CHEETAH: Detecting False Sharing Efficiently and Effectively

Abstract

False sharing is a notorious performance problem that may occur in multithreaded programs running on ubiquitous multicore hardware. It can dramatically degrade the performance by up to an order of magnitude, significantly hurting the scalability. Identifying false sharing in complex programs is challenging. Existing tools either incur significant overhead or do not provide adequate information to guide code optimization.

To address these weaknesses, we develop CHEETAH, a profiler that detects false sharing both **efficiently** and **effectively**. CHEETAH leverages lightweight hardware performance monitoring units that are available on most modern CPU architectures to sample memory accesses. CHEETAH is the first tool that can quantify the optimization potential of a false sharing problem without actual fixes. CHEETAH precisely reports false sharing and provides insightful optimization guidance for programers, with adding less than 7% runtime overhead on average. CHEETAH is ready for real deployment.

Keywords Multithreading, False Sharing, Address Sampling, Lightweight Profiling, Performance Prediction

1. Introduction

Multicore processors are ubiquitous in the whole computing spectrum, from mobile phones, personal desktops, to high-end servers. Multithreading is the de-facto programming model to exploit the massive parallelism in modern architectures. However, multithreaded programs may suffer from various performance issues because of complex memory hierarchies [21, 23, 29]. Among them, false sharing is a common flaw that can significantly hurt the performance and scalability of multithreaded programs [4]. False sharing occurs when different threads, which are running on different cores with private caches, concurrently access logically independent words of the same cache line. When a thread modifies the data of a cache line, the cache coherence protocol (managed by hardware) silently invalidates the duplicates of this cache line in the private caches of other cores [25]. Thus, even if other threads access independent words of this cache line, they have to unnecessarily reload the entire cache line from the shared cache or main memory.

Unnecessary cache coherence caused by false sharing can dramatically degrade the performance of multithreaded programs by up to an order of magnitude [4]. A concrete example shown in Figure 1 also shows this catastrophic performance effect. We meant to use threads to accelerate the computation for the `array`. However, when eight threads (on an 8-core machine) simultaneously access adjacent elements of `array`, this program actually runs around 13× slower (back bars) than its expectation (grey bars) with no false sharing. The hardware trend, including adding more cores into the same machine, introducing the Non-Uniform-Memory-Access (NUMA) architecture, or increasing the size of a cache line, will further degrade the performance of programs with false sharing.

Unlike true sharing, false sharing is generally avoidable. When threads unnecessarily share the same cache line, we can pad the data or make a thread to update its thread-private variable so that different threads can be forced to access a different cache line. Although the solution of fixing false sharing is somehow straightforward, detecting them is difficult and even impossible with manual checking, especially for a program with thousands or millions lines of code. Thus, it is very important to employ tools to pinpoint false sharing and provide insightful optimization guidance.

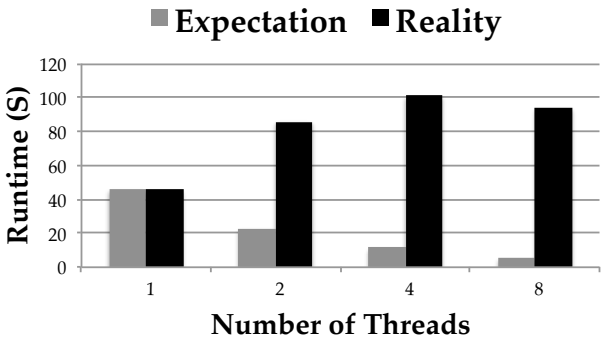
However, existing general-purpose tools do not provide enough details about false sharing [7, 11, 21]. Existing false sharing detection tools fall short in several ways. First, most tools cannot distinguish true and false sharing, requiring substantial manual effort to identify optimization opportunities [9, 12, 13, 16, 18, 24, 26, 30–32]. Second, software-only tools [9, 18, 20, 28] introduce very high runtime overhead, preventing them from being used in deployment environment. Third, OS-related tools have limitations, either with customized OS support [24], or only working on special applications [19]. Fourth, no prior tool assesses the benefit from eliminating the false sharing. Without this information, many optimization efforts may yield trivial or no performance improvement.

In this paper, we present CHEETAH to address these issues, with the following contributions:

- **First Approach to Predict False Sharing Impact.** This paper introduces the first approach to predict the performance impact of fixing false sharing instances inside multithreaded programs. Based on our evaluation, CHEETAH can precisely assess the performance improvement, with less than 10% difference. By ruling out trivial instances, we can avoid unnecessary manual effort leading to little or no performance improvement.

```
int array[total];
int window=total/numThreads;
void threadFunc(int start)
{
    for(index=start; index<start+window; index++)
        for(j=0; j<10000000; j++)
            array[index]++;
}
```

(a) A Program with False Sharing



(b) Performance Degradation

Figure 1. A false sharing example inside a multithreaded program (a) causes 13× performance degradation (b) on a 8-core machine.

- **An Efficient and Effective False Sharing Detection Tool.** CHEETAH is an efficient false sharing detector with only ∼7% runtime overhead. CHEETAH utilizes the Performance Monitoring Units (PMU) that are available on modern CPU architectures to sample memory accesses. CHEETAH can precisely and correctly report false sharing problems, by pointing out the lines of code or the name of variables with problems.

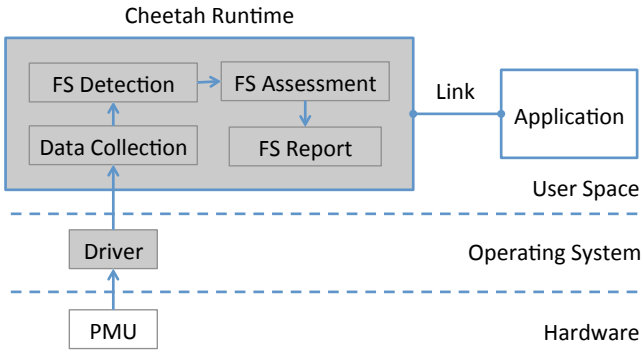


Figure 2. Overview of CHEETAH's components (shadow blocks).

Figure 2 shows the overview of CHEETAH. The "data collection" module collects memory accesses via the hardware performance monitoring unit (PMU) supported address sampling with the "driver"'s assistance, filters out memory accesses relating to heap or globals, and feeds them into the "FS detection" module. At the end of an application or when CHEETAH is notified to report, the "FS detection" module examines the number of cache invalidations of each cache line, differentiates false sharing from true sharing, and passes false sharing instances to the "FS assessment" module. In the end, the "FS report" module only reports false sharing instances with significant performance impact, with its predicted performance improvement after fixes for every instance provided by the "FS assessment" module.

The remainder of this paper is organized as follows. Section 2 describes the design and implementation of false sharing detection tool. Section 3 introduces how CHEETAH assesses the performance impact of a false sharing instance. Section 4 presents experimental results, including effectiveness, performance overhead, and the precision of assessment. Section 5 addresses main concerns on hardware dependence, performance and effectiveness. Section 6 discusses related work and Section 7 concludes this paper.

2. Detecting False Sharing

CHEETAH only reports false sharing with significant performance impact. Since only cache lines with a large number of cache invalidations can have significant performance impact on the performance, CHEETAH should locate those cache lines. However, this turns out to be difficult because cache invalidations depend on memory access patterns, cache hierarchies, and thread-to-core mappings. To address this challenge, CHEETAH proposes a simple rule to track cache invalidations: *if* a thread writes a cache line after other threads have accessed the same cache line, this write operation causes a cache invalidation. This rule is based on the following two assumptions:

- **Assumption 1:** Each thread runs on a separate core with its own private cache.
- **Assumption 2:** Cache sizes are infinite.

Assumption 1 is reasonable because thread over-subscription is generally rare for computation intensive programs. This assumption may over-estimate the number of cache invalidations, if multiple threads are actually scheduled to the same physical core or different cores sharing part of cache hierarchy. Under this circumstance, Assumption 1 shows the worst scenario for a given false sharing instance. With this assumption, CHEETAH does not have to track thread-to-core mapping or know the actual cache hierarchy.

Assumption 2 further defines the behavior related to cache eviction and invalidation. If there is a memory access within

a cache line, the hardware cache (of running this thread) always holds the data until an access issued by other threads (running on other cores by assumption 1) invalidates it. This assumption avoids tracking cache evictions caused by cache capacity. By combing these two assumptions, CHEETAH identifies cache validations simply based on memory access patterns, independent from the architecture configurations and parallel execution environments [20, 32].

In the remaining of this section, we elaborate how we install PMU-based sampling mechanism for memory accesses in Section 2.1, how we track memory accesses and data allocation in Section 2.2, how we analyze access patterns for cache invalidation in Section 2.3, and how we report false sharing in Section 2.4.

2.1 Sampling Memory Accesses

According to the basic rule described above, it is important to track memory accesses of different threads in order to compute the number of cache invalidations on each cache line. Software based approaches may introduce higher than 5× performance overhead [20, 32]. This high overhead can block people from using these tools in real deployment.

CHEETAH significantly reduces the performance overhead by leveraging PMU-based sampling mechanisms, such as AMD instruction-based sampling (IBS) [6] and Intel precise event-based sampling (PEBS) [14], pervasively available in modern CPU architectures. For each sample, PMU distinguishes whether it is a memory read or write, captures the memory address touched by the sampled memory access, and records the thread ID that triggers this sample. Those raw data will be utilized to analyze whether the sampled access incurs a cache invalidation or not, based on the basic rule described in Section 2.

Since PMU only samples one memory access out of a specified number, typically with a sampling frequency of tens of samples per second per thread, it doesn't pose significant performance overhead. Besides that, using PMU-based sampling does not need to instrument source or binary code explicitly, thus providing a non-intrusive way to monitor memory references. CHEETAH shows that PMU-based sampling, even with sparse samples (one out of 64K), can identify false sharing with significant performance impact.

Implementation. In order to sample memory accesses, CHEETAH programs the PMU registers to turn on sampling with a pre-defined threshold, before entering the `main` routine of the application. It also installs a signal handler to associate with the sampling so that we can collect detailed memory accesses. In order to simplify the signal handling, CHEETAH configures the signal handler to be responded by the current thread, by calling `fcntl` function with `F_SETOWN_EX` flag. CHEETAH also sets up the custom allocator and its internal heap in the initialization. Inside the signal handler, CHEETAH collects detailed information of every sampled memory access, including its memory address, thread id, read or write operation, and access latency, which can be fed into the computing module to compute the number of cache invalidations and the prediction module to predict performance impact.

2.2 Locating Cache Line

For each sampled memory access, CHEETAH will decide whether this access causes a cache invalidation or not and records the detailed information about this access. For this purpose, CHEETAH should quickly locate an access's related cache line. CHEETAH utilizes the shadow memory mechanism to speed up this locating procedure [20, 32]. To utilize the shadow memory mechanism, we should determine the range of heap memory, which is difficult to know beforehand if using the default heap. Thus, CHEETAH built its custom heap based on Heap Layers [2]. CHEETAH pre-allocates a fixed size of memory block (using `mmap`) and satisfies all memory allocations from that. CHEETAH adapts the per-thread heap organization used by Hoard so that two objects in the same cache line will never be allocated to two different threads [1]. This design prevents inter-objects false sharing, but also makes CHEETAH cannot report problems caused by the default heap allocator.

Implementation To use its custom heap, CHEETAH intercepts all memory allocations and deallocations. CHEETAH initializes the heap before an application enters `main` routine, by putting the initialization routine into the constructor attribute. CHEETAH maintains two different heaps, one for the application and one for internal uses. For both heaps, CHEETAH manages objects based on the unit of *power of two*. For each memory allocation from the application, CHEETAH saves the information of callsite and size, which helps CHEETAH to precisely report the line information of falsely-shared objects. CHEETAH allocates two large arrays (using `mmap`) to keep track of the number of writes and detailed access information on each cache line. For each memory access, CHEETAH uses the bit shift to compute the index of cache line and locates the placement of its corresponding unit quickly.

2.3 Computing Cache Invalidations

Prior work of Zhao et. al. proposed a method based on the ownership of cache lines to compute the cache invalidations: when a thread updates an object owned by others, this access causes an cache invalidation and resets the owner to the current thread [32]. But this approach cannot easily scale to more than 32 threads because of excessive memory consumption, since it needs a bit to track the ownership of a thread.

To address this problem, CHEETAH maintains a two-entry table (T) for each cache line (L), with eight words in total. In this table, each entry has two fields, thread ID and access type (read or write). It computes the invalidations according to the rule described in Section 2. In case of a cache invalidation, the current access (write) will be added into the corresponding table. Thus, each table always has an entry. More

specifically, CHEETAH checks possible cache invalidations as follows.

- For each read access, CHEETAH will decide whether to record this entry. If the table T is not full and the existing entry is coming from a different thread (with a different ID), CHEETAH will record this read access in the table.
- For each write access, CHEETAH will decide whether this access incurs an invalidation. If the table is already full, it will lead to a cache invalidation since at least an existing entry is from a different thread (assumption 1). If the table is not full (and not empty), a write access will incur a cache invalidation only if the existing entry is from a different thread. CHEETAH does nothing if this write is the same as the existing entry.

Implementation As aforementioned, only cache lines with a big number of writes can possibly have a big impact on the performance. Based on this observation, cache lines with a small number of writes are never be a target that can cause serious performance degradation. For this reason, CHEETAH tracks the number of writes on a cache line at first, and only tracks detailed information when this number is larger than a pre-defined threshold. Using this method can also save memory, since CHEETAH only allocates memory to record word-level read/write access: what is the address of an access; whether this is a read or write access; which thread issues this access. All of these are going to be checked in the reporting phase, described in the next section.

2.4 Reporting False Sharing

CHEETAH reports false sharing correctly and precisely, either at the end of programs or receiving instructions from users.

Correct Detection. CHEETAH keeps track of word-based (four bytes) memory accesses on susceptible cache lines: how many reads or writes issued by a specific thread on each word. When more than one thread access a word, CHEETAH marks this word to be *shared*. By identifying accesses on each word on a susceptible cache line, we can easily differentiate false sharing from true sharing. Word-based information can also help diagnose false sharing problems in more detail, which helps programmers to decide how to pad a problematic data structure in order to avoid false sharing. Because it is possible for a thread, particularly the main thread, to allocate an object and initialize it before being accessed by other threads, CHEETAH only tracks the behavior of memory accesses inside parallel regions.

Precise Detection. CHEETAH reports precise information for global variables and heap objects that are involved in false sharing. For global variables, CHEETAH reports names and addresses by searching through the ELF symbol table. For heap objects, CHEETAH reports the lines of code for their allocation sites. Thus, CHEETAH intercepts all memory allocations and de-allocations to obtain the whole call stack. CHEETAH does not monitor stack variables because they are normally accessed by their hosting threads, without sharing.

3. Assessing the Performance Impact

Fixing false sharing problems does not necessarily yield significant performance speedups, even for instances with a large number of cache invalidations [19, 20]. Zhao et. al. even observed that fixing may even slowdown a program because of excessive memory consumption or the lose of locality [32]. Reporting insignificant false sharing instances are not false positives, but that increases manual burden for programmers.

To resolve this problem, CHEETAH makes the first attempt to quantitatively assess the potential performance gain of fixing a false sharing instance. Thus, programmers can focus on severe problems only, avoiding unnecessary manual effort.

CHEETAH's assessment is based on the following observations:

- **Observation 1:** the sampling is evenly distributed over the whole execution. Based on this, we can use the results of sampling to represent the whole execution. The similar idea is widely used by exiting work like Oprofile and Gprof when they identify the hotspots of function calls and code [8, 17].
- **Observation 2:** PMU hardware provides the latency information (e.g. cycles) of each memory operation. We also observed that the latency of accessing falsely shared objects is significantly higher than that of normal accesses.

Based on these two observations, **we propose to use the sampled cycles to represent the runtime time of an execution and further predict the performance impact of a falsely-shared object** O. The assessment is performed in three steps listed as follows.

- CHEETAH first evaluates the performance impact on accesses of this object O, as discussed in Section 3.1.
- Then CHEETAH assesses the performance impact on the related threads by fixing this false sharing problem, which is discussed in Section 3.2.
- In the end, CHEETAH assesses the performance impact on the application in Section 3.3.

3.1 Impact on Accesses of the Object

CHEETAH first predicts the total cycles of all accesses after fixing. To assess the performance impact of fixing a falsely-shared object O, CHEETAH will collect the following information related to this object O: total cycles of accesses – $Cycles_O$, total number of accesses – $Accesses_O$, and the average cycles of every memory access after fixing – $AverCycles_{nofs}$.

It computes to the total cycles of all accesses after fixing ($PredictCycles_O$) according to EQ.(1).

$$PredictCycles_O = (AverageCycles_{nofs} * Accesses_O); \quad (1)$$

However, it is impossible to know $AverCycles_{nofs}$ for O after fixing without actual fixes. Thus, CHEETAH *utilizes the average cycles of each access in the serial phases to approximate this*. There is no false sharing in serial phases, thus $AverCycles_{nofs}$ represents the best scenario of fixing an existing false sharing instance. In reality, the cycles of every access after fixing can be larger than the $AverCycles_{nofs}$, since fixing false sharing may lead to excessive memory consumption or the lose of locality [32]. Thus, our prediction actually provides an upper bound of performance improvement. It is expected that $PredictCycles_O$ will be less than the actual cycles – $Cycles_O$, since fixing a false sharing problem can reduce the execution time of an object.

3.2 Impact on Related Threads

After that, CHEETAH assesses the possible execution time of different threads affected by fixing false sharing of this object O.

CHEETAH collects the following information of every thread: the execution time – RT_{thread}, the total number of accesses – $Accesses_{thread}$, and the total cycles of all memory accesses – $Cycles_{thread}$.

To collect these numbers, CHEETAH intercepts the creation of threads by passing a custom function as the start routine. In this custom function, CHEETAH gets the timestamps of before and after running the thread function, so that it can compute the execution time of a thread – RT_{thread}. To gather the number of accesses and cycles of every thread, CHEETAH makes every thread to respond the signal handler and records the number of accesses and cycles correspondingly.

After the collection, CHEETAH predicts the cycles of related thread ($PredictCycles_{thread}$) as the EQ.(2), after fixing false sharing of object O.

$$PredictCycles_{thread} = Cycles_{thread} - Cycles_O + PredictCycles_O \quad (2)$$

Based on this, CHEETAH assesses the predicted runtime of a thread — $PredictRT_{thread}$ — as the EQ.(3). We assume that **the execution time is proportional to the cycles of accesses**: the less of cycles means the less of the execution time.

$$PredictRT_{thread} = (PredictCycles_{thread}/Cycles_{thread}) * RT_{thread} \quad (3)$$

3.3 Impact on the Application

In the end, CHEETAH assesses how fixing a false sharing instance will change the performance of the application.

Actually, improving the performance of a thread may not increase the final performance if this thread is not in the critical path. To simplify the prediction and verify our idea, CHEETAH currently focuses on applications with the normal fork-join model, shown as Figure 3. This model is the most important and widely used model in reality. All applications that we have evaluated in this paper utilizes this fork-join model. The performance assessment will be more complicated if nested threads exist inside an application.

CHEETAH tracks creations and joins of threads in order to verify whether an application belongs to the fork-join model or not. CHEETAH also collects the execution time of different serial and parallel phases, using RDTSC (ReaD-Time Stamp Counter) available on X86 machines [10]. As Figure 3, an application enters the first parallel phase after the first thread creation. The application leaves a parallel phase after all children threads are joined successfully.

Based on the runtime information about every parallel phase and serial phase, CHEETAH can assess the final performance impact by recomputing the length of each phase and the total time after fixing: the length of each phase is decided by the thread with the longest execution time, while the total time is equal to the sum of different parallel phases and serial phases.

After CHEETAH computes the possible execution time of an application after fixing a false sharing problem, CHEETAH will compute the potential performance improvement based on EQ.(4), where runtime is denoted by RT. We will report the potential performance improvement for every falsely-shared object in the end as Figure 5. Then programmers can focus only on those ones with serious performance impact. We further verify the precision of CHEETAH's assessment in Section 4.3.

$$Perf_{improve} = RT_{App}/PredictRT_{App} \quad (4)$$

4. Evaluation

The evaluation will answer the following research questions in order as follows.

- What is the performance overhead of CHEETAH? (4.1)
- How effective can CHEETAH find known false sharing problems? How helpful are outputs in fixing false sharing problems? (4.2)
- How is the precision of assessment on each false sharing instance? (4.3)

Experimental Setup. We evaluate CHEETAH on an AMD Opteron machine, which has 48 1.6 GHz cores, 64 KB private L1 data cache, 512 KB private L2 cache, 10 MB shared L3 cache, and 128 GB memory. We use gcc-4.6 with `-O2` option to compile all applications. Since the machine is a NUMA machine and performance varies with different scheduling policy, we bind the threads to cores for consistent performance.

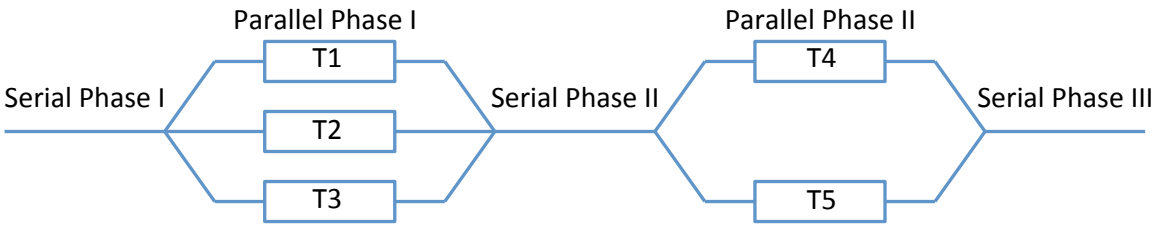


Figure 3. The fork-join model that CHEETAH currently focuses on to assess the impact of false sharing instances.

Evaluated Applications. As prior work [16, 19, 20, 32], we perform experiments on two well-known benchmark suites, Phoenix [27] and PARSEC [3]. We intentionally use 16 threads in order to run applications a little longer, since CHEETAH needs enough samples to detect false sharing problems. If that is still not enough, we change the source code or use a larger input so that every benchmark will run at least 5 seconds to get enough statistical samples.

4.1 Performance Overhead

We show the average runtime overhead of CHEETAH in Figure 4.1. We run each application for five times and show the average results here. According to this figure, CHEETAH only introduces around 7% performance overhead, which makes it practical to be utilized in the real deployment.

During the evaluation, we configure CHEETAH with the sampling frequency at 64K instructions. Thus, for every 64K instructions, the trap handler will be notified once so that CHEETAH can collect the information of each sampled memory access. Because of CHEETAH's very coarse sampling rate, we should let programs to run sufficiently long enough.

The performance overhead of CHEETAH mainly comes from every sampled memory access and each thread creation. For each sample, we will collect information about this memory access, such as the type of access (read or write), the number of cycles, and update corresponding data structure such as the history table of its corresponding cache line. CHEETAH also intercepts every thread creation to setup the PMU unit, get the timestamp and update the phase information. The setting of the PMU registers introduces non-negligible overhead since it invokes six pfmon APIs and six system calls. For applications with a big number of threads, including `kmeans` (with 224 threads, running around 14 seconds) and `x264` (with 1024 threads, running around 40 seconds), we observe that CHEETAH introduces significant more performance overhead than other threads. For other applications, CHEETAH introduces less than 12% performance overhead and 4% overhead on average (if these two applications are excluded).

4.2 Effectiveness

CHEETAH successfully detects two known false sharing problems with significant performance impact, including`linear_regression` in Phoenix and `streamcluster` in Parsec.

```
Detecting false sharing at the object: start 0x400004b8
end 0x400044b8 (with size 4000).
Accesses 1263 invalidations 27f writes 501 total
latency 102988 cycles.

Latency information:
totalThreads 16
totalThreadsAccesses 12e1
totalThreadsCycles 106389
totalPossibleImprovementRate 576.172748%
(realRuntime 7738 predictedRuntime 1343).

It is a heap object with the following callsite:
linear_regression-pthread.c: 139
```

Figure 5. CHEETAH reports a false sharing problem in `linear_regression`.

4.2.1 Case Study: linear_regression

Figure 5 shows the output of CHEETAH. It points out that the `tid_args` object allocated at line 139, with the structure type `lreg_args`, incurs a severe false sharing problem. According to the assessment, fixing it can possibly improve the performance by 5.7×. By examining the source code, we can discover that the `tid_args` object is passed to different threads–linear_regression_pthread. Then we can easily find out where false sharing has been exercised, which is shown as Figure 6. By checking the word-based accesses that is not shown here, we can understand the reason of this false sharing problem: different threads are updating different parts of the object `tid_args` simultaneously, where each thread updates the size of the structure lreg_args of this common array. This problem is similar to the example shown in Figure 1.

To address the problem, we pad the structure `lreg_args` with extra bytes, by adding 64 bytes useless content, so that we can force different threads not to access the same cache line. The simple optimization lead to a 5.7× speedup, which matches the assessment 5.76× predicted by CHEETAH.

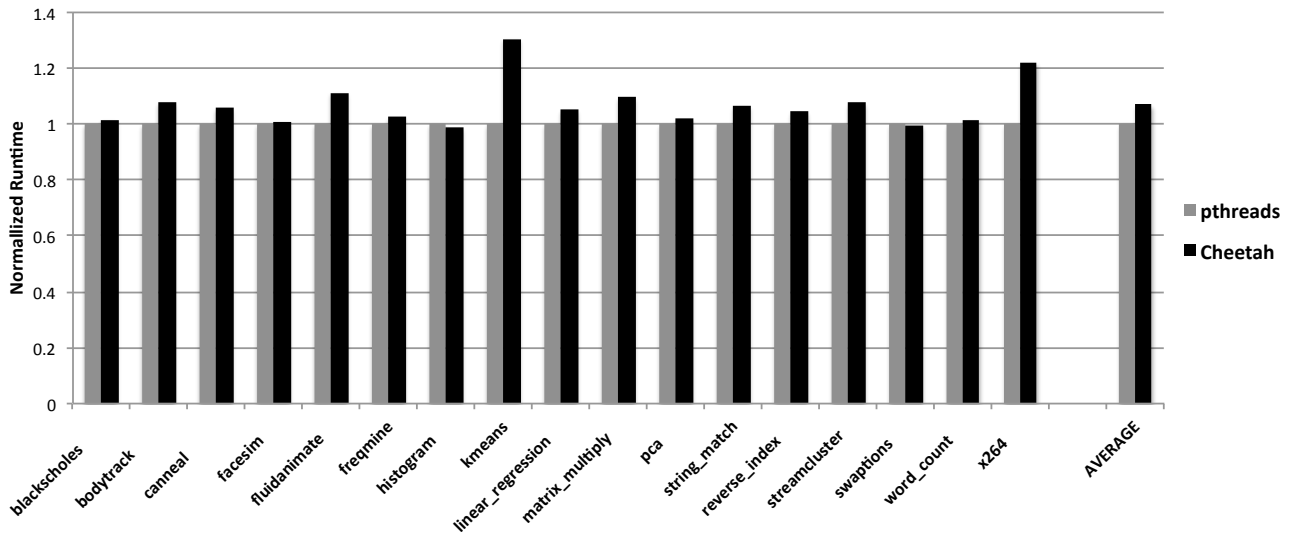


Figure 4. Runtime overhead of CHEETAH. We normalize the runtime to that of `pthreads`. On average, CHEETAH only introduces around 7% performance overhead on all evaluated benchmarks, which makes it practical to be used in the deployed software.

```
typedef struct
{
  ......
  long long SX;
  long long SY;
  long long SXX;
  ......
} lreg_args;

for (i = 0; i < args->num_elems; i++)
{
  //Compute SX, SY, SYY, SXX, SXY
  args->SX  += args->points[i].x;
  args->SXX += args->points[i].x
             *args->points[i].x;
  args->SY  += args->points[i].y;
  ......
}
```

Figure 6. The data structure and source code related to a serious false sharing instance in `linear_regression`.

4.2.2 Case Study: streamcluster

We do not show the report results of streamcluster because of space limit. For streamcluster, every thread will update the work_mem object concurrently, allocated in line 985 of the file `streamcluster.cpp`. The authors have already added some padding to avoid false sharing. However, the size of cache line (as a macro) is set to be 32 bytes, which is smaller than the size of actual cache line used in our experimental machine. Thus, streamcluster will still has a significant false sharing problem because of this reason. The performance impact of fixing false sharing problems inside is further discussed in Section 4.3.

4.2.3 Comparing with State-of-the-art

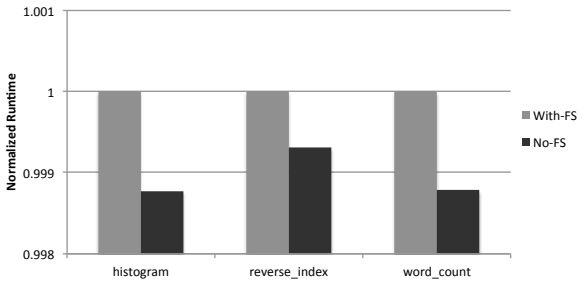


Figure 7. False sharing problems missed by CHEETAH have negligible (<0.2%) performance impact .

As discussed before, CHEETAH only detects false sharing problems that actually occur in the execution and can have significant performance impact on the final performance. As observed by Predator [20], the occurrences of false sharing can be affected by the starting address of objects or the size of the cache line. The performance impact of a false sharing can be greatly impacted by the cache hierarchy. If the number of accesses on a falsely-shared object is not large enough, CHEETAH may not be able to detect it because of its sampling feature.

Comparing with the state-of-the-art tool–Predator, CHEETAH misses false sharing problems in `histogram`, `reverse_index` and `word_count` [20]. We further fix these false sharing problems according to Predator's detection results. We run these applications on our experimental hardware, with and without false sharing problems.

Application	Threads (#)	Predict	Real	Diff (%)
linear_reg	16	6.44X	6.7X	-3.8
linear_reg	8	5.56X	5.4X	+3.0
linear_reg	4	3.86X	4.1X	-5.8
linear_reg	2	2.18X	2X	+9
streamcluster	16	1.016X	1.015X	0
streamcluster	8	1.017X	1.018X	0
streamcluster	4	1.024X	1.022X	0
streamcluster	2	1.033X	1.035X	0

Table 1. Precision of assessment. CHEETAH implements the first method to accurately predict the performance improvement of fixing an observed false sharing instance.

Figure 4.2.3 shows the performance impact. According to Figure 4.2.3, all of these benchmarks do not show a significant speedup after fixing, with less than 0.2% performance improvement. This behavior actually exemplifies the advantage of CHEETAH: *since* CHEETAH *only reports false sharing problems with significant performance impact, it can potentially save programmers manual effort unnecessarily spending on applications with trivial performance improvement*.

4.3 Assessment Precision

Different with all existing tools, CHEETAH can assess the performance impact of false sharing problems so that programmers can focus only on important problems. This section evaluates the precision of assessment.

We evaluate the precision of assessment on two applications, `linear_regression` and `streamcluster`, since they are observed to have serious false sharing problems. We listed the results in Table 1. In this table, `linear_regression` is abbreviated as "linear_reg". We evaluate these two applications when the number of threads is equal to 16, 8, 4, and 2 correspondingly. We list the predicted performance impact in the "Predict" column and the actual improvement in the "Real" column of the table. The last column of this table lists the difference between the predicted improvement and the real improvement. If the number is larger than 0, the predicted performance improvement is less than the real improvement. Otherwise, it is the opposite.

Results of Table 1 show that CHEETAH can perfectly assessment the performance impact of false sharing in every case, with less than 10% difference for every evaluated execution. *According to this novel assessment of* CHEETAH*, programmers may save huge amount of manual effort spending unnecessarily on applications with trivial false sharing problems*.

5. Discussion

This section addresses some possible concerns related to CHEETAH.

Hardware Dependence. CHEETAH is an approach that relies on the hardware PMU unit to sample memory accesses. To use CHEETAH, users should setup the PMU-based sampling beforehand. After that, they can connect to the CHEETAH library by calling only one API–`handleAccess`. Users should add the initialization of PMU at startup or in the beginning of every thread, with adding less than 5 lines of code in total.

Performance Overhead. On average, CHEETAH only introduces 7% performance overhead for all evaluated applications. However, CHEETAH does introduce more than 20% overhead for two applications with a large number of threads because CHEETAH should setup hardware registers for every thread. Although creating a large number of threads in an application is not normal, we expect to only setup once with better hardware support. We also expect that the overhead can be further reduced by only sampling memory accesses, but not non-related instructions such as arithmetic instructions and logic instructions. Hopefully, the hardware can overcome these two limitations in the future.

Effectiveness. CHEETAH can effectively detect false sharing problems that occur in the current execution and have high impact on the performance. For the effective detection, CHEETAH requires programs to run sufficiently long, maybe more than few seconds.

6. Related Work

CHEETAH proposes to predict the potential performance improvement after fixing an existing false sharing problem, by utilizing the access cycles provided by the PMU hardware. Prior work, including HPCToolkit [21] or ArrayTool [22], attributes latency metrics to both variables and instructions. But we did not find any work that is close to our target: using latency to predict the performance after fixes.

In this section, we will review existing tools for detecting false sharing issues, and other techniques to utilize PMU for dynamic analysis.

6.1 False Sharing Detection

Existing tools in false sharing detection can be classified into different types, based on the method of collecting memory accesses or cache related events.

Simulation Based Approaches. Simulation based approaches simulate the behavior of program executions and predict possible problems inside [28]. Simulation based approaches generally introduce more than 100× performance overhead and can not simulate large applications.

Instrumentation Based Approaches. Tools in this category can be further divided into dynamic instrumentation based approaches [9, 18, 32], and static or compiler-based instrumentation based approaches [20]. These approaches generally has less performance overhead, running around 5×

slower [20, 32] to 100× slower [9, 18]. Among them, Predator is the state-of-the-art tool in false sharing detection and uncovers the most number of false sharing instances [20]. However, their performance overhead is still too high to be used in deployed software.

OS Related Approaches. Sheriff proposes to turn threads into processes, relies on page-based protection and isolation to capture memory writes, and reports write-write false sharing problems with reasonable overhead (around 20%) [19]. Plastic provides a VMM-based prototype system that combines the Performance Counter Monitoring (PMU) and page granularity analysis (based on page protection) [24]. Both Sheriff and Plastic can automatically tolerate serious false sharing problems as well. However, these tools have their own shortcomings: Sheriff can only work on programs using `pthreads` libraries, and without ad-hoc synchronizations and sharing-the-stack accesses; Plastic requires that programs should run on the virtual machine environment, which is not applicable for most applications.

PMU-based approaches. For performance reason, more and more tools turns to the hardware support to detect false sharing problems. Jayasena et. al. proposes to collect hardware events such as memory accesses, data caches, TLBs, interactions among cores, and resources stalls, and then uses the machine learning approach to derive potential memory patterns for false sharing [16]. However, their approach misses many false sharing problems. DARWIN collects cache coherence events at the first round, then identifies possible memory accesses on data structures that are involved in frequent cache invalidations for the second round [31]. DARWIN requires manual effort or expertise to verify whether false sharing occurs or not. Intel's PTU relies on memory sampling mechanism but can not differentiate false sharing and true sharing since it discards the temporal information of memory accesses and can not report detailed memory accesses on a cache line [12]. DProf helps programmers identify cache misses based on AMD's instruction-based sampling hardware [26]. DProf requires manual annotation to locate data types and object fields, and cannot detect false sharing when multiple objects reside on the same cache line.

CHEETAH falls into the category of PMU-based approaches. But it avoids the shortcoming of existing PMU-based tools. CHEETAH does not rely on manual annotation, or experts' expertise, or multiple executions to locate the problem. CHEETAH does not have false positives. It has much better performance than the approaches of using simulation and instrumentation, and avoids the limitations of OS-related approaches. Moreover, CHEETAH is the first tool that uses the memory access latency metric to quantify the performance gains from eliminating false sharing problems.

6.2 Other PMU Related Analysis

PMU related techniques are widely used to identify other performance problems because of less than 10% overhead, such as memory system behavior and data locality. Itzkowitz et. al. introduced the memory profiling to Sun ONE Studio, which can collect and analyze memory accesses in sequential programs and report measurement data related to annotated code segment [15]. Buck and Hollingsworth developed Cache Scope to perform data-centric analysis by using Itanium2 event address registers (EAR) [5]. HPCToolkit [21] uses Intel PEBS and AMD IBS to associate memory access latency with both static and heap allocated data objects. ArrayTool [22] further provides optimization guidance for array regrouping using the latency metric.

However, these general-purpose tools can only point out data objects suffering from high access latency, but they do not tell whether the high latency comes from false sharing or not. In contrast, CHEETAH can correctly identify false sharing problems and associate with problematic data objects.

7. Conclusion

In this paper, we present CHEETAH, a lightweight profiler that identifies false sharing in multithreaded programs. CHEETAH is the first tool that can quantify the optimization potential, without the need of actually fixing a problem. For false sharing detection, CHEETAH distinguishes true and false sharing, and only reports problems that can significantly impact the whole program performance. CHEETAH provides insightful guidance for fixing problems. Experimental results show that CHEETAH can successfully identify serious false sharing problems and accurately assess the potential performance gain of each problem. CHEETAH is ready for real deployment, which is available for download at http://github.com/XXXXX.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM Press.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 114–124, New York, NY, USA, 2001. ACM.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *SEDMS IV: USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, Berkeley, CA, USA, 1993. USENIX Association.

[5] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the Intel ltanium 2 processor. In *SC '04: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, page 58, Washington, DC, USA, 2004. IEEE Computer Society.

[6] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf, November 2007. Last accessed: Dec. 13, 2013.

[7] Gprof community . Gnu gprof. https://sourceware.org/binutils/docs/gprof/.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[9] S. M. Günther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33, New York, NY, USA, 2009. ACM.

[10] Intel. Using the rdtsc instruction for performance monitoring. https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[11] Intel Corporation. Intel VTune performance analyzer. http://www.intel.com/software/products/vtune.

[12] Intel Corporation. *Intel Performance Tuning Utility 3.2 Update*, November 2008.

[13] Intel Corporation. Avoiding and identifying false sharing among threads. http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/, February 2010.

[14] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, Part 2, Number 253669-032, June 2010.

[15] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *SC '03: Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.

[16] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu. Detection of false sharing using machine learning. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 30:1–30:9, New York, NY, USA, 2013. ACM.

[17] J. Levon and P. Elie. Oprofile: A system profiler for Linux, 2004.

[18] C.-L. Liu. False sharing analysis for multithreaded programs. Master's thesis, National Chung Cheng University, July 2009.

[19] 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, New York, NY, USA, 2011. ACM.

[20] T. Liu, C. Tian, H. Ziang, and E. D. Berger. Predator: Predictive false sharing detection. In *Proceedings of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'14, New York, NY, USA, 2014. ACM.

[21] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2013.

[22] X. Liu, K. Sharma, and J. Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 405–416, New York, NY, USA, 2014. ACM.

[23] X. Liu and B. Wu. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proc. of the 2015 ACM/IEEE Conference on Supercomputing*, Austin, TX, USA, 2015.

[24] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield. Whose cache line is it anyway?: operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 141–154, New York, NY, USA, 2013. ACM.

[25] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[26] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 335–348, New York, NY, USA, 2010. ACM.

[27] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[28] M. Schindewolf. Analysis of cache misses using SIMICS. Master's thesis, Institute for Computing Systems Architecture, University of Edinburgh, 2007.

[29] W. Wang, T. Dey, J. Davidson, and M. Soffa. DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 380–391, Feb 2014.

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

[31] B. Wicaksono, M. Tolubaeva, and B. Chapman. Detecting false sharing in openmp applications using the darwin framework. In S. Rajopadhye and M. Mills Strout, editors, *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 283–297. Springer Berlin Heidelberg, 2013.

[32] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *The International Conference on Virtual Execution Environments*, Newport Beach, CA, Mar 2011.