Compile-Time Detection of False Sharing via Loop Cost Modeling

Munara Tolubaeva
Computer Science Department
University of Houston
Houston, TX USA
munarat@cs.uh.edu

Yonghong Yan
Computer Science Department
University of Houston
Houston, TX USA
yanyh@cs.uh.edu

Barbara Chapman
Computer Science Department
University of Houston
Houston, TX USA
chapman@cs.uh.edu

***Abstract*—False sharing, which occurs when multiple threads access different data elements on the same cache line, and at least one of them updates the data, is a well known source of performance degradation on cache coherent parallel systems. The application developer is often unaware of this problem during program creation, and it can be hard to detect instances of its occurrence in a large code. In this paper, we present a compile-time cost model for estimating the performance impact of false sharing on parallel loops. Using this model, we are able to predict the amount of false sharing that could occur when the loop is executed, and can indicate the percentage of program execution time that is due to maintaining the coherence of data from false sharing. We evaluated our model by comparing its predictions obtained on several computational kernels using 2 to 48 threads against that from actual execution. The results showed that our model can accurately quantify the impact of false sharing on loop performance at compile-time.**

***Keywords*-False Sharing, Performance Prediction and Modeling, Compiler, Loop Cost Modeling**

I. Introduction

Multicore processors have become ubiquitous and are used by many different users, ranging from inexperienced programmers to professionals and scientists. Parallel programming, once considered for the high end computing, is now becoming a common practice and required expertise for average programmers. Multithreaded programming APIs such as OpenMP [5] provide a productive programming environment for creating parallel programs from the sequential versions. However, obtaining scalable performance on large parallel machines still requires significant amount of effort in performance tuning, especially with regards to data locality. It becomes necessary for programmers to understand concepts such as caches and locality, data and work sharing, and synchronizations in order to write parallel programs that will deliver good performance.

The false sharing (FS) problem, which occurs when multiple threads read/write the same cache line, but different memory elements on the line, is a well-known performance degrading issue in parallel programs. The performance impact FS could incur when executing a victim[1] loop may be as astonishing as 60% [21]. FS occurs at the cache line granularity and closer to the architecture level; detection of FS is not obvious for both average and experienced programmers. It is related to private caches and the cache coherency protocol that enforces a consistent view of the memory by all private caches, concepts that are often hard to understand for average programmers.

FS reflects performance degrading data access pattern, but can only reveal such pattern after program execution. It is often hidden from programmers' view during the program creation and it is hard to correlate the performance degradation to FS when a program becomes very large. There have been substantial amount of studies conducted to detect FS at runtime [8], [15], [16], [11], [9], [23], [13], [14], [20], [21]. Other studies have been done in the direction of eliminating the problem by means of data padding, scheduling techniques and specific data allocation and layout methods [7], [10]. However, to the best of our knowledge, none of the existing approaches have been integrated into a compiler, so that the compiler is able to detect and to consider the possible overhead incurred by FS when conducting performance estimations.

The goal of this work is to develop a compile-time cost model [12] for FS, and to estimate the performance impact of FS on parallel loops using the defined model. With the help of cost models, the compiler is able to estimate whether the specific transformation is profitable in terms of execution time and determine the optimal level of the transformation, if applied. The FS cost model defined in this paper features: 1) ability to output the total number of FS cases that will occur during execution of the parallel loop; and 2) ability to analyze the performance impact of FS on a parallel loop as a percentage of execution time; and 3) introduces a linear regression model to reduce the modeling time by approximation without impacting its accuracy.

We validated our model by comparing the FS overhead percentages obtained by measuring from the execution time against the ones computed by our model. The modeling results are comparable to the real execution behavior from 2 to 48 threads tested, showing the model can accurately quantify the FS impact at compile-time. The FS cost model will be used by compilers to guide the parallel loop transformations by providing more accurate timing estimation for parallel loops. Compilers will also be able to use information

[1]data object or codes from which false sharing occurs

from our model to perform automatic optimizations, such as changing the loop iteration behavior or data alignment to eliminate FS or minimize its impacts. These modeling and estimation results could also be useful for programmers for performance tuning and locality optimizations. The quantitative performance impact information will be especially helpful when tuning an application for specific hardware architectures.

The paper is organized as follows: Section II describes the background and motivation of our work. Our FS modeling and prediction techniques are explained in Section III. Section IV discusses the experiments we have conducted and the results obtained. Related work is discussed in Section V. Lastly, Section VI concludes the paper and introduces future work.

II. Background and Motivation

In this section, we discuss the performance impact of FS using an OpenMP version of the *linear regression* kernel from Phoenix benchmarks [17]. We then introduce compile-time cost models used in the Open64 compiler [4] to drive loop nest optimizations.

A. False Sharing

FS occurs when two or more threads access the same cache line, but different elements of the line and at least one of these accesses is a write operation [6]. For example, consider two threads A and B executing on two distinct cores of a parallel system where cache coherency is maintained by a write-invalidate protocol. If the threads perform memory references that cause the processor to fetch the same cache line to their private caches, and when one thread modifies some element on the cache line, the state of the line stored in the other thread's private cache becomes invalid due to the protocol. Right after the invalidation, when the second thread accesses any element of the invalidated cache line, a cache miss occurs. The second thread will unnecessarily have to fetch the line from memory, even if the requested elements were not modified. If FS happens frequently during the execution of the program, then the overhead caused by the problem will significantly decrease the overall performance of the program.

Figure 1 shows an OpenMP version of one of the Phoenix benchmarks[2], the *linear regression* kernel. The *schedule(static,chunk_size)* clause on the loop directive causes the iterations of the loop to be distributed to threads in a round-robin fashion, where each thread gets *chunk_size* number of iterations of the outer loop at a time.

Figure 2 shows the execution time of the *linear regression* kernel with different chunk size configurations. The kernel clearly exhibits FS impact on performance because of the small chunk size. Since the chunk size is 1, threads

[2]Phoenix benchmarks implement MapReduce for shared memory systems.

```
#define N 9600
#define M 28887
#pragma omp parallel for private(i,j) schedule(static,1)
  for(j=0; j<N; j++)
     for (i = 0; i < M/num_threads; i++)
     {
    tid_args[j].SX  += tid_args[j].points[i].x;
    tid_args[j].SXX += tid_args[j].points[i].x *
             tid_args[j].points[i].x;
    tid_args[j].SY  += tid_args[j].points[i].y;
    tid_args[j].SYY += tid_args[j].points[i].y *
             tid_args[j].points[i].y;
    tid_args[j].SXY += tid_args[j].points[i].x *
             tid_args[j].points[i].y;
     }
```

Figure 1. OpenMP version of the *linear regression* kernel

execute consecutive iterations of the outer loop. Therefore, each thread accesses consecutive element of the *tid_args* array which causes FS. However as we increase the chunk size value, the execution time gradually decreases because threads will be accessing non-neighbor elements of the array. By increasing the values of the chunk size from 1 to 30, we are reducing the FS overhead and improving the execution time of the kernel by up to 30%.

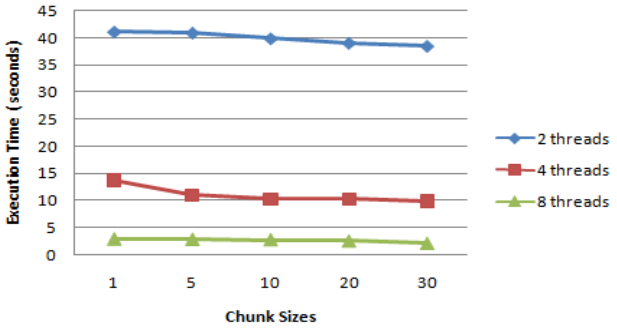

Figure 2. Execution time vs. chunk size of linear regression kernel.

FS is a performance degrading data access pattern when writing parallel codes for cache-coherent shared memory machines. The performance impact can sometimes increase proportionally with the number of threads executing the program. FS occurs at cache line granularity during the program execution, and it is often hidden from programmers' view during the program creation. It is a non-trivial process to correlate performance degradation to FS and then identify the data structure and codes that cause the FS.

B. Cost Models

In an optimizing compiler, loop nest optimizations (LNO) such as loop interchange, tiling, and unrolling, are widely used techniques for improving the performance of loops. Compiler transformations in LNO improve spatial locality of loops by changing the loop structures and the order of iterations. The locality and performance benefits from LNO

depend largely on the parameters of loop transformations such as the unrolling factor, the tile size. Poorly-chosen parameters will degrade the performance of a loop.

One approach for choosing good parameters is to use a cost model. Before applying a specific transformation with certain parameters, the compiler uses analytical models to estimate the costs of executing the loops in its original version and in the transformed version. The compiler then decides whether the transformation is beneficial or not by comparing the two costs. The costs are often calculated as CPU cycles needed to execute two versions of the loop, considering hardware architectures and software environment that affect the loop performance, such as cache organization, processor frequency, and runtime overhead.

We introduce a set of cost models used in the LNO phase of the Open64 compiler [22], which include *Processor model*, *Cache model*, and *Parallel model*.

1) Processor Model: The processor model estimates the time, in CPU cycles, needed to execute one iteration of the loop ($Machine_c\_per\_iter$) by modeling computational resources, registers and operation latencies as shown in Figure 3. The model tries to predict the scheduling of instructions ($Resource_c$) given the available amount of resources such as arithmetic/floating point units, memory and issue units. It considers the dependencies ($Dependency\_latency_c$) among instruction/memory operations and estimates the processor stalls caused by latencies. The Open64 compiler uses the processor model to make decisions regarding the best loop unrolling factor to apply to the loop. Detailed explanation of each function can be found in [12], [22].

$$Machine_c\_per\_iter = Resource_c + Dependency\_latency_c + Register\_spilling_c$$
$$Resource_c = maximum(Op_c, MEM\_ref_c, Issue_c)$$
$$MEM\_ref_c = (Num\_fp\_refs + Num\_int\_refs)/Num\_mem\_units$$
$$Issue_c = Num\_inst/Issue\_rate$$
$$Dependency\_latency_c = maximum(Sum\_of\_latencies_i/Sum\_of\_distances_i)$$
$$Register\_spilling_c = (Reg\_used - Target\_regs) * [Num\_reg\_refs/(Scalar\_regs + Array\_regs)]$$
$$Reg\_used = Base\_regs + Scalar\_regs + Array\_regs$$

Figure 3. Open64's Processor model

2) Cache Model: The cache model predicts the number of cache misses and estimates additional cycles needed to execute one iteration of an inner loop. In addition, the model is responsible for identifying the best possible loop block size for loop tiling. The number of cache misses is determined by summing up the footprints at the loop level [22]. The footprint is the number of bytes of single data reference in a cache. Due to spatial data locality, footprints of consecutive array references are counted only once. For example, references a[i] and a[i+1] lie in the same reference group, thus have only one footprint. When there is a reference to a[i], the cache line containing a[i] would be placed in a cache. The unused data in the same cache line would also be considered when counting the footprints. When the total amount of footprints is gathered, the model compares whether the footprint size is larger than the cache size; if so, a cache miss is considered to occur.

Figure 4 shows the equations of the cache model. The Translation Look-aside Buffer (TLB) cost is computed in the same way as the cache cost, because the TLB is modeled as another level of cache.

$$TLB\_miss = Num\_array\_ref - TLB\_entries, if(Num\_array\_ref - TLB\_entries > 0)$$
$$TLB_c = TLB\_miss\_penalty * TLB\_miss$$
$$Cache_c = \sum_{i}^{Levels} (Clean\_footprint_i * Clean\_penalty_i + Dirty\_footprint_i * Dirty\_penalty_i)$$

Figure 4. Open64's Cache model

3) Parallel Model: The parallel model helps the compiler to decide whether the parallelization of a loop is possible and if so which loop level is the best candidate for parallelization. The model makes use of the Processor and Cache models that are discussed in the previous sections, and also considers the cost of loop and parallel overheads as shown in Figure 5. The loop overhead considers the time needed to increment the loop indices, to check the loop boundary condition for each iteration which are aggregated into $Loop\_overhead\_per\_iter_i$. The parallel overhead is the time taken to actually execute the parallel loop. It includes overheads due to parallel startup, scheduling iterations, synchronizations and worksharing between threads [12]. In this work, the parallel overhead will refer to the OpenMP overhead incurred when parallelizing the loop via OpenMP constructs.

$$Total_c = Machine_c + TLB_c + Cache_c + Loop\_overhead_c + Parallel\_overhead_c$$
$$Machine_c = Machinec\_per\_iter * Num\_loop\_iter/Num\_threads$$
$$Loop\_overhead_c = Loop\_overhead\_per\_iter_i * Num\_loop\_iter/Num\_threads$$
$$Parallel\_overhead_c = Parallel\_startup_c + Parallel\_const\_factor_c * Num\_threads$$

Figure 5. Open64's Parallel model

C. Modeling Caching Effects from Parallel Executions

As discussed in Sections II-B2 and II-B3, the cache model analyzes the spatial and temporal locality of single thread execution, while the parallel model focuses on the worksharing benefits from concurrent execution of threads. However, neither of them, nor their combination takes into account the performance impact from the interference or contention for resources between parallel threads such as the FS effects, the competition to use shared cache or memory bus. With increasing number of cores and the decrease of average memory and bus bandwidth per cores, such interference

and contention will have significant performance impacts for applications.

This paper studies one of these interferences, FS, and enhances the Open64's cost model to include FS effects, as shown in Equation 1, for more accurate performance estimations. In Section III we will present our cost model for $False\_Sharing_c$, and how we use the model to detect FS at compile-time.

$$Total_c = False\_Sharing_c + Machine_c + Cache_c + TLB_c + Parallel\_Overhead_c + Loop\_Overhead_c \quad (1)$$

III. Methodology

Our FS cost model estimates the number of FS cases in a parallel loop at compile-time, and computes the overhead cost incurred by the problem on the whole execution of the loop. For wide applicability, we use OpenMP parallel loops in this model. Given an OpenMP parallel loop, there are four steps to analyze the cost incurred by FS:

1) Obtain array references made in the innermost loop of a loop nest
2) Generate a cache line ownership list for each thread
3) Apply a stack distance analysis to each cache line ownership list
4) Detect false sharing

As we mentioned, FS is often only revealed at runtime and is sensitive to lots of details about how the program is being executed, e.g. the alignment of allocated memory, the number of threads working on the victim data, and other background applications that may compete for the cache resources. It is necessary to supply enough runtime information to the compiler when estimating the FS effects. In this model, the compiler needs information about the number of threads executing the loop, loop boundaries, step sizes, index variables and the *chunk size*, if specified, for the OpenMP parallel loop. The chunk size is the number of iterations of a loop that are distributed to each thread. In this work, we assume that chunks of a loop are distributed to threads in a round-robin fashion. If the loop boundaries are not known at compile-time, the model only outputs the FS rate estimated per full cycle of iterations executed by all of the threads. One full cycle of iterations executed by the thread team is the sum of iterations executed by each thread in one chunk size.

A. Obtain array references from the source code

Our FS model identifies FS caused by only array references made in the innermost loop. The details about each array reference obtained from the compiler such as array base name, indices, types of access (read/write) are then stored in an array reference list. However, more specific information such as which thread will access which region/elements of an array will be generated automatically in the next step of the model.

B. Generate a cache line ownership list

In step 2, our cost model generates a cache line ownership list for each thread. At each iteration of a chunk of a loop the cache line ownership list is generated using the array reference list and the new values of loop indices. The cache line ownership list contains information about which cache line is being read/written by a thread at that specific iteration. The assumption we make at this step is that all array variables are aligned with the cache line boundary, so that it would be possible to know the relative cache lines on which array elements are located at compile-time.

C. Apply a stack distance analysis

In step 3, our cost model creates a separate *cache state* for each thread. The cache states store the current state of private caches that threads operate upon. When a new cache line ownership list is generated for each thread (in the previous step), we update the cache states with new cache line ownership lists and apply a stack distance analysis on cache states. The stack distance [19] is the number of distinct cache lines accessed between two accesses of the same cache line. The distance of the stack is the number of cache lines for a fully associative cache or number of lines in one set for a set associative cache. Basically, the stack distance analysis simulates the least recently used (LRU) cache and outputs the state of the cache at each distinct point of time. Before one element of the cache line ownership list is inserted into the cache state, the analysis checks whether the cache line already exists in the cache state. If so, the analysis moves the position of the existing cache line to the top of the stack; otherwise, it simply inserts the cache line at the top of the stack. If the number of distinct cache lines exceeds the stack size, the analysis evicts the cache line placed on the bottom of the stack, which is the LRU cache line. In this way, by using the stack distance analysis our cost model simulates the fully associative cache. We do not simulate the set associative caches due to two reasons:

1) It is impossible to know on which corresponding line in a set the cache line with a specific array reference will be placed at compile-time.
2) Modeling the fully associative cache is mostly valid especially for caches with a high level of associativity [18].

When the cache line is inserted into the cache state, our FS cost model proceeds to the next step which is to determine the number of FS cases that occured at that specific iteration.

D. Detect false sharing

In the final step, our model determines whether FS happens by performing a *1-to-All* comparison between the newly

inserted cache line and the cache lines that other cache states already contain. For that purpose assume the following:

$$\begin{aligned} &cs_k \in S \\ &cl_i \in CLOL_k, k = 0, 1, 2, .., num\_of\_threads \end{aligned} \tag{2}$$

where cs_k is a cache state of thread k; S is a set of all cache states; $CLOL_k$ is a cache line ownership list of thread k; and cl_i is a cache line element of $CLOL_k$. We define a function $\varphi(cs_k, cl_i)$ as

$$\varphi(cs_k, cl_i) = \begin{cases} 1, if(cl_i \in cs_k \ \ and \ \ cs_k^{cl_i} = W) \\ 0, otherwise \end{cases} \tag{3}$$

which returns 1 if a cache state cs_k includes cache line cl_i and the state of the cache line in the cache state - $cs_k^{cl_i}$ is modified, otherwise the function returns 0. Using the function $\varphi(cs_k, cl_i)$ our model determines whether FS happened or not. In order to compute the number of FS cases that occurred during one iteration, the model needs to execute the function until all cache lines in all threads' $CLOL_k$ lists are evaluated. Thus the number of FS cases occuring at one iteration can be determined using:

$$\begin{aligned} false\_sharing_{iter} &= \sum_{j=0}^{k-1} \sum_{i=0}^{n} \varphi(cs_j, cl_i) \times mask(cs_j, cl_i) \\ mask(cs_j, cl_i) &= \begin{cases} 0, if(\mathrm{cl}_i \in \mathrm{CLOL}_j^{cs_j}) \\ 1, \mathrm{otherwise} \end{cases} \end{aligned} \tag{4}$$

The $mask(cs_j, cl_i)$ function ensures that the newly inserted cache line is not compared with the cache lines of the same cache state.

In order to estimate the total number of FS cases that occurred throughout the whole loop, our model needs to evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ number of iterations where for each iteration it needs to perform the steps 2-4 and store the FS cases estimated at that iteration.

E. Prediction of false sharing using Linear Regression model

When the number of iterations of a loop is large, our FS model might take quite a long time to estimate the total number of FS cases. This is because the model needs to evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ number of iterations.

To overcome this limitation, we propose a FS prediction model that predicts the total number of FS cases by evaluating fewer iterations in much less time. The prediction model uses a *Linear Regression* model [3]. Figure 6 shows that the estimated number of FS cases increases linearly when different *chunk runs* are evaluated, where a single chunk run refers to $chunk_{size} \times num\_of\_threads$ iterations. Since the relation between chunk runs and the estimated FS cases is linear, the *Linear Regression* model is suitable for use in our cost model.

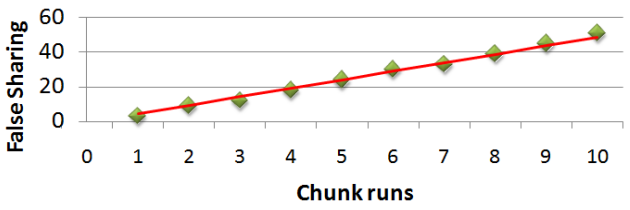

Figure 6. The false sharing cases increase linearly with the number of chunk runs, where one chunk run is number of iterations equal to the product of chunk size with the number of threads.

The prediction of the total number of FS cases using the *Linear Regression model* is as follows:

- We denote n iterations already evaluated by the model as $\vec{x} = \{x_1, ..x_n\}$; and estimated FS cases in n iterations as $\vec{y} = \{y_1, ..y_n\}$.
- The FS prediction can be modeled via $\vec{y} = a\vec{x} + b$ where initial $\vec{y}$ and $\vec{x}$ are known.
- We want our predicted results to be very close to the estimated results i.e. the error between predicted and estimated be minimal. The *Least Square Solution* therefore, suggests the function to be $f = \left\| a\vec{x} + b - \vec{y} \right\|_2 = (a\vec{x} + b - \vec{y})^T (a\vec{x} + b - \vec{y})$.
- We have to find a and b such that the result of the function f is minimal which is $a, b = \arg\min_{a,b} f(a, b) = (a\vec{x} + b - \vec{y})^T (a\vec{x} + b - \vec{y})$.
- Thus we differentiate the function f with respect to a and b and obtain $a = \sum_{i=0}^{n-1} x_i y_i / \sum_{i=0}^{n-1} (x_i)^2$, $b = \sum_{i=0}^{n-1} y_i - \frac{a}{n} \sum_{i=0}^{n-1} x_i$.
- After we compute a and b, we can predict $y_{\max}$, which is the total number of FS cases, using $y_{\max} = a x_{\max} + b$ where $x_{\max}$ is represented either as the maximum number of iterations or chunk runs of a loop.

IV. Evaluations

The false sharing cost model was implemented within the Open64 compiler's LNO phase, as shown in Figure 7. The Open64 compiler parses the source code in C/C++/Fortran programs, and generates a WHIRL tree - an intermediate representation (IR) of the source code with 4 different levels. Figure 7 shows that the compiler uses High-Level IR during LNO phase. We implemented a separate compiler pass that is applied to the IR to collect information needed to perform the modeling, including the information about the parallel OpenMP loop nest as well as memory loads/stores performed in the body of the innermost loop. The details about the loop nest include loop boundaries, step sizes, loop index variables and the chunk size, if specified, for OpenMP loops. Memory load/store information is collected by traversing the IR and obtaining array reference details such as array

base name, array index, and memory offsets for arrays storing structured data types. The analysis we implemented did not require any modification to the compiler's IR, and the information about loop nest and memory operations are generally available in most compiler IRs. Thus we believe that our cost model can be implemented in other compilers with a similar approach.

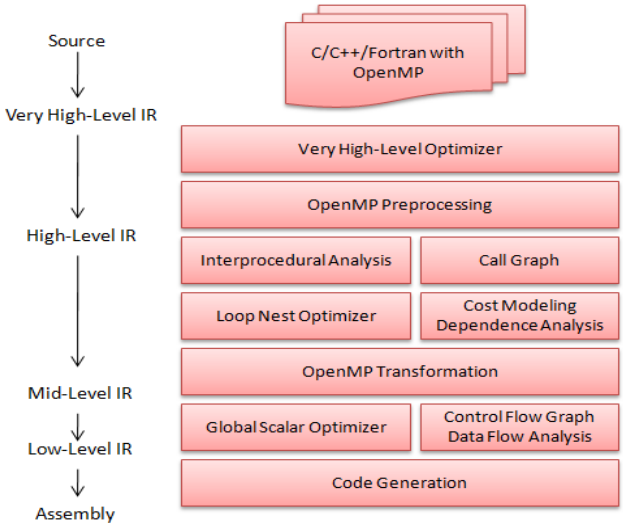

Figure 7. The components of the Open64 compiler.

A. Methodology for Accuracy and Efficiency Evaluation

Our FS cost model is evaluated in terms of both accuracy and efficiency. To demonstrate the accuracy, we compared the percentages of measured and modeled FS overhead costs on the total loop execution time. We expect that the measured percentage of FS overhead should be close to the percentage of FS overhead modeled by our FS cost model as follows:

$$\frac{T_{fs\_measure} - T_{nfs\_measure}}{T_{fs\_measure}} \approx \frac{N_{fs\_model} - N_{nfs\_model}}{N^*_{fs\_model}} \quad (5)$$

where $T_{fs\_measure}$ is the measured time needed to execute a loop exhibiting FS; $T_{nfs\_measure}$ is the measured time needed to execute the same, but optimized, loop that does not incur any FS; N_{fs_model} is the number of FS cases estimated by our model on a loop exhibiting FS; N_{nfs_model} is the number of FS cases estimated by our model on an optimized loop; $N^*_{fs\_model}$ is a normalized value for FS cases estimated by our model on a loop incurring FS.

The measured FS overhead cost is obtained by executing a kernel loop with two different chunk sizes and calculating the percentage using:

$$\frac{T_{fs\_measure} - T_{nfs\_measure}}{T_{fs\_measure}}$$

Two different chunk sizes represent a loop with FS and non-FS cases. For example, for one kernel in our experiments we are using chunk sizes 1 and 64, where a loop with chunk size=1 represents a loop with FS case, and a loop with chunk size=64 refers to a loop with non-FS case. A loop with FS case heavily suffers from FS, whereas a loop with non-FS case incurs less (or no) false sharing because we believe that increasing the chunk size decreases the FS. The execution times of the loop with FS and non-FS cases are represented by $T_{fs\_measure}$ and $T_{nfs\_measure}$, respectively.

The modeled FS overhead cost is obtained by our FS cost model, which evaluates $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations of a loop to compute the total number of FS cases for both FS and non-FS case loops. The computed percentage value is then calculated using:

$$\frac{N_{fs\_model} - N_{nfs\_model}}{N^*_{fs\_model}}$$

To demonstrate the efficiency of our FS prediction model, which uses the Linear Regression model, we compare the predicted amount of FS cases when a very small number of iterations are evaluated against the modeled total amount of FS cases when $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations are evaluated. The smaller the difference between the predicted and the modeled values, the more efficient our prediction model is.

B. Experimental Results

Our experiments are conducted on a system with four 2.2 GHz 12-core processors (48 cores in total). Each core has dedicated L1 and L2 caches of 64KB and 512KB respectively; L3 cache of 10240KB size is shared among 12 cores. All the caches at the three levels have the same cache line size, 64 bytes, which satisfies the assumption in our cost model.

We have used OpenMP versions of *heat diffusion* [2], *discrete fourier transform (DFT)* [1] and *linear regression* [17] programs for our experiments. The computation-intensive loop kernels of these programs, when parallelized with OpenMP, exhibit extensive data accesses to the boundary of each data segment that is processed by each OpenMP thread. These accesses could incur large occurence of FS during program execution if parameters such as the chunk size are not properly chosen when parallelizing the code.

Tables I, II and III show the results of our model compared to the measured FS effect. The measured execution times of the *heat diffusion* kernel with FS and non-FS cases are depicted in the second and third columns of Table I, respectively. Using the execution time information, the measured FS effect (on the total execution time of the kernel) is calculated and shown in the fourth column of the table. The last column, on the other hand, shows the FS effect modeled by our cost model. The accuracy of the model is assessed by comparing the fourth and fifth columns against each other.

Table I
COMPARISON OF % OF FALSE SHARING OVERHEADS INCURRED IN HEAT DIFFUSION KERNEL

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=64 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	0.3593	0.2901	19.2%	6.9%
4	0.2263	0.1646	27.2%	6.9%
8	0.1639	0.156	4.8%	6.9%
16	0.6586	0.6205	5.7%	7.0%
24	1.0049	0.9564	4.8%	7.1%
32	1.4671	1.3608	7.2%	7.2%
40	1.8455	1.6130	12.5%	7.2%
48	2.247	2.1501	4.3%	7.2%

Table II
COMPARISON OF % OF FALSE SHARING OVERHEADS INCURRED IN DFT KERNEL

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=16 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	2.0978	1.7624	15.9%	32.0%
4	1.762	0.9618	45.4%	31.6%
8	0.8976	0.6033	32.7%	31.5%
16	0.599	0.3688	38.4%	33.2%
24	0.5041	0.3163	37.2%	32.8%
32	0.4727	0.2827	40.1%	35.6%
40	0.4792	0.2669	44.3%	36.7%
48	0.4664	0.279	40.1%	35.8%

Table III
COMPARISON OF % OF FALSE SHARING OVERHEADS INCURRED IN LINEAR REGRESSION KERNEL

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=10 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	0.4302	0.4135	3.8%	16.1%
4	0.118	0.1074	9.0%	14.7%
8	0.0421	0.0331	21.2%	9.0%
16	0.02	0.0182	8.8%	4.9%
24	0.0116	0.01	13.9%	3.3%
32	0.0079	0.0068	13.4%	2.5%
40	0.0062	0.0051	18.0%	2.0%
48	0.0055	0.0046	15.6%	1.7%

The closer the values in both columns are, the more accurate our cost model is.

An important point worth mentioning here is that loop kernels in *heat diffusion* and *DFT* programs are parallelized at the innermost loop level, while the loop kernel in *linear regression* program is parallelized at the outermost loop level. Results for *heat diffusion* and *DFT* kernels given in Tables I and II, show that the modeled FS overhead percentages estimated by our cost model are close to the measured FS overheads, indicating that by modeling FS, we can accurately estimate the FS overhead cost at compile-time. However, due to the difference in loop parallelization style, we see that the modeled and the measured FS overhead percentage results for the *linear regression* kernel depicted in Table III are not close to each other. Moreover, one can observe that the modeled FS cases effect decreases proportionally with increasing number of threads. This is due to the fact that the total number of chunk runs that threads will execute in *linear regression* kernel is

$$x_{\max} = m/(num\_of\_threads * chunk\_size)$$

whereas in *heat diffusion* and *DFT* is

$$x_{\max} = (m * n)/(num\_of\_threads * chunk\_size)$$

where m and n are the upper bounds of the outer and inner loops, respectively. Thus, in the loop kernel of the *linear regression* program, the number of chunk runs and consequently the total number of FS cases are directly dependent on the number of threads.

Results in Tables IV, V and VI show the comparison between FS effects obtained from both the FS model and the FS prediction model, which uses the Linear Regression model. When modeling the effects with predictions, the number of FS cases is estimated with fewer iterations. For example, when the *heat diffusion* kernel is executed with 8 threads, with chunk run=20 and chunk sizes 1 and 64, our prediction model evaluates 8*1*20 and 8*64*20 iterations,

Table IV
COMPARISON OF PREDICTED VS. MODELED FALSE SHARING CASES AND THEIR OVERHEAD %'S IN HEAT DIFFUSION KERNEL

# of threads	Pred. # of FS cases chunk size=1 (chunk run=20)	Pred. # of FS cases chunk size=64 (chunk run=20)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=64	Modeled FS cases effect
2	91,991K	1,595K	6.8%	94,421K	2,107K	6.9%
4	92,979K	1,625K	6.8%	94,446K	2,145K	6.9%
8	93,496K	1,702K	6.8%	94,458K	2,070K	6.9%
16	93,990K	1,724K	6.9%	96,043K	1,888K	7.0%
24	94,155K	1,609K	6.9%	96,938K	1,699K	7.1%
32	93,986K	1,456K	6.9%	97,159K	1,509K	7.2%
40	94,286K	1,826K	6.9%	97,730K	1,889K	7.2%
48	94,319K	1,107K	7.0%	97,935K	1,126K	7.2%

Table V
COMPARISON OF PREDICTED VS. MODELED FALSE SHARING CASES AND THEIR OVERHEAD %'S IN DFT KERNEL

# of threads	Pred. # of FS cases chunk size=1 (chunk run=50)	Pred. # of FS cases chunk size=16 (chunk run=50)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=16	Modeled FS cases effect
2	52,233K	26,468K	32.4%	53,058K	27,358K	32.0%
4	52,697K	26,491K	32.8%	53,088K	27,702K	31.6%
8	52,928K	26,612K	32.8%	53,311K	27,882K	31.5%
16	52,936K	26,526K	32.9%	54,411K	27,257K	33.2%
24	52,967K	27,475K	31.8%	54,956K	28,003K	32.8%
32	52,983K	25,523K	34.2%	55,245K	25,865K	35.6%
40	53,077K	24,895K	35.1%	55,510K	25,154K	36.7%
48	52,998K	25,649K	34.1%	55,542K	25,878K	35.8%

Table VI
COMPARISON OF PREDICTED VS. MODELED FALSE SHARING CASES AND THEIR OVERHEAD %'S IN LINEAR REGRESSION KERNEL

# of threads	Pred. # of FS cases chunk size=1 (chunk run=10)	Pred. # of FS cases chunk size=10 (chunk run=10)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=10	Modeled FS cases effect
2	85,592K	703	16.0%	86,315K	719	16.1%
4	77,561K	720	14.7%	77,685K	678	14.7%
8	47,473K	840	9.5%	44,545K	791	9.0%
16	24,804K	900	5.2%	23,274K	855	4.9%
24	16,778K	920	3.6%	15,771K	874	3.3%
32	12,667K	930	2.7%	11,907K	899	2.5%
40	10,172K	936	2.2%	9,579K	897	2.0%
48	8,497K	940	1.8%	7,987K	893	1.7%

respectively. On the other hand, without the prediction model, our FS model would evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations. Thus, if there were 5000*5000 iterations in total, our model would need to evaluate 3,125,000 iterations to compute the total number of FS cases. However, with the FS prediction model we would need to evaluate only 160 or 10,240 iterations, for chunk sizes 1 and 64 respectively. Our prediction model clearly facilitates the process of estimating the number of FS cases. The percentage results of predicted and modeled FS impacts depicted in tables IV, V and VI are very close to each other, indicating that our FS prediction model is accurate and efficient.

As a summary, we have included Figures 8 and 9 that show the false sharing effects (percentage of execution time) obtained through the execution measurement, the compile-time modeling, and the modeling using Linear Regression predictions. These results demonstrate the effectiveness of the model. Our model, when combined with compiler machine models, could be used by compilers to assist in optimizing code in both high-level loop transformation, and low-level instruction scheduling and code generation. For example, it will be helpful for both programmers and compilers to choose the optimal chunk size for OpenMP loops and the optimal number of threads to execute the loop. It could be used to guide a compiler when performing traditional loop transformations to decide parameters suitable for executing parallel loops on multicore architecture.

V. RELATED WORK

Several different approaches have been taken in the area of detecting FS in a program. One approach is via cache simulation and memory tracing [14], [16], [13], [11], [8]. In this approach, compiler instruments the binary code with tracing routines, and a tracing tool then captures the memory accesses performed at runtime and stores the

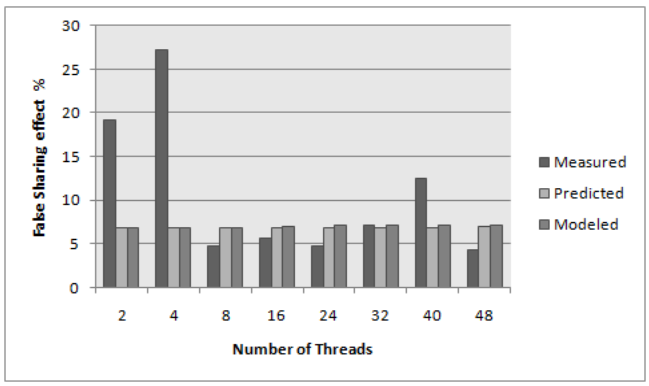

Figure 8. Comparison of %'s false sharing effects vs. different number of threads for *heat diffusion* kernel.

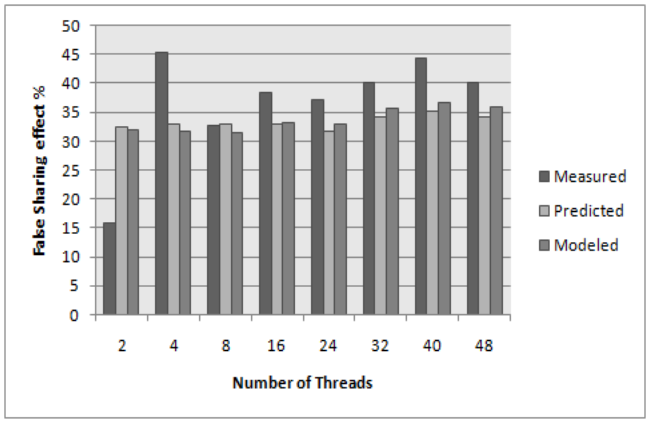

Figure 9. Comparison of %'s false sharing effects vs. different number of threads for *DFT* kernel.

tracing information for offline simulation. The tracing file is fed to a simulation tool which in turn simulates caches in the modeled architecture and determines the types and degree of cache misses that occurred during the simulation. However, tracing every memory reference performed during the execution of a program can degrade the overall execution performance greatly.

Analyzing performance of a program using hardware performance counters is another approach taken to detect false sharing in the code [21], [9], [15]. Performance counters are used to detect the major bottleneck in the program and to identify reasons for the bottleneck. The main drawback of this approach is that the programmer has to understand the results obtained from the performance counters and manually identify the source of the problem in the code.

FS detection techniques implemented in [20] and [23] differ greatly from previous approaches. The technique in [20] write-protects the shared data; when a process attempts to write to the shared data, a page fault occurs, the tool copies the page and process accesses the local copy of the page. The tool detects FS by observing every word modified in local copies of pages that processes access. The other approach uses a memory shadowing technique to track the cache access patterns [23].

The FS detection techniques discussed above are applied at runtime and incur some amount of overhead, while the approach taken in our paper is based on compile-time analysis of a loop nest and does not cause any performance degradation in program execution.

There are also reseach efforts to provide techniques for eliminating FS [10], [7]. In [7], the authors show how to mitigate FS by changing the scheduling parameters such as chunk size and chunk stride for parallel loops. In [10] several compiler transformations are described such as array padding and memory alignment that optimize the layout of data and decrease the possibility of FS to occur. In this work, we focus on the FS detection, and the FS elimination using the cost model presented in this paper will be studied as our future work.

VI. Conclusions

In this paper, we described our compile-time cost model for FS and discussed how to use the defined model to detect and estimate the performance impact of FS on parallel loops. The model estimates the total number of FS cases that occur throughout the loop, and computes the overhead cost incurred by FS to the whole execution of the loop. Moreover, we describe our FS prediction model that predicts the total FS cases by evaluating much fewer number of iterations, for the purpose of reducing the modeling time.

To evaluate our FS cost and prediction models, we experimented with loop kernels that exhibit FS. We compared the percentages of measured FS overhead costs (from the execution time of loops) with the estimated overhead costs (by our cost and prediction models). Our experimental results are promising, and we believe that by modeling FS, we can accurately estimate the FS overhead cost at compile-time.

Our ongoing and future work includes modeling more complex loop structures to refine the model. We will also add other cache contention issues in the model such as shared cache and bus interferences.

VII. Acknowledgement

The authors thank their colleagues in the HPCTools group for providing useful feedback during the writing of the paper. This work is supported by the National Science Foundation under grants CCF-0833201 and CCF-0702775.

References

[1] Discrete fourier transform. http://mathworld.wolfram.com/DiscreteFourierTransform.html. Last accessed 29-February-2012.

[2] Heat diffusion equation. http://ccl.northwestern.edu/papers/ABMVisualizationGuidelines/palette/examples/Heat%20Difussion/. Last accessed 29-February-2012.

[3] Linear regression analysis. http://www.weibull.com/DOEWeb/simple_linear_regression_analysis.htm. Last accessed 29-February-2012.

[4] The Open64 compiler. http://www.open64.net/. Last accessed 29-February-2012.

[5] OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org, 2006.

[6] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993.

[7] J.-H. Chow and V. Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *Proceedings of the 26th International Conference on Parallel Processing*, pages 396–403, 1997.

[8] S. M. Günther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, pages 26–33, 2009.

[9] Intel. Avoiding and identifying false sharing among threads. http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/, 2011.

[10] T. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 1994.

[11] R. P. LaRowe, C. S. Ellis, and V. Khera. An architecture-independent analysis of false sharing, 1993.

[12] C. Liao and B. M. Chapman. Invited paper: A compile-time cost model for OpenMP. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[13] C. Liu. False sharing analysis for multithreaded programs, 2009.

[14] J. Marathe and F. Mueller. Source code correlated cache coherence characterization of OpenMP benchmarks. *IEEE Trans. Parallel Distrib. Syst.*, 18:818–834, June 2007.

[15] J. Marathe, F. Mueller, and N. Carolina. Analysis of cache coherence bottlenecks with hybrid hardware/software techniques, 2006.

[16] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[18] A. Sandberg, D. Eklov, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[19] D. Schuff, B. Parsons, and V. Pai. Multicore-aware reuse distance analysis. In *IPDPS Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems*, 2010.

[20] L. Tongping and E. Berger. Precise detection and automatic mitigation of false sharing. In *Proceedings of ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA/SPLASH)*, 2011.

[21] B. Wicaksono, M. Tolubaeva, and B. Chapman. Detecting false sharing in OpenMP applications using the DARWIN framework. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing*, 2011.

[22] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 274–286, 1996.

[23] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 27–38, 2011.