Huron: Hybrid False Sharing Detection and Repair

Tanvir Ahmed Khan
University of Michigan, USA
takh@umich.edu

Yifan Zhao
University of Michigan, USA
evanzhao@umich.edu

Gilles Pokam
Intel Corporation, USA
gilles.a.pokam@intel.com

Barzan Mozafari
University of Michigan, USA
mozafari@umich.edu

Baris Kasikci
University of Michigan, USA
barisk@umich.edu

Abstract

Writing efficient multithreaded code that can leverage the full parallelism of underlying hardware is difficult. A key impediment is insidious cache contention issues, such as false sharing. False sharing occurs when multiple threads from different cores access disjoint portions of the same cache line, causing it to go back and forth between the caches of different cores and leading to substantial slowdown.

Alas, existing techniques for detecting and repairing false sharing have limitations. On the one hand, in-house (*i.e.*, offline) techniques are limited to situations where falsely-shared data can be determined statically, and are otherwise inaccurate. On the other hand, in-production (*i.e.*, run-time) techniques incur considerable overhead, as they constantly monitor a program to detect false sharing. In-production repair techniques are also limited by the types of modifications they can perform on the fly, and are therefore less effective.

We present Huron, a hybrid in-house/in-production false sharing detection and repair system. Huron detects and repairs as much false sharing as it can in-house, and relies on its lightweight in-production mechanism for remaining cases. The key idea behind Huron's in-house false sharing repair is to group together data that is accessed by the same set of threads, to shift falsely-shared data to different cache lines. Huron's in-house repair technique can generalize to previously-unobserved inputs. Our evaluation shows that Huron can detect more false sharing bugs than all state-of-the-art techniques, and with a lower overhead. Huron improves runtime performance by 3.82× on average (up to 11×), which is 2.11-2.27× better than the state of the art.

Tanvir Ahmed Khan and Yifan Zhao are co-lead authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314644

CCS Concepts • **Software and its engineering → Software performance**; *Multithreading*.

Keywords False sharing, Performance optimization

ACM Reference Format:
Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2019. Huron: Hybrid False Sharing Detection and Repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314644

1 Introduction

Over the past decade, pervasiveness of parallel hardware has boosted opportunities for improving performance via concurrent software. Today, almost every computing platform—data centers, personal computers, mobile phones—relies heavily on multithreading to best utilize the underlying hardware parallelism [1, 4, 12, 20, 45].

Alas, writing efficient, highly-multithreaded code is a challenge—subtle mistakes can drastically slow down performance. One prominent cause of this is cache contention, and true and false sharing are two of the most common reasons of cache contention [25]. True sharing occurs when multiple threads on different cores access overlapping bytes on the same cache line (*e.g.,* multiple threads accessing a shared lock object). False sharing, on the other hand, occurs when multiple threads access disjoint bytes on the same cache line [75]. In both cases, to maintain cache coherency [68], concurrent accesses will cause the cache line to go back and forth between the caches of different processor cores, thereby hurting performance.

In many cases, true sharing is intentional, *i.e.*, it may not be possible to prevent threads from sharing data on the same cache line. For example, developers intentionally share data among threads in order to implement a necessary functionality, such as a lock in a threading library or reference counters in a language runtime (*e.g.,* Java's Virtual Machine). Even when unintentional, developers can use existing tools (*e.g.,* profilers [19, 31, 50, 73]) to detect and fix true sharing.

False sharing, on the other hand, is more insidious. Developers may not be aware that threads accessing different variables at the source code level will end up on the same

cache line at runtime. Therefore, false sharing is almost invariably unintentional: it is challenging for developers to determine the presence of false sharing while programming.

Due to the challenging nature of false sharing—and its devastating impact on performance (*e.g.,* over 10× slowdown [8, 42])—there has been a lot of recent interest in automatically detecting and repairing it. Existing techniques for false sharing detection rely on static analysis [13, 30, 33], runtime monitoring [47, 75], compiler instrumentation [49], or hardware performance counters [25, 52]. In contrast, false sharing repair techniques rely on operating system support [21, 47], managed language runtime support [25], or custom in-memory data structures [52].

Alas, existing false sharing detection and repair techniques have limitations. In-house approaches [13, 33] use static analysis and compiler transformations to detect and repair false sharing, respectively. These techniques are therefore limited to eliminating false sharing for programs where the size and location of data elements can be determined statically. As a result, more recent false sharing detection and repair techniques work only in production. In-production approaches overcome the challenges of in-house techniques, but at the expense of suboptimal speedups [52], large runtime and memory overheads [21, 47, 49], narrow applicability [21, 25], and even memory inconsistency [47] (see §6 for details).

In this paper, we show that the common perception that in-house mechanisms are limited in terms of the types of programs for which they can detect and repair false sharing [47] is not necessarily correct. We show that it is possible to determine the effect of different program inputs on a false sharing bug, even after observing the bug for a single input.

Relying on the above observation, we present Huron, a hybrid in-house/in-production false sharing detection and repair system. For all false sharing bugs that can be detected in-house, Huron's novel algorithm generates a repair that can, in many cases, generalize to different program inputs. For false sharing instances that cannot be detected in-house, we leverage an existing in-production false sharing detection and repair mechanism [21], which we improved to only detect previously-unobserved false sharing instances with greater efficiency (*i.e.*, by caching the already-detected ones). Our insight is that, in many cases, developers will have in-house test cases that exercise the most performance-critical paths of their programs, which will allow our in-house false sharing detection and repair to be effective.

Performing false sharing detection and repair in house allows us to devise a novel repair mechanism that works by transforming the memory layout, which would have been otherwise too expensive to use in production. *The key insight behind our in-house false sharing repair is to group together data that is accessed by the same set of threads, and thereby shift falsely-shared data to different cache lines (i.e., eliminate false sharing).*

Despite repairing most false sharing in-house, we empirically show that, in many cases, Huron's repairs generalize to different inputs (*e.g.*, configuration parameters, thread counts, etc.), because the relation between a program's input and false sharing can usually be determined accurately using Huron's conservative static analysis. Huron can then use this relation to generate a fix that generalizes to any input.

In addition to eliminating false sharing, Huron's memory grouping improves spatial locality.

In summary, we make the following contributions:

- We present Huron, a hybrid in-house/in-production false sharing detection and repair technique. Huron's in-house technique uses a novel approach to eliminate false sharing by grouping together memory that is accessed by the same set of threads.
- We show that Huron can generate false sharing fixes that generalize to different program inputs.
- We show that Huron eliminates false sharing using benchmarks and real programs, delivering up to 11× (3.82× on average) speedup. Compared to the state of the art [21, 47], Huron delivers up to 8× (2.11-2.27× average) larger speedup, on average 33.33% higher accuracy, and up to 197-377× (on average 27-59×) lower memory overhead.

2 Background and Challenges

In this section, we first discuss the key challenges faced by false sharing detection and repair techniques. We then briefly describe how Huron addresses each of these challenges.

2.1 Effectiveness

The effectiveness of a false sharing repair[1] mechanism is the extent to which it can improve a program's performance.

In-production false sharing repair mechanisms modify the executable on the fly [21, 25, 47, 52], meaning they are limited in the extent of modifications they can perform and are less effective than in-house repair, as we show in §5.4. These techniques reduce false sharing by either splitting the falsely-shared data between different pages [21, 47] or by using special runtime support [52].

False sharing can be repaired more effectively if one can *surgically* modify a program's code and data at a fine granularity. A common and effective repair technique is to simply pad a cache line with dummy data in order to move the falsely-shared data to separate cache lines [69]. In fact, some in-production repair approches also use this technique [25]. However, these techniques are only applicable to managed languages (*e.g.,* Java), where programs pause at well-defined points (*e.g.,* garbage collection), thereby allowing for the code and memory layout to be restructured efficiently.

Even when existing approaches are able to repair false sharing, we show in §5.4 that, in many cases, they are much

[1]Since detection is a binary prediction, the effectiveness of a *detection* mechanism is the same thing as its accuracy.

less effective than Huron (up to 11×). In particular, by performing most of the false sharing repair in house, Huron achieves more speedup. Finally, since Huron does not rely on dummy padding, it can even outperform manual false sharing repair in many cases (7 out of 9, as we show in §5.4).

2.2 Efficiency

A false sharing detection and repair mechanism is considered *efficient* if its runtime performance overhead is low.

In-production techniques monitor the program for false sharing instances to fix them on the fly, thus, incurring considerable runtime overhead (*e.g.,* up to 11× [47]).

The efficiency of in-production false sharing detection techniques is further hindered by the fact that different program inputs may require detecting different instances of the same false sharing bug and generating a new repair strategy, both of which are costly. Execution #1 in Fig. 1 has a false sharing bug where threads t_1 and t_2 each access 60 bytes of data, and therefore share the last 4 bytes of cache line #1. Execution #2 shows that this false sharing instance can be repaired by padding each cache line with 4 bytes of data, which will force t_1 and t_2 to access their data on separate cache lines. On the other hand, in Execution #3, which is the result of a different input, each of the two threads accesses 30 bytes of disjoint data residing on a single cache line of 64 bytes. However, the 4-byte padding is not enough to shift the last 30 bytes to a separate cache line. Therefore, fixing the false sharing in this instance will require a 34-byte padding (or an additional 30 bytes), as shown in Execution #4.

To alleviate the overhead of in-production false sharing detection and repair, Huron performs as much of its detection and repair in house as possible, *e.g.*, by using test cases, etc. Consequently, Huron's in-production detection and repair is triggered less frequently than previous techniques and incurs less overhead (see §5.9). Furthermore, in many cases, Huron can generate an input-independent repair strategy that generalizes to multiple inputs (3.4).

2.3 Accuracy

The accuracy of a detection technique is the extent to which it can detect correct instances of false sharing (true positives) without flagging incorrect instances (false positives).

In theory, it is possible to build an in-production false sharing detector that does not suffer from false positives. However, since this is costly, all state-of-the-art false sharing detection strategies resort to sampling hardware events [21, 25] that are indicative of false sharing (*e.g.,* Intel HITM [18]) or use approximate algorithms [47].

By combining in-house and in-production false sharing detection, Huron achieves the best of both worlds. As shown in §5, Huron is not only more accurate than state-of-the-art detection approaches, but it is also more efficient.

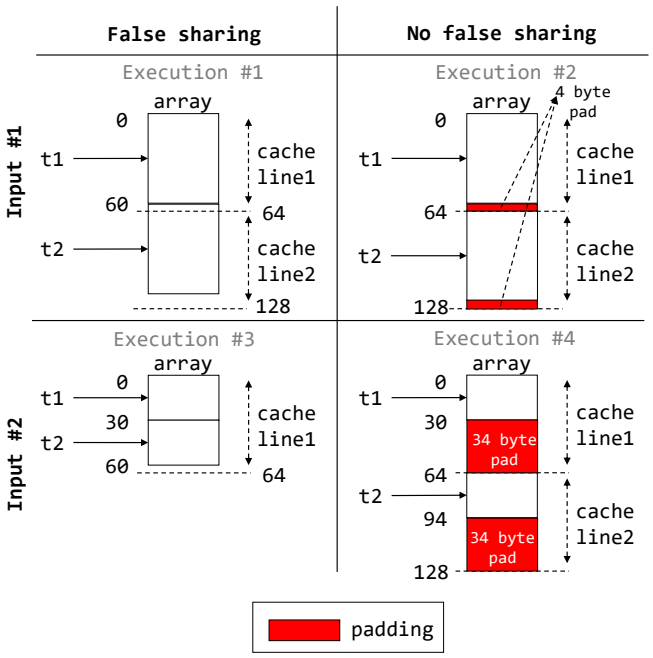

Figure 1. Input-dependent false sharing repair

3 Design

In this section, we describe the design of Huron, our hybrid in-house/in-production false sharing detection and repair system. Huron first detects and repairs false sharing in house using developer test cases. For false sharing instances that may not be detected or fixed in house (*e.g.,* due to thread nondeterminism or change of input), Huron uses its optimized version of an existing in-production technique [21].

Fig. 2 shows various components in Huron's design. Steps ①–④ occur in-house. In step ① (§3.1), the source code of the target program is fed into an instrumentation pass, responsible for instrumenting the program with false sharing detection code. In step ② (§3.2), Huron's in-house detection component detects false sharing using the instrumentation from the previous step. In step ③, Huron saves metadata (*e.g.,* program counter etc.) regarding the detected false sharing instances in a cache that is used to speed up in-production detection and repair. In step ④ (§3.3), the detected false sharing instances are used to perform memory layout transformations. The repair mechanism in this step groups together data that is accessed by the same set of threads, while separating falsely-shared data into different cache lines. Another key sub-step of the memory layout transformation is a special compiler pass (§3.4) that produces a generic false sharing repair strategy that works for multiple program inputs.

Steps ⑤ – ⑧ (§3.5) occur in production. In step ⑤, the program is deployed, while Huron performs its in-production false sharing detection. Moreover, Huron uses a cache of already-detected (in-house) false sharing instances – in step ⑥ – to reduce overhead. When Huron detects a new false

sharing instance in production, it fixes it by separating falsely-shared data into different pages using an existing tool [21] in step ⑦. Finally, in step ⑧, Huron saves the false sharing instances it detected in production, so that they can be repaired more effectively using the memory layout transformations the next time the program is built and deployed.

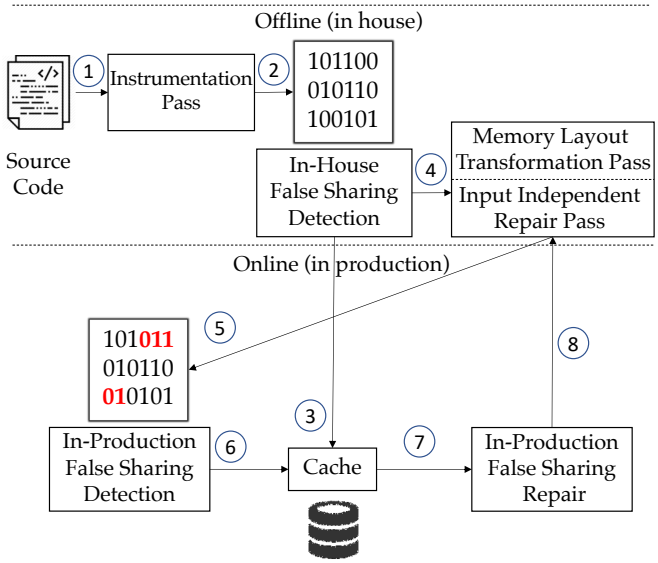

Figure 2. High-level design of Huron

3.1 Instrumentation Pass

Huron uses a compiler pass to instrument memory access and allocation instructions. Similar to all prior work, Huron's detection is geared towards detecting false sharing of global data and dynamically-allocated data. Huron does not monitor stack data for false sharing. While it is possible–although considered poor practice–for threads to share data through the stack, we have not observed this in practice.

Huron instruments all heap and global memory accesses, which is necessary for accurate detection. These include load and store instructions (including atomic load/store) as well as atomic exchange instructions. At runtime, the instrumentation logs the target size and the memory address of the load/store as well as the program counter of the instruction.

Huron also instruments all memory allocation instructions in order to collect the information necessary for generating false sharing fixes that generalize beyond the inputs observed in house. In particular, Huron's instrumentation inserts external function calls that log the program counter of the memory allocation instructions, as well as the start and end addresses of the allocated memory.

3.2 In-House False Sharing Detection

To ensure accurate detection, for each thread, Huron tracks how many bytes of data are accessed on which cache lines. If multiple threads access disjoint data on the same cache line at any point, Huron flags this as a false sharing instance.

```
1  struct global_t {
2    char *inputs[N_THREAD+1];
3    int red[N_THREAD][256];
4    int blue[N_THREAD][256];
5  };
6  global_t *global;
7  main(...) {
8    global = malloc(sizeof(global_t));
9    for(int i=0;i<N_THREAD;i++) {
10     for(int j=0;j<256;j++) {
11       global->red[i][j]=0;//Location 1
12       global->blue[i][j]=0;//Location 2
13     }
14   }
15 }
16 void calc_hist(int tid) {
17   char *begin=global->inputs[tid];
18   char *end=global->inputs[tid+1];
19   for (char *p=begin; p<end; p+=2) {
20     global->red[tid][*p]++;//Location 3
21     global->blue[tid][*(p+1)]++;//Location 4
22   }
23 }
```

Figure 3. Listing with false sharing that occurs when multiple threads execute the lines 20 and 21 in `calc_hist`.

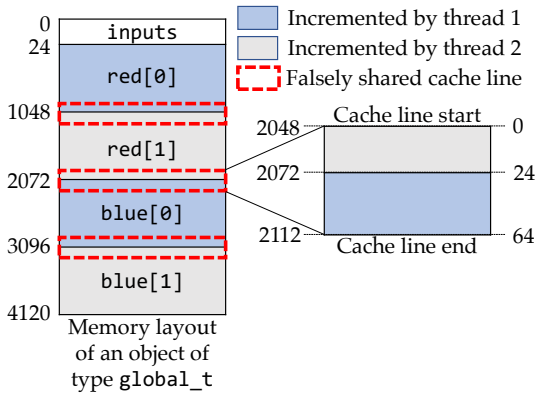

Figure 4. Memory layout of `global_t` data structure described in Fig. 3 and how the program suffers from false sharing due to thread 1 and thread 2 both incrementing non-overlapping values on the same cache line.

The listing in Fig. 3 shows a simplified example–adapted from `histogram` [63]–that suffers from false sharing when multiple threads spawned from `main` (thread creation code omitted for brevity) concurrently execute `calc_hist`, which increments thread-specific counters for pixel values, red and blue (green omitted for brevity). Since pixel values vary from 0 to 255, there are 256 counters (of type int) for each color. Each thread executing the function iterates over a portion of the image pixels (specified by pointers `begin` and `end`) to

retrieve and increment each counter. Fig. 4 shows the memory layout of an object of type `global_t` for a two-thread execution, *i.e.*, `N_THREAD = 2`. The `inputs` array is aligned at the beginning of a cache line and spans 24 bytes. Following `inputs` are the four subarrays `red[0]`, `red[1]`, `blue[0]`, `blue[1]`. The subarrays `red[0]` and `blue[0]` are accessed by thread 1 and the subarrays `red[1]` and `blue[1]` are accessed by thread 2. The dashed boxes denote the cache lines where subarrays both reside and cause false sharing when thread 1 and thread 2 access the lines from different cores.

To detect false sharing in house, Huron relies on testing workloads (*e.g.,* unit tests, integration test, stress tests). The rationale behind this design decision is that most of the time, developers already have test cases that cover the most performance-critical paths of their programs, which would allow Huron to detect and repair most false sharing bugs in-house. Those that are missed in house are detected and repaired by Huron in production (see §3.5).

During in-house false sharing detection, Huron records and computes certain information to account for dynamic program behaviors that vary across different runs in production. For instance, falsely-shared data will likely reside in different memory locations each time the program is run (*e.g.,* due to dynamic memory allocation [14] and ASLR [66]). Therefore, Huron needs to uniquely identify the location of false sharing during the detection phase to be able to repair it during any in-production run. Furthermore, Huron needs to uniquely identify the instructions involved in false sharing in order to modify their access offsets during repair. Finally, since Huron repairs false sharing by grouping together memory locations that are accessed by the same set of threads, it needs to track which threads access which part of the memory. To achieve these goals, Huron tracks and computes the following information:

1. A unique identifier of each *program location* accessing memory. This is a 3-tuple `(file name, line number, execution count)`, which uniquely identifies the location even if it is inside multiple loops. For brevity, in Fig. 5, step 1, we show the unique program locations as 1, 2, 3, 4 which correspond to lines 11, 12, 20, and 21 in Fig. 3.
2. A unique identifier of each *memory region*, defined as a combination of a memory allocation site and an access offset range. The allocation site is a 3-tuple `(file name, line number, execution count)` that uniquely identifies the allocation operation. In the example of Fig. 3, this would be `(histogram.c, line 8, 0)`. Huron converts the memory addresses accessed by each instruction into a range of offsets with respect to the allocation site. These offsets are calculated by subtracting the base address returned by the memory allocator from the accessed memory address. In Fig. 5, step ①, we omit the allocation site tuple and only show the memory offset ranges for brevity.
3. A *thread ID*, as shown in the last column of step ①, Fig. 5, where thread 0 is the main thread, and threads 1 and 2 are the worker threads executing `calc_hist` from Fig. 3.

These three pieces of information are used to produce the thread access bitmap (*i.e.,* the set of thread accesses) of each memory region (step ②, Fig. 3). Using this bitmap, Huron identifies and repairs false sharing (see §3.3).

Effect of Detection Window Granularity. Huron considers the entire execution of a program as a single time window when detecting false sharing instances. It is therefore possible for Huron to miss certain instances of false sharing (*i.e.*, incur false negatives). To see why, consider two disjoint time windows w_1 and w_2, where the entire execution runs for $w_1 + w_2$. Also consider two threads t_1 and t_2 accessing a 64 byte cache line l. Let's assume that in time window w_1, t_1 accesses the first 32 bytes of l, and t_2 accesses the last 32 bytes. Therefore in w_1, t_1 and t_2 are involved in false sharing. Now let's assume that in time window w_2, t_1 accesses the last 32 bytes of l, and t_2 accesses the first 32 bytes. Similarly, in w_2, t_1 and t_2 are involved in false sharing. However, if we consider the entire execution window $w_1 + w_2$, since both t_1 and t_2 access the first and last 32 bytes of l, they are truly sharing the cache line, hence there is no false sharing. Although it might seem useful to consider a fine-grained time window based on this example, this approach also suffers from false negatives, because it may not be possible to observe accesses involved in false sharing using a short window. Huron allows a developer to specify a detection time window. In §5.11, we show that using the entire exection as a single time window for false sharing detection is more effective.

3.3 Memory Layout Transformation Pass

We now describe how Huron repairs false sharing, assuming that program inputs and thread counts do not alter the false sharing behavior. We describe how Huron accounts for different inputs and thread counts in the next section.

Algorithm 1 describes a simplified version of Huron's memory layout transformation technique. The function Transform-Layout takes a list of memory bytes M as input. Each byte $m \in M$ has an attribute $m.bitmap$ denoting the IDs of threads which accessed this memory byte during in-house false sharing detection. The algorithm populates Q, which maps a thread ID bitmap (*i.e.*, a set of thread accesses) to a list of all the memory bytes accessed by the threads described via the bitmap (Line 1-5). For each memory byte m accessed by the same set of threads b, the algorithm sequentially assigns an offset (1 byte) in memory. The hashmap T keeps track of each new offset. After all memory bytes m with the thread access bitmap b are assigned an offset, the algorithm computes the new offset i to be the next multiple of cache line size CLSIZE (Line 13). This ensures that the next byte with a different thread access bitmap will be placed in a different cache line. Since false sharing occurs among

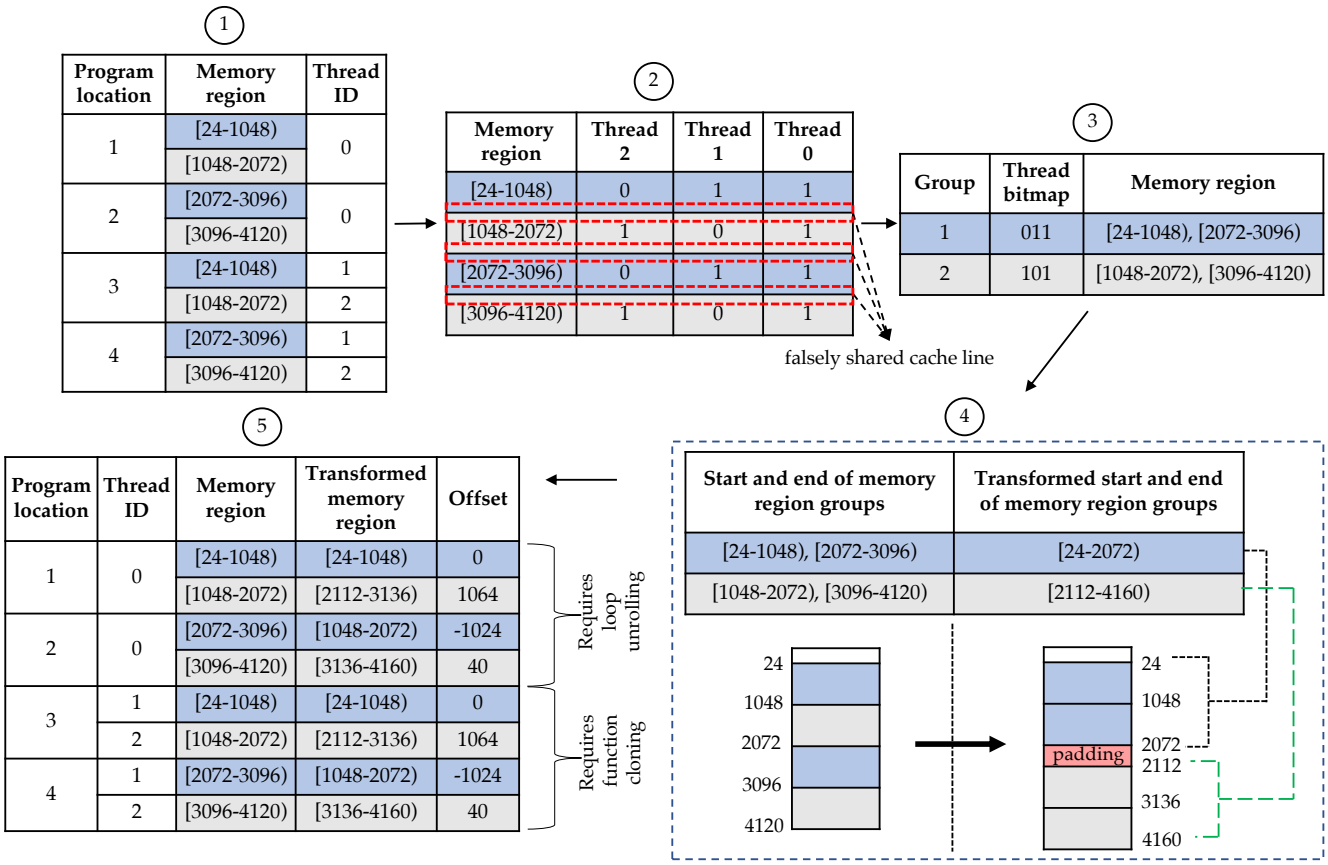

Figure 5. In-house false sharing repair via memory layout transformations for the example in Fig. 3. Memory region $[x - y)$ denotes bytes starting from x (inclusive) up to byte y (exclusive). For simplicity, we have omitted access to memory region $[0 - 24)$, *i.e.*, variable `inputs`. Cache line size is 64 bytes.

memory accesses with different corresponding bitmaps, the algorithm eliminates any potential false sharing.

Algorithm 1 False sharing repair via memory layout transformation

TRANSFORM-LAYOUT(M)

```
1:  Q ← Hashtable()
2:  for each m ∈ M do
3:      if Q[m.bitmap] = NIL then
4:          Q[m.bitmap] ← List()
5:      Q[m.bitmap].insert(m)
6:  i ← 0
7:  T ← Hashtable()
8:  for each b ∈ Q.keys do
9:      for each m ∈ Q[b] do
10:         T[m] ← i
11:         i ← i + 1
12:     if i%CLSIZE ≠ 0 then
13:         i ← i + (CLSIZE −(i%CLSIZE))
14: return T
```

For example, step ② in Fig. 5 shows how the threads from the example in Fig. 3 access various memory regions. The thread access bitmap in step ② shows where false sharing occurs with dashed boxes. Since memory regions [24-1048) and [1048-2072) share a cache line between offsets [1024-1098), and are accessed by different threads (i.e, with different thread bitmaps, where 011 ≠ 101), Huron detects false sharing. Huron then groups together memory regions with the same thread access bitmap as shown in step ③. Finally, Huron restructures the memory layout by placing memory regions in consecutive cache lines, as shown in step ④.

It is still possible for false sharing to occur between the end of a group of memory regions and the start of the next group. For instance, in Fig. 5, step ④, since the transformed groups share a cache line, Huron inserts 40 bytes of padding between the ranges [24-2072) and [2112-4160). As opposed to manual techniques that introduce dummy padding, Huron uses existing data in the program (*e.g.*, from the heap or the stack), thereby outperforming manual false sharing repair in most cases, as we show in §5.3.

Then, memory layout transformation updates instructions to access memory in the new layout. Huron uses two techniques for this in step ⑤: (1) *loop unrolling:* For some program locations, Huron needs to insert different memory access offsets for instructions for different iterations of a loop. For instance, the offsets that Huron needs to insert for program location 1 in Fig. 3 are 0 (iteration, `i=0`) and 1064 (iteration, `i=1`). (2) *function cloning:* For some program locations, Huron needs to add different offsets for different threads (example, location 3 and 4 in Fig. 5). Hence, Huron clones the function containing the program location and adds different offsets for each clone. For instance, `calc_hist` function contains the program location 3 where the offsets are 0 (Thread 1) and 1064 (Thread 2). As we explain in §3.4, Huron can adjust the offsets to account for thread counts in different executions.

Huron has to ensure correct and unmodified memory access semantics for the program after memory layout transformations. For this, Huron uses a complete, interprocedural, inclusion-based pointer analysis [2] to determine all the instructions that can access the modified layout. Huron instruments these instructions to (1) check whether they are accessing the new layout at runtime, and if so, (2) adjust the memory access offsets of the instructions accordingly. Alas, this analysis can have false positives, *i.e.,* instructions that Huron incorrectly considers as accessing the new layout. A large number of false positives will result in a large number of runtime checks, which might nullify Huron's performance improvements. To alleviate this, Huron only fixes false sharing if an instruction that needs to be modified took less than 1% of the total execution time during detection.

3.4 Input-Independent False Sharing Repair

In many cases, Huron can generalize its false sharing repair strategy to different program inputs after having detected false sharing for a single input. For such repairs, Huron does not need to rely on in-production false sharing detection and repair. Huron performs a *static range analysis* pass during layout transformation to compute the maximal range of memory that a given instruction can access, regardless of the program input. Consequently, Huron generates a memory layout transformation that will work for different inputs.

Algorithm 2 describes Huron's static range analysis, which leverages the type system to determine whether an operand of a memory access instruction (*e.g.,* store, load, atomic store, etc.) is an array type, and hence has fixed size. If that is the case, Huron determines the maximum memory range that the instruction can access based on its size.

FindMaxRange takes in as input an operand `Ptr` of a memory access instruction. The algorithm computes and returns the maximum range that the instruction can access by iterating over all aliases, `a`, of `Ptr` (Line 4). If the alias, `a`, is a function argument (Line 5), then all calls to this function, `f` invoked with argument `x`, are analyzed to determine the maximum range of `Ptr` (Line 9). All the ranges computed in each function are appended to the list `ranges` (Line 10). The algorithm then returns the widest possible range (Line 11). If `a` is found to be allocated via a `malloc` call (Line 12), then the algorithm returns `[S, S+R]` as the maximum range. Finally, if `a` is found to be derived using pointer arithmetic (Line 14), then `S` is recalculated based on `a`'s base pointer, `base`. If `base` is of array type, then `R` is multiplied by the number of elements in the array. If `Ptr` is found to be none of function argument, `malloc` call, or pointer arithmetic, then the algorithm will return an empty range (*i.e.*, the fix can not be input-independent).

Algorithm 2 Maximal memory range detection for input-independent repair

FindMaxRange(Ptr)

1: $R \leftarrow$ size of element type of Ptr ▷ size of the range
2: $S \leftarrow 0$ ▷ starting offset of the range
3: **while** True **do**
4: $a \leftarrow$ next alias of Ptr
5: **if** a is a function argument **then**
6: $ranges \leftarrow []$
7: $f \leftarrow$ function of a
8: **for each** call to function f invoked with argument x **do**
9: **for each** range (l, r) in FindMaxRange(x)) **do**
10: append $(l + S, r + S + R)$ to $ranges$
11: **return** [FindMin($ranges$), FindMax($ranges$)]
12: **if** a is a call to `malloc` **then**
13: **return** $[(S, S + R)]$
14: **if** a is derived from pointer arithmetic **then**
15: $base \leftarrow$ base pointer of a
16: $baseT \leftarrow$ type of $base$
17: **if** $baseT$ is an array type **then**
18: $n \leftarrow$ number of elements in $baseT$
19: $R \leftarrow R \times n$
20: $Ptr \leftarrow base$
21: **else**
22: **return** []

To understand how Huron's input range identification pass works, consider the statement `global->red[tid][*p]++;` on line 20 in Fig. 3, which increments a counter of how many pixels of an input image have matching *red* values. Since it is possible for an input to not contain all the 256 *red* levels, it is possible that during in-house false sharing detection, only `red[0][30]-red[0][56]` and `red[0][95]-red[0][197]` are accessed (assuming `tid = 0`).

Without input-independent false sharing repair, Huron would only perform memory layout transformations in this range and fail to repair false sharing for the rest of the array. However, with input-independent analysis, Huron discovers

the user code can access `red[0][0]-red[0][255]`, and generates a fix for the entire range.

If Huron statically determines a linear relationship between thread counts and the offsets in the transformed layout, it will parametrize the offset to be a function of the thread count. This allows Huron's fixes to generalize to different thread counts. For instance, the sub-array `blue[0][256]` (of Fig 3) has a parametrized offset function `(-(N_THREAD-1)*1024)`, allowing the offset to be changed from (−1024) to (−2048) when `N_THREAD` changes from 2 to 3.

Huron can generate input-independent false sharing repair for many programs, as we show in §5.5. However, this is not always possible. For example, the type information may be lost due to excessive pointer casts, or certain ranges may not be determined statically. In such cases, it relies on in-production false sharing detection and repair.

3.5 In-Production False Sharing Detection and Repair

In production, Huron deploys the program that was repaired in house and leverages a modified version of an existing in-production false sharing detection and repair tool, namely TMI [21]. In a nutshell, TMI works by monitoring hardware performance counters (i.e, HITM), which was shown in prior work [21, 52] to be indicative of false sharing.

A surge in HITM counts triggers TMI's false sharing detection. Huron's metadata cache of previously-detected false sharing instances (containing program locations, memory offsets, type of false sharing etc.) speeds detection up. We show in §5.9 that Huron's modified in-production false sharing detection technique is on average 2.1× faster than TMI.

If Huron discovers a false sharing bug in production, it uses TMI to create a *temporary* fix by converting threads to processes. However, as we demonstrate in our evaluation (§5), such a repair mechanism may not be effective or efficient. More specifically, TMI mistakenly treats true sharing as false sharing for a number of benchmarks and moves truly-shared data to different pages to "repair" this mistakenly-detected (i.e., false positive) false sharing instance. Even though TMI employs a memory-page-merging technique that ensures such false positives do not impact correctness (§2.2 of [21]), TMI's "repair" degrades performance due to the expensive nature of the merging technique. To overcome these challenges, Huron keeps a record of the detected false sharing instance for in-house repair, which it will attempt next time the program is built and deployed in production.

4 Implementation

We implemented Huron in 5,682 lines of C++ code. Huron uses LLVM [43] for instrumentation, analysis, and memory layout transformations. The static range analysis pass (§3.4) leverages the language type system. To infer the type of an object pointed to by a pointer, the pass traces the pointer back to the instruction where the memory for the object was allocated. Huron does this by recursively iterating over the use-def chain that leads up to the allocation site. We also integrated an existing Andersen-style alias analysis [10] into Huron for its input-independent repair pass.

Huron's in-house runtime tracks thread creations, memory allocations, and loads/stores using the instrumentation code as well as a shim library that intercepts and logs memory allocation and thread creation operations (*i.e.*, via `LD_PRELOAD`). Huron is open-sourced [24].

5 Evaluation

In this section, we answer several key questions:

- **Accuracy:** How accurately can Huron detect false sharing compared to the state of the art (§5.2)?
- **Effectiveness:** Can Huron repair more false sharing bugs than state-of-the-art tools? (§5.3)? How does the speedup provided by Huron compare to manual and state-of-the-art false sharing repair tools' speedup (§5.4)? How effective is Huron's input-independent false sharing repair (§5.5)? How the quality of in-house test cases affects Huron's effectiveness (§5.6)?
- **Efficiency:** What is the overhead of Huron's repair mechanism compared to the state of the art (§5.7)? How much overhead does Huron's in-house detection incur (§5.8)? How beneficial is Huron's false sharing cache in speeding up in-production detection (§5.9)? To what extent Huron's in-house component improves the efficiency of Huron's in-production component (§5.10)? How the false sharing detection time window granularity affects Huron's repair speedup (§5.11)?

5.1 Experimental Setup

Software. All experiments are conducted in Ubuntu 16.04, with Linux kernel version 4.4.0-127-generic using LLVM's front-end compiler clang 7 [43].

Hardware. We use a 32-core Intel E5-2683 machine with 128 GB of RAM.

Baselines. We compare Huron to the following state-of-the-art techniques:

1. **Sheriff** [47] is an in-production framework consisting of two tools: Sheriff-Detect and Sheriff-Protect. Sheriff-Detect tracks updates to a cache line by multiple threads to detect false sharing. Sheriff-Protect repairs false sharing by transforming threads into processes since, unlike threads, processes do not share the same address space.
2. **TMI** [21] is another in-production detection and repair mechanism. TMI monitors the surge in hardware events (*i.e.*, Intel HITM) to trigger its false sharing detection algorithm. TMI also uses thread-to-process transformation to eliminate false sharing.
3. **Manual** is a baseline where a human programmer repairs false sharing using dummy data padding to separate falsely-shared data onto different cache lines. Although laborious, manual repair can provide significant

speedups. In fact, state-of-the-art tools—*i.e.*, Sheriff and TMI—consider the speedups provided by manual repair an upper bound (which we show in §5.4 not to be the case). We do not include in-house false sharing detection and repair baselines in our evaluation, since these techniques [13, 33, 70] are targeted at specific applications and do not work well for the range of benchmarks we look at. For example, Jeremiassen et. al. [33] use an analysis that does not support complex access patterns, like accessing an array using values from another array as indices, just like `histogram` does.

Benchmarks. We use well-known benchmarks from the popular Phoenix [63] and PARSEC [7] suites, which have been used by many previous techniques for false sharing detection and repair [21, 47–49, 52, 59, 75]. We omit Parsec and Phoenix benchmarks that do not suffer from false sharing. We verified that these benchmarks do not contain false sharing by running all their available workloads with Huron's detector. We also evaluate Huron on three other benchmarks, `boost_spinlock` (from C++ boost [57] library), `ref_count` (adapted from Java's reference counting implementation [22]), and `histogramFS` (a modified version of `histogram` from the Phoenix [63] suite), which were all previously used by TMI [21]. We note that `boost_spinlock` and `ref_count` are from real world applications.

Aside from these benchmarks, we create and use two additional microbenchmarks, `lockless_writer` and `locked_writer`, that highlight the pros and cons of each false sharing repair technique. Both microbenchmarks incur false sharing due to multiple writer threads writing to the same cache line. As the name implies, `lockless_writer` does not rely on any synchronization instructions, while `locked_writer` uses locks for synchronization between the write operations.

We also use five microbenchmarks with true sharing (*i.e.*, multiple threads accessing overlapping data on the same cache line) when evaluating Huron's accuracy. In particular, state-of-the-art techniques suffer from accuracy issues and can incorrectly detect true sharing instances as false sharing bugs. The `single reader single writer`, `multiple readers single writer`, `multiple readers multiple writers` all read and write truly shared data to/from single/multiple threads. The `atomic writers` concurrently writes data from multiple threads using C++ atomic primitives [71] and `non-atomic writers` simply performs concurrent writes.

Metrics. Speedups in all the figures are relative to the execution time of the original benchmark.

We report all performance data as an average of 25 runs.

5.2 Accuracy of False Sharing Detection

Table 1 shows detection results for Huron, TMI and Sheriff. Here, TP stands for true positive, *i.e.*, correctly flagged real false sharing bugs; FP stands for false positive, *i.e.*, a true sharing instance incorrectly flagged as a false sharing bug; FN stands for false negative, *i.e.*, a real false sharing bug that a detector missed; and finally TN stands for true negative, a correct report of non-existence of false sharing.

Table 1 also reports the accuracy of each technique as $(TP + TN)/(TP + TN + FP + FN)$. We acknowledge that each technique can incur additional false negatives if the programs were run with different inputs (*e.g.,* besides all the existing test cases and workloads we used). Since such false negatives would impact all the techniques in the same way, we report accuracy numbers based on the executions we observe. In the observed executions, Huron's accuracy is 100%, whereas the accuracy of TMI and Sheriff is 66.67%.

Table 1. False sharing detection in existing benchmarks.

Benchmark	Sheriff Verdict	TMI Verdict	Huron Verdict
histogram	FN	TP	TP
histogramFS	TP	TP	TP
linear_regression	TP	TP	TP
reverse_index	TP	FN	TP
string_match	TP	TP	TP
lu_ncb	TP	TP	TP
word_count	TP	FN	TP
boost_spinlock	FN	TP	TP
lockless_writer	TP	TP	TP
locked_writer	FN	TP	TP
ref_count	TP	TP	TP
Volrend	TN	TN	TN
radix	TN	TN	TN
ocean	TN	TN	TN
fft	TN	TN	TN
canneal	FP	TN	TN
Single reader single writer	TN	FP	TN
Multiple readers single writer	TN	FP	TN
Multiple readers multiple writers	FP	FP	TN
Atomic writers	FP	FP	TN
Non-atomic writers	FP	FP	TN
Accuracy	**66.67%**	**66.67%**	**100.00%**

Out of 21 benchmarks, Sheriff and TMI mistakenly detect true sharing as false sharing (*i.e.*, FP) in 4 and 5 benchmarks, respectively. Huron's in-house detection does not incur false positives because of its fine-grained (i.e., cache-level) full memory tracing, as opposed to coarse-grained (e.g., page-level) and sampling-based detection employed by Sheriff or

TMI. Huron's in-production false sharing detection can temporarily incur a false positive since it relies on TMI. However, Huron eliminates this false positive for subsequent builds of the program during its in-house detection and repair.

Out of 21 benchmarks, Sheriff and TMI fail to detect false sharing (*i.e.*, FN) in 3 and 2 benchmarks, respectively. Sheriff suffers from false negatives due to reader-writer false sharing, as its detection mechanism compares only writes by different threads within a cache line. TMI's false negatives are due to inaccurate hardware events and sampling.

The detection inaccuracy of Sheriff and TMI has a substantial negative impact on the speedup they provide. We compute speedups for all the cases where Huron correctly detects and fixes a false sharing bug (*i.e.*, TP) and Sheriff and TMI miss (*i.e.*, `histogram`, `boost_spinlock`, `locked_writer` for Sheriff and `reverse_index`, `word_count` for TMI). For these benchmarks, Huron provides up to 5.3× and on average 3.6× greater speedup than Sheriff, and up to 4.3× and on average 2.6× greater speedup than TMI.

5.3 Ability to Repair False Sharing Bugs

As shown in Table 1, Huron successfully detects and eliminates false sharing in all 11 benchmarks (cells marked True Positive–TP–in Table 1). On the other hand, not only Sheriff detects fewer false sharing instances (*i.e.*, 8), it is also only able to repair 5 of them. Sheriff's repair fails for 3 out of 8 cases (`boost_spinlock`, `locked_writer`, `ref_count`) because, to preserve correctness, it is unable to repair false sharing due to synchronization primitives. Similarly, TMI only detects 9 false sharing bugs, out of which it repairs 7. The detection fails in 2 out of the 9 cases (*i.e.*, `lockless_writer`, `locked_writer`), because TMI causes the program to hang.

5.4 Effectiveness Comparison to State of the Art

We compare Huron's effectiveness (*i.e.*, the speedup it provides) to the state of the art only for the benchmarks where TMI and Sheriff are able to detect and repair false sharing bugs. These are: `linear_regression`, `histogram`, `histogramFS`, `string_match`, `lu_ncb`, `boost_spinlock`, `lockless_writer`, `locked_writer`, and `ref_count`.

Fig. 6 compares the speedup that Huron and Sheriff provide. Huron's speedup outperforms Sheriff's by up to 7.96× and on average 2.72×. Huron performs better than Sheriff for benchmarks with frequent synchronization, where Sheriff's repair mechanism incurs high overhead.

Fig. 7 compares the speedup that Huron and TMI provide. Huron outperforms TMI by up to 5.7× and on average 2.1×. Similar to Sheriff, TMI's repair is also not as efficient as Huron's for benchmarks with heavy synchronization.Huron is more effective than Sheriff and TMI largely due to its novel in-house repair technique.

Finally, we compare Huron with manual false sharing repair. Interestingly, as shown in Fig. 8, Huron outperforms manual repair in 7 out of 9 benchmarks–albeit with a small

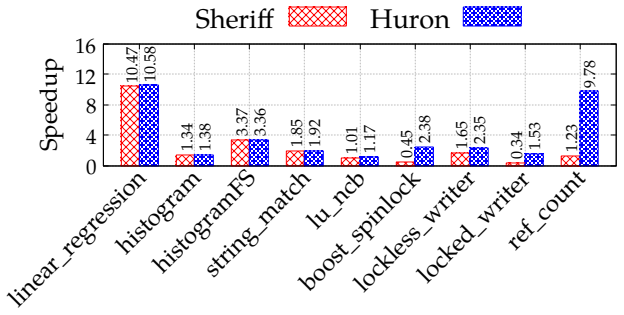

Figure 6. Speedup comparison to Sheriff [47]. All standard deviations are less than 1.62%.

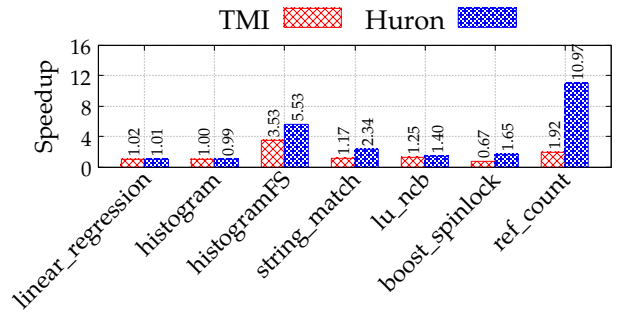

Figure 7. Speedup comparison to TMI [21]. All standard deviations are less than 3.2%.

margin of up to 8%. This is because, as explained in §3.3, when Huron needs to insert padding, it uses existing program data rather than dummy padding that manual repair uses.

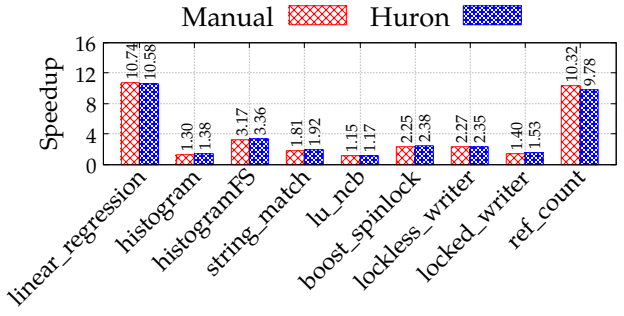

Figure 8. Speedup comparison to manual repair. All standard deviations are less than 0.71%.

In summary, compared to the state of the art [21, 47], Huron provides up to 8× (2.11-2.27× average) more speedup.

5.5 Effectiveness of Huron's Input-Independent Repair

In this section, we investigate the effectiveness of Huron in generating input-independent repairs based on false sharing bugs detected in house. To illustrate this, we do a detailed

analysis of `histogram`, which we have discussed previously in detail as part of the example in Fig. 3. The `histogram` benchmark experiences false sharing that is input-dependent. Specifically, the arrays `red` and `blue` (and `green` in the actual program) are involved in input-dependent false sharing.

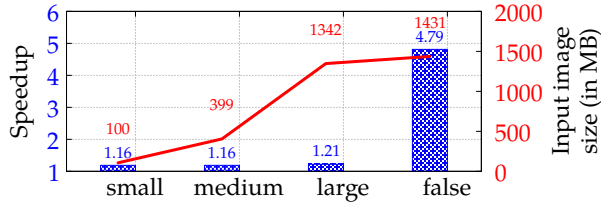

Figure 9. Huron' speedup for different input images of `histogram`. Huron generates an input-independent repair for the "small" input. All standard deviations are less than 3.47%.

Fig. 9 shows the speedup provided by Huron for `histogram`'s various input images, namely "small", "medium", and "large.". In house, Huron detects false sharing using the "small" input image and generates an input-independent repair that works for the other images in production. For "small", "medium", and "large" input images, the speedup varies between 1.16 – 1.21×. However, the speedup is 4.79× for the "false" input image. The "false" image (generated from a script provided by the authors of TMI [21]) is actually used as input for `histogramFS`. The pixel values of the image trigger a lot of false sharing and therefore Huron delivers considerable speedup for this benchmark.

5.6 Impact of Test Cases on Effectiveness

In this section, we evaluate the impact of test cases on Huron's effectiveness. For this, we initially only rely on Huron's in-production false sharing repair component to simulate a worst case scenario, where Huron does not have access to any test cases in house. Using these results, Huron then repairs all the false sharing instances in house. Fig. 10 shows the results. For all the benchmarks, Huron's in-production detection and repair provides speedup that is about the same as TMI's. This is expected since Huron relies on TMI, with the added facility to log metadata that Huron uses to subsequently repair false sharing instances in house. Huron's in-house repair that uses the feedback it receives from its in production component provides greater speedup than TMI.

5.7 False Sharing Repair Overhead

We now first evaluate the memory overhead of false sharing repair for Huron, TMI, and Sheriff. We then study the effect of these tools' memory usage on the speedup they provide.

We compare the memory overhead of Huron to the memory overhead of Sheriff and TMI in Fig. 11 and Fig. 12, respectively. The overheads in both plots are relative to the memory usage of the unmodified binaries. Huron uses up to 377× and on average 59× less memory than Sheriff. We

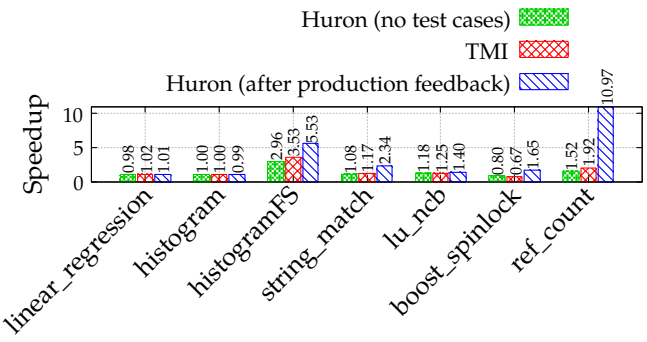

Figure 10. Huron's in-house repair that uses the feedback received from its in production component provides greater speedup than TMI.

also observe that on average, Huron's memory overhead is less than 8%. Only for `lockless_writer`, Huron incurs a high (60%) memory overhead. This happens because there is not enough data that is being accessed by the same set of threads in this program. Consequently, Huron has to rely on padding to eliminate false sharing, which incurs overhead. Sheriff, on the other hand, uses significantly more memory than the original benchmark, as it transforms each thread into a process (which creates multiple copies of many pages). TMI has a lower memory overhead than Sheriff, thanks to its various optimizations (*e.g.,* thread private memory). However, TMI also uses a few auxiliary buffers to accelerate the detection and elimination of false sharing. Nevertheless, Huron's memory overhead is on average 27× (and up to 197×) lower than TMI. Although Huron's in-production repair leverages TMI, Huron avoids much of TMI's overhead by fixing most false sharing instances in house.

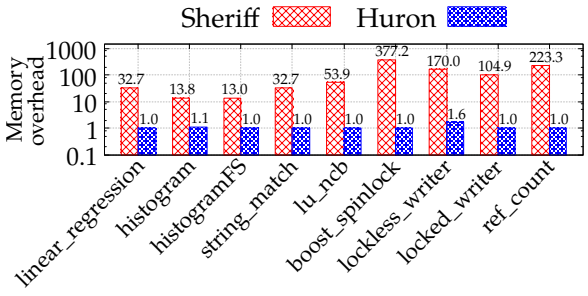

Figure 11. Memory overhead comparison to Sheriff. The y-axis is log scale.

The high memory overhead of TMI and Sheriff also has a significant impact on the speedup that they can provide. Specifically, if the underlying system's memory is constrained, a program with a large memory footprint will not enjoy the same speedups that Huron can provide.

To evaluate the effect of memory pressure, we vary the per-process memory limit from 50 to 25 megabytes and measure

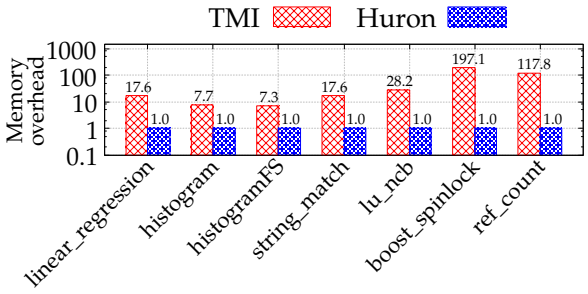

Figure 12. Memory overhead comparison to TMI. The y-axis is log scale.

the normalized speedup relative to the original benchmark for Sheriff, TMI, and Huron. As shown in Fig. 13, Huron provides up to 41% and on average 15% more speedup than Sheriff. Similarly, Huron provides up to 214% and on average 49% more speedup than TMI, as shown in Fig. 14.

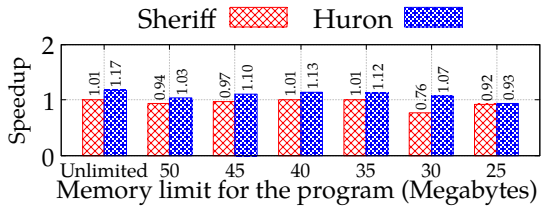

Figure 13. Huron achieves up to 41% (15% on average) higher speedup than Sheriff when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 0.59%.

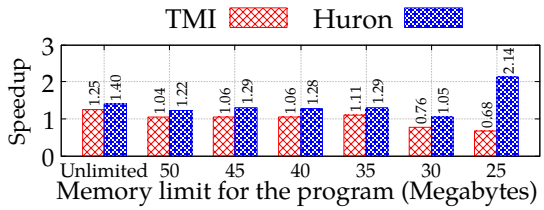

Figure 14. Huron achieves up to 214% (49% average) higher speedup than TMI when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 4.16%.

5.8 Overhead of Huron's In-House Detection

The key advantage of in-house detection is that it is an offline process, and hence does not affect the program execution time in production. However, we still measure the overhead of Huron's in-house detection as the time added to the execution of the user program, *i.e.*, the execution time of the instrumented binary minus that of the unmodified one. In this experiment, we use the same inputs as in §5.2. As shown in Fig. 15, this overhead is on average less than two minutes (115 seconds) and is no more than 257 seconds. These numbers are quite reasonable for an offline process and on par with the offline overhead of state-of-the-art memory performance profilers [16, 17, 51, 58, 60].

5.9 Effect of False Sharing Cache on In-Production Overhead

In this section, we evaluate how the cache of false sharing bugs detected in house reduces the overhead of Huron's in-production false sharing detection. Without Huron's cache, the TMI detector (which Huron relies on) does a lot of extra work to determine (1) the program counter of where false sharing occurs, (2) whether there is a read-write or write-write sharing, (3) whether there is true or false sharing.

As shown in Fig. 16, Huron's cache speeds up in-production detection up to 27.1× and on average by 11.3×. Note that for 3 out of 9 benchmarks (`linear_regression`, `string_match`, and `lu_ncb`) evaluated in § 5.4, speedups are not shown, because TMI removes false sharing from these programs using its allocator even before false sharing instances occur in production (*i.e.*, the cache is never used).

Finally, we measure the impact of caching on memory. In particular, we determine that Huron's cache takes 512KB in the worst case and 91.21KB on average. Considering that the memory overhead of Huron is considerably lower than state-of-the-art tools (on average 27-59×), we consider the modest cache overhead to be acceptable.

5.10 Contributions of In-House and In-Production Repair Techniques

We now quantify the extent to which Huron's in-house and in-production repair techniques contribute to the overall efficiency of Huron. For this, we use a benchmark with 10 false sharing instances. We then vary the number of false sharing instances repaired in house from 1 to 10. Since Huron's in-production repair converts threads into processes, it eliminates all the remaining false sharing bugs at once.

Fig. 18 shows our results. The *in-house* speedups are due to Huron's in-house repair only, and the *hybrid* speedups are due to Huron's hybrid in-house/in-production repair. Both in-house and hybrid speedups increase with an increase in the number of false sharing instances repaired in house. The contribution of the in-production component is the difference between the *hybrid* and the *in-house* speedup values. In-house repair contributes more to the overall speedup than in-production repair. For instance, when five of the false sharing bugs are fixed in house with the other five repaired in production, the in-house component provides 39.3% of the speedup, whereas the in-production component provides 6.8% of the speedup. This trend is stable across data points.

5.11 Effect of Detection Time Window Granularity on Repair Speedup

As discussed in §3.2, Huron's false sharing detection time window granularity can impact how many false sharing bugs it detects and fixes, and thus the speedup it provides. In this

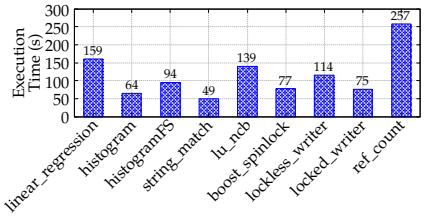

Figure 15. Overhead of Huron's in-house detection, in seconds.

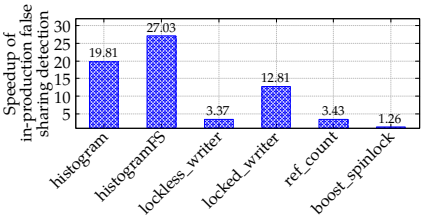

Figure 16. Speedup of in-production false sharing detection using Huron's false sharing cache.

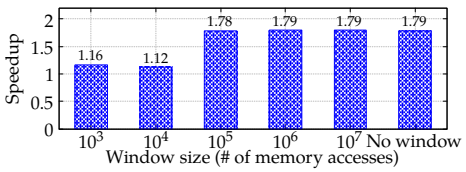

Figure 17. The effect of detection window granularity on Huron's speedup (`lockless_writer` benchmark).

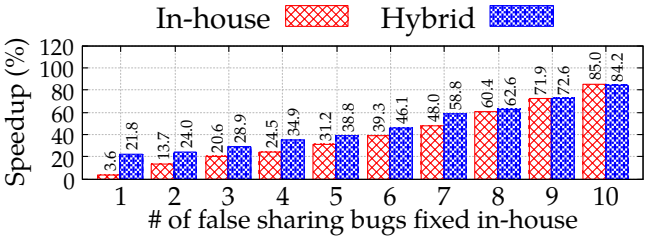

Figure 18. Huron's provides greater performance speedup as more false sharing instances are repaired in house. All standard deviations are less than 3.62%.

section, we evaluate the effect of the detection time window granularity on the speedup provided by Huron's repair.

Fig. 17 shows the speedup provided by Huron for the `lockless_writer` benchmark under different window granularities. The probability of missing the detection of a false sharing bug due to a large time window increases with the number of falsely-shared cache lines. We choose `lockless_writer`, which we know (via manual inspection of the small code base) suffers from false sharing in exactly 1024 cache lines, the largest number across all benchmarks.

The speedup provided by Huron increases with a larger window size. This is because a small window limits Huron's ability to detect out-of-window memory accesses that are involved in false sharing with in-window memory accesses. Therefore, Huron misses many false sharing instances for smaller window sizes and the speedup after repair is sub-optimal. As the window size increases, so does the number of detected false sharing instances and the ensuing speedup.

6 Related Work

There is a substantial amount of related work that has studied the detection and elimination of false sharing. In many cases, existing work attempts to detect and repair false sharing through dynamic analysis [25, 75]. Some approaches also rely on heuristics to make detection and repair more scalable and efficient [11, 15, 27, 55]. Approaches based on static analysis can also detect and eliminate false sharing by reorganizing a program's code [3, 5, 41]. Alas, static false sharing repair [13, 33, 39] was shown to be mostly effective in well-defined use cases [47]. Huron combines the best of both worlds to achieve good accuracy, effectiveness, and efficiency.

Simulators and profilers can be used to detect false sharing. For instance, [65] employs full system simulation using Simics [53] to identify cache miss causes. Other tools [29, 46] detect false sharing by instrumenting memory accesses using Intel Pin [51] or Valgrind [60]. Predator [49] uses LLVM [44] instrumentation to record memory accesses by different threads to detect false sharing. These tools are helpful for detection; however, they provide few hints as to how to best repair. These techniques can also incur high runtime overhead and suffer from false positives [47]. Huron's hybrid approach does not suffer from these problems.

In order to reduce the runtime overhead, many techniques [9, 21, 25, 28, 31, 32, 48, 52, 54, 56, 59, 62] rely on performance counter values that are correlated with false sharing (i.e, Intel HITM [18]). Once the counter events surge beyond a certain threshold, these techniques trigger a more rigorous detection algorithm [21, 25, 52]. Huron uses a similar approach for its in-production false sharing detection. However, because Huron can detect and repair many false sharing instances in house, it does not trigger in-production false sharing detection frequently, and thus incurs low overhead.

Another technique uses machine learning on hardware event counts [32] to detect false sharing. We plan to improve Huron by leveraging a machine learning-based approach to speed up its in-house false sharing detection.

A common false sharing elimination approach in prior work is to transform program threads to processes so that they no longer share the same address space. While Grace [6] first proposed this idea to avoid concurrency bugs, Sheriff [47] adopted this technique to repair false sharing. This approach incurs high memory overheads and can be inefficient because the pages shared across processes need to be merged frequently. TMI [21] partially addresses these challenges by introducing a data structure called page twinning store buffer (PTSB). PTSB has smaller memory footprint, and it allows pages to be merged more efficiently. Despite these benefits, the speedup provided by PTSB (and thus TMI) cannot fully attain the performance benefits provided by manual repair or Huron (see §5.4). Finally, unlike Sheriff and TMI, Huron does not suffer from false positives.

Many other studies [23, 26, 34–38, 40, 61, 64, 67, 72, 74] have investigated data layout optimizations to improve performance. The main goal of these tools is to improve the memory layout to maximize spatial locality, while Huron utilizes memory layout transformations to eliminate false sharing.

7 Conclusion

Detecting and fixing all false sharing is difficult. Even if false sharing instances are identified during development, repairing them manually can be a daunting task. In this paper, we described Huron, a hybrid in-house/in-production mechanism that detects and repairs false sharing automatically. Huron's repair mechanism groups together data accessed by the same set of threads to shift falsely-shared data to different cache lines. Huron detects and repairs all false sharing bugs with 100% accuracy in the 21 benchmarks that we evaluated. Huron achieves speedups of up to 11× and on average 3.82×. Overall, Huron is 33.33% more accurate than state-of-the-art detection and repair tools and it provides up to 8× and on average 2.11-2.27× greater speedup.

Acknowledgments

We thank our shepherd, Jennifer B. Sartor, and anonymous reviewers for their insightful feedback and suggestions. This work was supported by the Intel corporation and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

[3] Jennifer M Anderson and Monica S Lam. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM Sigplan Notices*, Vol. 28. ACM, 112–125.

[4] Jon Petter Asen, Jo Inge Buskenes, Carl-Inge Colombo Nilsen, Andreas Austeng, and Sverre Holm. 2014. Implementing capon beamforming on a GPU for real-time cardiac ultrasound imaging. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 61, 1 (2014), 76–85.

[5] Prithviraj Banerjee, John A Chandy, Manish Gupta, Eugene W Hodges, John G Holm, Antonio Lain, Daniel J Palermo, Shankar Ramaswamy, and Ernesto Su. 1995. The PARADIGM compiler for distributed-memory multicomputers. *Computer* 28, 10 (1995), 37–47.

[6] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: safe multithreaded programming for C/C++. In *ACM sigplan notices*, Vol. 44. ACM, 81–96.

[7] Christian Bienia, Sanjeev Kumar, and Kai Li. 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 47–56.

[8] William J Bolosky and Michael L Scott. 1993. False sharing and its effect on shared memory performance. In *Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems*.

[9] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 152–167.

[10] Jia Chen. 2018. Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen. [Online; accessed 16-Nov-2018].

[11] Mei-Ling Chiang, Chieh-Jui Yang, and Shu-Wei Tu. 2016. Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in NUMA multi-core systems. *Journal of Systems and Software* 121 (2016), 72–87.

[12] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.

[13] Jyh-Herng Chow and Vivek Sarkar. 1997. False sharing elimination by selection of runtime scheduling parameters. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*. IEEE, 396–403.

[14] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L Titzer. 2015. Memento mori: dynamic allocation-site-based optimizations. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 105–117.

[15] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. 2015. Lira: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2.

[16] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. *ACM SIGPLAN Notices* 47, 6 (2012), 89–98.

[17] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2013. Multithreaded input-sensitive profiling. *arXiv preprint arXiv:1304.3804* (2013).

[18] Intel Corparation. 2016. Intel (R) 64 and IA-32 Architectures Software Developer's Manual. *Combined Volumes, Dec* (2016).

[19] Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2014. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*.

[20] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[21] Christian DeLozier, Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2017. TMI: thread memory isolation for false sharing repair. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 639–650.

[22] David Dice. 2011. False sharing induced by card table marking. https://blogs.oracle.com/dave/false-sharing-induced-by-card-table-marking. [Online; last accessed 05-August-2018].

[23] Wei Ding and Mahmut Kandemir. 2014. CApRI: CAche-conscious data reordering for irregular codes. *ACM SIGMETRICS Performance Evaluation Review* 42, 1 (2014), 477–489.

[24] efeslab. 2019. Huron: A false sharing detection and repair tool. https://github.com/efeslab/huron. [Online; accessed 12-April-2019].

[25] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: online detection and repair of cache contention for the JVM. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 251–265.

[26] Olga Golovanevsky, Alon Dayan, Ayal Zaks, and David Edelsohn. 2010. Trace-based data layout optimizations for multi-core processors. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 81–95.

[27] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 254–269.

[28] Joseph L Greathouse, Zhiqiang Ma, Matthew I Frank, Ramesh Peri, and Todd Austin. 2011. Demand-driven software race detection using hardware performance counters. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 165–176.

[29] Stephan M Günther and Josef Weidendorfer. 2009. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 26–33.

[30] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J Ramanujam, and Ponnuswamy Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 129–144.

[31] Intel. 2017. Tutorial: Identifying False Sharing - C Sample Code. https://software.intel.com/en-us/vtune-memory-access-tutorial-linux-c

[32] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. 2013. Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 30.

[33] Tor E Jeremiassen and Susan J Eggers. 1995. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*. Vol. 30. ACM.

[34] Ismail Kadayif and Mahmut Kandemir. 2004. Quasidynamic layout optimizations for improving data locality. *IEEE Transactions on Parallel & Distributed Systems* 11 (2004), 996–1011.

[35] M Kandemir, A Choudhary, and J Ramanujam. 1998. Improving locality in out-of-core computations using data layout transformations. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, 359–366.

[36] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prithviraj Banerjee. 1999. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*. IEEE, 95–102.

[37] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prithviraj Banerjee. 1999. A graph based framework to detect optimal memory layouts for improving data locality. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*. IEEE, 738–743.

[38] Mahmut Kandemir, Alok Choudhary, Jagannathan Ramanujam, Nagaraj Shenoy, and Prithviraj Banerjee. 1998. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing*. Springer, 422–434.

[39] Mahmut Kandemir, Alok Choudhary, J Ramaujam, and Prithviraj Banerjee. 1999. On reducing false sharing while improving locality on shared memory multiprocessors. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. IEEE, 203–211.

[40] Mahmut Kandemir and Ismail Kadayif. 2001. Compiler-directed selection of dynamic memory layouts. In *Proceedings of the ninth international symposium on Hardware/software codesign*. ACM, 219–224.

[41] Ken Kennedy and Ulrich Kremer. 1995. Automatic data layout for high performance Fortran. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. ACM, 76.

[42] Randall L. Hyde and Brett D. Fleisch. 1996. An Analysis of Degenerate Sharing and False Coherence. 34 (05 1996), 183–195.

[43] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*. 1–2.

[44] Chris Lattner and Vikram Adve. 2004. 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*. IEEE Computer Society, 75.

[45] Ming Li, Shucheng Yu, Yao Zheng, Kui Ren, and Wenjing Lou. 2013. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *IEEE transactions on parallel and distributed systems* 24, 1 (2013), 131–143.

[46] CL Liu. 2009. False sharing analysis for multithreaded programs. *Master's thesis, National Chung Cheng University* (2009).

[47] Tongping Liu and Emery D Berger. 2011. Sheriff: precise detection and automatic mitigation of false sharing. *ACM Sigplan Notices* 46, 10 (2011), 3–18.

[48] Tongping Liu and Xu Liu. 2016. Cheetah: detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 1–11.

[49] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. 2014. PREDATOR: Predictive False Sharing Detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2555243.2555244

[50] Tongping Liu, Guangming Zeng, Abdullah Muzahid, et al. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 298–313.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[52] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. 2016. LASER: Light, accurate sharing detection and repair. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 261–273.

[53] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[54] Joe Mario. 2016. C2C - False Sharing Detection in Linux Perf. https://joemario.github.io/blog/2016/09/01/c2c-blog/. [Online; last accessed 04-August-2018].

[55] Jason Mars and Lingjia Tang. 2013. Understanding application contentiousness and sensitivity on modern multicores. In *Advances in Computers*. Vol. 91. Elsevier, 59–85.

[56] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 118–129.

[57] mcmcc. 2012. False sharing in boost::detail::spinlock_pool? https://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool. [Online; accessed 09-June-2018].

[58] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. 2016. End-to-end memory behavior profiling with DINAMITE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1042–1046.

[59] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T Meyer, William Aiello, and Andrew Warfield. 2013. 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*. ACM, 141–154.

[60] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[61] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns.

In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 562–571.

[62] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 335–348.

[63] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee, 13–24.

[64] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 140–153.

[65] Martin Schindewolf. 2007. Analysis of cache misses using SIMICS. *Master's thesis* (2007).

[66] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[67] Byoungro So, Mary W Hall, and Heidi E Ziegler. 2004. Custom data layout for memory parallelism. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 291–302.

[68] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (2011), 1–212.

[69] Herb Sutter. 2009. Eliminate false sharing. *Dr. Dobb's Journal* 5 (2009).

[70] O. Temam, E. D. Granston, and W. Jalby. 1993. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 410–419. https://doi.org/10.1109/SUPERC.1993.1263488

[71] WikiSysop. [n. d.]. Atomic operations library. https://en.cppreference.com/w/cpp/atomic. [Online; last accessed 07-August-2018].

[72] Zhichen Xu, James R Larus, and Barton P Miller. 1997. Shared-memory performance profiling. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 240–251.

[73] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering* 23, 5 (2018), 3034–3071.

[74] Yuanrui Zhang, Wei Ding, Jun Liu, and Mahmut Kandemir. 2011. Optimizing data layouts for parallel computation on multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 143–154.

[75] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic cache contention detection in multi-threaded applications. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 27–38.