Detecting and Eliminating 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 multi-threaded programs running on multicore processors, where it can silently degrade performance and scalability. Debugging false sharing problems is notoriously difficult. Previous approaches aimed at identifying false sharing are not only slow (degrading performance by orders of magnitude), but also severely limited: they do not distinguish false sharing from true sharing, cannot cope with dynamically allocated objects, generate numerous false positives, and fail to pinpoint the sources of false sharing.

This paper first presents SHERIFF, a runtime replacement for the standard `pthreads` library extended with *per-thread* memory protection and optional per-page isolation via a combination of processes, virtual memory support, and a differencing-based commit protocol. Using SHERIFF, we develop two tools to address false sharing: DETECTIVE and PATROL. DETECTIVE precisely tracks down the sources of false sharing, with no false positives and generally low overhead. A case study shows that DETECTIVE can quickly identify false sharing and guide programmers to remove it. As an alternative to manually removing false sharing, PATROL can automatically eliminate false sharing in unaltered, multithreaded C/C++ applications. To our knowledge, PATROL is the first system of its kind. PATROL works by isolating updates from different threads and merging their results only when needed. By protecting applications from false sharing, PATROL can dramatically improve performance (by up to $10\times$ versus `pthreads`).

1 Introduction

Writing multithreaded programs is challenging. Not only is it difficult to get them to run correctly (e.g., due to races, atomicity violations, or deadlocks), it is also difficult to make them efficient and scalable. Key to achieving high performance and scalability is reducing contention for shared resources. However, even when threads operate on disjoint objects, they can still suffer from *false* sharing when multiple objects reside on the same cache line. When false sharing is frequent enough, the resulting "ping-ponging" of cache lines across processors can cause performance to plummet [**?**, **?**]. False sharing can degrade application performance by as much as an order of magnitude. The trend towards larger cache lines together with an increased number of multithreaded applications makes it likely that false sharing will be an increasingly common problem.

Locating false sharing requires tool support. Past false sharing detection tools operate on binaries, either via simulation [**?**] or binary instrumentation [**?**, **?**], and intercept all reads and writes (leading to slowdowns of up to three orders of magnitude), or rely on hardware performance monitors and correlate cache invalidations with lines of code [**?**, **?**]

These false sharing detection tools suffer from two problems: imprecise reporting of false sharing sources, and large numbers of false positives. Reports generated by these tools include potentially falsely-shared addresses and the functions that accessed them, leaving it to the programmer to discover what objects these addresses correspond to. False positives include objects that were falsely shared so few times that they do not present a performance bottleneck, incorrectly-aggregated access information due to object reuse by memory allocators, and cases of true sharing.

Rather than detecting false sharing, in some restricted cases, it is possible to eliminate it. Compilers can minimize false sharing by adjusting memory layouts through padding and alignment, or alter parallel loop scheduling to avoid sharing [**?**, **?**]. Unfortunately, the effectiveness of these techniques is primarily limited to array-based scientific codes. For general-purpose applications, a scalable memory allocator can reduce the likelihood of false sharing of distinct heap objects [**?**], but cannot prevent false sharing within individual heap objects.

Contributions

This paper first presents **SHERIFF**, a software-only framework that replaces the standard `pthreads` library. SHERIFF extends `pthreads` with *per-thread memory protection* and optional *memory isolation* (on a per-page basis) for multithreaded C/C++ applications. SHERIFF works by shimming all `pthreads` calls with equivalents that operate on processes rather than threads, and manages isolated updates via page-level twinning and diffing. We believe that SHERIFF is of independent interest and may be useful for a range of applications, including language support and enforcement of data sharing, software transactional memory, thread-level speculation, and race detection [**?**].

We use SHERIFF to develop two tools to attack the problem of false sharing. The first, **DETECTIVE**, 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), and only those instances that might have a performance impact. 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. DETECTIVE is also efficient: it imposes just a 30% performance penalty on average and a worst-case overhead about 14×.

While DETECTIVE is effective at locating false sharing, rewriting a program to remove it can be impractical. Padding objects may degrade performance, or source code might not be available.

Our second tool, **PATROL**, eliminates false sharing automatically, without the need for code modifications or recompilation. When used as a false sharing resistant runtime, PATROL can dramatically increase performance in the face of false sharing sharing. On an 8-core system, PATROL accelerates one benchmark by approximately 10X versus the standard `pthreads` library. To our knowledge, PATROL is the first false sharing resistant runtime for shared-memory multithreaded programs.

2 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* [**?**] and *page-level twinning and diffing* [**?**, **?**].

Replacing `pthreads` with processes is surprisingly inexpensive, especially on Linux where both `pthreads`

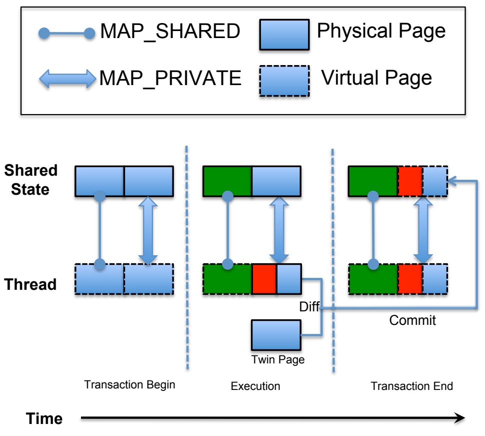


Figure 1: Sheriff overview. SHERIFF acts as replacement for `pthreads` but simulates threads using processes, enabling per-thread memory protection. Each process either operates directly on shared memory, or on private copies. For the latter, SHERIFF commits diffs to shared mappings at synchronization points.

and processes are invoked using the same underlying system call. Berger et al. show that process invocation can actually outperform threads because Linux initially assigns threads to the same CPU, while it spreads processes across all CPUs [**?**]. To achieve the effect of shared memory, SHERIFF maps globals and the heap into a shared region (Section **??**). It also intercepts the `pthread_create` call and replaces it with a process creation call (Section **??**).

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

To maintain the shared-memory semantics of a multithreaded 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 **??**). However, SHERIFF takes care to update only the changed parts of each page (Section **??**).

2.1 Shared Address Space

To create the effect of multi-threaded programs that different threads are sharing 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 **??**.

Note that clients of SHERIFF can choose, 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 DETECTIVE and PATROL take advantage of this facility.

Globals and Heap SHERIFF uses the same global and heap organization as Grace, which we describe briefly. 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 (currently set at 1GB). Memory allocations requirements 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 [**?**] and built using HeapLayers [**?**]. SHERIFF divides the heap into a fixed number of sub-heaps (currently 16), the metadata of each sub-heap are also shared by different threads and protected by different process-based mutex. In order to reduce lock contention, SHERIFF assigns different sub-heaps to each thread at creation time which it relinquishes on exit. 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.

```
int spawnWithSharedFiles{
  return syscall(SYS_clone,
   CLONE_FS|CLONE_FILES|SIGCHLD,(void*)0);
}
```

Figure 2: Linux code to create a new process that shares file descriptors with its parent, but does not share an address space.

```
  void pthread_sync (var) {
    closeTransaction();
    realVar = getRealVariable(var);
    real_pthread_sync (realVar);
    startTransaction();
  }
```

Figure 3: A template for the implementation of synchronization operations in SHERIFF (where "sync" denotes the appropriate operation).

2.2 Shared File Access

In multithreaded programs, different threads in the same process share the file descriptor that manages all opened files of each thread. 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 just memory but also file handles, sockets, device handles, and windows.

SHERIFF takes advantage of a low-level feature of Linux that allows selective sharing of memory and file descriptors. By setting the flag `CLONE_FILES` when creating new processes (Figure **??**), child processes can share the same physical file descriptor table with the parent process while not sharing the same address space.

2.3 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 to individual pages. The span between synchronization points thus constitutes single atomic transaction. Note that SHERIFF's approach differs significantly from previous transactional memory proposals [**?**], including Grace's. SHERIFF is not optimistic, does not replace locks with speculation (i.e., it actually acquires program-level locks), never needs to roll back (i.e., 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.

Figure **??** presents a template for how SHERIFF wraps most synchronization operations, including mutexes, condition variables, and barriers. For example, a call to `pthread_mutex_lock` first ends the current transaction. It then calls the corresponding `pthreads` library function but on a process-wide mutex. SHERIFF then starts a new transaction after the lock is acquired, which ends once the corresponding lock is unlocked.

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.

2.4 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 [**?**, **?**]. DETECTIVE uses diffs to identify modifications on a word-by-word basis.

Figure **??** presents an overview of both mechanisms at work (for this example, all pages are mapped private). All 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.

2.5 Example Execution

Figure **??** presents an overview of SHERIFF's execution.

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 all pages so that later writes on those 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 almost the same speed as that of 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 sets this page to writeable so that future accesses on this page can 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 the content obtained from the same address (i.e., it reads and writes the same value).

This step is essential in order to ensure that the twin is identical to the unmodified page. Since there is a time gap between the creation of twin pages and that of private pages, private pages are created by OS's copy-on-write mechanism after the signal handler. After forcing the copy-on-write, SHERIFF stores the twin in a local store.

Transaction End: At the end of each atomically-executed region—that is, the end of each thread, and before synchronization points, thread spawns and joins—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 to discard the physical pages backing both the private mapping and the twin pages.

2.6 Limitations

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

In addition, SHERIFF cannot currently intercept atomic operations written in assembly, so that programs that implement their own *ad hoc* synchronization operations are not guaranteed to work correctly (although Xiong et al. show that 22–67% of the uses of *ad hoc* synchronization they examined lead to bugs or severe performance issues [**?**]). 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 easy for SHERIFF to shim them.

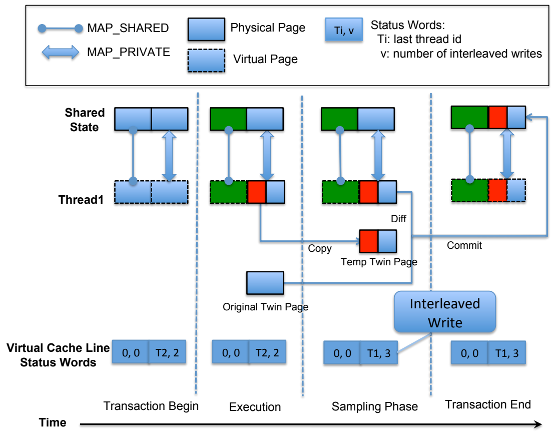


Figure 4: Overview of DETECTIVE's operation. DETECTIVE extends SHERIFF with sampling (Section **??**) and *virtual cache line status words* (Section **??**) to detect frequent false sharing within cache lines.

3 DETECTIVE

With the SHERIFF framework in hand, we now proceed to describe the tools we built using it to address false sharing. This section describes DETECTIVE, a tool that detects false sharing.

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" [**?**]).

Note that DETECTIVE does not report all cases of false sharing, but rather focuses exclusively on identifying those false sharing cases with the potential to seriously degrade performance. False sharing only causes significant performance degradation when multiple threads repeatedly update falsely-shared data, because updates lead to invalidations and thus cache misses.

Figure **??** presents an overview of DETECTIVE's key features.

3.1 Key Mechanisms

XXXX: added by Tongping

DETECTIVE detects false sharing in both heap variables and globals. In order to detect those false sharing problems, one naive mechanism is to separate all running of different threads using process at first (based on SHERIFF framework), then we can discover the false sharing problem by analyzing the accesses of different threads using the twining and diffing mechanism.

Since all local modification of different threads should be committed to the shared space at every synchronization point in order to ensure the correctness of execution (see Section **??**), this naive mechanism can introduce much overhead for one application when the amount of non-shared pages are large.

DETECTIVE leverages a key insight here in order to improve its performance: if two threads can cause a false sharing problem on a cache line, then these two threads must simultaneously access the page containing that cache line..

Based on this insight, DETECTIVE relies on page protection to gather information about whether pages are shared or not. Instead of placing everything in a private address space *a priori*, DETECTIVE utilizes the knowledge about pages' sharing and only separate those shared pages to different address space.

In the actual implementation, DETECTIVE sets the protection mode of all memory pages to read-only mode in the beginning and maintains one corresponding user counter(initialized to 0 in the beginning) for every page, which counts the number of threads writing on the page concurrently. Since those pages are read-only mode, the attempts to write on one page can invoke a page fault. By handling the page fault, DETECTIVE can get information about the accessers on this page and increment the user counter for this page. Whenever the user counter for one page is larger than two, the page is considered to be shared by two users concurrently, which can be detected in the periodical timer handler or in the end of one transaction. Thus, DETECTIVE can set those shared pages to a MAP_PRIVATE mode using mmap and then separate the running of those pages relying on the separate address space of processes.

XXXX: added end by Tongping

3.2 Discovering False Sharing

When DETECTIVE concludes each transaction, it compares each dirty page with its twin word by word to find any modifications, and thus identifies all writes made by the current thread. To detect false sharing, DETECTIVE simply compares the original contents of adjacent memory in the same cache line (in the twin) to those on the committed page (that is, those written by another thread). A modification of any adjacent data indicates the presence of false sharing: another thread has modified data on the same cache line that the current thread has also modified.

However, as mentioned earlier, the presence of one instance of false sharing does not necessarily indicate a performance problem. A single transaction may run for a long period of time (seconds or more), and one false sharing instance in this period is not a problem.

3.3 Identifying Problematic False Sharing

To precisely measure the impact of observed false sharing instances, DETECTIVE uses a sampling-based approach. There is a balance between choosing a finer sampling period and performance overhead. DETECTIVE currently uses 5 microseconds as its sampling interval.

DETECTIVE counts the number of writes by associating a temporary twin page with each shared dirty page (see Figure **??**). Handling of these temporary twin pages is slightly different than for the original twin pages. First, they are created in the sampling timer handler whenever a page is found to be shared by multiple threads (no temporary twins are created for pages only accessed by one thread). DETECTIVE records the users of each page in a global array. Second, temporary twin pages are updated with the working version at every sampling interval (triggered by a timer).

3.4 Capturing Cache Interleaved Writes

Only repeatedly interleaved writes can cause a performance problem by causing repeated cache line invalidations. DETECTIVE monitors interleaved writes across different threads in order to capture the effect of cache invalidations.

In order to capture interleaved writes on caches, DETECTIVE uses virtual cache line status words (Figure **??**). DETECTIVE assigns one status word to every cache line under protection. Each status word has two fields. The first field points to the last thread to write to this cache line, and the second field records the number of invalidations (a version number) to the cache line.

Every time a different thread writes to a cache line, DETECTIVE updates the associated status word with both the thread id and the version number. In the actual implementation, DETECTIVE splits the status word into two different arrays to allow the use of atomic operations instead of locks.

3.5 Reporting

At this point, 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 identifying the culprit objects. 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.

DETECTIVE identifies globals directly by using debug information that associates the address with the name of the global. For heap objects, 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 problems, as the case study in Section **??** 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.

3.6 Avoiding False Positives

DETECTIVE also 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 re-used for other objects.

To further reduce false positives, DETECTIVE uses another global array to record which threads have written each word, and the version numbers of each word. Threads writing on each word can tell whether one cache line is false sharing or true sharing. Associating a version number with each word allows DETECTIVE to avoid reporting those objects which do not contribute much on cache invalidations when there are multiple objects in the same cache line. In order to save space, DETECTIVE uses the upper 16 bits to store the thread id, and uses the lower 16 bits to store the word's version number. When one word is detected to have been modified by more than two threads, DETECTIVE sets the thread id field to 0xFFFF.

3.7 Reporting Falsely Shared Objects

At the end of program, DETECTIVE reports those objects causing false sharing problems. DETECTIVE scans the cache invalidation array for cache lines with invalidation times larger than 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. Finally, DETECTIVE ranks the falsely shared objects by the number of invalidations they caused.

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

3.8 Optimization

XXXX: Added by tongping
DETECTIVE employs the following optimizations.

Reducing the overhead to start timers . According to description in Section **??**, DETECTIVE are using the sampling mechanism to capture the continuous writes in one transaction by handling the timer handler (using ualarm, now 5ms). One optimization is to set the timer handler only when the average transaction time is larger than one certain number(10ms now). DETECTIVE are using exponential moving average(exponent is set to 0.9). Unabling to start timer once can only cause the miss of one time's checking, this optimization won't lower the possibility to find the false sharing problem since the target is to find out one object with large amount of interleaving writes from different threads.

Sampling mechanism to find shared pages . DETECTIVE relies on the page protection to get the knowledge about pages' sharing information. When one application has a lot of transactions or touches a lot of pages, the overhead to find the pages' sharing can dominate the running time. To improve the performance, DETECTIVE relies on sampling mechanism to find out those unknown shared pages. If one page has a serious false sharing problem inside, the behavior of sharing is assumed to happen frequently. In the actual implementation, we sampling the first 50 period out of every 500 period. One period here means one transaction or one checking period. In the beginning of the sampling period, all memory pages are set to the read-only mode thus writes to one page can be detected. When one page is known to be shared(already set to MAP_PRIVATE), thus we won't miss any false sharing problem inside those shared pages. When one page is un-known to be shared(set to be MAP_SHARED) but read-only, thus we can detect the shared status in the sampling period. While in the non-sampling period, those shared pages are kept inact while those un-known pages won't introduce the protection overhead anymroe.

XXXX: Added end by tongping

3.9 Limitations

Unlike previous tools, DETECTIVE has no false positives. It can differentiate true sharing and false sharing and avoid false positives caused by the reuse of heap objects.

However, a key question is to what extent DETECTIVE has *false negatives*; that is, when does it fail to report false sharing (that is, when such false sharing can lead to performance degradation)?

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, DETECTIVE cannot isolate 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. DETECTIVE therefore does not detect false sharing caused by the standard memory allocator. Since it is straightforward to simply use 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, DETECTIVE can miss writes that occur in the middle of sampling intervals. We hypothesize that false sharing problems that affect performance are unlikely to perform frequent writes exclusively during that time.

4 PATROL

While DETECTIVE can be effective at locating the sources of false sharing, it is sometimes difficult or impossible for programmers to remove the false sharing problems that 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. Time constraints may prevent programmers from investing in other solutions, or the source code may simply be unavailable. 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, PATROL relies on a key insight first observed by DuBois et al. [**?**]: **delaying updates avoids false sharing.** If it were possible to delay one thread's updates so that they do not cause invalidations to other threads, this would eliminate the performance impact of false sharing. Concretely, suppose 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 in the case when all pages are mapped private. By using processes instead of threads, updates occur locally on different threads between synchronization points. In this way, SHERIFF itself avoids the false sharing caused by updating the same physical cache line.

However, using SHERIFF in this way is impractical as a runtime replacement for `pthreads`, because it can impose excessive protection and copying overhead that negates 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.

In order to improve the performance for most of applications, PATROL implements two optimizations that address these two problems:

False sharing prevention for small objects only. PATROL focuses its false sharing prevention exclusively on small objects (those less than 1024 bytes in size) for two reasons.

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 likely, 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 DETECTIVE confirms, at least across our benchmark suite). This means that the cost of protection could easily outweigh the advantages of preventing false sharing for such objects.

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, PATROL limits protection to small objects, reducing overhead while capturing the benefit of false sharing prevention where it matters most.

Adaptive false sharing prevention. As mentioned above, short transactions do not give SHERIFF the chance to amortize its protection overheads. PATROL employs a simple adaptive mechanism that addresses this problem. 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, PATROL switches to using shared mappings for all memory and does no further page protection. As long as transactions remain too short for PATROL to have any benefit, all of its overhead-producing mechanisms remain switched off. Only when the average transaction length rises back above the threshold will PATROL re-establishes private mappings and page protection.

5 Evaluation

We perform our evaluations 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.

The evaluation focuses on answering the following questions:

- How effective is DETECTIVE at finding false sharing and guiding programmers to their resolution?
- How effective is PATROL at preventing false sharing? That is, what is its performance compared to `pthreads`?

5.1 DETECTIVE Effectiveness

This section evaluates whether DETECTIVE can be used to find false sharing problems, both in synthetic test cases and in actual applications.

5.1.1 Microbenchmarks

We first developed a number of simple test cases that exemplify false sharing problems and evaluate whether DETECTIVE can correctly detect them. For comparison, we also present the corresponding results of Intel's Performance Tuning Utility (PTU), version 3.2. Due to space limitations, we omit the code for these microbenchmarks.

Table **??** presents results of this evaluation. We can see that 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 benchmark 4 and benchmark 5. Note that benchmark 5 triggers a false positive due to heap object reuse: the two different allocations happen to occupy the same address. DETECTIVE avoids this false positive by cleaning up invalid counting information.

5.1.2 Actual Applications

In order to verify whether DETECTIVE can be used to detect false sharing problems in actual applications, we run DETECTIVE with the PHOENIX [**?**] and PARSEC [**?**] benchmark suites.

We use the simlarge inputs for all applications of PARSEC. For PHOENIX, we chose parameters that allow the

Microbenchmark	Performance-Sensitive False Sharing?	DETECTIVE	PTU
1. **False sharing (adjacent objects)**	Yes	✓	✓
2. **Pseudo sharing (array elements)**	Yes	✓	✓
3. **True sharing**	No		
4. **Non-interleaved false sharing**	No		✓
5. **Heap reuse (no sharing)**	No		✓

Table 1: False sharing detecting results using PTU and DETECTIVE. DETECTIVE correctly reports only actual false sharing instances with a performance impact.

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

Figure 5: A fragment of source code from `reverse_index`. False sharing arises when adjacent threads modify the `use_len` array.

programs to run as long as possible. [1]

Using DETECTIVE reveals that six benchmarks (out of sixteen tested) have some false sharing issues, **XXXX: tongping** totally there are 14 objects, but only 4 of them should be taken care. DETECTIVE can detect false sharing problems in `word_countreverse_index`, `kmeans` in PHOENIX suite and `canneal`, `fluidanimate`, `streamcluster` in PARSEC suite (more information can be seen in Table **??**). Those false sharing problems detected includes inter-object and inside-object false sharings. But some of them are not significant, the average interleaving times per cache line is lower than 10 times, which can be easily filtered using DETECTIVE. **XXXX. end tongping**

In `reverse_index` and `word_count`, multiple threads repeatedly modify the same heap object. The pseudo code for these two benchmarks are listed in Figure **??**. We can use thread-local copy to avoid the false sharing problem here; each thread can modify a temporary variable first and then modify the global `use_len` in the end of thread.

`Linear_regression`'s false sharing problem is a little different (see Figure **??**). Two different threads write to the same cache line when the structure `lreg_args` is not cache line aligned. This problem can be avoided easily by padding the structure `lreg_args`.

[1]As of this writing, we were unable to successfully compile `raytrace` and `vips`, and SHERIFF is currently unable to run `x264`, `bodytrack`, and `facesim`. Freqine currently can not support pthreads.

```
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 6: A fragment from `linear_regression`. Each thread is passed in a different address (`struct lreg_args`) and each thread can work on its corresponding `args_in`. Unfortunately, `struct lreg_args` is smaller than a cache line (52 bytes) so two different threads write to the same cache line simultaneously.

The false sharing problem detected in `streamcluster` (one of the PARSEC benchmarks) is similar to the false sharing problem in `linear_regression`; two different threads are writing on the same cache line. In fact, the author tried to avoid the false sharing problems and make every stride a multiple times of cache line size. But the default cache line size is 32 bytes, which is different from the actual physical cache line size that we are used in evaluation (64 bytes). By simply setting the `CACHE_LINE` macro to 64 bytes, it is possible to avoid this false sharing problem completely.

The performance of these four benchmarks is listed in Table **??**, before and after fixing the false sharing issues that DETECTIVE identified. To explain why there is so much difference in performance improvement, we also modified the code to count the possible updates caused

Benchmark	**Old** (s)	**New** (s)	**Speedup**	**Updates** (M)
linear_regression	9.1	0.89	1022%	1323.6
reverseindex	2.10	2.08	100.9%	0.4
word_count	2.19	2.15	101.8%	0.3
streamcluster	2.8	2.5	112%	28.7

Table 2: Performance data for 4 false sharing benchmarks. "Old" column shows the runtime (s) before we fix the false sharing problem and "New" column shows the runtime (s) after fix. All data are obtained using the standard `pthreads` library. "Updates" shows how many million updates (in total) occurred on falsely-shared cache lines.

by these false sharing objects. Updates listed here are the maximum possible number of interleaving writes of these objects (the actual number of interleaving writes depending on scheduling issues).

The benchmarks `reverse_index` and `word_count` do not exhibit substantial improvements after fixing false sharing because the number of updates is not very large. For example, the maximum number of interleaved updates for `reverse_index` is 416,000. However, for `linear_regression`, the number of updates is much larger: over 1 billion. However, even the relatively low numbers of updates indicates that eventually (as the number of threads grow, or for NUMA architectures), this false sharing will become problematic.

5.1.3 DETECTIVE vs. PTU

In order to show how effectively DETECTIVE can find false sharing problems, we compare the results with Intel's Performance Tuning Utility (PTU). PTU is a comercial product which we believe represents the state of art for detecting false sharing problems.

We focus on two things in this comparison:

- How many items are reported by different tools?
- How effective is the tool at helping us find actual false sharing problems?

Reporting Items For PTU, we list the numbers of cache line having false sharing problems (marked with pink color by the tool). To locate one false sharing problem, a programmer must examine every one of these reports. To compare, we list the number of cache linens and objects in the same time.

From the results listed in Table **??**, we can see that DETECTIVE will impose much less manual effort to check those false sharing problems. Across all of the benchmarks, PTU indicates the need to examine 2647 cache

Benchmark	**PTU**	**DETECTIVE**	
	Lines	Lines	Objects
histogram	0	0	0
kmeans	1916	198	2
linear_regression	5	5	1
matrix_multiply	468	0	0
pca	45	0	0
reverseindex	N/A	5	5
string_match	0	0	0
word_count	4	4	3
blackscholes	0	0	0
canneal	1	69317	1
dedup	0	0	0
ferret	0	0	0
fluidanimate	3	1119	1
streamcluster	9	26	1
swaptions	196	0	0
Total	2647	70674	14

Table 3: Detection results of PTU and DETECTIVE. For PTU, we show how many cache lines are marked as falsely shared. **XXXX : tonpging.** For DETECTIVE, we show both cache lines and objects. Actually the parameter can adjust easily to report those significant false sharing problem only. The item marked as "N/A" means PTU fails to show results because it runs out of memory.

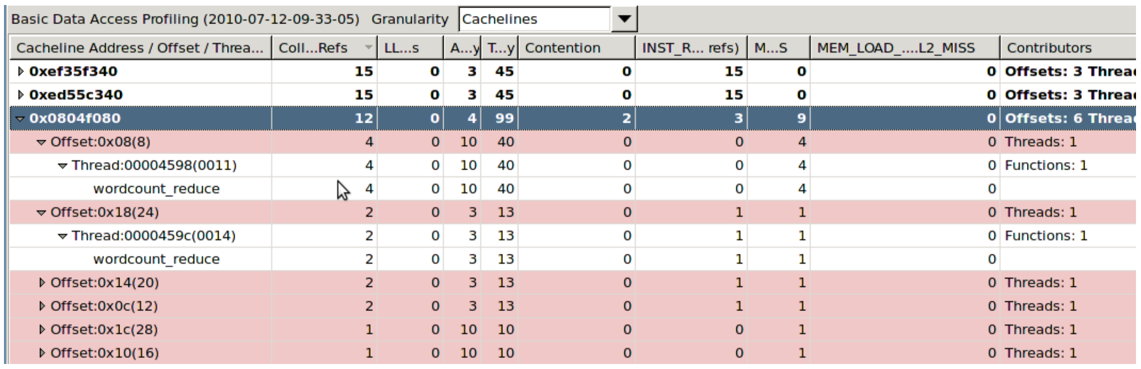
lines overall, (not including `reverse_index`) but DETECTIVE only indicates the need to check only 14 objects. By adjsuting the parameter, SHERIFF can report only 4 objects only: all of which are actually false sharing problems.

Several factors lead to this difference. First, DETECTIVE reports corresponding objects instead of cache lines, which can reduces the number of reports when one callsite has a huge number of inter-objects' allocations (`kmeans`) and one object spans multiple cache lines (`reverse_index`). Second, Sheriff distinguishes true from false sharing, reducing the number of reported items. Finally, DETECTIVE only reports those objects with interleaving writes larger than a threshold number, which significantly reduces the number of reports.

Ease of locating false sharing problems To illustrate how DETECTIVE can precisely locate false sharing problems, we use one benchmark (`word_count`, a PHOENIX benchmark) as an example. Our experience with diagnosing other false sharing issues is similar.

Here is an example output from DETECTIVE from `word_count`.

```
1st object, cache interleaving writes
13767 times (start at 0xd5c8e140).
Object start 0xd5c8e160, length 32.
```


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 7: PTU output for `word_count`.

```
It is a heap object with callsite:
callsite0:./wordcount_pthreads.c:136
callsite1:./wordcount_pthreads.c:441
```

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

```
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 by using the thread-local data copies is now straightforward [?].

By contrast, compare PTU's output in Figure **??**. Finding this problem is far more complicated with PTU, since it only presents functions using each cache line, not to mention the fact that PTU can report huge numbers of false positives. Another shortcoming of PTU is that "Collected Data Refs" number cannot be used as a metric to evaluate the significance of false sharing problems. For this example, PTU only reports 12 references (versus 13767 times for DETECTIVE).

5.2 Performance of DETECTIVE

This section shows the runtime overhead of DETECTIVE versus `pthreads` across two multithreaded benchmarks suites, PHOENIX and PARSEC. The results can be seen from Figure **??**.

`linear_regression` exhibits almost a $10\times$ speedup against the one using `pthreads` library even with the added overhead of sampling and other mechanisms of DETECTIVE. There is a serious false sharing problem inside (see Table **??**) which both DETECTIVE and PATROL eliminate automatically.

There are two benchmarks on which DETECTIVE does not perform well. One is `canneal`, the performance overhead of DETECTIVE on this benchmark is about 7X slower than the one using `pthreads` library. Another one is `fluidanimate`, the performance overhead is about 14X slower than that using `pthreads`.

Benchmark	**Trans** #	**DirtyPages** M	**Runtime** s
canneal	930	2.9	74.3
fluidanimate	18696114	2.15	21.56

Table 4: Characteristics of slower benchmarks in DETECTIVE.

According to our analysis, the transaction number and dirty pages are two main causes of the overhead. For most of time, more transaction number can cause more dirty pages. In order to find out what can affect the performance of these two benchmarks, we get some characteristics data (see Table **??** about these two benchmarks when they are using DETECTIVE.

From the table, we can easily find out that these two benchmarks share the same attribute, having large amount of dirty pages. For one dirty page, DETECTIVE need two protections, creation of the Copy-On-Write version and different version of twin pages, checking the false sharing problem inside every periodical checking cycle and commits to the shared mapping. Given large amount of dirty pages, copying alone is very expensive since one dirty page needs at least 3 copies. For `fluidanimate`, 2.2 million pages needs about 6.8 million copies, which can acount for about 20 seconds copying overhead since copying one gigabyte of memory takes approximates 0.75 seconds. shared pages, which leads to substantial overhead. Examination of the source code of `fluidanimate` reveals a large number of lock calls, SHERIFF replaces lock calls with their interprocess variants and triggers a transaction end and begin for each, adding some overhead if there are some shared pages.

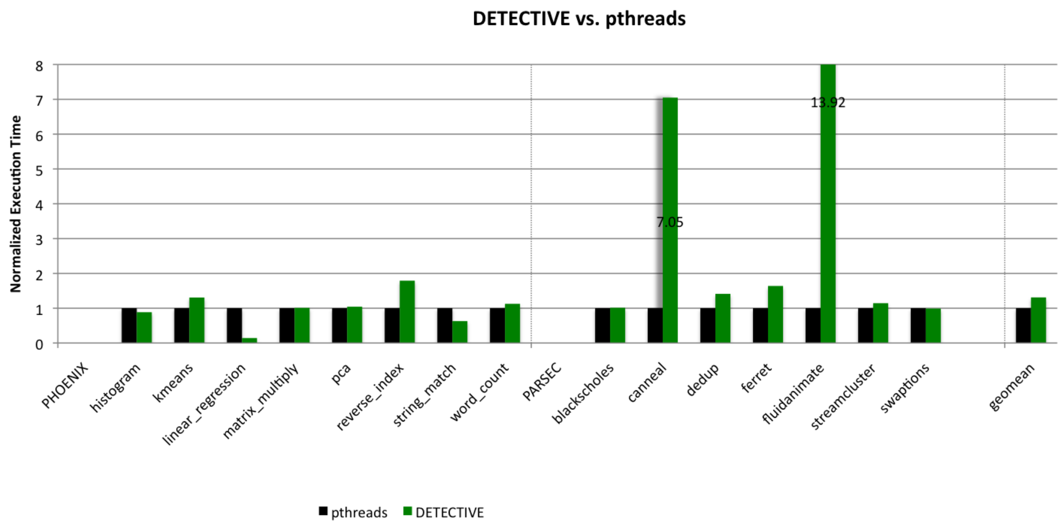


Figure 8: DETECTIVE overhead across two suites of benchmarks, normalized to the performance of the `pthreads` library (lower is better). With two exceptions, its overhead is acceptably low.

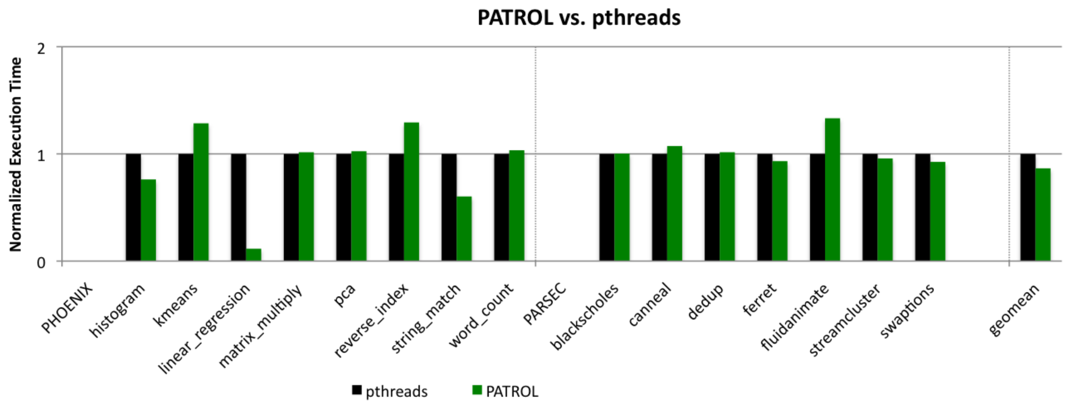


Figure 9: PATROL performance across two suites of benchmarks, normalized to the performance of the `pthreads` library (see Section **??**). In case of catastrophic false sharing, DETECTIVE dramatically increases performance.

5.3 Performance of PATROL

Here, we examine the performance improvement by tolerating the false sharing problems in PATROL. The performance improvement can be seen in Figure **??**.

For most of the benchmarks, PATROL either has no effect on performance or improves it. The most dramatic improvement comes from `linear_regression`, which runs 10× faster than with `pthreads` because PATROL eliminates its serious false sharing problem (see Table **??**). `string_match` runs 40% faster because of false sharing caused by the `pthreads` heap allocator (which is why DETECTIVE does not find it). `histogram` also runs substantially faster with PATROL (24%), although we currently are not certain why this is the case.

There are three benchmarks for which PATROL degrades performance by as much as 30% versus `pthreads`. `kmeans` over 375 threads per second. While process creation is relatively cheap, it is still more expensive than creating threads, so the cost of process creation and destruction degrades performance.

`reverse_index` and `fluidanimate` exhibit slowdown primarily due to a technical detail of the processes-as-threads framework. This performance impact arises from the use of a file-based mapping, which connects 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 status of heap-allocated pages). The first write on a file-mapped page repopulates information from the file's page table entry. Also, 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 there is only one user. We plan to investigate modifying the kernel to support an additional mapping mode to eliminate this overhead.

6 Related Work

6.1 Code Profiling and Data Profiling

While most existing profilers can identify cache misses (e.g., OProfile [**?**], Gprof [**?**]), they typically cannot distinguish between false sharing and true sharing.

The closest related work to this paper not mentioned previously is DProf [**?**]. DProf attempts to associate memory addresses with data types to locate cache problems related to the same type of object and is designed to help identify the flow of data moving from one core to the other. DProf requires some 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, DETECTIVE requires no manual intervention and precisely identify false sharing regardless of the flow of data or which data types involved.

6.2 False Sharing Detection

Past work on false sharing uses three different approaches. The first uses a hardware simulator to approximate the running of applications on actual hardware and to provide complete information about cache misses [**?**]. The second uses binary instrumentation. Pluto [**?**] uses the Valgrind framework to collect the sequence of load and store events on different threads and gives a worst-case estimate of possible false sharing. Liu [**?**] relies on Pin to collect memory access information and then reports total false sharing miss information. None of these systems can differentiate true sharing from false sharing, are misled by aliasing due to memory object reuse, and suffer from significant performance overhead (e.g., running 200$\times$ slower).

The third approach is based on event-based sampling, as in Intel's performance tuning utility (PTU) [**?**, **?**], which operate at far lower cost in execution time. PTU can discover shared physical cache lines, and provide some indication about possible false sharing problems caused by different functions. Like other systems (but unlike DETECTIVE), PTU can suffer from numerous false positives caused by aliasing due to reuse of heap objects, and reports false sharing instances that have no impact on performance. Also unlike DETECTIVE, PTU cannot differentiate true from false sharing or pinpoint the source of false sharing problems.

6.3 False Sharing Avoidance

It is well-known that false sharing problems can affect the performance of application greatly [**?**, **?**]. Jeremiassen and Eggers [**?**] describe a compiler transformation that adjusts the memory layout of applications though the computation of memory access pattern. Chow et al. [**?**] select different scheduling parameters for parallel loops. Berger et al. describe Hoard, a scalable memory allocator that eliminate false sharing caused by collocation of heap objects [**?**]. None of these tools can avoid false sharing to the extent that PATROL does.

6.4 Processes-as-threads

As described earlier, SHERIFF borrows and significantly extends the process-as-thread model first employed by Grace [**?**]. Grace is a process-based approach designed to tolerate concurrency errors, such as deadlock, race conditions, and atomicity errors by imposing a sequential semantics on multithreaded programs. Like SHERIFF, Grace exploits the use of processes as threads to provide complete separation and to capture read/write information from different threads. However, Grace has an entirely different target and is restricted to fork-join programs without inter-thread communication. SHERIFF extends the key insight of Grace of using processes to replace threads, but generalizes it to handle arbitrary multithreaded programs.

7 Conclusion

This paper presents SHERIFF, a software-only framework that enables per-thread memory protection and isolation. SHERIFF accomplishes this by converting threads into processes, isolating writes, and merging updates using a twinning mechanism inspired by distributed shared memory systems. We use SHERIFF to build two tools to attack the problem of false sharing. DETECTIVE precisely identifies false sharing instances with no false positives, and reports allocation call site information for any heap object responsible for false sharing problems with a possible performance impact. We show that DETECTIVE is useful in tracking down and resolving false sharing problems in an existing benchmark suite. PATROL reduces the performance impact of false sharing in unaltered applications through adaptive activation of SHERIFF's memory isolation and protection mechanisms and can in some cases dramatically improve program performance.