Cheetah: Detecting False Sharing Efficiently and Effectively

Tongping Liu *
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249 USA
Tongping.Liu@utsa.edu

Xu Liu *
Department of Computer Science
College of William and Mary
Williamsburg, VA 23185 USA
xl10@cs.wm.edu

Abstract

False sharing is a notorious performance problem that may occur in multithreaded programs when they are 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 performance overhead or do not provide adequate information to guide code optimization.

To address these problems, we develop `Cheetah`, a profiler that detects false sharing both efficiently and effectively. `Cheetah` leverages the lightweight hardware performance monitoring units (PMUs) that are available in most modern CPU architectures to sample memory accesses. `Cheetah` develops the first approach to quantify the optimization potential of false sharing instances without actual fixes, based on the latency information collected by PMUs. `Cheetah` precisely reports false sharing and provides insightful optimization guidance for programmers, while adding less than 7% runtime overhead on average. `Cheetah` is ready for real deployment.

Categories and Subject Descriptors D.2.8 [*SOFTWARE ENGINEERING*]: Metrics–Performance measures; D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming–Parallel Programming

General Terms Measurement, Performance

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

* Both Tongping Liu and Xu Liu are co-first authors.

1. Introduction

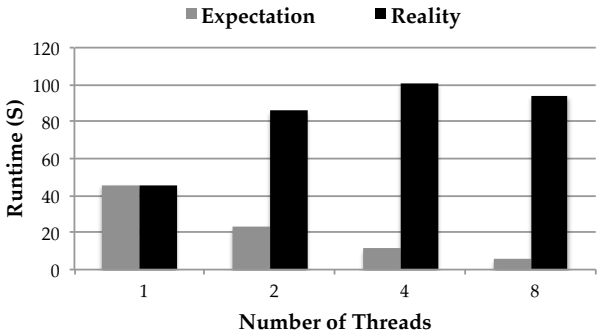
Multicore processors are ubiquitous in the computing spectrum: from smart phones, personal desktops, to high-end servers. Multithreading is the de-facto programming model to exploit the massive parallelism of modern multicore architectures. However, multithreaded programs may suffer from various performance issues caused by 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 in the same cache line. When a thread modifies the data of a cache line, the cache coherence protocol (managed by hardware) automatically invalidates the duplicates of this cache line residing in the private caches of other cores [25]. Thus, even if other threads access completely different words inside this cache line, they have to 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 illustrates this catastrophic performance issue. We meant to employ multiple threads to accelerate the computation. However, when eight threads (on an eight-core machine) simultaneously access adjacent elements of `array` sharing the same cache line, this program runs $\sim 13\times$ slower (black bars) than its linear-speedup expectation (grey bars). The hardware trend, such as adding more cores on chip and enlarging the cache line size, will further degrade the performance of multithreaded programs due to false sharing.

Unlike true sharing, false sharing is generally avoidable. When multiple threads unnecessarily share the same cache line, we can add byte paddings or utilize thread-private variables so that different threads can be forced to access different cache lines. Although the solution of fixing false sharing problems is somewhat straightforward, finding them is difficult and even impossible with manual checking, especially for a program with thousands or millions of lines of code.

```
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) A Program with False Sharing



(b) (b) Performance Degradation

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

Thus, it is 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 detectors fall short in several ways. First, most tools cannot distinguish true and false sharing, or require substantial manual effort to identify optimization opportunities [9, 12, 13, 16, 18, 24, 26, 30, 31]. Second, tools [9, 18, 20, 28, 32] based on memory access instrumentation introduce high runtime overhead, hindering their applicability to real, long-run applications. Third, some tools either require OS extensions [24], or only work on special applications [19]. Fourth, no prior tools assess the performance gain from eliminating an identified false sharing bottleneck. Without this information, many optimization efforts may yield trivial or no performance improvement [19, 32].

`Cheetah` is designed to address all these issues with the following two contributions:

- **The First Approach to Predict False Sharing Impact.** This paper introduces the first approach to predict the potential performance impact of fixing false sharing instances, without actual fixes. 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.
- **An Efficient and Effective False Sharing Detector.** `Cheetah` is an efficient false sharing detector, with only ∼7% performance overhead. `Cheetah` utilizes the performance monitoring units (PMUs) that are available in modern CPU architectures to sample memory accesses. `Cheetah` provides sufficient information on false sharing problems, by pointing out the lines of code, names of variables, and detailed memory accesses involved in the false sharing. `Cheetah` is a runtime library that is very convenient to be deployed; there is no need for a custom OS, nor recompilation and changing of programs.

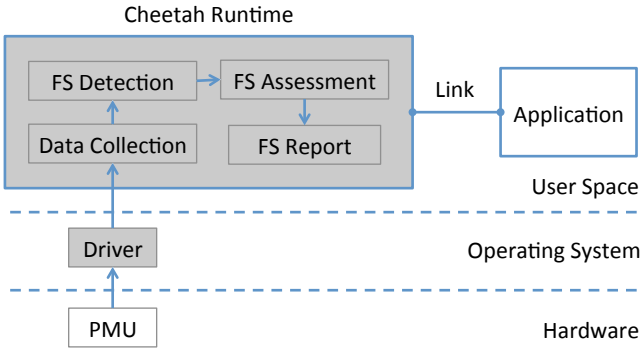


Figure 2. Overview of `Cheetah`'s components (shadow blocks), where "FS" is the abbreviation for false sharing.

Figure 2 shows the overview of `Cheetah`. The "data collection" module gleans memory accesses via the address sampling supported by the hardware performance monitoring units (PMUs) and, with the assistance of the "driver" module, filters out memory accesses associated with heap or globals, feeding them into the "FS detection" module. At the end of an application, or when `Cheetah` is requested 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 a significant performance impact, along with its predicted performance improvement after fixes provided by the "FS assessment" module.

The remainder of this paper is organized as follows. Section 2 describes false sharing detection components of `Cheetah`. Section 3 discuesses how `Cheetah` assesses the performance impact of a false sharing instance. Section 4 presents experimental results, including effectiveness, performance overhead, and assessment precision. Section 5 addresses main concerns about hardware dependence, perfor-

mance, and effectiveness. Lastly, Section 6 discusses related work, and Section 7 concludes this paper.

2. Detecting False Sharing

`Cheetah` reports false sharing problems with a significant performance impact, where they will incur a large number of cache invalidations on corresponding cache lines. However, tracking cache invalidations turns out to be difficult because invalidations depend on the memory access pattern, cache hierarchy, and thread-to-core mappings. To address this challenge, `Cheetah` proposes a simple rule to compute cache invalidations: *when a thread writes a cache line that has been accessed by other threads recently, this write access incurs a cache invalidation*. This rule is based on the following two assumptions, which are introduced in prior work [20, 32].

- **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 the over-subscription of threads is generally rare for computation-intensive programs. But we may **overreport** the number of cache invalidations in the following situations: (1) if multiple threads are actually scheduled to the same physical core, or (2) different cores may share part of the cache hierarchy (instead of having private caches), or (3) hyper-threading technology is enabled such that multiple threads may share the same CPU. However, we argue that the problem of overreporting can actually cancel out the weakness of using the sampling technique. This assumption avoids the tracking of thread-to-core mapping, as well as knowledge of the actual cache hierarchy.

Assumption 2 further defines the behavior related to cache eviction. In reality, a cache entry will be evicted when the cache is not sufficient to hold all active data of an application. Without this assumption, we should track every memory access and simulate the cache eviction in order to determine the accurate number of cache invalidations, which is notoriously slow and suffers from limited precision [28]. As with the first assumption, assumption 2 may also overestimate the number of cache invalidations. Based on this assumption, if there is a memory access within a cache line, the hardware cache for a running thread always holds the data until an access issued by other threads (running on other cores, by assumption 1) invalidates it. By combining these two assumptions, `Cheetah` identifies cache invalidations simply based on memory access patterns, independent of the underlying architecture and specific execution conditions.

In the remainder of this section, we elaborate on how we track memory accesses in Section 2.1, how we locate a sampled access's cache line in Section 2.2, how we compute cache invalidation based on sampled memory accesses 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 in order to compute the number of cache invalidations that occur on each cache line. Software-based approaches may introduce higher than 5× performance overhead [20, 32]. High overhead can block people from using these tools in real deployment.

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

Since PMUs only sample one memory access out of a predefined number of accesses, this approach greatly reduces the runtime overhead incurred by performance data collection and analysis. Moreover, 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 (e.g., one out of 64K instructions), can identify false sharing with a significant performance impact.

Implementation. In order to sample memory accesses, `Cheetah` programs the PMU registers to turn on sampling before the `main` routine. It also installs a signal handler to collect detailed memory accesses. `Cheetah` configures the signal handler to respond to the current thread, by calling the `fcntl` function with the `F_SETOWN_EX` flag. This method avoids any lookup overhead and simplifies signal handling. Inside the signal handler, `Cheetah` collects detailed information for every sampled memory access, including its memory address, thread ID, read or write operation, and access latency, which can be fed into the "FS detection" module to compute the number of cache invalidations, as well as the "FS assessment" module to predict the performance impact.

2.2 Locating Problematic Cache Lines

For each sampled memory access, `Cheetah` decides whether this access causes a cache invalidation or not, and records this access. For this purpose, `Cheetah` should quickly locate its corresponding 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 when using the default heap. Thus, `Cheetah` builds its custom heap based on Heap Layers [2]. `Cheetah` pre-allocates a fixed-size memory block (using `mmap`), and satisfies all memory allocations from this block. `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-object false sharing, but also makes Cheetah unable to report problems that are possibly 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 the main routine. Cheetah maintains two different heaps: one for the application itself, as well as one for internal use. 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 track the number of writes and detailed access information on each cache line. For each memory access, Cheetah uses bit-shifting to compute the index of its cache line.

2.3 Computing Cache Invalidations

Prior work of Zhao et al. proposes an ownership-based method to compute the number of cache invalidations: when a thread updates a cache line owned by others, this access incurs an cache invalidation, and then resets the ownership to the current thread [32]. However, this approach cannot easily scale to more than 32 threads because of excessive memory consumption, since it needs one bit for every thread to track the ownership.

To address this problem, Cheetah maintains a two-entry table (T) for each cache line (L), in which each thread will, at most, occupy one of these two entries. In this table, each entry has two fields: a thread ID and an access type (read or write). It computes the number of invalidations according to the rule described in Section 2. In case of a cache invalidation, the current access (write) is added into the corresponding table. Thus, each table always has at least one entry. More specifically, Cheetah handles each access as follows:

- For each read access, Cheetah decides 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 records this read access in the table. Otherwise, there is no need to handle this read access.
- For each write access, Cheetah decides whether this access incurs an invalidation. If the table is already full, based on assumption 1, it incurs a cache invalidation, since at least one of the existing entries in this table is from a different thread. If the table is both not full, and not empty, Cheetah checks whether the existing entry is from a different thread or not. If this write access is from the same thread as the existing entry, Cheetah skips the current write access, since there is no need to update the existing entry. In all other cases, this write access incurs at least a cache invalidation. Currently, Cheetah does not differentiate how many reads have occurred prior to this write. In case of a cache invalidation, the table is flushed, and the write access is recorded in the table to maintain the table as not empty.

Implementation As aforementioned, only cache lines with a large number of writes can possibly have a high impact on performance. Based on this observation, cache lines with a small number of writes are never the cause of the severe performance degradation. For this reason, Cheetah first tracks the number of writes on a cache line, and only tracks detailed information for cache lines with more than two writes. This simple policy avoids tracking detailed information for write-once memory.

2.4 Reporting False Sharing

Cheetah reports false sharing correctly and precisely, either at the end of an execution, or when interrupted by the user.

Correct Detection. Cheetah tracks word-based (four byte) memory accesses on susceptible cache lines using the shadow memory technique: that is, the amount of reads or writes issued by a particular thread on each word. When more than one thread access a word, Cheetah marks this word to be shared by multiple threads. By identifying accesses on each word of a susceptible cache line, we can easily differentiate false sharing from true sharing, since multiple threads will access the same words in true sharing. Word-based information also helps programmers to decide how to pad a problematic data structure during fixing phases. It is very common that the main thread may allocate and initialize objects before they are accessed by multiple child threads. Prior work, including Predator [20], may wrongly report them as true sharing instances. Cheetah avoids this problem by only recording detailed accesses inside parallel phases.

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 symbol table in the binary executable. For heap objects, Cheetah reports the lines of code corresponding to their allocation sites. Thus, Cheetah intercepts all memory allocations and de-allocations to obtain the entire call stack. Cheetah does not monitor stack variables because they are normally accessed only by their hosting threads. It is noted that the default backtrace function in glibc is extremely slow due to expensive instruction analysis. Cheetah utilizes the frame pointers to fetch the call stack efficiently. Moreover, we only collect five function entries on the call stack for performance reasons.

3. Assessing the Performance Impact

Fixing false sharing does not necessarily yield significant performance speedups, even for instances with a large number of cache invalidations [19, 20]. Zhao et al. even observes

that fixing false sharing may even slow down a program, since padding data structures may increase its memory footprint or lose cache locality [32]. Thus, it is very important to rule out insignificant false sharing instances, which are not false-positives, yet reporting them increases the manual burden for fixes.

Cheetah makes the first attempt to quantitatively assess the potential performance gain of fixing a false sharing instance based on the results of an execution. We agree that different executions may vary on the specific details, but this should not change the overall prediction result: whether a false sharing instance is significant or not. Actually, the evaluation described in Section 4.3 confirms that our predicted performance has less than a 10% difference from that of the actual fixes. Based on this prediction, programmers can focus on severe problems only, avoiding unnecessary manual effort spent on insignificant cases.

Cheetah's assessment is based on the following observations:

- **Observation 1:** Samples are evenly distributed over the whole execution. Based on this, we can use the sampling technique to represent the whole execution. The similar idea has been widely used by prior work, such as Oprofile [17] and Gprof [8] to identify the hotspots of function calls.
- **Observation 2:** The PMU provides the latency information (e.g. cycles) of each memory access; the latency of memory accesses with false sharing are significantly higher than that of other accesses.

Based on these two observations, we propose to use the sampled cycles to represent the whole execution, and further predict the performance impact of falsely-shared objects by replacing these cycles with the average cycles of memory accesses without false sharing. The assessment is performed in three steps, listed as follows:

- Cheetah first predicts the possible cycles after fixes by replacing actual cycles with the average cycles of memory accesses without false sharing, as discussed in Section 3.1.
- Then, Cheetah assesses the performance impact of fixes on the related threads, which is discussed in Section 3.2.
- In the end, Cheetah assesses the performance impact on the application in Section 3.3.

The remainder of this section discusses the detailed assessment step-by-step. For reasons of simplicity, we abbreviate the falsely-shared object as "*O*", the related thread as "*t*", the prediction as "*Pred*", the runtime as "*RT*", and the application as "*App*".

3.1 Impact on Accesses to the Object

At first, Cheetah predicts the possible cycles of accesses after fixing false sharing of the object *O*. Cheetah tracks the number of cycles and accesses on each word, thus it is convenient to compute the total cycles of accesses — $Cycles\_O$, and the total number of accesses — $Accesses\_O$, on a specific object *O*.

However, it is impossible to know the average cycles of every access after fixing — $AverCycles\_nofs$, without actual fixes (*Cheetah utilizes the average cycles in serial phases (AverCycles_serial) to approximate this value*). There is no false sharing in serial phases, and *AverCycles_serial* represents the least number of cycles for memory accesses after fixes. If Cheetah does not track any accesses in serial phases, a default value learned from experience will be utilized as *AverCycles_serial*. In reality, $AverCycles\_nofs$ can be larger than *AverCycles_serial*, since fixing false sharing may lead to excessive memory consumption or the loss of locality [32]. Cheetah actually predicts the best performance of fixing a false sharing instance.

Cheetah computes the total cycles of accesses after fixes (*PredCycles_O*) based on the EQ.(1). It is expected that *PredCycles_O* will be less than the total cycles before fixing — $Cycles_O$, since fixing a false sharing problem will reduce the execution time and cycles.

$$PredCycles\_O = (AverCycles\_nofs * Accesses\_O) \quad (1)$$

3.2 Impact on Related Threads

The second step is to assess how reducing the access cycles of *O* (*PredCycles_O*) can potentially affect the execution time of its related threads.

Cheetah collects the following runtime information of every thread: the execution time — RT_t, the total number of accesses — $Accesses\_t$, and the total cycles of all memory accesses — $Cycles\_t$. In order to avoid any lookup overhead, Cheetah lets every thread handle the sample events of the current thread, and records the corresponding number of accesses and cycles. To collect RT_t, Cheetah intercepts the creation of threads by passing a custom function as the start routine. Cheetah acquires the timestamp before and after the execution of a thread using the RDTSC (ReaD-Time Stamp Counter) [10], and regards the difference to be the execution time of a particular thread. In current implementation, we do not take into account the waiting time of different threads caused by synchronizations; we leave this for future work.

After the collection, Cheetah predicts the cycles of every related thread — $PredCycles\_t$ — after fixes as the EQ.(2).

$$PredCycles\_t = Cycles\_t - Cycles\_O + PredCycles\_O \quad (2)$$

Based on $PredCycles\_t$, Cheetah assesses the predicted runtime of a thread — $PredRT_t$ — as the EQ.(3). We assume that **the execution time is proportional to the cycles of**

accesses, such that fewer cycles indicates less execution time, and corresponds to a performance speedup. It is expected that fixing the false sharing problem inside the object O will improve the performance for its related threads, with less $PredCycles\_t$ and $PredRT_t$.

$$PredRT\_t = (PredCycles\_t / Cycles\_t) * RT\_t \quad (3)$$

3.3 Impact on the Application

In the end, `Cheetah` assesses how fixes will change the performance of the application.

Actually, improving the performance of a thread may not increase the final performance of an application if this thread is not in the critical path. To simplify the prediction, as well as 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 actuality. All applications that we have evaluated in this paper utilize this fork-join model. The performance assessment will be more complicated if nested threads are utilized inside an application.

`Cheetah` tracks the 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]. In the fork-join model, shown in Figure 3, an application leaves a serial phase after the creation of a thread; it leaves a parallel phase after all child threads (created in the current phase) have been successfully joined.

Based on the runtime information of every parallel and serial phase, `Cheetah` assesses 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 of an application is equal to the sum of different parallel and serial phases.

After `Cheetah` computes the possible execution time of an application after fixing a false sharing problem, `Cheetah` will compute and report the potential performance improvement of every falsely-shared object, based on EQ.(4). Then, programmers can focus on those with the most serious performance impact. We further verify the precision of `Cheetah`'s assessment in Section 4.3.

$$PerfImprove = RT\_App / PredRT\_App \quad (4)$$

4. Evaluation

The evaluation answers the following research questions:

- What is the performance overhead of `Cheetah`? (4.1)
- How effectively can `Cheetah` detect false sharing problems? How helpful are the outputs in fixing false sharing problems? (4.2)
- What 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. Because the machine is a NUMA machine, and the performance may vary with different scheduling policies, we bind the threads to cores in order to acquire consistent performance.

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 for sufficiently long time, as `Cheetah` needs enough samples to detect false sharing problems. Basically, we want to make every application run for at least 5 seconds in order to collect enough samples. For PARSEC benchmarks, we are utilizing the `native` input. For some applications of `Phoenix`, such as `linear_regression`, we explicitly change the source code by adding more loop iterations.

4.1 Runtime Overhead

We show the average runtime overhead of `Cheetah` in Figure 4. We run each application five times and show the average results here. These results are normalized to the execution time of `pthreads`, which means that an application is running slower if the value is higher. According to this figure, `Cheetah` only introduces around 7% performance overhead, which makes it practical to be utilized for real deployment.

During the evaluation, we configure `Cheetah` with a sampling frequency of one out of 64K instructions. Thus, for every 64K instructions, the trap handler is notified once so that `Cheetah` can collect the information relating to memory accesses on heap and global variables. Currently, `Cheetah` filters out those memory accesses in kernel, libraries or others.

The performance overhead of `Cheetah` mainly comes from the handling of each sampled memory access and each thread creation. For each sampled access, we collect information — such as the type of access (read or write) and the number of cycles — then update the history table of its corresponding cache line. `Cheetah` also intercepts every thread creation in order to setup the PMU unit, get the timestamp, and update the phase information. For applications with a large number of threads, including `kmeans` (with 224 threads in 14 seconds) and `x264` (with 1024 threads in 40 seconds), setting PMU registers introduces non-negligible overhead, since it invokes six `pfmon` APIs and six additional

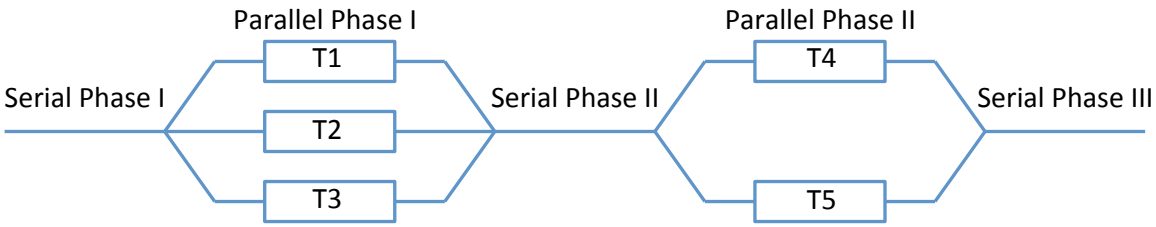


Figure 3. The fork-join model, currently supported by `Cheetah` to assess the performance impact of false sharing instances.

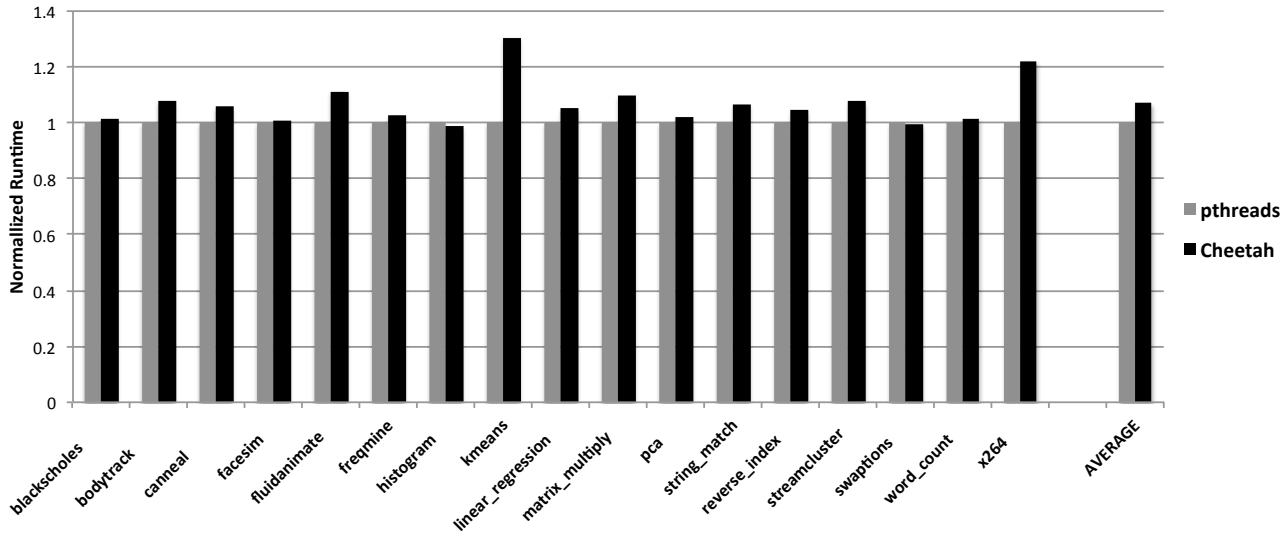


Figure 4. Runtime overhead of `Cheetah`. We normalize the runtime to that of native execution without monitoring by `Cheetah`. On average, `Cheetah` only introduces around 7% performance overhead on all evaluated applications, which makes its use practical for real deployment.

system calls. For other applications, `Cheetah` introduces less than 12% performance overhead, with 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.

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 word-based accesses that are reported by `Cheetah`, but not shown here, we can understand the reason for this false sharing problem: different threads are updating different parts of the object `tid_args` simultaneously, where each thread updates words with the size of the structure lreg_args. 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 of useless content, we can force different threads to not access the same cache line. This one-line code change leads to a 5.7× speedup of the performance, which matches the assessment of 5.76× improvement predicted by `Cheetah`.

4.2.2 Case Study: streamcluster

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

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`.

```
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`.

4.2.3 Comparing with State-of-the-art

Predator is the state-of-the-art in false sharing detection, which detects the highest number of instances, but with approximately 6× performance overhead [20]. Compared to Predator, `Cheetah` misses false sharing problems in `histogram`, `reverse_index` and `word_count` [20].

As discussed before, `Cheetah` only detects actual false sharing problems that may have significant performance impact on final performance. If the number of accesses on a falsely-shared object is not large enough, `Cheetah` may not be able to detect it due to its sampling feature. Additionally, occurrences of false sharing can be affected by the starting address of objects, the size of the cache line, or by the cache hierarchy, as observed by Predator [20]. Thus, we further check the seriousness of these problems based on Predator's detection results.

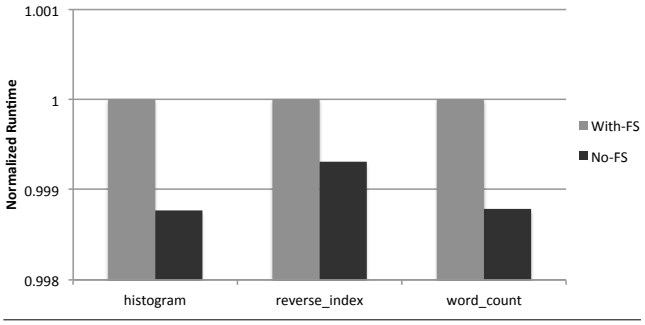


Figure 7. False sharing problems missed by `Cheetah` have negligible (<0.2%) performance impact.

We run these applications on our experimental hardware, with and without false sharing problems. Figure 7 shows the performance impact. Actually, 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`: because `Cheetah` only reports false sharing problems with significant performance impact, it can potentially save programmers manual effort unnecessarily spent on applications capable of only negligible performance improvement.

4.3 Assessment Precision

`Cheetah` is the first tool that can assess the performance impact of false sharing problems. Based on this information, programmers may save huge amounts of manual effort spent unnecessarily on applications with trivial false sharing problems.

We evaluate the precision of assessment on two applications that are reported to have false sharing problems: `linear_regression` and `streamcluster`. We list the precision results in Table 1. In this table, `linear_regression` is abbreviated as "`linear_reg`". We evaluate these 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 ("Diff") 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.

Table 1 shows that `Cheetah` can perfectly assess the performance impact of false sharing in every case, with less than 10% difference for every evaluated execution.

5. Discussion

This section addresses some possible concerns related to `Cheetah`.

Table 1. Precision of assessment.

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

Hardware Dependence. Cheetah is an approach that relies on hardware PMUs to sample memory accesses. To use Cheetah, users should setup the driver to enable the PMU-based sampling beforehand. Afterward, they can connect to the Cheetah library by calling only two APIs: one API is to setup PMU-based registers, while the other handles every sampled memory access, with less than 5 lines of code change.

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 that having a large number of threads, because Cheetah should monitor thread creations and setup hardware registers for every thread. However, this should not be of large concern when using Cheetah. First, the creation of a large number of threads in an application is atypical. Secondly, we expect this overhead could be further reduced with improved hardware support.

Effectiveness. Cheetah effectively detects false sharing problems that occur in the current execution, and have high impact on performance. For effective detection, Cheetah requires programs to run sufficiently long, perhaps more than few seconds. This should not be a problem for long-running applications, which are the primary targets of optimizations.

6. Related Work

In this section, we review existing tools for the detection of false sharing issues, as well as other techniques of utilizing PMUs for dynamic analysis.

Cheetah is the first work to predict the potential performance improvement after fixing false sharing problems, relying on access latency information provided by the PMU hardware. Existing work utilizes the latency information to identify variables and instructions suffering from high access latency [21, 22].

6.1 False Sharing Detection

Existing tools dealing with 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 may report possible false sharing problems within programs [28]. Simulation-based approaches generally introduce more than 100× performance overhead and cannot simulate large applications.

Instrumentation-Based Approaches. Tools in this category can be further divided into dynamic instrumentation-based approaches [9, 18, 32], and static-based or compiler-based approaches [20]. These approaches generally have large performance overhead, running from 5× slower [20, 32] to 100× slower [9, 18]. Predator is the state-of-the-art tool in false sharing detection, and uncovers the largest number of false sharing instances [20]. However, their performance overhead is still too high to be used in deployed software. Cheetah utilizes a different approach than Predator, and successfully reduces the detection overhead.

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

PMU-based approaches. *PMU-based tools* are introduced due to performance reasons. All existing PMU-based approaches actually detect false sharing by monitoring cache related events. Jayasena et al. use the machine learning approach to derive the potential pattern of false sharing by monitoring cache misses, TLB events, interactions among cores, and resources stalls [16]. DARWIN collects cache coherence events during the first round, then identifies possible memory accesses on data structures with frequent cache invalidations during the second round [30]. Intel's PTU relies on the PEBS mechanism to track cache invalidation-related memory accesses, but it cannot differentiate false sharing and true sharing [12]. However, existing PMU-based tools suffer from the following problems: (1) they cannot report all existing false sharing problems due to sampling on very rare cache events (e.g. cache invalidations) [16, 26, 30] (otherwise they experience a large quantity of false positives [12]); (2) they rely on manual annotation [26], experts' expertise [12, 30], or multiple executions to locate the problem [30]; (3) they cannot provide sufficient information for optimization [12, 16, 26, 30].

Cheetah specifically addresses these existing problems, and provides much richer information for optimization, including word-level accesses and potential performance impact after fixes. Also, Cheetah provides better effectiveness

than existing PMU-based approaches since it samples much richer events such as memory accesses.

6.2 Other PMU-Related Analysis

PMU-related techniques are widely used to identify other performance problems due to their low (less than 10%) overhead, such as memory system behavior and data locality. Itzkowitz et al. introduce 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 develop Cache Scope to perform data-centric analysis using Itanium2 event address registers (EAR) [5]. HPCToolkit [21] and ArrayTool [22] use AMD IBS to associate memory access latency with both static and heap-allocated data objects, and further provide optimization guidance for array regrouping. However, these general-purpose tools can only, at best, identify data objects suffering from high access latency. They do not determine whether the high latency originates from false sharing, nor can they provide rich information to assist with optimizations. In contrast, `Cheetah` can correctly identify false sharing problems, as well as associate detailed memory accesses with problematic data objects to help optimizations.

7. Conclusion

This paper presents `Cheetah`, a lightweight profiler that identifies false sharing in multithreaded programs. `Cheetah` employs the first approach that quantifies the optimization potentials of fixing false sharing instances, without actual fixes. For detection, `Cheetah` distinguishes true and false sharing, and only reports problems that significantly impact overall program performance. `Cheetah` provides insightful guidance for fixing problems while only introducing 7% runtime overhead, making it ready for real deployment.

Acknowledgements

This material is based upon work supported by startup packages provided by University of Texas at San Antonio and College of William and Mary, and Google Faculty Research Award. This research was also supported by the National Science Foundation (NSF) under Grant No. 1464157. We would like to thank Sam Silvestro, Jinpeng Zhou, Hongyu Liu, and Jin Han for their invaluable comments and suggestions that helped improve this paper. Finally, this paper integrates the implementation of Guangming Zeng on acquiring callstacks efficiently.

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, 1997.

[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.