3-14-2016

# LASER: Light, Accurate Sharing dEtection and Repair

Liang Luo
*University of Pennsylvania*

Akshitha Sriraman
*University of Pennsylvania*

Brooke Fugate
*University of Pennsylvania*, bfugate@cis.upenn.edu

Shiliang Hu

Gilles Pokam

***See next page for additional authors***

## Recommended Citation

Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J. Newburn, and Joseph Devietti, "LASER: Light, Accurate Sharing dEtection and Repair", *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* , 261-273. March 2016. http://dx.doi.org/10.1109/HPCA.2016.7446070

2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) was held March 12-16, 2016, in Barcelona, Spain.

# LASER: Light, Accurate Sharing dEtection and Repair

**Abstract**

Contention for shared memory, in the forms of true sharing and false sharing, is a challenging performance bug to discover and to repair. Understanding cache contention requires global knowledge of the program's actual sharing behavior, and can even arise invisibly in the program due to the opaque decisions of the memory allocator. Previous schemes have focused only on false sharing, and impose significant performance penalties or require non-trivial alterations to the operating system or runtime system environment.

This paper presents the Light, Accurate Sharing dEtection and Repair (LASER) system, which leverages new performance counter capabilities available on Intel's Haswell architecture that identify the source of expensive cache coherence events. Using records of these events generated by the hardware, we build a system for online contention detection and repair that operates with low performance overhead and does not require any invasive program, compiler or operating system changes. Our experiments show that LASER imposes just 2% average runtime overhead on the Phoenix, Parsec and Splash2x benchmarks. LASER can automatically improve the performance of programs by up to 19% on commodity hardware.

**Keywords**

false sharing, true sharing, cache contention, performance counters

**Disciplines**

Computer and Systems Architecture | Computer Engineering | Computer Sciences

**Comments**

2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) was held March 12-16, 2016, in Barcelona, Spain.

**Author(s)**

Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J. Newburn, and Joseph Devietti

# LASER: Light, Accurate Sharing dEtection and Repair

Liang Luo*, Akshitha Sriraman§, Brooke Fugate‡
Shiliang Hu†, Gilles Pokam†, Chris J. Newburn†, Joseph Devietti‡
*University of Washington, §University of Michigan, ‡University of Pennsylvania, †Intel Corporation
liangluo@cs.washington.edu akshitha@umich.edu bfugate@cis.upenn.edu
{shiliang.hu, gilles.a.pokam, chris.newburn}@intel.com devietti@cis.upenn.edu

## ABSTRACT

Contention for shared memory, in the forms of true sharing and false sharing, is a challenging performance bug to discover and to repair. Understanding cache contention requires global knowledge of the program's actual sharing behavior, and can even arise invisibly in the program due to the opaque decisions of the memory allocator. Previous schemes have focused only on false sharing, and impose significant performance penalties or require non-trivial alterations to the operating system or runtime system environment.

This paper presents the Light, Accurate Sharing dEtection and Repair (LASER) system, which leverages new performance counter capabilities available on Intel's Haswell architecture that identify the source of expensive cache coherence events. Using records of these events generated by the hardware, we build a system for online contention detection and repair that operates with low performance overhead and does not require any invasive program, compiler or operating system changes. Our experiments show that LASER imposes just 2% average runtime overhead on the Phoenix, Parsec and Splash2x benchmarks. LASER can automatically improve the performance of programs by up to 19% on commodity hardware.

## 1. INTRODUCTION

Multicore architectures continue to pervade every part of our computing infrastructure, from servers to phones and smart watches. While these parallel architectures bring established performance and energy efficiency gains compared to single-core designs, parallel code written for these architectures can suffer from subtle performance bugs that are difficult to understand and repair with current profiling tools. Chief among these performance issues are sources of *cache contention* such as false sharing and extraneous true sharing.

Cache contention bugs can be found in a wide range of parallel software, from benchmark suites like Phoenix [28] and Parsec [2] to production-quality code like the Boost libraries [26], MySQL [29] and the Linux kernel [4, 6, 7]. Cache contention is difficult to diagnose because it is often only implicitly represented in the program source code. An otherwise-performant piece of code can run very slowly due to remote accesses to the same data or, in the case of false sharing, an entirely different piece of data that happens to reside in the same cache line. Cache contention exacts a significant time and energy cost for generating and processing cache coherence traffic.

Uncovering and repairing cache contention in real systems requires low overhead techniques that are compatible with existing software infrastructure, so as to minimize perturbation to the system in pursuit of contention. Online repair of false sharing is particularly valuable because it does not require programmer intervention, access to source code, or system downtime. Previous work has employed a variety of techniques for online detection and repair that, unfortunately, face performance and compatibility challenges. For example, Sheriff [18] executes threads as processes, leading to large slowdowns for programs with frequent synchronization operations; Predator [19] uses intensive compiler instrumentation which results in large performance overheads; and the Plastic system [27] requires custom OS or hypervisor support. These previous proposals also focus exclusively on false sharing detection and repair, missing opportunities to detect true sharing that can equally sap performance.

To support low performance overheads and high compatibility simultaneously, we propose LASER: Light, Accurate Sharing dEtection and Repair. LASER leverages new approaches to contention detection and repair. The LASERDETECT **detection** approach is founded on a new performance counter feature made available in Intel's Haswell architecture. We use the chip's Precise Event-Based Sampling (PEBS) mechanism which provides records of key cache coherence events. LASERDETECT relies on records of *HITM events* (Section 2) which arise when a core performs a memory access that hits in a remote cache line, where the remote line is in Modified state. These HITM coherence events are the root cause of cache contention; once the contention is removed these expensive HITM events are replaced with much faster local cache hits. LASERDETECT receives a stream of HITM records from the chip's performance monitoring unit, and processes these events to discover the code and data involved in contention, and whether the contention is of sufficient intensity to merit programmer attention or automatic

*§Work done while Luo and Sriraman were students at the University of Pennsylvania.

*HPCA '16*, March 12-16, 2016, Barcelona, Spain.
978-1-4673-9211-2/16/$31.00 ©2016 IEEE

repair. We demonstrate that the reported locations correctly identify real performance bugs. The HITM mechanism we use in this work can also serve as an efficient underpinning for identifying inter-thread communication patterns, an important component of tools for program understanding [34], data race detection [13], and concurrency bug detection [21, 22]. Because of LASER's lightweight approach, we expect it to be amenable to integration with a wide variety of concurrency analyses. While we leverage Intel platforms in this work, our system could also be ported to take advantage of the Instruction-Based Sampling mechanism [9] available on recent AMD processors.

The LASERREPAIR **repair** scheme leverages LASERDETECT to find and modify the key parts of a program that trigger contention. The repair mechanism implements a software-based store buffer mechanism using dynamic binary rewriting. Like a hardware-based store buffer, our software store buffer mechanism defers the cost of cache coherence (and thus, contention) to improve performance. LASERREPAIR complies with the TSO consistency model, unlike other recent repair approaches (Section 5), allowing LASER to repair existing x86 binaries. LASERREPAIR also leverages Haswell's hardware transactional memory to improve the space efficiency of our software store buffer.

This paper makes the following contributions:

- The LASERDETECT contention detection system, which detects cache contention on Intel's Haswell architecture with virtually no performance overhead and no invasive changes to the compiler, OS or runtime system environment.

- The LASERREPAIR contention repair system, which repairs false sharing online via dynamic binary instrumentation. LASERREPAIR preserves the TSO consistency model and focuses repair just on the code that needs it.

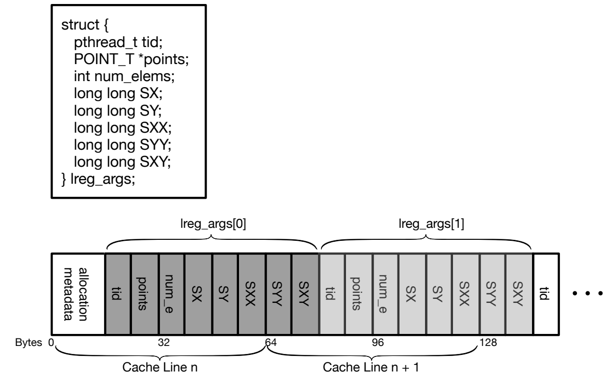
- The first detailed characterization of the accuracy and limitations of the HITM event recording mechanism on our Haswell machine, which can help others looking to make use of these performance counters. We also identify ways to compensate for the limitations of the currently-available performance counter mechanism.

- Detailed case studies of the cache contention behavior of the Phoenix, Parsec and Splash2x benchmark suites.

The remainder of this paper is organized as follows. Section 2 provides background on cache contention in multiprocessor systems. Section 3 explains and characterizes Haswell's hardware support for detecting cache contention. Section 4 details the LASERDETECT algorithm, Section 5 the LASERREPAIR algorithm, and Section 6 the architecture of the overall LASER system. Section 7 evaluates LASER's detection accuracy and performance overheads. Section 8 describes related work, and we conclude in Section 9.

## 2. BACKGROUND: CACHE CONTENTION

Common cache coherence protocols require that any given cache line generally be either in a writable state that permits a single mutable copy, a read-only state that permits multiple shared copies, or an invalid state. While sophisticated protocols with more states are commonplace, these three states of Modified, Shared and Invalid are the essential components of all protocols. Transitioning a line into and out of the Modified state is a slow process that requires expensive coordination among all processors with existing copies of the line. An excessive number of such transitions leads to cache contention, and can be triggered by frequent repetition of one of three sharing patterns: write-read, read-write, or write-write (Figure 1). Read-read sharing does not introduce contention as each core can obtain its own read-only copy of the line. Write-read sharing typically manifests as read-write sharing (and vice-versa) since the interleaving of instructions is nondeterministic.



```
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;
```

**Figure 2: False sharing in the linear_regression benchmark shown through the allocation of the lreg_args array on a 64-byte cache line. Figure from [27].**

Coherence protocols operate at cache line granularity, typically 64 bytes in modern systems, instead of the granularity of a single program variable. The sharing patterns in Figure 1 can trigger contention whenever they arise on a particular cache line. Contention for a cache line may be the result of contention over distinct variables that happen to reside in the same cache line, or over a single program variable. These two sub-cases are *false sharing* and *true sharing*.

With false sharing, two distinct program variables end up allocated in the same cache line with the line subject to one or more of the contention sharing patterns from Figure 1: each variable is frequently accessed by distinct threads and at least one variable is frequently written. Because a cache line must be in just one state at any given time, the cache line constantly undergoes expensive and serialized state transitions for what are logically parallel program accesses. The linear_regression benchmark from the Phoenix benchmark suite [28] exhibits extensive false sharing on the lreg_args struct. An array of these structs is allocated with one struct per thread. Each thread frequently increments the fields SY through SXY. On our 64-bit platform lreg_args is 64 bytes in size. This would seem to mesh well with our system's 64-byte cache lines, but depending on the allocator's choice of memory layout a cache line can still contain structs for two threads, causing frequent writes to distinct parts of the cache line and intense false sharing, as shown in Figure 2.

Padding is a general solution to false sharing as it spaces

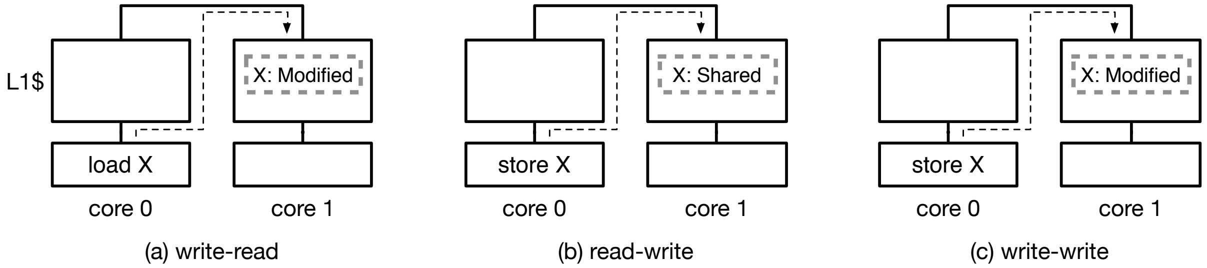


**Figure 1: The three sharing patterns that trigger cache contention: (a) remote write followed by a local read, (b) remote read followed by a local write, and (c) remote write followed by a local write. (a) and (c) are instances of HITM events, but only (a) reliably triggers accurate HITM records on Haswell.**

hot variables in memory so they reside in different cache lines, allowing cheap parallel access. For linear_regression, 52 bytes of padding is sufficient to eliminate false sharing regardless of how the allocator aligns the lreg_args structs.

In cases of true sharing, a single program variable is subject to frequent accesses by multiple threads where at least one thread writes to the variable. Padding cannot fix true sharing; instead the program must be rewritten to exhibit less true sharing. For example, in the kmeans benchmark from Phoenix [28], threads repeatedly and redundantly set the global modified flag to true. The code can be rewritten to cache the value of modified in a stack variable and perform just a single write to the global flag. Other instances of true sharing include naively-written spin locks that involve a single atomic compare-and-swap in a loop. Such locks can perform poorly when lots of threads attempt to acquire a lock that is held [1]. A better design is the test-and-test-and-set lock which allows potential acquirers to check the lock without trying to update it, allowing the lock state to be read-shared across processors.

# 3. HITM EVENTS ON HASWELL

In both true and false sharing, the ultimate performance culprit is one or more contention sharing patterns and the expensive coherence protocol transitions they require. Recent Intel processors provide information about HITM (as in hit-Modified) coherence events which arise whenever a local memory operation accesses a line that is in the Modified state in a remote cache as in Figure 1 parts a) and c). Because write-read sharing frequently manifests as read-write sharing as well, HITM events provide good coverage of the contentious sharing in a program. Programs without cache contention will have very few HITM events, and frequent HITM events suggest the presence of significant cache contention.

Intel processors since Nehalem have provided the ability to count the number of HITM events occurring on each core. Researchers have made use of this ability to detect false sharing [27] and data races [13]. Starting with Haswell, the performance counter mechanism can be configured to record the processor context for the instruction that triggers a HITM event, including the PC and data address in question. This information is made available via the Precise Event-Based Sampling (PEBS) facility [15]. The PEBS mechanism on Intel architectures supports hardware sampling of events via a Sample-After Value (SAV) parameter. Setting SAV to $n$ means that every $n^{\text{th}}$ event is sampled. Experience reports of those working with the PEBS mechanism [11] suggest that prime numbers are good SAV choices.

Prior to Haswell, the PEBS mechanism was imprecise in several ways. PCs were reported for a nearby but subsequent instruction, not the actual instruction triggering the PEBS event. In the presence of taken branches, determining the actual PC was difficult. On pre-Haswell machines the data addresses of PEBS events were also imprecise. Finally, Haswell is the first architecture to provide precise PC and data linear address information for the MEM_LOAD_-UOPS_LLC_HIT_RETIRED.XSNP_HITM event, adding support for PEBS HITM events triggered by load instructions (Figure 1a). Thus, Haswell is the first architecture to provide hardware support for cache contention detection.

As the next section shows, Haswell's support for detecting HITM events remains imprecise in a number of ways, but these obstacles can be overcome to build a low-overhead cache contention detector.

## 3.1 Haswell HITM Characterization

We undertook a detailed characterization of HITM event support in Haswell with over 160 test cases coded in assembly. These test cases each involve two threads engaged in true or false sharing, with either write-read/read-write or write-write sharing. Each thread performs the same operation repeatedly in an infinite loop, where the loop body varies across tests from a single memory operation to hundreds of branch, jump, arithmetic and memory instructions. Event sampling is disabled for all results in this section.

We present our test results in Figure 3, showing the percentage of HITM records that correctly identified the data address (top) and PC (bottom) involved in cache contention. Each point (dot or triangle) represents a test case, and the test cases are grouped by whether they exhibit false sharing (FS) or true sharing (TS) and writes by one thread (RW) or by both threads (WW). The light triangles in Figure 3 show that Haswell's HITM records, for the RW test cases, are quite accurate for data address (about 75% on average), and fairly accurate (about 40% on average) for PCs. However, HITM records are highly *inaccurate* for WW tests in terms of both data addresses and PCs.

The dark circles on the bottom of Figure 3 show the boost in PC accuracy we can obtain if we also classify PCs adjacent to the correct PC as correct. Allowing adjacent PCs boosts the percentage of correct PCs to 70% on average for

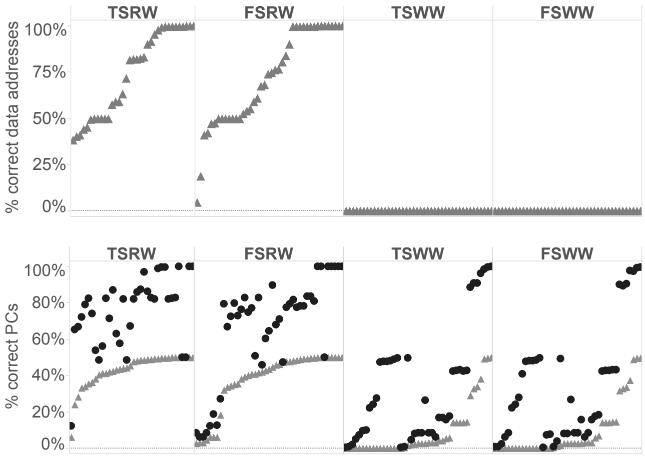


**Figure 3: The percentage of HITM records containing the correct data address (top) and PC (bottom) for our test cases. PC data shows both the percentage of exact PCs (light triangles) and adjacent PCs (dark circles).**

the RW tests, and 34% for the WW tests. Though over 99% of the incorrect PCs are from somewhere in the program's binary, we found no marginal benefit from enlarging the notion of "adjacent" to include two or more adjacent instructions. 95% of incorrect data addresses are from unmapped parts of the address space, with the remainder split between the stack and the kernel. Thus, we find that the PCs in Haswell's HITM records are often very close to the correct value, which helps LaserDetect locate contention accurately within the program source code.

The source of the accuracy discrepancy between RW and WW tests is that Haswell reports HITM events much more precisely for HITM-triggering *load* instructions (Figure 1a) than for stores, as the event name suggests. We found that stores which trigger HITM events (Figure 1c) still generate HITM records – total event counts are very similar between the RW and WW tests. However, HITM records arising from stores are far less precise, likely due to the delayed completion of stores in the presence of store buffers.

Despite the low accuracy of some aspects of HITM event reporting on Haswell, we find across the 33 workloads we evaluated that instances of cache contention exhibit two properties that Haswell's HITM support can exploit. First, contention typically consists of both reads and writes, instead of blind writes. Second, instances of cache contention are typically symmetric: all threads perform the same read-write operations, instead of (as in our test cases) one thread performing just reads and another performing just writes. Taken together, these properties combined with Laser's contention detection algorithm (Section 4) allow Laser to extract a meaningful signal from noisy HITM records by filtering obvious outliers and leveraging the sizable plurality of accurate HITM events.

# 4. HITM CONTENTION DETECTION

LaserDetect implements a pipeline, classifying and occasionally filtering HITM records as they are received from the hardware. The next section (Section 6) describes our overall system organization; here we describe how the detection pipeline works in detail (summarized in Figure 4).

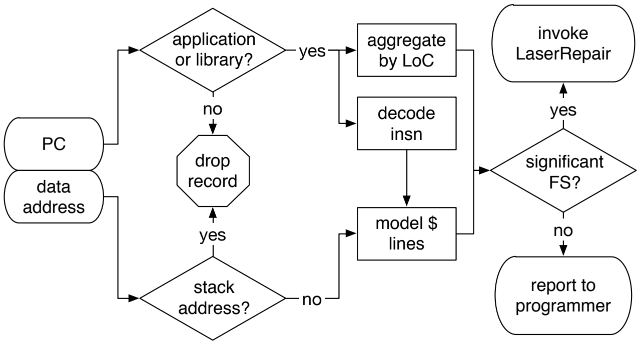


**Figure 4: LaserDetect's event processing pipeline for HITM records. The PC and data address of each record is used to identify the location and type of cache contention.**

## 4.1 Event filtering

When a HITM record first arrives from the hardware, its PC is classified as belonging to the application, a library, or some other code by parsing the application's virtual memory map (/proc/<pid>/maps on Linux). Our detector filters out HITM records with PCs that do not come from the application or its libraries as they are likely spurious HITM records.

Next, the data address is analyzed to determine if it is a stack address. Thread stacks are identified via the /proc memory map as described above. We currently ignore stack addresses as they are unlikely to be shared between threads and thus unlikely to be sources of cache contention.

## 4.2 Detecting source code locations

The next stage of the detector builds a map from PC to the number of HITM records received for that PC (regardless of data address), and reports the rate at which HITM events occur for each source code line. Source code lines with low rates of HITM events are filtered by a rate threshold (see Section 7.1) to avoid reporting uninteresting information to the programmer. It is straightforward for a programmer to adjust the rate threshold to filter more or less aggressively; adjustments can be made offline without rerunning the program.

Our process of identifying source code locations is robust to the errors that HITM records exhibit (Section 3). HITM records with incorrect data addresses have no effect as data addresses are not used to determine source locations, and HITM records that experience small shifts in the PC (Figure 3) often still map to the correct source code line.

## 4.3 Detecting the type of contention

In addition to finding the source code locations involved in contention, we seek to classify the type of contention exhibited: either true sharing or false sharing. To do so, we extend the detection pipeline by inserting a simple cache line model after the virtual memory map parsing. The first step in detecting the type of contention is to classify the PC of a HITM

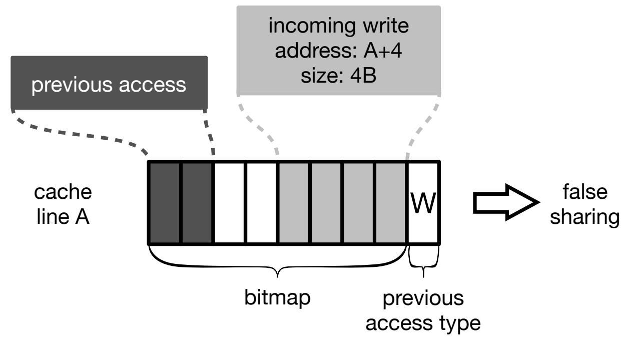


**Figure 5: The LaserDetect cache line model for a single 8-byte line. Each line records the type and location of its previous access in a bitmap. For line $A$, there is a previous 2B write and an incoming 4B write that triggers false sharing.**

record as a load or a store instruction, and to determine the size of the access. We analyze the application binary at runtime, to construct load and store sets identifying load PCs and store PCs and their sizes. These sets are then provided as inputs to the detector. On x86, it is possible for a memory instruction to be both a load and a store. We treat these instructions as such though any given HITM record contains only a single data address so such instructions are a potential source of inaccuracy in LaserDetect.

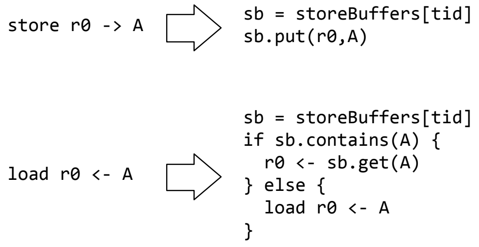
At runtime, the detector uses the load and store sets to determine whether the PC of a HITM record corresponds to a load or store instruction, and how many bytes that instruction accesses. As shown in Figure 5, the resulting memory access is sent to a simple cache line model that maintains information about the last access to each cache line: whether the access was a read or a write and what bytes were accessed (recorded in a bitmap). Cache lines are stored in a hash table to efficiently record just the small number of cache lines that experience contention. When a new access *N* arrives to a cache line, the detector calculates whether the access overlaps with the previous access *P* and whether one of the accesses was a write. If there is such overlap true sharing has occurred, otherwise false sharing has occurred. Each instance of true or false sharing is counted and associated with the PC of *N*. At application exit, for each source code line involved in contention, the detector also reports the number of true and false sharing events triggered by that source code line to allow a programmer to debug the contention.

### 4.4 Invoking LaserRepair

LaserDetect periodically checks the HITM event rate, triggering LaserRepair if the rate of false sharing events exceeds a given threshold. LaserRepair is provided with the PCs involved in false sharing and uses them to initiate its assembly code analysis (Section 5).

## 5. FALSE SHARING REPAIR

LaserDetect's ability to quickly and precisely detect cache contention at runtime allows false sharing to be repaired online with minimal application interference. While the Plastic [27] and Sheriff [18] schemes also perform online false sharing repair, neither of these approaches are sufficiently usable on commodity systems. Plastic requires custom OS or hypervisor support. Sheriff does not support the TSO memory consistency model, as its twin page mechanism for tracking the data written by each thread cannot detect silent stores [20]. With Sheriff, multi-byte stores that are atomic on conventional processors can instead appear to execute as a sequence of single-byte stores. Thus, Sheriff is unsuitable for use with programs that require TSO semantics, *e.g.*, x86 programs. We attempted to modify the Sheriff execution model to make it TSO-compliant, however this requires taking a page fault on *every* store to track all stores precisely. Such a TSO-compliant version of Sheriff has prohibitive performance overhead with slowdowns of up to three orders of magnitude. This section describes how LaserRepair's *software store buffer* approach, in contrast, can efficiently eliminate contention at runtime while preserving TSO semantics.



```
store r0 -> A        sb = storeBuffers[tid]
                     sb.put(r0,A)

                     sb = storeBuffers[tid]
                     if sb.contains(A) {
load r0 <- A           r0 <- sb.get(A)
                     } else {
                       load r0 <- A
                     }
```

**Figure 6: Pseudocode showing store (top) and load (bottom) instructions modified to use the SSB.**

### 5.1 Store Buffer Overview

LaserRepair uses a software store buffer (SSB) mechanism to temporarily eliminate the overheads of cache coherence, just as hardware store buffers do in modern multiprocessors. The SSB has higher latency, but better space-efficiency, than hardware store buffers (Section 5.5), hiding the cost of many more stores. LaserRepair uses the Pin dynamic binary instrumentation framework [23] to modify a running program to use the SSB mechanism. Because LaserDetect can precisely identify the PCs involved in contention, SSB modifications are focused on just the contending instructions.

LaserRepair's SSB operates similar to a hardware store buffer. Stores modified to use the SSB write into a thread-private buffer instead of writing to shared memory (Figure 6 top). Loads modified to use the SSB first check the buffer to see if it contains the address they are requesting (Figure 6 bottom). If so, the load returns the value from the buffer; otherwise, the load proceeds to shared memory. A bitmap records which bytes are contained within a buffer entry, to correctly handle unaligned memory accesses. The buffer can also be flushed through an explicit flush operation which writes all the buffered values to shared memory (analogous to a memory fence instruction).

### 5.2 Preserving Single-Threaded Semantics

To preserve single-threaded semantics, if a store uses the SSB all subsequent (with respect to program order) loads and stores must use the SSB as well, up to the next flush op-

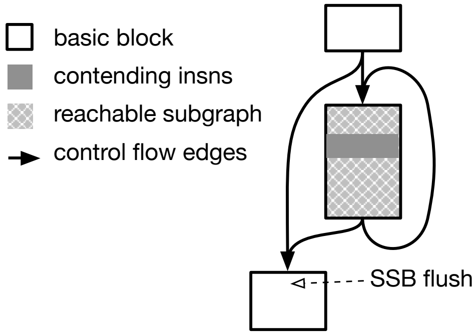


**Figure 7: Control-flow analysis focuses SSB instrumentation on just the contending region of code.**

eration. After a flush, subsequent operations no longer need to use the SSB until another store uses the SSB. Thus, the SSB mechanism can be toggled for sections of the dynamic execution.

Because the SSB operates on virtual addresses, if the same physical page is mapped to two different virtual pages within a single process, the SSB will not by default ensure that a load from one virtual page sees (buffered) values written to the other. To handle such rare cases, we could extend our userspace monitoring to track a process' `mmap` and shared memory object system calls and disable the SSB for any pages with such mappings. Note that the SSB does support interprocess communication directly, as each process has just a single virtual-physical mapping.

## 5.3 Pruning Instrumentation

To reduce performance overhead, we must minimize the use of the SSB mechanism on non-contended code. There is also an inherent tension in the placement of flush operations: flushing too frequently could itself trigger contention, but flushing too rarely requires more code to be modified to use the SSB.

We propose a simple static analysis, built using the Dytan program analysis framework for Pin [5], to balance these goals. Given the control flow graph and the locations of instructions involved in contention, our analysis identifies the basic blocks containing contending instructions. All memory operations in these basic blocks are modified to use the SSB. Flush operations are placed so that they post-dominate the modified basic blocks, which helps to minimize the dynamic occurrence of flushes. Finally, any additional blocks reachable from a modified block and not dominated by a flush are modified. For example, if contending instructions appear inside a loop (as in Figure 7), all memory operations in the loop body will be modified and a flush operation will be placed at the loop exit.

To reduce the number of loads using the SSB, we employ a simplified form of speculative alias analysis [12]. Our analysis assumes loads using a register unused by any store do not alias. Such loads do not require SSB modification.

To validate this speculation, an aliasing check is inserted between the def and use of each load address to see if it aliases the store addresses. If it does not alias, no further work is needed – the load (use) can safely skip the SSB. Multiple uses of the same def require only one check. If aliasing occurs, the SSB is flushed and the code is re-analyzed with speculative alias analysis disabled. Because (de)activating the SSB is a thread-local decision (see below), alias misspeculation can be handled locally without violating the consistency model.

## 5.4 Preserving TSO Semantics

In addition to preserving single-threaded semantics, our repair scheme must uphold the TSO memory consistency model of the x86 binaries running with LASERREPAIR. TSO permits some threads to use the SSB while others do not, allowing SSB usage to be focused on only the code that requires it regardless of remote threads' operations. TSO also adds two constraints to our repair scheme: 1) the SSB must be flushed at memory fences and 2) stores must be made visible in a total order consistent with program order.

Flushing at fences is straightforward to achieve by updating our control-flow analysis to account for fences. Fences may force flushes to occur with undesirable frequency, e.g., if a contending instruction is wrapped inside a small critical section. Such cases represent fundamental contention in the program that LASERREPAIR cannot repair. LASERREPAIR's static analysis estimates the dynamic cost of SSB usage and does not attempt contention repair if the ratio of stores to flushes is estimated to be low.

TSO's second requirement, that stores be made visible in a total order consistent with program order, constrains the implementation of our SSB to be logically FIFO. If a store to location A precedes a store to location B in program order, then A must become visible before B. Thus, TSO does not permit store A to use the SSB while store B goes straight to memory. While provably thread-private accesses can skip the SSB [31], such analysis is difficult with just assembly code so LASERREPAIR modifies all stores to use the SSB.

## 5.5 Efficient SSB Implementation

While a TSO-compliant SSB is most naturally implemented as a queue, enqueueing a store buffer entry for every store operation is impractical. Many of our workloads perform millions of stores before a flush operation. The only practical implementation maintains a single piece of storage for each memory location, similar to a *coalescing* hardware store buffer. However, a coalescing SSB permits memory reorderings illegal in TSO [32]. In particular, the problem arises when we try to flush the SSB, which is typically done one entry at a time: there is no order in which we can flush individual coalesced entries that is guaranteed to preserve TSO for all programs.

Fortunately, our Haswell platform provides a mechanism to preserve TSO semantics for our coalescing SSB with hardware transactional memory support. We perform a SSB flush with a single hardware transaction. Flush operations are thus strongly atomic [24], so no remote thread can observe one of the SSB's stores without the remaining stores. Thus, no memory reorderings are visible to remote threads. Because the SSB often contains just a handful of locations, a flush operation easily fits within the capacity of a hardware transaction. We insert a pre-emptive flush if the SSB grows beyond 8 entries (the L1 associativity of our machine) to avoid transaction overflow.

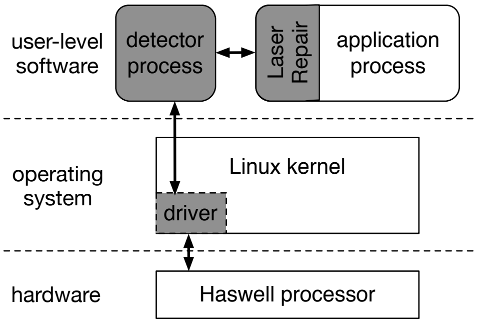


**Figure 8: An overview of the Laser system. Shaded blocks show Laser-specific components. Arrows indicate the flow of communication during detection.**

# 6. SYSTEM OVERVIEW

The LASER system has three main components (Figure 8): a Linux driver, a userspace detector, and a userspace online repair mechanism. The driver is a standard Linux kernel module compatible with stock Linux, and can be dynamically loaded and unloaded at runtime to minimize application interference. The driver configures the chip's performance monitoring unit to record HITM events into per-core memory buffers. The driver receives an interrupt whenever a per-core buffer is full, and empties the buffer by moving the records to an internal buffer that feeds into a kernel file-like device. The driver removes irrelevant information from the HITM records, such as the register file state, and sends only the PC, data address, and originating core to the detector. The driver tracks HITM events only for a specified process, and reconfigures the performance counters on each core appropriately on context switches to avoid tracking HITM events for irrelevant processes.

The detector is a standard userspace Linux process. Our current detector implementation is single-threaded and non-vectorized, though there is natural thread-level and data-level parallelism across HITM records that could be exploited to accelerate the detector. The detector forks the application process to be analyzed for cache contention and then configures the driver to record HITM events for the application process. By placing the detector in a separate process, we minimize interference with the application and require no application modifications. The detector reads HITM records from the driver and pushes the records through the detection pipeline (Section 4).

If the detector identifies significant false sharing in a benchmark, the repair scheme is invoked. The repair scheme is implemented using Intel's Pin dynamic binary instrumentation framework [23]. Pin can attach to and instrument a running process, allowing for online false sharing repair. The static analysis and store buffer instrumentation described in Section 5 are implemented as a Pintool.

# 7. EVALUATION

We evaluate the LASER system on a 4-core Haswell system with 32 GB of memory. The processor is a 64-bit Intel Core i7-4770K running at 3.4 GHz. TurboBoost is disabled in our experiments to make benchmarking more reliable [10]. Hyper-threading is also disabled as our driver prototype does not currently support hyper-threading. We used gcc version 4.7.2 with -O3 optimizations on openSUSE Linux 12.3. LASERREPAIR is built with Pin rev 62732. We present performance data as the average of 10 runs, after excluding the slowest and fastest runs. All LASER results are with a sample-after value (SAV) of 19, unless otherwise noted.

We evaluate LASER's detector on the Phoenix 1.0 [28], PARSEC 3.0 [2] and Splash2x [33] benchmark suites. We use the native inputs for Parsec and Splash2x. We use the largest available inputs for Phoenix, and furthermore run pca with a 4000 × 4000 matrix, extend histogram's inputs by 60x, and extend linear_regression's input by 120x to increase their running times to over a minute. For histogram, we examine two inputs: its standard large.bmp image (which we refer to as histogram) and an alternative image (histogram') that accentuates the false sharing present in the code. We exclude cholesky because its runtime is just 400ms on our system and it is not possible to scale its inputs.

We evaluate LASER's utility in terms of accuracy and performance, to see whether LASER is useful for finding performance bugs, and automatically fixing false sharing, at low performance cost. We compare LASER with Intel's VTune Amplifier XE 2015 [8] profiler and the open-source Sheriff scheme [18] for false sharing detection and repair.

## 7.1 Contention Detection Accuracy

We measure contention detection accuracy in terms of how many known performance bugs are missed (false negatives), how many spurious source code locations are reported (false positives) and whether the type of contention (true or false sharing) is identified. We created a database containing all known performance bugs in our benchmarks, by examining prior work [18, 19, 27]. Using LASERDETECT we found several novel performance bugs due to, *e.g.*, LASERDETECT's ability to uncover true sharing which most prior work cannot. These new and validated contention sources were integrated to create the final database.

We experimented with a range of values for LASERDETECT's *rate threshold* (Section 4.2), finding 1K HITMs/second to be a good balance between missing bugs and reporting too many false positives. We use this threshold for all benchmarks since LASER's accuracy is not particularly sensitive to the rate threshold as we demonstrate further below. VTune does not filter its reported locations of contention based on HITM rates, so for fairness we apply a similar balanced rate threshold (2K HITMs/sec works well for VTune) to exclude as many VTune false positives as possible without introducing false negatives. Sheriff-Detect uses its default filtering mechanism.

Table 1 shows the number of performance bugs present in our benchmarks and the number of false negatives missed by and false positives reported by LASERDETECT, VTune and Sheriff-Detect. LASERDETECT has superior accuracy to VTune, missing no bugs and reporting substantially fewer false positives. VTune misses an important bug in dedup (Section 7.4.2). VTune uses the same underlying PEBS HITM events that LASERDETECT leverages, and furthermore configures the PEBS mechanism to raise an interrupt after each HITM event for improved accuracy (which has significant

| Benchmark | Perf. Bugs | LASER FN | LASER FP | VTune FN | VTune FP | SheriffDet FN | SheriffDet FP |
|---|---|---|---|---|---|---|---|
| barnes | - | - | - | - | - | x | |
| blackscholes | - | - | - | - | - | - | - |
| bodytrack | 1 | - | 3 | - | 11 | x | |
| canneal | - | - | - | - | 1 | x | |
| dedup | 1 | - | - | 1 | 5 | i | |
| facesim | - | - | - | - | 2 | x | |
| ferret | - | - | - | - | 1 | - | 2 |
| fft | - | - | - | - | - | x | |
| fluidanimate | - | - | - | - | 5 | x | |
| fmm | - | - | - | - | - | x | |
| freqmine | - | - | 1 | - | 1 | i | |
| histogram | - | - | - | - | - | - | - |
| histogram' | 1 | - | - | - | 2 | 1 | - |
| kmeans | 1 | - | 10 | - | - | x | |
| linear_regression | 1 | - | - | - | - | 1 | - |
| lu_cb | - | - | - | - | 1 | x | |
| lu_ncb | 1 | - | 1 | - | 1 | x | |
| matrix_multiply | - | - | - | - | 1 | - | - |
| ocean_cp | - | - | - | - | 3 | x | |
| ocean_ncp | - | - | - | - | 3 | x | |
| pca | - | - | - | - | - | - | - |
| radiosity | - | - | - | - | 5 | x | |
| radix | - | - | 1 | - | 2 | x | |
| raytrace.parsec | - | - | - | - | 2 | i | |
| raytrace.splash2x | - | - | 3 | - | 5 | - | 1 |
| reverse_index | 1 | - | 3 | - | 1 | 1 | 1 |
| streamcluster | 1 | - | - | - | 3 | x | |
| string_match | - | - | - | - | - | - | - |
| swaptions | - | - | - | - | 1 | - | - |
| vips | - | - | - | - | 3 | i | |
| volrend | 1 | - | 1 | - | 3 | x | |
| water_nsquared | - | - | - | - | - | - | - |
| water_spatial | - | - | - | - | - | x | |
| word_count | - | - | 1 | - | - | x | |
| x264 | - | - | - | - | 2 | i | |
| Total | 9 | 0 | 24 | 1 | 64 | 3 | 4 |

**Table 1: The number of performance bugs identified in our benchmarks and, for each of LaserDetect, VTune and Sheriff-Detect, how many of these bugs are unreported false negatives (FN) and how many lines of code are spuriously reported false positives (FP). For Sheriff, x indicates a crash and i benchmark incompatibility. "-" indicates zero.**

performance ramifications, see Section 7.2), while LASERDETECT takes an interrupt only after thousands of events. Still, LASERDETECT's event processing pipeline (Figure 4) better filters spurious events and allows true sharing to be distinguished from false sharing. VTune simply reports source code locations where HITM events arise. Comparison with Sheriff-Detect is complicated by the fact that most of these workloads do not run with Sheriff-Detect. In contrast, LASER's ability to run a diverse range of workloads highlights its compatibility with the existing software ecosystem. Sheriff-Detect misses the performance bugs in linear_regression and histogram' which are significant sources of contention (Section 7.4.1). While Sheriff-Detect finds false sharing in reverse_index, it only identifies the allocation site of the falsely-shared object as being inside the program's malloc wrapper function, which is not helpful for locating the source code lines that participate in the false sharing.

We explored a range of *rate threshold* values with LASERDETECT and measured how they impact the number of reported false positives and false negatives. Figure 9 shows

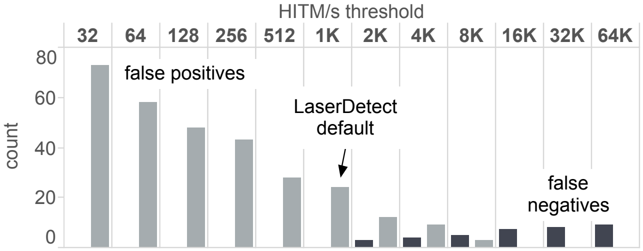


**Figure 9: The effect of different rate thresholds on Laser's detection accuracy. Note that the x-axis is a log scale.**

| Benchmark | contention | LASERDETECT | SheriffDet |
|---|---|---|---|
| bodytrack | TS | TS | x |
| dedup | TS | TS | i |
| histogram' | FS | FS | - |
| kmeans | FS | TS | i |
| linear_regression | FS | unknown | - |
| lu_ncb | FS | FS | x |
| reverse_index | FS | FS | FS |
| streamcluster | FS | FS | x |
| volrend | TS | TS | x |

**Table 2: Workloads with performance bugs. Columns show the actual contention type of each bug (false sharing: FS or true sharing: TS), and the type reported by LaserDetect and Sheriff-Detect. Shaded rows indicate benchmarks where LaserDetect finds the correct type of contention.**

that LASERDETECT's default threshold of 1K HITMs/second strikes a good balance between avoiding false negatives without producing too many false positives, but thresholds within a factor of 2 (note that the x-axis is a log scale) also produce good results. Thus, LASERDETECT is not particularly sensitive to the rate threshold and it is unlikely that developers would need to tune this threshold to achieve good detection results.

Detecting the correct type of contention (true versus false sharing) can help programmers triage bugs, *e.g.*, false sharing is typically easier to fix than true sharing as the former involves changing object padding or alignment, while the latter typically requires application restructuring. Detecting the correct type of contention is also crucial for avoiding fruitless attempts to automatically repair true sharing. Table 2 describes, for the benchmarks with performance bugs, the true type of contention, and the types reported by LASERDETECT and by Sheriff-Detect. LASER correctly identifies the contention type for six of the bugs, but is unable to conclusively identify the type of the linear_regression bug due to low data address accuracy (Figure 3). In contrast, VTune cannot distinguish true from false sharing and Sheriff-Detect reports the correct type only for reverse_index.

## 7.2 Performance

Figure 10 shows the performance overhead (lower is better) of running LASER and VTune Amplifier XE 2015, normalized to native execution (*i.e.*, without LASER or VTune). We discuss the LASER results first. kmeans, the slowest benchmark with LASER, experiences a 22% performance overhead but the majority of benchmarks experience little to

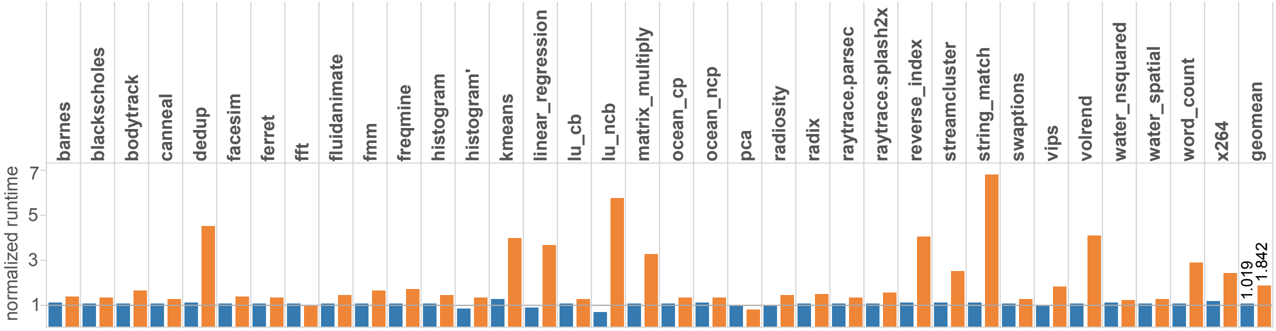


**Figure 10: The performance overhead (lower is better) of Laser (dark blue bars) and VTune (light orange bars), normalized to native runtime.**

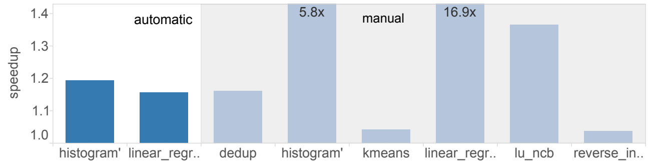


**Figure 11: Speedups (higher is better) resulting from LaserRepair (automatic) and LaserDetect's profiling information (manual).**

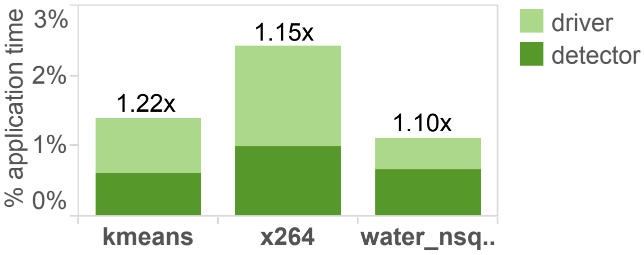


**Figure 12: Proportion of application time spent in the detector and driver for benchmarks with 10% or more performance overhead. Numbers atop each bar show the slowdown for each benchmark.**

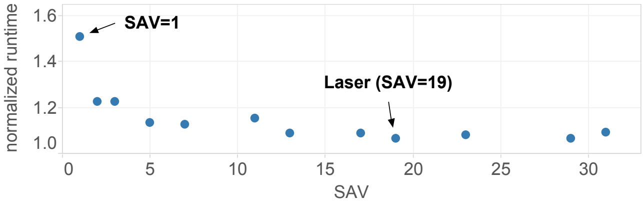


**Figure 13: The effect of sampling on the normalized runtime of dedup, for sample-after values from 1 to 31, including 1 and all prime values.**

no performance overhead with LASER– LASER's geometric mean overhead is just 2%. linear_regression is 16% faster and histogram' (histogram with an input that induces false sharing) 19% faster thanks to LASERREPAIR's online repair of their false sharing. lu_ncb is 30% faster with LASER due to a coincidental change in memory layout caused by LASER, which we explore in more depth in Section 7.4.

Figure 10 shows that VTune, which only performs contention detection and not repair, exhibits an average slowdown of 84% on our workloads, with a worst-case slowdown of 7x on string_match. As our accuracy results in Section 7.1 show, VTune's extra processing does not yield an accuracy advantage: LASER is substantially faster *and* more accurate than VTune in detecting cache contention. We compare with Sheriff's performance in detail in Section 7.3, again finding that LASER has a performance advantage over previous work both on average and in the worst case.

Figure 11 presents the speedups realized by using the LASER scheme. LASERDETECT's online profiling and LASERREPAIR's software store buffer repair mechanism automatically accelerate linear_regression and histogram'. The right, shaded part of Figure 11 shows the speedups obtained by making manual source code changes guided by LASER's contention reports, as detailed in Section 7.4.

### *7.2.1 Performance Characterization*

Figure 12 shows the proportion of time spent in the detector and driver as a proportion of the total CPU time of the application, for the benchmarks that have 10% or more performance overhead with LASER. Figure 12 shows LASER's direct contributions to performance overhead, not accounting for negative interference with the application. Figure 12 reveals that both the driver and detector are very lightweight. Generally, very little time is spent inside the LASER system, showing that even with high HITM rates contention detection can be performed cheaply.

Next we explore the impact of the sample-after value (SAV) on LASER's performance. Most benchmarks are relatively insensitive to SAV adjustments, but Figure 13 shows the behavior of dedup which sees large swings in runtime with different SAV values. Figure 13 shows that even modest sampling is effective at reducing dedup's performance overhead from 50% (with SAV=1) to 6% (with SAV=19, the LASER default), though there is no marginal benefit beyond this point.

## 7.3 Performance Comparison with Sheriff

We compared the performance of LASER against Sheriff [18] on our Haswell system, overlooking Sheriff's incompatibility with the x86 memory consistency model and the fact that a TSO-compliant version of Sheriff incurs prohibitive orders-of-magnitude slowdowns (Section 5). Using the latest code from github[1] we attempted to run Sheriff on

[1] https://github.com/plasma-umass/sheriff

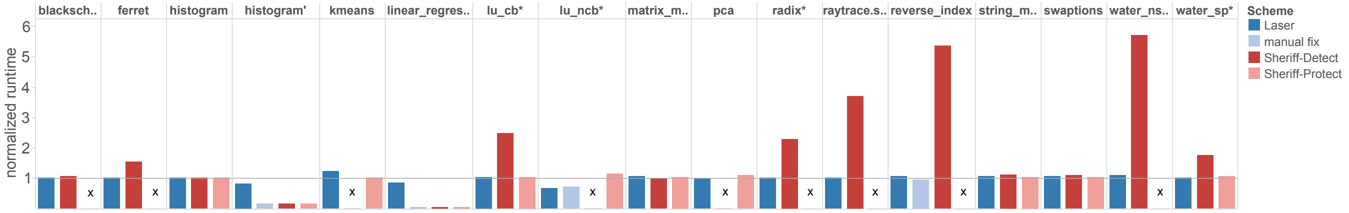


**Figure 14: Runtime (lower is better) of Laser, the code fixed manually based on Laser's reports (selected benchmarks), Sheriff-Detect and Sheriff-Protect, normalized to native execution.**

our benchmarks. Most of the Phoenix benchmarks work with Sheriff, but dedup, raytrace, vips and x264 use pthreads constructs that Sheriff does not currently support like spin locks, and freqmine requires OpenMP support. The remaining workloads encounter runtime errors with Sheriff.

Figure 14 shows the performance of LASER, Sheriff-Detect and Sheriff-Protect normalized to native execution for benchmarks where at least one Sheriff scheme works. An "x" indicates a runtime error. For histogram', linear_regression, lu_ncb and reverse_index we also show the performance of the manually-fixed program using the feedback from LASERDETECT. For lu_cb, lu_ncb, radix and water_spatial (indicated by a *), we used simlarge (instead of our default native) inputs as Sheriff would not run with the native input. On workloads with little synchronization (*e.g.*, swaptions) all three schemes have very low overheads. Both Sheriff-Detect and Sheriff-Protect isolate threads in separate address spaces whether a program suffers from false sharing or not, so both Sheriff schemes fix the false sharing in histogram' and linear_regression even though Sheriff-Detect does not detect anything (Table 1). On synchronization-intensive workloads, however, the cost of the Sheriff execution model becomes clear, with substantial slowdowns (*e.g.* water_nsquared). In contrast, LASER's non-intrusive approach has uniformly low performance overhead.

## 7.4 Case Studies

We examine the nine applications containing performance bugs (Table 2) in greater detail to qualitatively evaluate the utility of LASER, describing cases where LASER found instances of contention from prior work as well as new instances of contention missed by previous approaches.

### *7.4.1 Known performance bugs*

The Phoenix benchmark linear_regression is well-known for exhibiting extreme amounts of false sharing on the args array (Figure 2). LASER readily detects and repairs this false sharing despite a compiler optimization, present at the -O2 and -O3 optimization levels, that caches the fields SX through SY in registers to avoid repeatedly loading their values from memory. However, the updated value of each variable is still stored to memory at the end of every loop iteration, creating intense false sharing. This partial caching eliminates loads of these fields, converting the read-write false sharing present with -O1 compilation into write-write false sharing. While LASER is able to detect linear_regression's false sharing correctly at the -O3 optimization level, compiling with -O1 increases the HITM rate by nearly two orders of magnitude making detection even easier. Though each args array element is 64B in size on our platform (the same size as a cache line), the array itself is not 64B-aligned by default. Fixing linear_regression's false sharing by aligning the args array to a cache line boundary results in a dramatic 17x speedup (Figure 11).

The histogram benchmark is also known to exhibit false sharing [18, 19] as unpadded thread-private histogram counters can end up in the same cache line. On our system the default input does not generate false sharing, but the alternative input (histogram') exhibits significant false sharing, which illustrates how slippery false sharing can be. LASER dynamically adapts to histogram's runtime behavior, incurring no performance overhead for the default input and triggering online repair only when necessary for the alternative input.

LASER finds false sharing in reverse_index in the use_len[] array, as in previous work [18, 19]. The false sharing in use_len[] was fixed manually by padding the array elements for a 4% speedup. As this performance bug is minor, LASERREPAIR does not get automatically triggered to repair it.

### *7.4.2 Novel Performance Bugs*

LASER finds a novel instance of true sharing in dedup. The source code lines with the most contention are all identified as part of the concurrent queue implementation, which separates each pipeline stage in dedup's thread-parallel pipeline. Each queue is protected with a single lock, preventing enqueue and dequeue operations from proceeding in parallel. We replaced dedup's naive queue with the lock-free queue from the Boost C++ Lockfree library [3], which yields a 16% speedup (Figure 11).

The kmeans benchmark from Phoenix does not exhibit false sharing, thus previous approaches that focus exclusively on false sharing [18, 27] find the benchmark unremarkable. However, LASER is able to identify two new sources of contention in kmeans. First, our tool identifies read-write true sharing on the sum heap objects that are allocated in the main thread and then instantly handed off to worker threads. This contention can be avoided by allocating the sum objects on each worker thread's stack instead, which brings a 5% performance improvement by eliminating contention, parallelizing object allocation and leveraging cheap stack allocation as well (Figure 11). While kmeans allocates many of these heap objects during the life of the application, the amount of contention on each object is small. This pattern of contention that migrates from object to object is ill-suited to sampling-based approaches that assume contention will occur repeatedly on the same memory location (*e.g.* [18, 27]). Instead, low-overhead high-coverage approaches like

LASER are vital to detect migratory contention as in kmeans.

LASERDETECT uncovered a new performance bug in the Splash2x version of lu_ncb. lu_ncb experiences false sharing on the a array, its main data structure. While LASER detects this false sharing, online repair is not attempted because lu_ncb's sophisticated code structure is difficult for LASERREPAIR to analyze precisely, and the estimated impact of the SSB instrumentation is beyond the threshold deemed profitable. By modifying the source code directly to align the a array to a cache line boundary, a 36% speedup can be achieved.

In bodytrack, LASER identifies significant true sharing in the TicketDispenser::getTicket() function, which distributes unique counter values to threads. This communication is fundamental to load-balancing work amongst threads and would require significant application restructuring to remove.

### 7.4.3 Benchmarks without significant contention

LASER finds false sharing in the streamcluster and wordcount benchmarks, consistent with prior work [18, 27]. We also find a novel instance of read-write true sharing in volrend. Fixing these instances of contention does not result in any meaningful application speedup, but it does further demonstrate LASER's ability to detect contention precisely.

In streamcluster, the work_mem array is already padded to avoid false sharing, but the padding is insufficient for the 64-byte lines in our system. LASER identifies that additional padding is necessary. Increasing the padding reduces the number of HITM events in streamcluster by 3x but does not affect execution time on our system.

In wordcount, LASER detects that the use_len array induces false sharing when threads increment each element simultaneously. Introducing padding between the elements reduces HITM events but does not change performance.

In volrend, LASER identifies true sharing on the lock protecting the Global->Queue counter. Using batched atomic increments instead reduces the rate of HITM events by an order of magnitude but does not improve performance.

## 8. RELATED WORK

There have been many schemes proposed to detect false sharing and excessive true sharing in multithreaded programs, and even, in the case of false sharing, prevent it. We discuss the most relevant previous work here.

Some false sharing detectors rely on intensive program instrumentation, either via full-system simulation [30], via dynamic binary instrumentation tools like Pluto [14] and Liu [17], via memory shadowing along with dynamic instrumentation like the Dynamic Cache Contention Detection scheme [35] or extensive compiler instrumentation as with the Predator scheme [19]. Such approaches have the advantage of clear visibility into the program's memory access behavior, and can even, as Predator does, predict false sharing on machines with larger or smaller cache lines. The downside is that the runtime cost of such heavy instrumentation is prohibitive and can typically slow execution by an order of magnitude.

The Sheriff system [18] is a pure-software scheme that places each thread into its own private address space, sending updates between threads on synchronization operations. Sheriff provides two modes of operation. Sheriff-Detect detects false sharing by periodically write-protecting pages to see if multiple threads write to the same line. Sheriff-Protect avoids periodic page protection, relying on each thread's private address space to ensure that falsely-shared locations are mapped to distinct physical pages. Sheriff's performance overhead can be very high for applications that synchronize frequently: up to 11x with Sheriff-Detect and 47% with Sheriff-Protect according to [18]. Sheriff identifies the data, but not the code, involved in false sharing, making it harder to understand the root cause of false sharing. Finally, Sheriff focuses on false sharing, and does not discover the excessive true sharing in, *e.g.*, the kmeans benchmark. We conduct a detailed comparison with Sheriff in Section 7.3.

The Plastic system [27] focuses on both detecting and repairing false sharing, using a fine-grained memory remapping facility that allows individual bytes of virtual memory to be remapped to alternative physical memory addresses. This facility relies on custom OS or hypervisor support, along with standard page protection hardware and dynamic binary instrumentation. Plastic uses performance counters to measure the frequency of HITM events, and avoids doing additional work for programs where HITMs (and false sharing) are rare. Plastic's performance overheads are low – just 2% on average – but, like Sheriff, Plastic does not focus on detecting instances of true sharing. We did not undertake a direct comparison with Plastic due to the lack of source code availability. Furthermore, the current Plastic implementation relies on a custom hypervisor which would complicate our experimental setup.

Several projects, in addition to Plastic, have used performance counters to profile parallel code. [16] collects performance event counts to identify programs that suffer from false sharing. Based on the extent of false sharing, these instances are assigned suitable labels and a classifier is trained using this training data. Greathouse et al. [13] use HITM counts to guide a sampling-based data race detector. The TimeWarp system [25] uses counts of HITM events to infer the presence of software timers which can present a side-channel in security-critical code. None of these systems leverage the rich information available from PEBS events as LASER does; doing so would likely lower the performance cost and/or improve the accuracy of these previous schemes.

## 9. CONCLUSION

We have presented the design and implementation of LASER, a system for light, accurate sharing detection and repair. LASER leverages hardware support in Intel's Haswell architecture for recording the instructions involved in cache contention. We characterize the behavior of this hardware support on our Haswell machine to understand its capabilities and limitations. Our characterization data can inform future uses of this performance counter mechanism beyond our work, including concurrency bug detection and program analysis. We design our LASER system to maximize compatibility with the existing software ecosystem by leveraging Haswell's hardware support instead of changes to the compiler, OS or execution model. We evaluate LASER's accuracy and performance across a wide range of benchmarks, finding that it provides high detection accuracy and effective contention repair at low performance cost. LASER uncovers novel and important performance bugs missed by previous work, while incurring an average overhead of just 2%.

## 10. ACKNOWLEDGEMENTS

We thank John Demme for his help with exploring the PEBS mechanism, and Shobha Ranganathan and Andi Kleen for their technical help on the LASER driver development. This work was supported by NSF grant SHF-1525296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## 11. REFERENCES

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[2] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[3] Tim Blechmann. Boost.Lockfree 1.58.0. May 2015.

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[5] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[8] Intel Corporation. Intel VTune Amplifier 2015. May 2015.

[9] Advanced Micro Devices. *Preliminary BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 00h-0Fh (Kabini) Processors*, chapter 2.6.2 Instruction Based Sampling. 2013.

[10] Laurel Emurian, Arun Raghavan, Lei Shao, Jeffrey M Rosen, Marios Papaefthymiou, Kevin Pipe, Thomas F Wenisch, and Milo Martin. Pitfalls of accurately benchmarking thermally adaptive chips. *Power (W)*, 5:10.

[11] Stephane Eranian. Re: [Perfctr-devel] Re: quick overview of the perfmon2 interface. December 2005.

[12] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 222–231, 2002.

[13] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, New York, NY, USA, 2011. ACM.

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

[15] Intel Corporation. *IntelÂő 64 and IA-32 Architectures Software DeveloperâĂŹs Manual, Volume 3B: System Programming Guide, Part 2*, chapter 18.11. September 2014.

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

[17] C.-L. Liu. False sharing analysis for multithreaded programs. Master's thesis, National Chung Cheng University, 7 2009.

[18] Tongping Liu and Emery D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.

[19] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM.

[20] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 287–300, New York, NY, USA, 2014. ACM.

[21] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[22] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 553–563, New York, NY, USA, 2009. ACM.

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

[24] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006.

[25] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[26] mcmcc. false sharing in boost::detail::spinlock_pool? June 2012.

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

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

[29] Mikael Ronstrom. MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012. April 2012.

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

[31] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 524–535, Washington, DC, USA, 2012. IEEE Computer Society.

[32] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on*

*Computer Architecture*, 6(3):1–212, 2011.

[33] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[34] Benjamin P. Wood, Adrian Sampson, Luis Ceze, and Dan Grossman. Composable specifications for structured shared-memory communication. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 140–159, New York, NY, USA, 2010. ACM.

[35] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, New York, NY, USA, 2011. ACM.