Detection of False Sharing Using Machine Learning

Sanath Jayasena*, Saman Amarasinghe**, Asanka Abeyweera*, Gayashan Amarasinghe*, Himeshi De Silva*, Sunimal Rathnayake*, Xiaoqiao Meng[#], Yanbin Liu[#]
*Dept of Computer Science & Engineering, University of Moratuwa, Sri Lanka
**Computer Science & Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, USA
[#]IBM Research, Yorktown Heights, New York, USA
sanath@cse.mrt.ac.lk, saman@csail.mit.edu, asanka.09@cse.mrt.ac.lk, gayashan.09@cse.mrt.ac.lk, himeshi.09@cse.mrt.ac.lk, sunimalr.09@cse.mrt.ac.lk, xmeng@us.ibm.com, ygliu@us.ibm.com

ABSTRACT
False sharing is a major class of performance bugs in parallel applications. Detecting false sharing is difficult as it does not change the program semantics. We introduce an efficient and effective approach for detecting false sharing based on machine learning.

We develop a set of mini-programs in which false sharing can be turned on and off. We then run the mini-programs both with and without false sharing, collect a set of hardware performance event counts and use the collected data to train a classifier. We can use the trained classifier to analyze data from arbitrary programs for detection of false sharing.

Experiments with the PARSEC and Phoenix benchmarks show that our approach is indeed effective. We detect published false sharing regions in the benchmarks with zero false positives. Our performance penalty is less than 2%. Thus, we believe that this is an effective and practical method for detecting false sharing.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Measurement techniques, Modeling techniques, Performance attributes.

General Terms
Measurement, Performance.

Keywords
False Sharing, Performance Events, Machine Learning.

1. INTRODUCTION
Parallelism has become the major source of application performance in the era of multicore processors. As more applications rely on parallelism, performance issues related to parallel execution are becoming a major problem for the developers. One such issue is false sharing.

False sharing is a major class of performance bugs in parallel applications. It occurs when threads running on different processors/cores with local caches modify unshared data that happen to occupy (share) the same cache line. The performance penalty due to false sharing could be significant and can severely hinder achieving the expected speedup in a parallel application.

False sharing does not change program semantics and is hard to detect. Current detection methods are expensive, thus, are not generally used in practice. Unlike a true sharing issue, which is associated with a real data movement in the application, false sharing is not visible within the application. Two variables that can cause false sharing are completely independent. The fact they share the same cache line may be result of the data layout driven by the compiler or the runtime system. Thus, application analysis will not reveal any false sharing. Furthermore, false sharing cannot be revealed by localized analysis within a single core as it requires multiple cores, and each to access different part of the cache line. Thus, there are no simple localized hardware mechanisms such as performance counters to detect false sharing.

To understand the performance impact of false sharing, consider the example in Figure 1. Given vectors v1 and v2, it shows 3 different functions for the parallel dot-product computation.

```
int psum[MAXTHREADS]; // shared by threads
int v1[N], v2[N];

void *pdot_1(...) // Method 1: Good
{ ...
  int   mysum = 0;
  for (i = start; i < end; i++)
        mysum += v1[i] * v2[i];
  psum[myid] = mysum;
}
void *pdot_2(...) // Method 2: Bad -
{ ...             // False sharing
  for (i = start; i < end; i++)
        psum[myid] += v1[i] * v2[i];
}
void *pdot_3(...) // Method 3: Bad
{ ...             // Memory access
  // same as pdot_1() except non-sequential
  // vector element access (e.g, strided)
}
```

Figure 1: Code for parallel computation of dot-product

For a parallel dot-product computation on a multicore system using code in Figure 1, the main program will create multiple threads, each to run on a distinct core executing one of the `pdot_N( )` functions to complete its part of the computation, which is identified by `start` and `end`, i.e., the regions of the vectors v1 and v2 assigned to each thread. If all threads use Method 2, the repeated writes to `psum[myid]` in the loop by

SC13, November 17-21, 2013, Denver, CO, USA.

http://dx.doi.org/10.1145/2503210.2503269

each thread could lead to repeated false sharing misses, because some or all elements in `psum[ ]` array would share a cache line. False sharing in this case can be avoided by using Method 1which uses a thread-private variable `mysum` in the loop. Method 3, for comparison, demonstrates another form of bad memory access, when elements in v1 and v2 are accessed non-sequentially (e.g., randomly or strided) between iterations, thus causing cache misses. For a vector size of $N=10^8$ and on a 32-core Intel Xeon system, Table 1shows the program execution times.

Table 1: Execution time in seconds for programs using the code in Figure 1 on a 32-core Intel Xeon system for $N=10^8$

	# Threads				
Method Used	**1**	**4**	**8**	**12**	**16**
1: Good	44.1	11.5	6.2	4.5	3.7
2: Bad, false sharing	44.0	79.3	76.8	76.1	78.0
3: Bad, memory access	250	82.8	77.1	77.3	78.2

Table 1 shows false sharing has a drastic performance impact; the multi-threaded versions with false sharing are even slower than the single-threaded version.

The recent false sharing detection techniques [21][33] have mainly relied on tracing the data movement across multiple cores. This requires heavy instrumentation and excessive data gathering and analysis. Thus, they incur a high overhead. They also require special libraries which limit their applicability.

In this work, we take a completely different approach to false sharing detection. Instead of trying to directly identify false sharing, we are looking for the telltale signature created by false sharing. False sharing induces certain memory access patterns on multiple cores. By looking at the combined performance event counts of the cores, we are able to detect the pattern. We use supervised learning to train a classifier with a set of sample kernels (mini-programs) – with and without false sharing. The trained classifier is then used to analyze memory access patterns of other programs.

We have applied our trained classifier to the PARSEC and Phoenix benchmark programs, some of which have false sharing identified and validated previously [21][33]. Our light-weight technique is as successful as the previous heavy weight methods in identifying the false sharing programs. First, our classifier did not incorrectly classify any program as having false sharing, thus zero false positives. In 97.8% cases, our classifier correctly identified the problem confirmed by other heavyweight methods. However, our instrumentation cost is minimal where the performance penalty is less than 2%. Further, our approach is easy to apply and does not require specialized tools or access to the source code. Thus, we believe that this is an effective and practical method for detecting false sharing.

The key contributions of this paper are:

- a novel, practical and effective methodology to detect false sharing based on performance event counts and machine learning
- demonstrating the methodology for using a specialized set of small programs, in this case a set of multi-threaded mini-programs in which false sharing can be turned on and off, for training a machine-classifier that can be successfully applied to larger applications
- experimental results with PARSEC and Phoenix benchmarks show that we detect all cases with zero false positives and our results are verified by existing methods.

This paper is organized as follows. Section 2 gives an overview of our methodology and describes the set of mini-programs and the performance events. Section 3 describes the training data and the training of the machine classifier. Section 4 presents results on our detection of false sharing in PARSEC and Phoenix benchmarks. Section 5 discusses related work and Section 6 concludes the paper.

2. OUR APPROACH

2.1 Overview

In our approach, we rely on hardware performance event counts collected from running programs. Performance monitoring units (PMUs) in processors can count many hardware events [16][20] and one could easily collect the desired counts via APIs (e.g., libpfm, PAPI) or tools (e.g., perf, Intel PTU)[6] [9][10][17][28]. Hardware performance event counts can help understand how the hardware is being used by a program and potentially provide hints on performance issues in a program.

Yet raw performance event data are difficult to handle and confusing due to lack of standards among processors, poor documentation and being tightly coupled with the system architecture. For our purpose, having studied the events of Intel micro-architectures Nehalem, Westmere and Sandy Bridge [16], there is no single event, or even a small subset of events, that can indicate the presence of false sharing. Further, while it is possible to collect different kinds of performance event counts (and large amounts of them) from running programs, such data are too overwhelming for human processing.

Thus we rely on machine-learning techniques for the analysis of performance event data.

The basic idea in our approach is to train a machine classifier with a set of relevant performance event counts collected from a set of mini-programs (or *problem-specific programs*), while running each with and without false sharing, by turning false sharing on and off as desired. However, false sharing is one of the many memory system issues that can affect performance. In order to be able to distinguish false sharing from these potential problems related to memory access, we also included an inefficient form of memory access in these programs whenever possible. This way, a mini-program can be classified into three possible modes of operation:

- good, i.e., no false sharing, no bad memory access
- bad, with false sharing ("bad-fs")
- bad, with inefficient memory access ("bad-ma").

These modes make our problem a three-way classification instead of a binary classification, which would result if only (the presence or absence of) false sharing was considered.

The steps in our methodology would be as follows:

1. Design and develop a set of representative mini-programs that can run in any one of the three possible modes
2. Identify a set of relevant performance events for the underlying hardware with the help of the mini-programs

3. Collect performance event counts by running the mini-programs in all possible modes
4. Label each instance of the collected performance event data as "good", "bad-fs" or "bad-ma" based on the mode to which the instance corresponds
5. Train a classifier using the labeled data set as training data
6. Use the classifier on programs previously unseen by the classifier and evaluate its performance.

The expectation is that the trained classifier will take as input new data instances (i.e., counts of the same set of performance events) from unseen arbitrary programs and classify them correctly, informing us if false sharing is present or not.

Our approach does not require access to the source code of a program and can be widely applied across different hardware/OS platforms as long as performance event counts can be collected.

Our methodology of steps 1-6 above is general and can be adapted in different ways. For example, one could iterate through steps 1-6 a few times, adding new min-programs in step 1 in each iteration and thereby gradually improving the classification accuracy, until desired level is reached. With an existing set of mini-programs, we can apply our approach to a new hardware platform with the workflow being steps 2-6 above; here, steps 2-6 can be iterated a few times by selecting different performance events in step 2 in each iteration, until a satisfactory level of accuracy is reached. One could also try out different machine classifiers in step 5 and select the most suitable.

2.2 Mini-programs for Training

Selecting the set of mini-programs is crucial for building an effective classifier. In training our classifier we used two sets of mini-programs.

2.2.1 Multi-threaded Program Set

The first set of programs is multi-threaded (using pthreads) and can be summarized as follows, with the names of programs within parenthesis:

- 3 "scalar" programs (psums, padding, false1): each thread processes its own share of scalar data
- 3 "vector" programs (psumv, pdot, count): each thread processes its own share of vector data (e.g., Figure 1 shows a part of the pdot program)
- matrix multiplication (pmatmult); each thread computes its share of elements in the final matrix
- matrix compare (pmatcompare); each thread compares a share of element pairs of two matrices

Each of the 3 scalar programs is different in what it does, the amount of memory used and the way memory is accessed. Similar differences exist among the 3 vector programs. In all programs, each thread repeatedly writes to its own variable; there is false sharing when these variables happen to share a cache line. The vector programs are also parameterized to have a "bad-ma" mode, introduced via inefficient access to array elements.

In each program, we parameterize the size of the computation (problem size), the number of threads and the memory-access mode. This means, for a specific problem size and a thread number, we have a "good" version, a "bad-fs" version with false sharing and a "bad-ma" version with bad-memory-access.

2.2.2 Sequential Program Set

We used a second set of mini-programs that are sequential (single-threaded) to have more training data and improve the training on "bad-ma" mode. This indeed improved the classification accuracy.

These programs exercise the memory system in different ways, so that the overall program performance between "good" and "bad-ma" modes differ significantly due to memory access pattern alone (i.e., due to cache misses).

We have 3 programs in this set as follows:

- read data element-wise from an array
- write data element-wise to an array
- read data element-wise from an array, modify the data and write it back

In each of the above programs, we parameterize the size of array and the access pattern. There are 3 types of access to array elements: (i) linear (sequential order, as stored in memory), (ii) random, and (iii) in strides (stride can be varied).

The idea is that, linear access would result in good memory access performance ("good" mode) while random and strided-access would result in lots of cache misses ("bad-ma" mode). We have another program that performs two-dimensional matrix multiplication using different memory access patterns and loop structures.

For each program above, the different versions perform the same computation, the only difference being the way the data in memory are accessed; "good" memory access results in significantly better program performance than "bad-ma".

2.3 Identification of Performance Events

We first go through the available list of performance events for the hardware platform (this can be a couple of hundreds) and compile a candidate list. In our case, since we focus on false sharing and data access in memory, events that correspond to memory access (loads and stores), data caches (e.g., cache line state, cache misses), TLBs, interaction among processor cores, resource stalls are included in the candidate list. The number of instructions is also included. On Intel Nehalem EX and Westmere DP micro-architectures, for example, we had about 60-70 candidate events [16][20].

We next use the mini-programs to identify a set of relevant events from the candidate list, in two steps, as follows.

First, for each candidate event, we run each of our multi-threaded mini-programs in "good" and "bad-fs" modes, with different numbers of threads (e.g., 3, 6, 9, 12 on a 12-core system) and note the event counts. If there is significant enough difference in the counts between "good" and "bad-fs" cases (we used minimum 2x ratio as a heuristic) for a majority of the mini-programs, then we select that event as a relevant one from the candidate list, because it can help to distinguish "good" and "bad-fs" cases. Second, for each of the remaining (unselected) candidate events, we run each of our mini-programs in "good" and "bad-ma" modes and select the event as relevant, as was done before, if it can help distinguish between "good" and "bad-ma" cases.

In our experiments we noted that some event counts associated with L1D caches can be noisy and inconsistent, confirming the caution in [20]. Further a candidate event like `Memory_Uncore_Retired.Other_core_L2_HITM` did not end up in the final set, despite our expectations based on information in [15][16][20]. Table 2 shows the identified set of relevant performance events for the Intel Westmere DP platform.

Table 2: Selected performance events for Westmere DP

Event #	Event Code	Umask Code	Description
1	26	01	L2_Data_Requests.Demand."I" state
2	27	02	L2_Write.RFO."S" state
3	24	02	L2_Requests.LD_MISS
4	A2	08	Resource_Stalls.Store
5	B0	01	Offcore_Requests.Demand_RD_Data
6	F0	20	L2_Transactions.FILL
7	F1	02	L2_Lines_In."S" state
8	F2	01	L2_Lines_Out.Demand_Clean
9	B8	01	Snoop_Response.HIT
10	B8	02	Snoop_Response.HIT "E"
11	B8	04	Snoop_Response.HIT "M"
12	CB	40	Mem_Load_Retd.HIT_LFB
13	49	01	DTLB_Misses
14	51	01	L1D-Cache_Replacements
15	A2	02	Resource_Stalls.Loads
16	C0	00	Instructions_Retired

The first 15 events in Table 2 were selected based on the process described above. The last event, Instructions_Retired, was added to allow us to normalize all other event counts by dividing each of them by it. Such normalized counts of the first 15 events from one program are comparable to corresponding normalized counts from another program, whereas the absolute counts are not.

While having a large set of relevant performance events is potentially desirable from a machine learning point of view, a small set is desirable due to the constraints (e.g., limited number of hardware registers in PMUs) that affect the accuracy of counts. In our experience, the set in Table 2 is a reasonable balance for the Intel Westmere DP platform and the problem we address.

3. TRAINING A MACHINE CLASSIFIER

After experimenting with several classifiers available in the public domain, we selected J48 in Weka [13], an implementation of the C4.5 decision-tree classification algorithm [23], as it produced the best classification results. Our experimental platform is a 12-core (2x6-cores) Intel Xeon X5690, 3.4GHz (Westmere DP) system that has 32KB/core L1-D and L1-I caches, 256KB/core L2 cache, 12MB/CPU L3 cache and 192GB (96GBx2) RAM and running x86_64 GNU/Linux 2.6.32.

3.1 Collection of Training Data

Training data are the counts of the selected performance events collected by running the mini-programs. They were collected in two parts, as follows. Table 3 summarizes the result of data collection.

Part A consists of training data collected from the multi-threaded mini-programs. Each mini-program is run for multiple problem sizes, each with a few different thread numbers and in all 3 modes. This way, we collected an initial set of 675 data instances (324 good, 216 bad-fs, 135 bad-ma). We manually examined each of them and removed 22 bad-ma instances where the difference from corresponding good cases was not significant enough and therefore considered not suitable as training data. The result is a final set A of 653 instances.

Part B consists of training data from the sequential mini-programs. This was collected by running each mini-program for multiple problem sizes, each in both good and bad-ma modes. Here we collected an initial set of 271 instances (171 good, 100 bad-ma). As before, we manually examined each of them and removed 44 (41 good and 3 bad-ma) instances and ended up with 227 instances.

Table 3: Summary of collected training data

	good	bad-fs	bad-ma	Total
Part A (multi-threaded	324	216	113	653
Part B (sequential only)	130	-	97	227
Full training data set	**454**	**216**	**210**	**880**

Thus, our overall training data set, A+B, has 880 instances. We manually label (classify) each training data instance by adding the corresponding mode ("good", "bad-fs" or "bad-ma") as a separate field.

3.2 Classifier Training and Model Validation

With the training data, the classifier constructs a decision-tree model with 6 leaves and 11 nodes, as shown in Figure 2.

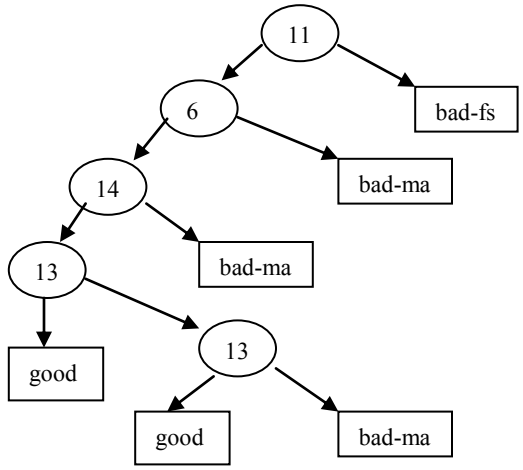

Figure 2: Decision-tree (non-leaf nodes labeled with 'event #' from Table 2; leaf nodes labeled with 'classification')

As can be seen in Figure 2, the model uses only four events (events numbered 11, 6, 14 and 13 in Table 2). We note that the event 11 (Snoop_Response.HIT "M") alone determines the "bad-fs" classification. At every non-leaf node in the tree, branching is to the right if the normalized count of the corresponding event is above a threshold and to the left otherwise.

Stratified 10-fold cross validation on the training data itself shows 875/880 (or 99.4%) overall success rate. The confusion matrix is shown in Table 4.

Table 4: Confusion matrix for training data

		Predicted Class		
		good	bad-fs	bad-ma
Actual Class	good	453	1	0
	bad-fs	0	216	0
	bad-ma	4	0	206

From the preceding, we see our training data and the model are good. We proceed to detect false sharing in arbitrary programs, which is discussed in the next section.

4. DETECTION OF FALSE SHARING

To test our trained classifier model on how well it can detect false sharing, we used the programs in the Phoenix [24][30] and PARSEC [4] benchmark sets. This allows us to compare our results with results from two recent works ([21] and [33]) that use different approaches for detecting false sharing in them.

Table 5 shows the classification summary of the two benchmark suites by our classifier. We ran the programs with all provided input sets (e.g., 3 input sets generally for Phoenix programs and 4 input sets for PARSEC programs), each with different numbers of threads and also with different compiler optimizations (e.g., -O0,…,-O3 in gcc), because false sharing could be reduced to some extent by compiler transformations.

Table 5: Classification results for benchmark programs

Phoenix	Class	PARSEC	Class
histogram	good	ferret	good
linear_regression	bad-fs	canneal	good
word_count	good	fluidanimate	good
reverse_index	good	streamcluster	bad-fs
kmeans	good	swaptions	good
matrix_multiply	bad-ma	vips	good
string_match	good	bodytrack	good
pca	good	freqmine	good
		blackscholes	good
		raytrace	good
		x264	good

The classification for each program in Table 5 is the overall (majority) result considering all cases. For each "good" classification in Table 5, the results were 100% among all cases, except in histogram where 35/36 cases were "good" and 1/36 was "bad-fs". The "bad-ma" classification for matrix_multiply was 100% among all cases. For linear_regression, 24/36 were "bad-fs", 11/36 were "good" and 1/36 was "bad-ma". For streamcluster, 15/36 were "bad-fs", 11/36 were "good" and 10/36 were "bad-ma".

Performance overhead on programs in our approach is minimal. Program execution time often remains almost the same or insignificantly increased, at most by 2%, when collecting performance event counts. In contrast, [21] and [33] reported the program slowdown in the range of 20% and 5x, respectively.

Let us discuss the results in detail in the next subsections.

4.1 Phoenix Benchmarks

In the Phoenix benchmark set, our approach classifies only linear_regression as having false sharing, matrix_mulitply as "bad-ma" and others as "good".

Results in [33] show that linear_regression has been identified as having significant false sharing, with a false positive for histogram. They cannot handle programs kmeans and pca due to a 8-thread limit.

Results in [21] show that in addition to linear_regression, reverse_index and word_count also have been detected as having significant false sharing and kmeans with insignificant false sharing. Subsequently, however, they report that fixing the false sharing in the source code in reverse_index and word_count give only small speedups (2.4% and 1%), indicating that the false sharing in them are in fact insignificant. Thus our detection of false sharing in linear_regression agrees with the common detection of the same by [21]and [33].

4.2 PARSEC Benchmarks

In PARSEC, streamcluster is classified by our approach as having false sharing, and all others as "good". (We could neither build dedup nor run facesim with the given inputs in our test environment).

In [21], streamcluster has been detected as having significant false sharing and canneal and fuidanimate with insignificant false sharing. They have not reported on raytrace, vips, x264, bodytrack, facesim and freqmine due to build/execution issues. In [33], PARSEC programs have not been evaluated.

4.3 Detailed Analysis of Results

The single case of "bad-fs" out of 36 total cases for the program histogram was for 10MB input, with –O2 compiler flag and 12 threads. This result was, however, not consistent in repeated runs and could change to a "good" classification equally well; this case is being investigated.

Table 6 shows the detailed results for linear_regression, with execution time in seconds and the color indicating our classification for each case.

Table 6: Execution time and classification result (bad-fs, good, bad-ma) for different cases of linear_regression

Input	Compiler Flag	Sequential (T=1)	Parallel (T = # of threads)			
			T=3	T=6	T=9	T=12
50MB	-O0	0.28s	0.78s	0.87s	0.63s	0.48s
	-O1	0.08s	0.19s	0.22s	0.17s	0.12s
	-O2	0.06s	0.02s	0.01s	0.01s	0.01s
100 MB	-O0	0.53s	1.46s	1.48s	1.19s	0.91s
	-O1	0.15s	0.40s	0.35s	0.29s	0.23s
	-O2	0.12s	0.05s	0.02s	0.02s	0.01s
500 MB	-O0	2.67s	7.44s	5.81s	5.88s	4.77s
	-O1	0.76s	1.70s	1.80s	1.56s	1.18s
	-O2	0.63s	0.23s	0.12s	0.08s	0.06s

We note that in Table 6, in all "bad-fs" cases, with –O0 and –O1, the sequential version is much faster than the multi-threaded versions. This indicates there is indeed a critical performance

issue and false sharing can be the cause of it. The –O2 flag seem to have resolved that issue, as seen by the execution times, and correspondingly our classification is also "good". With the –O3 flag (not included here), the execution times were very close to those with the –O2 flag. It seems that aggressive compiler optimizations have resolved the issue of false sharing. The isolated "bad-ma" case is not fully understood yet and could be an error that should have been classified as good.

To verify our detection of false sharing as well as to get further insight, we used the technique in [33] and their tool based on Umbra [32] to analyze the cases in Table 6. They count cache contention events among threads and conclude that there is false sharing if the false sharing rate and the contention rate are above 10^{-3}.

Table 7 shows the false sharing rates thus obtained, along with our classifications in color for the T=3 and T=6 cases in Table 6 (their approach can handle maximum of 8 threads).

In Table 7, we see that for cases classified by our approach as "bad-fs", the false sharing rates are 15x-25x greater than the rates for the "good" cases. According to the criteria in [33], however, even these "good" cases have false sharing because the rates are greater than 10^{-3}.

Table 7: False sharing rates [33] and our classifications for `linear_regression` (bad-fs, good, bad-ma)

Input	Compiler Flag	False Sharing Rate (T=# of threads)	
		T=3	T=6
50MB	-O0	0.027500829	0.035161502
	-O1	0.023529737	0.032091656
	-O2	0.001447973	0.001447919
100 MB	-O0	0.025712384	0.032117975
	-O1	0.022127058	0.033850679
	-O2	0.001448503	0.001448311
500 MB	-O0	0.026797920	0.033536338
	-O1	0.022081920	0.033776372
	-O2	0.001449212	0.001449164

As with `linear_regression` shown in Table 7, we applied the approach in [33] on our multi-threaded mini-programs and other programs in the Phoenix and PARSEC benchmark sets. Except for the special cases of `linear_regression` above and the `streamcluster` discussed below, we get consistent results that verify our classifications; i.e., each of our classification as false sharing or not was verified by the corresponding false sharing rate being above 10^{-3} or below 10^{-3}, respectively. In our mini-programs, there is a significant gap (an order of magnitude or more) in the false sharing rates between each pair of cases with and without false sharing.

Table 8 shows the detailed results for `streamcluster` with execution time and our classification for each case.

Table 8: Execution time and classification result for different cases of `streamcluster` (bad-fs, good, bad-ma)

Input	Compiler Flag	# of Threads (T)		
		T=4	T=8	T=12
simsmall	-O1	0.182s	0.194s	**0.445s**
	-O2	0.161s	0.197s	0.231s
	-O3	0.179s	0.189s	0.232s
sim medium	-O1	0.456s	0.347s	0.381s
	-O2	0.377s	0.335s	0.334s
	-O3	0.311s	0.344s	0.444s
simlarge	-O1	1.670s	0.954s	0.899s
	-O2	1.256s	0.816s	0.782s
	-O3	1.273s	0.803s	0.685s
native	-O1	3m12.78s	1m48.59s	1m20.59s
	-O2	2m52.98s	1m37.39s	1m16.41s
	-O3	2m51.71s	1m36.72s	1m14.64s

In Table 8 we see that for cases classified as "bad-fs", the execution time in general does not improve when the number of threads increases along a row. False sharing can cause this.

The top-right cell in Table 8 with 0.445s time (which is quite high, considering the numbers around it) and classified as "good" highlights another issue. With repeated experiments, we observed that the same case with a much shorter execution time and a classification result as "bad-fs" can also happen. Further investigation noted that the longer execution time corresponds to excessively larger number of instructions being executed than with the shorter execution time.

Dramatic increase (or decrease) in execution time together with the instruction count from one execution of a program to another is usually a result of non-deterministic behavior of threads waiting on spin-locks. In the program source code we verified that there is spin-lock waiting by threads. In our method we normalize event counts by dividing them with the number of instructions. Thus the classification of the top-right cell can be either "good" or "bad-fs" depending on whether the number of instructions is quite high or not.

Next let us compare our results with results from the approach in [33] for `streamcluster`. Table 9 shows the false sharing rates obtained from the method in [33] based on Umbra [32] for `streamcluster`, along with our classifications, for the cases T=4 and T=8 in Table 8. We could not run the experiments with the "native" input set as it takes a long time.

Table 9: False sharing rates [33] and our classifications for `streamcluster` (bad-fs, good, bad-ma)

Input	Compiler Flag	# of Threads (T)	
		T=4	T=8
simsmall	-O1	0.00173319	0.001929289
	-O2	0.00194437	0.002242494
	-O3	0.00169222	0.002446181
simmedium	-O1	0.00092664	**0.001120633**
	-O2	0.00117458	0.001551658
	-O3	0.00117411	0.001372999
simlarge	-O1	0.00060055	0.000703761
	-O2	0.00082323	0.000998671
	-O3	0.00082370	0.000909993

According to the criteria in [33], which says there is false sharing if the false sharing rate is greater than 10^{-3}, our classifications are correct for all cases in Table 9 (all "good" and "bad-ma" cases are with no false sharing), except for the single case where the false sharing rate is 0.001120633 and our classification is "good".

In `streamcluster` source, there is a defined constant CACHE_LINE set to 32. It is expected that changing it to 64 would eliminate false sharing [21]. Our approach, however, detected false sharing even after this fix, for the simsmall input for T=8, and it was verified via the approach in [33].

Tables 10 shows the overall summary of verification of our results for the Phoenix and PARSEC benchmarks by the approach in [33], on which the "Actual" columns are based.

Table 10: Verification of our detection of false sharing in Phoenix and PARSEC benchmarks by the approach in [33], on which the "Actual" columns are based (FS=false sharing is present, No FS= no false sharing)

	# cases	Actual		Detected	
		FS	No FS	FS	No FS
Phoenix					
`histogram`	18	0	18	0	18
`linear_regression`	18	18	0	12	06
`word_count`	18	0	18	0	18
`reverse_index`	06	0	06	0	06
`kmeans`	12	0	12	0	12
`matrix_multiply`	18	0	18	0	18
`string_match`	18	0	18	0	18
`pca`	18	0	18	0	18
PARSEC					
`ferret`	18	0	18	0	18
`canneal`	18	0	18	0	18
`fluidanimate`	18	0	18	0	18
`streamcluster`	18	11	07	10	08
`swaptions`	18	0	18	0	18
`vips`	18	0	18	0	18
`bodytrack`	18	0	18	0	18
`freqmine`	16	0	16	0	16
`blackscholes`	18	0	18	0	18
`raytrace`	18	0	18	0	18
`x264`	18	0	18	0	18
Total	**322**	**29**	**293**	**22**	**300**

The quality of our detection of false sharing is shown in Table 11, based on results in Table 10.

Table 11: Performance of our detection of false sharing, based on Table 10 (FS=False sharing is present)

		Detection (Our Classification)	
		FS	No FS
Actual	**FS**	22	7
	No FS	0	293
Correctness		(22+293)/(22+7+0+293) = 97.8%	
False Positive (FP) Rate		0/(293+0) = 0%	

We have been able to detect false sharing with 0 false positives and 97.8% overall correctness, when compared against the "actual" in the Phoenix and PARSEC benchmarks. This is a very good result and we can conclude that our approach can successfully detect false sharing in `linear_regression` and `streamcluster`.

5. RELATED WORK

Different possible definitions of false sharing and its adverse effect on performance are presented in [5][14][19][29].

There have been attempts to prevent or reduce false sharing in programs automatically. Compile-time techniques to reduce false sharing are reported in [18] where information collected on access to shared data are used to identify data structures that could lead to false sharing and they are subjected to transformations including padding. An approach for eliminating false sharing targeting parallel loops has been proposed in [8] where loop iterations are scheduled such that concurrently executed iterations access disjoint cache lines. Static analysis based approaches such as [8] and [18] have limited usage to simple code and data layouts and will not be effective with today's applications that have diverse program structures and complex forms of parallelism. In our experiments with benchmark programs, for example, as seen in Section 4, while some compiler optimizations could reduce false sharing in `linear_regression`, it was not so in `streamcluster`. Hoard [3] is a memory allocator that tries to prevent false sharing of heap objects caused by concurrent requests. Hoard ensures that data allocated for separate threads do not share the same cache line. But this approach cannot prevent false sharing within heap objects as well as those caused by thread contentions due to poor programming or thread scheduling.

Several tools based on simulation or instrumentation with the ability to report on false sharing have been presented. A major drawback common to these is the significant run-time overhead and typically orders of magnitude slowdown. CacheIn [27] is a tool that uses simulation and monitoring to collect and report cache performance data. It has an inefficient form of data processing and also a simple machine model for simulation. Pluto [12] that uses dynamic binary instrumentation to estimate false sharing cannot differentiate between true and false sharing leading to many false positives and has not been tested adequately with real-world applications.

Techniques that use hardware performance events to analyze and enhance performance are numerous, but none of them addresses the difficult task of accurate detection of false sharing. Azimi et al. [2] use dynamic multiplexing and statistical sampling techniques to overcome limitations in online collection of event counts. HPCToolkit [1], PerfExpert [7], Periscope [11] and PTU [17] are tools that use hardware performance events to assess performance bottlenecks automatically in programs and suggest solutions. Using hardware performance events for adaptive compilation and run-time environment for Java has been reported in [25]. Shen et al. [26] use hardware performance events to model service requests in server applications for OS adaptation. Using machine-learning techniques to analyze hardware performance event data is not new; [22] and [31] use regression trees and decision-tree classifications, respectively, to first build models and then predict performance problems in a given program.

The two most recent works on detecting false sharing were [21] and [33]. In Section 4 we compared our approach with these

based on detection of false sharing in Phoenix and PARSEC benchmarks.

Liu and Berger [21] present two tools, one for detection and the other for automatic elimination of false sharing. Both are built on a framework called SHERIFF that replaces the standard pthread library and transforms threads to processes. Their detection has about 20% performance penalty on average. Their detection of significant false sharing in `reverse_index` and `word_count` (with filtering for low performance impact enabled) appears problematic because the speedup they gain after improvement is insignificant and we noted that there is no false sharing in them (see Section 4). While their tool for mitigating false sharing seems performing reasonably well, the reported significant improvement in the execution times of `histogram` and `string_match` cannot be due to removal of false sharing because there is no false sharing in them (see Section 4). Furthermore, when we tried out their tools with the Phoenix and PARSEC benchmarks we encountered unexpected results. We have been in communication with the authors to clarify matters.

Zhao et al. [33] present an approach using memory shadowing [32] to analyze interactions among threads to detect true and false sharing. They define false sharing rate as the total number of false sharing misses divided by the total number of instructions executed and consider there is false sharing if the rate > 10^{-3}. There is heavy runtime overhead (5x slowdown) in their approach, because it uses dynamic instrumentation to keep track of cache contention and cache-line ownership among threads. Another major limitation is that they can track only up to 8 threads for detection of false sharing. Moreover, cold cache misses in `histogram` are incorrectly detected as false sharing misses due to an inherent limitation in their instrumentation method.

6. CONCLUSION

False sharing can seriously degrade performance and scalability in parallel programs yet it is difficult to detect. In this paper we presented an efficient and effective approach based on machine-learning to detect false sharing. Our approach can be applied across different hardware and OS platforms provided performance event counts can be collected and requires neither specialized tools nor access to the source code. It has minimal performance overhead (< 2%), easy to apply and still effective. Applied on programs in the well known PARSEC and Phoenix benchmark sets, our approach detected all cases where false sharing exists with 0 false positives. Compared to the recent work [21] and [33], our approach has unique advantages that include better accuracy in detection.

Ongoing and future work include detecting false sharing at a finer granularity, for e.g., in short time slices or at function-level (work reported in this paper is considering the whole duration of the program) and the study of how the effectiveness of our approach depends on the number and types of performance events and the number and types of mini-programs. We will also test our approach with more real applications and on other hardware platforms.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and suggestions. Part of this work was done when the first author was spending sabbatical at the Massachusetts Institute of Technology. This work was partially supported by DOE award DE-SC0005288, DOD DARPA award HR0011-10-9-0009, NSF awards CCF-0632997, CCF-0811724 and Open Collaborative Research (OCR) program from IBM.

8. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey and N. R. Tallent, HPCToolkit: Tools for performance analysis of optimized parallel programs, In *Concurrency and Computation: Practice and Experience*, 22(6):685–701, John Wiley, 2010.

[2] R. Azimi, M. Stumm and R. Wisniewski, Online performance analysis by statistical sampling of microprocessor performance counters, In *Proceedings of 19th International Conference on Supercomputing* (ICS'05), pages 101–110, ACM, 2005.

[3] E. Berger, K. McKinley, R. Blumofe, and P.Wilson,. Hoard: A scalable memory allocator for multithreaded applications, *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[4] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors, In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[5] 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*, pp. 57—71, 1993.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, A Portable Programming Interface for Performance Evaluation on Modern Processors, *International Journal of High Performance Computing Applications*, Volume 14 Issue 3, pages 189-204, Sage Publications, 2000. (http://icl.cs.utk.edu/papi/)

[7] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, Perfexpert: An easy-to-use performance diagnosis tool for hpc applications, In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, IEEE Computer Society, 2010.

[8] J.-H. Chow and V. Sarkar, False sharing elimination by selection of runtime scheduling parameters. In *Proceedings of the international Conference on Parallel Processing* (ICPP '97), pages 396–403, IEEE Computer Society, 1997.

[9] S. Eranian, Perfmon2: a flexible performance monitoring interface for Linux, In *Proceedings of the 2006 Linux Symposium*, Vol. I, pp. 269-288, (http://perfmon2.sourceforge.net)

[10] S. Eranian, What can performance counters do for memory subsystem analysis?, In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with ASPLOS '08*, pp. 26—30, ACM, 2008.

[11] M. Gerndt and M. Ott, Automatic performance analysis with periscope, In *Concurrency and Computation: Practice and Experience*,, 22(6):736–748, John Wiley, 2010.

[12] S.M. Gunther and J. Weidendorfer, Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation, In

Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09), pages 26—33, ACM, 2009.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, The WEKA Data Mining Software: An Update, *ACM SIGKDD Explorations Newsletter*, Volume 11 Issue 1, pages 10-18, ACM, 2009.

[14] R. L. Hyde and B. D. Fleisch, An analysis of degenerate sharing and false coherence, *Journal of Parallel and Distributed Computing*, 34(2):183–195, 1996.

[15] Intel Corporation, *Avoiding and identifying false sharing among threads*, http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads, January 2013.

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide Part 2*, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html?wapkw=software+developer%E2%80%99s+manual+volume+3b, 2012.

[17] Intel Corporation, *Intel® Performance Tuning Utility 4.0 User Guide*, 2011.

[18] T. Jeremiassen and S. Eggers, Reducing false sharing on shared memory multiprocessors through compile time data transformations, *ACM SIGPLAN Notices*, 30(8):179–188, 1995.

[19] V. Khera, R.P. LaRowe and C.S. Ellis, *An Architecture-Independent Analysis of False Sharing*, Technical Report TR-CS-1993-13, Duke University, 1993.

[20] D. Levinthal, *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*, Intel Corporation, http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.

[21] T. Liu and E.D. Berger, SHERIFF: Precise Detection and Automatic Mitigation of False Sharing, In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (OOPSLA'11), pages 3—18, ACM, 2011.

[22] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. Doshi, and S. Abraham, Using model trees for computer architecture performance analysis of software applications, In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (ISPASS'07), 116–125, IEEE, 2007.

[23] J.R. Quinlan, *C4. 5: Programs for Machine Learning*, Morgan Kaufmann, 1992.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (HPCA'07), pages 13-24, IEEE Computer Society, 2007.

[25] F.T. Schneider, M. Payer, and T.R. Gross, Online optimizations driven by hardware performance monitoring, In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (PLDI'07), pages 373–382, ACM, 2007.

[26] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang, Hardware counter driven on-the-fly request signatures, In the *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS'08), pages 189--200, ACM, 2008.

[27] J. Tao and W. Karl, CacheIn: A Toolset for Comprehensive Cache Inspection, In *Proceedings of the 5th international conference on Computational Science* (ICCS'05) - Volume Part II, 2005, pages 174–181, Springer-Verlag, 2005.

[28] V. Weaver, The Unofficial Linux Perf Events Web-Page, http://web.eece.maine.edu/~vweaver/projects/perf_events/, February 2013.

[29] J. Weidendorfer, M. Ott, T. Klug and C. Trinitis, Latencies of conflicting writes on contemporary multicore architectures, In *Proceedings of the 9th international conference on Parallel Computing Technologies* (PaCT 2007), pages 318–327, Springer-Verlag, 2007.

[30] R. M. Yoo, A. Romano, and C. Kozyrakis, Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System, In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization* (IISWC'09), pages 198-207, IEEE Computer Society, 2009.

[31] W. Yoo, K. Larson, L. Baugh, S. Kim and R.H. Campbell, ADP: Automated diagnosis of performance pathologoes using hardware events, In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 283—294, ACM, 2012.

[32] Q. Zhao, D. Bruening and S. Amarasinghe, Umbra: efficient and scalable memory shadowing, In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (CGO'10), pages 22-31, ACM, 2010.

[33] Q. Zhao, D. Koh, S. Raza, D. Bruening, W. 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'11), pages 27—38, ACM, 2011.