Precise Detection and Automatic Mitigation of False Sharing

Tongping Liu Emery D. Berger

Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{tonyliu,emery}@cs.umass.edu

Abstract

False sharing is an insidious problem for multithreaded programs running on multicore processors, where it can silently degrade performance and scalability. Previous tools for detecting false sharing are severly limited: they cannot distinguish false sharing from true sharing, have high false positive rates, and provide limited assistance to help programmers locate and resolve false sharing.

This paper presents two tools that attack the problem of false sharing: FS-DETECTIVE and FS-PATROL. Both tools leverage a framework we introduce here called SHERIFF. SHERIFF breaks out threads into separate processes, and exposes an API that allows programs to perform per-thread memory isolation and tracking on a per-page basis. We believe SHERIFF is of independent interest.

FS-DETECTIVE finds instances of false sharing by comparing updates within the same cache lines by different threads, and uses sampling to rank them by performance impact. FS-DETECTIVE is precise (no false positives), runs with low overhead (on average, 20%), and is accurate, pinpointing the exact objects involved in false sharing. We present a case study demonstrating FS-DETECTIVE's effectiveness at locating false sharing in a variety of benchmarks.

Rewriting a program to fix false sharing can be infeasible when source is unavailable, or undesirable when padding objects would unacceptably increase memory consumption or further worsen runtime performance. FS-PATROL mitigates false sharing by adaptively isolating shared updates from different threads into separate physical addresses, effectively eliminating most of the performance impact of false sharing. We show that FS-PATROL can improve performance for programs with catastrophic false sharing by up to $9\times$, without programmer intervention.

1. Introduction

Writing multithreaded programs can be challenging. Not only must programmers cope with the difficulty of writing concurrent programs (races, atomicity violations, and deadlocks), they must also make sure they are efficient and scalable. Key to achieving high performance and scalability is reducing contention for shared resources. However, threads can still suffer from *false* sharing when multiple objects that are not logically shared reside on the same cache line. When false sharing is frequent enough, the resulting "pingponging" of cache lines across processors can cause performance to plummet [5, 11]. False sharing can degrade application performance by as much as an order of magnitude. Two trends—the prevalence of multicore architectures and the expected increase in the number of multithreaded applications in broad use, and increasing cache line sizes—are likely to make false sharing increasingly common.

Locating false sharing requires tool support. Past false sharing detection tools operate on binaries, either via simulation [22] or binary instrumentation [10, 18, 24], and intercept all reads and writes (leading to slowdowns of up to $200\times$), or rely on hardware performance monitors and correlate cache invalidations with function calls [12, 13]

These false sharing detection tools suffer from problems that greatly limit their usefulness. They generate numerous false positives: addresses that appear shared but are just reused by the memory allocator, instances of true sharing, and objects that were falsely shared so few times that they do not present a performance bottleneck. They also provide little actionable information for programmers seeking to resolve false sharing in their programs. Their reports range from a list of suspicious memory addresses with functions that accessed them at some point [12, 22] to a single number representing the overall false sharing rate for the entire program [10, 18, 24].

This paper presents two tools designed to help programmers effectively address the challenges of locating and dealing with false sharing in multithreaded applications. FS-DETECTIVE finds and reports false sharing accurately (no false positives) and precisely, indicating the exact objects responsible for false sharing. When rewriting an application to

resolve false sharing is infeasible (because source is unavailable, or padding data structures would unacceptably reduce cache utilization and/or increase memory consumption), FS-PATROL can be used as a runtime system that eliminates false sharing automatically. Both tools leverage a common framework we introduce here called SHERIFF that enables them to operate efficiently.

Figure 1 presents a sample workflow for using FS-DETECTIVE and FS-PATROL. A multithreaded program is first executed with FS-DETECTIVE to uncover any false sharing. The programmer can then act on FS-DETECTIVE's reports to correct false sharing by applying padding or aligning objects to cache line boundaries. If these fixes resolve the problem, then the modified program can be used with the standard `pthreads` library. If the fixes degrade performance or introduce excessive memory consumption [24], or when it is impractical or impossible to modify the program, then FS-PATROL can be used as a substitute runtime system to automatically eliminate false sharing.

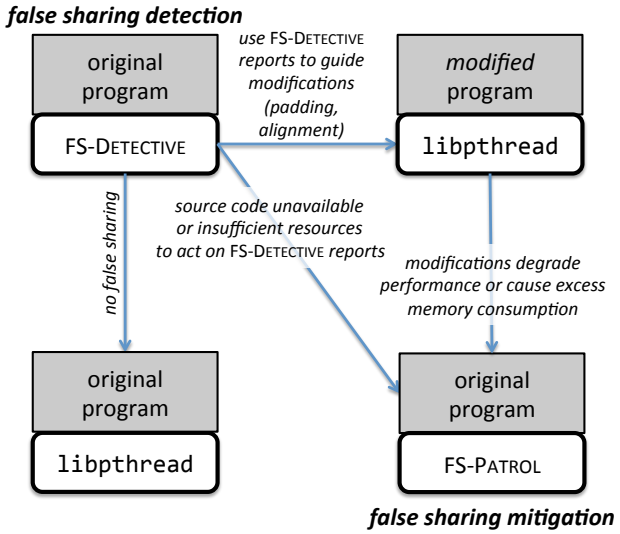


Figure 1. A workflow for using FS-DETECTIVE and FS-PATROL.

Contributions

This paper makes the following contributions:

- It presents **SHERIFF**, a software-only framework that replaces the standard `pthreads` library and transforms threads into OS processes. It exposes an API that enables *per-thread memory protection* and optional *memory isolation* on a per-page basis. We believe SHERIFF is of independent interest since it enables a range of possible applications, including language support and enforcement of data sharing, software transactional memory, thread-level speculation, and race detection [19], though we use SHERIFF here to build two tools focused on false sharing.
- It presents **FS-DETECTIVE**, a tool that finds and reports false sharing with high precision and with no false positives. It only reports actual false sharing—not true sharing, and not artifacts from heap object reuse. It uses sampling to rank instances of false sharing by their potential performance impact. FS-DETECTIVE pinpoints false sharing locations by indicating offsets and global variables or heap objects (with allocation sites), making false sharing relatively easy to locate and correct. FS-DETECTIVE is generally efficient: on average, it slows down execution time by just 20%.
- It presents **FS-PATROL**, a tool that eliminates false sharing automatically without the need for code modifications or recompilation. FS-PATROL can dramatically increase performance in the face of false sharing sharing. To our knowledge, FS-PATROL is the first false sharing resistant runtime for shared-memory multithreaded programs.

We evaluate FS-DETECTIVE and FS-PATROL over a range of applications, including the Phoenix and PARSEC benchmark suites; the latter is designed to be representative of next-generation shared-memory programs for chip-multiprocessors [4]. We show that FS-DETECTIVE can successfully guide programmers to the exact sources of false sharing, and use its results to remove false sharing in several applications. We also apply FS-PATROL to automatically mitigate false sharing in these applications; in one case, an applicaton suffering from catastrophic false sharing runs 9× faster with FS-PATROL than with the standard `pthreads` library.

Outline

The remainder of this paper is organized as follows. Section 2 describes key related work. Section 3 describes SHERIFF's mechanisms and algorithms in detail. Section 4 describes the FS-DETECTIVE tool and how it works to locate false sharing problems. Section 5 presents the FS-PATROL runtime system. Section 6 presents experimental results, including several case studies of using FS-DETECTIVE to detect and guide the correction of false sharing. Finally, Section 7 concludes.

2. Related Work

The proliferation of multicore systems has increased interest in tool support to detect false sharing, since standard profilers like OProfile [17] or gprof [9] only report overall cache misses.

Simulation and Instrumentation Approaches: Schindewolf describes a system based on the SIMICS functional simulator that reports full cache miss information, including invalidations caused by sharing [22]. Pluto uses the Valgrind framework to track the sequence of load and store events on different threads and reports a worst-case estimate of possible false sharing [10]. Similarly, Liu uses Pin to collect

memory access information and then reports total false sharing miss information [18].

Independently and in parallel with this work, Zhao et al. developed a tool designed to detect false sharing and other sources of cache contention in multithreaded applications [24]. This tool uses shadow memory to track ownership of cache lines and cache access patterns. It is currently limited to at most 8 simultaneous threads. Like Liu above, it only reports an overall false sharing rate for the whole program, placing the burden on programmers to examine the entire source base to locate any instances of false sharing.

Unlike FS-DETECTIVE, these systems generally suffer from high performance overhead ($5-200\times$ slower) or memory overheads. They cannot differentiate true sharing from false sharing, yielding numerous false positives. Because they operate at the binary level, they can all be misled by aliasing due to memory object reuse. Finally, and most importantly, they do not point to the objects responsible for false sharing, limiting their value to the programmer.

Sampling-Based Approaches: Intel's performance tuning utility (PTU) [12, 13] uses event-based sampling, allowing it to operate efficiently. PTU can discover shared physical cache lines, and can identify possible false sharing at the grain of individual function calls. PTU suffers from a high false positive rate caused by aliasing due to reuse of heap objects, and reports false sharing instances that have no impact on performance. PTU cannot differentiate true from false sharing or pinpoint the source of false sharing problems, unlike FS-DETECTIVE. Section 6 contains an extensive empirical comparison of PTU to FS-DETECTIVE demonstrating PTU's relative shortcomings.

Pesterev et al. describe DProf, a tool that leverages AMD's instruction-based sampling hardware to help programmers identify the sources of cache misses [20]. 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. By contrast, FS-DETECTIVE is architecture independent, requires no manual intervention, and precisely identifies false sharing regardless of the flow of data or which data types are involved.

False Sharing Avoidance: In some restricted cases, it is possible to eliminate false sharing, obviating the need for detection. Jeremiassen and Eggers describe a compiler transformation that adjusts the memory layout of applications through padding and alignment [14]. Chow et al. describe an approach that alters parallel loop scheduling to avoid sharing [7]. The effectiveness of these static analysis based approaches is primarily limited to regular, array-based scientific codes, while FS-PATROL can prevent false sharing in any application.

Berger et al. describe Hoard, a scalable memory allocator can reduce the likelihood of false sharing of distinct heap objects [1]. Hoard limits accidental false sharing of entire heap objects by making it unlikely that two threads will use the same cache lines to satisfy memory requests, but this has no effect on false sharing within individual heap objects, which FS-PATROL avoids.

Other Related Work: SHERIFF borrows the process-as-thread model pioneered by Grace [2], but otherwise differs from it in its semantics, generality, and goals.

Grace is a process-based approach designed to prevent concurrency errors, such as deadlock, race conditions, and atomicity errors by imposing a sequential semantics on speculatively-executed threads. Grace supports only fork-join programs without inter-thread communication (e.g., condition variables or barriers), and rolls back threads when their effects would violate sequential semantics.

SHERIFF extends Grace to handle arbitrary multithreaded programs; for example, Grace is incompatible with any applications that employ inter-thread communication, including the PARSEC benchmarks we examine here. SHERIFF applies diffs at synchronization points in the program to enable updates without rollback, giving it far greater performance (but different semantics) than Grace. SHERIFF does not eliminate concurrency errors, but instead allows applications to selectively track updates and isolate memory, enabling tools like FS-DETECTIVE and FS-PATROL.

3. SHERIFF

SHERIFF is a functional replacement for the `pthreads` library that extends it with two novel features: *per-thread memory protection* (allowing each thread to track memory accesses independently of each other thread's accesses) and optional *memory isolation* (allowing each thread to read and write memory without interference from other threads). SHERIFF works through a combination of *replacing threads by processes* [2] and *page-level twinning and diffing* [6, 15].

Replacing `pthreads` with processes is surprisingly inexpensive, especially on Linux where both `pthreads` and processes are invoked using the same underlying system call. Process invocation can actually outperform threads because operating systems initially assign threads to the invoking CPU to maximize locality, while it spreads processes across all CPUs [2]. To achieve the effect of shared memory, SHERIFF maps globals and the heap into a shared region (Section 3.1). It also intercepts the `pthread_create` call and replaces it with a process creation call (Section 3.2).

Using processes to simulate threads has two key advantages. First, converting threads to processes enables the use of per-thread page protection, allowing SHERIFF to track memory accesses by different threads, a feature that FS-DETECTIVE uses in its false sharing detection algorithm (Section 4). Second, it isolates threads' memory accesses from each other. This isolation ensures that threads do not update shared cache lines, and because each process naturally has its own distinct set of cache lines this eliminates false sharing: FS-PATROL takes advantage of this feature (Section 5).

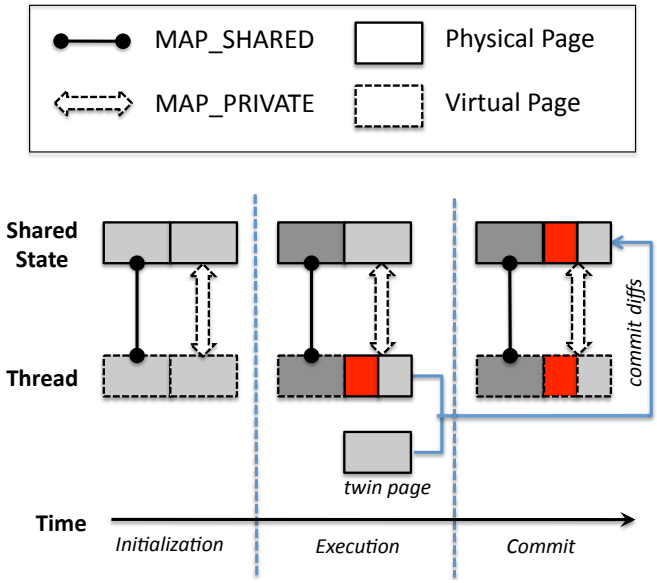


Figure 2. SHERIFF replaces `pthreads` by simulating threads with processes. It exposes an API that enables per-thread memory protection and memory isolation on a per-page basis. Each "thread" thus either operates directly on shared memory, or on its own private copy. For the latter, SHERIFF commits diffs to shared mappings at synchronization points (Section 3.3).

To maintain the shared-memory semantics of a multi-threaded program, SHERIFF must periodically update the shared heap and globals so that modifications become visible to other threads. SHERIFF delays these updates until synchronization points such as lock acquisition and release (Section 3.3). However, SHERIFF takes care to update only the changed parts of each page (Section 3.4).

3.1 Simulating a Shared Address Space

To create the effect of multi-threaded programs where different threads share the same address space, SHERIFF uses memory mapped files to share the heap and globals across different processes. Note that SHERIFF does not share the stack across different processes because different threads have their own stacks and, in general, multithreaded programs do not use the stack for cross-thread communication.

SHERIFF creates two different mappings for both the heap and the globals. One is a shared mapping, which is used to hold shared state. The other is a private, copy-on-write (COW) per-process mapping that each process works on directly. Private mappings are linked to the shared mapping through the one memory mapped file. Reads initially go directly to the shared mapping, but after the first write operation, both reads and writes are entirely private. SHERIFF updates the shared image at synchronization points, as described in Section 3.5.

SHERIFF uses a fixed-size mapping to hold globals, which it checks to ensure is large enough to hold all globals. SHERIFF also uses a fixed-size mapping to store the heap (by default, 1GB). Memory allocation requests from user applications are satisfied from this fixed-size private mapping.

Since different threads allocate memory from this single fixed-size mapping, the global superheap data structure is shared among different threads and allocations are protected by one process-based mutex. In order to avoid false sharing induced by the memory allocator, SHERIFF employs a scalable "per-thread" heap organization that is loosely based on Hoard [1] and built using HeapLayers [3]. SHERIFF divides the heap into a fixed number of sub-heaps (currently 16). The metadata for each sub-heap is also shared by different threads and protected by a cross-process mutex. In order to reduce lock contention, SHERIFF assigns different sub-heaps to each thread at creation time. Since each thread's heap allocates from different pages, the allocator itself is unlikely to collocate two objects from different threads on the same cache line.

Note that tools built with SHERIFF can specify, on a per-page basis, whether to use a shared mapping (so that updates are immediately visible to other "threads"), or a private mapping (so that updates are delayed). Both FS-DETECTIVE and FS-PATROL take advantage of this facility.

3.2 Shared File Access

In multithreaded programs, all threads share the same file descriptor table that tracks the process' open files. For example, if one thread opens a file, the other threads see that the file has been opened. However, multiple processes each have their own resources, including not only memory but also file handles, sockets, device handles, and windows.

While SHERIFF could manage these directly, our current prototype takes advantage of a feature of Linux that allows selective sharing of memory and file descriptors. SHERIFF sets the `CLONE_FILES` flag when creating new processes, resulting in child processes with different address spaces but the same shared file descriptor table.

3.3 Synchronization

SHERIFF supports the full range of POSIX synchronization operations (mutexes, conditional variables, barriers), as well as all thread-related calls, including cancellation.

At each synchronization point, SHERIFF must commit all changes made by the current thread. The span between synchronization points thus constitutes a single atomic transaction. Note that SHERIFF's approach differs significantly from previous transactional memory proposals [16], including Grace. SHERIFF is not optimistic, does not replace locks with speculation (it actually acquires program-level locks), never needs to roll back (it is always able to commit successfully), and achieves low overhead for long transactions.

To simulate multithreaded synchronization, SHERIFF intercepts all synchronization object initialization function

calls, allocates new synchronization objects in a mapping shared by all processes, and initializes them to be accessible by different processes.

SHERIFF wraps all synchronization operations, including mutexes, condition variables, and barriers in a similar fashion. For example, a call to `pthread_mutex_lock` first ends the current transaction, then calls the corresponding `pthreads` library function but on a cross-process mutex. SHERIFF then starts a new transaction after the lock is acquired, which ends with the next synchronization operation.

Thread-related calls are implemented in terms of their process counterparts. For example, `pthread_join` ends the current transaction and calls `waitpid` to wait for the appropriate process to complete.

3.4 Updating Shared Memory

At each synchronization point, SHERIFF updates the shared globals and heap with any updates that thread made. To accomplish this SHERIFF uses *twinning* and *diffing*, mechanisms first introduced in distributed shared memory systems to reduce communication overhead [6, 15].

Figure 2 presents an overview of both mechanisms at work. All private pages are initially write-protected. Before any page is modified, SHERIFF copies its contents to a "twin page" and then unprotects the page. At a synchronization point, SHERIFF compares each twin page to the modified page (byte by byte) to compute diffs.

3.5 Example Execution

This section walks through an example of SHERIFF's execution from the start of a program to its termination.

Initialization: Before the program begins, SHERIFF establishes the shared and local mappings for the heap and globals, and initiates the first transaction.

Transaction Begin: At the beginning of every transaction, SHERIFF write-protects any shared pages so that later writes to these pages can be caught by handling `SEGV` protection faults. In later transactions, SHERIFF only write-protects pages dirtied in the last transaction, since the others remain write-protected.

Execution: While performing reads, SHERIFF runs at the same speed as a conventional multithreaded program. However, the first write to a protected page triggers a page fault that SHERIFF handles.

SHERIFF records the page holding the faulted address and then unprotects this page so that future accesses run at full speed. Each page thus incurs at most one page fault per transaction. Although protection faults and signals are expensive, these costs are amortized over the entire transaction.

However, before servicing the fault, SHERIFF must first obtain an exact copy of this page (its twin). SHERIFF accomplishes this by forcing a copy-on-write operation on this page by writing to the start of this page with contents from the same address (i.e., it reads and writes the same value).

This step is essential to ensure that the twin is identical to the original, unmodified page. After the signal handler, the OS's copy-on-write mechanism creates a new, private page.

Transaction End: At the end of each atomically-executed region—at thread exit and before synchronization points—SHERIFF commits changes from private pages to the shared space and reclaims memory holding old private pages and twin pages.

SHERIFF commits only the differences between the twin and the modified pages. Once it has written these diffs, SHERIFF issues an `madvise` call with the `MADV_DONTNEED` flag that discards the physical pages backing both the private mapping and the twin pages. This action allows the OS to reclaim this memory, helping to ensure that SHERIFF's memory overhead remains close to that of the original program.

3.6 Discussion

As Section 3.1 notes, SHERIFF does not share the stack between different threads. When using `pthreads`, threads are allowed to share stack variables with their parent. As long as threads do not modify these variables, SHERIFF operates correctly. However, SHERIFF does not preserve `pthreads` semantics for applications whose threads *modify* stack variables from their parent thread that their parent then reads. Fortunately, passing stack variables to a thread for modification is generally frowned upon, making it a rare coding practice.

SHERIFF cannot currently intercept atomic operations written in assembly, so programs that implement their own *ad hoc* synchronization operations are not guaranteed to work correctly (Xiong et al. have shown that 22–67% of the uses of *ad hoc* synchronization they examined led to bugs or severe performance issues [23]). We expect this limitation to be less of a problem in the future because the forthcoming C++0x standard exposes atomic operations in a standard library, making it possible for SHERIFF to intercept them.

4. FS-DETECTIVE

We use the SHERIFF framework to build two tools that address false sharing. This section describes the first of these, FS-DETECTIVE, which detects false sharing.

FS-DETECTIVE detects both types of false sharing described in the literature. The first is the sharing of structurally-unrelated objects that happen to be located on the same cache line (i.e., different variables). The second is when multiple processors access different fields of the same object, which Hyde and Fleisch describe as "pseudo sharing" [11].

FS-DETECTIVE is designed to report only those instances of false sharing with the potential to seriously degrade performance. False sharing only causes a significant performance degradation when multiple threads concurrently and repeatedly update falsely-shared data, leading to large numbers of invalidation misses.

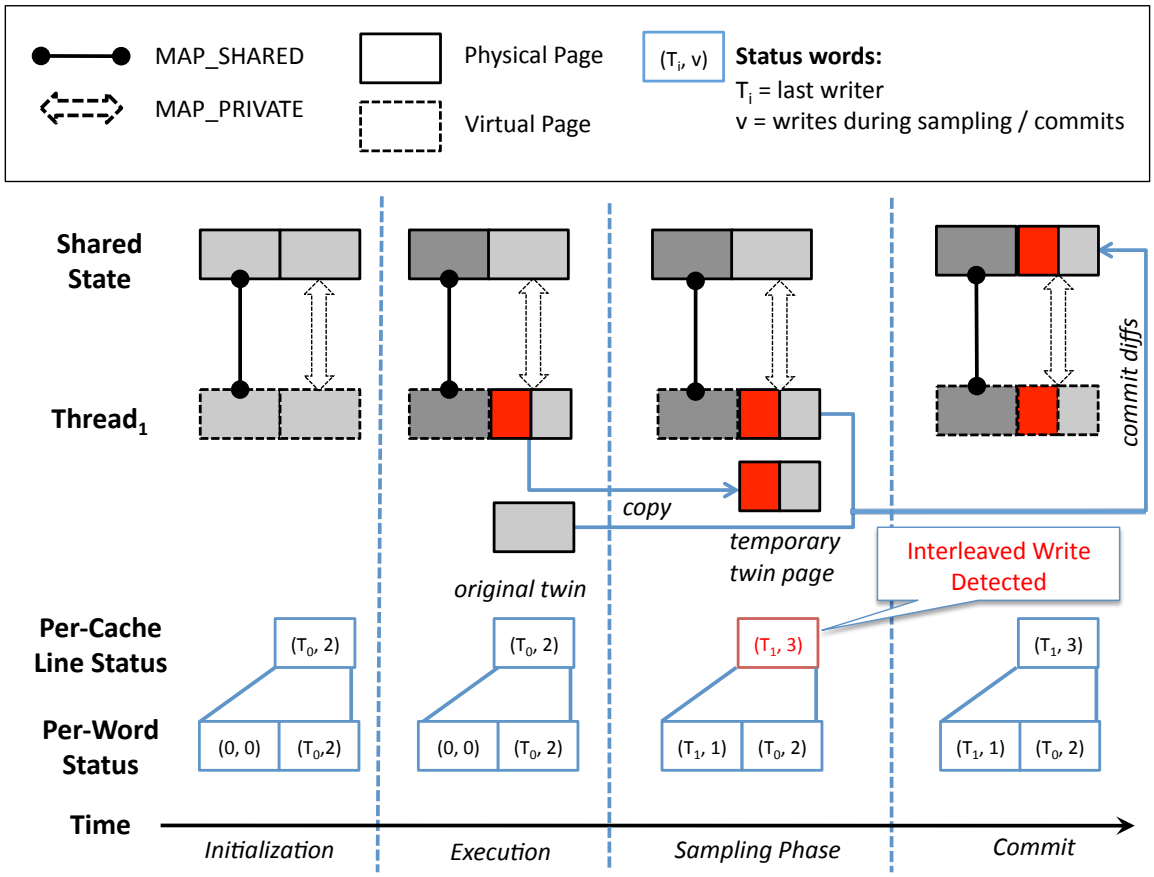


Figure 3. Overview of FS-DETECTIVE's operation. FS-DETECTIVE extends SHERIFF with page ownership tracking, sampling, and *per-cache line and per-word status* arrays that track frequent false sharing within cache lines. For clarity of exposition, the diagram depicts just one cache line per page and two words per cache line.

4.1 Page Ownership Tracking

One approach to implementing FS-DETECTIVE would be to use SHERIFF and direct it to protect all pages and map them private. The only change needed to detect false sharing would be an added check of twins and diffs against committed pages. Any cache line whose contents differ from the twin indicates it was changed by another thread. False sharing has occurred whenever a diff (a local update) lies on the same cache line as a local update.

However, this naïve approach would be quite costly. Since all local modifications of different threads must be committed to the shared space at every synchronization point to ensure correct execution (see Section 3.3), this approach would introduce substantial and unnecessary overhead for applications with a large number of unshared pages.

To reduce this overhead, FS-DETECTIVE leverages a simple insight: if two threads can falsely share a cache line, then they must simultaneously access the page containing that cache line.

Guided by this insight, FS-DETECTIVE relies on page protection to gather information about whether pages are shared or not. Instead of placing everything in the private address space, FS-DETECTIVE utilizes its knowledge about page sharing patterns and only maps those shared pages private.

FS-DETECTIVE initially read-protects all memory pages and tracks the number of threads that attempt to write a page concurrently. Any attempt to write to a page will trigger a page fault. FS-DETECTIVE then increments the access counter for this page before unprotecting the page. Once the access counter for a given page reaches two, the page is considered to be shared, and the page is mapped privately to each process to allow FS-DETECTIVE to locate possible false sharing.

4.2 Discovering Local Modifications

When FS-DETECTIVE concludes each transaction, it compares each dirty page with its twin a word at a time to find any modifications. FS-DETECTIVE thus identifies all writes made by the current thread. Whenever local modifications are found, either in the sampling period or at the end of a transaction (outside a critical section), FS-DETECTIVE sets

the virtual status words that indicate local modifications by the current thread.

4.3 Identifying Problematic False Sharing

FS-DETECTIVE uses sampling to measure the performance impact of false sharing, which it uses to rank its reports. FS-DETECTIVE currently uses a sampling interval of 10 milliseconds, which we empirically observe balances accuracy and performance overhead.

FS-DETECTIVE tracks the frequency of updates made to cache lines by associating a temporary twin page with each private, modified page (see Figure 3). These temporary twins are created only during sampling, and are updated to reflect the current working version at every sampling interval.

Only repeatedly interleaved writes (that is, by different threads) can degrade performance by repeatedly forcing cache line invalidations. FS-DETECTIVE monitors interleaved writes across different threads in order to capture this effect.

FS-DETECTIVE associates a *per-cache line status* with each cache line in every tracked page (Figure 2). This status contains two fields. The first points to the last thread to write to this cache line, and the second records the number of interleaved updates to the cache line. Every time a different thread writes to a cache line, FS-DETECTIVE updates the associated status word with both the thread id and the version number. To reduce overhead, FS-DETECTIVE splits the status into two different arrays to allow the use of atomic operations instead of locks.

In addition, during sampling and at commit time, FS-DETECTIVE updates *per-word status* values for every modified word. This information is later used to report the approximate frequency of updates at the individual word granularity. Programmers can then use this information to decide where to place padding. For example, if two struct fields are falsely shared, padding should be placed between the fields that are most frequently updated.

FS-DETECTIVE imposes relatively low memory overhead by maintaining status values only for pages that are shared by multiple threads. To further save space, FS-DETECTIVE stores each status in a single 32-bit word. The upper 16 bits stores the thread id, and the lower 16 bits stores the version number. When a word is detected to have been modified by more than two threads, FS-DETECTIVE sets the thread id field to `0xFFFF`, indicating that it is shared.

4.4 Reporting

At this point, FS-DETECTIVE has detected individual cache lines that are responsible for a large number of invalidations, and thus potential sources of slowdowns. The next step is to identify the culprit objects.

FS-DETECTIVE aims to provide as much context as possible about false sharing in order to reduce programmer effort, identifying global variables by name, heap objects by allocation context, and where possible, the fields modified by different threads. FS-DETECTIVE also provides an option to print out detailed information for every word in a given cache line, including the number of updates and the accessor threads.

FS-DETECTIVE identifies globals directly by using debug information that associates the address with the name of the global. For heap objects, FS-DETECTIVE instruments memory allocation to attach the call site to the header of each heap object. This calling context indicates the sequence of function calls that led to the actual allocation request, and is useful to help the programmer identify and correct false sharing, as the case study in Section 6.1.1 demonstrates. Any heap object responsible for a large number of invalidations is not deallocated so that it can be reported at the end of program execution.

4.5 Avoiding False Positives

FS-DETECTIVE instruments memory allocation operations to clean up cache invalidation counts whenever an object is de-allocated. This approach avoids the false positives caused by incorrectly aggregating counts when one address is reused for other objects.

4.6 Reporting Falsely Shared Objects

Once execution is complete, FS-DETECTIVE generates a ranked list of falsely shared objects. FS-DETECTIVE scans the cache invalidation array for cache lines with a number of invalidations above a fixed threshold (currently 100). The corresponding invalidation times and offset of this cache line are added to a global linked list sorted by invalidation times.

After scanning the cache invalidation array, FS-DETECTIVE obtains object information for all cache lines in the linked list, and reports the allocation site and offsets of all falsely-shared allocated objects.

4.7 Optimizations

FS-DETECTIVE employs several optimizations that further reduce its overhead.

Reducing timer overhead. As explained in Section 4.3, FS-DETECTIVE uses sampling to track interleaved writes by triggering a timer signal via `ualarm`. To reduce the impact of timer interrupts, FS-DETECTIVE activates sampling only when the average transaction time is larger than a threshold time (currently 10 milliseconds). FS-DETECTIVE uses an exponential moving average to track average transaction times ($\alpha = 0.9$). This optimization does not significantly reduce the possibility of finding false sharing since FS-DETECTIVE's goal is to find an object with a large amount of interleaved writes from different threads.

Sampling to find shared pages. FS-DETECTIVE relies on page protection to determine whether pages are shared or not. When one application has a large number of transactions or page touches, the protection overhead to gather this sharing information can dominate running time.

FS-DETECTIVE reduces overhead by using sampling to detect shared pages. If objects on a page are frequently falsely shared, the page itself must also be frequently shared, so even relatively infrequent sampling will eventually detect this sharing. FS-DETECTIVE currently samples the first 50 out of every 1,000 periods (one period equals one transaction or one sampling interval). At the beginning of each sampled period, all memory pages are made read-only so that any writes to each page will be detected. Once a page is found to be shared, FS-DETECTIVE will track any false sharing inside it. FS-DETECTIVE only updates the shared status of pages during sampled periods and at commit points. During unsampled periods, pages whose sharing status is unknown impose no protection overhead.

4.8 Discussion

Unlike previous tools, FS-DETECTIVE has no false positives, differentiates true sharing from false sharing, and avoid false positives caused by the reuse of heap objects. FS-DETECTIVE can under-report false sharing instances in the following situations:

Single writer. False sharing usually involves updates from multiple threads, but it can also arise when there is exactly one thread writing to part of a cache line while other threads read from it. Because its detection algorithm depends on multiple, differing updates, FS-DETECTIVE cannot detect this kind of false sharing.

Heap-induced false sharing. SHERIFF replaces the standard memory allocator with one that behaves like Hoard and thus reduces the risk of heap-induced false sharing. FS-DETECTIVE therefore does not detect false sharing caused by the standard memory allocator. Since it is straightforward to deploy Hoard or a similar allocator to avoid heap-induced false sharing, this limitation is not a problem in practice.

Misses due to sampling. Since it uses sampling to capture continuous writes from different threads, FS-DETECTIVE can miss writes that occur in the middle of sampling intervals. We hypothesize that false sharing instances that affect performance are unlikely to perform frequent writes exclusively during that time, and so are unlikely to be missed.

5. FS-PATROL

While FS-DETECTIVE can be effective at locating the sources of false sharing, it is sometimes difficult or impossible for programmers to remove the false sharing that FS-DETECTIVE can reveal. For instance, padding data structures to eliminate false sharing within an object or different elements of an array can cause excessive memory consumption or degrade cache utilization [24]. Time constraints may prevent programmers from investing in other solutions, or the source code may simply be unavailable. FS-PATROL, our second tool developed with the SHERIFF framework, can take unaltered C/C++ binaries and eliminate the performance degradation caused by false sharing.

To accomplish its goals, FS-PATROL relies on a key insight due initially to DuBois et al. [8]: *delaying updates avoids false sharing.* Delaying one thread's updates so that they do not cause invalidations to other threads eliminates the performance impact of false sharing. Consider the case when two threads are updating two falsely-shared objects A and B. If one thread's accesses to A were delayed so that they preceded all of the other thread's accesses to B, then the false sharing would cause no invalidations and hence have no performance impact.

In effect, this is exactly what SHERIFF does already when all pages are mapped private. By using processes instead of threads, all updates between synchronization points are applied to different physical addresses belonging to the different "threads". In this way, SHERIFF itself prevents false sharing by not updating the same physical cache lines.

However, using SHERIFF with all pages mapped private would be impractical as a runtime replacement for `pthreads`, because it would impose excessive overhead. This overhead arises due to protection and copying costs that would counteract the benefit of preventing false sharing.

For example, when a thread updates a large number of pages between transactions, SHERIFF must commit all local changes to the shared mapping at the end of every transaction, *even in the absence of sharing*. The associated protection and copying overhead can dramatically degrade performance. The same problem arises when transactions are short (e.g., when there are frequent lock acquisitions), because the cost of protection and page faults cannot be amortized by the protection-free part of the transaction.

FS-PATROL implements two key optimizations that improve performance by directly addressing these issues.

Focus on smaller objects: FS-PATROL focuses its false sharing prevention exclusively on small objects (those less than 1024 bytes in size). All large objects are mapped shared and are never protected.

First, we expect small objects to be a likely source of false sharing because they fit on a cache line. False sharing in large objects like arrays is also possible, but we expect the amount of false sharing relative to the size of the object to be far lower than with small objects (an intuition that FS-DETECTIVE confirms, at least across our benchmark suite). For such objects, the cost of protection would outweigh the advantages of preventing false sharing.

Second, the total amount of memory consumed by small objects tends to be less than that consumed by large objects. Because the cost of protecting and committing changes is proportional to the volume of updates, FS-PATROL limits protection to small objects, reducing overhead while capturing the benefit of false sharing prevention where it matters most.

Microbenchmark	Performance-Sensitive / Actual False Sharing?	FS-DETECTIVE	PTU
False sharing (adjacent objects)	Yes	✓	✓
Pseudo sharing (array elements)	Yes	✓	✓
True sharing	No		
Non-interleaved false sharing	No		×
Heap reuse (no sharing)	No		×

Table 1. False sharing detection results using PTU and FS-DETECTIVE. FS-DETECTIVE correctly reports only actual false sharing instances, and only those with a performance impact; ✓ denotes a correct report, and × indicates a spurious report (a false positive).

Adaptive false sharing prevention: Short transactions do not give SHERIFF the chance to amortize its protection overheads. To address this, FS-PATROL employs a simple adaptive mechanism. FS-PATROL tracks the length of each transaction and uses exponential weighted averaging ($\alpha = 0.9$) to calculate the average transaction length. Once the average transaction length is shorter than an established threshold, FS-PATROL switches to using shared mappings for all memory and does no further page protection. As long as transactions remain too short for FS-PATROL to have any benefit, all of its overhead-producing mechanisms remain switched off. If the average transaction length rises back above the threshold, FS-PATROL re-establishes private mappings and page protection.

6. Evaluation

We perform our evaluation on a quiescent 8-core system (dual processor with 4 cores), and 8GB of RAM. Each processor is a 4-core 64-bit Intel Xeon running at 2.33 Ghz with a 4MB L2 cache. For compatibility reasons, we compiled all applications to a 32-bit target. All performance data is the average of 10 runs, excluding the maximum and minimum values. Our evaluation answers the following questions:

- How effective is FS-DETECTIVE at finding and guiding programmers to resolve false sharing? (§ 6.1)
- What is FS-DETECTIVE's performance overhead? (§ 6.2)
- How effectively does FS-PATROL mitigate false sharing? (§ 6.3)

6.1 FS-DETECTIVE Effectiveness

We first evaluate FS-DETECTIVE with microbenchmarks we developed that exemplify a range of sharing scenarios. For comparison, we also present the corresponding results of Intel's Performance Tuning Utility (PTU), version 3.2 [12].

Table 1 presents the results of this evaluation. We can see that FS-DETECTIVE reports both false sharing and pseudo-sharing problems successfully, and correctly ignores the benchmarks with no actual false sharing performance impact (3–5). However, PTU reports false sharing for benchmarks 4 and 5. Note that benchmark 5 triggers a false positive due to heap object reuse: the two different allocations happen to occupy the same address. FS-DETECTIVE avoids this false positive by cleaning up invalid counting information.

Benchmark	PTU *cache lines*	FS-DETECTIVE *objects*
`blackscholes`	0	0
`canneal`	1	1
`dedup`	0	0
`ferret`	0	0
`fluidanimate`	3	1
`histogram`	0	0
`kmeans`	1,916	2
`linear_regression`	5	1
`matrix_multiply`	468	0
`pbzip2`	14	0
`pca`	45	0
`pfscan`	3	0
`reverse_index`	**N/A**	5
`streamcluster`	9	1
`string_match`	0	0
`word_count`	4	3
`swaptions`	196	0
Total	2,664	14

Table 2. Detection results for PTU and FS-DETECTIVE. For PTU, we show how many cache lines are reported as (potentially) falsely shared. For FS-DETECTIVE, we provide the number of objects reported (with significance filtering turned off). "N/A" indicates that PTU failed because it ran out of memory.

We next evaluate FS-DETECTIVE's effectiveness at detecting false sharing problems across a range of applications: the Phoenix [21] and PARSEC [4] benchmark suites, and two open source multithreaded applications, `pbzip2` (a parallel compressor) and `pfscan` (a parallel file scanner). We use the simlarge inputs for all applications of PARSEC. For Phoenix, we chose parameters that allow the programs to run as long as possible. [1]

[1] As of this writing, we are unable to successfully compile `raytrace` and `vips`, and SHERIFF is currently unable to run `x264`, `bodytrack`, and `facesim`. `Freqmine` is not included because it does not support `pthreads`.

```
int * use_len;
void insert_sorted(int curr_thread) {
   ......
   // After finding a new link
   (use_len[curr_thread])++;
   ......
}
```

Figure 4. False sharing detected by FS-DETECTIVE in `reverse_index`. False sharing arises when adjacent threads modify the `use_len` array.

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

void *lreg_thread(void *args_in) {
  struct lreg_args * args = args_in;
  for(i = 0; i < args->num_elems; i++) {
    args->SX  += args->points[i].x;
    args->SXX += args->points[i].x
              * args->points[i].x;
  }
  ......
}
```

Figure 5. False sharing detected by FS-DETECTIVE in `linear_regression`. Each thread is passed in a pointer to a struct as an argument, but the struct is smaller than a cache line (52 bytes), so two different threads write to the same cache line simultaneously.

Table 2 shows that FS-DETECTIVE reveals false sharing in seven out of seventeen benchmarks (when filtering for low performance impact is disabled; see below). FS-DETECTIVE detects false sharing in four benchmarks from the Phoenix suite and three from the PARSEC suite. However, for three of these benchmarks (`kmeans`, `canneal`, and `fluidanimate`), the average number of interleavings per cache line is lower than 10, indicating that the false sharing would not have a significant performance impact; FS-DETECTIVE is normally configured not to report these.

In `reverse_index` and `word_count`, multiple threads repeatedly modify the same heap object. A simplified version of the code is shown in Figure 4. Using a thread-local copy avoids false sharing: each thread can operate on a temporary variable and modify the global `use_len` only at the end of the thread.

`Linear_regression`'s false sharing problem is slightly different (see Figure 5). In this case, two different threads falsely share the same cache line if the structure `lreg_args` is not cache line aligned. This problem can easily be avoided simply by padding the structure `lreg_args`.

The false sharing detected in `streamcluster` (one of the PARSEC benchmarks) is similar to the false sharing in `linear_regression`.

Benchmark	Orig (s)	Mod (s)	Speedup	Updates (M)
`linear_regression`	3.40	0.37	818%	1323.6
`reverse_index`	2.08	2.03	2.4%	0.4
`streamcluster`	2.78	2.63	5.4%	28.7
`word_count`	2.20	2.18	1%	0.3

Table 3. Performance data for benchmarks with significant false sharing, as reported by FS-DETECTIVE. "Orig" and "Mod" are the runtimes before and after fixing the false sharing revealed by FS-DETECTIVE. All data are obtained running with `pthreads`. "Updates" is the maximum number of updates to falsely-shared cache lines.

Examination of the source code indicates that the author tried to avoid false sharing by padding, but the amount of padding, 32 bytes, was insufficient to accomodate the actual physical cache line size used in the evaluation (64 bytes). Setting the `CACHE_LINE` macro to 64 bytes eliminates the false sharing.

The performance of these four benchmarks is listed in Table 3, before and after fixing the false sharing issues that FS-DETECTIVE identifies. To understand the differences in performance improvements, we modified the code to compute the number of updates to these falsely shared objects. Updates listed here are the *maximum* possible number of interleaving writes of these objects; the actual number of interleaving writes depends on scheduling.

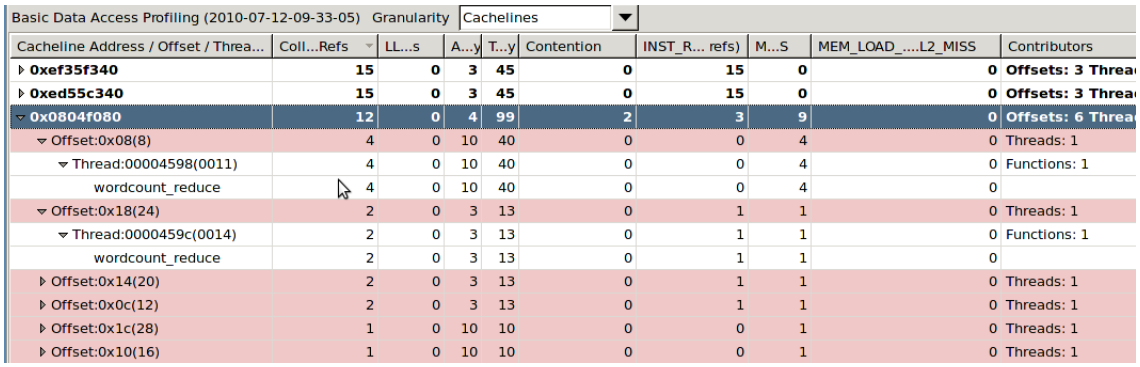
The `reverse_index` and `word_count` benchmarks do not exhibit substantial performance improvements after removing false sharing because the number of updates is relatively low. For example, the maximum number of interleaved updates for `reverse_index` is 416,000. The `streamcluster` benchmark has around 28 million updates, and eliminating false sharing provides a modest performance improvement (5.4%). The most dramatic improvement is for `linear_regression`. After removing its false sharing, it runs $9\times$ faster.

6.1.1 FS-DETECTIVE vs. PTU

To evaluate FS-DETECTIVE's effectiveness at finding false sharing, we compare it to Intel's Performance Tuning Utility (PTU), a commercial product which represents the state of the art for detecting false sharing.

This comparison evaluates the number of reports that each tool generates, and the effectiveness of each at helping the programmer find and resolve actual instances of false sharing.

Reporting. For PTU, we list the number of cache lines reported as possibly falsely-shared. To locate a single case of false sharing, a programmer must sift through every one of these reports. FS-DETECTIVE reports sharing at the object granularity.



Basic Data Access Profiling (2010-07-12-09-33-05) Granularity: Cachelines

Cacheline Address / Offset / Threa...	Coll...Refs	LL...s	A...y	T...y	Contention	INST_R... refs)	M...S	MEM_LOAD_....L2_MISS	Contributors
0xef35f340	15	0	3	45	0	15	0	0	Offsets: 3 Threa
0xed55c340	15	0	3	45	0	15	0	0	Offsets: 3 Threa
0x0804f080	12	0	4	99	2	3	9	0	Offsets: 6 Threa
Offset:0x08(8)	4	0	10	40	0	0	4	0	Threads: 1
Thread:00004598(0011)	4	0	10	40	0	0	4	0	Functions: 1
wordcount_reduce	4	0	10	40	0	0	4	0	
Offset:0x18(24)	2	0	3	13	0	1	1	0	Threads: 1
Thread:0000459c(0014)	2	0	3	13	0	1	1	0	Functions: 1
wordcount_reduce	2	0	3	13	0	1	1	0	
Offset:0x14(20)	2	0	3	13	0	1	1	0	Threads: 1
Offset:0x0c(12)	2	0	3	13	0	1	1	0	Threads: 1
Offset:0x1c(28)	1	0	10	10	0	0	1	0	Threads: 1
Offset:0x10(16)	1	0	10	10	0	0	1	0	Threads: 1

Figure 6. A fragment of Intel Performance Tuning Utility's report for `word_count` (see Section 6.1.1); the full report is 863 lines long.

From the results listed in Table 2, we can see that FS-DETECTIVE imposes a far lower burden on the programmer. Across all of the benchmarks, PTU indicates the need to examine 2,664 cache lines overall (not including `reverse_index`, which PTU cannot run). By contrast, FS-DETECTIVE reports just 14 objects as potential source of false sharing. After increasing its significance threshold parameter, FS-DETECTIVE reports just 4 objects, all of which are truly false sharing.

Several factors lead to this difference. First, FS-DETECTIVE reports objects rather than cache lines, which reduces the number of reports when one callsite is the source of a number of allocations (as in `kmeans`) or when one object spans multiple cache lines (as in `reverse_index`). Second, FS-DETECTIVE distinguishes true from false sharing, reducing the number of reported items. Finally, FS-DETECTIVE only reports those objects with interleaving writes above a threshold, which significantly reduces the number of reports.

Ease of locating false sharing. To illustrate how FS-DETECTIVE can precisely locate false sharing instances, we use the `word_count` benchmark as a representative example. Our experience with diagnosing other false sharing issues is similar. Below is an extract of FS-DETECTIVE's report for `word_count`:

```
1st object, cache interleaving writes
13767 times (start at 0xd5c8e140).
Object start 0xd5c8e160, length 32.
It is a heap object with callsite:
[0] ./wordcount_pthreads.c:136
[1] ./wordcount_pthreads.c:441
```

Line 136 (`wordcount_pthreads.c`), contains the following memory allocation:

```
use_len=malloc(num_procs*sizeof(int));
```

Grepping for `use_len`, a global pointer, quickly leads to this line:

```
use_len[thread_num]++;
```

Now it is clear that different threads are modifying the same object (`use_len`). Fixing the problem with thread-local copies of this object is now straightforward [13].

By contrast, compare PTU's output for the same benchmark, shown in Figure 6. Finding this problem is far more complicated with PTU. The full report consists of 863 lines describing cache lines and the functions that access them, not individual objects. The task of finding false sharing is further exacerbated by the fact that many of these reports are false positives. PTU is also unable to effectively rank the performance impact of false sharing instances. The "Collected Data Refs" metric (the second column) is inaccurate at best: for this example, PTU only reports 12 references, while FS-DETECTIVE observes 13,767 references.

6.2 FS-DETECTIVE Performance

Figure 7 presents the runtime overhead of FS-DETECTIVE versus `pthreads` across our benchmark suites. FS-DETECTIVE executes with surprisingly low overhead: 20% on average, with the exception of three outliers.

There are two benchmarks on which FS-DETECTIVE does not perform particularly well. One is `canneal`, where the performance overhead of FS-DETECTIVE is about 8× compared to `pthreads`, while `fluidanimate`'s overhead is approximately 11× slower than that using `pthreads`.

The first reason for these overheads is that both benchmarks trigger a high number of dirtied pages (3.4 million and 2.15 million, respectively). For each dirty page, FS-DETECTIVE applies memory protection twice, creates the copy-on-write version and twin page, checks for false sharing at every checking period, and finally commits updates to the shared mapping. Copying alone is a major part of the overhead associated with dirty pages, since one dirty page needs at least three copies. For `canneal`, copying alone accounts for about 20 seconds of additional execution time.

`fluidanimate` also runs slowly with FS-DETECTIVE because of an unusually high number of transactions (16.7 million). Examination of the source code of `fluidanimate` reveals frequent locking and unlocking. SHERIFF replaces

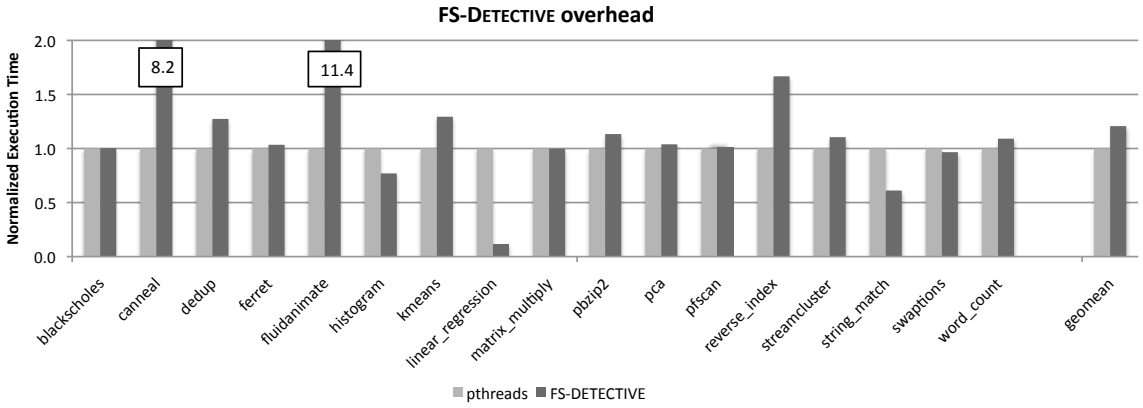


Figure 7. FS-DETECTIVE overhead across two suites of benchmarks, normalized to the performance of the `pthreads` library (lower is better). With two exceptions, its overhead is quite low (on average, just 20%).

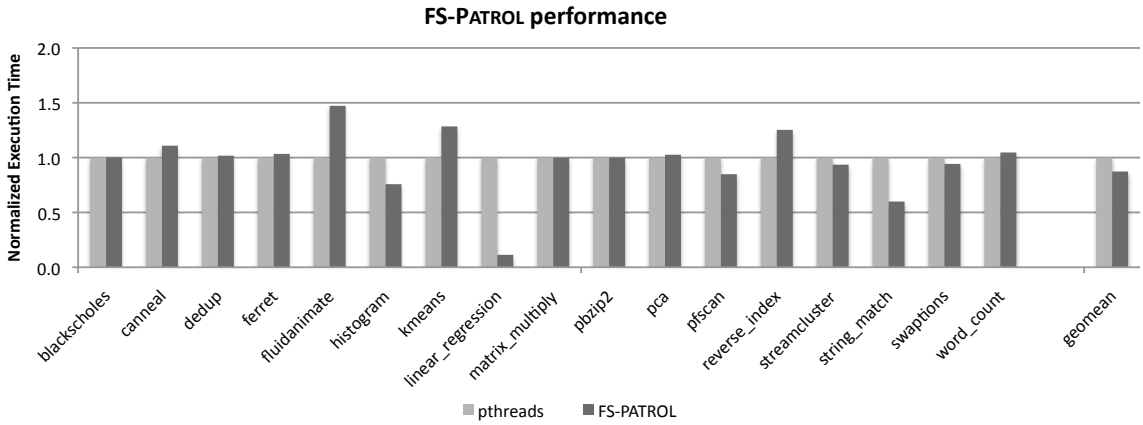


Figure 8. FS-PATROL performance, normalized to the performance of the `pthreads` library (see Section 6.3; lower is better). For applications with significant false sharing, FS-PATROL can substantially increase performance.

lock calls with their interprocess variants and triggers a transaction end and begin for each, adding overhead if there are shared pages (as here).

While these outliers force FS-DETECTIVE to run slowly, FS-DETECTIVE's overhead is generally acceptable and far lower than most previous tools.

`linear_regression` is an outlier in the opposite direction: with FS-DETECTIVE, it runs 8× faster than with `pthreads`. The reason is a serious false sharing issue (see Table 3) that both FS-DETECTIVE and FS-PATROL eliminate automatically, thus dramatically improving performance. Other cases where FS-DETECTIVE outperforms `pthreads` are also due to false sharing elimination; FS-PATROL further reduces overhead for these and other applications, as the next section describes.

6.3 Effectiveness of FS-PATROL

Here, we examine the effectiveness of FS-PATROL at mitigating false sharing; Figure 8 presents execution times versus `pthreads`. For most applications, FS-PATROL either has no effect on performance (when there is no false sharing to eliminate) or improves it. Table 4 provides detailed performance numbers for both FS-PATROL and FS-DETECTIVE.

For three applications, FS-PATROL degrades performance by up to 47% versus `pthreads`. For `kmeans`, which creates over 375 threads per second, the added cost stems from using processes instead of threads. While process creation on Linux is relatively cheap, it is still more expensive than creating threads.

The slowdowns for `reverse_index` and `fluidanimate` are due to more subtle technical details of the processes-as-threads framework. SHERIFF uses a file-based mapping to connect the private and shared mappings. The Linux page fault handler does more work when operating on file-based pages than on anonymous pages (the normal case for heap-allocated pages). For example, the first write to a file-mapped page repopulates information from the file's page table entry. In addition, the shared store for all heap pages is initially set to `MAP_SHARED`, so writing to one shared page can cause a copy-on-write operation in the kernel even when

Benchmark	Normalized Runtime	
	FS-DETECTIVE	FS-PATROL
blackscholes	1.00	1.00
canneal	8.23	1.11
dedup	1.27	1.02
ferret	1.03	1.03
fluidanimate	11.39	1.47
histogram	**0.77**	**0.76**
kmeans	1.29	1.28
linear_regression	**0.12**	**0.11**
matrix_multiply	1.00	1.00
pbzip2	1.13	1.00
pca	1.04	1.03
pfscan	1.02	**0.85**
reverse_index	1.67	1.25
streamcluster	1.10	**0.94**
string_match	**0.61**	**0.60**
swaptions	**0.97**	**0.94**
word_count	1.09	1.05
Geomean	1.21	**0.87**

Table 4. Detailed execution times with FS-DETECTIVE and FS-PATROL, normalized to execution with the pthreads library; numbers below 1 (boldfaced) indicate a speedup over pthreads.

there is only one user. As future work, we plan to investigate extending the kernel with an additional mapping mode to eliminate this overhead.

FS-PATROL improves the performance of the programs implicated by FS-DETECTIVE as suffering from false sharing, as well as several others. For example, histogram also runs substantially faster with FS-PATROL (24%), although we currently are not certain why this is the case. string_match runs 40% faster because of false sharing caused by the pthreads heap allocator (which is why FS-DETECTIVE does not find it). The most dramatic improvement comes from linear_regression, which runs 9× faster than with pthreads because FS-PATROL eliminates its serious false sharing (see Table 3). These results show that FS-PATROL is effective at mitigating false sharing without the need for programmer intervention or access to source code.

7. Conclusion

This paper presents two tools that attack the problem of false sharing in multithreaded programs. Both are built with SHERIFF, a software-only framework that enables per-thread memory protection and isolation. SHERIFF works by converting threads into processes, and uses memory mapping and a difference-based commit protocol to provide isolated writes. FS-DETECTIVE identifies false sharing instances with no false positives, and pinpoints the objects involved in performance-critical false sharing problems. We show that FS-DETECTIVE can greatly assist programmers in tracking down and resolving false sharing problems. When it is not feasible for programmers to resolve these problems, either because code is unavailable or because the fixes would degrade performance further, FS-PATROL can be used to automatically eliminate the false sharing problems indicated by FS-DETECTIVE. We show that FS-PATROL can substantially improve the performance of applications with moderate to severe false sharing, without the need for programmer intervention or source code.

We plan to release the source code for SHERIFF, FS-DETECTIVE, and FS-PATROL by publication time.

8. Acknowledgments

The authors thank Luis Ceze and Charlie Curtsinger for their valuable feedback and suggestions which helped improve this paper. We acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CCF-0910883. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, Nov. 2000.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

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

[5] 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.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP '91: Proceedings*

of the thirteenth ACM symposium on Operating systems principles, pages 152–164, New York, NY, USA, 1991. ACM.

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

[8] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 197–206, New York, NY, USA, 1991. ACM.

[9] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[10] 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.

[11] R. L. Hyde and B. D. Fleisch. An analysis of degenerate sharing and false coherence. *J. Parallel Distrib. Comput.*, 34(2):183–195, 1996.

[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] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–188, New York, NY, USA, 1995. ACM.

[15] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[16] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, first edition, 2007.

[17] J. Levon. OProfile internals. http://oprofile.sourceforge.net/doc/internals/index.html, 2003.

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

[19] M. Olszewski and S. Amarasinghe. Outfoxing the mammoths: PLDI 2010 FIT presentation, June 2010.

[20] 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.

[21] 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.

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

[23] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation*, pages 163–176, Berkeley, CA, USA, 2010. USENIX Association.

[24] 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.