Sheriff: Detecting and Eliminating False Sharing

Tongping Liu Emery D. Berger

Dept. 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 prohibitively slow (degrading performance by 200X), but also cannot 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 presents Sheriff, a software-only system that both withstands and identifies false sharing in C/C++ applications. As a false sharing resistant runtime system, Sheriff replaces the standard pthreads library and eliminates false sharing, dramatically improving performance in the face of catastrophic false sharing (in one case, by almost 10X versus pthreads). As a false sharing detection tool, Sheriff precisely identifies the sources of false sharing with no false positives and low overhead. A case study with the Phoenix and PARSEC benchmark suites shows that Sheriff can quickly identify false sharing and guide programmers to remove it.

1. Introduction

Writing multithreaded programs is challenging for a variety of reasons. They are famously difficult to debug and present numerous opportunities for error, including races, atomicity violations, and deadlock. However, getting a multithreaded program to run correctly is not the only challenge. The primary reason to write multithreaded programs is to improve performance, either by hiding latency or by taking advantage of multiple processing cores.

Reducing contention for shared resources is crucial to ensure that multithreaded applications scale. However, contention can arise even when multiple threads are accessing different objects. This *false* sharing arises when multiple, logically distinct objects happen to reside on the same cache line. For example, one thread may update object A, while another thread updates object B. If these objects happen to reside on the same cache line, the cache coherence protocol will invalidate the entire cache line containing A and B, causing subsequent accesses to miss on both processors.

When false sharing happens frequently enough, the resulting "ping-ponging" of cache lines across processors can cause performance to plummet [7, 11]. False sharing can slow a synthetic microbenchmark by up to 160X, and slows one actual benchmark from the Phoenix suite by 10X (see Section 7). The trend towards increasingly larger cache lines together with an expected increase in the number of multithreaded applications is conspiring to make false sharing an increasingly common phenomenon.

False sharing is of course not new, and there have been several past approaches to deal with it. One approach is to attempt to eliminate false sharing altogether. The compiler can minimize false sharing by adjusting memory layouts (e.g., through padding and alignment), or alter parallel loop scheduling to avoid sharing [8, 14]. However, the effectiveness of these techniques is limited to array-based scientific codes. For more general-purpose applications, a scalable dynamic memory allocator can reduce the likelihood of false sharing of distinct heap objects [3], but cannot prevent false sharing within individual heap objects.

Since automatically eliminating false sharing is difficult, other past work has focused on detecting it. These tools operate on binaries, either via simulation [21], binary instrumentation [18], or hardware performance monitors [12, 13], and intercept all reads and writes in order to detect false sharing.

Not only are some of these tools prohibitively slow (imposing order-of-magnitude slowdowns), none of them provide the programmer with sufficient information. They fail to pinpoint the source of false sharing problems, at best pointing to particular addresses accessed by functions, leaving it to the programmer to discover what these addresses correspond to.

Worse, these previous tools report numerous false positives. First, they can report objects that were falsely shared so few times that they do not present a performance bottleneck. Second, because these tools are oblivious to object reuse by memory allocators, they incorrectly aggregate access information when one address is reused for multiple objects. Third, they are unable to distinguish true sharing, which is generally intended by the programmer, from false sharing, which is not.

Contributions

This paper presents **Sheriff**, a software-only runtime system that both eliminates and detects false sharing in C/C++ applications. Sheriff acts as a plug-in replacement runtime system for the pthreads library. Sheriff can be used in two modes: to eliminate false sharing automatically, or to detect it.

When used as a false sharing resistant runtime, Sheriff can dramatically increase performance in the face of catastrophic false sharing. On an 8-core system, Sheriff accelerates one benchmark by approximately 10X versus the standard pthreads library. To our knowledge, Sheriff is the first false sharing resistant runtime for shared-memory systems.

When used in detection mode, Sheriff detects false sharing 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. Sheriff 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. As a detection tool, Sheriff imposes a 2% performance penalty on average and a worst-case overhead of under 4X, making it both precise and efficient.

The remainder of this paper is organized as follows. Section 2 gives an overview of Sheriff, including its key mechanisms. Section 3 describes Sheriff's runtime mode and how it prevents false sharing. Section 4 describes Sheriff's detection mode and how it works to locate false sharing problems. Section 5 discusses some key performance optimizations and Section 6 presents some of Sheriff's limitations. Section 7 then presents experimental results on microbenchmarks and across two multithreaded benchmark suites (Phoenix and PARSEC), then presents a case study of using Sheriff to detect and guide the correction of false sharing in these suites. Section 8 describes key related work, and Section 9 concludes with future directions.

2. Sheriff Overview

Sheriff replaces the pthreads library and the memory allocator with a runtime system that allows it to eliminate the effects of false sharing, and, in detection mode, to indicate which objects or fields within objects are causing performance problems.

2.1 Definitions and Goals

False sharing can take two different forms. It can arise due to sharing of structurally-unrelated objects that happen to be located on the same cache line (i.e., different variables), or when multiple processors access different fields of the same object (which Hyde and Fleisch refer to as "pseudo sharing" [11]). Sheriff aims to either eliminate or detect both of these types of sharing, which, for simplicity of exposition, the remainder of this paper refers to simply as false sharing.

However, not all cases of false sharing are created equal. The presence of false sharing does not always lead to performance problems. The effect of false sharing on performance depends on the memory access pattern of the application. False sharing only causes significant performance degradation when multiple threads repeatedly update falsely-shared data, because updates lead to invalidations and thus cache misses.

Sheriff thus focuses exclusively on eliminating or identifying those false sharing cases that lead to numerous invalidations and thus have the potential to seriously degrade performance.

Reporting Detection of false sharing is only the first step. To maximize its usefulness to the programmer, a false sharing tool should identify falsely-shared objects precisely. Sheriff's goal is 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.

2.2 Insights

To achieve the goals outlined above, Sheriff leverages the following two key insights.

- **Delaying updates avoids false sharing.** We observe that *delaying updates so that they do not cause invalidations to other threads eliminates the performance impact of false sharing*. 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.
- **Updates outside critical sections are false sharing.** We also observe that, in a correctly-synchronized program, *the only updates outside of critical sections are to unshared data*. In other words, any sharing outside of a critical section in a race-free program must be false sharing.

2.3 Mechanisms

Processes as Threads As mentioned above, it is possible to prevent false sharing from impacting performance by delaying each thread's updates. To accomplish this, Sheriff converts threads to processes, an approach first used by Grace (a runtime system that ensures safety for multithreaded C/C++ programs [4]). By suitably sharing the heap and globals via copy-on-write shared memory, processes can simulate threads, but in isolation from each other until their individual updates are merged. This approach has the effect of delaying all updates and eliminates the performance impact of false sharing.

Replacing pthreads with processes is surprisingly inexpensive, especially on Linux where both pthreads and processes are invoked using the same underlying system call (see Berger et al. for extensive timing results [4]).

More importantly, using processes rather than threads 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. Converting threads to processes also effectively enables the use of page protection effectively on a per-thread basis, allowing Sheriff to track memory accesses by different threads and thus detect false sharing.

To maintain the shared-memory semantics of a multithreaded program, Sheriff must eventually update the shared image so that threads can see each other's modifications. Sheriff delays these updates until synchronization points (e.g., lock acquisition and release).

However, the granularity of pages is too coarse: updates to different areas of the same page appear to conflict, leading to a question of how we manage updates by different threads to the same page, and adding an unacceptable amount of imprecision to false sharing detection.

Twinning and Diffing Sheriff assumes that it only observes race-free executions. Since races are generally rare, especially in about-to-be deployed applications whose performance is being debugged, we consider this to be a reasonable assumption.

Therefore, outside of critical sections, Sheriff only needs to write back its modifications rather than the full contents of any pages it has modified.

To achieve this, Sheriff adapts the twinning and diffing mechanism first used in distributed shared memory systems to reduce communication overhead [15]. In Sheriff, twinning and diffing allows it to identify modifications on a word-by-word basis.

Figure 1 presents an overview of both mechanisms at work. 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 and the modified page byte by byte to compute diffs.

The next sections describe Sheriff's threads-as-processes and twinning mechanisms in detail.

3. Threads as Processes

This section discusses the details of replacing threads with processes. Sheriff significantly extends Grace's mechanisms, which only supported a restricted class of multithreaded programs. Sheriff provides support for general-purpose multithreaded programs, including full support for file I/O and cross-thread synchronization.

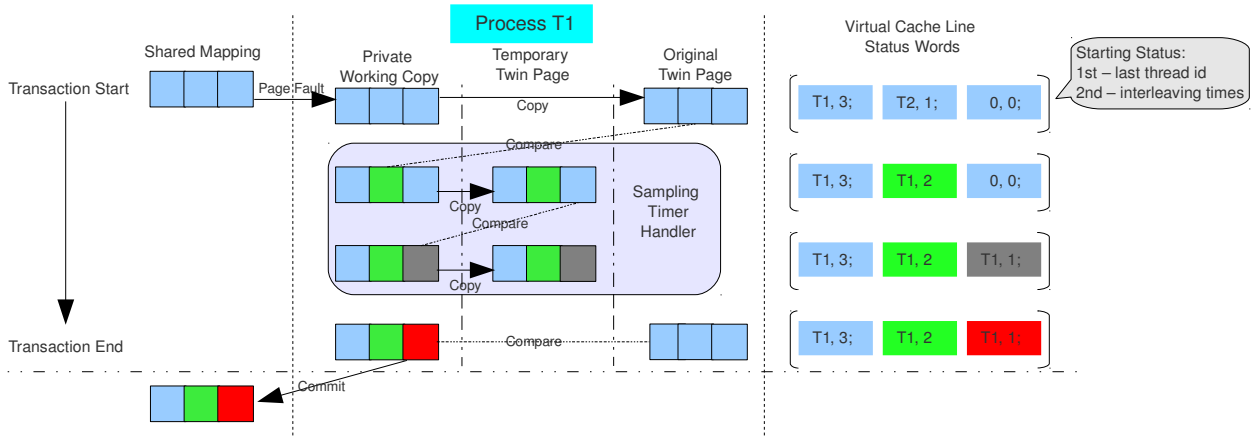

Figure 1. Overview of Sheriff execution (Section 3.5). Sheriff simulates threads using processes (Section 3). Each process operates on private copies of data and commits diffs to shared mappings at synchronization points (Section 3.5.3) In detection mode, Sheriff uses virtual cache line status words (Section 4.3) and sampling (Section 4.2) to detect frequent false sharing within cache lines.

3.1 Thread Creation

Like Grace, Sheriff intercepts the `pthread_create()` call and replaces it with a process creation call. While Grace used `fork()`, Sheriff uses the Linux system call `clone()` directly; see Section 3.3 for details.

3.2 Shared Address Space

In order to create the illusion 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 try to 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 handles updating the shared image at synchronization points, as described in Section 3.5.

Globals and Heap Sheriff uses the same global and heap organization as Grace, which we describe here briefly.

Sheriff uses a fixed-size (larger than normal globals size) file 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 1.6GB). Memory allocations requirements from user applications are satisfied from this fixed-size private mapping.

Since different threads can get the memory from this fixed-size mapping, the heap 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 [3] and built using HeapLayers [5]. Sheriff divides the heap into a fixed number of sub-heaps (currently 16). Each thread uses a hash of its process id to obtain the index of the heap that can be used to satisfy memory allocations. 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 spawnWithShareFiles{
  return syscall(SYS_clone,
   CLONE_FS|CLONE_FILES|SIGCHLD,(void*)0);
}
```

Figure 2. Pseudo-code (for Linux) to create a new process with shared file descriptors but a distinct address space.

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

Figure 3. Pseudo-code for all synchronization operations (where "sync" denotes the appropriate operation).

3.3 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 2), child processes can share the same physical file descriptor table with the parent process while not sharing the same address space.

3.4 Synchronization

Sheriff also goes beyond Grace in providing support for synchronization (in Grace, locks are treated as no-ops). Sheriff handles the following synchronization operations: *mutex*, *conditional variable*, *barrier* and *thread join*.

At each synchronization point, Sheriff must commit all changes made to individual pages. The span between synchronization points

is thus a single atomic transaction. Note that Sheriff's approach differs significantly from previous transactional memory proposals [16], 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.

In order 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 accessed by different processes.

Figure 3 presents a template for how Sheriff wraps synchronization operations. For example, a call to `pthread_mutex_lock` first ends the current transaction. It then calls the corresponding pthreads library's functions but on a process-wide mutex object. Sheriff then begins a new transaction after the lock is acquired (which will end when the corresponding lock is unlocked).

Conditional variable and barriers use the same mechanism, but `pthread_join` is slightly different. When joining, Sheriff ends the current transaction and calls `waitpid()` to wait for the appropriate process to complete.

3.5 Example Execution

Figure 1 presents an overview of Sheriff's execution.

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

3.5.1 Transaction Begin

In 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.

3.5.2 Execution

While performing reads, Sheriff runs almost the same speed as that of a conventional multithreaded program. A 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 write-able so that future accesses on this page can run at a full speed (won't invoke page fault any more). Thus, one page incurs only one page fault in one transaction. Although protection faults and signal faults are expensive, those costs are amortized across the whole transaction.

However, 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 (Section 2.3 describes the motivation and use of twins for diffing).

3.5.3 Transaction End

At the end of each atomically-executed region—the end of each thread, right before and end of those synchronization points, right before a thread spawn, and right before joining another thread—Sheriff acquires a commit lock, commits changes from private pages to the shared space, releases the commit lock, and then reclaims memory holding old private pages and twin pages.

Sheriff commits only the differences between the twin and the modified pages, as described in Section 2.3. Once it has written these diffs, Sheriff issues an madvise call with the `MADV_DONTNEED` flag that discards the current physical pages backing both the private mapping and twin pages.

4. Detecting False Sharing

While the mechanisms described so far allow Sheriff to withstand false sharing, they do not address how it can detect it. This section describes in detail how Sheriff identifies problematic false sharing; the mechanisms used here are only active when Sheriff is used in detection mode.

4.1 Discovering False Sharing

When Sheriff 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, Sheriff 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.

4.2 Identifying Problematic False Sharing

To precisely measure the impact of observed false sharing instances, Sheriff uses a sampling-based approach. There is a balance between choosing a finer sampling period and performance overhead. Sheriff currently uses 10 microseconds as its sampling interval.

Sheriff counts the number of writes by associating a temporary twin page with each shared dirty page (see Figure 1). 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). Sheriff 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).

4.3 Capturing Cache Invalidations

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

In order to capture interleaved writes on caches, Sheriff uses virtual cache line status words (Figure 1). Sheriff 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, Sheriff updates the associated status word with both the thread id and the version number. In the current implementation, Sheriff splits the status word into two different arrays to allow the use of atomic operations instead of locks (see Figure 4).

4.4 Identifing Objects inside Cache Lines

At this point, Sheriff has detected individual cache lines that are responsible for a large number of invalidations, and thus potential

```
void recordCacheInvalidates(int cacheNo) {
    int myTid = getpid();
    int lastTid;

    // Try to check last thread to modify this cache.
    lastTid = atomic_exchange(&LastThreadModifyCache[cacheNo], myTid);
    if(lastTid != myTid) {
       // Increment cache invalidation only when current thread is different.
       atomic_increment(&cacheInvalidation[cacheNo]);
     }
}
```

Figure 4. Record the cache invalidation atomically.

sources of slowdowns. The next step is identifying the culprit objects.

Sheriff identifies globals directly by using debug information that associates the address with the name of the global.

For heap objects, Sheriff 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 7.1.3 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

Sheriff 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, Sheriff 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 Sheriff 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, Sheriff use one word's higher 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, we set the thread id field to 0xFFFF.

4.6 Reporting Falsely Shared Objects

At the end of program, Sheriff reports those objects causing false sharing problems. Sheriff 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. Later, Sheriff ranks the falsely shared objects by the number of invalidations they caused.

After scanning the cache invalidation array, Sheriff 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.

5. Optimizations

To reduce overhead, Sheriff employs the following performance optimizations:

- Applying memory protection to the entire heap is expensive. To reduce this cost, Sheriff uses a protected heap only for allocations smaller than the number of cores times the cache line size. This heuristic is effective since small objects are more likely to be the source of false sharing because they fit on a cache line.
- Sheriff does not apply memory protection at all when there is just one thread running: when there is only one thread, false sharing cannot occur. Sheriff only initiates protection after a `pthread_create()` call to start a new thread. Also, Sheriff disables protection as soon as it detects that there is only one thread running, which it checks after each call to `pthread_join()`.
- Sheriff uses sampling to capture continuous writes in the same transaction. In the timer handler, Sheriff must compare written pages (dirty) to the committed pages. Since false sharing problems can happen on those pages that are shared by multiple process simultaneously, Sheriff only checks those shared pages to reduce the checking overhead. Thus, Sheriff uses a shared page-based array to track the status of all pages. When one page is faulted because of a write operation, Sheriff increments the number of writers to this page.
- Sheriff uses the shared page version number to improve performance on commits. Notice that Sheriff must compare the corresponding twin page and working page in order to locate and commit only the modifications to the shared mapping. In fact, if one private page's corresponding shared mapping has not been modified by other threads, it is safe to commit the entire content of this page to the shared mapping. Copying the entire page is faster than diffing, so Sheriff does this whenever possible.

  To detect whether there are potential conflicts that would preclude copying, Sheriff associates a version number with each page in shared mapping. Sheriff increments each page's version number after every commit of that page. Before the creation of each twin page, Sheriff saves the corresponding page version number for one page. In the end of transaction, if the saved page version number is still the same as the shared page version number, that means that no one else has committed a new version to corresponding shared mapping, so it is safe to copy the entire working page as next version of shared mapping.

6. Discussion

This section describes some of Sheriff's limitations, which slightly restrict its ability to run certain programs and to find all instances of false sharing.

6.1 Limits on Application Class

As Section 3.2 describes, Sheriff does not share the stack between different threads. When using pthreads, it is possible for different threads to share stack variables allocated by their parent. Sher-

iff currently is not able to run correctly with applications whose threads modify stack variables from their parent thread.

6.2 Precision

Unlike previous tools, Sheriff has no false positives. It can differentiate true sharing and false sharing (see Section 4.5) and avoid false positives caused by heap objects (Section 4.4).

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

- **Heap-induced false sharing.** Sheriff is based on a Hoard-like memory allocator, instead of using pthreads memory allocator. That means that any false sharing problem introduced by pthreads memory allocator can be undetected by Sheriff. Since the use of a memory allocator like Hoard that avoids heap-induced false sharing easily resolves this problem, this limitation is not a problem in practice.
- According to previous section, Sheriff chooses to protect small objects only in order to improve performance. If one false sharing problem unfortunately happens on those large objects, then Sheriff cannot detect the false sharing problems. For the suite of benchmarks explored here, this problem does not arise.
- Since it uses sampling to capture continuous writes from different threads, Sheriff can miss writes that occur in the middle of sampling intervals. We hypothesize that false sharing problems that affect performance are unlikely to only perform frequent writes during that time.

7. 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. Note that those applications are generated for a 32bit environment using "-m32".

All performance data in this section are the average of 10 executions, with the maximum value and minimum values excluded here.

In this section, we are going to answer the following questions:

- How effective is Sheriff at finding false sharing problems?
- What is the performance of Sheriff compared to pthreads?
- What application characteristics affect Sheriff's performance?

7.1 Effectiveness

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

7.1.1 Micro-benchmarks

As Section 2.1 describes, Sheriff can be used to detect false sharing problems including false sharing, pseudo sharing. We first developed test cases that exemplify these problems and evaluate whether Sheriff can successfully detect them. For comparison, we also present the corresponding results of Intel's Performance Tuning Utility (PTU), version 3.2.

Table 1 presents results of this evaluation. We can see that Sheriff reports both false sharing (Figure 5) and pseudo sharing (Figure 6) problems successfully, and correctly ignores the benchmarks (3–5) with no actual false sharing performance impact). However, PTU reports false sharing for benchmark 4 and benchmark 5. Note that benchmark 5 (Figure 9) triggers the typical false positives due to dynamic heap object reuse: the two different allocations happen to occupy the same address. Sheriff avoids this false positive by cleaning up invalid counting information (after the call to `free(pcount1)`).

```
int count1 = 0; int count2 = 0;
void * thread1(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    count1++;
}
void * thread2(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    count2++;
}
```

Figure 5. Benchmark 1 (false sharing)

```
int count[CORE_NUM];
void * thread(void * param) {
  int tid = (int)param;
  for(i = 0; i < COUNT_NUM; i++)
    count[tid]++;
}
```

Figure 6. Benchmark 2 (pseudo sharing)

```
int count = 0;
void * thread(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    count++;
}
```

Figure 7. Benchmark 3 (true sharing)

```
int count1 = 0; int count2 = 0;
void * thread1(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    count1++;
}
void * thread2(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    count2++;
}
int main() {
    spawn(&tid[0], thread1); join(tid[0]);
    spawn(&tid[1], thread2); join(tid[1]);
}
```

Figure 8. Benchmark 4 (noninterleaving-falsesharing).

7.1.2 Actual Applications

In order to verify whether Sheriff can be used to detect false sharing problems in actual applications, we run Sheriff against phoenix [20] and PARSEC [6] benchmark suite.

We used the simlarge inputs for all applications of PARSEC. For Phoenix, we chose parameters that allow the programs to run as long as possible. [1]

According to our experiments, Sheriff have found that four benchmarks (of a total 16) have some false sharing issues. In

[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`.

```
int * pcount1; int * pcount2;
void * thread1(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    pcount1[0]++;
}
void * thread2(void * param) {
  for(i = 0; i < COUNT_NUM; i++)
    pcount2[1]++;
}
int main() {
    pcount1 = malloc(16);
    spawn(&tid[0], thread1);  join(tid[0]);
    free(pcount1);
    // New allocation here.
    pcount2 = malloc(16);
    spawn(&tid[1], thread2); join(tid[1]);
}
```

Figure 9. Benchmark 5 (heap-induced false positive).

Microbenchmark	Perf-Sensitive False Sharing	Sheriff	PTU
Benchmark 1	✓	True	True
Benchmark 2	✓	True	True
Benchmark 3		False	False
Benchmark 4		False	True
Benchmark 5		False	True

Table 1. False sharing detecting results using PTU and Sheriff. *True* indicates the tool reported false sharing, while *False* indicates no reports. Sheriff 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 10. `reverse_index` example. Here different threads can modify the same `use_len` array when there is a new link found.

`reverse_index` and `word_count`, multiple threads repeatedly modify the same heap object. The pseudo code for these two benchmarks are listed in Figure 10. 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 11. 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`.

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. But the reason is different. In fact, the author tried to avoid 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

```
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 11. `linear_regression` false sharing example code. In the creation of thread, each thread will be passed in a different address (`struct lreg_args`) and each thread can work on its corresponding `args_in`. But unfortunately, the size of `struct lreg_args` is not cache line aligned (52 bytes) and that causes two different threads to write to the same cache line simultaneously.

Benchmark	Old (s)	New (s)	Speedup	Updates (M)
linear_regression	9.116	0.89	1024.3%	1323.6
reverseindex	5.449	5.427	100.41%	0.4
word_count	2.188	2.151	101.72%	0.3
streamcluster	2.825	2.501	112.95%	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 got based on the same pthreads library. "Updates" shows how many million updates (in total) occurred on falsely-shared cache lines.

`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 2, before and after fixing the false sharing issues that Sheriff identified. To explain why there is so much difference in performance improvement, we also modified the code to count the possible updates caused 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.

7.1.3 Comparison between Sheriff and PTU

In order to show how effectively Sheriff 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?

Benchmark	PTU # Cachelines	Sheriff # Objects
histogram	0	0
kmeans	1916	0
linear_regression	5	1
matrix_multiply	468	0
pca	45	0
reverseindex	N/A	1
string_match	0	0
word_count	4	1
blackscholes	0	0
canneal	1	0
dedup	0	0
ferret	0	0
fluidanimate	3	0
freqmine	0	0
streamcluster	9	1
swaptions	196	0
Total	2647	4

Table 3. Detection results of PTU and Sheriff. For PTU, we show how many cache lines are marked as falsely shared. For Sheriff, we show how many objects are reported by Sheriff (with interleaving writes larger than 100). The item marked as "N/A" means PTU fails to show results because it runs out of memory.

- 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. For Sheriff, we list the number of objects reported by Sheriff.

From the results listed in Table 3, we can see that Sheriff 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 lines overall, (not including `reverse_index`) but Sheriff only indicates the need to examine 4 cache lines: all of which are actually false sharing problems.

Several factors lead to this difference. First, Sheriff distinguishes true from false sharing, dramatically reducing the number of reported items. Second, Sheriff only reports those objects with interleaving writes larger than a threshold number, which significantly reduces the number of reports. Third, Sheriff reports corresponding objects instead of cache lines, which also reduces the number of reports if one object spans multiple cache lines.

Ease of locating false sharing problems To illustrate how Sheriff 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 Sheriff from `word_count`.

```
1st object, cache interleaving writes
13767 times (start at 0xd5c8e140).
Object start 0xd5c8e160, length 32.
It is a heap object with callsite:
callsite 0:./wordcount_pthreads.c:136
callsite 1:./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 [13].

By contrast, compare PTU's output in Figure 12. 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 Sheriff).

7.2 Performance of Sheriff

We have evaluated Sheriff in both modes (eliminating and detecting) on two multithreaded benchmarks suites, Phoenix and PARSEC. The results can be seen from Figure 13. There are two outliers in the results. One is `linear_regression`, which exhibits almost a 10X speedup against the one using pthreads library. There is a serious false sharing problem inside (see Table 2) which Sheriff eliminates automatically. Even in detection mode, with the overhead of sampling, etc., Sheriff achieves a significant performance benefit. Another outlier is the ferret benchmark. The performance overhead of Sheriff on this benchmark is about 4.97X slower than the one using pthreads library.

In order to find out what can affect Sheriff's performance, we measured charecteristics of our benchmark suites in Table 4. According to our analysis, the following parameters can affect the performance of Sheriff.

- Pages written: each write on a protected page imposes additional overhead to unprotect the page in the page fault handler. In the sampling handler, Sheriff must check for cache writes for each shared written page, and at the end of transaction, Sheriff must check cache writes for each page and commit the modification to the shared space.
- Transaction length: Sheriff introduces overhead in the beginning of transaction and in the end of each transaction. Longer transactions amortizes this overhead.
- Allocation times: Sheriff (in detection mode) attaches callsite information for every allocated object, slowing allocation.
- Cache cleanup size: Sheriff cleans up the invalid cache counting information in the memory allocation if one allocation is involving in the re-usage of memory of those freed memory objects.

From the results from Table 4, we can confirm our analysis. Allocation times and cache cleanup size have little impact on performance. However, when the number and rate of pages written is large, performance suffers.

Figure 13 shows that Sheriff's overhead is highest for the following four benchmarks: benchmarks `ferret`, `reverse_index`, `dedup` and `fluidanimate`. Characteristics showed in Table 4 that the first three benchmarks have a very high rate of page updates (**PagesPerMs**). `fluidanimate` is an outlier if we are just using the **PagesPerMs** metrics. The reason of `fluidanimate` has a high overhead is that there are huge amounts of transactions inside (about 10M). Examination of the source code revealed a large number of lock calls in this application. Sheriff replaces lock calls with their interprocess variants and triggers a transaction end and begin for each, adding overhead. The worst case for Sheriff is exemplified by `ferret`, which modifies a huge number of pages (about 3.45G) and has a large number of transactions (about 1M).

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 12. PTU output for `word_count`.

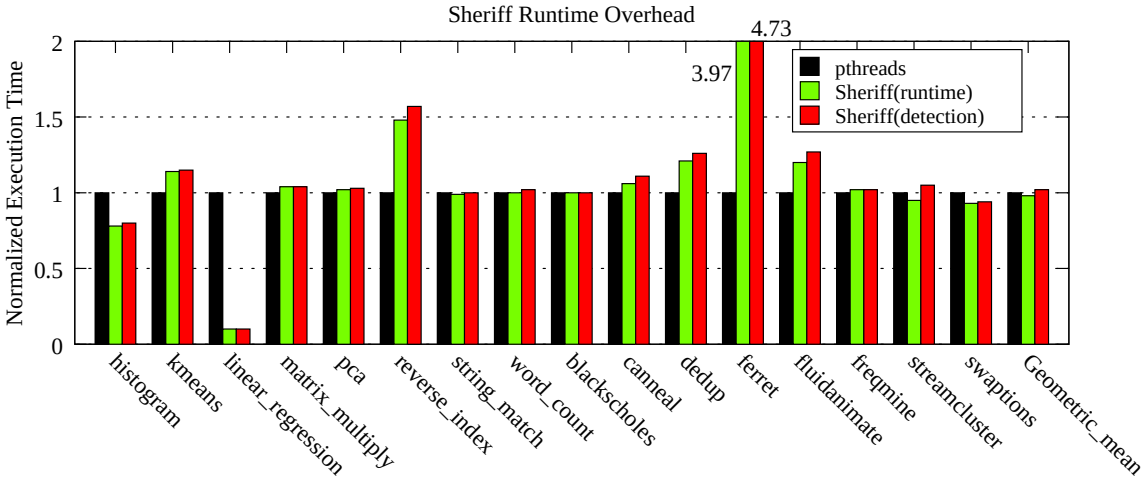

Figure 13. Runtime overhead for Sheriff (in both modes) across a suite of benchmarks, normalized to the performance of the pthreads library (see Section 7.2). In case of catastrophic false sharing, Sheriff dramatically increases performance; with the exception of a notable outlier, its overhead is acceptably low.

8. Related Work

8.1 Code Profiling and Data Profiling

While most existing profilers can identify cache misses (e.g., OProfile [17]), they typically cannot distinguish between false sharing and true sharing.

The closest related work to this paper not mentioned previously is DProf [19]. 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, Sheriff requires no manual intervention and precisely identify false sharing regardless of the flow of data or which data types involved.

8.2 False Sharing Detection

Intel's performance tuning utility (PTU) [12, 13] utilizes event-based sampling to discover instructions sharing physical cache lines, and can give some indication about possible false sharing problems caused by different functions. Unlike Sheriff, 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 Sheriff, PTU cannot differentiate true from false sharing.

8.3 False Sharing Avoidance

It is well-known that false sharing problems can affect the performance of application greatly [14, 8]. Jeremiassen and Eggers [14] describe a compiler transformation that adjusts the memory layout of applications though the computation of memory access pattern. Chow et al. [8] 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 [3]. None of these tools can avoid false sharing to the extent that Sheriff does.

8.4 Processes-as-threads

As described earlier, Sheriff borrows and significantly extends the process-as-thread model first employed by Grace [4]. 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

Benchmark	PagesWritten	Commits	Allocs	CleanupSize	TranLength(ms)	PagesPerTran	PagesPerMs
histogram	0	24	2	0	12.5	0	0
kmeans	1312	3836	101002	0	4.15	0.34	0.08
linear_regression	16	24	3	0	38.6	0.67	0.02
matrix_multiply	16	24	11	0	313.23	0.67	0.0
pca	0	47	2	0	450.69	0	0.0
reverseindex	260201	156409	250927	0	0.05	1.66	30.99
string_match	0	24	7	0	104.75	0	0.00
word_count	145	89	38	32	25.08	1.63	0.06
blackscholes	0	23	4	0	453.51	0	0.0
canneal	8	1056	5974612	0	10.32	0.01	0.0
dedup	76184	45636	8291	0	0.04	1.67	44.9
ferret	904381	1072258	110558	0	0.01	0.84	76.04
fluidanimate	8	10018550	135430	352	0.00	0.00	0.00
freqmine	0	1	33	0	11524.6	0	0.0
streamcluster	32824	128557	12	294	0.02	0.26	10.42
swaptions	48	24	388	0	167.23	2	0.01

Table 4. Characteristics of benchmarks.

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.

9. Future Work

We plan to extend Sheriff to find more performance related problems in multithreaded programs. For example, if one frequently-read word happens to be in the same cache line with one frequently-written word, it would be better to separate those two words. But in the current framework, Sheriff can not detect the memory read operation using the twin page mechanism. We are examining the combination of hardware watchpoints to help us locate this kind of performance error. In addition, we plan to exploit watchpoints to capture those program counters that touch specific addresses so as to point the programmer to specific lines of code responsible for false sharing.

10. Conclusion

This paper presents Sheriff, a software-only system that eliminates and precisely detects false sharing in multithreaded programs. Sheriff accomplishes this by converting threads into processes, isolating writes, and merging updates using a twinning mechanism inspired by distributed shared memory systems that both allows it to avoid false sharing and to distinguish true from false sharing. Sheriff uses sampling to identify false sharing instances responsible for actual performance degradation, and avoids the false positives of previous tools by intercepting memory allocation operations. Sheriff reports allocation call site information for any heap object responsible for false sharing problems with a possible performance impact. We show that Sheriff is useful in tracking down and resolving false sharing problems in an existing benchmark suite, in one case allowing us to increase performance by 10X.

Sheriff can operate directly on unaltered binaries, making deployment simple. Since Sheriff does not require advanced hardware support, it can be used to find false sharing problems for those legacy applications running on commodity hardware. For most of applications, Sheriff incurs reasonable runtime overhead.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, 2010.

[2] D. Bacon. *SETL for Internet Data Processing*. PhD thesis, New York University, January 2000. Section 2.12, Passing File Descriptors.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[8] 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.

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

[13] Intel. Avoiding and identifying false sharing among threads. http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threa 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 program. Master's thesis, National Chung Cheng University, July 2009.

[19] 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.

[20] 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.

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