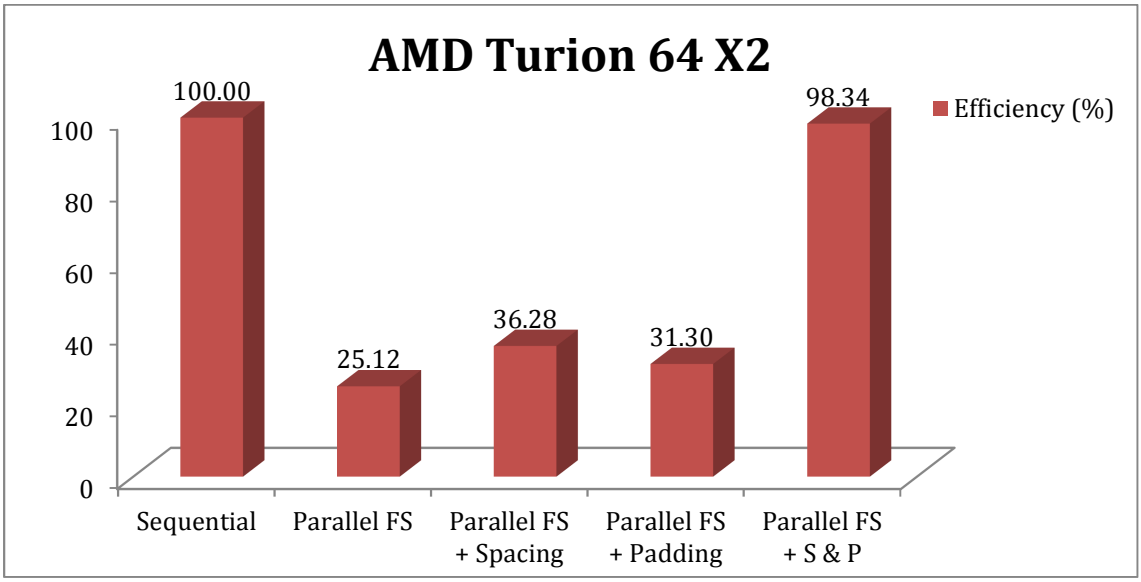


Figure 21. Efficiency percentage on AMD Turion 64 X2 test system

5.2.3 Intel Core i5 520M results

In table 3, the runtime of the four parallel cases are compared to the Sequential case as the same to the two former systems. The Parallel FS case spends 154.03ms to process the workload, which is as twice as much as the sequential runtime (87.74ms). In addition, the implementation either Spacing or Padding technique remedies the effect of

false sharing, and leads to better runtime compared to the Parallel FS case, 82.95ms on the Parallel FS + Spacing remedy and 89.16ms on the Parallel FS + Padding remedy case. The Parallel FS + Spacing and Padding remedies case reaches the best runtime (40.41ms), which is a half runtime of the Sequential case.

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + Spacing & Padding
Runtime (millisecond)	87.74	154.03	82.95	89.16	40.41
Speed-up (X)	1	0.57	1.06	0.98	2.17
Efficiency (%)	100	28.48	52.89	49.21	108.57

Table 3. Intel Core i5 520M experiment results

Figure 22 exhibits the runtime bar graph of the Intel Core i5 520M. The lesser time is the better performance.

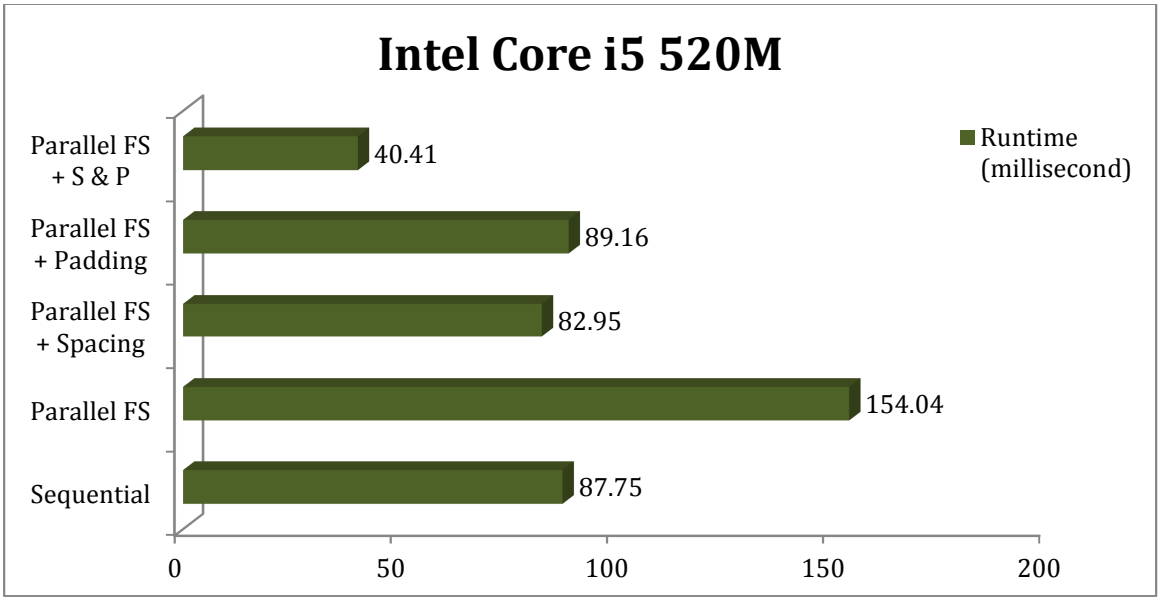


Figure 22. Average runtime on Intel Core i5 520M test system

Figure 23 shows speed-up ratios on the Intel Core i5 520M test system. The Parallel FS 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

Parallel FS + Padding remedy case (1.06x) and the Parallel FS + Spacing remedy case (0.98x). The Parallel FS + Padding and Spacing remedies case gains the highest speed-up ratio than two previous systems at 2.17x in speed.

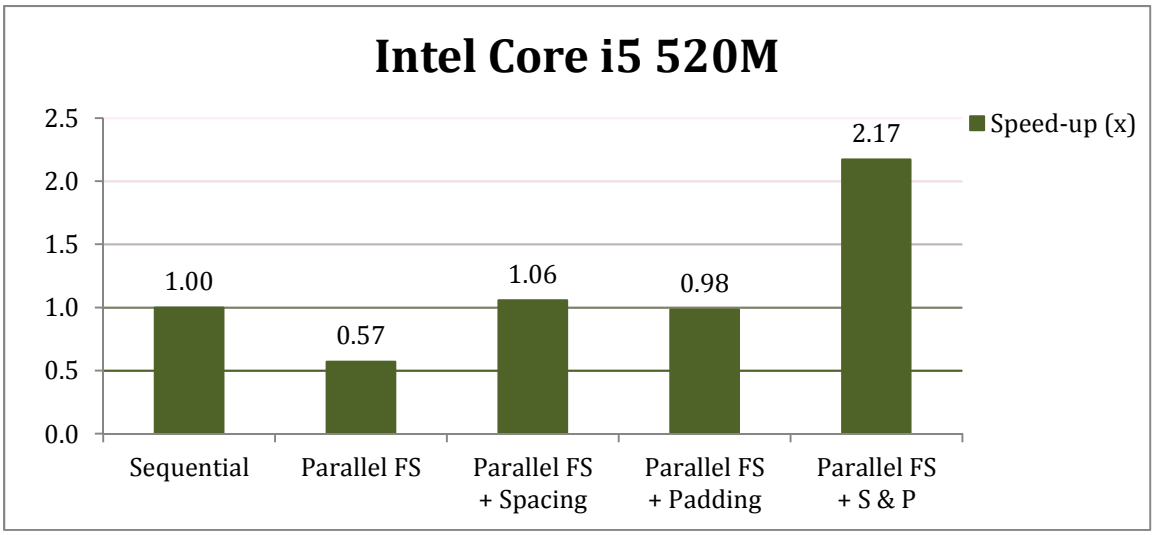


Figure 23. Speed-up ratios on Intel Core i5 520M test system

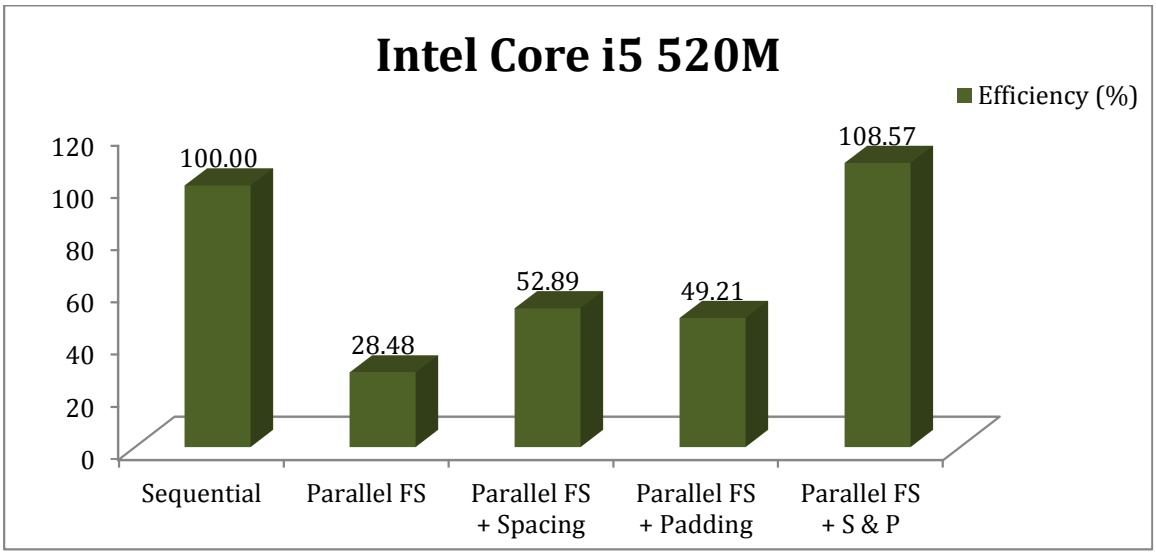


Figure 24. Efficiency percentage on Intel Core i5 520M test system

In figure 24, efficiency percentage of all five test cases is likely to be the same as the previous tests on Intel Core2 Duo T5270, and AMD Turion 64 X2. The efficiency of

the Parallel FS + Padding and Spacing remedies is noticeable with a “superlinear” number (108.57%). It is the case that efficiency exceeds 100%. The term Superlinear is explained in “Superlinear: an investigation into concurrent speed-up” [24]. The work exemplified a program that makes use of data stored in a shared cache. When the program is repeatedly executed, the performance will substantially boost up because of memory locality, both temporal and spatial.

In addition to benefits from locality of references, another condition to achieve a superlinear efficiency is capable of executing multiple concurrent threads. The Intel Core i5 520M processor comes up with Hyper-Threading technology which is able to execute two threads on a core at a time. Therefore, it increases probability for threads to take advantage of memory locality; thereby reaching to the point of the superlinear efficiency.

5.3 Performance drops caused by false sharing

This section illustrates performance drops caused by *false sharing*. From prior experiment results in section 5.2, the numbers of efficiency loss are observed as follows.

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + S & P
Intel Core2 Duo T5270					
Efficiency (%)	100	25.48	37.85	49.43	87.58
Loss (%)	-	74.52	62.15	50.57	12.48
AMD Turion 64 X2					
Efficiency (%)	100	25.12	36.28	31.30	98.34
Loss (%)	-	74.88	63.72	68.70	1.66
Intel Core i5 520M					
Efficiency (%)	100	28.48	52.89	49.21	108.57
Loss (%)	-	71.52	47.11	50.79	0 (+8.57)

Table 4. Efficiency loss caused by false sharing on the test systems

The Parallel FS 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 Parallel FS + Spacing remedy and the Parallel FS + Padding remedy cases also have significant performance degradation approximate 50-70% in loss, but less efficiency deficit compared to Parallel FS. Thus, the Parallel FS + Spacing and Padding remedies case performs efficiently, especially on the Core i5 520M processor. The case has a small number of losses on all three test systems: Intel Core2 Duo T5270 at 12.48% in loss, AMD Turion 64 X2 at 1.66% in loss, and Intel Core i5 520M at 8.57% in excess.

5.4 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) [8]. However, the experiment results in this project 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.

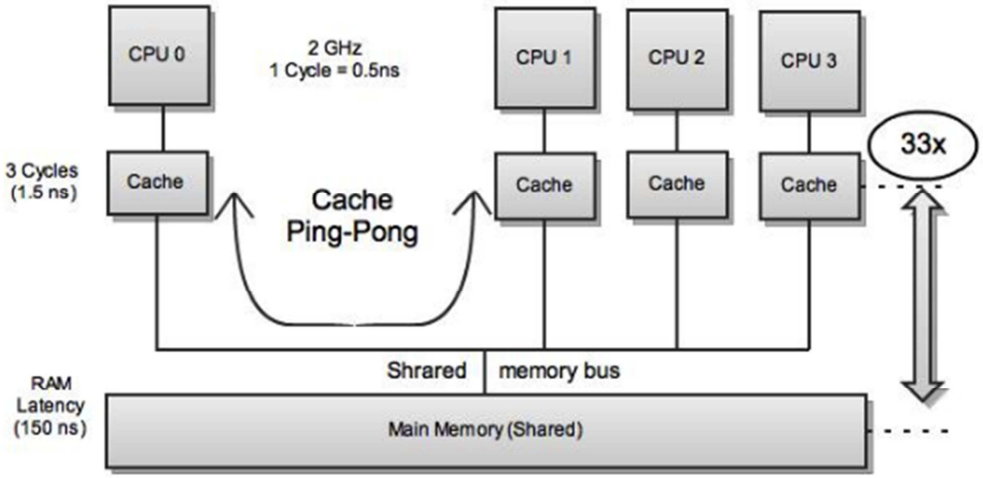


Figure 25 Cache Ping-ponging on multi-level memory in a multiprocessor system

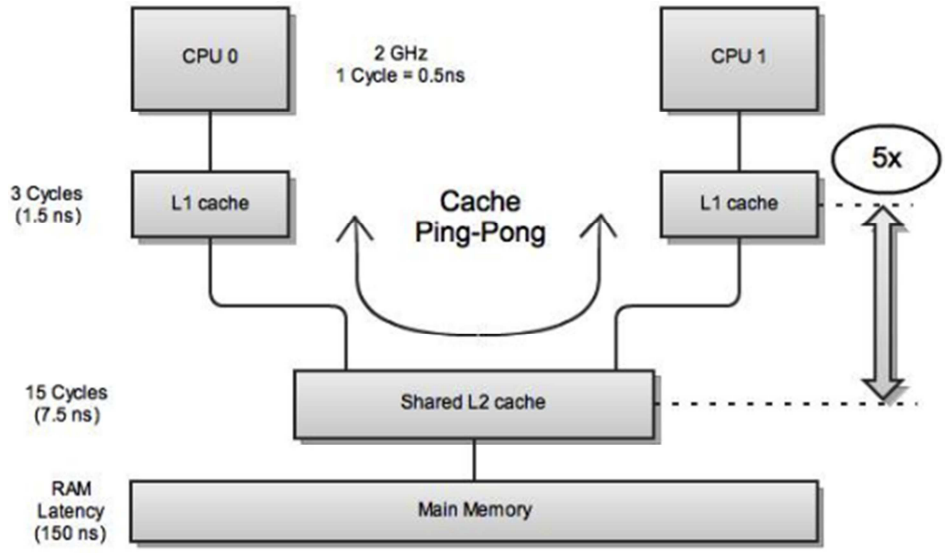


Figure 26. Cache Ping-ponging on multi-level memory in a dual core system

Figure 25 and 26 show block diagrams of a multiprocessor system and an Intel dual core processor system with multi-level memory hierarchies. Supposed that the program similar to the one that runs in the test experiment is executed in a multiprocessor system, *false sharing* will happen on the system. In the Parallel FS case, the array elements in a cache line are updated by many processors; *false sharing* happens leading to 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 25. 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:

Cache miss penalty (X bytes) = main memory access latency + X bytes/data receive rate [26]

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.

In case the similar scenario of *false sharing* occurs on a dual core system, the Cache Ping-Ponging also happens in the system as shown in figure 26. Yet, the cache invalidation in the dual core system takes place in between the L1 cache and the shared L2 cache, instead of in between the cache and the main memory in multiprocessor systems. The on-die caches are local memories having low latency since they reside internally in the CPU package. Data transfers among caches do not require bus transactions like data transfers between cache and main memory. Therefore, the severity degree of *false sharing* on a dual core system does not cause significant performance degradation like 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 case of Parallel FS running 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 FS resolved cases, the Parallel FS + Spacing remedy and Parallel FS + Padding remedy 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, the Parallel FS + Spacing and Padding remedies case completely avoids *false sharing*, and obtains performance at nearly 100% efficiency.

All the test systems, Intel Core2 Duo T5270, AMD Turion 64 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 project indicates that 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 come from the

differences in 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. This interconnection technology is called AMD 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*, because a program working four times slower in parallel on a dual core system means it runs even slower than sequential execution on a single core processor. The *false sharing* issue, therefore, is a major potential issue in parallel programming on multicore CPUs.

This project proposed and implemented the Spacing and Padding techniques to avoid *false sharing*. The Spacing technique separates many variables in a shared cache line into a variable for each cache line. The Padding technique isolates shared array metadata from the actual variables with a pad. The combination of both techniques is necessary to completely eliminate *false sharing* on the test scenario. Nevertheless, there is a trade-off for the implemented techniques. The implementation of Spacing and Padding techniques barters with memory space. On the dual core test systems, the amount of memory used in the Parallel FS case is 8 bytes for the array plus the metadata size, which can be rounded up to be 16 bytes. The modified array size in the Parallel FS + Spacing and Padding remedies case becomes three cache lines, or 192 bytes, which are one element in a cache line per core plus another 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 [27]. All break-though 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. However, this project shows the existence of *false sharing* on dual core CPUs. It is most likely 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] Sae-eung, S., 2009. A Sequential to Parallel Code-Converter Mentor. Computer Science Department, San Jose State University

[3] The Many Cores of Intel. Web site: http://www.forbes.com/2009/02/14/intel-gelsinger-microprocessors-technology-cio-network_0216_intel.html

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

[5] Yan, L., et al, 2009. Performance evaluation of the memory hierarchy design on CMP prototype using FPGA. 2009. In *ASICON '09. IEEE 8th International Conference on ASIC*, pp.813-816, 20-23. doi: 10.1109/ASICON.2009.5351570

[6] Kulick, J. Multiprocessor on Chip. Web site: http://cst.mi.fu-berlin.de/teaching/WS0708/19565-PS-TI/reports/kulick08multiprocessing_slides.pdf

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

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

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

[10] Jeremiassen, T. E., Eggers, S. J. 1995. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *SIGPLAN Not.* 30, vol. 8, pp. 179-188. DOI= http://doi.acm.org/10.1145/209937.209955

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

[12] Toub, S., Ostrovsky, I., Yildiz, H. .Net Matter: False Sharing. Web site: http://msdn.microsoft.com/en-us/magazine/cc872851.aspx

[13] CPU-Z 1.55. Web site: http://www.cpuid.com

[14] Intel Core2 Duo Mobile processor T5270 specifications Web site: http://ark.intel.com/Product.aspx?id=33096

[15] AMD Turion64 X2 specifications Web site: http://www.amd.com/us/products/Pages/products.aspx

[16] Ananth, G., et al. 2003. Introduction to Parallel Computing. Addison-Wesley, San Francisco

[17] Charles, N. Shared Data Considered Harmful. Web site: http://blog.headius.com/2008/04/shared-data-considered-harmful.html

[18] Hewlette-Packard. False Cache Line Sharing. Web site: http://docs.hp.com/en/B6056-96006/ch13s02.html

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

[20] Vivek Khera, P R LaRowe, Jr., and S C Ellis. 1993. An Architecture-Independent Analysis of False Sharing. Technical Report. Duke University, Durham, NC, USA.

[21] Smith, A. J. 1982. Cache Memories. ACM Computing Survey, 14, 3

[22] Wikipedia. MESI Protocol. Web site: http://en.wikipedia.org/wiki/MESI_protocol

[23] Wikipedia. MOESI Protocol. Web site: http://en.wikipedia.org/wiki/MOESI_protocol

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

[25] Hewlette-Packard. Parallel Programming Guide for HP-UX Systems. Web site: http://docs.hp.com/en/B6056-96006/ch13s02.html

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

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

[28] 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

[29] 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.

[30] 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

Appendix A: Source codes

CL_Reader.c

```
#include "stdafx.h"
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include "conio.h"

ULONG get_basic_info(void);

int _tmain(int argc, _TCHAR* argv[])
{
        printf("Cache line size is: %u bytes",get_basic_info());
        getch();

        return 0;
}

ULONG get_basic_info(void){
        _asm{
                MOV EAX,80000006h
                CPUID
                MOV EAX,ECX
                AND EAX,0xff
        }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace FS
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Form1.cs

```
#define PERF_FALSEX
#define PERF_TRUEX

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
using ZedGraph;

namespace FS
{
    public partial class Form1 : Form
    {
        SynchronizationContext _Sync = null;

        public Form1()
        {
            InitializeComponent();

            Shown += new EventHandler(HandlesShown);

            _Sync = SynchronizationContext.Current;
        }

        void HandlesShown(object sender, EventArgs e)
        {
#if PERF_FALSE
                        for ( int i = 0 ; i < 9 ; i++ ) Work( Environment.ProcessorCount, false
); Close();
#elif PERF_TRUE
                        for ( int i = 0 ; i < 9 ; i++ ) Work( Environment.ProcessorCount, false,
padding: true, spacing: true  ); Close();
#endif
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Begin();
        }

        void Begin()
                {
                        int REPEAT = 1;
                        bool oneWriter = false;
                        //pb.Value = 0;

            var nn = new PointPairList();
            var ny = new PointPairList();
            var yn = new PointPairList();
            var yy = new PointPairList();

                        var task = Task.Factory.StartNew( () =>
                                {
                                        Trace.WriteLine( "\n\nRun: " + DateTime.Now.TimeOfDay +
"\n" );


                                        int max = Environment.ProcessorCount * 4;
                                        int cur = 0;

                                        for ( int threads = 1 ; threads <=
Environment.ProcessorCount ; threads++ )
                                        {
                                                nn.Add( threads, Enumerable.Range( 0, REPEAT
).Median( i => Work( threads, oneWriter ) ) );

                                                ny.Add( threads, Enumerable.Range( 0, REPEAT
).Median( i => Work( threads, oneWriter, spacing: true ) ) );
```

```
                        yn.Add( threads, Enumerable.Range( 0, REPEAT ).Median( i => Work( threads,
oneWriter, padding: true ) ) );

                                                yy.Add( threads, Enumerable.Range( 0, REPEAT
).Median( i => Work( threads, oneWriter, padding: true, spacing: true ) ) );

                    }
                                } );

                }

                double Work( int threadCount, bool oneWriter, bool padding = false, bool spacing
= false, int affinity = -1 )
                {
                        int iPadding = padding ? 16 : 0;
                        int iSpacing = spacing ? 16 : 1;

                        var trace = String.Empty;
                        trace += "ThreadCount: " + threadCount;
                        trace += " - Padding: " + iPadding;
                        trace += " - Spacing: " + iSpacing;
                        if ( affinity != -1 ) trace += " - Affinity: " + affinity;
                        Trace.WriteLine( trace );

                        if ( affinity == -1 ) affinity = 1;

                        var sequential = Task.Factory.StartNew<TimeSpan>( new Worker( 1,
oneWriter, iPadding, iSpacing, affinity ).Work );

                        var parallel = sequential.ContinueWith<TimeSpan>( prev => new Worker(
threadCount, oneWriter, iPadding, iSpacing, affinity ).Work() );

                        double y = 0d;

                        var results = parallel.ContinueWith( prev =>
                                {
                                        var speedup = sequential.Result.TotalSeconds /
parallel.Result.TotalSeconds;
                                        var slowdown = 1d / speedup;
                                        var efficiency = 100d * speedup / threadCount;

                                        Trace.WriteLine(
                                                "Speedup: " + speedup.ToString( "N2" ) +
                                                " ; Slowdown: " + slowdown.ToString( "N2" ) +
                                                " ; Efficiency: " + efficiency.ToString( "N2" ) +
"%\n" );

                                        y = speedup;
                                } );

                        results.Wait();

                        return y;
                }

    }
}
```

Worker.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Threading;
```

```
using System.Runtime.InteropServices;

namespace FS
{
        public class Worker
        {
                const int ITERS = ( int ) 1e7;

                int _ThreadCount = 0;
                bool _OneWriter = false;
                int _Padding = 0;
                int _Spacing = 0;
                int _Affinity = 0;

                public Worker( int threadCount, bool oneWriter, int padding, int spacing, int
affinity )
                {
                        _ThreadCount = threadCount;
                        _OneWriter = oneWriter;
                        _Padding = padding;
                        _Spacing = spacing;
                        _Affinity = affinity;
                }

                [DllImport( "kernel32.dll" )]
                static extern IntPtr GetCurrentThread();

                [DllImport( "kernel32.dll" )]
                static extern UIntPtr SetThreadAffinityMask( IntPtr hThread, UIntPtr
dwThreadAffinityMask );

                public TimeSpan Work()
                {
                        var data = new int[ _Padding + ( _ThreadCount * _Spacing ) ];
                        Array.Clear( data, 0, data.Length );

                        var iters = ITERS / _ThreadCount;

                        using ( var mre = new ManualResetEvent( false ) )
                        using ( var countdown = new CountdownEvent( _ThreadCount ) )
                        {
                                TimeSpan[] tss = new TimeSpan[ Environment.ProcessorCount ];

                                for ( int i = 0 ; i < _ThreadCount ; i++ )
                                {
                                        int iThread = i;

                                        if ( !_OneWriter || iThread == 0 )
                                        {
                                                new Thread( () =>
                                                {
                                                        SetThreadAffinityMask(
GetCurrentThread(), new UIntPtr( 1u << ( iThread * _Affinity ) ) );

                                                        var offset = _Padding + ( iThread *
_Spacing );

                                                        mre.WaitOne();

                                                        for ( int x = 0 ; x < iters ; x++ )
data[ offset ]++;
                                                        countdown.Signal();

                                                } ) { IsBackground = true, Priority =
ThreadPriority.Highest }.Start();
                                        }
                                        else
                                        {
```

```
                                                new Thread( () =>
                                                {
                                                        SetThreadAffinityMask(
GetCurrentThread(), new UIntPtr( 1u << ( iThread * _Affinity ) ) );

                                                        var offset = _Padding + ( iThread *
_Spacing );
                                                        int dummy = 0;
                                                        mre.WaitOne();
                                                        for ( int x = 0 ; x < iters ; x++ )
dummy = data[ offset ];
                                                        countdown.Signal();

                                                } ) { IsBackground = true, Priority =
ThreadPriority.Highest }.Start();
                                        }
                                }
                                Thread.Sleep( 100 );

                                var sw = Stopwatch.StartNew();
                                mre.Set();
                                countdown.Wait();
                                var ts = sw.Elapsed;

                                Trace.WriteLine( "False : " + data.Sum( i => ( long ) i
).ToString( "N0" ) + " in " + ts.TotalSeconds + " secs" );

                                return ts;
                        }
                }
        }
}
```

EnumerableEx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FS
{
        static class EnumerableEx
        {
                public static U Median<T, U>( this IEnumerable<T> e, Func<T, U> fn )
                {
                        var list = e.Select( fn ).OrderBy( u => u ).ToList();

                        if ( list.Count == 0 ) return default( U );

                        return list[ list.Count / 2 ];
                }
        }
}
```

Appendix B: Result tables

Intel Core 2 Duo T5270

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + S & P
Run 1	0.1163863	0.1166894	0.1648988	0.1152791	0.0684922
Run 2	0.1192282	0.202907	0.1875805	0.1140728	0.0845467
Run 3	0.1153833	0.2469059	0.1250019	0.1178538	0.0847937
Run 4	0.1109213	0.2465726	0.1693793	0.1201258	0.0643503
Run 5	0.111287	0.1944945	0.1638333	0.122339	0.0882128
Run 6	0.1121318	0.2439458	0.1679624	0.1171785	0.0643919
Run 7	0.1178423	0.2424299	0.1507663	0.0684436	0.0649143
Run 8	0.1186988	0.2437536	0.1575376	0.1039701	0.0653907
Run 9	0.119711	0.118541	0.1553775	0.1059232	0.0642609
Run 10	0.1106445	0.1962922	0.0710894	0.1156297	0.0652434
Run 11	0.1152157	0.1799734	0.1464814	0.1198414	0.0655823
Run 12	0.1117273	0.1899749	0.1469859	0.1177579	0.0646953
Run 13	0.1313647	0.1584975	0.1706342	0.1242836	0.0731129
Run 14	0.1171766	0.2654264	0.1655056	0.1190972	0.0649755
Run 15	0.115473	0.2713819	0.1243029	0.1219775	0.064274
Run 16	0.1187363	0.2638603	0.1605715	0.119043	0.0738674
Run 17	0.1148833	0.2527399	0.1595881	0.1204496	0.0658807
Run 18	0.1134353	0.2492632	0.1705026	0.0703986	0.0655452
Run 19	0.1138292	0.1589822	0.0648518	0.1204834	0.0645908
Run 20	0.1196124	0.259736	0.1240978	0.1182301	0.0653658
Run 21	0.1153462	0.2503658	0.1618054	0.1137119	0.0676974
Run 22	0.1114705	0.2437496	0.1213894	0.1196012	0.0644316
Run 23	0.1380485	0.1909373	0.1720525	0.1048675	0.0660567
Run 24	0.1154143	0.1983179	0.1636168	0.1200063	0.0670546
Run 25	0.1103374	0.1976837	0.1631693	0.1168229	0.0646023
Run 26	0.1104897	0.2504888	0.1715332	0.1173017	0.0641992
Run 27	0.1134498	0.2633097	0.1900356	0.1294441	0.064138
Run 28	0.1207801	0.1994795	0.1654271	0.1068839	0.0646693
Run 29	0.1137963	0.257914	0.1647617	0.1152283	0.0653063
Run 30	0.1171732	0.2460578	0.1575895	0.1188349	0.0662235
Run 31	0.1174568	0.1962844	0.0643442	0.1248831	0.0679058
Run 32	0.1189816	0.2355944	0.1469748	0.118075	0.0641522
Run 33	0.1324319	0.1188821	0.0714238	0.1087384	0.068046
Run 34	0.1149972	0.2429143	0.1649047	0.1228781	0.0644945
Run 35	0.1170564	0.1165321	0.1256352	0.1177772	0.0640031
Run 36	0.1152266	0.2425576	0.1741059	0.1207946	0.0658664
Run 37	0.1296438	0.2461751	0.1815102	0.1281872	0.0676767
Run 38	0.1124229	0.2462891	0.1533088	0.1192615	0.0651163
Run 39	0.1163762	0.2548483	0.1460316	0.1068091	0.0652152
Run 40	0.1135214	0.1932932	0.1470739	0.1147958	0.065068
Run 41	0.1255637	0.2422662	0.0845557	0.1178412	0.0696267
Run 42	0.1166874	0.2531805	0.2122903	0.1154359	0.0746368

Run 43	0.1150442	0.2451118	0.1747428	0.1186946	0.0644227
Run 44	0.1126391	0.2498711	0.0671247	0.1150813	0.0704315
Run 45	0.1116261	0.2426386	0.1689354	0.1132931	0.0704597
Run 46	0.1239244	0.1925761	0.1264965	0.1216216	0.0685559
Run 47	0.112564	0.2572349	0.159482	0.1186628	0.063781
Run 48	0.1147472	0.117816	0.1670477	0.1189436	0.0643556
Run 49	0.1118812	0.2465656	0.1627371	0.1136152	0.064552
Run 50	0.1180127	0.2553268	0.124168	0.1160155	0.0644308

AMD Turion 64 X2

	Sequential	**Parallel FS**	**Parallel FS + Spacing**	**Parallel FS + Padding**	**Parallel FS + S & P**
Run 1	0.1463781	0.2441405	0.2169434	0.2303356	0.0761409
Run 2	0.1536357	0.2388756	0.21555	0.2360635	0.0737962
Run 3	0.1457229	0.3525182	0.2135168	0.2330824	0.0738638
Run 4	0.1855121	0.2369482	0.1915695	0.2321286	0.0722237
Run 5	0.1602119	0.3270278	0.2374659	0.231849	0.0744494
Run 6	0.1464915	0.3447681	0.216735	0.2342638	0.0728081
Run 7	0.1422543	0.2936454	0.1855607	0.2444081	0.0728701
Run 8	0.1512125	0.3468837	0.21496	0.2370787	0.0769103
Run 9	0.1529554	0.3478101	0.2142459	0.235367	0.0741298
Run 10	0.2037918	0.3450033	0.1812182	0.0876114	0.0746075
Run 11	0.1574694	0.2372723	0.2169007	0.240794	0.0774048
Run 12	0.1579535	0.3424778	0.1928736	0.2358428	0.0770159
Run 13	0.1456025	0.2440986	0.1937749	0.2351667	0.0832072
Run 14	0.1431011	0.3412816	0.194461	0.2529905	0.0742591
Run 15	0.1470983	0.3573348	0.1911857	0.2290014	0.0784493
Run 16	0.1436333	0.310785	0.0756286	0.2395279	0.0738317
Run 17	0.1442406	0.3500905	0.1881378	0.2456362	0.071518
Run 18	0.1602577	0.3486124	0.2082024	0.2366948	0.0723499
Run 19	0.1428782	0.3114418	0.1920431	0.2345457	0.0738619
Run 20	0.1425094	0.2709243	0.2166166	0.2382411	0.0725494
Run 21	0.1435327	0.309118	0.2207962	0.2377173	0.073126
Run 22	0.1455545	0.3557021	0.1929449	0.2333	0.0718029
Run 23	0.1427994	0.2982937	0.1906697	0.2334992	0.0721678
Run 24	0.1556072	0.2259845	0.2124121	0.2369242	0.0767723
Run 25	0.143495	0.3543159	0.2091536	0.2353762	0.0715909
Run 26	0.144551	0.2674887	0.1928828	0.2384699	0.081472
Run 27	0.1426469	0.361062	0.1945009	0.2150616	0.0727128
Run 28	0.1438098	0.2808326	0.2080121	0.2344591	0.081724
Run 29	0.1448128	0.3072971	0.0715705	0.0806717	0.0754813
Run 30	0.1484476	0.2134748	0.2095459	0.2429817	0.0760764
Run 31	0.1428161	0.2399237	0.2149792	0.23223	0.0804688
Run 32	0.1443496	0.2349167	0.1840963	0.236487	0.0866962
Run 33	0.1461437	0.2411161	0.2191935	0.2329994	0.0727924
Run 34	0.1430712	0.2841059	0.1918908	0.2328094	0.0771223
Run 35	0.1440129	0.3408651	0.2114726	0.2330653	0.0744918

Run 36	0.1449723	0.2374653	0.2106136	0.2505709	0.0722097
Run 37	0.155903	0.2424062	0.0734121	0.237003	0.0761753
Run 38	0.1466027	0.2203274	0.227701	0.2320979	0.0831745
Run 39	0.1502291	0.2470489	0.1825768	0.2355866	0.0726368
Run 40	0.143747	0.3424248	0.207549	0.2320864	0.0713819
Run 41	0.1435391	0.2415865	0.2139785	0.2355514	0.077949
Run 42	0.154069	0.3452544	0.1923037	0.2323887	0.073685
Run 43	0.1432777	0.2477275	0.1933413	0.2329594	0.0765102
Run 44	0.1452005	0.2330815	0.215032	0.237134	0.076653
Run 45	0.1438029	0.3405902	0.1943358	0.2367342	0.0772408
Run 46	0.1540486	0.341228	0.2149806	0.2310327	0.0724801
Run 47	0.1453307	0.2421662	0.189721	0.2334679	0.0733459
Run 48	0.1426022	0.2390775	0.0713867	0.2369703	0.0715677
Run 49	0.1544028	0.2372234	0.2162995	0.2306645	0.0749335
Run 50	0.1442864	0.3503886	0.2140878	0.217441	0.0729634

Intel Core i5 520M

	Sequential	Parallel FS	Parallel FS + Spacing	Parallel FS + Padding	Parallel FS + S & P
Run 1	0.084880	0.085782	0.036394	0.037759	0.035863
Run 2	0.085164	0.092071	0.036957	0.038240	0.035942
Run 3	0.085194	0.097520	0.037173	0.043674	0.036421
Run 4	0.085333	0.116838	0.039118	0.043867	0.036699
Run 5	0.085340	0.117396	0.042244	0.060990	0.036787
Run 6	0.085447	0.119555	0.065374	0.078303	0.037076
Run 7	0.085468	0.121064	0.065441	0.080320	0.037163
Run 8	0.085649	0.122478	0.070888	0.080473	0.037166
Run 9	0.086041	0.122833	0.071614	0.080515	0.037234
Run 10	0.086063	0.126431	0.071843	0.080788	0.037367
Run 11	0.086136	0.147522	0.073237	0.081040	0.037461
Run 12	0.086269	0.147558	0.074111	0.081515	0.037498
Run 13	0.086529	0.147669	0.078568	0.081594	0.037563
Run 14	0.086671	0.148318	0.079253	0.081774	0.037716
Run 15	0.086778	0.148882	0.082583	0.082260	0.037773
Run 16	0.086835	0.149359	0.082889	0.082281	0.037803
Run 17	0.086894	0.149412	0.083346	0.082453	0.037814
Run 18	0.086922	0.149487	0.083366	0.082505	0.038688
Run 19	0.087093	0.149742	0.083532	0.082776	0.038772
Run 20	0.087104	0.150505	0.083630	0.083050	0.038957
Run 21	0.087192	0.151366	0.083760	0.083069	0.039489
Run 22	0.087208	0.151890	0.083812	0.083152	0.039503
Run 23	0.087265	0.152022	0.084123	0.083702	0.039647
Run 24	0.087271	0.152210	0.084478	0.084741	0.039837
Run 25	0.087297	0.152671	0.084525	0.085098	0.039856
Run 26	0.087323	0.153750	0.084631	0.087071	0.040086
Run 27	0.087432	0.154017	0.084995	0.087223	0.040208
Run 28	0.087566	0.154271	0.085021	0.087301	0.040242

Run 29	0.087617	0.154716	0.085741	0.088082	0.040508
Run 30	0.087668	0.154876	0.085939	0.088925	0.040867
Run 31	0.087696	0.155394	0.085996	0.089573	0.041131
Run 32	0.087700	0.156354	0.086130	0.090966	0.041171
Run 33	0.087881	0.156692	0.086776	0.096338	0.042474
Run 34	0.087925	0.157802	0.086862	0.097293	0.042621
Run 35	0.087960	0.159466	0.086913	0.097507	0.042685
Run 36	0.088336	0.159751	0.087424	0.097528	0.042690
Run 37	0.088421	0.161695	0.087499	0.097750	0.042830
Run 38	0.088689	0.164490	0.087521	0.098482	0.043274
Run 39	0.089500	0.167511	0.087654	0.098523	0.043725
Run 40	0.089635	0.173836	0.087661	0.098662	0.043928
Run 41	0.089701	0.179274	0.088607	0.099555	0.043947
Run 42	0.089973	0.179446	0.089201	0.105104	0.044681
Run 43	0.090087	0.182819	0.089671	0.105724	0.044786
Run 44	0.092762	0.184848	0.091036	0.106250	0.045009
Run 45	0.093898	0.189566	0.092395	0.107221	0.045137
Run 46	0.093944	0.189953	0.098260	0.107317	0.045725
Run 47	0.097575	0.192927	0.099534	0.107730	0.045842
Run 48	0.099108	0.195304	0.120271	0.109740	0.046145
Run 49	0.103327	0.197995	0.127399	0.110509	0.069305
Run 50	0.103458	0.198995	0.129841	0.121024	0.070142