TMI: Thread Memory Isolation for False Sharing Repair

Christian DeLozier
delozier@cis.upenn.edu
University of Pennsylvania

Ariel Eizenberg
arieleiz@cis.upenn.edu
University of Pennsylvania

Shiliang Hu
shiliang.hu@intel.com
Intel Corporation

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

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania

ABSTRACT

Cache contention in the form of false sharing and true sharing arises when threads overshare cache lines at high frequency. Such oversharing can reduce or negate the performance benefits of parallel execution. Prior systems for detecting and repairing cache contention lack efficiency in detection or repair, contain subtle memory consistency flaws, or require invasive changes to the program environment.

In this paper, we introduce a new way to combat cache line oversharing via the Thread Memory Isolation (Tmi) system. Tmi operates completely in userspace, leveraging performance counters and the Linux ptrace mechanism to tread lightly on monitored applications, intervening only when necessary. Tmi's compatible-by-default design allows it to scale to real-world workloads, unlike previous proposals. Tmi introduces a novel code-centric consistency model to handle cross-language memory consistency issues. Tmi exploits the flexibility of code-centric consistency to efficiently repair false sharing while preserving strong consistency model semantics when necessary.

Tmi has minimal impact on programs without oversharing, slowing their execution by just 2% on average. We also evaluate Tmi on benchmarks with known false sharing, and manually inject a false sharing bug into the `leveldb` key-value store from Google. For these programs, Tmi provides an average speedup of 5.2x and achieves 88% of the speedup possible with manual source code fixes.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Software and its engineering → Runtime environments**;

KEYWORDS

false sharing, performance counters, memory consistency, C, C++

ACM Reference format:
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, Cambridge, MA, USA, October 14-18, 2017 (MICRO-50),* 12 pages.
https://doi.org/10.1145/3123939.3123947

1 INTRODUCTION

Multicore architectures, which have invaded nearly every market segment from datacenters to smartphones and credit-card computers, promise improved performance and energy efficiency compared to single-core designs. However, parallel code can suffer from many subtle performance bugs that can quickly sap these potential benefits. In this work we target false sharing bugs, a subtle class of performance bugs not well-supported by existing tools.

False sharing bugs are widespread, arising both in benchmark suites like Phoenix [31] and Parsec [4] and also in production-quality code like the Boost libraries [28], MySQL [32] and the Linux kernel [5, 8, 9]. False sharing is notoriously difficult to debug because it is not explicitly visible in the source code, and instead often arises due to a complex interplay between the compiler, runtime memory allocator and hardware cache coherence protocol. False sharing can slow memory accesses by an order of magnitude in our experiments, and exacts a significant energy penalty for generating and processing cache coherence traffic.

In this work, we target online repair of false sharing, as with previous systems [19, 24, 30]. Online repair is particularly attractive because it requires no programmer intervention, no access to source code (which may be unavailable or subject to bit-rot), and no system downtime. We know first-hand of cloud and datacenter workloads that suffer from false sharing issues and whose execution cannot be stopped for debugging due to availability requirements.

In this paper we show how to build such an online repair scheme leveraging access to performance counters from commodity multicore machines. In contrast with prior work, we seek a system that satisfies four criteria simultaneously: 1) compatibility with a commodity hardware and system stack, 2) upholding memory consistency model semantics, 3) low performance overhead for programs without false sharing and 4) speedups close to that of manual fixes for automatically-repaired programs. As Table 1 shows, existing work falls short along multiple dimensions. The Sheriff scheme [19] executes each thread as a process, resulting in large slowdowns for programs with frequent synchronization operations; Sheriff also does not preserve correct semantics for programs with C/C++ atomics or inline assembly. Plastic [30] relies on custom OS or hypervisor support, and uses dynamic binary instrumentation that limits the effectiveness of its repairs. LASER [24] leverages performance counters for accurate and low-overhead detection and

Requirement	Sheriff [19]	Plastic [30]	LASER [24]	Tmi
compatible	×	×	✓	✓
memory consistency	×	✓	✓	✓
overhead w/o contention	27%	6%	2%	2%
% of manual speedup	92%	~30%	24%	88%

Table 1: Requirements for effective false sharing repair.

preserves program semantics but its repair mechanism is much slower than manual repairs.

Plastic's repair mechanism, based on dynamic binary instrumentation, is complex and can capture only about 1/3 of the benefit of manual repairs for the sole workload, `linear-regression`, on which its repair scheme is activated. LASER's repair mechanism achieves only 1.2x speedups on `linear-regression` and `histogram`. In contrast, Sheriff's online repair mechanism achieves nearly all of the performance gains of manual repairs. Unfortunately, Sheriff's design does not preserve program semantics in the presence of inline assembly or C/C++ atomic operations ([24], Section 2.2). We find several uses of such mechanisms among our benchmarks, including the real-world workload `leveldb`, and their use is widespread in popular libraries like glibc. Indiscriminate use of Sheriff's repair mechanism can thus lead to invalid execution.

To meet all our requirements simultaneously, we have designed the Thread Memory Isolation (Tmi) system for combating cache line oversharing. To ensure **high compatibility** with existing systems, we have designed Tmi to operate entirely from userspace on a standard Linux system stack. Tmi leverages the `ptrace` and `perf` mechanisms to monitor and, when necessary, modify a program's execution with minimal overhead. Tmi provides compatibility-by-default, intervening only if a program exhibits significant false sharing. Tmi reduces false sharing by dynamically making contended regions of memory thread-private, converting a running thread into a process so that its virtual address space mapping can be modified independently of other threads. Tmi focuses false sharing repair only when and where it is needed, allowing for the performance gains of manual source code fixes without any programmer intervention.

The efficiency of Sheriff's online repair mechanism motivates understanding the precise conditions under which its use is permitted. To **uphold memory consistency model guarantees**, we propose a new *code-centric* memory consistency model which enables consistency-aware runtime optimizations. Ours is the first work, we believe, to consider the implications of multiple consistency models co-existing within a single program. Without code-centric consistency, a runtime optimization is forced to be conservative, *e.g.*, providing strong TSO semantics [35] for an entire program when only parts are written in assembly. With Tmi we demonstrate that online false sharing repair can leverage code-centric consistency, and we anticipate that future runtime systems and hardware can benefit as well.

Finally, Tmi leverages **lightweight hardware performance counters** available on recent Intel processors to detect false sharing cheaply and without any application modifications, which underpins Tmi's compatible-by-default design. This helps minimize performance overheads for programs without contention.

This paper makes the following contributions:

- The design and implementation of the Tmi system. Tmi exacts just a 2% slowdown for programs without false sharing. For programs with false sharing, including benchmarks and a version of `leveldb` with an injected bug, Tmi provides a 5.2x speedup on average.
- A code-centric consistency model which illuminates the interactions between high-level languages and assembly code, and allows for consistency-aware runtime optimizations
- Formal demonstration that Tmi's runtime optimizations preserve required consistency model semantics
- A highly-compatible and low-overhead framework for adjusting per-thread virtual memory mappings in multithreaded Linux processes

The rest of this paper is organized as follows. Section 2 provides background on cache contention and previous online repair mechanisms. Section 3 describes Tmi's implementation on a standard Linux platform, and discusses its correctness. Section 4 presents an evaluation of Tmi's ability to detect and repair false sharing online. Section 5 discusses related work and Section 6 concludes.

2 BACKGROUND

Modern invalidation-based cache coherence protocols enforce a *single-writer multiple-reader* (SWMR) invariant which enforces that a given cache line can either be writable by a single core or readable by multiple cores. Cache contention occurs when two or more cores make repeated, conflicting accesses to a given line, *i.e.*, at least one of those cores writes to the line. The SWMR invariant causes these accesses to be serialized which is slow and energy-intensive.

There are two kinds of cache contention: *true sharing*, when two cores access overlapping bytes within a cache line, and *false sharing*, when cores access disjoint bytes. True sharing typically arises from lock contention, though contention on regular data is also possible. False sharing is more subtle, as it manifests whenever two frequently-accessed locations end up in one cache line, which can result from opaque compiler or memory allocator decisions.

It is generally more difficult to repair true sharing than false sharing, as true sharing requires some kind of application modification like switching to more scalable data structures. False sharing, however, can always be resolved by modifying memory layout in a semantics-preserving way by introducing padding or changing memory alignment.

2.1 HITM-Based Contention Detection

Intel multicore chips provide visibility into certain coherence protocol events which are useful for detecting cache contention. AMD chips provide a similar facility with the Instruction-Based Sampling mechanism [1] that records the microarchitectural events (e.g., cache misses) that instructions experience as they move through the pipeline. Because these events emanate directly from the hardware, they offer a direct language-agnostic view into the cache contention of a program that can uniformly detect true and false sharing alike. Tmi leverages Intel's Precise Event-Based Sampling (PEBS) performance counter mechanism [17], in particular the HITM[1] event

[1]We use the `MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM` PEBS event.

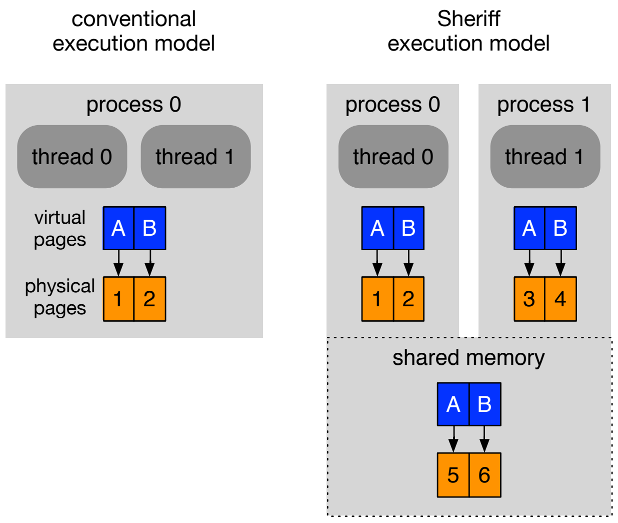

Figure 1: A conventional multithreaded execution model and Sheriff's model. With Sheriff, each thread is wrapped in a process. Each process maintains the same set of virtual pages, mapped to different physical pages in each process and in a shared memory region accessible by all processes.

introduced in the Sandy Bridge microarchitecture in 2011. This HITM ("HIT Modified") event occurs when one core's memory request hits in a remote core's private cache, and the line that is hit in the remote private cache is in the Modified state.

When an instruction *i* triggers a HITM event, the hardware generates a PEBS record which is logged to an in-memory buffer. Each PEBS record contains the virtual memory address accessed by *i*, and the values of the general-purpose registers as of *i*'s commit (including *i*'s PC). The PEBS record does not directly record whether the access was a load or store, but this information can be recovered by disassembling the PC (as we describe in Section 3.1). When the buffer becomes full of PEBS records, an interrupt notifies an OS driver to make the records available for userspace client programs. Recent versions of Linux provide good support for PEBS events via the `perf` API [37], which we discuss further in Section 3. Despite their name, PEBS events are not fully precise as the LASER [24] paper demonstrates. We have found similar results on our experimental platform, notably that the PC in a PEBS record is more accurate than the data address and that, while the name of the HITM event we use implies that only load instructions generate PEBS records, HITM events caused by stores generate PEBS records as well though at a lower rate than with loads. In addition to LASER, Intel's VTune profiler [10] and Linux's `perf c2c` utility [26] also measure PEBS HITM events, but do not provide a false sharing repair mechanism as LASER and TMI do.

2.2 Online Contention Repair

TMI's automatic false sharing repair is inspired by prior systems that replace threads with processes in order to enable process-private memory mappings. These systems use process-private mappings to provide efficient false sharing repair [19] and deterministic parallel execution [2, 20, 23, 29]. In a single process, threads have access

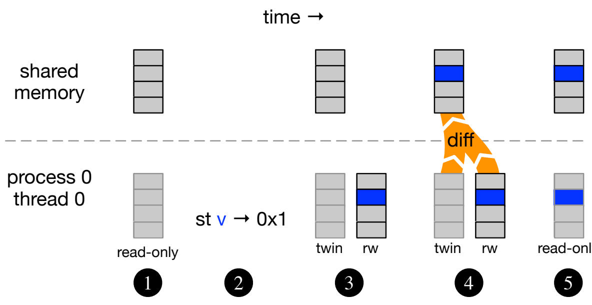

Figure 2: The Sheriff twin page mechanism.

to a shared address space with the same virtual-physical mapping for all threads. As discussed in the prior section, when threads access adjacent locations in physical memory, cache contention can occur. Unlike threads in a process, separate processes are not required to share the same mappings from virtual pages to physical pages. A process can freely map the same virtual pages as another process to a separate physical page. Figure 1 demonstrates how Sheriff maintains a thread-private physical page for each virtual page using processes. A shared memory region accessible by all processes is also maintained (with separate physical pages) to facilitate exchanging state between threads (as we discuss below). Running threads as processes and using process-private memory mappings eliminates false sharing as two threads that access nearby locations in the virtual address space (that would normally occupy the same cache line) will access distinct underlying physical pages, avoiding any cache line sharing.

While private virtual-memory mappings can protect against false sharing automatically, this execution model imposes both performance and compatibility challenges. Keeping memory state consistent across processes is expensive: it requires communicating each thread's memory updates to other threads when synchronization occurs. For programs that synchronize frequently, this can impose large performance overheads. For communicating memory updates to other processes, Sheriff uses a mechanism called *page twinning* to determine what memory a thread has updated. Figure 2 illustrates how twinning works. When a thread first begins executing, all of its pages are marked read-only ❶. Whenever a thread writes to a page ❷, copy-on-write is used to create a read-only twin of the initial version of the page, and a mutable copy of the page used for subsequent updates ❸. Whenever the memory updates of a thread need to be determined, the mutable pages are diffed with their initial versions to determine the bytes that have changed ❹. The twin pages are then cleared and mutable pages marked as read-only again to track subsequent changes ❺.

The page twinning store buffer (PTSB) approach is an efficient way of tracking memory updates, requiring essentially only the fixed costs of the twinning and diffing (paid once per page per synchronization operation) with zero additional per-access cost. However, a PTSB breaks the atomicity of an aligned multi-byte store instruction, treating the instruction as a sequence of single-byte stores leading to a violation of single-copy atomicity, sometimes also called "word tearing" [24]. Note that this notion of atomicity is distinct from the traditional notion of "store atomicity" in the

x is aligned on a 2B boundary
initially, x == 0

```
    Thread 0                Thread 1
store x ← 0xAB00       store x ← 0x00CD

             assert x != 0xABCD
```

Figure 3: A simple assembly program that reveals the semantics of aligned multi-byte stores. For all assembly languages we are aware of, the assert can never fail. However, it can fail with PTSBs.

consistency model literature[2] and for clarity we refer to our notion as aligned multi-byte store atomicity (AMBSA) throughout this paper.

Figure 3 shows a program that can break single-copy atomicity in the presence of PTSBs. The result arises because of the page diffing process, which cannot detect bytes that have been overwritten with an identical value. Thus, each thread's 2-byte store is seen as a 1-byte store. The updates are then merged and x receives a value that no thread has written. Note that merging must change *only* the bytes identified by the diff; updating other bytes is tantamount to fabricating stores that the program did not perform.

While PTSB semantics violate AMBSA they are nevertheless sufficient in very weak models like regular C11 or C++11 code [3] as we show in Section 3.4. However, PTSBs violate the memory consistency models of high-level languages like Java and of C/C++ atomic operations. Even more alarmingly, all of the hardware-level memory models we are aware of, including x86, SPARC, POWER and ARM [25, 33, 35, 36], guarantee AMBSA, *i.e.*, that the 2-byte stores to x are performed atomically because x is suitably aligned. Thus, a PTSB is incompatible with assembly languages. Inline assembly is fairly common in C and C++ programs, even if the program in question does not explicitly use inline assembly. Inline assembly appears in common glibc functions such as `memcpy`, and also in application code from `canneal` (for atomics), `dedup` (in the SSL library), and `leveldb` (for atomics).

Nevertheless, PTSBs have low performance overheads that allow for very efficient false sharing repair. Our insight with Tmi is that we can leverage PTSBs for regular C/C++ regions of code, while carefully handling other regions to preserve correctness. We next describe the implementation of the Tmi system and how it preserves correctness in the presence of inline assembly and C/C++ atomics.

3 THE Tmi SYSTEM

Although false sharing can impact the performance of a parallel application, not all applications suffer from false sharing, and some inputs to an application may be more sensitive to false sharing than others. For example, `histogram` exhibits a pattern of false sharing that is dependent on the image input to the application because different counters are incremented in each thread depending on the color of the input image's pixels. Aside from changing inputs, changes to the memory allocator, operating system environment, host hardware and nondeterministic thread scheduling can all impact the cache contention behavior of an application. Considering that many different factors may affect cache contention, several of which may be specific to a given execution, it is important that false sharing repair be always-on. Thus, instead of imposing drastic environmental changes on all programs (such as Sheriff's threads-as-processes execution model or Plastic's custom OS/hypervisor support), Tmi adopts a *compatible-by-default* approach. If no false sharing is detected for an application, the application should execute with minimal perturbation. If false sharing is detected, the repair mechanism should enable itself with as few invasive changes to the application as possible to avoid compatibility pitfalls. Tmi provides compatible-by-default execution in three ways: 1) by performing low-overhead false sharing detection via hardware performance counters, 2) by enabling the repair mechanism only once meaningful false sharing is detected and 3) by targeting repair only at those regions of memory that actually exhibit false sharing.

3.1 Low-Overhead Detection

Tmi leverages the Linux `perf` library to monitor HITM events for running applications. LASER also monitors PEBS HITM events, but because LASER requires a custom Linux driver, it is more work to port LASER to a new OS or architecture. In contrast, Tmi uses existing PEBS support within Linux and uses the existing Linux `perf` driver.

When a thread is created, Tmi establishes a `perf` event buffer for the thread that accumulates HITM events for that thread. Tmi also launches a per-application detection thread that consumes all HITM events produced by running threads and performs false sharing detection using the generated HITM events. Performing detection using the `perf` library only requires intercepting the `pthread_create` function to create the HITM event buffers and launch the per-application detection thread, and so has minimal overhead.

To improve the accuracy of false sharing detection, Tmi adopts similar strategies as those used in LASER [24]. To improve the accuracy of false sharing detection, the detection thread performs a few initialization tasks when it is launched. First, the detection thread references the `/proc/pid/maps` address map to filter memory addresses from system libraries and the stack. This restricts false sharing detection and repair to the heap and globals of the running application and any libraries that it has linked. Second, the detection thread analyzes the application binary using a disassembler to determine which instruction addresses are loads and stores, and to determine the width of each memory access. This information is used during false sharing detection to distinguish true sharing from false sharing. For example, if a 1-byte load to location $L1$ followed by 1-byte store to location $L2$ with $L1 \neq L2$ produces a HITM event, the false sharing detector would classify the HITM event as read-write false sharing.

The perf library provides a *period* parameter to control how often hardware events are recorded to the software buffer. A period of n causes records to be generated after (approximately) every n HITM events. With $n > 1$, multiple events to the same address appear as just one record. As shown in Figure 4, the sample period affects the overall performance and accuracy of HITM event recording. With

[2]The traditional notion refers to the fact that a single store may become visible to some processors before others.

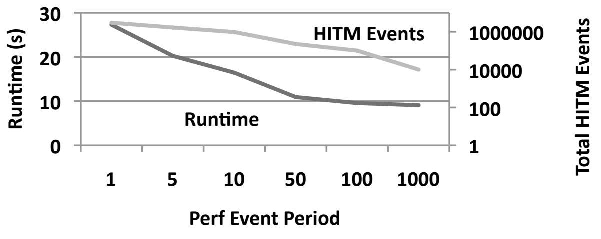

Figure 4: Performance and precision of HITM events reported by *perf* with various sampling periods on `leveldb`

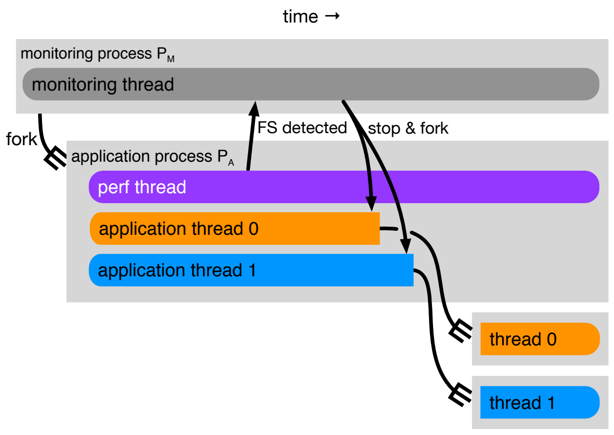

Figure 5: The process and thread organization of an application running with TMI.

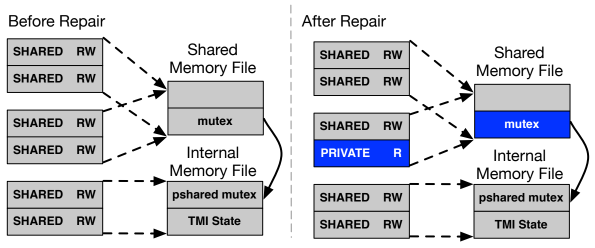

Figure 6: The shared memory layout before and after targeted repair with TMI as described in Section 3.2

a small period, the performance of some applications may suffer. However, a larger period leads to lower total event counts. To avoid under-reporting false sharing and true sharing, TMI assumes that if a period of n produces r records, each record corresponds to n/r actual events.

3.2 Making a Running Thread into a Process

To implement a PTSB, per-thread control over the virtual-to-physical memory mapping is required. We take inspiration from Sheriff and use processes to achieve this control from userspace. To provide compatibility-by-default, TMI does not convert threads into processes until false sharing has been detected. Once false sharing has been detected, TMI uses the `ptrace` library to inject a `fork()` call into the running thread to convert the thread to a process. TMI performs thread conversion via multiple steps, as shown in Figure 5.

At the beginning of the execution, a monitoring process P_M launches the application process P_A. P_M performs `ptrace` operations on P_A such as creating the `perf` detector thread (Section 3.1) and managing false sharing repair as described next. If the false sharing detector determines that repair is necessary, the detector thread signals P_M to bring all application threads in P_A to a stop (Figure 5). P_M stops the application threads in P_A by attaching to them with `ptrace`. After P_A's threads have been stopped, P_M converts each thread to a process. P_M retrieves and stores the current context (PC and registers) for each thread T. P_M then replaces the current context with the PC of a trampoline function that enables page protection and calls `fork()` to create a new process P_T. Once the trampoline function has finished executing in P_T, it signals P_M that it is ready to be resumed. P_M reinstalls the former context for thread T into P_T. P_M then detaches from the new process P_T and allows it to resume its execution at the same instruction address and with the same register and memory state that thread T had. Converting a running thread into a process is quite cheap in modern Linux: less than 1ms in our experiments.

To ensure that applications can transition between threads and processes seamlessly, P_A creates and initializes two memory mappings at program start that enable the operations required for the PTSB. P_A creates a shared memory region using `shm_open`. Stacks, globals and the heap all live in this shared memory region, allowing TMI to support stack sharing and repair false sharing no matter where it arises among a program's data. TMI creates two memory mappings (using `mmap`) to this shared memory region. The first memory mapping is always shared and allows both reads and writes to all pages. The second mapping is also initially process-shared but can later be remapped on a per-process basis to change the sharing and permissions on individual pages. For example, when TMI decides to repair false sharing on a page, its permissions in the second mapping are changed to be process-private (and therefore copy-on-write) and read only, allowing TMI to intercept writes to the repaired page.

TMI also creates a separate shared memory region that is always mapped as process-shared and writable (bottom of Figure 6). This memory region is used to allocate synchronization objects and other internal state for the TMI system. Synchronization objects must always be process-shared in case TMI needs to repair false sharing. To prepare for this possibility, all synchronization objects are allocated in process-shared memory, and TMI replaces synchronization objects in application memory with a pointer to the process-shared object. For example, under TMI, a `pthread_mutex_lock` struct contains a pointer to our own TMI-specific lock object which lives in process-shared memory. This pointer (and the lock it points to) are created by TMI's wrapper of `pthread_mutex_init()`.

3.3 Targeted Page Protection

Once the TMI detector has detected false sharing, and threads have been converted into processes, the TMI repair scheme aims to provide *targeted* false sharing repair for the running application. The first aspect of this targeted repair scheme enables the PTSB for only the specific pages that have been identified by the false sharing detector. This is in contrast with Sheriff [19] which enables a PTSB for all of virtual memory by default and with LASER [24] which enables a software store-buffer for targeted regions of code. With Tmi, accesses to pages that have not exhibited false sharing continue to access shared memory directly at native speed. For PTSB pages, Tmi prevents false sharing by forcing copy-on-write behavior for those pages. When a thread T resumes execution as a process P_T, any PTSB pages that it attempts to write to will be copied-on-write to a new physical page, preventing false sharing for all accesses to these protected pages (Section 2.2). Tmi commits a diff of PTSB pages at synchronization operations. By targeting the PTSB only to those pages where it is needed, Tmi keeps diff and commit costs low. The page size also has a significant impact on performance, which we evaluate in Section 4.4.

3.4 Consistency Model Issues

Tmi must deploy PTSBs judiciously to maximize their performance benefits while avoiding their semantic pitfalls. We discuss in this section how Tmi uses PTSBs correctly. First, we identify the conditions under which AMBSA violations are visible. Without loss of generality, we consider pairwise interactions between two threads, and assume that all synchronization is expressed via C++ atomics or assembly code.

Lemma 3.1. *For an assembly-free program, if AMBSA for a location x is broken there is a data race on x.*

Proof. Proof by contradiction: assume that AMBSA has been broken on x but the program is free of data races. Because there is no assembly code, race-freedom precisely characterizes the behavior of the entire program. We know that two threads t_0 and t_1 are both writing to x – with no or just one thread writing, diffing and merging preserve written values exactly.

Since the program is race-free, there must be atomic synchronization between the accesses. Assume that the synchronization uses a lock l to protect x, and that t_0 writes first. x's initial value is v_0. When t_0 writes to x, it writes a value v_1. When t_0 releases l, a diff operation will identify the bytes of x that t_0 changed and merge them into the shared memory space S. When t_1 acquires l, t_1 also empties its PTSB so that t_1 will access S for all locations. Thus, t_1 will see v_1. t_1 then performs its write to x, setting it to v_2. While t_1's write may have used its PTSB, when t_1 releases l it will merge the bytes of x that changed between v_2 and v_1 into S. Thus, the final value of $S[x]$ will be v_2. But v_2 is the value that t_1 wrote, so AMBSA has not been broken. Thus, breaking AMBSA requires a data race on x. □

3.4.1 Code-Centric Consistency. Code-centric consistency identifies points in a program's execution at which the memory consistency model changes, *e.g.*, because the source language has changed from C11 to x86 assembly. Code-centric consistency is a mechanism for 1) identifying where these semantic changes occur in the static code, and 2) adding callbacks at those points to allow appropriate runtime adaptation. A runtime system designed for code-centric consistency, such as Tmi, implements these callbacks to modify its behavior to, effectively, implement different consistency models at different points in time.

	regular	atomic	x86 asm
regular	1: undefined	1: undefined	3: unknown
atomic	1: undefined	2: atomic	4: unknown
x86 asm	3: unknown	4: unknown	5: TSO

Table 2: Semantics of concurrent conflicting accesses between different code regions. *undefined* semantics permit any program behavior, while *unknown* semantics have not yet been addressed in specifications. Shaded cells indicate where TMI permits PTSB use.

To argue for Tmi's correct use of PTSBs, we consider the different interactions that can arise between concurrent threads. We partition code regions into *regular*, *atomic* and *assembly* regions. Table 2 shows the semantics of concurrent, conflicting accesses between different regions, where concurrent conflicting accesses are defined as two accesses from two distinct threads, where at least one access is a write. This is very similar to the notion of a data race, but more general to handle the case of conflicting atomic operations which are defined to be data-race-free. From Table 2, there are five (numbered) cases of interacting code regions to consider, which we discuss below. Note that the case of data-race-free code is not represented in Table 2, but in such programs PTSB use remains invisible per Lemma Theorem 3.1.

Case 1: Undefined Semantics Lemma 3.1 shows that violations of AMBSA require a data race. In C++ a data race results in undefined behavior. Breaking AMBSA, and the use of PTSBs, is thus permissible in the presence of races. In addition to permitting PTSB usage, undefined semantics grant substantial flexibility to the implementation of the merge operation, *e.g.*, it does not require synchronization.

Case 2: Atomic Semantics All interactions between atomic operations are considered race-free, so undefined semantics are not permitted. As all atomic operations explicitly guarantee atomicity [15], AMBSA is required and PTSBs are not permitted for atomic code. Tmi guarantees that PTSBs are flushed before, and disabled during, each atomic region. Atomics operate directly on shared memory where the compiler's implementation of the C++ memory model ensures correctness. We make a finer distinction between `memory_order_relaxed` and stronger memory orders: `relaxed` does not require memory ordering, only atomicity. Thus, as long as `relaxed` atomics operate directly on shared pages, their atomicity semantics will be satisfied, and `relaxed` atomics needn't induce a PTSB flush.

As far as we are aware, existing memory consistency models do not specify the semantics of interactions between assembly code and regular code. We believe this is an important direction for future work as many real-world programs are multi-lingual with a mixture of, say, C++ and assembly. Next we propose informal semantics for these cases.

Case 3: Assembly Code and Regular Code In the spirit of the C++ consistency model, we can permit undefined semantics for conflicting accesses between regular code and assembly code. However, Tmi still flushes and disables the PTSB for uniformity with other cases, though undefined semantics make this unnecessary.

Case 4: Assembly Code and Atomic Code We adopt *non-undefined* semantics in this case. One can imagine a sophisticated lock-free algorithm where some cases are handled with atomics but others require the precise control of assembly. It is thus highly desirable to avoid undefined semantics in this case. Tmi ensures that PTSBs are disabled during all assembly-atomic interactions, so correctness is again provided by the C++ memory model (once it is extended to this case). This approach is conservative: a more sophisticated analysis of the assembly code within a given region could, along the lines of `relaxed` atomics, deem a PTSB flush unnecessary because, *e.g.*, the region operates only on thread-local data.

Case 5: TSO Semantics Interactions within assembly code are precisely specified by the semantics of the assembly language in question. None of the assembly languages we are aware of permit AMBSA violations. Thus, PTSBs are not permitted within assembly code.

3.4.2 Implementing Code-Centric Consistency. Code-centric consistency requires callbacks to identify the start and end of atomic and assembly regions. Identifying these regions in a binary is unfortunately impossible in general. `memory_order_seq_cst` loads on x86 are just plain loads, and assembly code regions are always inlined by construction. Instead, we have implemented a straightforward instrumentation pass in LLVM that adds the necessary code-centric consistency callbacks fully automatically. We note that recompilation is not strictly necessary. With access to source code, a simple static analysis could create a list of callbacks to insert, allowing existing binaries to be rewritten statically, or at load time via binary code probe injection [7] to avoid any overhead when Tmi is disabled.

In our current implementation, these callbacks are library function calls. The functions called are NOPs by default. Runtime systems that leverage code-centric consistency, such as Tmi, instruct the loader to replace the callbacks' implementations with runtime-specific versions. While we explore only one particular use of these callbacks in this paper, we envision them being useful for a variety of other software and hardware systems that operate on native code. Runtime systems that dynamically optimize native code, like DynamoRIO [6], could use our proposed callbacks to perform more aggressive optimizations while ensuring correct consistency semantics. A future processor could use these callbacks to relax consistency guarantees in the common case, providing stronger semantics only when necessary.

4 EVALUATION

We evaluate Tmi's false sharing detection and false sharing repair mechanisms. We demonstrate that Tmi is compatible with many applications and can perform false sharing detection at low runtime overhead. We show that for most benchmarks, Tmi's false sharing detector requires little additional memory relative to the baseline. On benchmarks with known false sharing, we show that Tmi is able to repair the existing false sharing and automatically improve the performance of the application. Additionally, we detail the runtime cost of enabling Tmi's false sharing repair mechanisms. We also demonstrate the performance tradeoff in enabling huge pages for both detection and repair.

4.1 Experimental Setup

We evaluate the Tmi system using two Haswell systems. For repair experiments, we use a 4-core Haswell system with 32 GB of memory and an Intel Core i7-4770K running at 3.4 GHz that matches the repair experiments presented by LASER [24], as it was not feasible to run LASER on another machine due to system-specific requirements for its HITM detector. For detection experiments, we use an 8-core Haswell system with 32 GB of memory and an Intel Core i7-5960X running at 3.0 GHz. We used gcc version 4.9.2 with -O3 optimizations on Ubuntu Linux 14.04. We present performance data as an average of 25 runs.

We evaluate Tmi on the Phoenix 1.0 [31], PARSEC 3.0 [4] and Splash2x [38] benchmark suites, and `leveldb` 1.20. We use the native inputs for Parsec and Splash2x. We use the largest available inputs for Phoenix and extend `histogram`'s inputs by 60x and `linear-regression`'s input by 100x 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 (`histogramfs`) 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 also evaluate three microbenchmarks that target the `boost` C++ library v1.62 (the latest version). `spinlockpool` targets a well-known false sharing bug in `boost::spinlock` [28]. `shptr-relaxed` and `shptr-lock` perform reference-counted smart pointer operations on one memory page while false sharing occurs on a separate memory page. The reference count updates are synchronized using either relaxed atomic operations (the default for modern platforms) or mutex locks, respectively. These microbenchmarks demonstrate how code-centric consistency enables sound performance optimizations in runtime systems like Tmi.

For all detection and repair experiments, we configured `perf` to use a sample period of 100 events. We use a modified version of the Lockless Allocator [14]. On our benchmarks, the Lockless Allocator outperformed the glibc allocator by 16% on average. We use the Lockless Allocator in our pthreads baseline to account for these speedups.

4.2 False Sharing Detection

Tmi's false sharing detector aims to have a low runtime overhead and to provide compatibility with existing parallel applications. Figure 7 shows the runtime overheads of *sheriff-detect*, *tmi-alloc*, and *tmi-detect* as compared to the baseline of pthreads execution using the Lockless Allocator. For the few benchmarks that it is compatible with, *sheriff-detect* shows the runtime performance of the Sheriff false sharing detection tool. This tool is incompatible with the native inputs for most of the benchmarks even with modified parameters to accommodate the additional required heap space. *sheriff-detect* requires threads to run as processes at all times and page protects all of memory, leading to incompatibility with many of the benchmarks. Overall, Sheriff works with just 11 of our 35 workloads.

tmi-alloc shows the overhead of replacing the Lockless Allocator's requests for system memory with memory from Tmi's process-shared memory. With huge pages enabled, *tmi-alloc* performs slightly better than the baseline Lockless allocator with a 1% speedup. *tmi-detect* includes *tmi-alloc* and additional overheads

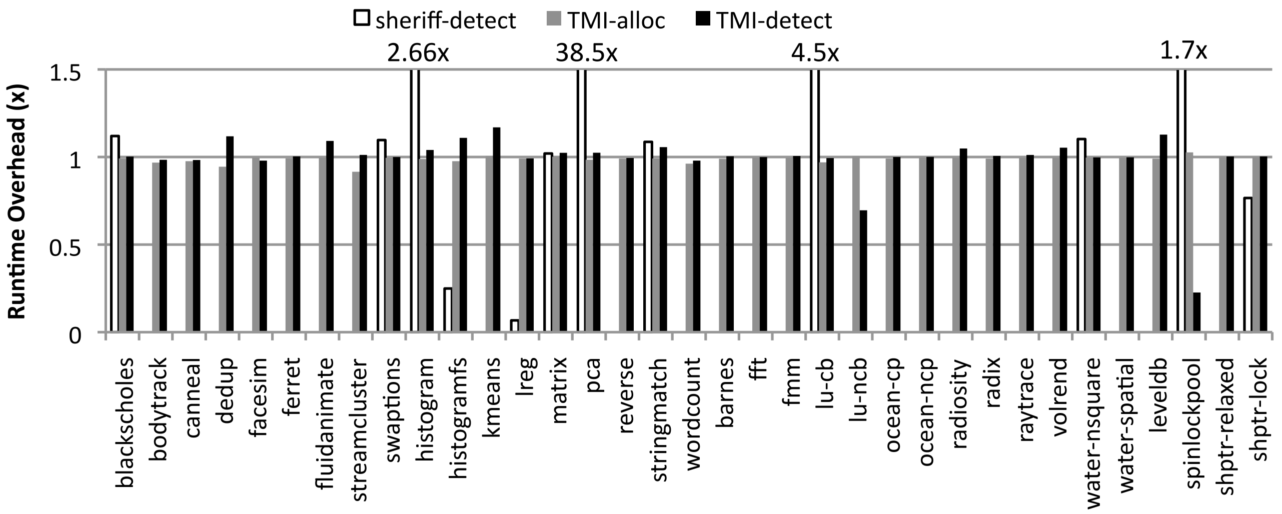

Figure 7: Performance of TMI's allocator and false sharing detection compared to sheriff-detect. All bars are normalized to pthreads execution using the Lockless allocator (lower is better). *TMI-alloc* redirects all memory alloctions to TMI's process-shared memory. *TMI-detect* enables all other necessary features of TMI's false sharing detector (Section 3).

including replacing thread synchronization with process-shared synchronization, monitoring HITM events using `perf`, and performing false sharing detection using the recorded HITM events. `dedup`, `kmeans`, `histogram`, and `leveldb` are all sensitive to perf's sample period parameter because each of these benchmarks contains either true sharing or false sharing. The runtime overhead of these benchmarks can be reduced by using a larger sample period (e.g. 1000) at the cost of lower accuracy detection, as demonstrated in Figure 4. Overall, *tmi-detect* exhibits an average runtime overhead of 2% with a maximum overhead of 17% on `kmeans`. Tmi's compatible-by-default design allows it to run with a wide range of benchmarks, including the large-scale open-source key-value store `leveldb`. Tmi detects minor amounts of false sharing in `leveldb` in a `std::deque` that is used as a queue for writes to the database. Unfortunately, the operations on this queue are heavily synchronized, and thus the performance gains from repairing this false sharing would be negligible. `leveldb` exhibits roughly 10x more HITM events attributable to true sharing rather than false sharing.

Figure 8 shows the memory overheads (on a logarithmic scale) for *tmi-detect* compared to the baseline memory usage. For most benchmarks, the memory overhead for Tmi is relatively small compared to the baseline. The Phoenix benchmarks and a few of the Splash2 benchmarks have small baseline memory usage around 10 MB. For these benchmarks, Tmi adds only about 90MB of memory overhead for the `perf` event buffers and false sharing detector data structures. Aside from these benchmarks with very small memory usage, Tmi requires 19% more memory than baseline execution. The majority of the memory overheads can be attributed to the false sharing detector's data structures. In order to perform accurate detection, the false sharing detector records disassembly information about the target application's static load and store instructions and records information about dynamic memory operations as well. `fluidanimate` and `water-spatial` both have high memory overheads due to synchronization because Tmi must replace (via an extra indirection) the fine-grained locks used by these benchmarks with process-shared locks. The memory overheads for `perf` stem from a per-thread buffer for HITM events. For the benchmarks in which Tmi repairs false sharing, additional memory overhead is required for twin pages and buffered page state, but these costs are small due to Tmi's targeted page protection.

4.3 False Sharing Repair

Tmi is able to automatically repair known false sharing bugs in the Phoenix and Splash benchmark suites. `histogram`, `linear-regression`, `stringmatch`, and `lu-ncb` all exhibit false sharing that leads to noticable performance degradation. All of these benchmarks contain structures that produce over 100,000 HITM events per second. `histogram` contains false sharing in thread-private histogram counters that can be located on the same cache-line. `linear-regression` repeatedly accesses an `args` array that is not 64-byte aligned by default. `stringmatch` accesses two thread-private structures, `cur_word` and `cur_word_final`, that can partially overlap on the same cache line. `lu-ncb` exhibits false sharing in the array input to its `daxpy` implementation. For `lu-ncb`, Tmi does not need to repair the false sharing because it is automatically repaired by changing the allocator. This behavior is common in all of the false sharing repair mechanisms that we tested. To evaluate Tmi's performance on real-world code, we injected a false sharing bug into Google's `leveldb` key-value store. By default, each thread maintains a local count of operations performed; in our buggy version these are packed into a single cache line. This bug is emblematic of many of the false sharing bugs we have seen in other programs.

On each of these benchmarks, Tmi is able to automatically repair false sharing to provide a performance benefit to the parallel application. Figure 9 shows the runtime speedup provided by Tmi on each of these benchmarks. Each column is normalized to the baseline runtime using pthreads and the Lockless Allocator. For each benchmark, we force the discovered false sharing behavior

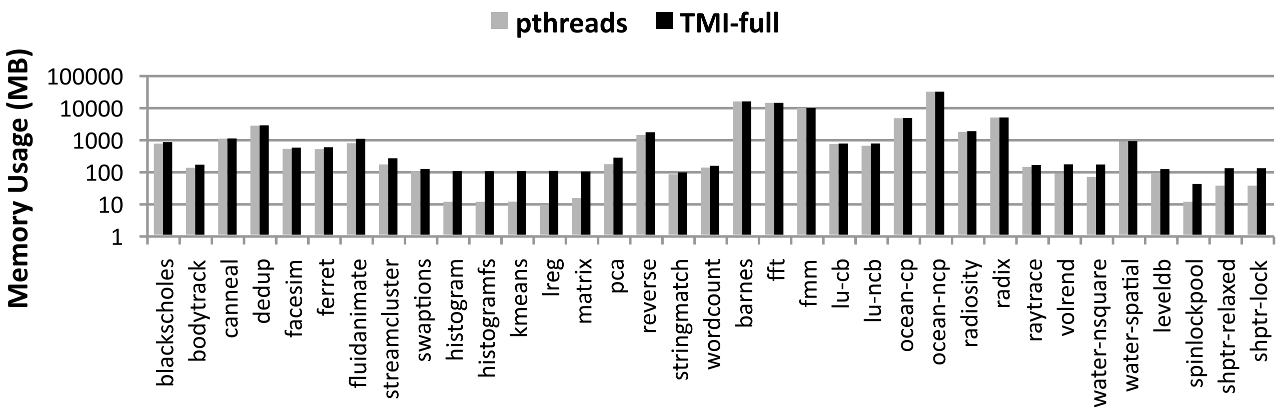

Figure 8: Memory overheads for TMI. Bars are absolute value in MB (log scale). Lower is better.

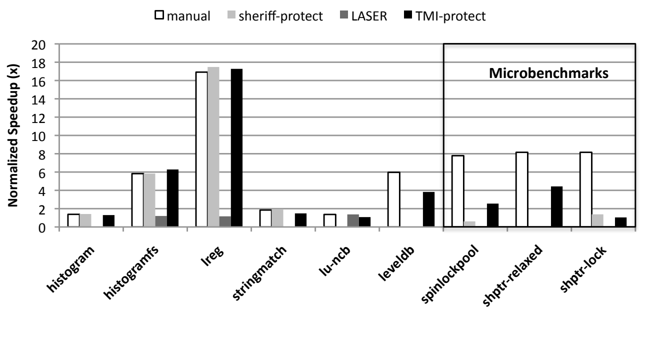

Figure 9: Speedup over pthreads (higher is better) for benchmarks where TMI automatically repairs false sharing.

by requiring a mis-aligned allocation when appropriate. *manual* shows the performance benefit of manually fixing the false sharing in each of the benchmarks. TMI provides significant speedups on benchmarks as well as a 3.8x speedup on the real-world `leveldb` program, providing 88% of the performance benefit of manual instrumentation completely automatically.

The *sheriff-protect* bars in Figure 9 show the runtime speedup provided by the Sheriff repair tool. In most cases, TMI performs similarly to Sheriff. Sheriff offers slightly higher performance benefits on some benchmarks because it begins preventing false sharing as soon as the application starts up whereas TMI does not begin preventing false sharing until it reaches a threshold specified by the detector. However, Sheriff suffers from compatibility issues due to this design decision and does not work on `lu-ncb`, `leveldb` or `shptr-relaxed`. LASER is able to repair false sharing on `histogramfs`, `linear-regression`, and `lu-ncb`, but it only attains 24% of the manual speedup on the benchmarks that it repairs. LASER is unable to provide a speedup on `histogram` (standard input) or `stringmatch` due to its expensive repair mechanism. LASER does not enable repair on the Boost microbenchmarks because LASER's TSO consistency is too restrictive to efficiently handle their frequent synchronization operations.

The right part of Figure 9 shows microbenchmark results. `spinlockpool` exhibits false sharing on an array of `pthread_mutex_locks` that are redirected by TMI in order to enable the conversion from threads to processes (Section 3.2). TMI adds a level of indirection to a new `pthread_mutex_lock` that is cache-line sized to avoid false sharing, automatically fixing the false sharing present in `boost::spinlockpool`.

`shptr-relaxed` exhibits false sharing along with occasional smart pointer manipulation, using C++ relaxed atomic operations to increment reference counts (Boost's default for modern platforms). Thanks to code-centric consistency, TMI is able to fully leverage these relaxed atomic operations, executing them without forcing a PTSB flush. Thus, the PTSB can continue to prevent false sharing on other memory pages not targeted by the relaxed atomics. TMI repairs false sharing in `shptr-relaxed` with a speedup of 4.43x.

`shptr-lock` demonstrates the overheads of fixing false sharing without code-centric consistency. In this version, the smart pointer reference counts are protected by a pthread mutex. This forces the PTSB to be flushed on every lock acquire and release, negating almost all of the performance benefits of fixing the false sharing. TMI exhibits a speedup of just 1.04x on `shptr-lock`. The performance gain in `shptr-relaxed` over `shptr-lock` demonstrates what can be achieved with the consistency-aware optimizations that code-centric consistency enables.

We examined the cost of applying false sharing repair to *all* program memory with code-centric consistency enabled, to better highlight store buffer overheads. For some benchmarks with false sharing, this PTSB-everywhere approach does not change performance. However, `histogram` and `histogramfs` suffered from additional performance overheads when the PTSB is used indiscriminately. `histogram` suffers a 36% slowdown with PTSB-everywhere, instead of a 29% speedup with TMI. `histogramfs` exhibits a 3.26x speedup with PTSB-everywhere but TMI achieves a 6.27x speedup instead. This shows that the PTSB-everywhere approach can have significant performance implications, which further motivates TMI's targeted false sharing repair.

Table 3 characterizes the false sharing repair performed by TMI. The "Unrepaired" column lists the application runtime before false sharing is detected. While TMI constantly receives HITM events,

App	Unrepaired (s)	T2P (μs)	Commits/s
histogram	1	73	1.25
histogramfs	1	112	0.79
lreg	1	138	0.65
stringmatch	1	114	0.64
leveldb	2	161	2.04
spinlockpool	1	93	0.38
shptr-relaxed	1	179	3.68
shptr-lock	1	179	33.99

Table 3: Characterization of TMI's false sharing repair

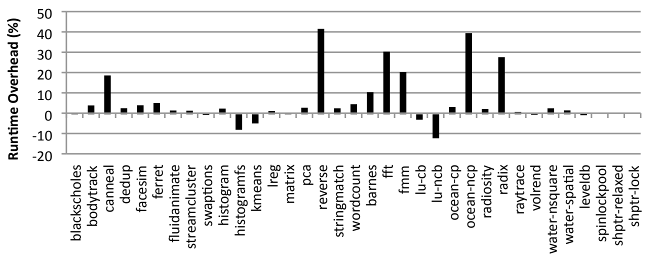

Figure 10: Performance overheads (lower is better) of using 4KB standard sized pages versus 2MB huge pages for process-shared, file-backed memory allocation

analyzing these events to find false sharing (Section 3.1) occurs once per second. Once false sharing is detected, Tmi converts threads to processes and begins to repair the false sharing. The amount of time necessary for this conversion is shown in the Threads-to-Process (T2P) column. As shown, threads can be converted to processes in under 200 microseconds for all applications. The Commits/s column shows the rate of synchronization events in each application, which cause the PTSB to commit. While commits are a prime source of overhead, Tmi is able to provide a speedup across a wide range of commit frequencies. However, in the pathological `shptr-lock` microbenchmark frequent commits limit the speedup to just 4%.

4.4 Huge Pages

Most standard memory allocators service memory requests with process-private memory from either `sbrk` or an anonymous (zeroed) `mmap`. Tmi's allocator provides memory from a shared and file-backed `mmap` in order to enable permission and mapping changes during execution. Unlike process-private anonymous mappings, shared file-backed mappings must carry their changes through to the underlying file. Using the default 4KB pages, we found that `canneal`, `reverse-index`, `fft`, `fmm`, `ocean-ncp`, and `radix` all exhibited a large number of page faults that lead to some performance degradation. These benchmarks all require relatively large amounts of memory, ranging from 1GB for `canneal` to 27GB for `ocean-ncp`. To reduce this overhead, we enabled huge pages using the `MAP_HUGETLB` and `MAP_HUGE_2MB` flags. By using huge pages, we reduce the number of page faults and reduce the pressure on the TLB. Figure 10 compares the runtime performance of using 4KB versus 2MB pages for Tmi's allocator. Enabling huge pages provides a 6% speedup over the standard page size.

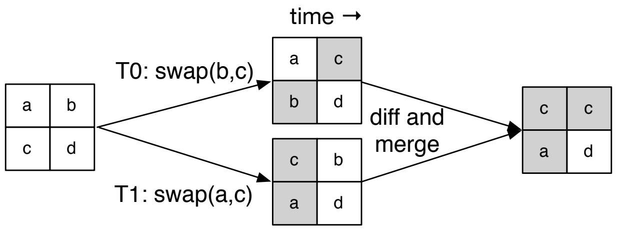

Figure 11: Atomic swaps in canneal that require code-centric consistency. In the pictured example, element c is replicated and element b is lost. (netlist.cpp:84)

$T0$	$T1$
`while (flag) {}`	`flag = false;`
`pthread_bar_wait();`	`pthread_bar_wait();`

Figure 12: Racy code in cholesky that executes incorrectly without code-centric consistency. $T0$'s version of *flag* never updates to true. (Simplified from mf.C:135-156)

Using huge pages has other performance ramifications for Tmi. When repair is enabled, Tmi must perform a diff and merge on each page that has been modified at each synchronization operation. With huge pages, this requires scanning and merging 2MB of memory instead of 4KB. We optimize huge page commit by comparing 4KB regions of the 2MB page using `memcmp` before comparing the individual bytes. While 4KB pages reduce commit costs by about 5x, commit with huge pages took 10ms or less on our benchmarks, so using huge pages is an overall win.

4.5 Code-Centric Consistency

As described in Section 3.4, Tmi preserves memory consistency semantics for assembly language and atomics by disabling the PTSB of the thread executing these kinds of code. Figure 11 demonstrates how atomic swaps can produce an incorrect final result in `canneal` in the presence of a PTSB. Without code-centric consistency, the use of a PTSB prevents the atomic swaps from immediately synchronizing to shared memory, which in turn leads to lost or duplicated elements in the grid. Figure 12 shows another case from `cholesky`, from the `splash2` suite, in which the use of a PTSB without code-centric consistency can lead to unexpected results. This code marks `flag` with the C volatile keyword, which is insufficient for correct synchronization behavior in the current C standard but was common practice in older C programs. Without code-centric consistency, the value of `flag` may never be updated in $T0$'s private memory, $T0$ will never exit the while loop, and the program hangs. Code-centric consistency can take account of the volatile keyword and provide the SC semantics that the original programmer intended. Even though there is technically a data race on `flag`, modern versions of gcc also eventually update `flag` to shared memory, allowing the code to terminate. Beyond correctness and performance, code-centric consistency also provides a good match for programmer expectations.

Aside from these two specific instances, we found inline assembly in multiple applications that might require code-centric consistency. `canneal` and `leveldb` contain 6 and 8 instances of inline assembly code, respectively, for their atomic pointer implementations. dedup contains 7 instances of inline assembly code from openssl. Multiple splash2 benchmarks use custom, flag-based synchronization.

Tmi does not identify significant enough false sharing in any of these benchmarks to trigger its repair mechanisms. The function hooks required to identify the start and end of assembly code regions are included in all our experiments and incur negligible performance overheads. On the simlarge input, *sheriff-detect* causes canneal to produce an incorrect result. *sheriff-detect* and *sheriff-protect* hang on cholesky. Tmi is able to perform false sharing detection on all of these benchmarks without causing incorrect results.

5 RELATED WORK

The most closely-related work to Tmi are systems that detect and automatically repair false sharing at runtime. The Plastic [30], Sheriff [19], and LASER [24] systems represent the state-of-the-art in false sharing detection and repair for unmanaged code. Remix [11] also performs online false sharing repair, but only for programs running on the JVM which provides significant extra flexibility.

The Plastic system [30] provides byte-granularity virtual-to-physical memory mapping, allowing adjacent virtual bytes to map to disjoint physical bytes. Plastic relies on custom OS/hypervisor support and dynamic binary instrumentation to implement this mapping. We were unable to obtain Plastic's source code for a direct comparison.

We explain the operation of Sheriff's [19] false sharing repair mechanism in Section 2.2 and provide an extensive comparison against it in Section 4.

LASER [24] performs false sharing detection in a similar manner to Tmi by using HITM performance counters. LASER also preserves the memory consistency semantics of applications by using a software store-buffer to repair false sharing. However, LASER's repair mechanism provides only a fraction of the performance benefit that Tmi can provide, as we show in Section 4.

Some schemes focus exclusively on false sharing detection, relying on programmers to implement the repair via source code changes. These systems typically rely on extensive program instrumentation, e.g., via full-system simulation [34], dynamic binary instrumentation as in Pluto [13] and Liu [18], memory shadowing along with dynamic instrumentation like the Dynamic Cache Contention Detection scheme [39] or extensive compiler instrumentation as with Predator [22]. Such instrumentation-based approaches typically incur significant performance overheads, but they can provide a clear view of the program's cache contention behavior. Predator even predicts false sharing behavior on alternate machines with larger or smaller cache lines. The Cheetah system [21] uses performance counters (different from those used by Tmi) to detect false sharing with low overhead and to predict the speedup of manual fixes.

Tmi's use of HITM performance counters for performance debugging is shared with prior work. Intel's VTune profiler [10] and Linux's `perf c2c` utility [26] use the same PEBS events as Tmi, but do not provide false sharing repair. The Plastic scheme [30] uses HITM counters (not PEBS events) to identify whether programs suffer from cache contention, but requires other mechanisms to localize the contention. [16] uses machine learning to correlate performance counter results with false sharing, but uses non-PEBS counters and provides no repair mechanism. In [12], HITM counts are used to determine samples for low-overhead data race detection. The TimeWarp system [27] uses HITM counts to infer the presence of software-based clocks that can be used to construct side-channel attacks even if access to high-resolution OS timers is disabled. If these prior schemes were extended to use PEBS HITM events, it is likely that greater accuracy could be achieved.

6 CONCLUSION

Tmi detects and repairs false sharing in parallel applications with low overhead and high compatibility. Tmi avoids the memory consistency pitfalls of prior approaches, and we present proofs of the correctness of Tmi's repair mechanism. Tmi introduces a novel technique for performing false sharing detection without page protection and enabling the repair scheme only when necessary.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback on prior versions of this work. This work is supported by the National Science Foundation through grant #1525296.

REFERENCES

[1] Advanced Micro Devices, Inc. 2013. *Preliminary BIOS and Kernel DeveloperâĂŹs Guide (BKDG) for AMD Family 16h Models 00h-0Fh (Kabini) Processors.* Chapter 2.6.2 Instruction Based Sampling.

[2] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism. *CoRR* abs/1005.3450 (2010). http://arxiv.org/abs/1005.3450

[3] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[4] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton, NJ, USA. Advisor(s) Li, Kai. AAI3445564.

[5] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10).* USENIX Association, Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924944

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03).* IEEE Computer Society, Washington, DC, USA, 265–275. http://dl.acm.org/citation.cfm?id=776261.776290

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2016. Living on the Edge: Rapid-toggling Probes with Cross-modification on x86. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/2908080.2908084

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. 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).* ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/2150976.2150998

[9] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13).* ACM, New York, NY, USA, 211–224. https://doi.org/10.1145/2465351.2465373

[10] Intel Corporation. 2015. Intel VTune Amplifier 2015. (May 2015). https://software.intel.com/en-us/intel-vtune-amplifier-xe

[11] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* ACM, New York, NY, USA, 251–265. https://doi.org/10.1145/2908080.2908090

[12] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. 2011. Demand-driven Software Race Detection Using Hardware Performance Counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2000064.2000084

[13] 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 (WBIA '09)*. ACM, New York, NY, USA, 26–33. https://doi.org/10.1145/1791194.1791198

[14] Lockless Inc. 2015. *Lockless Performance*. http://locklessinc.com/

[15] International Standard ISO/IEC 14882:2011. 2011. *Programming Languages – C++*. International Organization for Standards.

[16] 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 (SC '13)*. ACM, New York, NY, USA, Article 30, 9 pages. https://doi.org/10.1145/2503210.2503269

[17] David Levinthal. [n. d.]. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*. Intel Corporation.

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

[19] Tongping Liu and Emery D. Berger. 2011. 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)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048066.2048070

[20] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[21] 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 (CGO 2016)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2854038.2854039

[22] 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

[23] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2555243.2555252

[24] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris Newburn, and Joseph Devietti. 2016. LASER: Light, Accurate Sharing Detection and Repair. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*.

[25] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*. Technical Report. INRIA and University of Cambridge. https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

[26] Joe Mario. 2016. C2C - False Sharing Detection in Linux Perf. (September 2016). https://joemario.github.io/blog/2016/09/01/c2c-blog/

[27] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. 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)*. IEEE Computer Society, Washington, DC, USA, 118–129. http://dl.acm.org/citation.cfm?id=2337159.2337173

[28] mcmcc. 2012. false sharing in boost::detail::spinlock_pool? (June 2012). http://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool

[29] Timothy Merrifield and Jakob Eriksson. 2013. Increasing Concurrency in Deterministic Runtimes with Conversion. (2013).

[30] 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 (EuroSys '13)*. ACM, New York, NY, USA, 141–154. https://doi.org/10.1145/2465351.2465366

[31] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, Washington, DC, USA, 13–24. https://doi.org/10.1109/HPCA.2007.346181

[32] Mikael Ronstrom. 2012. MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012. (April 2012). http://mikaelronstrom.blogspot.com/2012/04/mysql-team-increases-scalability-by-50.html

[33] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

[34] Martin Schindewolf. 2007. *Analysis of Cache Misses Using SIMICS*. Master's thesis. Institute for Computing Systems Architecture, University of Edinburgh.

[35] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[36] David L. Weaver and Tom Germond (Eds.). 1994. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall.

[37] Vince Weaver. 2015. *perf events Library Man Page*. http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html

[38] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

[39] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '11)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/1952682.1952688