RELIABLE AND EFFICIENT MULTITHREADING

A Dissertation Presented

by

TONGPING LIU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2014

School of Computer Science

RELIABLE AND EFFICIENT MULTITHREADING

A Dissertation Presented

by

TONGPING LIU

Approved as to style and content by:

Emery D. Berger, Chair

Scott F. H. Kaplan, Member

Yuriy Brun, Member

Israel Koren, Member

Lori A. Clarke, Chair
School of Computer Science

ACKNOWLEDGMENTS

I would first thank Emery Berger, my Ph.D. thesis advisor, for his wonderful supervision and enthusiastic support in the development of this thesis work. Emery taught me to work on those important and practical projects, and take everything seriously. I hope that I could be as lively, enthusiastic, and energetic as Emery one day to my students. I also thank my dissertation committee: Scott Kaplan, Yuriy Brun, and Israel Koren, for their valuable insights, feedback, and support.

I am also fortunate to work with Chen Tian, Timothy Richards, Prashant Shenoy, Ziang Hu, Daniel Waddington, Seetharami Seelam, Wei Tan, Liana Fong, and Arun Iyengar. They provided me valuable guidance and suggestions on those projects we worked on together. I also thank Leeanne Leclerc, James Allan, and Laurie Downey for their help to make my experience at UMASS go as smoothly as possible.

I couldn't finish my thesis without the help of PLASMA labmates, including Charlie Curtsinger, Ting Yang, Gene Nowark, Kaituo Li, John Altidor, Divya Krishnan, Dan Barowy, Dimitar Gochev, John Vilk, Nitin Gupta, Jacob Evans, Justin Aquadro, and Emma Tosch. I also feel lucky to meet many friends in the computer science department, including Rui Wang, Ming Li, Tingxin Yan, Zongfang Lin, Kun Tu, Xiaojian Wu, Pengyu Zhang, Bo Jiang, Xiaozhen Tie, Fangyuan Zhou, Hong Zhang, Yue Wang, Wenzhao Liu, etc. I will never forget your help in my life.

Finally, I would give my special thanks for my wife, Yuyu Tang. She quit her job to support my crazy idea of getting a Ph.D. degree. She also took care of most of the household duties and spent much of her time taking care of our two adorable kids, Yanbin Liu (Grace) and Yanlin Liu (Eileen). My buddy, Guangming Zeng,

also deserves my special thanks for his generous help and valuable discussion when I chose the career in the computer science field. I also want to thank my kids, my late grandma, my parents, sisters, and brothers for their understanding and support.

ABSTRACT

RELIABLE AND EFFICIENT MULTITHREADING

MAY 2014

TONGPING LIU
B.S., HARBIN INSTITUTE OF TECHNOLOGY
M.E., HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
M.S., UNIVERSITY OF MASSACHUSETTS AMHERST
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

The advent of multicore architecture has increased the demand for multithreaded programs. It is notoriously far more challenging to write parallel programs correctly and efficiently than sequential ones because of the wide range of concurrency errors and performance problems.

In this thesis, I developed a series of runtime systems and tools to combat concurrency errors and performance problems of multithreaded programs.

The first system, DTHREADS, automatically ensures determinism for unmodified C/C++ applications using the `pthreads` library without requiring programmer intervention and hardware support. DTHREADS greatly simplifies the understanding and debugging of multithreaded programs. DTHREADS often matches or even exceeds the performance of standard thread libraries, making deterministic multithreading a practical alternative for the first time.

The second system attacks one notorious performance problem of multithreaded programs: false sharing. We provide the first accurate and precise detection tool, Sheriff-Detect, which can pinpoint the name of global variables or the allocation context of heap objects that involve in false sharing problems, without false positives. However, rewriting a program to fix false sharing can be infeasible when source code is unavailable, or undesirable when padding objects can increase excessive memory consumption or further worsen runtime performance. To resolve this problem, we provide a runtime system, Sheriff-Protect, to automatically boost the performance of programs with false sharing problems.

The third system, Predator, improves the effectiveness of false sharing detection. It can detect one more type of false sharing: *read-write* false sharing. Also, it can even detect false sharing problems without occurrences, thus overcomes a shortcoming of all existing tools: they can only detect those observed false sharing problems. Predator is the first tool to uncover false sharing problems of real applications.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

INTRODUCTION

For decades, applications enjoyed automatic and regular performance gains from increasing CPU speed. However, the increase of CPU speed results in consuming more energy and generating more heat. Thus, Intel and other vendors have turned to providing multiple cores on a single machine. To take advantage of multiple cores, software needs to be written using multithreading.

Building efficient and reliable multithreaded programs is still a challenging task because of the following reasons. First, concurrency requires programmers to think in an unnatural way that humans find difficult. Second, existing languages and tools are inadequate to detect or prevent concurrency errors and performance anomalies.

Concurrency errors of multithreaded programs, such as race conditions, atomicity violations, order violations, and deadlocks, are very hard to debug [42], because their occurrences highly depend on some specific conditions, such as thread interleavings and CPU scheduling [3, 17]. Instead of detecting possible concurrency errors, one promising alternative approach is to attack the problem of concurrency bugs by eliminating its source: non-determinism. A fully *deterministic multithreading system* would prevent Heisenbugs by ensuring that executions of the same program with the same inputs always yield the same results, even in the face of race conditions in the code. Such a system would not only dramatically simplify debugging of concurrent programs [19] and reduce their attendant testing overhead, but would also enable a number of other applications. For example, a deterministic multithreaded system would greatly simplify record-and-replay for multithreaded programs [20, 39] and the deterministic replication of a multithreaded application on different machines for fault tolerance [5, 8, 14, 50].

It is also difficult to write efficient multithreaded programs. The *false sharing* problem is a notorious performance problem for multithreaded programs [12, 27]. It occurs when multiple threads, running on different cores with their separate caches, access logically independent words in the same cache line. If a thread modifies a cache line, the cache coherence protocol invalidates the duplicates of this cache line in other caches, which is crucial for true sharing cases. However, it is totally unnecessary for false sharing cases. False sharing can force one core to wait unnecessarily for updates from another processor, thus wasting both the CPU time and precious memory bandwidth.

Contributions

This thesis handles two categories of problems for multithreaded programs, *reliability* and *performance*. It makes the following contributions:

- SHERIFF framework: I developed a novel processes-as-threads framework derived from Grace [7]. SHERIFF is a software-only drop-in replacement of the stand `pthreads` library. It turns threads into processes, with separate address spaces but a shared file table. SHERIFF provides per-thread memory protection and isolation on page granularity by relying on the stand memory protection mechanism and a twinning-and-diffing mechanism. SHERIFF enables a range of possible applications, including language support and enforcement of data sharing, software transactional memory, thread-level speculation, and race detection.

- I developed an efficient deterministic multithreading system, DTHREADS, for unmodified C/C++ applications, without programmer intervention and hardware support. DTHREADS is based on the SHERIFF framework to isolate executions of different threads. DTHREADS outperforms the previous state-of-the-art runtime system (CoreDet) by a factor of 3, and often matches and sometimes exceeds the performance with the standard `pthreads` library. Dthreads enforces robust/stable determinism even in the face of data races, greatly simplifying program understanding and debugging: programs always behave identically, even with different inputs and on different hardware, as long as the synchronization order is the same. Because of this, Dthreads can also be used to support replicated executions of multithreaded applications for fault tolerance purposes.

- Based on the Sheriff framework, I developed another two tools, Sheriff-Detect and Sheriff-Protect, to deal with false sharing problems of multithreaded programs, one of the notorious performance problems. Sheriff-Detect find instances of false sharing accurately (no false positives), runs with low overhead (on average 20%), and can pinpoint global variables and heap objects involving in false sharing. Sheriff-Protect mitigates false sharing by adaptively isolating shared accesses on a cache line from different threads into separate physical addresses, effectively eliminating the performance impact of false sharing. It can automatically boost the performance of multithreaded applications with false sharing problems.

- I also developed a tool, Predator, to improve the effectiveness of false sharing detection. Instead of relying on the Sheriff framework to track memory writes, Predator employs compiler instrumentation to track read and write memory accesses, which make it possible to detect one more type of false sharing, *read-write* false sharing. Predator also overcomes a key limitation of previous detection tools: existing tools can only detect observed false sharing problems. However, occurrences of false sharing highly depend on memory layout and size of a cache line, which are affected by a lot of dynamic properties. Predator can predict potential false sharing that does not manifest in a given execution but may appear—and greatly degrade application performance–in a slightly different execution environment. PREDATOR is the first false sharing tool able to automatically and precisely uncover false sharing problems in real applications, including MySQL and the Boost library.

Outline

The rest of this thesis is organized as follows. Chapter 1 describes reliability and performance problems of multithreaded programs, which we are going to handle in this thesis. Chapter 2 describes the processes-as-threads framework, SHERIFF, which is the basis of DTHREADS, SHERIFF-DETECT and SHERIFF-PROTECT. Chapter 3 describes DTHREADS that ensures deterministic execution for multithreaded programs linking to this drop-in library. Chapter 4 discusses how to precisely detect and automatically tolerate false sharing problems based on the SHERIFF framework. Chapter 5 describes a generalized false sharing detection tool by combining compiler instrumentation and runtime system, which improves the effectiveness of false sharing detection. Chapter 6 provides a substantial comparison between previous work and our approaches. Chapter 7 concludes the thesis with its contributions and possible future work.

CHAPTER 1

PROBLEMS OF MULTITHREADED PROGRAMS

Writing Multithreaded programs can encounter concurrency errors and performance anomalies. This thesis discusses in detail two different types of problems, non-determinism and false sharing. We discuss the definitions, causes of these problems and their possible consequence as follows.

1.1 Non-determinism

1.1.1 Background

Deterministic behavior of programs is the most desirable behavior: given the same input, a program produces the same output and generates the same execution. Relying on this behavior, it is able to figure out problems of programs.

In reality, it is relatively easy for sequential programs to achieve this target if a program do not explicitly rely on a randomized mechanism. However, it is hard to do this for parallel programs. In shared memory multithreaded programs, an application can only experience one of many possible schedules at a time. Thread scheduling, the order of memory accesses on the shared data, operations depending on timing and non-deterministic synchronizations, can easily lead to different executions of the same program.

A simple example of non-deterministic execution can be seen in Figure 1.1. When using the standard `pthreads` library, this program can print “1,0”, “0,1” or “1,1” in the end, depending on the order of memory accesses from different threads. We actually run this simple program for one million times. About 99.43% of time, it will

```
int a = b = 0;        void * t1 () {        void * t2 () {
main() {                if (b == 0) {         if (a == 0) {
 spawn(p1, t1);           a = 1;                b = 1;
 spawn(p2, t2);         }                     }
 join(&p1);             return NULL;          return NULL;
 join(&p2);           }                     }
 print(a, b);
}
```

Figure 1.1. Non-determinism problem

print “1,0”, while 0.56% it will print “0,1” and 0.01% it will print “1,1”. According to the semantics of this program, both “1,0” and “1,0” are correct results. Thus, the unexpected result (“1,1”) caused by race conditions happens very rarely, only about 0.01%. It is very difficult to observe/reproduce these rare cases that caused by race conditions.

1.1.2 Source of Non-determinism

Non-determinism can be caused by a lot of sources, both external sources and internal sources. For example, the timing of external inputs is one of the sources that can lead to non-determinism. This section only lists internal sources of non-determnism [1].

Thread Communication: Thread communication is the most important source of non-determinism for multithreaded programs. First, the order of accesses on shared variables may change from one execution to the other. Second, the orders on shared resources, such as memory allocation, synchronization, and library/system calls, vary across different executions. Third, the interaction between compiler and run-time can be changed. For example, lazy binding may cause the thread that performs address resolution to execute much more instructions than others.

Memory Layout: Address space layout randomization (ASLR) in Linux environment brings non-deterministic memory addresses of instructions and data across different executions. Thus, a program relying on memory addresses lead to non-deterministic execution of a program.

System or Library Dependence: Some library or system calls cannot return deterministic results. For example, the `gettimeofday()` library call returns different time values at different time, and `read` system calls may return different number of bytes, depending on the timing of issuing `read` calls. An application relying on them can execute non-deterministically too.

1.1.3 Effect of Non-determinism

Because of different sources of non-determinism, listed in the above section, existing multithreaded applications can not run deterministically: given the same input, a program can have different executions that may or may not lead to different outputs.

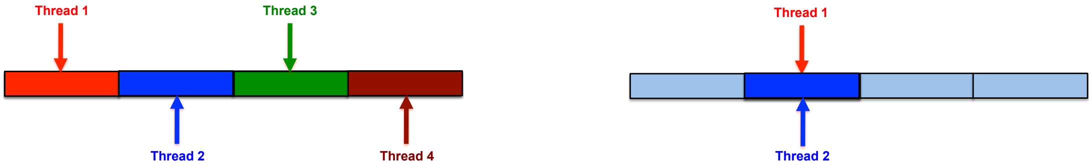
Non-determinism can greatly complicate the reasoning and debugging in development phases, which makes it hard for programmers to reproduce program errors. Even worse, since executions of deployment can vary from executions of development phase, a lot of programmer errors can be easily leaked to customers.

By contrast, determinism greatly simplifies the understanding and debugging of multithreaded programs. We can always guarantee the same executions on both development phases and the deployment phases, thus there is no need to worry about erroneous results.

1.2 False Sharing

1.2.1 Definition

False sharing occurs when different processors in a shared-memory parallel system are referencing distinct fields within the same coherence block (page or cache line) simultaneously, thereby inducing “unnecessary” coherence traffic [13].



(a) False sharing

(b) True sharing

Figure 1.2. False sharing and true sharing in a cache line with four words.

Although it is difficult or impossible to know where a thread runs in an actual execution, we can conservatively assume that different threads are running on different processors with separate cache. Thus, in the multithreaded environment, false sharing simply implies: multiple threads access distinct parts of the same cache line simultaneously, while one of them is a write operation. False sharing is shown in Figure 1.2(a). Based on the relationship of false sharing objects, false sharing can be classified into inter-object and intra-object false sharing. When two different objects in the same cache line are accessed by different threads simultaneously, that is inter-object false sharing. Otherwise, it is intra-object false sharing.

There is another concept, true sharing, which is opposite of false sharing. In true sharing (Figure 1.2(b)), multiple threads are accessing the same word.

There is another way to differentiate false sharing with true sharing. False sharing is avoidable, while true sharing is not.

1.2.2 Reason of False Sharing

As shown in Figure 1.2, false sharing only occurs when the size of coherence block is larger than that of a single word. Multiple processors may reference different words of the same coherence block. In this perspective, a single-word block size can avoid false sharing problems.

However, using a single-word block size is not the actual case. In reality, the size of a coherence block (cache line) is normally 32 or 64 bytes. The reason of using multiple words in a cache line is to reduce the groups of transfers between the main memory and the cache since programs always have some spatial locality of reference. Those adjacent words are very likely to be referenced in the future.

From the performance perspective, reducing the coherence block size to one word may minimize the data to transferred, but can increase the number of transfers. Thus, the overhead of transferring less data at a time can be larger than the benefit of eliminating false sharing coherence traffic. Actually, the hardware trend of cache line is to increase the size of cache line, which makes false sharing problems increasingly common.

1.2.3 Performance Impact

False sharing can greatly slowdown the execution of multithreaded programs, which depends on many factors, including the cache block size, data layout, program access patterns, and the cost of coherence operations [13].

In a typical shared-memory system, each processor may have a separate cache. In order to increase the access speed, when a processor references a word, all the data inside the same cache line is fetched from the main memory to its corresponding cache. When multiple processors are accessing distinct words of the same cache line simultaneously, the shared data can be replicated into caches of different processors that access this cache line. Thus, it is very important to maintain the coherence across different processors: if any copy is changed, this change should be propagated to other processors immediately for correctness purposes. In real hardware, this data propagation only happens lazily when the data is accessed again, thus duplicates are invalidated at first. When a processor access an invalidated cache line, it should wait for the data propagating from other processors, wasting CPU time and memory bandwidth simultaneously.

In the false sharing case, this propagation is totally unnecessary because different threads are actually accessing different parts of the same cache line. Thus, there is no need for a processor to get the updated data that is not going to access. However, hardware can only tracks the change of data on the granularity of a cache line and have to propagate those changes if any word has been changed. When there are interleaved writes, issued by different processors, on the same cache line, the ping-pong effect of loading-and-invalidating of data on this cache line can greatly slow the execution of programs. Programs with false sharing can even run slower in a multi-core machine than in a single-core machine, losing the benefit of multiple cores.

Many common programming practices can easily cause false sharing. For example, different threads accessing different entries of the same global array, listed in Figure 1.3, is such an example. This example has no correctness problem, but a serious performance problem.

```
int Array[8];                    void * child (int S) {
int W = 1;                         for(i = S; i< S + W; i++)
                                     for(j = 0; j < 1M; j++)
int main(int THREADS) {                Array[i]++;
 W = 8/THREADS;                  }
  for(i = 0; i< 8; i += W)
   spawn(increment, i);
}
```

Figure 1.3. False sharing problem

We actually run this program on a real machine with 8 cores and Figure 1.4 presents performance results. On this evaluation, we specifically choose a different number of threads, matching the number of hardware cores, from 1 thread to 8 threads, to perform the same amount of workload. We find out that false sharing can greatly impact the performance, which brings around 13× difference between actual performance and the expected performance. Two trends–the prevalence of multicore

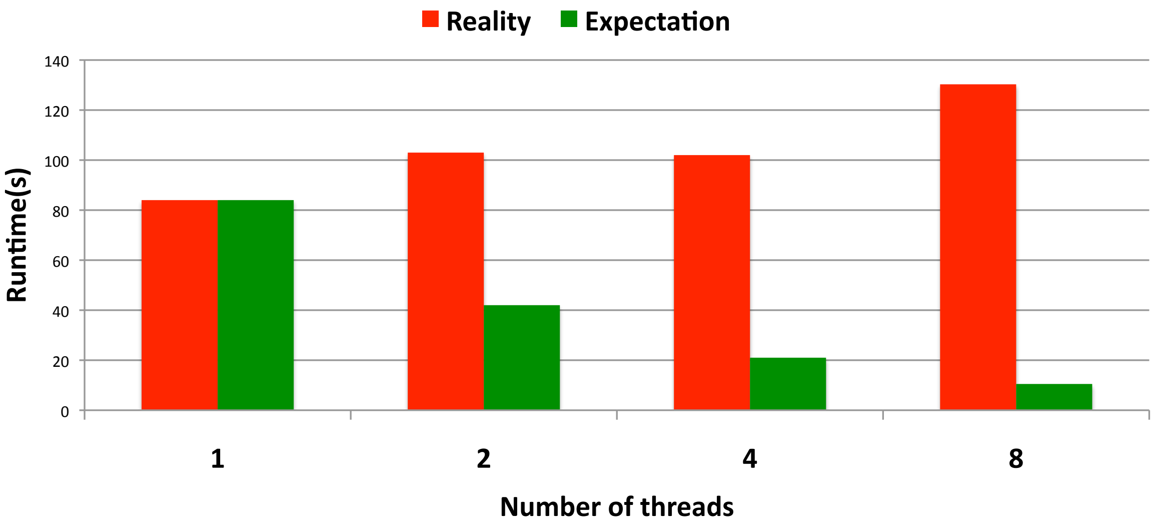


Figure 1.4. False sharing performance impact for the simple program shown in Figure 1.3.

architectures and the expected increase in the number of multithreaded applications in broad use, and increasing cache line sizes–are likely to make false sharing increasingly common.

1.2.4 Fixing False Sharing

There are several ways to fix false sharing problems after they are identified. The basic idea is to prevent multiple threads from accessing the same cache line simultaneously.

The first way is to change the size of corresponding structure or class, by padding some useless words. Thus, we can prevent two threads concurrently accessing the same cache line. One example of prevention, linear_regression, can be seen in Section 4.3.1.

The second way is to assign the value to thread-local variables at first. Then different threads only update their own local variables, and commit those changes back to the shared variable in the end. For example, the problem shown in Figure 1.3 is fixed using this method, see Figure 1.5.

```
void * child (int S) {
  for(i = S; i< S + W; i++) {
    int temp = Array[i];

    for(j = 0; j < 1M; j++)
      temp++;

    Array[i] = temp;
  }
}
```

Figure 1.5. Fixing the false sharing problem shown in Figure 1.3.

Some other approaches, to fix false sharing problems automatically, is described in detail in Section 6.3.2, but they all suffer different shortcomings.

CHAPTER 2

PROCESSES-AS-THREADS FRAMEWORK

Sheriff extends the processes-as-threads idea, first introduced in Grace [7], to be a drop-in replacement of the standard `pthreads` library. It interposes those thread-spawning calls and replaces them with `clone` system calls with `CLONE_FILES` flag, turning threads into processes. Since different processes have separate address spaces and signal handlers, different processes can isolate their executions and employ page-based "per-thread" memory protection. In order to achieve the shared-memory semantics of multithreaded programs, Sheriff replaces synchronizations with process-based synchronizations (Section 2.2), runs the regions between synchronizations in the isolation mode (Section 2.4), and commits process-private changes to the shared mapping (Section 2.3).

2.1 Thread Creation and Exit

For thread creations, Sheriff interposes `pthread_create()` functions and replaces them with `clone` system calls. By taking advantage of a feature of Linux that allows selective sharing of memory and file descriptors, Sheriff sets the `CLONE_FILES` flag when creating new processes, resulting in child processes with different address spaces but the same shared file descriptor table. However, this attribute may not be applicable to other systems, e.g., Solaris. That would require shims on I/O operations to allow processes to share open file descriptors by sending them over UNIX domain sockets [56, Section 17.4].

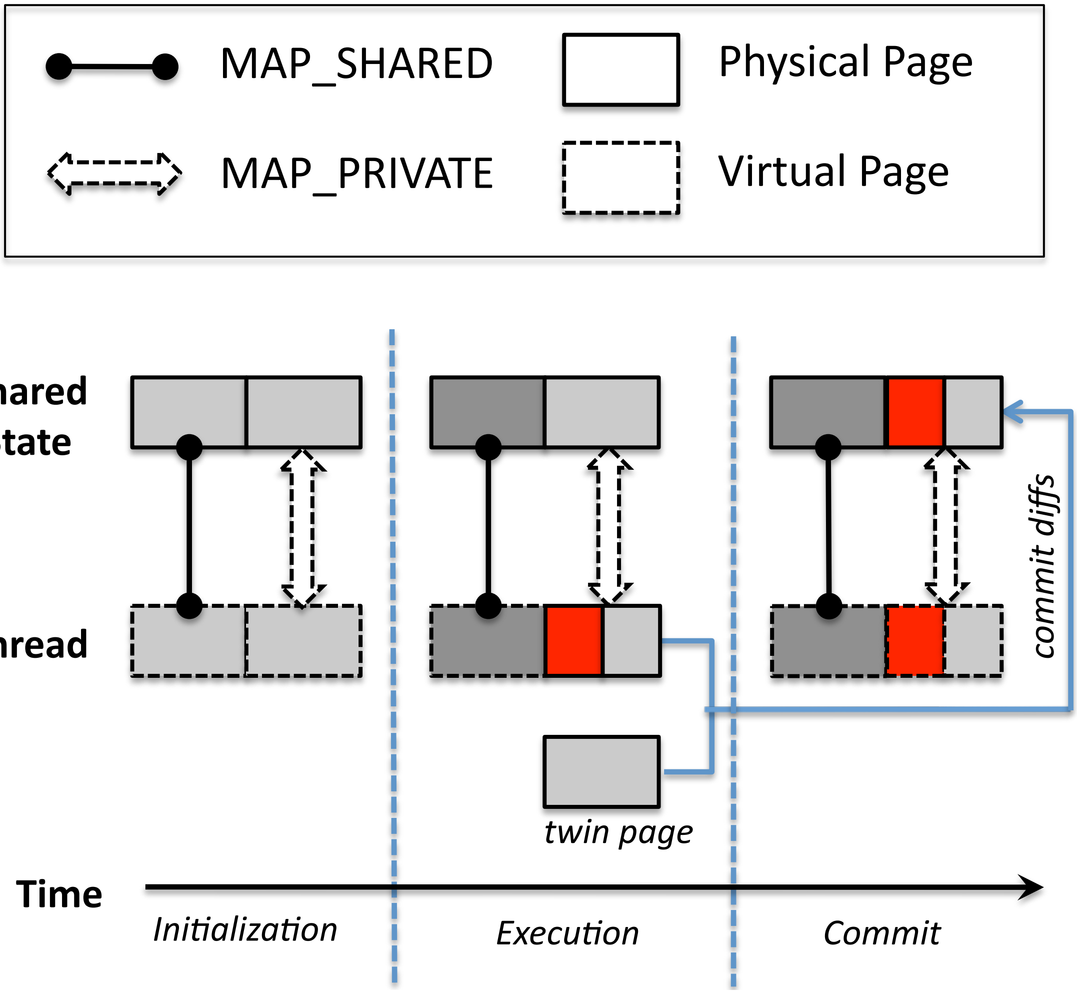


Figure 2.1. SHERIFF replaces threads with processes, thus it enables page-based "per-thread" memory protection and memory isolation. Upon synchronization points, local changes of different "threads" are committed to the shared state by comparing the difference between those working pages and their twin pages.

For those children threads, SHERIFF specifically invokes the `exit` function in order to exit those processes. For `pthread_join`, joiners call `waitpid` to wait for a corresponding process to complete.

2.2 Synchronizations

SHERIFF supports the full range of synchronizations, including mutexes, conditional variables, barriers, and signals.

By definition, synchronization is used to coordinate activities and data accesses among different threads. For example, a program calls `mutex_lock()` before accessing the shared data. Leveraging on the processes-as-threads mechanism, Sheriff actually runs the regions between synchronizations in an isolated mode, which actually divides a program execution into different "transactions." In the same transaction, all reads/writes happen only on private pages after the first write operation on those pages. Reads still perform on the shared mapping directly if a page is not written by the current thread.

At synchronization points, Sheriff commits those private changes of each thread to the shared mapping in order to achieve the shared memory semantics of multithreaded programs. Detailed implementation about the execution inside a transaction is discussed in Section 2.4.

It is noted that the transaction concept here is different from that of transactional memory [36]. Sheriff does not support rollback and favors more on a longer transaction to better amortize the overhead.

Sheriff turns threads into processes and runs an application in an isolated mode when there is no synchronization. But this isolation mechanism should not work for those synchronization variables. For example, in the mutex_lock(), if a process only updates its private page holding this lock variable, then this update is not seen by other processes, which can cause multiple processes to enter into the same critical section concurrently. In order to coordinate different threads, Sheriff invokes process-based synchronizations on those synchronization variables that are shared across different processes, shown in Figure 2.2. Whenever there is a synchronization, Sheriff ends the current transaction, gets its process-shared variable, and synchronizes on this variable by using a process-based synchronization. To quickly locate its process-shared variable for a synchronization variable, Sheriff simply stores the pointer of it into the first word of this synchronization variable.

```
void sync(var) {
  endTransaction();
  realVar = getRealVariable(var);
  sync_process_based(realVar);
  beginTransaction();
}
```

Figure 2.2. Pseudo-code for a synchronization.

2.3 Shared Memory Semantics

In order to create the shared memory illusion for the process-as-threads framework, Sheriff employs the memory-mapped files to share the heap and globals across different processes, but not the stack. Different threads are using their own stacks and the stack is not used as a cross-thread communication in general.

Sheriff creates two different mappings for both the heap and the globals. One is a shared mapping, which is used to hold the shared state. Another is a private, copy-on-write(COW) mapping (per-process) that each process works on directly. User applications can only access private mappings.

Private mappings are linked to shared mappings through the same memory mapped file. In the isolated mode, reads initially go to the shared mapping until the first write on a page. After the first write operation, both reads and writes happen on the private mappings only. In order to achieve the shared memory illusion, Sheriff commits the current thread's local changes to the shared mapping at synchronization points using the twinning-and-diffing mechanism described in Section 2.3.1. More details of this are discussed in Section 2.4.

In the initialization phase, Sheriff checks its `/proc/pid/maps` file to find the range of its globals and creates a shared mapping for the globals. For the heap, Sheriff uses a customized memory allocator, which is discussed in Section 2.3.2.

2.3.1 Twinning-and-Diffing mechanism

In order to find out those local changes made by each thread, SHERIFF] borrows the twin page mechanism, which is introduced in TreadMarks and Munin [35, 18] for tracking modifications on a page in the distributed share memory system.

The basic idea is to create an additional "twin" page before the actual modification, by handling those memory protection faults. It is essential to ensure that the "twin" page is identical to its "working" page. To achieve this target, SHERIFF issues a write operation to the original page, which specifically invokes a copy-on-write operation to create a "working" page. Then SHERIFF creates a "twin" page by copying this "working" page. At synchronization points, SHERIFF compares the "twin" page and its "working" page, using a byte-by-byte comparison, in order to find out those changes made by a thread: the difference of two pages simply implies the local changes made by the current thread.

2.3.2 Custom Memory Allocation

For the program heap, SHERIFF replaces the default heap allocator with a BiBOP-style memory allocator, built on HeapLayers [9]. SHERIFF pre-allocates a fixed chunk of memory from its underlying operating system using `mmap` system calls and satisfies memory allocations from this block by redirecting all memory allocations and deallocations. In the heap, all heap objects have the block size of *power of* 2, using an object header to mark its status and size information. There is no split and merge operation on heap objects. If the size of an allocation is less than *power of 2*, SHERIFF allocates an object with the size of the next *power of 2*.

In order to minimize possible false sharing induced by the memory allocator, SHERIFF borrows a "per-thread-heap" idea from Hoard [6]. SHERIFF divides the heap into a fixed number of sub-heaps (currently 16), with the shared metadata of the super heap. A thread can only allocate memory from its own sub-heap. When an object is freed, this object is returned to the subheap owned by the current thread. Since the subheap of each thread is allocated from different pages, this custom memory allocator is unlikely to allocate two objects from different threads on the same cache line, helping reduce the false sharing effect.

2.4 Execution of a Transaction

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

Transaction Begin: At the beginning of every transaction, SHERIFF write-protects all shared pages so that later writes to these pages can be caught by handling SEGV protection faults.

Inside a Transaction: Inside each transaction, SHERIFF runs at the same speed as a conventional multithreaded program for program reads. However, the first write to a protected page triggers a page fault that SHERIFF handles: in the page fault handler, SHERIFF obtains an exact copy of this page (a "twin" page), records the page holding the faulted address, and then unprotects this page so that future accesses run at full speed. Since SHERIFF only exposes the private mapping to user applications, write accesses on a private mapping actually create a "working" page for every page written inside a transaction.

Although protection faults are expensive, these costs are amortized over the entire transaction because each page only incurs at most one page fault per transaction.

Transaction End: At the end of each transaction, at thread exits and before synchronization points, SHERIFF commits local changes of a thread to the shared mapping to achieve the shared memory semantics. SHERIFF commits only the differences between those "twin" pages and their "working" pages, using a byte-by-byte comparison.

After those local changes are committed, Sheriff reclaims memory holding “twin” pages and “working” pages. Sheriff issues the `madvise` call, with the `MADV_DONTNEED` flag, to discard those “working” pages. Then, the current thread can observe those changes made by other threads from now on.

CHAPTER 3

DTHREADS:EFFICIENT DETERMINISTIC MULTITHREADING

As described in Section 1.1, non-determinism can greatly complicate the reasoning and debugging of parallel programs. To resolve this problem, several recent software-only proposals aim at providing deterministic multithreading. However, all of these existing approaches suffer from a variety of disadvantages. Language-based approaches are effective at removing non-determinism but require programmers to write code in specialized languages, which can be impractical [11, 16, 55]. Recent deterministic systems that target legacy programming languages (especially C/C++) are either incomplete or impractical. Kendo ensures determinism of synchronization operations with low overhead, but does not guarantee determinism in the presence of data races [48]. Grace prevents all concurrency errors but is limited to fork-join programs, and although it is efficient, it requires code modifications to avoid large runtime overhead [7]. CoreDet, a compiler and runtime system, enforces deterministic execution for arbitrary multithreaded C/C++ programs [4]. However, it exhibits prohibitively high overhead (running up to 8× slower than `pthreads`; see Section 3.4) and generates thread interleavings at arbitrary points in the code, complicating program reasoning, debugging, and testing.

Contributions: We develop **Dthreads**, an efficient deterministic runtime system for multithreaded C/C++ applications. Dthreads guarantees deterministic execution of multithreaded programs even in the presence of data races (notwithstanding external sources of non-determinism like I/O): given the same sequence of inputs, a program using DTHREADS always produces the same output. DTHREADS's deterministic commit protocol not only eliminates data races but also prevents lock-based deadlocks.

DTHREADS is easy to deploy: it works as a direct replacement for the `pthreads` library, requiring no code modifications or recompilation. DTHREADS is also very efficient. DTHREADS leverages process isolation and virtual memory protection to track and isolate concurrent memory updates, based on the SHERIFF framework. Not only does this approach greatly reduce overhead, comparing to approaches that track memory reads and writes, it also eliminates cache-line based false sharing, a notorious performance problem for multithreaded programs. These two features combine to enable DTHREADS to nearly match or even exceed the performance of `pthreads` for the majority of benchmarks examined here. DTHREADS thus marks a significant improvement over the state-of-the-art in deployability and performance, and provides promising evidence that fully deterministic multithreaded programming may be practical.

3.1 Dthreads Overview

Figure 1.1 shows an example multithreaded program that, because of data races, non-deterministically produces the outputs: “1,0,” “0,1,” and “1,1.” The order of instructions are changed from one execution to the other, resulting in these nondeterministic outputs. Using DTHREADS, this program will *deterministically* produce the same output “1,1.” Although this output can be a undesired one, the fact that results are always reproducible would make it easy for developers to reproduce and locate data races inside parallel programs.

DTHREADS employs the following mechanisms to ensure the deterministic execution, illustrated by Figure 3.1:

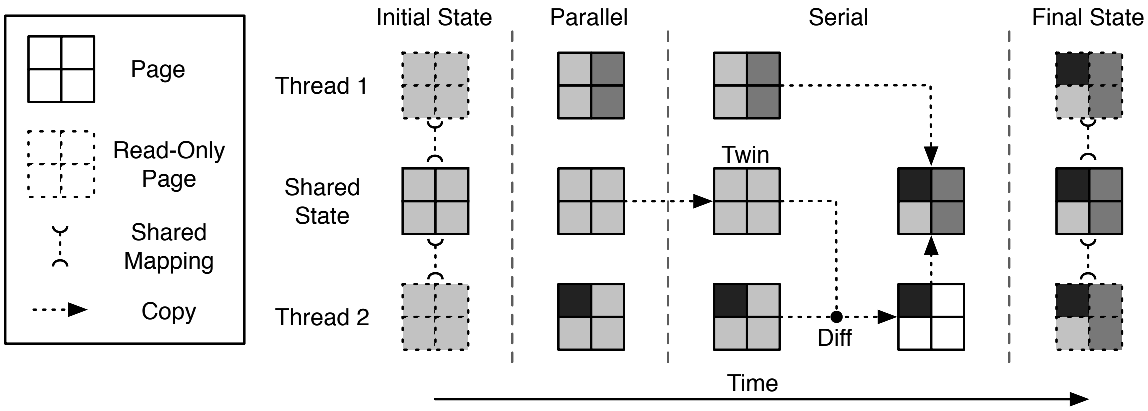


Figure 3.1. An overview of DTHREADS execution.

Isolated Memory Access: Based on the SHERIFF framework, DTHREADS runs threads as separate processes with private and shared views of memory, thus isolating executions of different "threads." DTHREADS uses this isolation mechanism to control the visibility of memory state, so that updates made by a thread cannot be seen by other threads if those updates are not committed explicitly to the shared mapping. By doing this, we guarantee that each "thread" can operate independently until synchronization points. Implementations are discussed in depth in Section 3.2.1.

Deterministic Memory Commit: Multithreaded programs use shared memory for communication, thus DTHREADS must make a thread's changes seen by other threads. To guarantee determinism, DTHREADS should publish updates of different threads in a deterministic order at deterministic points.

DTHREADS actually commits the changes of a thread to the shared state in sequence at synchronization points. These points includes thread creation and exit; mutex lock and unlock; condition variable wait and signal; posix sigwait and signal; and barrier waits. Commits are ordered using a global token that is passed from one thread to the next; a thread can only commit when it holds the token. The token-passing protocol is described in Section 3.2.2.1 and the implementation of synchronization primitives is described in Section 3.2.3.

Dthreads relies on the twinning-and-diffing mechanism to find out local changes of different threads, which has been discussed in Section 2.3.1.

Deterministic Synchronization: There is no deterministic guarantee on synchronizations under existing operating systems. Thus, Dthreads re-implements the full range of pthreads synchronization primitives and discusses them in details in Section 3.2.3.

3.1.1 Fixing the data race example

About the example program in Figure 1.1, Dthreads effectively isolates the execution from each thread until it completes, and then orders updates from different threads by thread creation time using a deterministic last-writer-wins protocol.

In the beginning of every execution, thread 1 and thread 2 have the same view of shared state, with a = 0 and b = 0. Since changes by one thread to the value of a or b are not visible to the other until this thread exits, both checks on two threads at line 2 will be true. So thread 1 sets the value of a to 1, and thread 2 sets the value of b to 1. These threads then commit their updates to the shared state and exit, with thread 1 always committing before thread 2. The main thread then should always print "1, 1" on every execution.

Determinism not only enables replay-without-recording and replicated executions, but also effectively converts "Heisenbugs" into "Bohr" bugs, making them reproducible. In addition, Dthreads optionally reports any conflicting updates due to racy writes, further simplifying debugging.

3.2 Dthreads Architecture

This section describes Dthreads key algorithms—isolated execution, deterministic (diff-based) memory commit, deterministic synchronization, and deterministic memory allocation—as well as other implementation details.

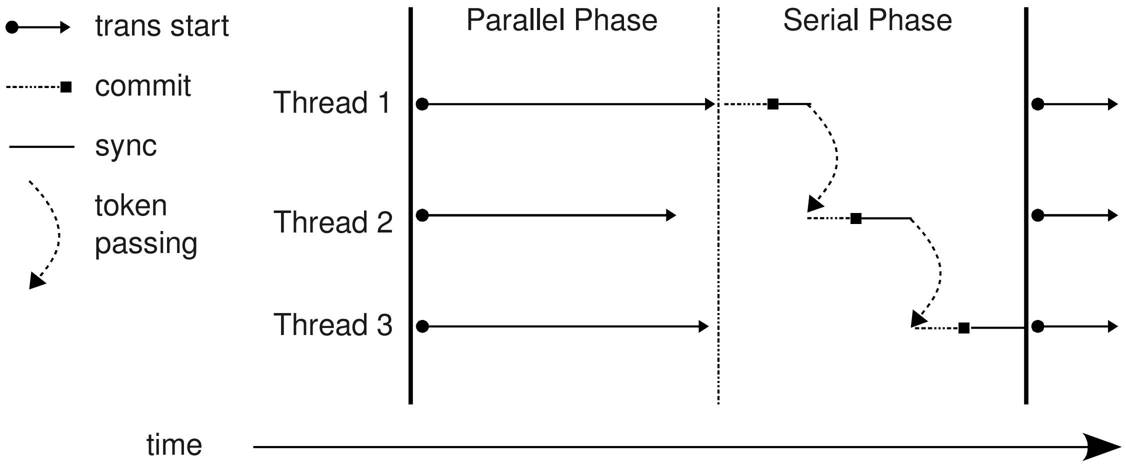


Figure 3.2. An overview of DTHREADS phase. Program execution with DTHREADS alternates between parallel and serial phases.

Figure 3.2 illustrates the execution of programs under DTHREADS. DTHREADS divides the execution of each thread into alternating parallel phases and serial phases. Based on the SHERIFF framework, DTHREADS isolates memory accesses in parallel phases. These accesses work on private copies of memory; that is, updates are not shared between threads during the parallel phases. When a synchronization point is reached, updates are applied (and made visible) in a deterministic order, as well as synchronizations.

3.2.1 Isolated Execution

Relying on the SHERIFF framework, DTHREADS turns threads into processes, with separate address spaces but the shared file table (Section 2.1). Thus, DTHREADS isolates memory accesses among different threads between synchronization points: different threads can only see their own local changes. Those changes are merged together at synchronization points in order to achieve the shared memory semantics.

```
void waitFence(void) {
  lock();

  while(!isArrivalPhase()) {
    CondWait();
  }

  waiting_threads++;
  if(waiting_threads < alive_threads) {
    while(!isDeparturePhase()) {
      CondWait();
    }
  }
  else {
    setDeparturePhase();
    CondBroadcast();
  }

  waiting_threads--;
  if (waiting_threads == 0) {
    setArrivalPhase();
    CondBroadcast();
  }

  unlock();
}
```

Figure 3.3. Pseudocode for the internal fence.

3.2.2 Deterministic Memory Commit

This section describes the mechanisms used to guarantee deterministic commits to the shared memory. These mechanisms are not provided by the Sheriff framework.

3.2.2.1 Fence and Token

Dthreads places internal fences between parallel and serial phases. Dthreads re-implements the fence because the standard `pthreads`'s barrier mechanism does not support dynamic changes of threads number.

Figure 3.3 shows the pseudocode code for the internal fence. Threads must wait at an internal conditional variable until all threads depart from the last departure

```
void waitToken() {
  waitFence();
  while(isNotMyToken()) { yield(); }
}
void putToken() {
  passTokenToNextOfTokenQueue();
}
```

Figure 3.4. Pseudocode for waitToken and putToken.

phase (lines 4-5). Then those threads are waiting at the fence until all alive threads have arrived at the same fence (lines 8-11). The last thread initiates the departure phase and wakes up all threads on the conditional variable (lines 14-15). As threads leave the fence, they decrement the number of waiting threads. The last thread to leave sets the fence to the arrival phase and wakes any waiting threads (lines 19-21).

To reduce overhead, whenever the number of running threads is less than or equal to the number of cores, waiting threads use spin locks, instead of expensive cross-process `pthreads` mutexes. When the number of threads exceeds the number of cores, Dthreads falls back to using `pthreads` mutexes.

Another key mechanism of Dthreads is the global token, which Dthreads uses to order memory commits and synchronizations. The token implementation is listed in Figure 3.4. The token is a shared pointer that points to the next runnable thread entry, which guarantees the global order for all operations in serial phases.

Dthreads introduces two subroutines to manage tokens. The `waitToken()` function first waits at the internal fence and then waits to acquire the global token in order to enter or leave the serial phases. The `putToken()` function passes the token to the next thread in the token-passing queue.

As shown in Figure 3.2, it is very important for a thread to wait at the internal fence before a thread enters or leaves serial phases, even for a thread that is guaranteed to have the token next. Otherwise, memory commits of a thread can affect other threads' behavior, bringing non-deterministic behavior for programs.

3.2.2.2 Commit Protocol

Figure 3.1 shows the steps to track modifications of every thread and expose them in a deterministic order.

At the beginning of parallel phases, different threads have a read-only mapping for all shared pages. In parallel phases, if a thread writes to a page, this write is trapped in order to create a private copy and a identical twin page for this page. After that, reads and writes on this page happen on the private copy only. For those non-trapped pages, reads still go directly to the shared state.

In serial phases, threads first commit their local changes that made in last parallel phase, guided by the global token. The first thread committing to a page can directly copy its private working copy to the shared state (page-based commits), but subsequent commits can only commit the modified bytes (byte-based commits), using the twinning-and-diffing mechanism discussed in Section 2.3.1. The byte-based commits are much slower than the page-based commits, but they won't overwritten changes committed by those predecessors. After a thread commits its local changes, it issues synchronizations before it passes the token to its next thread in the token-passing queue.

In the end of serial phases, every thread has to release those private pages and twin pages, recover the read-only mapping, wait at the internal fence before entering into the next parallel phase. By removing those private pages and recovering those mappings, a thread is able to observe changes made by other threads, achieving the shared memory semantics.

3.2.3 Deterministic Synchronization

Dthreads supports the full range of synchronizations of `pthreads` library, including locks, conditional variables, barriers and different types of thread exits. Since the Sheriff framework can not provide any deterministic guarantee, Dthreads implements different types of synchronizations in a deterministic way as follows.

3.2.3.1 Locks

Before a thread acquires a lock, it has to wait for the global token, by calling `waitToken`.

Dthreads treats multiple locks as the same one. It only ends the current serial phase for a thread when all locks held by this thread are released. Because of that, it is possible for a program to avoid deadlock problems.

At acquisitions of locks, Dthreads checks at first whether the current thread is already holding any locks. If not, the thread first waits for the token, commits those changes happened in the last parallel phase to the shared state, and begins a new atomic section. Then it increments the number of locks that it is currently holding before entering into critical sections.

At deacquisitions of locks, Dthreads decrements the number of locks that the current thread holds first. A thread does nothing if it still holds some locks, with the number of locks not equal to 0. If all locks are released, Dthreads commits the memory changes made in this serial phase to the shared state. Then it passes the global token to the next thread in the token-passing queue, and waits on the internal fence before entering into the next round's parallel phase.

3.2.3.2 Condition Variables

Guaranteeing determinism for condition variables is much more complex than for other synchronization primitives. The underlying operating system can not guarantee that threads are going to be waken-up in the same order as their waits. Thus, a naive implementation easily leads to a no-progress problem if the first waken-up thread can not get the global token to proceed first.

When a thread calls `pthread_cond_wait`, it first acquires the global token and commits local modifications made in the current serial phase since `pthread_cond_wait` is generally issued inside a critical section. It then removes itself from the token-passing queue, so that those threads waiting on condition variables do not participate in the token pass. Then, it adds itself to the conditional variable's waiting queue, decreases the number of alive threads (used in the internal fence mechanism), and passes the token to the next thread in the token-passing queue before actually waiting on a process-shared conditional variable.

When a thread is awaken, it should check at first whether the current thread is ready to run or not. For a deterministic reason, `pthread_cond_signal` should only wake up the first thread waiting on a conditional variable and `pthread_cond_broadcast` wakes up all waiting threads. However, the underlying operating system, like Linux, can not guarantee this. To resolve this problem, in `pthread_cond_signal`, we specifically wake up all threads, but only the first thread is given the permission to run: If a thread is not able to run, it waits on this conditional variable again; If a thread is the candidate thread to be waken up, it waits for the global token to enter into the next serial phase; The candidate thread should get the token immediately in order to avoid a no-progress problem.

For both `pthread_cond_signal` and `pthread_cond_broadcast`, the calling thread first waits for the global token, and then commits any local modifications before issuing an actual wake-up signal. When no threads are waiting on a condition variable, it passes the token to the next thread immediately, treating those calls as no-ops basically. Otherwise, it migrates corresponding threads, one for `pthread_cond_signal` and all for `pthread_cond_broadcast`, from the queue of this condition variable to the head of the token-passing queue, marks them as ready, increments the number of alive threads, and passes the token to the first thread in the token queue.

```
void thread_create () {
  waitToken();
  clone(CLONE_FS| CLONE_FILES | CLONE_CHILD);
  if(isChild) {
    allocGlobalThreadIndex();
    insertToTokenQueue();
    notifyChildRegistered();
        // Wait for the parent to reach next sync point
    waitParentBroadcast();
  }
  else if (isParent) {
    waitChildRegistered();
  }
}
```

```
void thread_exit() {
  waitToken();
  atomicEnd(false);
  removeFromTokenQueue();
  decreaseInternalFence();
  putToken();
  exitThread();
}
```

Figure 3.5. Pseudocode for thread creation and exit(§ 3.2.3.4).

3.2.3.3 Barriers

Threads waiting on a barrier should not disrupt the token passing of running threads: Dthreads removes those waiting threads from the token-passing queue, and places them in corresponding barrier queue.

In order to ensure determinism, the calling thread first waits for the global token to commit any local modifications. If the current thread is the last one to enter the barrier, it moves all threads on the barrier queue to the token-passing queue, increases the number of alive threads, and passes the token to the first thread in the barrier queue. Otherwise, it removes itself from the token-passing queue, places itself in the barrier queue, releases the token, and waits on this actual barrier.

3.2.3.4 Thread Creation and Exit

To guarantee determinism, thread creations and exits must be performed in serial phases.

In order to improve the parallelism and performance, a thread is allowed to create multiple threads without waiting for a new serial phase. Figure 3.5 shows the pseudocode for thread creation and thread exit. First, the calling thread waits for the global token before proceeding (line 2). It then creates a new process, with shared file descriptors but a distinct address space, by invoking the `clone` system call (line 3). Then the parent thread is waiting until its newly spawned child has registered itself.

The newly spawned child obtains the global thread index (line 5), places itself in the token-passing queue (line 6), and notifies the parent that registration has finished(line 7). Then it waits for the notification from the parent to proceed when the parent to reach the next synchronization point, not a thread creation. In this way, we can allow a parent thread to create multiple children threads in the same serial phase.

When `thread_exit()` is called, the caller first waits for the global token before committing any local modifications (line 3). It then removes itself from the token-passing queue (line 4), and decreases the number of alive threads (line 5). Finally, it passes the global token to the next thread in the token queue (line 6) and exits (line 7).

3.2.3.5 Thread Cancellation

Dthreads performs thread cancellations in serial phases for the deterministic reason. A thread can only invoke `pthread_cancel` while holding the global token. If the thread being cancelled is waiting on a condition variable or a barrier, it is removed from the queue deterministically. Finally, to cancel a thread, Dthreads kills the target process using kill(tid, SIGKILL) and decrements the number of alive threads after the cancellation.

3.2.4 Deterministic Memory Allocation

Sometimes, programs may rely on the addresses of objects returned by the memory allocator intentionally (for example, by hashing objects based on their addresses), or accidentally. A program with a memory error, like a buffer overflow, will yield different results for different memory layouts.

The reliance on memory addresses can undermine other efforts to provide determinism. For example, CoreDet is unable to fully enforce determinism because it relies on the Hoard scalable memory allocator [4]. Hoard was not designed to provide determinism and several of its mechanisms, thread id based hashing and non-deterministic assignment of memory to threads, lead to nondeterministic execution in CoreDet for the Canneal benchmark. To resolve this problem, DTHREADS employs both deterministic thread index and custom memory allocation mechanism.

3.2.4.1 Deterministic Thread Index

POSIX does not guarantee deterministic process or thread identifiers. To avoid exposing this nondeterminism to threads that run as processes, DTHREADS shims `pthread_self()` in order to return a deterministic thread index on different executions. This thread index is managed using a single global variable that is incremented on every thread creation. This unique thread index is also used to manage per-thread heaps and as an index into an array of thread entries.

3.2.4.2 Custom Memory Allocation

To preserve determinism in the face of intentional or inadvertent reliance on memory addresses, we designed the DTHREADS memory allocator to be fully deterministic.

Dthreads assigns subheaps to each thread based on its deterministically assigned thread index. In addition to guarantee the same mapping of threads to subheaps on different executions, Dthreads allocates superblocks (large chunks of memory) deterministically by acquiring a lock (under the global token) on each superblock allocation. Thus, threads always use the same subheaps, and these subheaps always acquires the same superblocks on each execution. The superblocks themselves are allocated via mmap: while Dthreads could use a fixed address mapping for the heap, we currently simply disable ASLR to provide deterministic mapping from mmap calls. If a program does not rely on absolute addresses, Dthreads can guarantee determinism even with ASLR enabled. However, hash functions and lock-free algorithms frequently use absolute addresses, and any deterministic multithreading system must disable ASLR to provide deterministic results for these cases.

3.3 Optimizations

Dthreads performs a number of optimizations to improve its performance.

Lazy commit: Dthreads reduces its copying overhead and the time spent in serial phases by lazily committing pages. When only one thread has ever modified a page, Dthreads considers this thread to be the owner of this page. An owned page is committed to the shared state only when another thread attempts to read or write this page. Dthreads tracks accesses from other threads using page protection, and signals the owning thread to commit pages on demand. To reduce the number of read faults, pages holding global variables (which we expect to be shared) and any pages in the heap that have ever had multiple writers are all considered unowned and are not read-protected.

Single-threaded-execution: When only one thread is running, Dthreads does not enable memory protection and treats all synchronization operations as no-ops. In addition, when only one thread is active and other threads are waiting on conditional variables, DTHREADS does not commit local changes to the shared mapping (and discard private dirty pages). Updates are only committed when the current thread issues a `cond_signal` or `cond_broadcast` call, which can wake up other threads and thus require publication of all updates made by this thread.

Lazy twin creation and diff elimination: To further reduce DTHREADS's copying and memory overhead, twin pages are only created for those pages that have multiple writers during the same transaction. In commit phases, a single writer of a page can directly copy its private working page to shared state, without performing a byte-by-byte comparison. Thus, when one thread is the sole writer of a page, this optimization saves a page allocation and a page copy during the execution (either parallel phases or serial phases), and a comparison in commit phases. In addition, DTHREADS eliminates unnecessary comparisons for all first committers, by associating a global version number (incremented at each commit) for every dirty page: In the page fault handler, every thread gets a local version number for current dirty page additionally; A thread can directly copies its working copy for each page whenever its local version number equals its global version number, since this thread is the first committer on this page and there is only one thread that can commit at a time in serial phases.

Lock ownership: DTHREADS uses lock ownership to avoid unnecessary waiting when threads are using distinct locks. Initially, all locks are unowned. Any thread attempting to acquire a lock that it does not own must wait until a serial phase to do so. If multiple threads attempt to acquire the same lock, this lock is marked as "shared". If only one thread attempts to acquire a lock, this thread takes the ownership of this lock and can acquire and release it during parallel phases. Lock ownership can result in starvation if a thread continues to re-acquire an owned lock without entering serial phases, while other threads are aiming to acquire the same lock (and waiting on the fence). To avoid this problem, each lock has a maximum number of times that it can be acquired during a parallel phase before a serial phase is required.

Parallelization: DTHREADS attempts to exploit as much parallelism as possible in the runtime system. One optimization is that at the start of transactions, DTHREADS performs certain cleanup tasks, including releasing private page frames or resetting pages to a read-only mode. It is safe to perform these cleanup tasks concurrently since these operations do not affect other threads's behavior. Thus, DTHREADS parallelizes a thread's cleanup tasks with other threads commit operations, without holding the global token. With this optimization, the token is passed to the next thread as soon as possible, reducing time in serial phases.

3.4 Evaluation

We perform our evaluation on an Intel Core 2 dual-processor CPU system, equipping with 16GB of RAM. Each processor is a 4-core 64-bit Xeon, running at 2.33GHZ with a 4MB L2 cache. The operating system is an unmodified CentOS 5.5, running with Linux kernel version 2.6.18-194.17.1.el5.

3.4.1 Methodology

We evaluate the performance and scalability of DTHREADS (versus CoreDet and `pthreads`) across the PARSEC [10] and Phoenix [52] benchmark suites.

In order to compare performance directly against CoreDet, which relies on the LLVM infrastructure, all benchmarks are compiled with the LLVM compiler at the "-O3" optimization level [37]. Since DTHREADS does not currently support 64-bit binaries, all benchmarks are compiled for 32 bit environments (using the "-m32" compiler flag). Each benchmark is executed ten times on a quiescent machine. To reduce the effect of outliers, results with the worst and best performance for each benchmark are discarded, so each result is the average of the remaining eight runs.

Tuning CoreDet: The performance of CoreDet [4] is extremely sensitive to three parameters: the granularity for the ownership table (in bytes), the quantum size (in number of instructions retired), and the choice between full serial mode and reduced serial mode. We compare the performance and scalability of DTHREADS with the best possible results that we could obtain for CoreDet on our system—that is, with the lowest average normalized runtime—after an extensive search of the parameter space (six possible granularities and 8 possible quanta, for each benchmark). The results presented here are for a 64-byte granularity, a quantum size of 100,000 instructions, and in full serial mode.

Unsupported Benchmarks: We do not include results for 7 benchmarks from PARSEC, since they do not currently work with DTHREADS (note that many of these also do not work for CoreDet). `vips` and `raytrace` would not build as 32-bit executables; `bodytrack`, `facesim`, and `x264` depend on sharing of stack variables; `fluidanimate` uses ad-hoc synchronization, so it cannot run without modifications; and `freqmine` does not use `pthreads`.

Scalability Experiment: For all scalability experiments, we logically disable CPUs using Linux's CPU hotplug mechanism, which allows us to disable or enable a specific CPU by writing "0" or "1" to a file: `/sys/devices/system/cpu/cpuN/online`.

3.4.2 Determinism

We first experimentally verify DTHREADS' ability to ensure determinism by executing the *racey* determinism tester [48]. This stress test contains, as its name suggests, numerous data races and is thus extremely sensitive to memory-level nondeterminism. DTHREADS reports the same results for 2,000 runs. We also verify that the schedules and outputs of all benchmarks of every execution are identical.

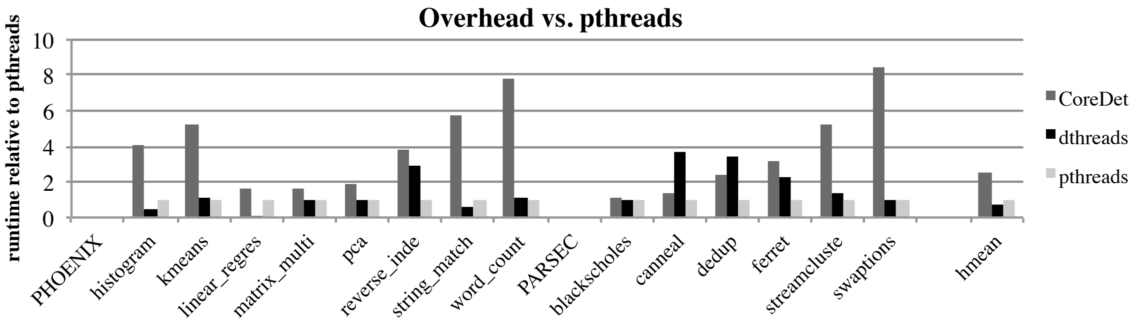


Figure 3.6. Normalized execution time with respect to `pthreads` and CoreDet(lower is better). For 9 of the 14 benchmarks, Dthreads runs nearly as fast or faster than `pthreads`, while providing deterministic behavior.

Benchmark	$\frac{\textbf{CoreDet}}{\texttt{pthreads}}$	$\frac{\textbf{Dthreads}}{\texttt{pthreads}}$	**Input**
histogram	4.35×	0.52×	*large.bmp*
kmeans	5.05×	0.91×	*-d 3 -c 1000 -p 100000 -s 1000*
linear_regression	1.50×	0.13×	*key_file_500MB.txt*
matrix_multiply	1.55×	1.00×	*2000 2000*
pca	1.94×	1.03×	*-r 4000 -c 4000 -s 100*
reverse_index	4.64×	2.73×	*datafiles*
string_match	5.95×	0.65×	*key_file_500MB.txt*
word_count	7.67×	1.09×	*word_100MB.txt*
blackscholes	1.13×	0.98×	*8 in_1M.txt prices.txt*
canneal	1.00×	4.12×	*7 15000 2000 400000.nets 128*
dedup	2.69×	3.39×	*-c -p -f -t 2 -i media.dat output.txt*
ferret	3.69×	2.84×	*corel lsh queries 10 20 1 output.txt*
streamcluster	4.87×	1.44×	*10 20 128 16384 16384 1000 none output.txt 8*
swaptions	7.61×	0.95×	*-ns 128 -sm 50000 -nt 8*

Table 3.1. Benchmarks: normalized execution time and input parameters.

3.4.3 Performance

For performance, We compare Dthreads to CoreDet and `pthreads`. Figure 3.6 presents these results graphically (normalized to the runtime of `pthreads`); Table 3.1 provides detailed information about the normalized execution time and input parameters.

Dthreads outperforms CoreDet on 12 out of 14 benchmarks (running between 20% and 12× faster). For 9 benchmarks, Dthreads runs nearly the same as or better performance than `pthreads`. Because Dthreads isolates updates in separate processes, it can improve performance by eliminating false sharing: since concurrent "threads" actually perform at different physical pages, there is no coherence traffic caused by false sharing between synchronization points. Dthreads eliminates catastrophic false sharing in the linear_regression benchmark, allowing it to execute over 8× faster than pthreads and 12× faster than CoreDet. The string_match benchmark exhibits a similar, though less dramatic, false sharing problem, allowing Dthreads to run almost 56% faster than pthreads and 9× faster than CoreDet. Two benchmarks, histogram and swaptions, also run faster with Dthreads than with pthreads (2× and 6%, respectively; 2.7× and 9× faster than with CoreDet). We believe but have not yet verified that the reason is false sharing.

Dthreads runs substantially slower than pthreads for 4 of the 14 benchmarks examined here. The ferret benchmark relies on an external library to analyze image files during the first stage in its pipelined execution model; this library makes intensive (and in the case of Dthreads, unnecessary) use of locks. Lock acquisitions and deacquisitions in Dthreads imposes higher overhead than ordinary pthreads mutex operations. More importantly in this case, the intensive use of locks in one stage forces Dthreads to effectively serialize the other stages in the pipeline, which must repeatedly wait on these locks to enforce a deterministic lock acquisition order. The other three benchmarks (canneal, dedup, and reverse_index) modify a large number of pages. With Dthreads, each page modification triggers a segmentation violation, a system call to change memory protection, the creation of a private copy of the page, and a subsequent copy into the shared space during commit phases. We note that CoreDet also substantially degrades performance for dedup and reverse_index), so much of this slowdown may be inherent to any deterministic runtime system.

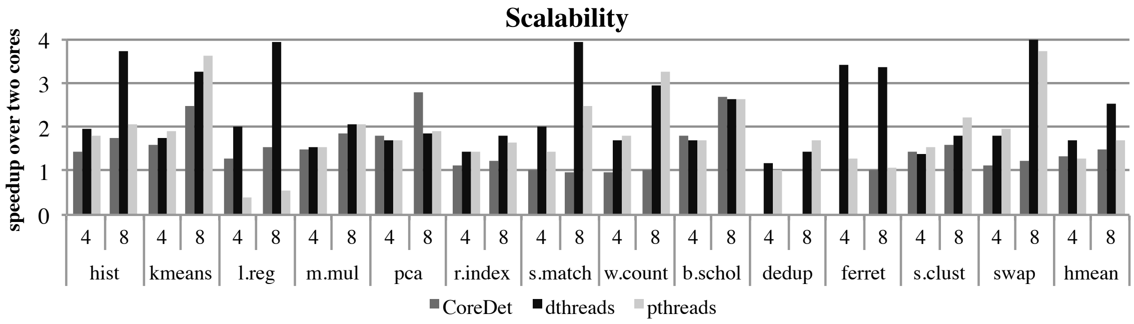


Figure 3.7. Speedup of eight cores versus two cores (higher is better). When possible to control with command line options, the number of threads was matched to the number of cores enabled.

3.4.4 Scalability

To measure the scalability cost of running Dthreads, we ran two benchmark suite (excluding `canneal`) on the same machine with eight cores, four cores, and just two cores enabled. Whenever possible without source code modifications, the number of threads was matched to the number of CPUs enabled. We have found that Dthreads scales at least as well as `pthreads` for 9 of 13 benchmarks, and scales as well or better than CoreDet for all but one benchmark. On average, Dthreads outperforms CoreDet by 3.5×. Detailed results of this experiment are presented in Figure 3.7 and discussed as follows.

`canneal` was excluded from the scalability experiment because this benchmark does more work when more threads are present, making the performance comparison between eight and two threads unfair. Dthreads hurts scalability (relative to `pthreads`) for four of the benchmarks: `kmeans`, `word_count`, `dedup`, and `streamcluster`, although only marginally in most cases. In all of these cases, Dthreads scales better than CoreDet.

Dthreads is able to match the scalability of `pthreads` for three benchmarks: `matrix_multiply`, `pca`, and `blackscholes`. With Dthreads, scalability actually improves over pthreads for 6 out of 13 benchmarks: histogram, linear_regression, reverse_index, string_match, ferret, and swaptions.

3.4.5 Performance Analysis

3.4.5.1 Benchmark Characteristics

The data presented in Table 3.2 are obtained from the executions running on all 8 cores. Column 2 shows the percentage of time spent in serial phases. In Dthreads, all memory commits and actual synchronization operations are performed in serial phases. The percentage of time spent in serial phases thus can affect performance and scalability. Applications with higher overhead in Dthreads often spend a higher percentage of time in serial phases, primarily because they modify a large number of pages that need to be committed during serial phases.

Column 3 shows the number of transactions in each application and Column 4 provides the average length of each transaction (ms). Every synchronization, including locks, conditional variable, barriers, and thread exits, demarcate transaction boundaries in Dthreads. For example, reverse_index, dedup, ferret and streamcluster perform numerous transactions whose execution time is less than 1ms, imposing a performance penalty for these applications. Benchmarks with longer (or fewer) transactions run almost the same speed as or faster than pthreads, including histogram or pca. In Dthreads, longer transactions amortize the overhead of memory protection and copying over a longer period, thus reducing performance overhead.

Column 5 and 6 provides more detail on the costs associated with memory updates (the number and total volume of dirtied pages). From the table, it is clear why canneal (the most notable outlier) runs much slower with Dthreads than with pthreads. This benchmark updates over three million pages, leading to large per-

Benchmark	Serial Phase (% of time)	Transactions (#)	TransLength (ms)	DirtyPages (#)	DirtyPages (GB)
histogram	0	23	15.47	29	0
kmeans	0	3929	3.82	9466	0.04
linear_regression	0	24	23.92	17	0
matrix_multiply	0	24	841.2	3945	0.02
pca	0	48	443	11471	0.04
reverseindex	17%	61009	1.04	451876	1.72
string_match	0	24	82	41	0
word_count	1%	90	26.5	5261	0.02
blackscholes	0	24	386.9	991	0
canneal	26.4%	1062	43	3606413	13.75
dedup	31%	45689	0.1	356589	1.36
ferret	12.3%	484127	0.05	844184	3.21
streamcluster	18.4%	130001	0.04	131992	0.50
swaptions	0	24	163	867	0

Table 3.2. Benchmark characteristics.

formance overhead caused by creating private copies, handling protection faults, and committing modifications on those pages to the shared memory mapping.

Conclusion: Most benchmarks examined here contain either a small number of transactions, thus having long running transactions, and modify a modest number of pages during execution. For these applications, Dthreads is able to amortize its overhead: by eliminating false sharing, it can even run faster than `pthreads`. However, for the few benchmarks that perform numerous short-lived transactions, or modify a large amount of pages, Dthreads can introduce substantial overhead.

3.4.5.2 Performance Impact Analysis

We further evaluate the performance impact of two important components of Dthreads: deterministic synchronization (sync-only) and memory protection(prot-only).

Sync-only: This configuration enforces a deterministic synchronization order. However, memory protection is not enabled so all “threads” (actually processes) access the shared memory directly. We want to use this to show the performance impact of load imbalance, caused by synchronization based scheduling.

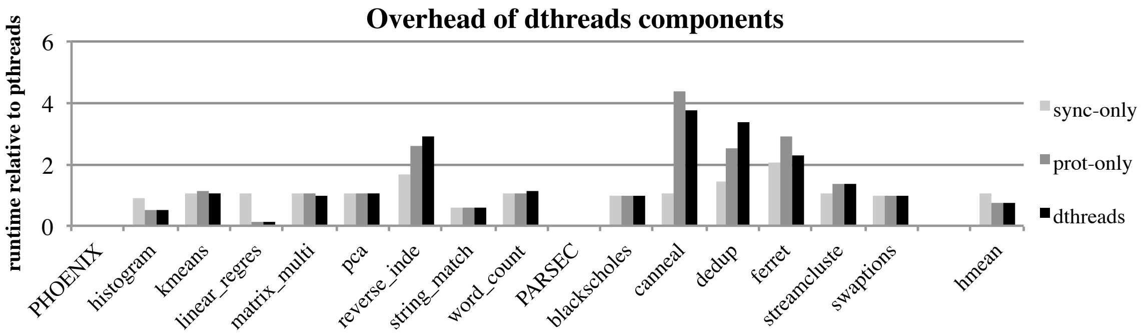


Figure 3.8. Normalized execution time with respect to `pthreads` (lower is better) for three different configurations.

Prot-only: This configuration runs threads in isolation, and commits at synchronization points. The order of synchronization and memory commits are non-deterministic. This configuration eliminates false sharing, but also introduces the performance overhead of isolation and memory commits. In order to guarantee correct execution, we replaced those synchronizations as corresponding cross-processes synchronizations. The lazy twin creation and single-threaded execution optimizations are disabled here because they are unsafe without deterministic synchronization. Thus, this configuration actually evaluates the performance of the SHERIFF framework.

The performance results of these two configurations are shown in Figure 3.8 and discussed in the following.

- The `reverse_index`, `dedup` and `ferret` benchmarks show significant load imbalance under *sync-only* configuration. Additionally, these benchmarks introduces significant overhead with *prot-only* configuration because of a large number of transactions there. That explains why DTHREADS doesn't have good performance on these benchmarks.

- The `string_match` benchmark shows performance improvement with *sync-only* configuration. The exact reason is not clear, may be due to our custom memory allocator (described in Section 2.3.2) that eliminates false sharing problems.

- The `linear_regression`, `histogram` and `swaptions` benchmarks improve performance with *prot-only* configuration. The memory isolation mechanism eliminates false sharing problems inside and contributes to the performance speedup.

- Normally the performance of DTHREADS is not better than the performance of the *prot-only* configuration. However, both `ferret` and `canneal` run faster with determinism enabled (under DTHREADS). Both are benefited from specific optimization described in Section 3.3. `ferret` benefits from the *single-threaded-execution*. The performance improvement of `canneal` is coming from the shared twin pages for all threads in parallel phases.

3.5 Discussion

All DMT systems must impose an order on updates to shared memory and synchronization operations. The mechanism used to isolate updates affects the limitations and performance of the system. DTHREADS represents a new point in the design space for DMT systems with some inherent advantages and limitations as follows.

3.5.1 Design Tradeoffs

CoreDet and DTHREADS both use a combination of parallel and serial phases to execute programs deterministically. These two systems take different approaches during parallel phases, as well as the transitions between phases:

Memory isolation: CoreDet orders updates to the shared memory by instrumenting all memory accesses that could reference shared data. Synchronization operations and updates to shared memory must be performed in serial phases, unless those updates are performed by owners of a block, which can issued in parallel phases. This approach results in high instrumentation overhead during parallel phases, but incurs no additional overhead when exposing updates to the shared state since they are shared already.

Dthreads takes an alternate approach: updates proceed at full speed, but are isolated using hardware-supported virtual memory. When a serial phase is reached, these updates are committed to the shared state in a deterministic order, with the help of the twinning-and-diffing mechanism described in Section 2.3.1.

A pleasant side-effect of Dthreads is the elimination of false sharing. Because threads work in separate address spaces, there is no need to keep caches coherent between threads during parallel phases. For some programs, this results in a performance improvement as large as 7× when compared to `pthreads`.

Phases: CoreDet employs a quantum-based scheduler: after the specified number of instructions is executed in a parallel phase, the scheduler transitions to a serial phase. This approach bounds the waiting time for any thread that are blocked to the quantum, reducing the load imbalance problem. One drawback of this approach is that transitions to a serial phase do not correspond to static program points. Any changes of code and input will result in a new, previously-untested schedule.

Transitions between phases are static in Dthreads. Any synchronization operation will result in a transition to a serial phase, and a parallel phase will resume once all threads have finished their critical sections. This makes Dthreads susceptible to delays due to load imbalance between threads, but results in more robust determinism. With Dthreads, only the order of synchronization operations affects the schedule. For most programs, this means that different inputs, and even many code changes, will not change the schedule produced by Dthreads, as long as those changes won't affect the order of synchronizations.

3.5.2 Limitations

This section analyzes some key limitations of Dthreads that restrict its ability to run certain programs, limit the extent of determinism it can guarantee, or potentially affect performance.

Unsupported programs: Dthreads supports programs that use the pthreads library, but does not support programs that bypass it by using their own ad hoc synchronization operations, such as those that use atomic operations. However, the upcoming C++0X standard includes a library interface for atomic operations [31, pp. 1107–1128], and a future version of Dthreads could intercept these library calls and treat them as synchronization points. While ad hoc synchronization is a common practice, it is also a notorious source of bugs; Xiong et al. show that 22–67% of the uses of ad hoc synchronization lead to bugs or severe performance issues [57].

Currently, Dthreads also does not share the stack across threads, so any updates to stack variables are only locally visible, which could cause a program to fail. However, communicating across different threads using stack variables is extremely error-prone and generally deprecated, making this a rare coding practice.

External determinism: While Dthreads provides internal determinism, it does not guarantee determinism when a program's behavior depends on external sources of non-determinism, such as system time or I/O events. Incorporation of Dthreads in the dOS framework, an OS proposal that enforces system-level determinism, would provide full deterministic execution, although this remains future work [5].

Runtime performance: Section 5.3 shows that Dthreads can provide high performance for a number of applications; in fact, for the majority of the benchmarks examined, Dthreads matches or even exceeds the performance of `pthreads`. However, Dthreads could occasionally degrade performance, sometimes substantially. One way it could do so would be to exhibit an intensive use of locks (that is, acquiring and releasing locks at very high frequency), which are much more expensive in Dthreads than in `pthreads`. However, because of its determinism guarantees, Dthreads could allow programmers to greatly reduce their use of locks, and thus improve performance. Other application characteristics, also explored in Section 3.4.3, can also impair performance with DTHREADS.

Memory consumption: Because DTHREADS creates private, per-process copies of modified pages between commits, it can increase a program's memory footprint by the number of modified pages between synchronization points. This increased footprint does not seem to be a problem in practice, both because the number of modified pages is generally far smaller than the number of pages read, and because it is transitory: all private pages are relinquished to the operating system (via `madvise()`) at the end of every commit operation.

Memory consistency: DTHREADS provides a form of release consistency for parallel programs, where updates are exposed at static program points. CoreDet's DMP-B mode also uses release consistency, but the update points depend on when the quantum counter reaches zero. To the best of our knowledge, DTHREADS cannot produce an output that is not possible with `pthreads`, although for some cases it will result in unexpected outputs. But the same unexpected output will be produced on every run with DTHREADS, making it easier for developers to track down the source of the problem than with `pthreads`.

CHAPTER 4

PRECISE DETECTION AND AUTOMATIC MITIGATION OF FALSE SHARING

False sharing is a well-known performance issue [12, 27]. We have discussed this problem in Section 1.2.

Detecting false sharing requires tools support. Existing tools share a similar shortcoming, where they can not pinpoint the exact place with false sharing problems, leaving the burden of finding actual places to programmers. Besides that, existing tools suffer from one or more different shortcomings. Simulation based approaches [53] and binary instrumentation based approaches [26, 40] normally introduce very significant performance overhead, slowing down the execution over 100×. Hardware performance counter based approaches generally provide much better performance, but they cannot differentiate false sharing from true sharing problems [28, 29].

We provide two systems, Sheriff-Detect and Sheriff-Protect, to tackle with false sharing problems, based on the Sheriff framework that discussed in Section 2. Sheriff is a drop-in replacement of the standard `pthreads` library, but providing "per-thread" protection and isolation mechanism. Sheriff-Detect detects false sharing problem accurately (without false positives) and precisely, by pointing out the exact places with false sharing problems. It is also very efficient, only introducing 20% performance overhead. Sheriff-Protect automatically tolerate false sharing problems when rewriting an application to resolve false sharing is infeasible or impractical. The reasons can be caused by either source code is unavailable, or padding data structures would degrade performance because of reduced cache utilization and/or increase memory footprint.

4.1 Detecting False Sharing

This section first describes the basic idea of detecting false sharing.

4.1.1 Basic Idea

False sharing occurs when more than two threads concurrently access independent data within the same cache line, at least one of them are writes. False sharing does not necessarily cause performance problems. It can greatly degrade performance only when those accesses, caused by threads running on different cores with separate cache, actually cause a big number of cache invalidations. This is our **basic observation**.

Generally, there are two known approaches to know how many cache invalidations actually occurring on a specific cache line, but they all suffer different shortcomings.

The first approach relies on the underlying hardware, called as "hardware-based approach". We may rely on specific hardware performance counters, existing in some special hardware but not all, to know this information. But we cannot have thorough information about cache invalidations since existing mechanisms are based on sampling, which can lost a lot of information. Also, a tool based on this approach cannot apply to a different hardware that do not have specific hardware support.

The second approach is to simulate the cache activity on different cache lines. To do that, we have to know all hardware-related information, including cache hierarchy, cache capacity and cache eviction rule, and the relationship between a thread and a specific core (that is hard to match actual situation). Even worse, simulation-based approaches are normally very slow and cannot be generalized to an execution running on a different hardware environment.

To avoid these problems, we provide a software-only and generalized approach that can only rely on memory access history of each cache line, which is used by both Sheriff-Detect and Predator (discussed in Section 5). Our approach is based on two conservative assumptions.

First assumption: All threads are running on different cores, with separate caches. Using this assumption can avoid knowing actual hardware cache hierarchy and the running situation between a thread and different cores. Although in a particular execution, two threads may run on the same core, thus reducing the effect of possible false sharing problems. Assuming that two threads are running on different cores can always represent a worst-case scenario that can happen in future executions. Thus, this assumption is very conservative, helping report any possible false sharing problem.

Second assumption: A cache entry is never evicted from its private cache by cache eviction, meaning that all caches have infinite capacity. This assumption allows us to compute a cache invalidation without considering whether this entry is still in the cache or not.

These two assumptions together allow us to compute cache invalidations based on memory accesses only. Based on these assumptions, we have the following **observation**: there is a cache invalidation if a thread writes a cache line after another thread's access on the same cache line. Because the last thread accessing this cache line creates a copy of the same cache line on its running core's private cache (first assumption) and holds this copy(second assumption), this write operation definitely causes a cache invalidation, which invalidates the data copy on the core accessed by last thread.

To locate cache lines with a big number of cache invalidations, we maintain a cache line status word for each cache line in the globals and heap, shown in Figure 4.1. We share a similar mechanism as another concurrent work of Zhao et.al. [58]. However, the detailed implementation is totally different. Zhao et.al. utilize the detailed ownership bitmap to track those cores that have a duplicate copy of data, which can even track how many cache invalidations may happen in a write operation. However, their design cannot be easily scaled to more than 32 threads, requiring more memory

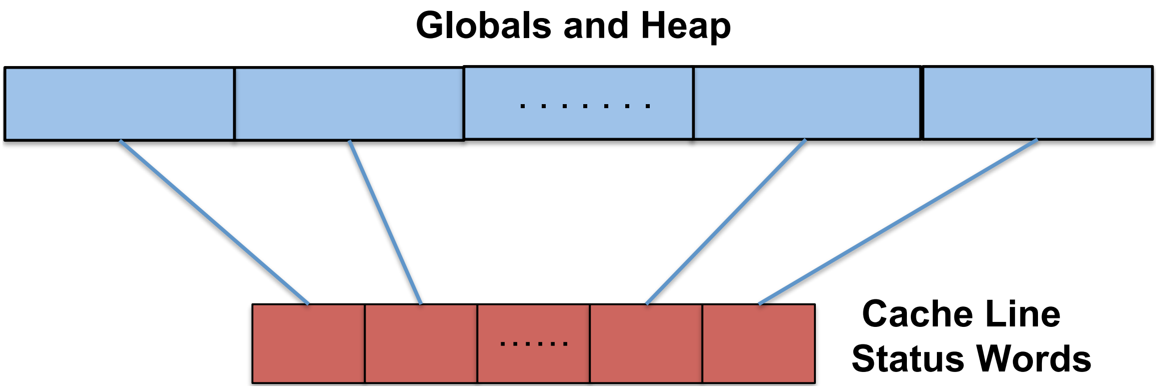


Figure 4.1. To detect false sharing, each cache line of the globals and heap maintains a cache line status word, which is updated on each memory access.

overhead caused by more bits and more checking performance overhead. Also, their approach misses one important factor – how many cache invalidations happening on a specific cache line. Without this information, it is impossible to pinpoint false sharing problems that can cause performance problems. Our approach overcomes these shortcomings, by only tracking the last thread index and the number of cache invalidations. Thus, we can rank the seriousness of false sharing problems based on the number of cache invalidations.

4.1.1.1 Accurate Detection

Accurate detection implies that we only report those false sharing problems that can cause performance problems. We employs the following mechanisms to avoid false positives.

First, we only report false sharing problems with a big number of cache invalidations, larger than a pre-defined but changeable threshold, thus can potentially cause performance problems. Utilizing the number of cache validations as an indicator avoids the problem of some existing tools, like PTU [28, 29]. PTU aggregates memory accesses without considering memory access interleaving, which can report some cases that has a big number of memory accesses but without many cache invalidations.

Second, we can differentiate false sharing from true sharing since true sharing can also cause cache invalidations. To do this, Zhao et.al. update bitmaps for every read and write in order to precisely determine whether an invalidation is related to a false sharing or a true sharing [58]. However, this approach brings scalability problem that can not scale to more threads, bringing more memory overhead. We achieve the same target differently: we do not differentiate false sharing from true sharing during normal executions, but only track word-level accesses information: how many reads or writes are issued by which thread, where a word accessed by multiple threads is as "shared". This design lets us accurately distinguish false sharing from true sharing in the reporting phase, while do not have the scalability issue. It also helps diagnose where actual false sharing occurs when there are multiple fields or multiple objects in the same cache line, as this can greatly reduce the manual effort required to fix the false sharing problems.

Third, we can avoid pseudo false sharing (false positives) caused by memory reuses. We intercept those memory allocations and deallocations, update information at memory deallocations for those objects without false sharing problems; heap objects involved in false sharing are never reused so that they can be reported in the end or on demand.

4.1.1.2 Precise Detection

Precise detection implies that we can precisely point out where the problem is. Thus, programmers can leverage on that to identify and correct false sharing problems.

For global variables, we identify the name of global variables involving in false sharing problems, by looking up corresponding debug information. For heap objects, we report the callsite of those memory allocations by presenting the line of source code. In order to capture the origins of heap objects, we intercept those intercepting memory allocations and deallocations and use different ways to get callsite information, which are discussed in Section 4.1.3.1 and Section 5.1.3.2.

To help programmers precisely identify culprits of performance problem, we also present word-level accesses information so that the exact variables or fields that cause performance problems can be determined precisely.

4.1.1.3 Flexible Reporting

We provide two different ways to report those false sharing problems. Normally, we can report those false sharing problems in the end of a program. However, this way does not work for those long-running applications. Thus, we provide a on-demand reporting way. User can send a specified signal to those applications that are installed with our tool. By intercepting those signals, we can report false sharing problems on demand.

In order to find out those cache lines with false sharing problems, we scan cache line status words of all memory, including the globals and heap, and only report those false sharing problems that can possibly cause performance problems, with the number of cache invalidations larger than a pre-defined but adjustable threshold.

4.1.2 Detailed Implementations

Sheriff-Detect relies on the Sheriff framework to track memory writes, thus detecting the write-write type of false sharing problems.

4.1.2.1 Tracking Memory Accesses

As discussed in Section 4.1.1, we can detect cache invalidations only based on memory accesses: for every memory access, we check recent memory access history, update the number of cache invalidations if possible, and update corresponding memory access history. Base on our observation, if the current access is a write, and other threads have accessed this cache line since last invalidation, then there is a cache in-

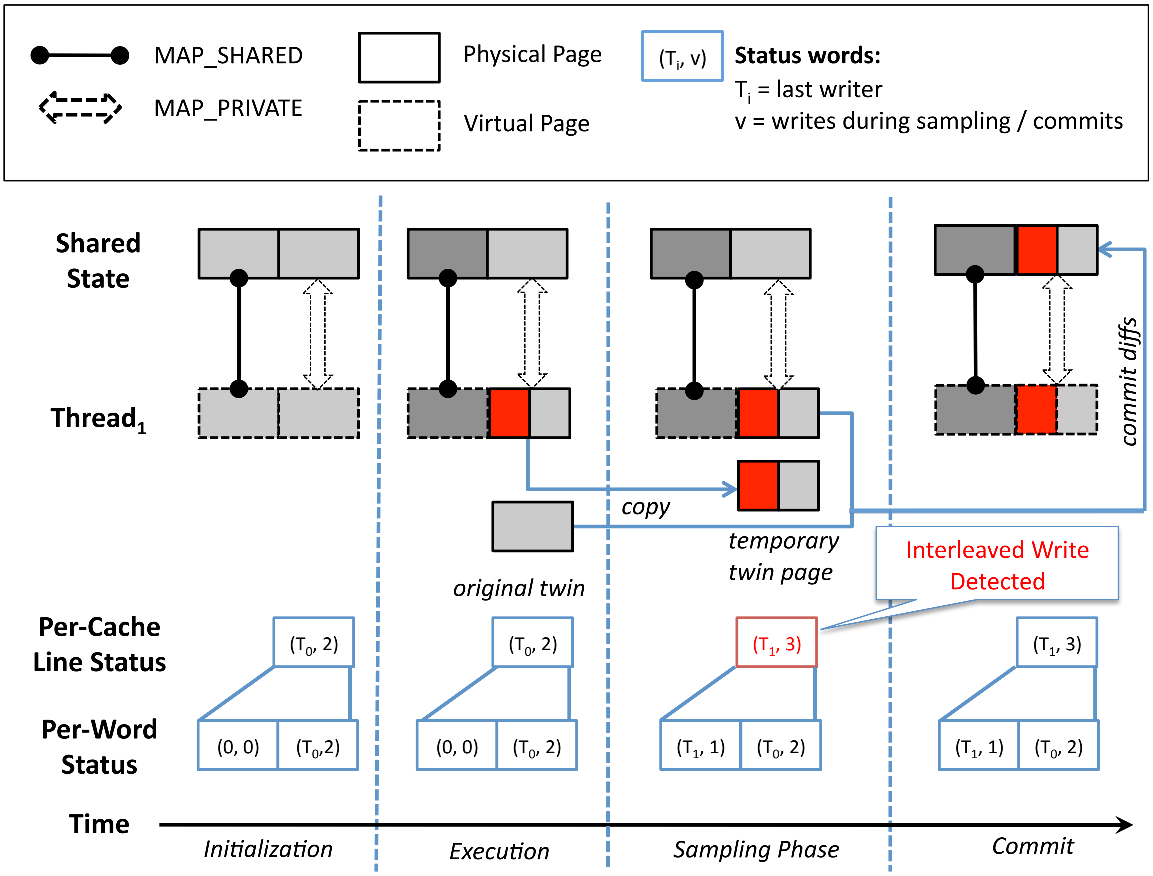


Figure 4.2. Overview of SHERIFF-DETECTs operations. SHERIFF-DETECT extends SHERIFF with sampling, per-cacheline status arrays, and per-word status arrays. For clarity of exposition, the diagram depicts just one cache line per page and two words per cache line.

validation. The SHERIFF framework isolates executions from different threads, only commits those changes of different threads to the shared mapping at synchronization boundaries. Thus, by comparing a "working" page against its "twin" page, SHERIFF-DETECT can discover those accumulative memory writes that occurred in the last transaction.

However, if a transaction is long-running, finding memory changes at the end of every transaction is not enough to find those false sharing problems happening in the middle of a transaction. For example, the `linear_regression` benchmark (described in Section 5.3), degrading the performance by more than 10× because of its false sharing problem, only has a single transaction per thread.

In order to detect memory writes in the middle of an transaction, Sheriff-Detect employs a sampling mechanism, employing the timer mechanism of the underlying operating system. We utilize the `alarm` library API to generate a periodical alarm to our detection system: by handling the `SIGALRM` signal, Sheriff-Detect tracks memory writes accumulatively in the current period using the twinning-and-diffing mechanism (section 2.3.1). To do this, Sheriff-Detect also keeps and updates a "temporary twin" page at every alarm interval, by simply copying from its "working" page. The difference between a "working" page and its "temporary" page implies those memory writes happening in the current sampling period.

Currently, Sheriff-Detect samples memory accesses of each thread at every 10 microsecond, which is adjustable in our implementation. More frequent sampling may uncover more false sharing problems, but at the cost of increasing performance overhead. The tradeoff between effectiveness and performance overhead is further discussed and evaluated in Section 4.3.3.

4.1.2.2 Tracking Cache Invalidations

As the discussion in Section 4.1.1, Sheriff-Detect tracks and reports those cache lines with a big number of cache invalidations, which may cause serious performance problems.

In order to track cache invalidations, Sheriff-Detect introduces a cache line status word for every cache line of the globals and heap, showed in Figure 4.1. Sheriff-Detect introduces two fields for every cache line status word, the last thread writing to this cache line and the number of cache invalidations of this cache line. Every time, when Sheriff-Detect tracks a memory write, either at the end of each transaction or during the sampling timer handler, it updates these two fields correspondingly. Based on the assumptions described in Section 4.1.1, Sheriff-Detect increments the number of cache invalidations when there is a write from a different thread and changes the last thread to the current thread (by recording thread id). To avoid using lock, Sheriff-Detect updates those counters using atomic primitives. Since we base on thread id to identify whether there is a cache invalidation, without keeping track of detailed ownership id, this approach can scale up to any number of threads.

4.1.3 Optimizations

Sheriff-Detect employs the following optimizations in order to reduce its performance overhead.

4.1.3.1 Getting Callsite Information.

Sheriff-Detect intercepts memory allocation operations in order to collect callsites for every heap object. To reduce the performance overhead, Sheriff-Detect does not use the `bracktrace()` function call, but identify the callsite by analyzing the return or frame address using GCC extensions. However, this can not work on applications without debugging information.

4.1.3.2 Reducing timer overhead.

As explained in Section 4.1.2.1, Sheriff-Detect uses a sampling mechanism to track cache invalidations. To reduce the performance overhead caused by by handling those alarm signals, Sheriff-Detect activates sampling only when the average transaction time is larger than a pre-defined threshold (currently 10 milliseconds). Sheriff-Detect uses an exponential moving average to track the average transaction time ($\alpha = 0.9$). This optimization does not significantly reduce the possibility of finding false sharing, since Sheriff-Detect can track those accumulative writes inside every short transaction by checking only at the end of transactions.

4.1.3.3 Sampling to find shared pages.

If an application has a large number of transactions or a large memory footprint, the overhead of handling page protection can dominate the total running time. To reduce the number of pages that should be tracked, Sheriff-Detect leverages a simple insight: if two threads can falsely share (write-write share) a cache line, then they must simultaneously write to the same page containing this cache line. Leveraging on this insight, Sheriff-Detect only tracks those pages written by multiple threads.

In order to identify those shared pages, Sheriff-Detect is based on the following assumption: if objects on a page are frequently falsely shared, the corresponding page must also be frequently shared; thus, even relatively infrequent sampling on memory accesses can reveal the shared relationship. Sheriff-Detect currently samples the first 50 out of every 1,000 periods (one period equals one transaction or one sampling interval). At the beginning of each sampling period, all memory pages are made read-only so that any writes to each page will be detected. Upon finding a page that is shared across multiple threads, Sheriff-Detect tracks all memory accesses happening on this page, thus possibly finding any false sharing inside this page.

By using this sampling mechanism, those pages, with sharing status unknown, impose no protection overhead at all. Sheriff-Detect only pays protection overhead for those shared pages outside the sampling period, instead of all memory pages.

4.1.4 Limitation

Unlike previous tools, Sheriff-Detect reports no false positives, differentiates true sharing from false sharing, and avoids false positives caused by the reuse of heap objects.

However, Sheriff-Detect can under-report false sharing instances in the following situations:

4.1.4.1 Single writer.

False sharing usually involves concurrent 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 this cache line. Because SHERIFF-DETECT can only track writes, it cannot detect this single-writer false sharing, missing some false sharing problems.

4.1.4.2 Heap-induced false sharing.

SHERIFF replaces the standard memory allocator with one that, like the Hoard allocator, avoids most heap-induced false sharing. SHERIFF's memory allocator (like Hoard), carves memory into page-sized chunks; each thread allocates from its own set of chunks, and the allocator never splits cache lines across threads. Because SHERIFF-DETECT uses a different custom memory allocator, it cannot detect false sharing that is caused by using the standard memory allocator. Since it is straightforward to deploy Hoard or a similar allocator to avoid heap-induced false sharing, this limitation is not a problem in practice.

4.1.4.3 Misses due to sampling.

Since it uses sampling to find shared pages, SHERIFF-DETECT may fail to track those pages that written in the middle of sampling intervals. We hypothesize that false sharing instances that affect performance are unlikely to perform frequent writes exclusively during that time, and so are unlikely to be missed.

4.2 Tolerating False Sharing

While SHERIFF-DETECT can effectively find those false sharing problems of multithreaded programs, it is sometimes difficult or impossible to fix them. For example, padding memory to avoid false sharing may even slowdown the performance because of excessive memory consumption or reducing cache utilization [58]. Also, time constraints or unavailable source code may prevent the fixes.

Based on the SHERIFF framework, we provide the second tool, SHERIFF-PROTECT, to automatically boost the performance for multithreaded applications with false sharing problems, without programmer intervention.

SHERIFF-PROTECT borrows the insight initially introduced by Dubois et.al. [25]: *delaying updates avoids false sharing.* Because SHERIFF replaces threads with processes, executions of different threads are actually isolated from each other. Thus, different "threads" (processes) actually access different physical pages (and cache lines), when originally they are accessing the same cache line in the multithreading environment. This helps avoid false sharing problems.

However, simply using the SHERIFF framework introduces excessive performance overhead because of the following reasons:

- The overhead of protecting and committing all pages may be too high. As we already know in Section 2, SHERIFF has to commit all local changes of different threads to the shared mapping at the end of every transaction (synchronization points) in order to achieve the shared memory semantics.

- If the length of a transaction is short, the overhead of protecting and committing pages in the SHERIFF framework can be easily higher than the performance benefit by tolerating possible false sharing problems inside. Thus, there is no benefit to tolerate false sharing problems for short-running transactions.

SHERIFF-PROTECT provides two corresponding mechanisms to avoid these possible overhead.

Selective Protection. SHERIFF-PROTECT only prevents false sharing on small objects, with size less than 1024 bytes. All large objects are mapped shared and are never protected, thus can not tolerate false sharing problems caused by these large objects. We expect small objects to be a likely source of false sharing because more of them can fit on a cache line. Also, for large objects, the cost of protecting and committing changes can be bigger than the benefit of tolerating possible false sharing problems inside.

Adaptive Prevention. SHERIFF-PROTECT employs a simple adaptive mechanism: it only isolates threads' executions if the average transaction length is large than a pre-set threshold. SHERIFF-PROTECT keeps track of the length of each transaction and uses a exponential weighted averaging ($\alpha = 0.9$) to calculate the average transaction length. If the average transaction length falls below an established threshold, SHERIFF-PROTECT switches to the shared mappings for all memory and does no further page protections. As long as transactions remain too short, without any benefit to tolerate false sharing problems inside, the protection mechanisms remain switched off. If the average transaction length rises back above the threshold, SHERIFF-PROTECT re-establishes private mappings and page protections, thus avoiding possible false sharing to achieve better performance.

4.3 Experimental Evaluation

We perform all of our evaluations on a quiescent dual processor (totally 8 cores) system with 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 compile all applications to a 32-bit target using the GCC compiler. All performance data is the average of ten runs, excluding the maximum and minimum values.

The evaluation answers the following questions:

- How effective is SHERIFF-DETECT at finding false sharing and guiding programmers to their resolution? (Section 4.3.1)

- What is SHERIFF-DETECT's performance overhead? (Section 4.3.2)

Microbenchmark	**Perf Sensitive**	**Sheriff-Detect**	**PTU**
False Sharing (adjacent objects)	YES	✔	✔
False Sharing (same object)	YES	✔	✔
True Sharing	NO		
Non-interleaved False Sharing	NO		✕
Heap Reuse(no sharing)	NO		✕

Table 4.1. False sharing detection results using PTU and SHERIFF-DETECT. SHERIFF-DETECT correctly reports only actual false sharing instances that have performance impact; ✔ indicates a correct report and ✕ indicates a false alarm.

- How sensitive is SHERIFF-DETECT to different sampling rates? (Section 4.3.3)
- How effective does SHERIFF-PROTECT mitigate false sharing? (Section 4.3.4)

4.3.1 Detection Effectiveness

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

We developed a range of microbenchmarks that exemplify different situations related to false sharing. We evaluate these benchmarks on both SHERIFF-DETECT and Intel's Performance Tuning Utility(PTU v3.2), the previous state-of-the-art work of false sharing detection.

Detection results are shown in Table 4.1. SHERIFF-DETECT only reports those false sharing instances that can possibly affect performance, while correctly ignores those cases without performance impact. PTU has false alarms/positives. It does not track access patterns, which reports false positives for those non-interleaved accesses. Also, PTU does not track memory deallocations, thus it can not filter out those pseudo false sharing caused by memory reuse. SHERIFF-DETECT avoids all of these problems and reports false sharing problems correctly.

We further evaluate SHERIFF-DETECT and PTU on two widely-used benchmark suites, Phoenix [52] and PARSEC [10]. We use the simlarge inputs for all applications of PARSEC. For Phoenix, we choose available parameters that allow the programs to

Benchmark	**PTU** # Lines	**Sheriff-Detect** # Objects
kmeans	1916	2
linear_regression	5	1
matrix_multiply	468	0
pca	45	0
reverseindex	N/A	5
word_count	4	3
canneal	1	1
fluidanimate	3	1
streamcluster	9	1
swaptions	196	0
Total	2647	14

Table 4.2. Overall detection results of PTU and SHERIFF-DETECT on Phoenix and PARSEC benchmark suites. We only list those benchmarks that at least one of tools reports false sharing problems. For PTU, we show how many cache lines are marked as falsely shared. For SHERIFF-DETECT, we show how many objects are reported by SHERIFF-DETECT (with cache invalidations larger than 100). The item marked as "N/A" means that PTU fails to show results because it runs out of memory.

run as long as possible. We were unable to successfully compile `raytrace` and `vips`, and SHERIFF is currently unable to run `x264`, `bodytrack`, and `facesim`. `Freqmine` currently can not support `pthreads`. Thus, those benchmarks are excluded here.

The overall results are shown in Table 4.2. PTU reports that 2647 cache lines may exist false sharing problems. SHERIFF-DETECT reveals that seven out of sixteen evaluated benchmarks have false sharing problems. Totally, only 14 objects are reported, but only 4 of them shows a big number of cache invalidations, thus needs to be fixed.

Several reasons contribute to the number difference between these two approaches. First, PTU reports cache lines involving in false sharing, while SHERIFF-DETECT only reports objects. If an object has a size larger than the size of cache line, PTU can report multiple times, one on each cache line. Second, PTU reports multiple times if a heap object, with the same allocation site, is allocated multiple times, while

Benchmark	**Performance Improvement**	**Updates** (M)
linear_regression	818%	1323.6
reverseindex	2.4%	0.4
streamcluster	5.4%	28.7
word_count	1%	0.3

Table 4.3. Performance data for four false sharing benchmarks. All data are obtained using the standard `pthreads` library. "Updates" shows how many million updates (in total) occurred on falsely-shared cache lines.

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

Figure 4.3. A fragment of source code from `reverse_index`. False sharing arises when different threads modify different words in the same `use_len` array.

Sheriff-Detect only reports once. Third, PTU may report false positives since it does not track interleaved accesses and overrates the problems caused by heap reuses.

We manually fix these four false sharing problems based on reports of Sheriff-Detect, and show the performance gains after fixes in Table 4.3. To explain why performance improvement are different, we also examine the maximum possible updates that can occur on a false sharing object, although the actual number of interleaved accesses depends on actual scheduling. For example, `linear_regression` has the largest updates, thus causing the most serious performance problem.

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 4.3. For these two benchmarks, we can use thread-local variables to avoid performance problems: each thread can operate on a temporary variable first, and then modify the `use_len` array at the end.

```
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 4.4. A fragment of `linear_regression` code. Each thread works on its independent elements of the array. Unfortunately, the size of `struct lreg_args` is not large enough (only 52 bytes) on 32-bit machine, which causing two different threads to write to the same cache line simultaneously.

`Linear_regression`'s false sharing problem is a little different (see Figure 4.4). Two different threads write to two independent parts of the same cache line, when these parts (caused by the size of`lreg_args` structure) are not large enough to occupy a cache line. This problem can be avoided easily by padding the structure `lreg_args`, thus preventing different threads concurrently accessing the same cache line.

The false sharing problem detected in `streamcluster` (one of the PARSEC benchmarks) is similar to that in `linear_regression`: two different threads are writing to the same cache line. Examination of the source code indicates that the author tried to avoid false sharing by padding, but the amount of padding, 32 bytes, was insufficient to accommodate the actual physical cache line size used in the evaluation (64 bytes). Setting the `CACHE_LINE` macro to 64 bytes reduces the effect of false sharing, improving the performance by 5.4%.

4.3.1.1 Ease of Locating False Sharing Problems

To illustrate how SHERIFF-DETECT can precisely locate false sharing problems, we use one benchmark (`word_count`, a Phoenix benchmark) as an example. Diagnosing other false sharing issues is similar to this one.

Here is an example output from SHERIFF-DETECT for `word_count`.

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

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

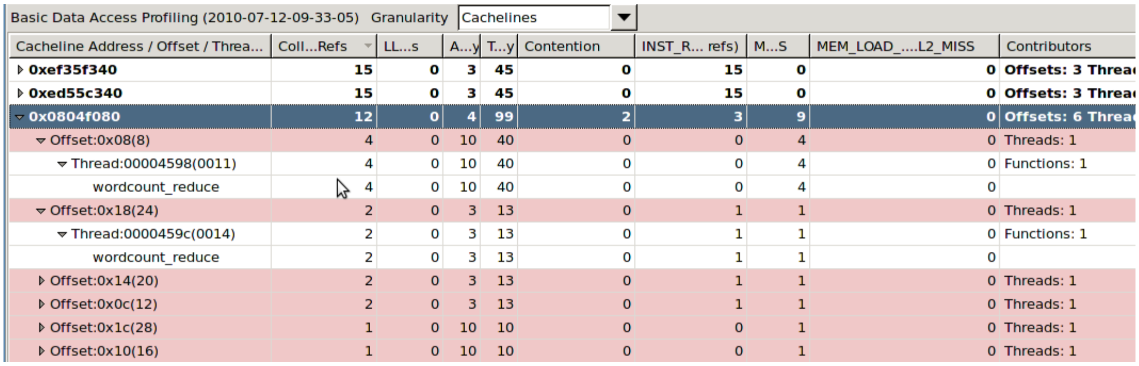
```
use_len=malloc(num_procs*sizeof(int));
```

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

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

Now it is very clear that different threads are modifying the same object(use_len). Fixing the problem by using a thread-local data copy is now straightforward.

By contrast, we can compare PTU's output that shown in Figure 4.5. Pinpointing the false sharing problem inside is far more complicated with PTU: it only reports functions involving in a questionable 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, while SHERIFF-DETECT observes 13767 cache invalidations.



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 4.5. PTU output for `word_count`.

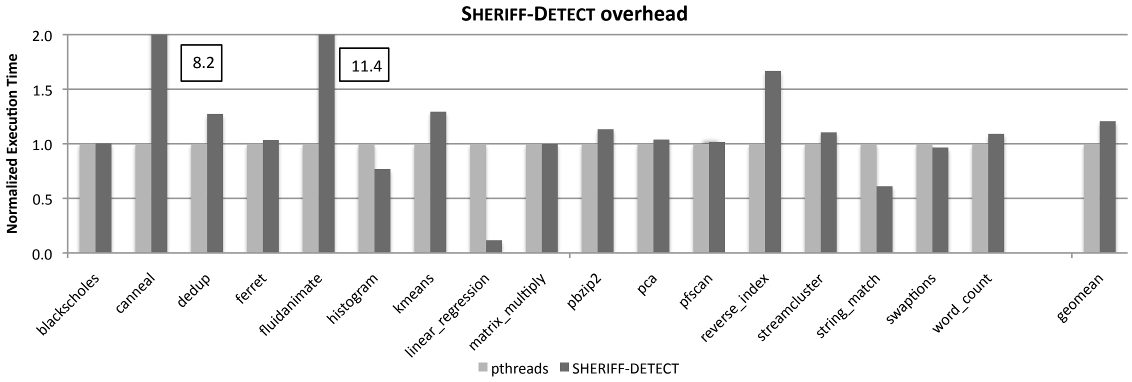


Figure 4.6. SHERIFF-DETECT performance overhead across two suites of benchmarks, normalized to the runtime of using the `pthreads` library (lower is better).

4.3.2 Detection Performance Overhead

SHERIFF-DETECT's runtime overhead (comparing to `pthreads`) on two multithreaded benchmarks suites, Phoenix and PARSEC, is shown in Figure 4.6. SHERIFF-DETECT only introduces 20% performance on average, with the exception of three outliers. For other benchmarks, SHERIFF-DETECTs overhead is generally acceptable and far lower than most existing tools.

SHERIFF-DETECT do not perform well on two benchmarks. `canneal` runs about 7× slower than that with `pthreads`. `fluidanimate`'s performance overhead is about 11× slower than that using `pthreads`.

The first reason is that both benchmarks trigger a high number of dirtied pages (3.4 million and 2.15 million, respectively). For each dirty page, SHERIFF-DETECT applies page protection twice, creates a "copy-on-write" page and a "twin" page, checks false sharing problems at every sampling interval, and commits those local changes to the shared mapping. Thus, given large amount of dirty pages, copying overhead alone is very expensive and can dominate most of overhead. For example, `Canneal` invokes around 3.4 million dirty pages, thus leading to substantial overhead.

Another reason for `fluidanimate` is that it invokes an unusually high number of transactions (16.7 million). SHERIFF-DETECT introduces page protection and commits overhead at every transaction boundary, thus, adding overhead if there are dirty pages.

However, even with these outliers that run slowly, the overhead of SHERIFF-DETECT is generally acceptable and far lower than most existing tools. SHERIFF-DETECT actually improves performance by eliminating false sharing, using its process-as-threads framework. SHERIFF-PROTECT further reduces overhead as the next section describes.

`linear_regression` runs 8× faster with SHERIFF-DETECT than with `pthreads`, even with the added overhead of protection, memory commits, sampling and other mechanisms. There is a serious false sharing problem inside (see Table 4.3,) which both SHERIFF-DETECT and SHERIFF-PROTECT eliminate automatically. Other cases where SHERIFF-DETECT outperforms `pthreads` are also due to false sharing elimination.

4.3.3 Detection Sampling Rate Sensitivity

SHERIFF-DETECT employs the sampling mechanism to detect false sharing happening in long-running transactions. Sampling is only triggered when the length of a transaction exceeds a pre-defined threshold, usually 10ms. By handling those

SIGALARM signals, Sheriff-Detect tracks memory accesses by by comparing the temporary twin page against its corresponding working version, and updates status words of specific cache lines. Thus, increased sampling rates may uncover more false sharing problems, but at the cost of increase performance overhead.

To measure Sheriff-Detect's sensitivity to different sampling rates, we evaluate on three different sampling rates: 2ms, 10ms (our baseline), and 50ms.

4.3.3.1 Sampling Overhead

Figure 4.7 shows the performance overhead under different sampling rates, normalized to the runtime of using the default 10ms sample rate. For most of these benchmarks, sampling imposes relatively little overhead either because the average number of shared pages is small, or because the transaction length is often shorter than the sampling interval (thus no adding checking overhead).

One outlier is `canneal`, which is extremely sensitive to a different sampling rate. When the sampling rate is 2ms, `canneal` runs about 2.3× slower than that with a 10ms sampling rate; `canneal` runs 35% slower with a 50ms sampling rate than 2 10ms sampling rate. The reason is that `canneal` dirties a large number of shared pages. More frequent sampling thus creates more temporary "twin" pages and increases checking overhead.

4.3.3.2 Sampling Effectiveness:

The choice of sampling rates has relatively little impact on detection and ranking, shown in Table 4.4. As expected, the number of falsely-shared objects reported and the number of interleaved writes observed are not significantly different.

Using a different sampling rate does affect the number of falsely-shared objects detected, but Sheriff-Detect already reports all instances with a significant performance impact under the default sampling rate. Increasing the sampling rate to 2ms (more frequent sampling) reveals two additional falsely-shared objects (in `ferret`

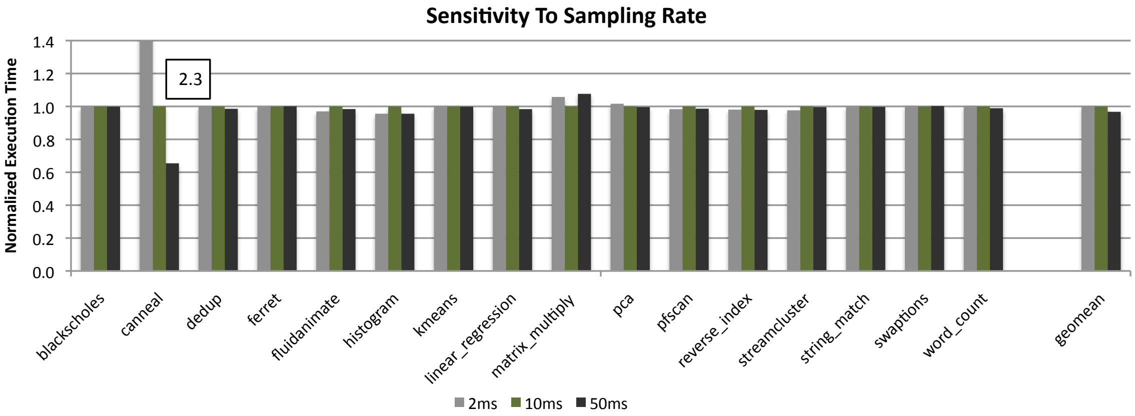


Figure 4.7. Sheriff-Detect performance with different sampling rates, normalized to the performance with a sampling interval of 10ms (presented in Figure 4.6); lower is better.

Benchmark	**2ms**		**10ms**		**50ms**	
	objs	*writes*	*objs*	*writes*	*objs*	*writes*
`canneal`	1	21444321	1	26369324	1	30580451
`ferret`	1	3	0	0	0	0
`fluidanimate`	1	3370	1	4064	1	2851
`kmeans`	2	2974	2	1122	1	98
`linear_regression`	1	1050	1	311	1	71
`reverse_index`	5	14494	5	14782	5	14981
`streamcluster`	2	52462	1	52283	1	52420
`word_count`	4	9849	4	2699	3	622

Table 4.4. Sheriff-Detect precision with different sampling rates, including the number of falsely-shared objects and interleaved writes. We omit those benchmarks with no observed cases of false sharing.

and `streamcluster`), but these two objects do not have a significant performance impact since they can only cause few cache invalidations(under 10). Similarly, reducing the sampling rate to 50ms (less frequent sampling) cannot detect two false sharing problems (in `kmeans` and `word_count`), but these objects also have little impact on performance.

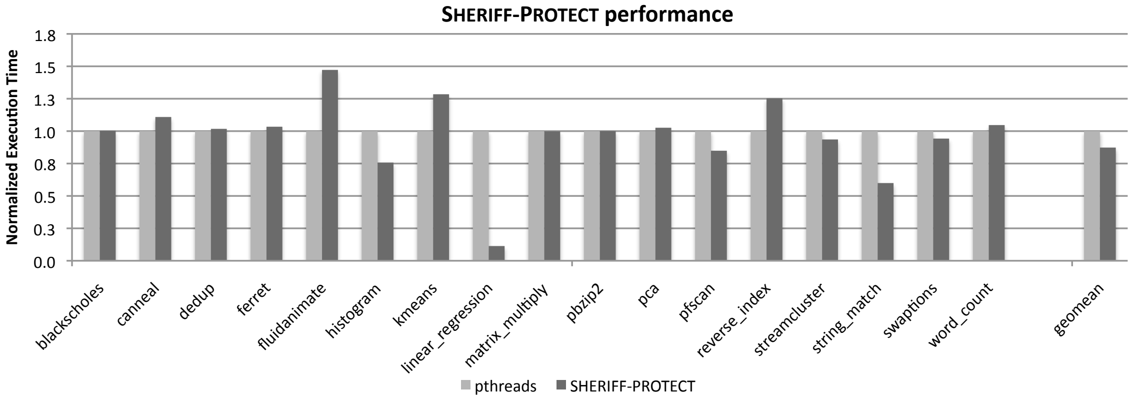


Figure 4.8. Sheriff-Protect performance across two suites of benchmarks, normalized to the performance of `pthreads` (see Section 4.3.2). In case of catastrophic false sharing, Sheriff-Detect dramatically increases performance.

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

Table 4.5. Detailed execution times with Sheriff-Detect and Sheriff-Protect, normalized to execution with the `pthreads` library; numbers below 1 (boldfaced) indicate a speedup over `pthreads`.

4.3.4 Prevention Effectiveness

We also examine the effectiveness of eliminating false sharing problems of using Sheriff-Protect. Figure 4.8 presents the performance under Sheriff-Protect and `pthreads`. For most cases, Sheriff-Protect either has no effect on performance (when there is no false sharing problem inside) or improves the performance. Table 4.5 presents detailed performance results of Sheriff-Detect and Sheriff-Protect.

Sheriff-Protect improves the performance when an application is detected to have false sharing problems inside. `linear_regression` exhibits almost a 10× speedup against the one using `pthreads`, by tolerating a serious false sharing problem inside (see Table 4.3). `histogram` runs substantially faster with Sheriff-Protect (24%) because of preventing a read-write false sharing problem, see Section 5.3.1. `string_match` runs 40% faster because of its custom memory allocator, preventing two threads allocating different objects from the same cache line, which is why Sheriff-Detect does not find.

Using Sheriff-Protect, three benchmarks runs up to 47% slower than using `pthreads` because of different reasons. `kmeans` creates more than 3000 threads in eight seconds. Since the overhead of creating one process is higher than that of creating one thread, this dominates most of its overhead. For `reverse_index` and `fluidanimate`, they exhibit slowdown because of using the processes-as-threads framework: Operating on those file-based pages is more expensive than operating on anonymous pages (the normal status of heap pages) under the Linux operating system; Writing to one page (`MAP_SHARED`) cause a Copy-On-Write operation in the kernel even when there is only one user.

Because `fluidanimate` has an enormous number of transactions(18 Million), Sheriff-Protect introduces some additional overhead for every transaction. That also accounts for part of overhead.

CHAPTER 5

PREDATOR: PREDICTIVE FALSE SHARING DETECTION

This chapter presents PREDATOR, which improves the effectiveness of false sharing detection. SHERIFF-DETECT reports false sharing accurately and precisely with only 20% performance overhead. However, it can only detect the write-write type of false sharing for those programs using `pthreads`. SHERIFF-DETECT can also break programs that communicate across different threads using stack variables or self-defined synchronizations. These shortcomings greatly limit SHERIFF-DETECT's usage on real-world applications.

In contrast to SHERIFF-DETECT, PREDATOR detects all types of false sharing and has no limitations on applications. PREDATOR has been utilized to find actual false sharing problems of real applications, including `MySQL` and the `Boost` library.

In addition, SHERIFF-DETECT and other systems share one key limitation: they can only report *observed* cases of false sharing. As Nanavati et al. point out, false sharing is sensitive to where objects are placed in cache lines and so can be affected by a wide range of factors [46]. For example, using the gcc compiler *accidentally* eliminates false sharing of the linear_regression benchmark at certain optimization levels, while LLVM does not do so at any optimization level. A slightly different memory allocation sequence (or different memory allocator) can reveal or hide false sharing, depending on where objects end up in memory; using a different hardware platform with different addressing or cache line sizes can have the same effect. All of this means that existing tools cannot root out potentially devastating cases of false sharing that could arise with different inputs, in different execution environments, and on different hardware platforms.

PREDATOR is the first system that can *predict* potential false sharing that does not manifest in an execution, but may appear and greatly degrade the performance of programs in a slightly different environment. Predictive false sharing generalizes from a single execution to identify potential false sharing instances that fall into two adjacent cache lines, which could be exposed by slight changes in object placement and alignment. It also can predict false sharing in hardware platforms with larger cache line sizes by tracking accesses within *virtual cache lines* that span multiple physical lines. Predictive false sharing detection thus help avoids the predicament of previous detection tools: those problems can easily leak to deployment environment because of the changed execution environment.

Here, we first describe PREDATOR's false sharing detection mechanism in Section 5.1, which consists of both compiler and runtime system components. Section 5.2 then explains how PREDATOR predicts potential false sharing based on a single execution.

5.1 False Sharing Detection

5.1.1 Overview

PREDATOR follows the same idea of detecting false sharing, described in Section 4.1.1. We compute cache invalidations based only on memory accesses history of each cache line, and only report those instances that may affect performance.

PREDATOR is based on the similar *basic observation*: if a thread writes a cache line after other threads have accessed the same cache line, this write operation causes at least a cache invalidation. Drawing from this observation, PREDATOR tracks cache invalidations of all cache lines and ranks the severity of performance degradation according to the number of cache invalidations.

To track memory accesses, PREDATOR relies on the compiler instrumentation. The design tradeoff of choosing compiler instrumentation, instead of other mechanisms, has been discussed in Section 5.4.1. A compiler can easily identify read or write accesses. However, a compiler does not know how and when those instructions are being executed, since that depends on a specific execution, input, and runtime environment.

Therefore, PREDATOR combines a runtime system with compiler instrumentation to track cache invalidations: the compiler instruments memory accesses so the runtime system is notified when an access is executed (see Section 5.1.2), and the runtime system is responsible for collecting and analyzing actual memory accesses to detect and report false sharing (see Section 5.1.3).

5.1.2 Compiler Instrumentation

PREDATOR relies on LLVM to perform instrumentation at the intermediate representation level [38]. It traverses all functions one by one and searches for memory accesses to those global and heap variables. For each memory access, PREDATOR instruments a function call to invoke the runtime system, passing with the address, access type and unit of this memory access (how many bytes). PREDATOR currently omits accesses to stack variables by default because stack variables are normally used for thread local storage and therefore do not normally introduce false sharing. However, instrumentation on stack variables can always be turned on when necessary.

The instrumentation pass is placed at the very end of the LLVM optimization passes so that only those memory accesses surviving all previous LLVM optimization passes are instrumented. This technique is similar to the one used by AddressSanitizer [54].

5.1.3 Runtime System

PREDATOR's runtime system collects every memory access by handling those functions calls inserted during the compiler instrumentation phase. It analyzes possible cache invalidations based on the basic observation discussed in Section 5.1.1. Finally, it precisely reports any performance-degrading false sharing problems it finds. For global variables involved in false sharing, PREDATOR reports their name, address and size; for heap objects, PREDATOR reports the callsite stack for their allocations, address and size. In addition, PREDATOR provides word granularity access information for those cache lines involved in false sharing: how many reads or writes are issued by which thread. This information can further help users diagnose and fix false sharing instances.

5.1.3.1 Tracking Cache Invalidations

PREDATOR only reports those global variables or heap objects on cache lines with a large number of cache invalidations, thus possibly affecting performance of applications. To track cache invalidations, PREDATOR maintains a two-entries-cache-history table for each cache line. In this table, each entry has two fields: thread ID and access type (read or write). Thread ID is used to identify the origin of each access. As stated earlier, only accesses from different threads (with a different thread ID) can cause a cache invalidation.

For every new access to a cache line L, PREDATOR checks L's history table T to decide whether the current access leads to a cache invalidation. As described in Section **??**, only write accesses can cause cache invalidations and read accesses only create a copy of data in the cache of the current core that the current thread is running on. Also, it is noticed that table T only has two status: full and not full. There is no "empty" status since every cache invalidation should replace its table with the current write access, setting the first entry to the current access (with its thread ID and write access type).

- For a read access R,
  - If T is full, there is no need to record this read access.
  - If T is not full and another existing entry has a different thread ID, then PREDATOR records this R (and its thread) by adding a new entry to the table.
- For a write access W,
  - If T is full, then W can cause a cache invalidation since at least one of two existing accesses are issued by a different thread (one thread can only occupy one entry). After recording this invalidation, PREDATOR updates the existing entry with W(and its thread).
  - If T is not full, PREDATOR checks whether W and the existing entry has the same thread ID. If so, W cannot cause a cache invalidation and there is no need to do anything. Otherwise, PREDATOR identifies a possible cache invalidation on this line: it increments the number of cache invalidations and updates the existing entry with the current W access.

5.1.3.2 Reporting False Sharing

PREDATOR reports false sharing precisely and accurately. Accurately means PREDATOR only reports those false sharing instances with a large number of cache invalidations, which may possibly cause performance problems. PREDATOR also differentiate actual false sharing from true sharing, since true sharing can also induce a large number of cache invalidations.

PREDATOR employs the following mechanisms to achieve this target, as well as reducing the performance overhead.

- In order to accurately differentiate those false sharing problems with true sharing problems, Predator tracks word-level accesses for those cache lines involved in a big number of cache invalidations, which has been discussed in Section 4.1.1.1.

- Predator relies on `backtrace()` function in the `glibc` library to obtain the whole callsite stack, which is slower but much more robust to be used than the ways used in Sheriff-Detect. Thus, it can report the callsite stack for those heap objects.

- For every access, Predator needs to lookup the corresponding cache line's metadata. Because this operation is so frequent, at every access, lookups need to be very efficient. Like AddressSanitizer [54] and other systems [47, 58], Predator uses a shadow memory mechanism to store metadata for every piece of application data. Thus, Predator can compute and locate corresponding metadata directly via address arithmetic.

- In order to support shadow memory, Predator uses a pre-defined starting address and fixed size for its heap. It also contains a custom memory allocator, which is described in Section 2.3.2. However, using this custom memory allocator also implies that false sharing caused by a memory allocator cannot be detected by Predator: two threads allocate heap objects from the same cache line concurrently. But this should not be a serious problem since all modern memory allocators, like Hoard, already avoid this kind of false sharing and we should always use this kind of memory allocator.

5.1.4 Optimizations

Tracking every memory access can be extremely expensive, thus Predator utilizes the following mechanisms to further reduce performance and memory overhead.

5.1.4.1 Threshold-Based Tracking Mechanism

Predator aims to detect false sharing that significantly degrades performance. Since cache invalidations are the root cause of performance degradation and only writes can possibly cause cache invalidations, cache lines with a small number of writes are never be a target with a significant performance impact. For this reason, Predator only tracks cache invalidations once the number of writes to a cache line crosses a pre-defined threshold, which we refer to as the *Tracking-Threshold*. Before this threshold is reached, Predator only tracks the number of writes on a cache line while skipping tracking reads. This mechanism reduces performance and memory overhead at the same time.

In the current implementation, Predator maintains two arrays in shadow memory: *CacheWrites* tracks the number of memory writes on every cache line, and *CacheTracking* tracks detailed information for each cache line once the number of writes on a cache line exceeds the *Tracking-Threshold*. If the threshold is not reached, there is no need to check the corresponding *CacheTracking*. Figure 5.1 illustrates the detailed mechanism.

To avoid expensive lock operations, Predator uses atomic instruction to increment the *CacheWrites* counter for each cache line. When the number of writes of a cache line reaches the predefined threshold, it allocates space to track detailed cache invalidations and word-level information. Predator also uses an atomic compare-and-swap to set the cache tracking address for this cache line in the shadow mapping. After *CacheWrites* on a cache line reaches the *Tracking-Threshold*, all read and write accesses on this cache line are tracked.

5.1.4.2 Selective Compiler Instrumentation

Predator relies on compiler instrumentation to provide memory access information to the runtime system and detects false sharing based on the sequences of

```
void HandleAccess(unsigned long addr, bool isWrite) {
 unsigned long cacheIndex=addr>>CACHELINE_SIZE_SHIFTS;
 cachetrack *track=NULL;

 if(CacheWrites[cacheIndex]<TRACKING_THRESHOLD) {
  if(isWrite) {
   if(ATOMIC_INCR(&CacheWrites[cacheIndex])
      ==TRACKING_THRESHOLD-1) {
     track=allocCacheTrack();
     ATOMIC_CAS(&CacheTracking[cacheIndex],0,track));
   }
  }
 }
 else {
  track=CacheTracking[index]);
  if(track){
    // Track cache invalidations and detailed accesses
    track->handleAccess(addr, isWrite);
  }
 }
}
```

Figure 5.1. Pseudo-code of handling an access in Predator.

memory accesses on every cache line. The performance overhead of a specific program is always proportional to the degree of instrumentation: more instrumentation generally indicates more performance overhead. Thus, Predator provides a flexible framework to instrument programs depending on the performance requirements of the user.

Currently, Predator only instruments once for each type of memory access on each address to the same basic block. This selective instrumentation does not normally affect the effectiveness of detection. Because Predator aims to detect false sharing cases with a large number of cache invalidations, less tracking of accesses inside a basic block can induce fewer cache invalidations, but it should not affect the overall behavior of cache invalidations.

To improve performance further, Predator can be easily extended to support more flexible instrumentation as follows:

- Predator could selectively instrument both reads and writes or only writes. Instrumenting only writes reduces overhead while detecting write-write false sharing, as Sheriff does.

- Predator can be set to instrument or skip specific code or data. For example, the user could provide a black-list so that given modules, functions or variables are not instrumented. Conversely, the user could provide a white-list so that only specified functions or variables are instrumented.

5.1.4.3 Sampling Mechanism

As described in Section 5.1.4.1, when the number of writes on a cache line is larger than *Tracking-Threshold*, every access must be tracked in detail: we have to track word-level information, update the number of accesses and possible cache invalidations, and update the cache access history table of this cache line. When a cache line is involved in false or true sharing, updating those counters can exacerbate the performance impact of sharing: not only is there an cache invalidation on this application's cache line, but there is also at least another cache invalidation caused by updating the metadata of this corresponding cache line.

To further reduce the performance overhead, Predator only samples the first specified number of accesses during each sampling interval. Currently, Predator maintains an access counter for each cache line and only tracks the first $10,000$ accesses out of every 1 million accesses on a cache line, with 1% sampling rate. Section 5.3.4 further evaluates the effect of different sampling rates on performance and effectiveness.

5.2 False Sharing Prediction

This section further motivates predictive false sharing and explains how to support it in the runtime system.

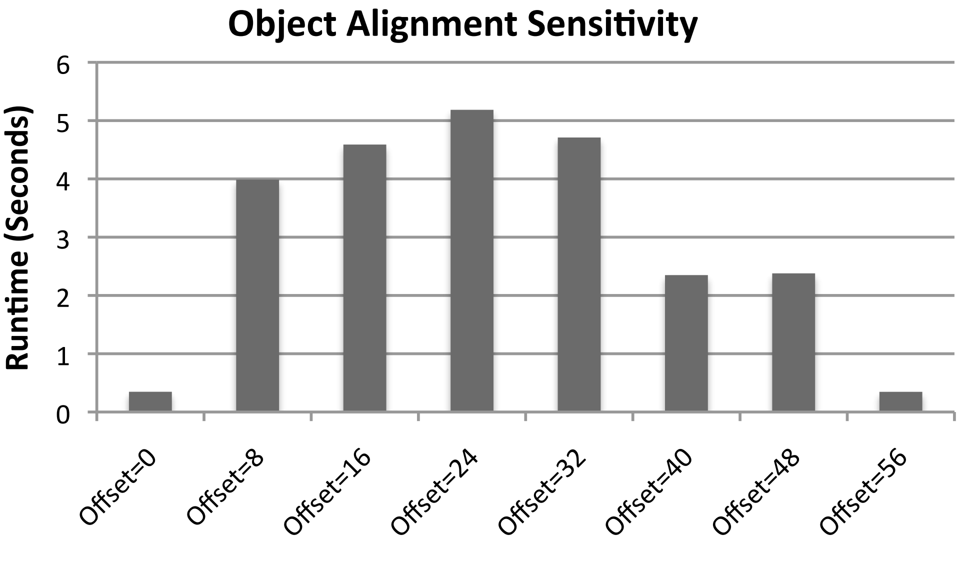
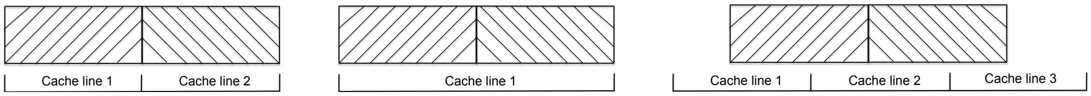


Figure 5.2. Performance of the linear_regression benchmark (from Phoenix) is highly sensitive to the memory layout between the (potentially) falsely-shared object and corresponding cache lines.

5.2.1 Overview

The appearance of false sharing depends on the memory layout between objects and corresponding cache lines. The performance of a real example, linear_regression, is shown in Figure 5.2: When the offset of the starting address between the potentially falsely-shared object and corresponding cache lines is 0 or 56 bytes, there is no false sharing; When the offset is 24 bytes, we see the most severe performance effect caused by false sharing. The performance difference caused by false sharing can affect the performance as large as 15× on an 8-core machine.

Existing detection tools can only report observed false sharing. That means, they may miss such a very severe false sharing problem that could occur in the wild if the offset of the starting address was 0 bytes or 56 bytes in their test environment. Predator overcomes this shortcoming by accurately predicting potential false sharing, without the need of occurrences.



(a) No false sharing (b) False sharing with larger cache size (c) False sharing with different memory layout

Figure 5.3. False sharing under different environments.

Predator predicts *potential false sharing*, which does not manifest in the current execution but may appear and greatly affect performance of programs in a slightly different environment.

Figure 5.3 shows a simplified example why occurrences of false sharing can change in different situations. In this figure, two rectangles with different patterns represents two portions of the same object, updated by different threads. In Figure 5.3(a), there is no false sharing when thread T1 only updates "cache line 1" and T2 only updates "cache line 2". However, false sharing appears in one of the following cases, even with the same access pattern.

- Doubling cache line size (Figure 5.3(b)). When the size of a cache line doubles, both T1 and T2 access the same cache line concurrently, thus causing false sharing.

- Different starting address of an object(Figure 5.3(c)). When the starting address of this object is not aligned with the starting address of the first cache line, then T1 and T2 can update the second cache line simultaneously, causing a false sharing problem.

Predator predicts whether programs can have potential false sharing in one of these two situations, where they can be caused by different dynamic properties. These dynamic properties include choosing a different compiler, enabling different compiler optimizations, using a different memory allocator, adding or removing code involving in memory allocations, switching to a different target platform with a different address mode (32-bit or 64-bit), and changing the size of cache line (64 Bytes or 128 Bytes). All dynamic properties, except changing the size of cache line, can lead to a different memory layout, thus can possibly affect occurrences of a false sharing problem. Thus, predicting false sharing in changing the memory layout or changing the size of cache line actually explores many possibilities caused by all of these dynamic properties.

5.2.2 Basic Prediction Workflow

Predator focuses on potential false sharing that can cause performance problems. It is based on two key observations. First, only accesses to adjacent cache lines can lead to potential false sharing, i.e., introducing cache invalidations when the cache line size or an object's starting address changes. Second, only those cache lines with a large number of cache invalidations can degrade performance.

Based on these two observations, Predator develops the following workflow to predict potential false sharing. Those detection optimizations listed in Section 5.1.4 can also be applied to here. We do not repeat these optimizations in this section.

1. Track the number of writes on different cache lines.

2. When the number of writes to a cache line L reaches *Tracking-Threshold*, Predator tracks the detailed read and write accesses for every word on both cache line L and its adjacent cache lines.

3. When the number of writes to a cache line L reaches a second threshold (called as *Predicting-Threshold*), Predator identifies whether there exists false sharing in L and its adjacent cache lines by analyzing word accesses information collected in Step 2, which are described in Section 5.2.3 in detail.

4. If a potential false sharing is found, PREDATOR starts to track cache line invalidations in order to confirm its seriousness, which are discussed in Section 5.2.4. Otherwise, go back to Step 2 to track more detailed accesses.

5.2.3 Searching for Potential False Sharing

To describe potential false sharing in two different cases, we first introduce a concept – "virtual cache line". A virtual cache line is a contiguous memory range that spans one or more physical cache lines.

In the case of double cache line size, a virtual line is composed of two originally contiguous cache lines, where it starts with a even number cache line. Thus, only cache lines $2 * i$ and $2 * i + 1$ can form a virtual cache line.

To evaluate a potential false sharing problem that can be caused by changing memory layout, a virtual line should have the same size as an actual cache line, but with a different starting address: unlike actual cache lines, the starting address of a virtual cache line does not need to be multiple of the cache line size. For instance, a 64-byte-long virtual line can consist of the range $[0, 64)$ bytes or $[8, 72)$ bytes.

To search for a potential false sharing problem, PREDATOR searches for a pair of hot accesses, one on L and one on its previous or next cache line, based on detailed word information collected in Step 2. Two accesses happening in the same actual cache line should be detected by the normal detection mechanism, thus they can lead to actual false sharing problems but not a potential false sharing problem.

A hot access refers to an access that has the number of read or write accesses larger than the average number of accesses. In fact, for every hot access X in a specific cache line L, PREDATOR searches another hot access Y in L's previous cache line or next cache line, satisfying the following conditions:

- X and Y reside on the same virtual line.

- One of X and Y is a write access.

- X and Y are issued by different threads.

Whenever it finds such a pair, X and Y, PREDATOR identifies a potential false sharing problem: they can degrade performance when the number of cache invalidations possibly caused by X and Y (on a possible virtual line), is larger than a pre-defined threshold. This approach is based on a similar observation as in detection: *if a thread writes a virtual line after other threads have accessed the same virtual line, this write operation causes at least one cache invalidation.*

However, before tracking detailed memory accesses on a specific virtual line, it is impossible to know exactly how many cache invalidations actually happen on this virtual line. Thus, PREDATOR conservatively assumes that accesses from different threads occurs in a interleaved way, with the maximum number of cache invalidations. Then PREDATOR starts to track possible cache invalidations on a virtual covering both X and Y, described in Section 5.2.4.

5.2.4 Verifying Potential False Sharing

PREDATOR verifies potential false sharing by tracking possible cache invalidations on a specific virtual line covering such a hot access pair, X and Y.

For potential false sharing caused by double cache line size, as described in Section 5.2.3, a virtual line is always composed of cache line with index $2*i$ and $2*i+1$. PREDATOR tracks cache invalidations on a virtual line that covering X and Y. This virtual line is unique for a given X and Y pair.

However, for the case of changing memory layout, two hot accesses with distance less than the cache line size can actually form multiple virtual lines. There is thus an additional step to determine which virtual line to be tracked. Although a virtual line to be chosen here is never a real cache line of actual hardware because of unaligned addresses, we utilize this virtual line to simulate the effect of changing memory layout correspondingly.

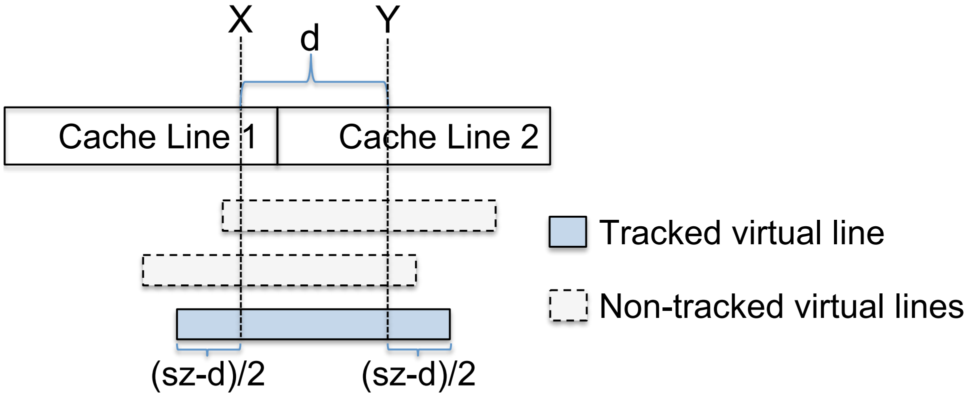


Figure 5.4. Determining a virtual line with size sz according to hot accesses.

Figure 5.4 shows that multiple virtual lines can cover X and Y. However, PREDATOR only chooses one of these virtual lines. PREDATOR chooses the virtual line that leaves the same space before X and after Y. That is, the virtual line starting at location $X - ((sz - d)/2)$ and ending at $Y + ((sz - d)/2)$ is tracked by PREDATOR. This choice allows tracking of more possible cache invalidations caused by adjacent accesses of X and Y. Since adjusting the starting address of a virtual line has the same effect of adjusting the starting address of an object in detecting false sharing, all cache lines related to the same object must be adjusted at the same time. PREDATOR then tracks cache invalidations based on these adjusted virtual lines.

In the end, PREDATOR can report accurately whether the change of memory layout can affect the performance or not, based on the possible number of cache invalidations.

Currently, PREDATOR only determines a specific virtual line to be tracked. However, we plan to extend this in the future work by using a much more flexible mechanism: we can choose a different virtual line after a number of accesses if the current choose cannot reveal a big number of cache invalidations.

5.3 Evaluation

This section answers the following questions:

- How effective is PREDATOR at detecting and predicting false sharing?

- What is PREDATOR's overhead, in terms of execution time and memory ?

- How sensitive is PREDATOR to different sampling rates?

Experimental Platform. All evaluations are performed on a quiescent Intel Core 2 dual-processor system, equipped with 16GB RAM in total. Each processor is a 4-core 64-bit Intel Xeon running at 2.33 GHz, with a 4MB shared L2 cache and 32KB private L1 cache. The underlying operating system is an unmodified CentOS 5.5, running with Linux kernel version 2.6.18-194.17.1.el5. The glibc version is 2.5.

Evaluated Applications. This paper evaluates two popular benchmark suites, Phoenix (with large input) [52] and PARSEC (with simlarge input) [10]. Even with unmodified LLVM-3.2, Facesim cannot be compiled successfully (having complaints on an undefined template) and Canneal aborts unexpectedly. Thus, these two benchmarks are excluded. We also evaluate PREDATOR on six real applications, including MySQL, Boost, Memcached, aget, pbzip2 and pfscan.

5.3.1 Detection and Prediction Effectiveness

```
FALSE SHARING HEAP OBJECT: start 0x40000038 end 0x40000238 (with size 200).
Number of accesses: 5153102690; Number of invalidations: 175020; Number of writes: 13636004.

 Callsite stack:
   ./stddefines.h:53
   ./linear_regression-pthread.c:133

Word level information:
   ......
   Address 0x40000070 (line 16777217): reads 339508  writes 339507 by thread 1
   Address 0x40000080 (line 16777218): reads 2716059 writes 0      by thread 2
   ......
   Address 0x400000b0 (line 16777218): reads 339507  writes 339508 by thread 2
   Address 0x400000c0 (line 16777219): reads 2716061 writes 0      by thread 3
   Address 0x400000c8 (line 16777219): reads 339507  writes 0      by thread 3
```

Figure 5.5. An example reported by PREDATOR, indicating a potential false sharing problem in the linear_regression benchmark.

Benchmark	Source Code	New	Without Prediction	With Prediction	Improvement
histogram	histogram-pthread.c:213	✔	✔	✔	46.22%
linear_regression	linear_regression-pthread.c:133			✔	1206.93%
reverse_index	reverseindex-pthread.c:511		✔	✔	0.09%
word_count	word_count-pthread.c:136		✔	✔	0.14%
streamcluster	streamcluster.cpp:985		✔	✔	7.52%
streamcluster	streamcluster.cpp:1907	✔	✔	✔	4.77%

Table 5.1. False sharing problems in the Phoenix and PARSEC benchmark suites.

For every false sharing problem, Predator reports source code information and detailed memory access information in order to help users fix those problems. Figure 5.5 shows an example for the linear_regression benchmark. This report shows that the heap object starting with $0x40000038$ potentially causes a large number of cache invalidations. The call stack of this memory allocation is provided to help locate culprits. In addition, Predator also reports word-level access information of this object, which helps to identify where and how false sharing occurs. From that, we can know that it is a latent false sharing problem predicted by Predator, since different threads are accessing different cache lines.

5.3.1.1 Benchmarks

We evaluate Predator's effectiveness on two benchmark suites, Phoenix and PARSEC, and Table 5.1 presents those benchmarks with false sharing problems. The first column lists those programs with false sharing problems. The second column shows precisely where the problem is. Because all discovered false sharing occurs inside heap objects, we show the source code information of callsite here. The third column, "New", marks whether this false sharing was newly discovered by Predator. A checkmark in the following two columns indicates whether the false sharing was identified without prediction or with prediction enabled. The final column, "Improvement", shows the performance improvement after fixing false sharing. Note that the performance improvement shown here is different with that in Table 4.3 because Sheriff-Detect evaluates on a 32bit platform and Predator evaluates on a 64bit platform. This also shows that performance effect is every sensitive to hardware platform, which is one of dynamic properties that we discussed above.

As shown in the table, PREDATOR reveals two unknown false sharing problems. It is the first tool to uncover false sharing in histogram and at line 1907 of streamcluster. In histogram, multiple threads simultaneously modify different locations of the same heap object, thread_arg_t. Padding this data structure fixes the false sharing problem and improves the performance by around 46%. In streamcluster, multiple threads are simultaneously accessing and updating the same `bool` array, switch_membership. Simply changing all elements of this array to a long type reduces the false sharing effect, improving performance by about 4.7%.

Other false sharing problems were discovered by previous work [41]. The detailed reason of false sharing problems and how they are fixed are discussed in Section 4.3.1.

It is worth noting that linear_regression has a potential false sharing problem according to the execution environment of PREDATOR. According to the observation of Nanavati et al., this false sharing problem occurs when using clang and disappears when using gcc with the -O2 and -O3 optimization level [46]. But we observed a different result: when we are using the clang-3.2 compiler and our custom memory allocator, the false sharing problem does not occur at all because the offset happens to be 56 bytes (see Figure 5.2). However, it does occur in the original execution environment, with the default memory allocator and using gcc compiler. That is why fixing it improves the performance by more than 12×. This also exemplifies the necessity of PREDATOR predictive detection: existing tools may miss a false sharing problem if it does not occur at their test environments.

5.3.1.2 Real Applications

To verify PREDATOR's practicality, we further evaluate several widely-used real applications, whereas no previous work has done this. These real applications include a standard C++ library (Boost [44]), a server application (MySQL [45]), a distributed memory object caching system (Memcached), a network retriever (aget), a parallel bzip2 file compressor (pbzip2), and a parallel file scanner (pfscan).

MySQL-5.5.32 and boost-1.49.0 are known to have false sharing problems. Other applications (memcached-1.4.15, aget-0.4.1, pbzip2-1.1.6, and pfscan) do not have known false sharing problems.

The false sharing of MySQL has caused a significant scalability problem and was very difficult to be identified. According to the architect of MySQL, Mikael Ronstrom, "we had gathered specialists on InnoDB..., participants from MySQL support... and a number of generic specialists on computer performance...", "[we] were able to improve MySQL performance by 6× with those scalability fixes" [45]. The false sharing inside Boost is caused by the usage of a spinlock pool. Different threads may utilize different spinlocks located in the same cache line in this case. Reducing the number of spinlocks on per cache line to 1 brings a 40% performance improvement. PREDATOR is able to pinpoint false sharing locations in both MySQL and the Boost library. For the other four applications, PREDATOR does not find severe false sharing problems.

5.3.1.3 Prediction Effectiveness

In this section, we verify whether prediction can always reveal un-observed false sharing problems.

The linear_regression benchmark is evaluated here because of the following two reasons: (1) The false sharing problem of this benchmark cannot be detected without prediction; (2) The false sharing problem severely degrades performance when it actually occurs, thus it is a serious problem that should be detected.

Figure 5.6 shows the data structure and the code exercising corresponding false sharing. The size of this data structure, lreg_args, is 64 bytes when the program is compiled to a 64-bit binary using llvm compiler, with optimization level "-O3". In

```
struct
{
  pthread_t tid;  POINT_T *points;
  int num_elems;  long long SX;
  long long SY;   long long SXX;
  long long SYY;  long long SXY;
} 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;
    args->SY+=args->points[i].y;
    args->SYY+=args->points[i].y*args->points[i].y;
    args->SXY+=args->points[i].x*args->points[i].y;
  }
}
```

Figure 5.6. The false sharing problem inside the linear_regression benchmark: multiple threads simultaneously update distinct entries of a global array.

this benchmark, the main thread allocates an array, containing as the same number of elements as hardware cores. Each element is a lreg_args type with 64 bytes. This array is then passed to different threads (lreg_thread function) so that each thread only updates its thread-dependent area. False sharing occurs if two threads happen to update a cache line.

Figure 5.2 shows how sensitive the performance is to different starting addresses of this falsely-shared object. When the offset is 0 or 56 bytes, this benchmark achieves its optimal performance and has no false sharing. When the offset is 24 bytes, the benchmark runs about 15 times slower than its optimal performance because of the false sharing problem.

Our evaluation shows that PREDATOR can always detect the false sharing problem with prediction enabled, when this false sharing object starts with different offsets. This demonstrates the effectiveness of its prediction mechanism.

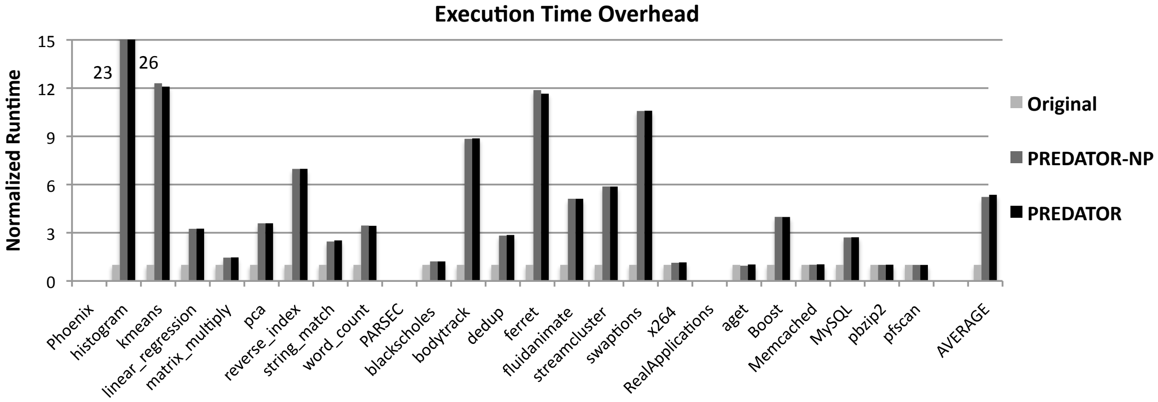


Figure 5.7. Performance overhead of PREDATOR with and without prediction(PREDATOR-NP).

5.3.2 Performance Overhead

We perform evaluations on different benchmarks and application 10 times and show the average of 8 runs in in Figure 5.7. To avoid the effect caused by extreme outliers, the maximum and minimum values are excluded here.

For 16 benchmarks from the Phoenix and PARSEC benchmark suites and six real applications, PREDATOR imposes 5.4× performance overhead averagely. There is no noticeable difference on performance whether the prediction mechanism is enabled or not.

Among five of these programs, histogram, kmeans, bodytrack, ferret, and swaptions, PREDATOR introduces more than 8× performance overhead. The histogram benchmark runs more than 26× slower than the one with `pthreads`, because tracking detailed access on cache lines with false sharing exacerbates the false sharing effect (see more discussion in Section 5.1.4.3). For bodytrack and ferret, although there is no false sharing, PREDATOR detects a large amount of cache lines with writes larger than *Tracking-Threshold*. Thus, tracking those accessing details for those cache lines imposes significant performance overhead. Currently, we have not identified the exact cause of PREDATOR's high performance for kmeans.

PREDATOR imposes a small performance overhead for IO-bound applications, such as matrix_multiply, blackscholes, x264, aget, Memcached, pbzip2, and pfscan, since PREDATOR does not add any performance overhead for IO operations.

5.3.3 Memory Overhead

We evaluate physical memory overhead of PREDATOR, instead of virtual memory overhead, because PREDATOR allocates 4GB virtual memory for its custom memory allocator beforehand. Proportional set size (PSS) of a specific memory mapping (in `/proc/self/smaps`) reflects the physical memory increase because of running the current application [34]. Thus, we periodically collect this data and use the sum of different memory mappings as the total physical memory usage of running an application. Figure 5.9 presents the normalized physical memory usage of running different applications, comparing to that using `pthreads`.

PREDATOR imposes less than 50% memory overhead for 17 out of 22 applications. For swaptions and aget, PREDATOR introduces more memory overhead because the original memory footprints of them are very small, only 3 kilobytes. Adding the code of detection, prediction and reporting (constant overhead) contributes to a large ratio of memory overhead. The increase of memory consumption in MySQL, from 132 MB to 512 MB, is due to PREDATOR's heap organization, which does not aggressively reclaim memory held by individual threads. In all cases where PREDATOR's imposes substantial memory overhead, the applications continue to comfortably fit into RAM on modern platforms.

5.3.4 Sampling Rate Sensitivity

Section 5.1.4.3 describes PREDATOR's sampling mechanism to reduce tracking overhead. This section evaluates the effect of different sampling rates on performance and effectiveness. Note that running an application with different sampling rates does not affect its memory usage, thus memory overhead is not examined here.

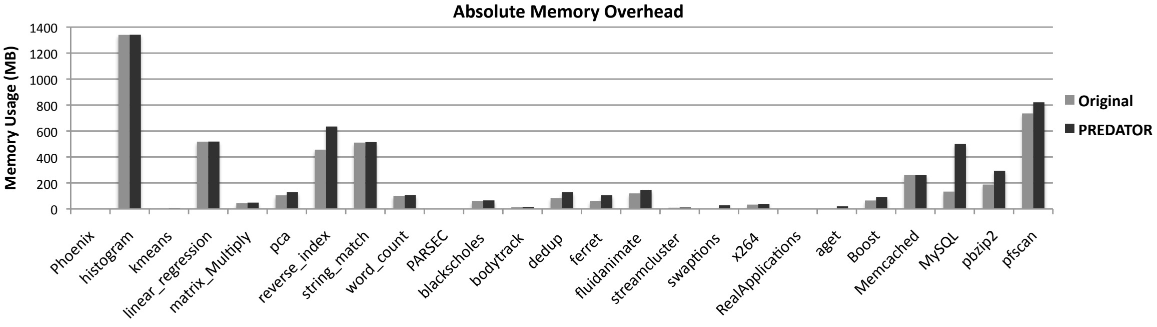


Figure 5.8. Absolute physical memory usage overhead with Predator.

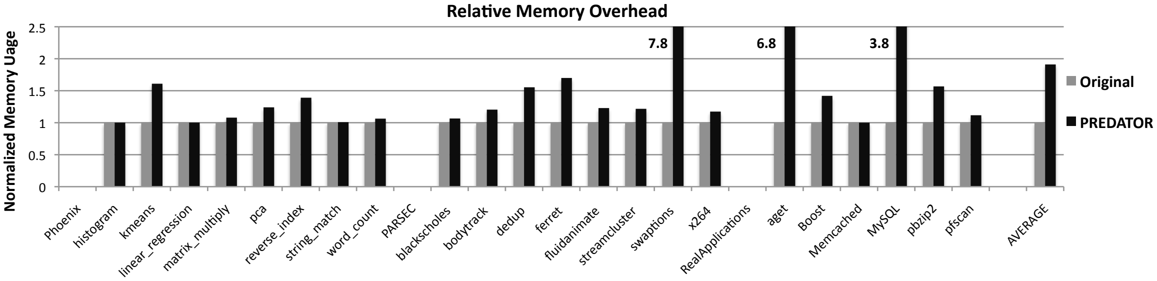


Figure 5.9. Relative physical memory usage overhead with Predator.

The default sampling rate used by Predator is 1%. In this section, we also evaluate two other sampling rates, 0.1% and 10%. Figure 5.10 presents performance results under the three different sample rates. We only show the results of those programs having false sharing problems inside, since only their performance are most likely to be affected by different sampling rates. As expected, Predator introduces less performance overhead under a lower sampling rate, but with a very minor performance impact. About effectiveness, even when using the 0.1% sampling rate, Predator can still detect all false sharing problems, although it reports a lower number of cache invalidations. Thus, different sampling rates do not affect the detection effectiveness.

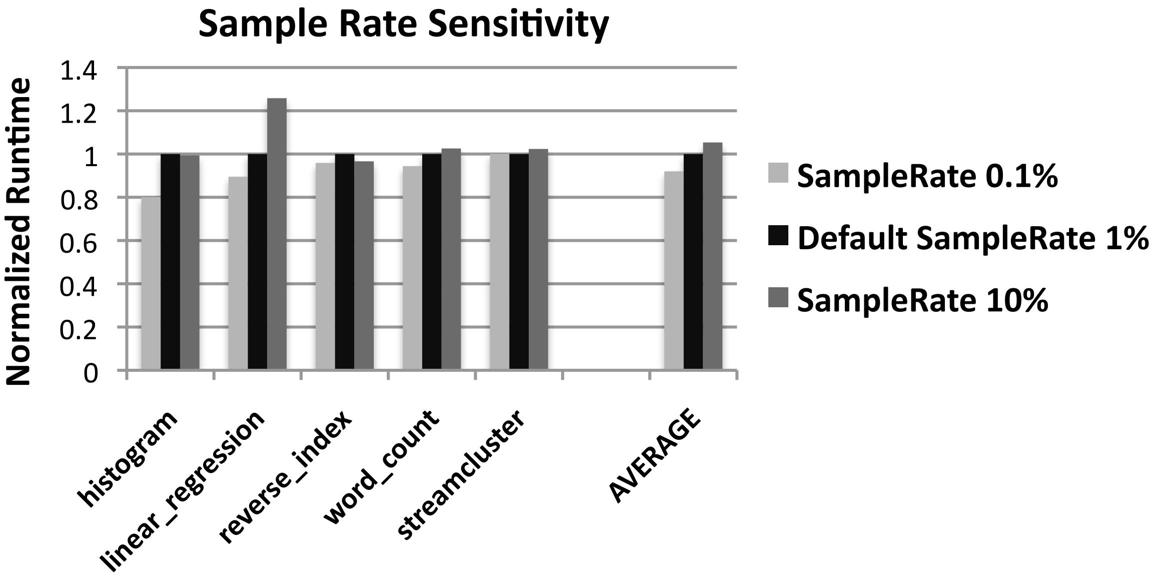


Figure 5.10. Sampling rate sensitivity (execution time).

5.4 Discussion

5.4.1 Instrumentation Selection

Dynamic binary instrumentation and compiler-based instrumentation are two alternative approaches to perform instrumentation [30]. They exhibit different tradeoffs of performance and generality. Dynamic binary instrumentors, such as Valgrind [47], Pin [43], and DynamoRIO [15], typically analyze the program's code just before execution in order to insert instrumentation. They introduce significant performance overhead, mostly caused by run-time encoding and decoding, but the fact that they operate directly on binaries makes them extremely convenient. By contrast, compiler instrumentation inserts instrumentation in the compilation phase, which requires recompilation of all source code. Predator employs compiler-based instrumentation both because of its better performance and its greater flexibility, as discussed in Section 5.1.4.2.

5.4.2 Effectiveness

Several factors can affect Predator's ability to identify false sharing.

Different Inputs. Different inputs trigger distinct executions of a program. If a specific input does not exercise the code with false sharing problems, PREDATOR cannot necessarily detect them. However, PREDATOR does generalize over inputs to find latent false sharing problems on those exercised code. When any reasonably representative set of inputs are exercised, as is required by any testing regime, PREDATOR can effectively predict false sharing.

Input Size. Input size may affect detection results. As discussed in Section 5.1.4, PREDATOR introduces several threshold values to reduce tracking overhead, which can be adjusted as needed. If the input size is so small that it cannot generate enough false sharing events to cross the pre-defined thresholds, then the detection mechanism will not be triggered. In such cases, PREDATOR will miss actual cases of false sharing. However, realistically large inputs should be enough to trigger PREDATOR's detection mechanisms.

Hardware Independence. PREDATOR's compiler-based approach make it independent of the underlying hardware platform. This approach increases generality, but may lead to over-report false sharing. PREDATOR conservatively assumes that different threads are running on different cores and detects false sharing problems based on possible cache invalidations. However, if multiple threads involved in false sharing are on the same core, then there will be no performance impact.

CHAPTER 6

RELATED WORK

This chapter first describes those related work to processes-as-threads framework and deterministic execution. Then it describes related work in false sharing detection, prevention, or both.

6.1 Processes-As-Threads framework

BOP relies on strong isolation of processes to automatically and safely parallelize the execution of programs [24]. BOP forks a new process to do speculation, based on those pre-defined possibly parallel regions (PPR). In order to check the correctness, BOP tracks accesses on a page-based granularity. When there is no conflict and a speculative process reaches the end of its current PPR, its predecessor always commits its changes to the current process. However, BOP does not provide any synchronization support and cannot be used to run normal multithreaded programs.

Grace is a process-based approach designed to prevent concurrency errors, such as deadlock, race conditions, and atomicity errors by imposing a sequential semantics on speculatively-executed threads [7]. Grace supports only fork-join programs without inter-thread communication (e.g., condition variables or barriers), and rolls back threads when accesses of threads would violate sequential semantics: a thread accesses pages that have been accessed by its predecessors. Grace cannot support arbitrary multithreaded programs. Similar to the Grace system, Sammati is a processes-as-threads system to detect and tolerate deadlock problems [51]. However, Sammati does not support the full range of synchronizations, without the support of conditional variables, barriers, and signals. Also, Semmati cannot avoid race conditions happening in creating twin pages, which are avoided by the SHERIFF framework.

6.2 Deterministic Multithreading

The research on deterministic multithreading is a very active area these years. We describe some software-only, non- language-based approaches here.

Grace prevents deadlocks, race conditions, ordering and atomicity violations errors for those fork-join multithreaded programs by imposing a sequential semantics at join points [7]. However, Grace does not support programs with inter-thread communications, such as conditional variables and barriers.

CoreDet is a compiler-based approach to support general-purpose multithreaded programs [4]. CoreDet instruments those memory read and write operations as long as those operations cannot be proved to be thread-local in static analysis. In the runtime phase, CoreDet divides the execution into alternating parallel and serial phases and guides all memory operations using a memory ownership table: only those owned locations can be accessed in the parallel phases; all non-owned locations and synchronizations can only be accessed in the serial phases guided by a global token. CoreDet guarantees deterministic execution for racy programs without memory errors, but with very high performance overhead: averagely 3.5× slower than those using `pthreads`. In order to guarantee determinism, CoreDet has to serialize *all* external library calls without instrumentation. CoreDet does not provide deterministic memory allocations, which can not guarantee determinism for programs with memory errors. dOS [5] is an extension to CoreDet that uses the same deterministic scheduling framework. dOS supports deterministic communication for those threads and processes inside the same deterministic process groups (DPGs) and handle those external non-determinism by recording and replaying interactions across DPG boundaries.

Determinator is a microkernel-based operating system that enforces system-wide determinism [2]. Determinator provides separate address spaces and supports interprocess communications at explicit synchronization points. Determinator is a proof-of-concept system, which can not support the whole rage of threads APIs and can not work on legacy programs.

Some other works can only support limited determinism or need user annotation. Kendo can only guarantee the determinism for race-free programs [48]. TERN [23] provides a best-effort system to apply memoized schedules for future runs with similar inputs. It can not guarantee the determinism for racy programs, as Kendo. Peregrine [22] is a system based on TERN, which tries to record the order of memory accesses for racy portions and apply those schedules for future runs possibly. However, both TERN and Peregrine do not support complete determinism (using a best effort) and requires program annotations.

6.3 False Sharing

This section describes related work in false sharing detection, prevention, or both. There is no previous system to predict unobserved false sharing.

6.3.1 False Sharing Detection

Based on the SIMICS functional simulator, Schindewolf et al. designed a tool to report different kinds of cache usage information, such as cache misses and cache invalidations [53]. Pluto relies on Valgrind dynamic instrumentation framework to track the sequence of memory read and write events on different threads, and reports a worst-case estimation of possible false sharing [26]. Similarly, Liu uses Pin to collect memory access information, and reports total cache miss information [40]. These tools impose about $100 - 200\times$ performance overhead.

Zhao et al. developed a tool based on DynamoRIO framework to detect false sharing and other cache contention problems for multithreaded programs [58]. It uses a shadow memory technique to maintain memory access history and detects cache invalidations based on the ownership of cache lines. However, it can only support at most 8 threads currently and it is hard to scale up, because of its per-bit-each-thread bitmap design. In addition, it cannot differentiate cold cache misses from actual false sharing problems.

Intel's performance tuning utility (PTU) uses Precise Event Based Sampling (PEBS) hardware support to detect false sharing problems [28, 29]. PTU cannot distinguish true sharing from false sharing. In addition, PTU aggregates memory accesses without considering memory reuses and access interleaving, leading to numerous false positives. Sanath et al. designed a machine learning based approach to detect false sharing problems. They train their classifier on mini-programs and apply this classifier to general programs [32]. Instead of instrumenting memory accesses, this tool relies on hardware performance counters to collect memory accesses events. It achieves very low performance overhead (about 2%). But it relies on hardware support for its efficiency. Also, it cannot detect a lot of actual false sharing problems that can greatly affect performance, such as histogram and streamcluster. We guess that this incompleteness can be caused by their problematic training method or hardware's sampling mechanism, but the specific reason is not clear to us.

In addition to their individual disadvantages, all approaches discussed above share two common shortcomings: They cannot pinpoint the exact location of false sharing in the source code, so programmers have to examine the source code and identify problems manually; they can only detect those observed false sharing problems.

Pesterev et al. present DProf, a tool that help programmers identify cache misses based on AMD's instruction-based sampling hardware [49]. DProf requires manual annotation to locate data types and object fields, and cannot detect false sharing when multiple objects reside on the same cache line.

6.3.2 False Sharing Prevention

Jeremiassen and Eggers use a compiler transformation to automatically adjust the memory layout of applications through padding and alignment [33]. Chow et al. alter parallel loop scheduling in order to avoid false sharing [21]. These approaches only works for regular, array-based scientific code.

Berger et al. describe Hoard, a scalable memory allocator that can reduce the possibility of false sharing by making different threads use different heaps [6]. Hoard cannot avoid false sharing problem in global variables or within a single heap object: the latter appears to be the primary source of real false sharing problems.

6.3.3 False Sharing Detection and Prevention

Plastic leverages the sub-page granularity memory remapping facility provided by the Xen hypervisor to detect and tolerate false sharing automatically [46]. However, the sub-page memory remapping mechanism is not currently supported by most existing operating system, reducing its generality. In addition, Plastic cannot pinpoint the exact source of false sharing. In order to utilize Plastic's prevention tool, a program has to run on the Xen hypervisor, limiting the applicability of their prevention technique.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Because of hard physical limits, computer manufacturers have turned to providing more and more cores on a single machine. This phenomenon drives the biggest revolution of software development: software has to be programmed in a concurrent and parallel way in order to exploit the benefits of multi-core machines.

Building efficient and reliable concurrent software is still a challenging task. First, concurrency requires programmers to think in an unnatural way that humans find difficult. Second, existing languages and tools are inadequate to detect or prevent concurrency errors.

7.1 Contributions

This thesis helps boost the performance and ease the reasoning and debugging, by providing different tools and runtime systems. We present a novel processes-as-threads replacement library, the SHERIFF framework, which providing per-thread memory protection and isolation on the page granularity. First, based on this framework, we provide DTHREADS to ensure deterministic execution of multithreaded programs, even with race conditions. DTHREADS outperforms the previous state-of-the-art runtime system (CoreDet) by a factor of 3, and is the new basis of all later deterministic multithreading systems. Second, we presents two tools based on the SHERIFF framework, SHERIFF-DETECT and SHERIFF-PROTECT, to deal with false sharing problems of multithreaded programs, one of the notorious performance problems. SHERIFF-DETECT is the first tool to correctly and precisely identify false sharing problems inside parallel applications. SHERIFF-PROTECT is the first generalized system to automatically mitigate false sharing problems, without the need of programmer intervention. Finally, we present another tool, PREDATOR, to improve the effectiveness by revealing read-write false sharing problems and overcome a generalized issue of false sharing detection: Existing tools can only detect those observed false sharing problems; PREDATOR can predict potential false sharing that does not manifest in a given execution but may appear—and greatly degrade application performance–in a slightly different execution environment. PREDATOR is the first false sharing tool that is able to automatically and precisely uncover false sharing problems in real applications, including MySQL and the Boost library.

7.2 Future Work

DTHREADS performs synchronizations inside serial phases, which is susceptible to delays due to load imbalance between threads. To handle this problem, one direction of future work is to reduce the waiting time caused by load imbalance problem. We observed that the overhead of DTHREADS depends on the number of synchronizations: with less synchronizations, DTHREADS can achieve much better performance since it can amortize the overhead better. Another direction of future work is to design programs with DTHREADS's mechanism in mind, by extending a set of APIs, so that users can design programs with less load imbalance problem and less synchronizations. Thus, we could possibly achieve better performance.

This thesis also presents a set of tools to detect false sharing problems inside multithreaded programs. But false sharing problems can exist in the entire software stack, including hypervisors, operating systems, and applications using different threading libraries or other languages. In the future, we would like to extend the detection mechanism, coming from PREDATOR, to the entire software stack. Also, we can leverage memory trace information to suggest fixes, in order to help programmers to eliminate false sharing.

Sheriff-Protect introduces some performance overhead when a parallel program does not have false sharing problem inside. It is helpful if this protection mechanism can leverage the output of detection: we only use this mechanism to boost the performance if an application has some false sharing problems inside; further, we can employ isolation on specific objects in order to further reduce performance overhead.

BIBLIOGRAPHY

[1] Abdelrahman, Cedomir Segulja Tarek S. False t is the cost of determinism? In *5th Workshop on Determinism and Correctness in Parallel Programming* (2014), WoDet'14.

[2] Aviram, Amittai, Weng, Shu-Chan, Hu, Sen, and Ford, Bryan. Efficient system-enforced deterministic parallelism. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2010), USENIX Association, pp. 193–206.

[3] Ball, Thomas, Burckhardt, Sebastian, de Halleux, Jonathan, Musuvathi, Madanlal, and Qadeer, Shaz. Deconstructing concurrency heisenbugs. In *ICSE Companion* (2009), IEEE, pp. 403–404.

[4] Bergan, Tom, Anderson, Owen, Devietti, Joseph, Ceze, Luis, and Grossman, Dan. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 53–64.

[5] Bergan, Tom, Hunt, Nicholas, Ceze, Luis, and Gribble, Steven D. Deterministic process groups in dOS. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2010), USENIX Association, pp. 177–192.

[6] Berger, Emery D. The Hoard memory allocator. Available at `http://www.hoard.org`.

[7] Berger, Emery D., Yang, Ting, Liu, Tongping, and Novark, Gene. Grace: safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 81–96.

[8] Berger, Emery D., and Zorn, Benjamin G. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2006), ACM Press, pp. 158–168.

[9] Berger, Emery D., Zorn, Benjamin G., and McKinley, Kathryn S. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 114–124.

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

[11] Bocchino, Jr., Robert L., Adve, Vikram S., Dig, Danny, Adve, Sarita V., Heumann, Stephen, Komuravelli, Rakesh, Overbey, Jeffrey, Simmons, Patrick, Sung, Hyojin, and Vakilian, Mohsen. A type and effect system for deterministic parallel Java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 97–116.

[12] Bolosky, William J., and Scott, Michael L. False sharing and its effect on shared memory performance. In *SEDMS IV: USENIX Symposium on Experiences with Distributed and Multiprocessor Systems* (Berkeley, CA, USA, 1993), USENIX Association, pp. 57–71.

[13] Bolosky, William J., and Scott, Michael L. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (Berkeley, CA, USA, 1993), Sedms'93, USENIX Association, pp. 3–3.

[14] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM Press, pp. 1–11.

[15] Bruening, Derek, Garnett, Timothy, and Amarasinghe, Saman. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2003), CGO '03, IEEE Computer Society, pp. 265–275.

[16] Burckhardt, Sebastian, Baldassin, Alexandro, and Leijen, Daan. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 691–707.

[17] Burckhardt, Sebastian, Kothari, Pravesh, Musuvathi, Madanlal, and Nagarakatte, Santosh. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS* (New York, NY, USA, 2010), James C. Hoe and Vikram S. Adve, Eds., ASPLOS '10, ACM, pp. 167–178.

[18] Carter, John B., Bennett, John K., and Zwaenepoel, Willy. Implementation and performance of Munin. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1991), ACM, pp. 152–164.

[19] Carver, Richard H., and Tai, Kuo-Chung. Replay and testing for concurrent programs. *IEEE Softw. 8* (March 1991), 66–74.

[20] Choi, Jong-Deok, and Srinivasan, Harini. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (New York, NY, USA, 1998), SPDT '98, ACM, pp. 48–59.

[21] Chow, Jyh-Herng, and Sarkar, Vivek. False sharing elimination by selection of runtime scheduling parameters. In *ICPP '97: Proceedings of the international Conference on Parallel Processing* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 396–403.

[22] Cui, Heming, Wu, Jingyue, Gallagher, John, Guo, Huayang, and Yang, Junfeng. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)* (October 2011).

[23] Cui, Heming, Wu, Jingyue, Tsa, Chia-che, and Yang, Junfeng. Stable deterministic multithreaded through schedule memoization. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2010), USENIX Association, pp. 207–222.

[24] Ding, Chen, Shen, Xipeng, Kelsey, Kirk, Tice, Chris, Huang, Ruke, and Zhang, Chengliang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 223–234.

[25] Dubois, Michel, Wang, Jin Chin, Barroso, Luiz A., Lee, Kangwoo, and Chen, Yung-Syau. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1991), Supercomputing '91, ACM, pp. 197–206.

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

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

[28] Intel Corporation. Intel performance tuning utility 3.2 update. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility`, November 2008.

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

[30] Iskhodzhanov, Timur, Kleckner, Reid, and Stepanov, Evgeniy. Combining compile-time and run-time instrumentation for testing tools. *Programmnye produkty i sistemy 3* (2013), 224–231.

[31] ISO. Programming languages – c++. Available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`.

[32] Jayasena, Sanath, Amarasinghe, Saman, Abeyweera, Asanka, Amarasinghe, Gayashan, De Silva, Himeshi, Rathnayake, Sunimal, Meng, Xiaoqiao, and Liu, Yanbin. Detection of false sharing using machine learning. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 30:1–30:9.

[33] Jeremiassen, Tor E., and Eggers, Susan J. 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* (New York, NY, USA, 1995), ACM, pp. 179–188.

[34] Justin L. A way to determine a process's "real" memory usage, i.e. private dirty rss? `http://stackoverflow.com/questions/118307/a-way-to-determine-a-processs-real-memory-usage-i-e-private-dirty-rss`, October 2011.

[35] Keleher, Pete, Cox, Alan L., Dwarkadas, Sandhya, and Zwaenepoel, Willy. Treadmarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 10–10.

[36] Larus, James, and Rajwar, Ravi. *Transactional Memory (Synthesis Lectures on Computer Architecture)*, first ed. Morgan & Claypool Publishers, 2007.

[37] Lattner, Chris, and Adve, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[38] Lattner, Chris, and Adve, Vikram. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[39] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36* (April 1987), 471–482.

[40] Liu, Chien-Lung. False sharing analysis for multithreaded programs. Master's thesis, National Chung Cheng University, July 2009.

[41] Liu, Tongping, and Berger, Emery D. SHERIFF: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 3–18.

[42] Lu, Shan, Park, Soyeon, Seo, Eunsoo, and Zhou, Yuanyuan. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 329–339.

[43] Luk, Chi-Keung, Cohn, Robert, Muth, Robert, Patil, Harish, Klauser, Artur, Lowney, Geoff, Wallace, Steven, Reddi, Vijay Janapa, and Hazelwood, Kim. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[44] mcmcc. False sharing in boost::detail::spinlock pool? `http://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool`, June 2012.

[45] Mikael Ronstrom. Mysql team increases scalability by ¿50mysql 5.6 labs release april 2012. `http://mikaelronstrom.blogspot.com/2012/04/mysql-team-increases-scalability-by-50.html`, April 2012.

[46] Nanavati, Mihir, Spear, Mark, Taylor, Nathan, Rajagopalan, Shriram, Meyer, Dutch T., Aiello, William, and Warfield, Andrew. Whose cache line is it anyway?: operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 141–154.

[47] Nethercote, Nicholas, and Seward, Julian. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[48] Olszewski, Marek, Ansel, Jason, and Amarasinghe, Saman. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ACM, pp. 97–108.

[49] Pesterev, Aleksey, Zeldovich, Nickolai, and Morris, Robert T. Locating cache performance bottlenecks using data profiling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), ACM, pp. 335–348.

[50] Pool, Jesse, Sin, Ian, and Lie, David. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS 2007)* (May 2007).

[51] Pyla, Hari K., and Varadarajan, Srinidhi. Avoiding deadlock avoidance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 75–86.

[52] Ranger, Colby, Raghuraman, Ramanan, Penmetsa, Arun, Bradski, Gary, and Kozyrakis, Christos. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 13–24.

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

[54] Serebryany, Konstantin, Bruening, Derek, Potapenko, Alexander, and Vyukov, Dmitry. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 28–28.

[55] Simpson, David J., and Burton, F. Warren. Space efficient execution of deterministic parallel programs. *IEEE Trans. Softw. Eng. 25* (November 1999), 870–882.

[56] Stevens, W. Richard, and Rago, Stephen A. *Advanced Programming in the UNIX Environment: Second Edition.* Addison Wesley Professional, 2005.

[57] Xiong, Weiwei, Park, Soyeon, Zhang, Jiaqi, Zhou, Yuanyuan, and Ma, Zhiqiang. Ad hoc synchronization considered harmful. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2010), USENIX Association, pp. 163–176.

[58] Zhao, Qin, Koh, David, Raza, Syed, Bruening, Derek, Wong, Weng-Fai, and Amarasinghe, Saman. Dynamic cache contention detection in multi-threaded applications. In *The International Conference on Virtual Execution Environments* (Newport Beach, CA, Mar 2011).