Detecting False Sharing in OpenMP Applications Using the DARWIN Framework

Besar Wicaksono, Munara Tolubaeva, and Barbara Chapman

University of Houston,
Computer Science Department, Houston, Texas, USA
http://www2.cs.uh.edu/~hpctools

Abstract. Writing a parallel shared memory application that achieves good performance and scales well as the number of threads increases can be challenging. One of the reasons is that as threads proliferate, the contention among shared resources increases and this may cause performance degradation. In particular, multi-threaded applications can suffer from the false sharing problem, which can degrade the performance of an application significantly. The work in this paper focuses on detecting performance bottlenecks caused by false sharing in OpenMP applications. We introduce a dynamic framework to help application developers detect instances of false sharing as well as identify the data objects in an OpenMP code that cause the problem. The framework that we have developed leverages features of the OpenMP collector API to interact with the OpenMP compiler's runtime library and utilizes the information from hardware counters. We demonstrate the usefulness of this framework on actual applications that exhibit poor scaling because of false sharing. To show the benefit of our technique, we manually modify the identified problem code by adjusting the alignment of the data that are causing false sharing; we then compare the performance with the original version.

Keywords: Cache coherence, false sharing, OpenMP, DARWIN, hardware counter, OpenMP collector API.

1 Introduction

With the widespread deployment of multi-core processors, many applications are being modified to enable them to fully utilize the hardware. The de-facto standard OpenMP [4], a shared memory programming model, is a popular choice for programming these systems. OpenMP offers a simple means to parallelize a computation so that programmers can focus on their algorithm rather than expend effort managing multiple threads. However, the simplicity of OpenMP also masks some potential problems from the programmer. One of the well-known problems is avoiding false sharing [21].

False sharing may occur on multi-core platforms as a result of the fact that blocks of data are fetched into cache on a per-line basis. When one thread accesses data that happens to be on the same line as data simultaneously accessed

S. Rajopadhye and M. Mills Strout (Eds.): LCPC 2011, LNCS 7146, pp. 283–297, 2013.

by another thread, both need up-to-date copies of the same cache line. In order to maintain the consistency of the shared data, the processor may then generate additional cache misses that degrade performance. It can be very hard for the application developer to correctly identify the source of such performance problems, since it requires some amount of understanding of the way in which the hardware supports data sharing in the presence of private caches.

We have created a dynamic optimization framework for OpenMP programs, called DARWIN, that is based on the open-source OpenUH [10] compiler, and have shown how it can be used to optimize applications that exhibit ccNUMA data locality problems [23]. The core feature of this framework is its usage of the OpenMP collector API [6] to interact with a running OpenMP program. The collector API can track various OpenMP states on a per-thread basis, such as whether a thread is in a parallel region. DARWIN also utilizes hardware counters to obtain detailed information about the program's execution. When combined with its other capabilities, such as relating the performance data to the data structures in the source code, DARWIN is able to help the application developer to gain insights about dynamic code behavior, particularly in the hot-spots of a parallel program.

In this paper, we explain how false sharing may arise in OpenMP applications and how the DARWIN framework may be used to identify data objects in an OpenMP code that cause false sharing. We describe DARWIN's ability to interact with the OpenMP runtime library and to access hardware counters, which is the starting point for our false sharing detection strategy. We then discuss the two stages involved in our approach. The first stage observes the degree of coherence problem in a program. The second stage isolates the data structure that leads to the false sharing problem.

The paper is structured as follows: section 2 describes the false sharing problem. Section 3 gives an introduction to the DARWIN framework. After presenting our methodology in section 4, we discuss our experiments in section 5. Section 6 discusses prior research closely related to our work. Finally, section 7 summarizes our findings and offers conclusions.

2 False Sharing

In a multi-threaded program running on a multicore processor, data sharing among threads can produce cache coherence misses. When a processor core modifies data that is currently shared by the other cores, the cache coherence mechanism has to invalidate all copies in the other cores. An attempt to read this data by another core shortly after the modification has to wait for the most recent value in order to guarantee data consistency among cores.

The data sharing that occurs when processor cores actually access the same data element is called true sharing. Such accesses typically need to be coordinated in order to ensure the correctness of the program. A variety of synchronization techniques may be employed to do so, depending on the programming interface being used. Some of these techniques are locks, monitors, and semaphores. In the

OpenMP API, *critical* and *atomic* are two constructs that prevent the code they are associated with, from being executed by multiple threads concurrently. The *critical* construct provides mutual exclusion for code blocks in a critical section, whereas the *atomic* construct ensures safe updating of shared variables. When the order of the accesses is important, the OpenMP *barrier* and *taskwait* constructs can be used to enforce the necessary execution sequence.

In contrast to true sharing, false sharing is an unnecessary condition that may arise as a consequence of the cache coherence mechanism working at cache line granularity [2]. It does not imply that there is any error in the code. This condition may occur when multiple processor cores access different data elements that reside in the same cache line. A write operation to a data element in the cache line will invalidate all the data in all copies of the cache line stored in other cores. A successive read by another core will incur a cache miss, and it will need to fetch the entire cache line from either the main memory or the updating core's private cache to make sure that it has the up-to-date version of the cache line. Poor scalability of multi-threaded programs can occur if the invalidation and subsequent read to the same cache line happen very frequently.

```
int *local_count = (int*)malloc(sizeof(int)*NUM_THREADS*PADDING);
int *vector = (int*)malloc(sizeof(int)*VECTOR_SIZE);
for(i=0;i<COUNT;i++)
{
#pragma omp parallel
    {
        int tid = omp_get_thread_num()*PADDING;
        if(tid < 0) tid = 0;

        #pragma omp for
        for(j = 0; j < VECTOR_SIZE; j++)
            local_count[tid] += vector[j]*2;

        #pragma omp master
        {
            int k;
            for(k = 0; k<NUM_THREADS; k++)
            result += local_count[k];
        }
    }
}
```

Fig. 1. OpenMP code snippet with false sharing problem

Figure 1 shows a code snippet from an OpenMP program that exhibits the false sharing problem. This code will read each value of a vector, multiply it by two, and calculate the sum. Its performance is inversely proportional to the number of threads as shown in Table 1.

Mitigating the false sharing effect can lead to an astonishing 57x performance improvement for this code. The reason for the poor performance of the unoptimized code lies in the way it accesses the *local_count* array. When the *PADDING* variable is set to 1, the size of the array is equal to the number of threads. The cache line size of the underlying machine being used is 128 bytes, so even though

the threads access different elements of the array, they fetch the same cache line frequently and interfere with each other by causing unnecessary invalidations. By taking the cache line size into account and increasing the *PADDING* value, we can prevent threads from accessing the same cache lines continuously.

Table 1. Execution time of OpenMP code from Figure 1

Code version	Execution time(s)			
	1-thread	2-threads	4-threads	8-threads
Unoptimized	0.503	4.563	3.961	4.432
Optimized	0.503	0.263	0.137	0.078

3 DARWIN Framework

The DARWIN framework is a feedback dynamic optimization system that primarily leverages the features of the OpenMP collector API [6] to communicate with a running OpenMP program. Figure 2 illustrates the major components of the framework.

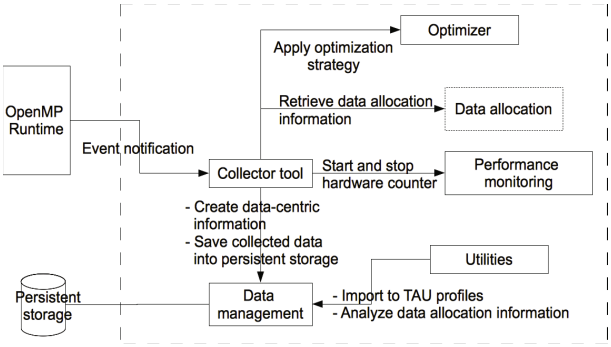


Fig. 2. The DARWIN Framework

The collector tool is the central component of DARWIN. It utilizes the OpenMP collector API to facilitate transparent communication with the OpenMP application. The OpenUH OpenMP runtime library throws a collector event notification that represents transition to a particular state in the OpenMP program's execution. For example, the *OMP_EVENT_FORK* event is thrown when the OpenMP master thread enters a parallel region. The collector tool catches the event notification and provides it to the event handler that performs the appropriate profiling activity, such as starting and stopping the performance measurement for each thread in a parallel region.

DARWIN has two execution phases, monitoring and optimization. This work focuses on the monitoring phase, which collects performance data to enable the

false sharing analysis. In the monitoring phase, the collector tool starts and stops the performance monitoring component when it receives a collector event notification about the start and end of an OpenMP parallel region, respectively.

Currently, the performance monitoring component utilizes Data Event Address Register (DEAR) of the Itanium 2 processor to capture samples of memory load operations, which contain meaningful information such as the referenced memory addresses and the load latency. Other processor models such as AMD Opteron and Intel Core support similar functionality via Instruction Base Sampling (IBS) and Precise Event Base Sampling (PEBS), respectively. Support for these hardware counters will be included in the future work.

At the end of the monitoring phase, the collector tool invokes the data manager to create data-centric information by relating the referenced memory addresses in the sampling results with the data structures at the source code level, thus making the information comprehensible to the programmer. Finally, the data manager saves all of the gathered information to persistent storage.

To support the creation of data-centric information, the data allocation component captures the allocation information of global, static, and dynamic data, which includes the parallel region id, thread id, starting address, allocation size, and the variable name. The allocation information for global and static data is provided by the symbol table of the executable file. For dynamic data, we interpose the memory allocation routines to capture the resulting starting address and allocation size parameter. We also retrieve the program counter of the allocation caller, which is used to find the caller's name and line number from the debug information. Both name and line number are used as the variable name for dynamic data. It is possible that more than one dynamic data allocation are attributed with the same variable name, which may indicate that these data are elements of an array, list, or tree. The allocation information is stored into a B-Tree [1] that offers very good performance for searching, especially when the number of allocation records, and hence the search space, is very large.

DARWIN provides several utilities to support data analysis by the programmer. One tool is used to export the collected performance data into text files that follow the Tuning and Analysis Utilities (TAU) [18] profile format. A second tool can be used to analyze the data allocation information.

TAU is a portable toolkit for performance analysis of parallel programs that are written in Fortran, C, C++, and Python. The programmer can run TAU's *Paraprof* [20] to observe the behavior of the program through its 3D visualization capabilities, and draw conclusions about the observed performance bottlenecks.

4 False Sharing Detection Methodology

Our approach for determining the data that exhibits false sharing consists of two stages. The first stage checks whether the cache coherence miss contributes to a major bottleneck in the program. The second stage isolates the data structures that cause the false sharing problem.

4.1 Stage 1 : Detecting Cache Coherence Problem

False sharing is a cache coherence problem related to the way processors maintain memory consistency. Therefore, observing the hardware behavior is a good way to determine if a program is suffering from coherence misses. Modern processors accommodate a performance monitoring unit (PMU) that can provide hints about the potential existence of cache coherence problems.

For example, the Intel Itanium 2 processor supports the PMU event *BUS_MEM_READ_BRIL_SELF* that gives the number of cache line invalidation transactions [22]. The Intel Core i7 family supports an event called *MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM* that indicates the number of retired memory load instructions that hit dirty data in sibling cores [7]. If a large number of each event is detected during a program's execution, it indicates that a serious cache coherence problem can occur when the program is executed with multiple threads.

4.2 Stage 2 : Isolating the Data Structures

To identify those data structures that have a major false sharing problem, our method starts by observing the accesses to cache lines with high level symptoms, which are a high memory access latency, and a large number of references. When a cache coherence miss occurs, a read operation from the processor needs to fetch the data from another processor's cache or wait until the memory is updated with the latest version, after which the processor reads directly from memory. This means that a cache coherence miss has longer latency than other kinds of cache misses. It has been reported [9] that a cache coherence miss on the Itanium 2 processor can exceed 180-200 cycles, while the average latency for reading from memory is 120-150 cycles. We are also aware that a modest amount of false sharing misses will not lead to a significant problem for application performance. Therefore, cache lines with a low number of references are ignored.

After the data structures showing the symptoms above have been identified, we examine the data allocation information to look for other data structures that share the problematic cache line. The search is also performed within the elements of the data structure if it is a linear or non-linear data structure. As discussed in section 3, these kinds of data structures are represented by multiple allocations with the same variable name. If the search attempt returns one or more results, we conclude that the problem is due to false sharing. Otherwise, we observe the data access pattern of the data structures that experience the false sharing symptoms.

False sharing can exist on a data structure that is accessed by multiple threads, where each thread only accesses a portion of the data. It is possible that some elements near the boundary of two disjoint data portions are in the same cache line. If the data structure size is small, e.g. it takes up about as many bytes as there are in a cache line, most elements of the data structure might be contained in the same cache line causing an increased risk of false sharing.

The results of our false sharing detection are validated by performing manual optimization to the source code that allocates the falsely shared data, without making extensive program restructuring. The optimization is performed by modifying the data layout to prevent falsely shared data from residing on the same cache line. We use GCC's *aligned* variable attribute and *posix_memalign* function to allocate data on the cache line boundary. The result of the detection is considered to be valid when the performance of the optimized code is substantially better.

5 Experiments

To evaluate our methodology, we performed experiments using the Phoenix suite [17] that implements MapReduce for shared memory systems. Phoenix provides eight sample applications parallelized with the Pthreads library that we have ported to OpenMP. The programs were compiled using the OpenUH compiler with optimization level O2. Each sample program included several input configurations. We chose the one that results in a reasonable execution time (not too long nor too short). The platform that we used was an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium 2 processors per node running the SUSE 10 operating system. The experiment was conducted using an interactive PBS[1] with four compute nodes and eight physical CPUs. Each program was executed with the *dplace* command to enforce thread affinity. The overall wall clock execution time was measured using the shell's *time* function.

5.1 Results of Detecting Cache Coherence Problem

Figure 3 presents the speedup for each program executed with different numbers of threads. The speedup is defined as the execution time of the serial run divided by the execution time of the multi-threaded version. The experiment for each configuration was performed three times and the average execution time was determined.

From all of the programs, only *kmeans* experienced an ideal speedup. The *matrix_multiply* and *pca* programs had reasonable results but did not come close to the ideal speedup. The speedup of *histogram*, *reverse_index*, and *word_count* programs was even lower than that obtained by these benchmarks. The *linear_regression*, and *string_match* programs suffered from a heavy slowdown when using more than one thread.

To determine whether the slowdown or poor scaling came from the cache coherency problem, we observe the cache invalidation event measurement. Table 2 shows the measurement results. It shows that *histogram*, *linear_regression*, *reverse_index*, *string_match*, and *word_count* had a very large number of cache invalidation events when using higher numbers of threads. This is an indication that these programs suffer from a cache coherence problem. Although both

[1] Portable Batch Session.

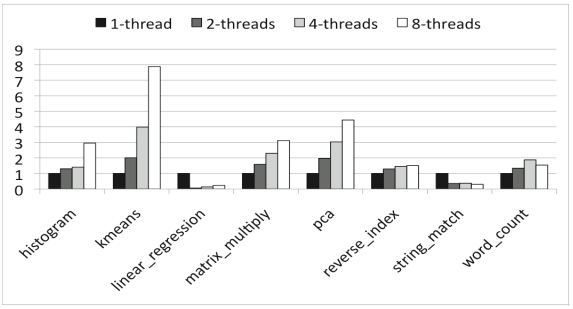


Fig. 3. The speedup of the original program

Table 2. Cache invalidation event measurement result

Program Name	**Total Cache Invalidation Count**			
	1-thread	2-threads	4-threads	8-threads
histogram	13	**7,820,000**	**16,532,800**	**5,959,190**
kmeans	383	28,590	47,541	54,345
linear_regression	9	**417,225,000**	**254,442,000**	**154,970,000**
matrix_multiply	31,139	31,152	84,227	101,094
pca	44,517	46,757	80,373	122,288
reverse_index	4,284	89,466	217,884	**590,013**
string_match	82	**82,503,000**	**73,178,800**	**221,882,000**
word_count	4,877	**6,531,793**	**18,071,086**	**68,801,742**

programs experienced sub-linear speedup in the number of threads as shown in Figure 3, *matrix_multiply* and *pca* had a low number of cache invalidation events.

An examination of the code reveals that the *matrix_multiply* program writes its results into an output file at the end of the program and that the *pca* program has many synchronizations using critical regions, which limits the speedup of both of them. The number of cache invalidation events in *reverse_index* program was pretty high when executed with eight threads, but the increase was not as extreme as with *histogram*, *linear_regression*, *string_match*, and *word_count*. We conclude that the coherence problem may not contribute significantly to the overall performance of *reverse_index*.

We then focused on the programs with a high number of cache invalidation events and collected the information required for identifying the variables that exhibit a false sharing problem. To help us analyze the collected information, we used the TAU *Paraprof* utility that can visualize the collected memory references information of each thread in a parallel region.

Due to the page limitation, we only present the analysis result for *linear_regression*, and *string_match* as shown in Figure 4 and 6 in the following subsections. Both programs experienced an enormous amount of cache invalidations and were slower when executed with multiple threads.

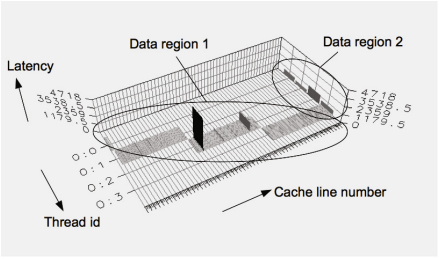
5.2 Linear_Regression

Figure 4(a) shows the average memory latency of the accesses to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the memory access latency. The depth axis shows the thread id. There are two distinct data regions that can be identified in this figure. In data region 1, two cache lines referenced by thread 2 experienced a high latency, while the accesses from all threads to the cache lines in data region 2 had higher latency than most accesses to cache line in data region 1. Next we examine whether the two cache lines in data region 1 and the cache lines in data region 2 also had a high number of references.

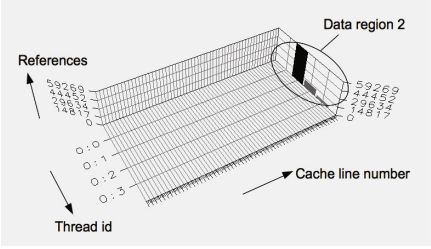
Figure 4(b) shows the number of memory references to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the number of references. The depth axis shows the thread id. The accesses to data region 1 were not shown by *Paraprof* because the number of references to this data region was very small compared to the number of references of region 2.

Figure 4(c) gives the total amount of memory latency for each cache line. The total latency is defined as the average memory latency multiplied by the total number of references. Since data region 2 dominated the total memory access latency, we suspected the data structures in this data region to be the leading cause of the false sharing problem.

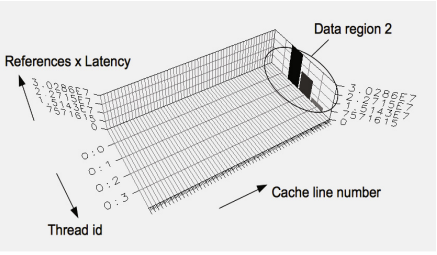
According to DARWIN's data-centric information, data region 2 contained accesses to a variable named *main_155*. It was dynamic data allocated by the



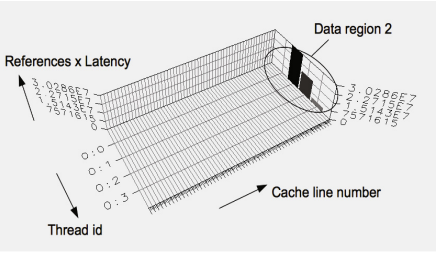
(a) Average memory latency



(b) Memory reference count



(c) Total memory latency



(d) main_155 access pattern

Fig. 4. *Linear_regression* memory access visualization

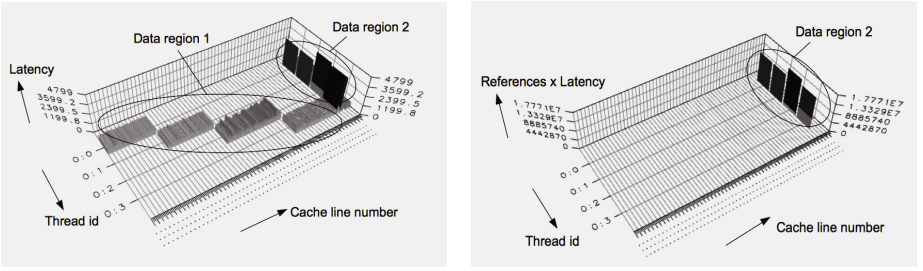
master thread in the main function, at line number 155. Its access pattern, presented in Figure 4(d), shows that updates to this data were distributed among the threads, and that in some cases, multiple threads were accessing the same cache line. In this case, the accesses from thread 1 to 3 hit elements of *main_155* that reside in the same cache line and have much higher latency than the accesses from thread 1. Therefore, we concluded that accesses to *main_155* caused the main false sharing problem in *linear_regression*. We also found a similar situation with *histogram*, where dynamic data shared among threads was causing the most significant bottleneck. We validated the findings by adjusting the data allocation alignment using the *aligned* attribute, as shown in Figure 5. The validation results are presented in Section 5.4.

```
typedef struct
{
    POINT_T *points  __attribute__((aligned (256)));
    int num_elems;
    long long SX, SY, SXX, SYY, SXY;
} lreg_args;
...
tid_args = (lreg_args *)calloc(sizeof(lreg_args),num_procs);
```

Fig. 5. Adjusting the data alignment in *linear_regression*

5.3 *String_match*

Figures 6(a) and 6(b) show the average and total memory access latency of each cache line, respectively. As with *linear_regression*, we identified two distinct



(a) Average memory latency (b) Total memory latency

Fig. 6. *String_match* memory access visualization

data regions with different access patterns. Both figures clearly show that the accesses to data structures in data region 2 were causing the major bottleneck of this program. According to the data-centric information, data region 2 contained the memory accesses to variable *key2_final* and *string_match_map_266*. The first variable was a global variable allocated in the same cache line with other global variables. We encountered the same situation in *reverse_index*, and *word_count*,

where some of their global variables resided in the same cache line and were accessed frequently and had high latency.

string_match_map_266 was a dynamic data object allocated in the *string_match_map* function at line number 266. In contrast to the *main_155* variable, whose accesses were distributed among threads in *linear_regression*, *string_match_map_266* was allocated only by thread 3 and privately used within this thread. However, the problematic cache line of this variable was shared with a dynamic variable allocated by other thread.

Table 3 presents several data structures allocated in this program. It confirms that *key2_final* resided in the same cache line with *fdata_keys*. The first cache line of *string_match_map_266*, which is the problematic one, was shared with *string_match_map_268* that was allocated by thread 2.

Table 3. Data allocation information of several variables in *string_match*

Parallel region id	Thread id	Variable name	Starting cache line	Last cache line	Size (bytes)
0	0	fdata_keys	**0x00004a80**	**0x00004a80**	8
0	0	key_2_final	**0x00004a80**	**0x00004a80**	8
0	0	key_3_final	0x00004b00	0x00004b00	8
2	2	string_match_map_268	0x0c031f00	**0x0c032300**	1024
2	3	string_match_map_266	**0x0c032300**	0x0c032700	1024

The findings were validated by adjusting the data allocation alignment using the *aligned* attribute on *key2_final* and *fdata_keys*. We substitute the *malloc* routine for the allocation of *string_match_map_266* with the *posix_memalign* routine. These attempts are presented in figure 7.

```
char *key2_final __attribute__((aligned (256)));
char *fdata_keys __attribute__((aligned (256)));
...
posix_memalign(&cur_word,256,MAX_REC_LEN);
```

Fig. 7. Adjusting data alignment in *string_match*

5.4 Results of Memory Alignment

Figure 8 shows the speedup of each program over the original one after we performed the adjustments to the source code. The speedup is defined as the execution time of the original program divided by the execution time of the modified one. The memory alignment attempt produced visible improvement of the performance of *histogram*, *linear_regression*, *string_match*, and *word_count* with up to 30x speedup. The *reverse_index* program did not experience any significant improvement. However, this does not mean that our finding on *reverse_index* is invalid.

Table 4 presents the cache invalidation count of *reverse_index* after the memory alignment. It shows that the memory alignment successfully reduced the number of cache invalidations.

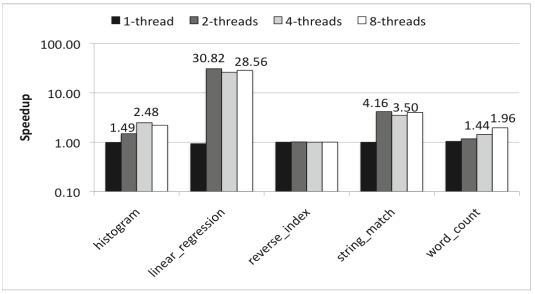


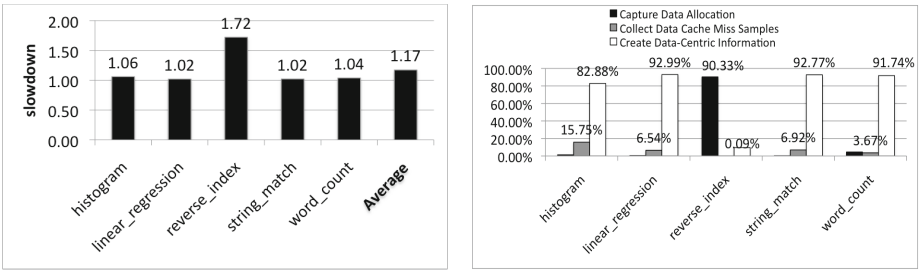
Fig. 8. The speedup after adjusting the memory alignment

Table 4. *Reverse_index* cache invalidation event measurement result

Code Version	Total Cache Invalidation Count			
	1-thread	2-threads	4-threads	8-threads
Unoptimized	4,284	89,466	217,884	590,013
Optimized	4,275	60,115	122,024	240,368

5.5 Performance Overhead

To determine overheads, we compare the execution time of the monitoring attempt with the original program execution time. Figure 9(a) shows the slowdown of the monitoring phase, defined as the monitoring execution time divided by the original program execution time. The monitoring overhead consisted of the time taken to capture data allocation, collect the data cache miss samples, and create data-centric information. Each of them was measured using the *gettimeofday* routine. Figure 9(b) gives the percentage of each overhead component in the total overhead time.



(a) (b)

Fig. 9. (a) The slowdown of the monitoring phase. (b) Overhead breakdown.

According to figure 9(a) the monitoring phase generated a slowdown ranging from 1.02x to 1.72x, with the average around 1.17x. *reverse_index* had the highest

overhead with 1.72x slowdown, while the rest of the programs had less than 1.1x slowdown.

Figure 9(b) shows that the majority of *reverse_index*'s overhead was incurred during capture of the data allocation information. The reason for this is because *reverse_index* contained an excessive number of dynamic data allocations. *reverse_index* had 99,675 allocations, while the other programs had less than 50 allocations. Based on our investigation, traversing the stack frame to get the program counter during the dynamic data allocation can consume a significant amount of time, especially when the program has a large number of allocations. Reducing the need to traverse the stack frame is a subject for future work.

6 Related Work

False sharing is a performance problem that occurs as a consequence of the cache coherence mechanism working at cache line granularity. Detecting false sharing accurately requires complete information on memory allocation and memory (read and write) operations from each thread. Previous work [5,15,13,14] has developed approaches for memory analysis that use memory tracing and cache simulation. This starts by tracking the memory accesses (both loads and stores) at runtime. A cache simulator then takes the captured data to analyze the sequence of each memory operation and determine the type and amount of cache misses generated during the simulation. The main drawback of this approach is within the memory tracing part, which can incur very large overheads. A memory shadowing technique was used [24] in an attempt to minimize the overhead of tracking the changes to the data state. Our approach does not analyze the sequence of memory operations. We exploit higher level information, such as the latency, and the number of memory access references. By utilizing the hardware performance monitoring unit, the overhead of capturing this information can be minimized. Furthermore, our approach can quickly pinpoint the falsely shared data that has a significant impact on the performance.

Others [12,16,7,14] also use information from the hardware performance monitoring unit to support performance analysis. HPCToolkit[12], and Memphis[16] use the sampling result from AMD IBS to generate data centric information. HPCToolkit utilizes the information to help a programmer find data structures with poor cache utilization. Memphis focuses on finding data placement problem on ccNUMA platform. Intel PTU[7] utilizes event-based sampling to identify the data address and function that is likely to experience false sharing. Our work is more focused on finding the actual data objects that suffer greatly from false sharing. Our work is similar to [14] in terms of the utilization of the performance monitoring unit to capture memory load operations. While [14] directs the captured information to the cache simulator in order to analyze the memory access sequences, our approach uses it to analyze the latency, and the number of references.

Several attempts have been made to eliminate the false sharing problem. For example, careful selection of runtime scheduling parameters such as chunk size and chunk stride when distributing loop iterations to threads has been used to prevent false sharing [3]. Proposed data layout optimization solutions include array padding [8] and memory alignment methods[19]. A runtime system called Sheriff [11] performs both detection and elimination of false sharing in C/C++ applications parallelized using the Pthreads library. Sheriff eliminates false sharing by converting each thread into a process and creating a private copy of shared data for each process. It detects false sharing by observing each word in the private copy of each process. False sharing is detected when there is a modification to a word adjacent with another word of another process's private copy, and both words reside on the same cache line. Our work is primarily concerned with detecting the false sharing problem, rather than eliminating it.

7 Conclusion

This paper describes our implementation of a technique to detect false sharing in OpenMP applications in the DARWIN framework. DARWIN provides features to enable communication with the OpenMP runtime library, and to capture the data access pattern.

The technique consists of two stages, which are 1) detection of coherence bottlenecks in the program with the help of hardware counters and 2) identification of data objects that cause the false sharing problem. The second stage utilizes the data allocation and access pattern information to distinguish the data structures that cause major bottlenecks due to false sharing.

To observe the effectiveness of our method, the technique was applied to several Phoenix programs that exhibit false sharing. Bottlenecks caused by the cache coherence problem were detected, and the data causing serious false sharing problems were identified. Optimizing the memory alignment of the problematic data greatly improved the performance, thus indicating that our method is capable of identifying the data responsible for the false sharing problem. DARWIN's monitoring approach created an average of 1.17x slowdown of the programs used. Traversing the stack frame when capturing the dynamic data allocation can incur large overhead, especially when the program has a large number of dynamic data allocations.

Acknowledgement. The authors thank their colleagues in the HPCTools group for providing useful feedback during the writing of this paper and the Texas Learning and Computation Center (TLC2) for use of their hardware[2]. This work is supported by the National Science Foundation under grant CCF-0702775 (see also `http://www.cs.uh.edu/~hpctools/darwin`).

[2] `http://tlc2.uh.edu`

References

1. Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indices. Mathematical and Information Sciences Report No. 20 (1970)
2. Chapman, B., Jost, G., Pas, R.V.D.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press (2008)
3. Chow, J.-H., Sarkar, V.: False Sharing Elimination by Selection of Runtime Scheduling Parameters. In: Proceedings of the ICPP (1997)
4. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering (1998)
5. Günther, S.M., Weidendorfer, J.: Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation. In: Proceedings of the WBIA (2009)
6. Hernandez, O., Chapman, B., et al.: Open Source Software Support for the OpenMP Runtime API for Profiling. In: P2S2 (2009)
7. Intel. Avoiding and Identifying False Sharing Among Threads (2010)
8. Jeremiassen, T.E., Eggers, S.J.: Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformations. SIGPLAN (1995)
9. Kim, J., Hsu, W.-C., Yew, P.-C.: COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications. In: ICPP (2007)
10. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. In: CPC (2006)
11. Liu, T., Berger, E.: Sheriff: Detecting and Eliminating False Sharing. Technical report, University of Massachusetts, Amherst (2010)
12. Liu, X., Mellor-Crummey, J.: Pinpointing Data Locality Problems Using Datacentric Analysis. In: CGO (2011)
13. Marathe, J., Mueller, F.: Source-Code-Correlated Cache Coherence Characterization of OpenMP Benchmarks. IEEE Trans. Parallel Distrib. Syst. (June 2007)
14. Marathe, J., Mueller, F., de Supinski, B.R.: Analysis of Cache-Coherence Bottlenecks with Hybrid Hardware/Software Techniques. ACM TACO (2006)
15. Martonosi, M., Gupta, A., Anderson, T.: MemSpy: Analyzing Memory System Bottlenecks in Programs. SIGMETRICS Perform. Eval. Rev. 20, 1–12 (1992)
16. McCurdy, C., Vetter, J.: Memphis: Finding and Fixing Numa-Related Performance Problems on Multi-Core Platforms. In: ISPASS (2010)
17. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: HPCA (2007)
18. Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. Int. J. High Perform. Comput. Appl. (2006)
19. Torrellas, J., Lam, H.S., Hennessy, J.L.: False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Trans. Comput. 43, 651–663 (1994)
20. University of Oregon. ParaProf User's Manual
21. van der Pas, R.: Getting OpenMP Up To Speed (2010)
22. Vogelsang, R.: SGA Altix Tuning OpenMP Parallelized Applications (2005)
23. Wicaksono, B., Nanjegowda, R.C., Chapman, B.: A Dynamic Optimization Framework for OpenMP. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 54–68. Springer, Heidelberg (2011)
24. Zhao, Q., Koh, D., Raza, S., Bruening, D., Wong, W.-F., Amarasinghe, S.: Dynamic Cache Contention Detection in Multi-threaded Applications. In: VEE (2011)