Leveraging Cache Coherence to Detect and Repair False Sharing On-the-fly

Vipin Patel
Indian Institute of Technology Kanpur
Kanpur, India
vipinpat@cse.iitk.ac.in

Swarnendu Biswas
Indian Institute of Technology Kanpur
Kanpur, India
swarnendu@cse.iitk.ac.in

Mainak Chaudhuri
Indian Institute of Technology Kanpur
Kanpur, India
mainakc@cse.iitk.ac.in

***Abstract*—Performance bugs due to false sharing do not manifest as observable correctness errors, and hence are challenging to detect and repair. Prior approaches aim to both detect and repair false sharing instances automatically but most of them suffer from one or more of the following drawbacks: (i) high performance overhead due to expensive tracking of shadow memory, (ii) reliance on imprecise hardware events, and (iii) limited applicability and portability.**

We present extensions to the MESI cache coherence protocol for efficiently identifying and mitigating false sharing instances. The *FSDetect* protocol tracks the frequency of coherence misses per cache block to identify harmful instances of falsely shared lines while incurring negligible performance overhead. The *FSLite* protocol extends FSDetect to transparently privatize the falsely shared lines on accesses after detection, thereby eliminating the performance problem arising from false sharing. FSLite maintains coherence by performing precise byte-level updates of privatized blocks at the LLC on termination of privatization.

Our simulation results on a variety of multithreaded workloads show that FSDetect can precisely identify *all* known harmful instances of false sharing. FSLite, on average, improves the performance of applications suffering from false sharing by 1.39X over the unmodified baseline, at the cost of a minimal increase in the chip area. Furthermore, applications running with FSLite stress the network less and show improved energy behavior.

***Index Terms*—coherence protocol, false sharing, shared memory, performance bugs**

I. Introduction

The invalidation-based coherence protocols used in multicore architectures enforces a single-writer multiple-reader (SWMR) invariant, which means that, at any given time, a cache line can be written by only a single core but can be read concurrently by multiple cores [1], [2]. When multiple cores try to update a cache line repeatedly, the updates get serialized. Resolving cache line contention to keep the cache hierarchy coherent incurs substantial overhead since it requires synchronous decision-making across multiple cores, and is a barrier to developing efficient and scalable shared-memory multithreaded applications. Contended lines in a cache-coherent architecture ping-pong between the private caches of the sharing cores, hurting performance and inflating communication traffic. Cache line contention arises from two types of read-write data sharing: *true* sharing and *false* sharing. True sharing occurs when threads on different cores access overlapping bytes in the same cache line, where at least one access is a write. True sharing can either be intentional, e.g., threads communicate through a shared variable, or unintentional, leading to data races [3]. False sharing arises when threads access disjoint bytes of the same cache line and at least one access is a write [4], [5].

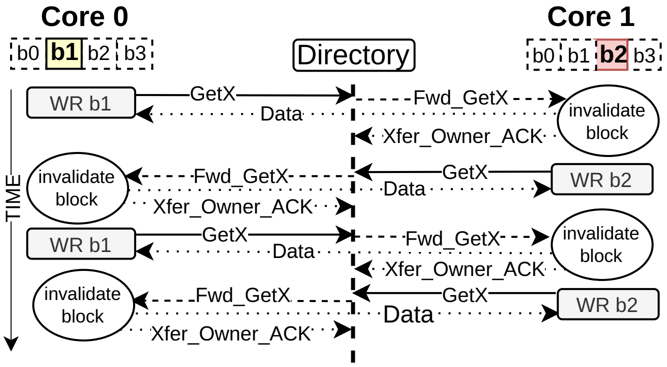


Fig. 1: A write-write false sharing instance resulting in repeated interventions (Fwd_GetX) across cores. WR a denotes a coherent write to address a.

Cache line contention arising from false sharing is almost always *unintentional*, is not fundamental to the application's correctness, and only serves as a performance bottleneck. Figure 1 shows an example of write-write false sharing where the sharing cores write to disjoint bytes b1 and b2 of a shared cache line. The figure shows frequent message exchanges under the MESI protocol [1], [2] because of repeated accesses to the falsely shared line by Cores 0 and 1. False sharing is expensive because of the (i) invalidation and intervention[1] penalties leading to remote fetch stalls in the associated memory operations, and (ii) cache and interconnect bandwidth wastage [6]. The presence of false sharing and its impact on application performance is sensitive to the application behavior (e.g., input and thread-to-core binding), the compiler toolchain, and the execution environment (e.g., cache line size) [7]–[10]. Two independent memory locations may end up in the same cache line due to optimizations by the compiler and the language runtime or because of the memory hierarchy

We acknowledge the support received from the SERB Grant SRG/2019/000384, TCS Research Scholar Program, and Research-I Foundation of CSE, IIT Kanpur.

[1] Intervention is a Get/GetX (read/read-exclusive) request forwarded by the directory to the owner core caching the requested block in E/M state.

configuration of the target machine [9]. For example, a known false sharing instance in the linear-reg benchmark from the PHOENIX suite [11] is unintentionally hidden by GCC at certain optimization levels, but not by LLVM [8]. Furthermore, false sharing can manifest dynamically with managed languages when copying garbage collectors move objects in the heap [12].

Although frequent false sharing can severely degrade performance and inflate on-chip interconnect traffic in cache-coherent multicore systems, the negative impact can go unnoticed [8], [13], [14]. Furthermore, manual resolution of false sharing can be very challenging for complex real-world applications (e.g., see Table I). The negative impact of false sharing on interconnect traffic can also be exploited by an attacker to launch denial-of-service attacks. Launching an attack requires running a multithreaded program with a very high volume of falsely shared blocks. Such a program would drive the interconnect bandwidth toward saturation, severely hampering progress of other co-scheduled processes. The presence of false sharing in several real-world applications (Table I) and in benchmark applications from the PHOENIX [11], SPLASH-2 [15], PARSEC [16], SynchroBench [7], and CCBench [9] suites highlights the need for automated detection and repair mechanisms [8]–[10].

TABLE I: False sharing detected in real-world applications

Linux kernel	[17]	MySQL and MariaDB	[18], [19]
Boost library	[20]	Lmax disruptor	[21]
OpenJDK	[22], [23]	Libdes	[7], [24]
Spin Checker	[7], [25], [26]	Netflix	[14], [27]

Impact of False Sharing: Figure 2 estimates the potential performance speedup that can be achieved when false sharing is manually repaired in a set of benchmarks. The results are obtained on an eight-core processor modeled using the gem5 simulator[2] [28], [29]. The processor model uses a two-level cache hierarchy with the L2 cache being shared across all cores. The private L1 caches are kept coherent using a directory-based MESI protocol. The average performance speedup is 1.34X, with RC enjoying a peak speedup of 3.06X. Furthermore, eliminating false sharing leads to an average reduction of 84% in the interconnect messages, and 25% in the energy expense of the cache hierarchy. The improvements from fixing false sharing can be *more* pronounced for longer-running applications that suffer from repetitive false sharing with cycles expended in false sharing episodes increasing more rapidly than those spent in non-false sharing portions of the applications. The amount of false sharing that manifests during the run time of an application can also vary with the compiler toolchain as we show later in Section VIII-B.

Existing Approaches: Although the aforementioned study considers manual repair of false sharing, it is difficult to identify and fix false sharing manually for complex multi-threaded applications [8], [13], [14], [30]. As a result, several techniques have been proposed to automatically detect and repair false sharing [4]–[10], [12], [30]–[36]. However, most existing approaches do not scale [4], [33], can have false negatives [4], [7], [8], [10], [30]–[32], require changes targeted to the language, compiler, and the runtime [8], [10], [12], need access to the application source and the software development process [8], [10], rely on fundamental changes to the core cache organization [6], [34], introduce complex speculation in coherence protocols [35], or can only detect false sharing but cannot repair on-the-fly [5], [7], [8], [32], [36], [37].

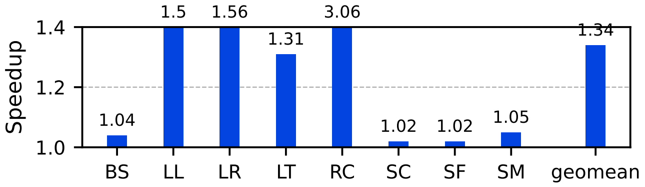


Fig. 2: Speedup achieved after manually fixing false sharing.

Our Approach: In this work, we propose practical architectural support for *online* detection and repair of false sharing. Given that the root cause of false sharing is the invalidation-based SWMR coherence protocol, it is natural to look for solutions in the coherence protocol layer only. We design *false-sharing-aware* coherence protocols without significantly deviating from the traditional core microarchitecture and cache memory design. The baseline is a cache-coherent multicore system with private caches, and all the cores share an inclusive L2 cache, which is also the last-level cache (LLC). A directory-based MESI cache coherence protocol [2] keeps the L1 caches coherent. We assume that the directory entries are embedded in the LLC tag array as extended tags/states, and follow cache-centric notation for memory blocks assuming that the directory is the owner (Section 8.6, [2]).

We propose *FSDetect*, a coherence protocol that helps track *more* harmful instances of false sharing (i.e., frequent false sharing that can potentially degrade performance). FSDetect maintains a per-core metadata array to track per-byte access information that helps distinguish between true and false sharing. FSDetect backs up the private access metadata in a shared metadata array associated with each coherence directory slice. FSDetect tracks the frequency of coherence misses to the shared blocks, and marks a line as potentially falsely shared when both the invalidation count and the fetch request count of the line crosses a threshold and no true sharing is detected. *FSLite* extends FSDetect by invoking the repair mechanism for the potentially falsely-shared lines. For repair, the coherence directory controller switches to a protocol extension that allows a requesting core to *privatize* a cache line flagged as falsely shared by FSDetect. The requesting cores can, therefore, operate on private copies of the line, thereby eliminating false sharing. Figure 3 shows the high-level steps of how FSDetect first detects a falsely shared block and after that, the block is governed by FSLite. Privatizing the contended cache block avoids future intervention and invalidation messages as long as there is no true sharing. FSLite maintains coherence when a privatized line is evicted from a private cache or privatization is terminated on certain conditions, e.g., a privatized line experiencing a true sharing.

[2] Section VIII provides details about the applications and the simulator.

We discuss the protocols and several optimizations and extensions in Sections III–VII.

We implement FSDetect and FSLite in gem5. Our simulation results show that FSDetect can precisely identify and FSLite can repair *all* known harmful instances of false sharing with minimal overhead (Section VIII). FSLite improves the run-time performance of applications experiencing false sharing by 1.39X on average over the unmodified baseline because it eliminates unnecessary cache line invalidations. The improvement with FSLite is significant for several applications (up to 3.9X for RC). Importantly, the speedup achieved by FSLite surpasses that of the manually fixed versions (shown in Figure 2) for a few applications as FSLite improves performance without inflating the working set size, thereby avoiding any negative impact on the cache performance. The volume of on-chip network traffic and the count of messages also reduce with FSLite, thereby placing fewer constraints on the network. FSLite reduces the energy expense in the cache hierarchy by 27%, on average.

Our results show that FSDetect and FSLite provide a competitive automated solution to the long-standing problem of false sharing while introducing modest overheads in terms of the protocol complexity and on-chip area. Our approach improves parallel software development productivity and avoids most drawbacks of prior work. Our solution requires no modification to the application source, the compiler toolchain, or the binary, does not inflate the memory requirement, is portable across software development toolchains, and leaves the basic private and shared cache organization unchanged.

Contributions: This paper makes the following contributions:

- a low-overhead coherence protocol, *FSDetect*, to identify harmful instances of false sharing,
- a repair protocol, *FSLite*, that fixes false sharing on the fly in a manner that is transparent to applications,
- practical architecture support for automated online detection and repair of false sharing, and
- an evaluation that shows compelling benefits from automated repair of false sharing instances.

II. Related Work

In this section, we briefly discuss related work for automatically detecting and repairing false sharing.

Detecting False Sharing: Much prior work has focused on *only* detecting instances of false sharing by using static analysis [38], dynamic instrumentation [8], [33], [39], runtime monitoring [4], [40], machine learning techniques [37], and simulation with architectural extensions [5]. Many recent techniques rely on accurate tracking of hardware performance counters (e.g., Intel HITM) to identify remote cache hits and use that to detect false sharing [7]–[9], [12], [30]–[32], [37]. *DeFT* is a hardware-based technique to automate detection of false sharing [5]. DeFT maintains per-core private metadata to track the last reader(s) and writer for each private cache line and to identify overlapped accesses from different cores. The metadata required in DeFT per L1 data cache line is double that of FSDetect, and the false sharing detection logic is per core. DeFT needs to communicate with a shared access tracking table on every load/store. Furthermore, DeFT requires post-mortem analysis to report false sharing instances and does not offer any automated repair mechanism. Only detecting false sharing has limited utility since it does not help with the observed efficiency of program execution.

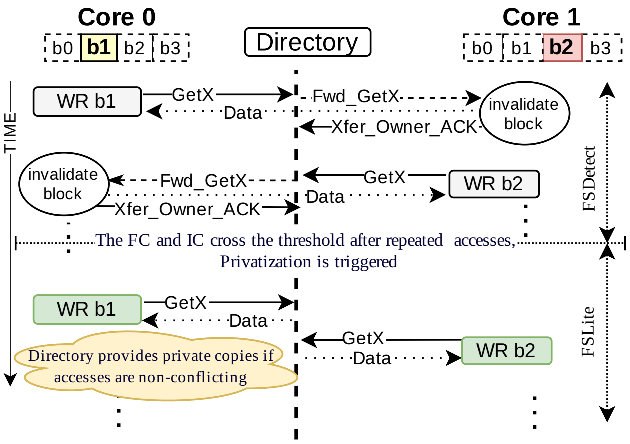


Fig. 3: Cores 0 and 1 continue to access disjoint block offsets (b1 and b2) after privatization without intervention overhead. FC=fetch request count, IC=invalidation and intervention count.

Detecting and Repairing False Sharing: There is a rich body of work to automatically detect *and* repair false sharing instances [4], [9], [10], [12], [30], [31]. Software techniques to repair false sharing use compiler instrumentation, program analyses, shadow memory tracking, and support from operating system or managed runtimes to transform the data layout by padding and moving locations and modifying the application binary [10], [12], [38], [40]–[42]. Compile-time approaches mostly target array-based applications with regular strided accesses, inflate memory footprint, require access to the application source, are tightly coupled with the language toolchain, and cannot generalize to the target architecture where the application will be run [38], [43].

Sheriff [4] transforms shared-memory threads to processes so that each thread updates its private copy of contended data. Sheriff cannot detect write-write false sharing, incurs high memory overhead because of page-level access tracking, and suffers from significant performance overhead in programs with frequent lock operations. *Plastic* [9] uses hardware performance counters to detect memory contention, and requires tight integration with custom hypervisors to resolve false sharing by remapping the contended locations to different physical memory. Plastic has a high overhead because of dynamic binary instrumentation. *Laser* [30] detects false sharing using the HITM events available in Haswell and newer architectures. All the stores in a basic block containing a contended cache line are performed using a software store buffer. *Remix* utilizes performance counter support to detect false sharing, and then pads class definitions to repair false sharing for managed languages like Java [12]. TMI is a userspace analysis tool that uses `perf` and `ptrace` utilities to detect false sharing, and uses Sheriff to privatize contended regions of memory dynamically [31]. *Huron* identifies false sharing with compiler instrumentation and repairs false sharing with memory layout transformations. Huron uses TMI during production runs to identify undetected false sharing instances. In summary, these software-based approaches do not scale [4], introduce suboptimal fixes [9], [30], inflates memory footprint [10], [12], can miss false sharing [4], can have false reports [4], are runtime-specific [12], or require extensive support and tight coupling with the system libraries and the software development process [4], [9], [10], [31].

Coherence decoupling [35], a hardware solution to address false sharing, introduces complex speculation in the coherence protocol to enable consumption of values from invalidated cache lines. It suffers from high volume of interconnect traffic arising from false-sharing-induced invalidations and interventions. Also, the core microarchitecture needs to be augmented with support similar to load value prediction and recovery from value misprediction.

Sub-block-grain coherence protocols that help reduce the impact of false sharing have also been studied [6], [34], [44], [45]. Minerva [44] is an adaptive sub-block coherence protocol that uses sector caches to support fine-grained communication (e.g., four bytes grain). However, Minerva requires synchronization among caches when multiple cores share the requested sub-block, and it is challenging to merge multiple copies of a word arriving from different sources. In general, sub-block coherence maintains coherence states at sub-block grain leading to a significantly more complex and bulkier protocol in terms of the state space. Sub-block coherence can solve false sharing only when falsely shared bytes fall on different sub-blocks. Our proposal has no such constraints. The *Protozoa* family of protocols supports multiple non-overlapping writers per cache block to reduce frequent invalidations from false sharing [6], [34]. These proposals require support for cache blocks of variable granularity and introduce complex changes to the existing cache hierarchy. Furthermore, false sharing can still arise when the size of sub-blocks that are invalidated is greater than one byte.

The scope consistency protocol [46] reduces cache misses and invalidations by invalidating only the pages associated with the current lock participating in lock-acquire operations. Such an approach enforces additional constraints to maintain synchronization variables in a specialized region and manage updates to shared variables from within critical sections.

Other Approaches: Memory allocators like Hoard [47] avoid heap allocations of thread-shared data on the same cache line. However, such memory allocators cannot *prevent* false sharing within a heap object or those caused by thread contentions due to poor programming or thread scheduling [37].

Self-invalidation-based coherence protocols aim to reduce the verification complexity of MESI-based protocols by relying on the data-race-free (DRF) assumption of language-level memory consistency models (e.g., Java [48] and C++ [49]) to enforce coherence only at synchronization operations [50]–[55]. A core self-invalidates its valid lines at an acquire operation and writes back dirty lines at a release operation. Although such approaches are competitive with MESI for applications that have false sharing, they often require additional hardware support and suffer from performance overheads for programs with frequent synchronization.

It is desirable to have an automated false-sharing detection and repair mechanism that overcomes the drawbacks of the existing proposals discussed in this section. Such a technique should be accurate, efficient, and portable across compiler toolchains. The approach should not require the application source, should not modify the application binary, and should be independent of the software development process.

III. High-level Design Overview

This section presents a high-level overview of our proposed false-sharing detection and repair protocols. Figure 4 shows the overall architecture. Later sections discuss the protocol and the architectural extensions in more detail.

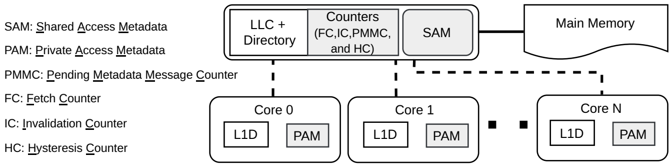


Fig. 4: Our proposed architecture (not according to scale). The new structures added by our proposal are shaded.

A. Detecting False Sharing

The FSDetect protocol precisely and efficiently detects cache lines that suffer from false sharing. FSDetect piggybacks on MESI transitions [1], [2] to track the frequency of fetch and invalidation/intervention requests needed to identify contended cache lines, and uses byte-granular access metadata to distinguish between true and false sharing. The protocol maintains precise access information in per-core private access metadata (PAM) table. When a core reads (writes) a byte of memory, the private cache controller updates the corresponding read (write) metadata in the PAM table. It is important to maintain the access metadata for a block as long as it is privately cached in *any* core in the multiprocessor. A shared access metadata (SAM) table associated with each LLC/directory slice maintains the access information for the blocks across all the cores. An entry in the SAM table is updated whenever the corresponding private metadata is received from a core. The directory uses the shared metadata along with the metadata in incoming messages to ensure that the accesses from different cores do not conflict. FSDetect maintains additional state per entry in the PAM and the SAM tables to limit metadata communication (see Section IV).

The key insight in FSDetect is that false sharing generates frequent invalidation and intervention messages along with high volume of fetch requests in a write-invalidation-based coherence protocol. FSDetect tracks the number of invalidations and interventions along with the number of fetch requests received by the LLC for a cache block B. Block B is marked potentially falsely shared when (i) there is no true sharing involving B and (ii) the number of fetch requests received by the LLC from any core for B and the total number of invalidations and interventions sent by the directory for B

exceed certain thresholds. Setting these thresholds high filters out the less impactful instances of false sharing. The FSLite protocol is enabled for potentially falsely shared blocks.

B. *Repairing False Sharing*

The false sharing problem can be alleviated if each core accesses the falsely shared bytes of a block in isolation, thereby avoiding unnecessary invalidations and interventions. The directory invokes the repair of a falsely shared cache block by initiating privatization of the block and informs the existing owner/sharer(s) that the block will be privatized. Any core that has the block in its private cache communicates the PAM table entry of the block to the directory slice and clears the PAM entry of the block. The directory checks for true sharing conflicts for all incoming private access metadata messages. In the absence of a true sharing conflict, the directory privatizes the cache block. The privatization procedure is aborted if a true sharing conflict is detected.

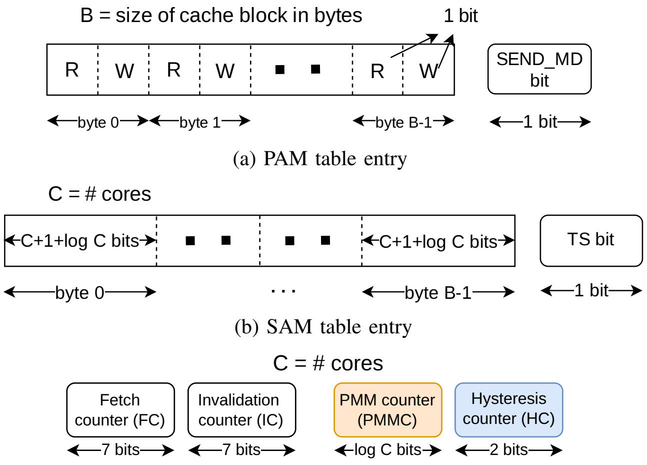
FSLite *continues* to check for true sharing conflicts at the granularity of individual bytes even *after* a cache block has been privatized. FSLite performs a conflict check whenever a byte offset of a privatized cache block is accessed for the first time after the block is privatized (determined from the PAM table entry). A privatized episode of a block is terminated if the directory entry, the LLC copy, or the SAM table entry of the block is evicted. At the end of a privatized episode of a block, the modified bytes from the privatized copied of the block are merged with the LLC copy of the block.

IV. FSDetect: Coherence Protocol to Detect False Sharing

PAM Table: FSDetect maintains metadata corresponding to reads and writes performed by a core in a per-core *private access metadata* (PAM) table. Each PAM table entry *corresponds* to an L1D cache block and maintains one read and one write bit per byte (see Figure 5a). A PAM table has 512 entries for a 32 KB L1D cache with 64-byte cache lines. A core's PAM table is connected to its L1D cache controller. On every load/store access, the L1D cache controller updates the bits corresponding to the cache line bytes touched by the core. The read and write bits can be maintained at a coarser grain than a byte to reduce the PAM table overhead.

FSDetect maintains a per-block SEND_MD bit in the PAM table to limit metadata communication on eviction of private blocks. On an L1D cache eviction, the corresponding PAM table entry is invalidated and the core sends the contents of the PAM entry to the directory if the SEND_MD bit is set in the entry. The directory acknowledges the receipt of the PAM entry. We assume MESI with silent evictions enabled. A 129-bit PAM table entry is organized as two separately accessible 8-byte segments (read and write bit-vectors) and the SEND_MD bit.

SAM Table: The *shared access metadata* (SAM) table associated with each LLC/directory slice maintains the last writer and a list of the readers for each byte of a cache block for the shared blocks across all the cores (see Figure 5b). For each byte, the SAM table uses C bits to track all readers and $\log_2 C + 1$ bits to identify the valid last writer, where C is the number of cores. Each SAM table entry also maintains a TS bit that is set when a true sharing involving any byte of the cache block is detected. For an 8-core system with 64-byte cache lines, a SAM table entry is $(8 + 1 + \log_2 8) * 64 + 1 = 769$ bits (i.e., $\sim$ 97 bytes) in size. This overhead can be reduced by maintaining the metadata at a coarser grain than a byte. Given that only a few lines in an application are falsely shared at a time, the SAM table implemented as a set-associative cache having 128 entries per 2 MB LLC slice and exercising LRU replacement works well (Section VIII-B). We discuss optimizations to reduce the width of a SAM entry in Section VI.



(a) PAM table entry

(b) SAM table entry

(c) Metadata per directory entry

Fig. 5: Schematics of (a) private access metadata entry, (b) shared access metadata entry, and (c) directory entry metadata.

A SAM table entry is read and written when metadata from the corresponding PAM table entry is received. A SAM table entry is invalidated at the end of each sharing episode and cleared at the beginning and end of a privatized episode of the corresponding block (see Section V).[3] An entry in the SAM and PAM tables is invalidated once the corresponding block is evicted from the LLC as per the inclusion policy. A SAM table entry may also get evicted due to space constraints.

Directory Metadata: Each directory entry maintains a fetch count (FC) and an invalidation count (IC) (see Figure 5c). We discuss the pending metadata message count (PMMC) and the hysteresis count (HC) in Sections V and VI, respectively. The FC of a cache block B tracks the total number of Get, GetX, or Upgrade requests received by the LLC from any core for B. The IC tracks the number of invalidations and interventions sent by the directory for B. FSDetect initializes FC and IC to zero when a block is filled into the LLC. The directory controller also resets both FC and IC of a directory entry if any of them saturates to the maximum possible value (127 for 7-bit FC and IC). In an 8-core system, each directory entry is extended by 19 bits to accommodate all the counters.

[3] A *sharing* episode of a block B starts from the time the first sharer core fetches a copy of B into its private cache and ends when all sharers/owner have/has evicted all copies of B from their private caches. Each LLC residency period of a block can experience multiple sharing episodes. A *privatized* episode of a block B starts when B moves to the privatized state and lasts till privatization is terminated or the application completes execution.

Metadata Maintenance: When a core C' requests for a block B cached in core C, the directory consults the TS bit in the corresponding SAM table entry to check whether the line has already experienced true sharing. In addition to sending an intervention or invalidation, the directory requests metadata from the owner (if state is E/M) or sharers by setting a spare bit (REQ_MD) in the header of the intervention/invalidation message if the TS bit is unset. On receiving an intervention/invalidation message, core C copies the REQ_MD bit from the header of the received message into the header of the intervention response or the invalidation acknowledgment that it sends to the requesting core C' and C' further copies the REQ_MD bit into the SEND_MD bit of B's PAM table entry. This allows core C' to decide whether to send the private metadata to the directory on a possible eviction of B in future. Core C also sends its PAM entry for block B to the home LLC/directory slice in a separate REP_MD metadata message if the REQ_MD bit was set in the header of the intervention/invalidation message. The REP_MD message carries the read and write bit-vectors as a 16-byte payload. On receiving an intervention due to a Get request, C copies the REQ_MD bit from the intervention message into the SEND_MD bit of the requested block's PAM entry. On the other hand, on receiving an intervention due to GetX or on receiving an invalidation message for block B from the directory, core C invalidates B and the corresponding PAM entry.

On receiving a REP_MD message, the directory controller updates the corresponding SAM table entry and sets the TS bit if true sharing is detected. A true sharing involving a byte offset $b \in B$ is inferred if either of the following two conditions holds: (i) b is marked as read-only in the currently received private metadata entry from core C, there is a valid last writer $C^{'}$ recorded in b's SAM table entry and $C \neq C^{'}$, or (ii) b is marked as written to in the currently received private metadata entry from core C, and (a) there is a valid last writer $C^{'}$ such that $C^{'} \neq C$, or (b) b has one or more reader cores recorded in the reader bit-vector of its SAM table entry and at least one reader $C^R \neq C$. Verifying these conditions does *not* require maintaining a full bit-vector of readers for each byte. However, a precise bit-vector is useful when reporting the complete set of cores involved in a detected instance of false sharing. We discuss a possible optimization in Section VI.

Example: Figure 6 shows an example to highlight the steps in on-the-fly detection of false sharing. The example assumes 4-byte cache blocks, and the numbers (in black circles) indicate the order of events. On Core C0's GetX request for accessing block offset b1 (❶), the directory increments FC by one and responds with the data block (❷). Core C1's Get request (❸) increments both the FC and IC by one. The directory sets the REQ_MD bit in the intervention message before forwarding Core C1's Get request (❹) to C0. On receiving C1's forwarded request, C0 responds to the directory with the data (5a). C0 also sends the PAM entry (5b) because REQ_MD was set, and the directory checks for conflicts and updates the relevant bits in the corresponding SAM entry of B to mark C0 as the last writer. Similarly, on the next UPG request from C0 (❻), the directory sets the REQ_MD bit before forwarding an invalidation request to C1 (❼). The directory updates C1 as a reader for byte b2 after C1 responds with the metadata (❽). The directory will identify block B to be *potentially* falsely shared when both IC and FC for the block exceed their trigger thresholds and the accesses from different cores do not conflict (i.e., TS bit is zero). High thresholds can filter out less impactful instances of false sharing. In our implementation, we set the thresholds for both FC and IC to the same value and refer to it as the privatization threshold τ_P. Once a block has been identified as potentially contended, the repair mechanism is invoked to mitigate false sharing.

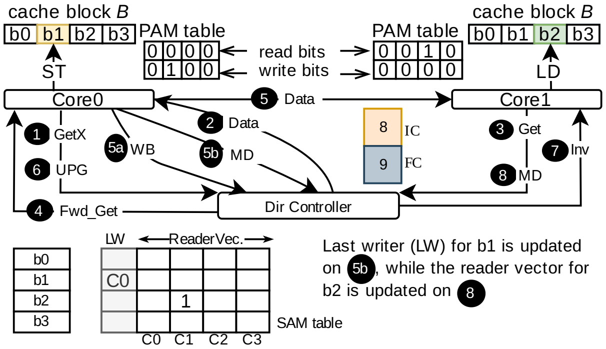


Fig. 6: Detecting false sharing instance in FSDetect protocol. UPG and MD denote Upgrade and REP_MD, respectively.

V. FSLite: Protocol to Repair False Sharing

The FSLite protocol automatically repairs false sharing by allowing multiple cores to access falsely shared bytes of a cache line in isolation. FSLite introduces a new stable state called PRV for a private cache block as well as directory entry. The PRV state indicates that multiple cores may cache the line marked PRV with write permissions, as long as the updates do not conflict. The updates to FC and IC to a block are disabled when it is in the PRV state. The FSLite protocol uses a pending metadata message counter (PMMC) in each directory entry (Figure 5c) to track how many metadata message responses the directory should expect. The PMMC is set to the number of owner/sharers every time the directory sends out a message that triggers metadata responses.

A. Initiating Privatization

The FSLite protocol triggers the privatization of a block when the directory receives a request for the block that has already been identified to be falsely shared. Figure 7 shows the four basic messages involved in privatization which in this case is triggered by a GetX request from Core C0 (❶). The existing owner/sharer(s) of the block is/are informed about this through a trigger privatization message TR_PRV (❷) and PMMC in the directory entry is set to the number of owner/sharers. If the block is in the M state in a private cache, the owner core sends a copy of the block to the LLC. Additionally, a private cache having a valid copy of the block in *any* state (e.g., Core C1 in Figure 7) sends the private access metadata of the block to the directory slice if the SEND_MD bit is set in the PAM entry (discussed earlier), and clears the PAM entry (❸). The

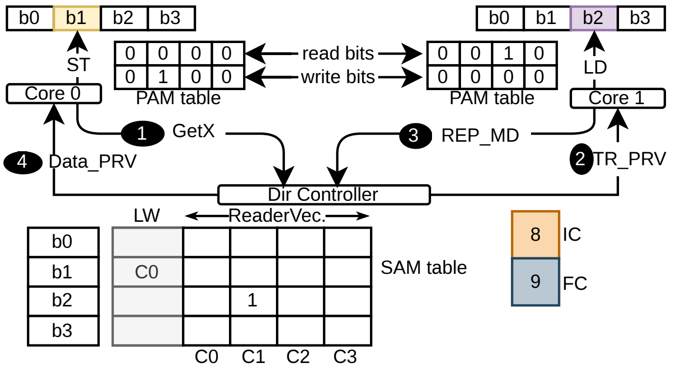


Fig. 7: Initiating privatization in FSLite.

directory controller waits for all in-flight metadata response messages to arrive (i.e., PMMC drops to zero). On receiving a metadata response message (REP_MD), the directory updates the SAM entry, checks for true sharing, and updates the TS bit in the SAM entry if a true sharing is detected.

Next, FSLite checks whether the *current* request (❶ in Figure 7) introduces any true sharing for the block, which requires knowing the bytes touched by the request. FSLite uses two spare bits in the message header to convey the number of bytes (four choices—1, 2, 4, or 8) touched by a GetX/Get/Upgrade. This information, in conjunction with the starting byte address (required for optimizations such as critical word first) of the request, enables the directory to compute the byte addresses touched by the request. Privatization is initiated for the shared block after all true sharing checks fail. The directory resets the SAM entry, updates it for the bytes accessed by the *current* request, and responds to the core which triggered privatization with a private copy of the data block using a Data_PRV message (❹). Prefetch requests are distinguished from the demand requests through message opcodes different from GetX/Get/Upgrade. For prefetch requests, the FSLite protocol takes the number of bytes touched to be zero. A privatized episode of a block initiated by a prefetch request or a wrong-path load/store does not lead to any correctness issue because the demand accesses to a privatized block are verified for correctness at runtime as discussed next.

B. Read/Write Requests to Privatized Blocks

Figure 8 shows how Get/GetX requests are handled *after* a block B is privatized. In the figure, assume that B is privatized because of repeated false sharing accesses to offsets b1 and b2 from cores C0 and C1. Suppose B is currently cached by cores C0, C1, C2, and C3 in the PRV state. A store operation to block B from C2 examines the read and write bits in the PAM table entry of B in C2. If at least one of the accessed bytes of B does not have the write bit set in the PAM table entry, the L1D cache controller sends a GetXCHK request to the directory (❶) along with the number of bytes touched in the message header. The directory checks whether each of the accessed bytes in B's SAM table entry satisfies one of the following conditions to rule out any true sharing: (i) the last writer is not valid and the reader bit-vector has at most one reader that is same as the current requesting core, or (ii) the last writer is valid and is same as the requesting core. If one of these conditions is satisfied, the directory responds to the requesting core with an acknowledgment (❷) and updates the last writer of the accessed bytes in B's SAM table entry. On receiving the acknowledgment, the core proceeds with the write and updates the relevant write bits in B's PAM table entry.

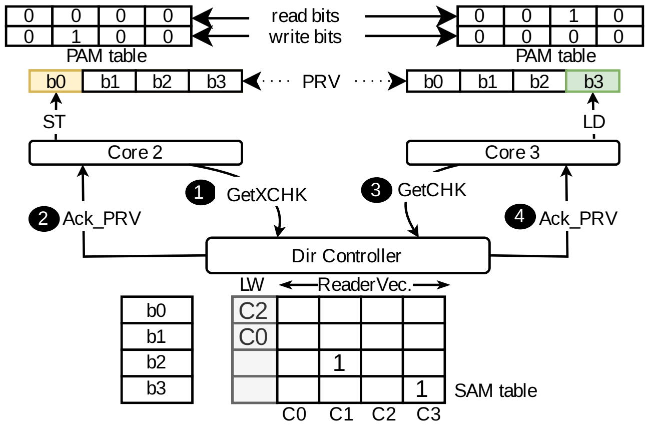


Fig. 8: Serving a request to a PRV block in FSLite.

The steps performed on a load operation to block B from C3 are similar. If neither the read bit nor the write bit is set in the PAM entry of B in C3 for at least one of the bytes touched by the operation, the L1D cache controller sends a GetCHK request to the directory (❸). This request, like GetXCHK, encodes the number of bytes touched in the message header. On receiving a GetCHK request for B, the directory looks up the SAM table entry of B and checks whether each of the accessed bytes satisfies one of the following conditions: (i) the last writer is not valid, or (ii) the last writer is valid and is same as the current requesting core. Satisfying one of these conditions confirms absence of true sharing. In that case, the directory responds to the requesting core with an acknowledgment (❹) and updates the reader bit-vectors of the accessed bytes in the SAM entry. On receiving the acknowledgment, the core proceeds with the read access and updates the read bits of the accessed bytes in the PAM entry of B. If, on the other hand, a true sharing is detected, the directory initiates a privatization termination sequence (discussed later).

Within a privatized episode of a falsely shared block B, only the first access (read or write) to a byte and the first write to a byte of B from a core having B in the PRV state see a two-hop critical path (requester to directory and back). This is significantly better than three hops incurred by *every* access to a falsely shared block in the baseline (requester to directory, directory to owner, and owner to requester). The blocking implementation of GetCHK can be improved through speculative forwarding of the load values to the dependents. The state checkpointing support that non-blocking GetCHK would require can also be used to implement non-blocking GetXCHK. We leave this exploration to future work.

The PRV state does not impose any change to the implementation of a fence instruction. Although the stores done to a privatized block B by a core C are *not* made visible to other cores when core C executes a fence instruction, any attempt by another core C′ to access the bytes of B modified by C leads to termination of the privatized episode of B and core C′ receives the latest values of these bytes. Since any true sharing automatically exposes the stores done by all the cores to a privatized block, the fence instruction can remain oblivious to the PRV state.

The loads and stores to a privatized cache line from a core are not visible to other cores, but the bytes of the privatized line accessed by different cores are disjoint. As a result, within a privatized episode of a block, the loads and stores to the block from different cores can be interleaved arbitrarily to form a legal total order. All these possible total orders are equivalent to a total order that is legal for the baseline memory consistency model as long as the baseline partial order within each core remains unaltered. Since GetCHK and GetXCHK operations are blocking, the partial order within a core conforms to the baseline partial order. In summary, FSLite does not alter the guarantees of the baseline memory consistency model.

C. Terminating Privatization

The directory controller initiates the termination of the privatized episode of a falsely shared block if any of the following holds: (i) there are conflicting accesses to the same byte of a privatized block by two or more cores, (ii) a directory entry or LLC block in the PRV state is evicted, (iii) the SAM table entry of a privatized block gets evicted, or (iv) an access to the privatized block is forwarded from another socket.

Conflict Detection: Figure 9 shows the termination of a privatized episode of a shared block B by the directory on detecting an access conflict. The figure shows a 4-byte block falsely shared among four cores (each core accesses the block offset using its core ID). A true sharing conflict is detected when a GetCHK or a GetXCHK request fails to satisfy the necessary and sufficient conditions of conflict-freedom, as already discussed. In Figure 9, Core 0 attempts to write to a previously untouched offset b1 triggering a GetXCHK request (❶). In such a situation, the directory controller initiates termination by sending a special invalidation message Inv_PRV (❷) to each of B's sharers, caching B in the PRV state. Each sharer responds by sending B to the home LLC slice (Prv_WB, ❸) and invalidating it from its private cache. On receiving a response from a core, the directory controller updates the LLC block at the byte positions where the last writer matches the responding core. After the LLC block is updated with all the responses, the SAM entry of B is cleared and the IC and FC counters are reset. This completes the switch-over to the FSDetect protocol. Finally, the directory responds to the original GetCHK or GetXCHK request by treating it as a traditional Get or GetX request, respectively.

Eviction of a Directory Entry or LLC Block: When a directory entry or an LLC block is evicted, all privately cached copies are invalidated as per the inclusion requirement, and the privatized episode of a block is terminated. When an LLC block in the PRV state is evicted, it is copied into a write buffer, as is done on eviction of a dirty block. As and when responses from the sharers arrive, the appropriate bytes are updated in the write buffer. When all updates are done, the block from the write buffer is sent for writing to the next lower level of the memory hierarchy, and the SAM table entry of the block is invalidated.

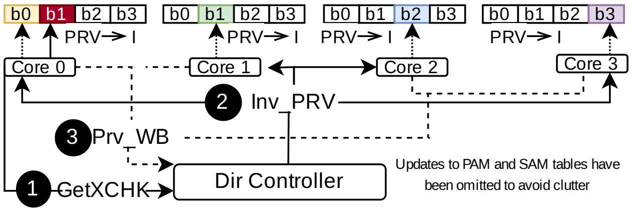


Fig. 9: Terminating privatization due to access conflict.

Privatizing a falsely shared block increases the chances of its directory entry or the LLC block getting aged and evicted because the accesses can now happen within each core without requiring directory or LLC access. We can reduce the chances of premature termination by devising a replacement policy that attaches a low eviction priority to the directory entries/LLC blocks in the PRV state. The controller can evict a block in the PRV state only if all the ways in a set are occupied by privatized lines. We do not implement this optimization.

Eviction of SAM Table Entry: Eviction of the shared access metadata of a privatized block results in losing the access history of the block and can lead to incorrect execution due to the inability to detect true sharing instances in future. Therefore, the directory controller terminates privatization once the SAM table entry of a privatized block is evicted. The privatized copies of the block are retrieved and the LLC block is updated at the appropriate byte positions.

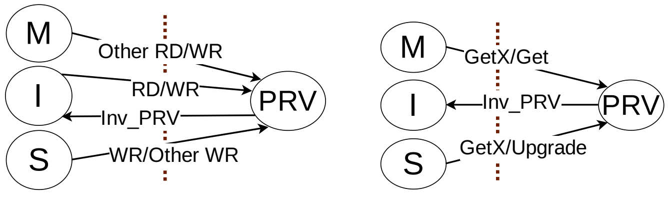
Access from External Socket: Let us suppose socket S caches a block B which is privatized. Since this socket is necessarily the owner (M state holder) of this block, an access to B from another socket S' is forwarded to S by the inter-socket coherence directory. Before responding to S', the privatized episode of B must be first terminated in S and then the updated block is used to respond to S'. However, if FSLite is also extended to the inter-socket coherence protocol, the request from S' would be handled by an inter-socket table similar to the intra-socket SAM table and privatization in S may not be terminated (see Section VII).

D. Other Protocol Details

Eviction of Privatized Blocks: When a core evicts a block in the PRV state, it sends a usual writeback message to the home LLC/directory slice. On receiving a writeback message to a block in the PRV state, the home LLC controller reads out the block from the LLC slice. In parallel, the directory controller accesses the SAM entry of the block and compares the last writer of each byte of the block against the ID of the evicting core to prepare a bitmask for updating the relevant bytes of the LLC block. The block read out from the LLC slice is updated using the block in the writeback message at the byte positions where the last writer matches the evicting core. The updated block is written back to the LLC slice. On successful update of the LLC slice, the directory removes the evicting core ID from the list of sharers and the last writer.

Phantom Message: A private core may receive an invalidation or intervention for a line when the line has already been evicted from the L1D cache and the corresponding entry has been invalidated in the PAM table. As an example, consider a core C having a block B in the M state. The directory receives a GetX request from core C′ for B. The directory finds that the TS bit for B is zero, increments PMMC of B, and sets the REQ_MD bit in the header of the intervention message before forwarding the request to C. Meanwhile, core C evicts block B and issues a writeback to the LLC for B before receiving the intervention. On receiving the late intervention, core C forwards the data for block B from its local writeback buffer to C′, but the metadata for B no longer exists in the PAM table for C. In such a scenario, core C sends out a dataless *phantom* message notifying the directory that the core no longer caches the block and metadata. The directory uses the phantom message to only decrement PMMC and does not update the SAM entry of B.

Protocol Modifications: Figure 10 shows the state transitions for a private cache block and the corresponding directory entry into the PRV state after the block has been identified as falsely shared and state transitions out of the PRV state when the privatized episode of the block is terminated. A private cache block B can switch to the PRV state in three scenarios (Figure 10a): (i) B is in the M state and a read/write from another core triggers privatization of B, (ii) B is in the S state and a write from another core or the core caching B triggers privatization, and (iii) a core reads/writes to bytes of an invalid line B that is already in PRV state in another core. Block B transitions to the I state when privatization terminates. A directory entry transitions into the PRV state on an invalidation or an intervention, provided the conditions for conflict-freedom are satisfied (Figure 10b).



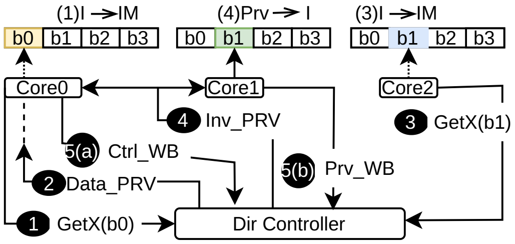
(a) Private cache line (b) Directory entry

Fig. 10: Transition diagrams of FSLite.

E. Protocol Races

In this section, we briefly discuss two scenarios to show how FSLite handles protocol races.

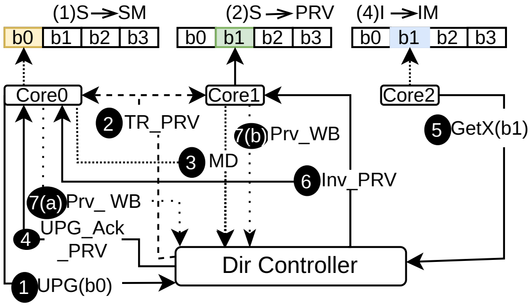
Figure 11 shows a race involving accesses to a block B privatized in core C1. Core C0 issues a GetX for B (❶). The directory does not detect a conflict and responds (❷). A new request is sent by C2 (❸). The directory detects that C2's access conflicts with C1's access, and initiates termination by sending out invalidations (❹) to C0 and C1. Suppose ❹ reaches C0 before the response ❷. C0 responds with a special Ctrl_WB message 5a as B is not present in the cache. C0 must reissue the request on arrival of response ❷. Similarly, for a Get request, the load will be reissued by the L1D cache controller on detecting a similar race.



Updates to PAM and SAM tables have been omitted to avoid clutter

Fig. 11: Race between GetX response and invalidate privatization request.

Figure 12 depicts a protocol race involving an upgrade request. Core C0 issues an upgrade (❶). Suppose C0's request triggers privatization of the requested block, and the directory sends the initiate-privatization message (TR_PRV) to C0 and C1 (❷). Both the cores respond with metadata messages (❸). The directory performs a conflict check for the upgrade request of C0, and sends an upgrade acknowledgment with privatization (❹) on detecting no conflict. On C2's conflicting request (❺), the directory sends invalidations to terminate privatization (❻). Each sharer responds with writeback messages to the directory (7a and 7b). The directory merges the copy of the cache line received from each core and updates the line in the LLC. If the response ❹ to C0 is delayed and reaches after the invalidation (❻), the Upgrade will be reissued by C0 as a GetX.



Updates to PAM and SAM tables have been omitted to avoid clutter

Fig. 12: Race between Upgrade and terminate privatization. UPG and MD denote Upgrade and REP_MD, respectively.

VI. Design Optimizations

Data Initialization: In many applications, often the main thread initializes the data (e.g., an array of work items). Later, the worker threads repeatedly read and write to their partitions of the shared arrays leading to false sharing. However, in such a scenario, the FSDetect protocol correctly detects a write-write true sharing between the *only* write by the main thread during data initialization and the subsequent falsely shared updates by the worker threads, thereby losing the opportunity to privatize such cache lines. In general, an application may exhibit a phased behavior where a short-lived true sharing episode on a cache block is followed by a substantial stretch of falsely shared accesses. To be able to successfully privatize falsely shared blocks in such scenarios, FSDetect periodically resets a private or shared metadata entry including the TS bit whenever either FC and IC both cross a threshold τ_{R1} or FC attains a value of τ_{R2}. We set $\tau_{R1} = 16$ and $\tau_{R2} = 127$ for our simulated system. In general, these values may require tuning.

Optimizing the SAM Table Size: Maintaining a precise reader list per byte is the primary contributor to the size of the SAM table. For a system with C cores and B-byte cache blocks, the basic design requires $C \times B$ bits to maintain information about all the reader cores for each byte of a block to primarily identify read-write true sharing for a block. However, a read-write true sharing for a byte can be identified without the knowledge of precise IDs of the reader cores involved in the sharing. For each byte, it is sufficient to maintain the ID of the last reader along with a valid bit and an "overflow" bit indicating the presence of any reader other than the last reader. The overflow bit for a byte is set when a core C reads the byte and there is already a valid last reader different from C. A read-write true sharing is detected on a write from core C to a byte if and only if the byte has a valid last reader different from C or the overflow bit is set for the byte. Thus, we can replace C bits by $(\log_2(C) + 2)$ bits per byte in the SAM table entry. This optimized SAM table entry is 577 bits wide as opposed to 769 bits in the basic design leading to a 25% storage saving in an 8-core system with 64-byte cache blocks. The only drawback of this optimization is that FSDetect cannot report the precise reader core IDs involved in a false sharing instance, although it continues to correctly identify all harmful false sharing instances.

Hysteresis counter: A cache line can experience repeated interspersed episodes of false and true sharing. In such scenarios, FSDetect repeatedly identifies the line as falsely shared, FSLite privatizes the block, but the short-lived privatized episode of the block gets terminated when FSLite detects true sharing. The overhead of frequently triggering and terminating privatization for such cache lines eclipses the performance benefit of privatization. We introduce a 2-bit saturating hysteresis counter (HC) with each directory entry (Figure 5c) to address this issue. HC is initialized to zero, incremented whenever a true sharing conflict is detected with $\text{TS} = 0$, and decremented whenever both FC and IC cross the privatization threshold provided $\text{HC} > 0$ and $\text{TS} = 0$. The privatization sequence for a block is initiated if (i) $\text{TS} = 0$, (ii) $\text{HC} = 0$, and (ii) both FC and IC cross the privatization threshold τ_P. If both FC and IC cross the privatization threshold for a block B, but the privatization sequence cannot be initiated for not meeting the aforementioned conditions, the SAM table entry as well as FC and IC of B are reset so that the most recent access metadata can be gathered for B.

VII. Discussion

Utility Beyond False Sharing: The proposed microarchitecture support for detecting and repairing false sharing can be extended easily to carry out other shared memory analyses and optimizations. First, FSDetect can be easily adopted for identifying contended true-shared lines. Since cache lines accommodating synchronization variables fall in this category, FSDetect can identify and report contended synchronization variables. Second, FSDetect, with simple extensions, can identify region conflicts and data races [56]–[60]. Third, cache line privatization in some form has been employed to accelerate commutative and associative parallel reduction operations [61]–[64]. The privatization component of FSLite can be augmented with a set of reduction merge operations in the LLC controller to efficiently carry out these parallel reduction operations.

Support for Three-level Cache Hierarchy: The presentation of our proposal so far assumes a two-level cache hierarchy. A typical three-level cache hierarchy has a large mid-level (L2) cache ranging in capacity from 256 KB to more than a megabyte. We implement a PAM table connected to the L1D cache controller as in a two-level cache hierarchy. Since a block is not operated on by load or store operations while it resides in the L2 cache, we do not need to implement an access metadata table for the L2 cache blocks in a three-level cache hierarchy. When an L1D cache block is evicted, the corresponding PAM table entry is invalidated and the contents of the invalidated entry are communicated to the SAM table of the target LLC (in this case the L3 cache) slice. The SAM table controller updates the corresponding SAM entry using the received PAM entry. Thanks to the SEND_MD bit, we observe that the traffic generated due to communication of the evicted PAM table entries is only about 3% of the total L1D to LLC traffic generated in the baseline MESI system with a two-level cache hierarchy. This PAM table eviction traffic will remain unchanged in the three-level cache hierarchy as well assuming that the L1D cache is similar in both hierarchies. All other aspects of our proposal remain unaffected in a three-level cache hierarchy.

Support for Sparse Directory: The implementation of our proposal assumes an in-cache directory for simplicity. However, a space-efficient baseline design would implement the directory as a set-associative cache, usually referred to as the sparse directory. Our proposal does not rely on the actual organization of the directory and therefore, continues to work seamlessly in the presence of a sparse directory. On eviction of a sparse directory entry, when the copies of the block being tracked by the evicted directory entry are invalidated from the private caches, the corresponding PAM table and SAM table entries must also be invalidated.

Support for Non-inclusive LLC: The primary difference between a non-inclusive and an inclusive LLC is that in the former an entry found in the sparse directory may not have the corresponding block cached in the LLC. Our proposal, except in one scenario, does not rely on the presence of a block in the LLC when its entry is found in the sparse directory. The only one exception arises at the end of a privatized episode of a block when the modified bytes from the privatized copies of the block are merged with the LLC copy. If the LLC does not have a copy of the block, the first writeback of a privatized block allocates the block in the LLC. The rest of the protocol is unchanged.

Extension to Scalable Multi-socket Systems: Our intra-socket coherence protocol extensions can be easily incorporated in the inter-socket directory protocol of a scalable multi-socket system. That would enable detecting and repairing intra-socket as well as inter-socket false sharing instances. Such a design would require a metadata access table attached to the inter-socket coherence directory. This table can be implemented as an SRAM cache embedded in the memory controller. An evicted SAM table entry from a socket can be communicated to the inter-socket metadata access table for identifying the cross-socket contended cache blocks.

VIII. Evaluation

A. Simulation Environment

We implement FSDetect and FSLite in the gem5 simulator [28], [29] and the source code is available online (refer to Appendix A for details). The gem5 simulator implements a two-level blocking MESI coherence protocol [2] called MESI_Two_Level. In MESI_Two_Level, the directory transitions to a blocking state on (i) receiving a GetX/Upgrade request for a block in S state, (ii) receiving a Get/GetX request for a block owned by LLC, and (iii) receiving a request for a block owned by a core. An explicit message from the requestor is required to unblock the directory. We have modified MESI_Two_Level so that the directory does not block in the first two of the aforementioned three cases, obviating the need for an unblock message in these two cases. We use this improved MESI as the baseline cache coherence protocol, which closely resembles the SGI Origin 2000 protocol [65].

The simulated system configuration is shown in Table II. The latency and area numbers are computed using CACTI [66]. Table II shows that the total storage overhead of the PAM table, SAM table, and the directory extension is less than 5% of the total capacity of the cache hierarchy. The SAM table's size includes its tag overhead and LRU state bits assuming a 48-bit physical address. Its size *drops* to 9.7 KB when the reader metadata optimization discussed in Section VI is applied. The evaluation of all applications is done in the *Full System* (FS) mode of gem5 with 4 child threads. We use an in-order CPU model because that allows us to run a maximum number of applications in the FS mode. Later in this section, we evaluate our proposal on an out-of-order issue CPU model running with the *Syscall Emulation* (SE) mode of gem5.

Benchmark Applications: Our approach is evaluated on applications chosen from the PHOENIX [11], Synchrobench [67], and PARSEC [16] benchmark suites, and the Huron artifact [10], [68]. Table III lists the applications from different suites used in the evaluation. These applications have been used by prior work for studying false sharing [7], [10]. To evaluate the performance and energy benefits of our proposal, we include applications that are known to have false sharing. To understand the overheads of our proposal, we also include applications that do not exhibit any false sharing. We evaluate the correctness of our protocols on several custom-designed micro-benchmarks and with programs provided by Feather [69], but do not discuss the results for microbenchmarks. FSDetect is able to detect *all* known harmful instances of false sharing in both microbenchmarks and benchmarks. Figure 13 shows the fraction of loads/stores that miss in the L1D cache for applications with false sharing. Since a fraction of the L1D cache misses leads to false sharing, this figure shows that less than 5% of the L1D demand accesses lead to false sharing on average. In the following, we discuss the performance and energy benefits of our approach, the impact of optimizations, and sensitivity to different parameters.

TABLE II: System configuration simulated with gem5

Cores	8
CPU type	In-order CPU with 3 GHz clock frequency
L1I cache	32 KB per core, 8-way, area: 3.02mm^2
L1D cache	32 KB per core, 8-way, area: 7.43mm^2
L1D Latency	data: 3 cycles and tag: 1 cycle
L2 (LLC)	2 MB per core, 16 way, area: 13.74mm^2
LLC Latency	data: 8 cycles and tag: 2 cycles
Cache line size	64 bytes
Memory	3 GB, DDR3-1600, 8 ranks, 64-bit channel
Kernel and OS	4.19.83, Ubuntu 18.04
Compiler and flags	gcc 7.5.0, -g, -static, default optimization
PAM table	8 KB (512 entries) per L1D cache, 8-way, area: 0.017mm^2
SAM table	12.7 KB (128 entries) per LLC slice, 16-way, area: 0.095mm^2
Directory extension	76 KB per LLC slice (FC, IC, HC, PMMC)
Conflict detection	2 cycles for conflict checking a PRV block
Tunable parameters	$\tau_P = 16, \tau_{R1} = 16, \tau_{R2} = 127$

TABLE III: Benchmark applications used in our evaluation

W/ false sharing			W/o false sharing		
Boost-Spinlock	BS	[10]	Blackscholes	BL	[16]
Lockless-Toy	LL	[10]	Bodytrack	BO	[16]
Linear-Regression	LR	[11]	Canneal	CA	[16]
Locked-Toy	LT	[10]	Facesim	FA	[16]
Reference-Count	RC	[10]	Fluidanimate	FL	[16]
StreamCluster	SC	[16]	Swaptions	SW	[16]
ESTM-SFtree	SF	[67]			
String-Match	SM	[11]			

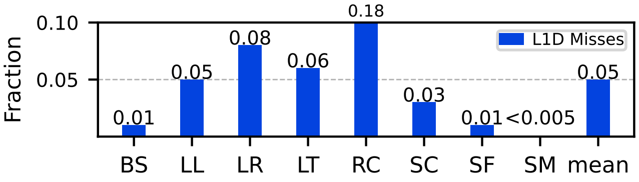
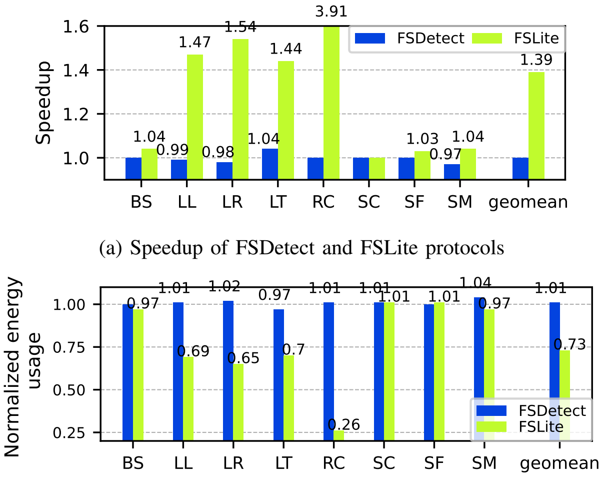


Fig. 13: Fraction of L1D cache accesses that lead to misses.

B. Performance and Energy Evaluation

Figure 14a shows the performance speedup achieved by our proposal over baseline MESI for different applications that are known to suffer from false sharing. FSDetect detects all known impactful false sharing instances with negligible performance overhead of **0.3**% on the average, with a maximum overhead of 3% for SM. FSLite repairs all instances of false sharing identified by FSDetect, and achieves an average speedup of **1.39X**. Several applications show significant speedup with FSLite, e.g., LL, LR, LT, and RC, with a maximum benefit of **3.9X** for RC. The observed performance gain with FSLite can be attributed to lower miss rates in the L1D cache. The L1D cache miss rates of LL, LT, and RC drop to nearly zero from the baseline values of 5%, 6%, and 18% respectively (shown in Figure 13), while that of LR drops to 0.8% from 8% in the



(a) Speedup of FSDetect and FSLite protocols

(b) Normalized energy expense in FSDetect and FSLite protocols

Fig. 14: Speedup and energy expense of FSDetect and FSLite compared to the baseline MESI protocol.

baseline. On the other hand, applications with a muted benefit in their L1D cache miss rate (BS, SM, and SF) have less impact on their run time (≤4% speedup). SC does not show any speedup because the volume of false sharing is small. We exclude SC from the studies presented later in this section.

The FSLite protocol delivers performance close to the manually fixed versions (see Figure 2) for most of the applications. Interestingly, FSLite *outperforms* the manually-fixed versions of two applications namely, LT and RC. FSLite outperforms the manual fix for LT as the working set size of the original application is inflated by 4X due to padding in the manually fixed version. The manually fixed version reduces the L1D cache miss rate from 6.4% to 2.4%, but FSLite eliminates nearly all L1D cache misses leading to an L1D cache miss rate of only 0.01%. In the manually fixed version, the benefit of eliminating false sharing is offset by additional cache misses due to the increased working set size. FSLite experiences good locality as the working set fits in the L1D cache resulting in a negligible L1D cache miss rate. The data layout of RC is modified in the manually fixed version due to the padding of the falsely shared field. Modifying the data layout introduces additional arithmetic instructions in the application binary for address computation of array indices requiring extra execution cycles. These results highlight the important benefits of our automated repair approach: does not *inflate* the memory footprints of applications and does not *modify* the binary.

Figure 14b shows the energy expense of our protocols normalized to the baseline. The computed energy includes static energy expended in the cache hierarchy and the structures added by our proposal as well as the dynamic fill energy expended in the L1D cache and LLC. The figure shows that the energy consumption of FSDetect is comparable to baseline, while FSLite saves 27% energy on average. The savings in static energy (not shown separately) in FSLite is 35%, which is because of the reduced application execution time.

We observe that FSLite reduces the number of request messages originating from the L1 caches by 80% on average for the applications that suffer from false sharing. However, the metadata messages introduce, on average, 5% additional interconnect traffic in FSLite leading to an overall reduction of 75% interconnect traffic from the cores to the LLC. FSDetect does not stress the network bandwidth due to metadata messages and the overhead is within 1-2% of the baseline.

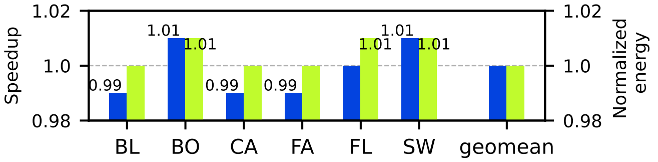


Fig. 15: Speedup (left bar) and energy expense (right bar) of FSLite for applications without false sharing.

Figure 15 shows the speedup and normalized energy expense for applications without false sharing. FSLite has negligible impact on the performance and energy expense. The mean slowdown and energy expense are both within 0.1% of the baseline. Overall, when averaged over all 14 applications listed in Table III, FSLite achieves a 1.21X speedup and 16% energy saving.

Sensitivity to Privatization Threshold τ_P: Choosing a reasonable value of τ_P is important for the performance of FSLite. While applications may benefit substantially from aggressive privatization with a smaller τ_P, a small threshold may also repeatedly privatize blocks that later become truly shared negating the benefit of privatization. The default value of τ_P is 16. Setting τ_P to an integral multiple of the total number of cores guards against premature privatization when all cores are involved in sharing. Figure 16 shows the speedup of FSLite with larger τ_P (32 and 64) relative to using the default τ_P of 16. The results show that there is a small slowdown (∼1%) on average with increasing τ_P. A larger τ_P delays privatization leading to loss in performance. The performance of the LL benchmark decreases with increasing τ_P due to a small rise in the L1D cache miss rate.

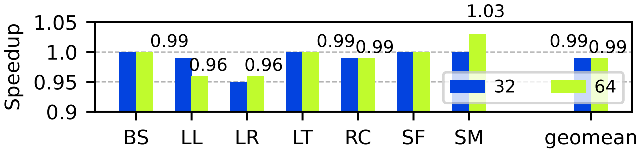


Fig. 16: Sensitivity of FSLite to the threshold τ_P.

Sensitivity to SAM Table Size: We observe that with the default SAM Table size of 128 entries (8 sets × 16 ways) per LLC slice, only 0.13% accesses replace a valid entry averaged over all applications. As a result, when we increase the SAM Table size to 256 entries per LLC slice, we do not see any performance difference. A small SAM table works well because a limited number of lines are falsely shared at a time.

Reader Metadata Optimization: The reader metadata optimization introduced in Section VI reduces the SAM table width by 25%. We observe that the total number of blocks privatized with this optimization is the same as seen without this optimization across all applications. Therefore, this optimization offers a significant reduction in the SAM table's area overhead while enjoying the same performance benefit.

Impact of Coarse-grain Tracking: We explore tracking of access metadata at a granularity coarser than a byte to reduce the area overhead of the PAM and SAM tables. We observe that the false sharing applications do not show any performance degradation when the access metadata are maintained at 2- and 4-byte granularity. This is expected because most false sharing instances manifest with 4-byte data types. Tracking access information at a 4-byte granularity reduces the size of the PAM table to 2 KB per L1D cache and that of the SAM table with reader metadata optimization to 3 KB per LLC slice.

Comparing FSLite with 128 KB L1D Cache: For a per core and per LLC slice iso-storage evaluation, we compare the performance of FSLite having a 32 KB L1D cache to that of the baseline having a 128 KB L1D cache. Relative to the baseline with 32 KB L1D cache, SM and LL degrade in performance by 3% and 9% respectively in the baseline with 128 KB L1D cache due to increased volume of false sharing instances. With a smaller L1D cache, some of these falsely shared lines get evicted early, thereby preventing the corresponding false sharing instances from materializing. On the other hand, baseline performance of LR, LT, SF, BO, FA, and SW improve by 3%, 6%, 1%, 5%, 1%, and 4% respectively with a 128 KB L1D cache. When averaged across all 14 applications listed in Table III, FSLite having a 32 KB L1D cache continues to deliver a 1.21X speedup relative to the baseline having a 128 KB L1D cache.

Sensitivity to Larger Private Caches: A three-level cache hierarchy introduces a large mid-level cache. To mimic its behavior, we evaluate our proposal using a 512 KB L1D cache in the two-level cache hierarchy. The average speedup achieved by FSLite in this configuration for the false sharing applications is 1.39X. Thus, FSLite continues to achieve similar performance improvement with a larger private cache.

Performance on Out-of-order Issue Cores: We evaluate FSLite with out-of-order issue cores using the syscall emulation (SE) mode of gem5[4] to study whether dynamic scheduling and out-of-order issue can hide some of the inefficiencies of false sharing. Using 8-wide cores in SE mode, we could simulate six out of the eight applications with false sharing listed in Table III. The baseline performance of these applications improves by 5.1X on average when using the out-of-order issue cores compared to the in-order cores. This large speedup comes from an 86% reduction in commit stalls due to partial hiding of false sharing overhead and 40% reduction in useful commit cycles due to wider instruction processing pipeline and 8-wide commit. When FSLite is enabled on top of this, it further eliminates nearly 55% of the residual commit stalls achieving an average speedup of 1.63X over the out-of-order issue baseline. FSLite achieves an average speedup of 1.56X for the same set of six applications on the in-order cores in FS mode.

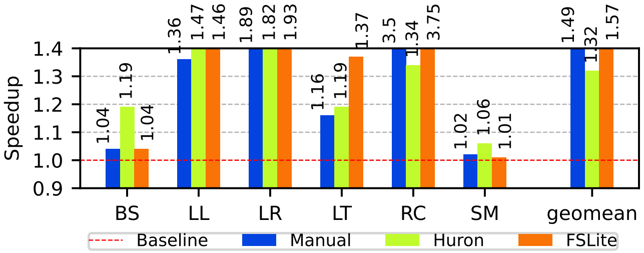


Fig. 17: Comparison between Huron and FSLite.

Comparison with Huron: Figure 17 compares the speedup achieved by manual fix, Huron [10], and FSLite relative to the baseline. We evaluate only those applications from the Huron artifact that run successfully with gem5. The inputs to the applications across all the techniques are same. The applications are compiled using the clang compiler that comes with the Huron artifact and linked statically. Due to the use of a different compiler toolchain, the speedup numbers of manual fix and FSLite shown in Figure 17 may not match with those shown earlier in the paper. FSLite outperforms Huron and manual fix on average by 19.8% and 6.8% respectively. Huron outperforms manual fix as well as FSLite by 14% on BS as it commits 15% fewer instructions. For RC, Huron offers limited benefit (34% speedup over baseline) and lags behind FSLite and manual fix by significant margins. Huron fails to mitigate all false sharing instances in RC. FSLite and Huron deliver nearly similar performance on LL and SM, while FSLite outperforms Huron on LR and LT. The benefits on SM are limited due to short-lived false sharing episodes.

In addition to delivering better performance, FSLite enjoys several other advantages over Huron. Huron requires access to the application's source and test cases for static pointer analysis, profiling, and logging memory accesses. Furthermore, Huron's approach needs to be replicated for *every* software toolchain. Its compile-time data layout transformations may not work for managed languages, languages with a moving garbage collector, and dynamic languages in which code is generated and loaded on-the-fly [12]. In contrast, FSLite does not require additional compiler passes for tracking the memory footprint of an application, does not assume the availability of application's source or application-related data or metadata, does not require tight integration with the operating system, does not alter the data layout, and does not inflate the memory requirement of the applications. FSLite is agnostic of the programming language features and compiler toolchain, and can automatically and transparently fix false sharing in an application irrespective of the development stack.

IX. Summary

The performance degradation arising from the presence of falsely shared cache lines is an artifact of the invalidation-based SWMR cache coherence protocols. Unlike most prior work, we propose coherence protocol extensions to automatically detect and repair instances of false sharing, with no dependence on the application source and the software stack. Our simulation results show improved run time, energy behavior, and lower bandwidth demands on the on-chip network, with a modest increase in hardware complexity, indicating that FSDetect and FSLite protocols have the potential to solve the long-standing performance problem of false sharing.

[4] The version of gem5 (v20) used in our implementation does not support out-of-order issue cores with FS mode [70].

Appendix
Details of the Artifact

A. Abstract

The artifact includes implementation of our detection and repair coherence protocols for false sharing in the gem5 simulator and the image files for full system simulation with gem5. To ease the setup process and portability, we provide a Docker image with all dependencies pre-installed and ready-to-execute scripts that automate the execution of applications and plotting of graphs. The micro24-artifact-README file [71] is provided with the instructions to set up the Docker image on the host machine, and run the scripts.

B. Artifact Check-list (Meta-information)

- **Program:** The `gem5-false-sharing` directory in the `false-sharing-micro24` artifact repository [72] contains the source files that implement our approach. Search for the string "FalseSharing" to browse the changes we have introduced in the Gem5 source code.
- **Compilation:** GCC v9+, Clang v7+, Python2.7, Scons
- **Binary:**
  1) The pre-built binaries of applications are provided as tar image files in the Docker image [73]. The image files of the applications can be accessed using QEMU [74].
  2) The raw images are built using instructions provided in the Gem5-v20.0.0.3-resource [75]. The images include the dependencies required for compiling applications that are not included in the resources.
- **Data set:** The artifact includes applications from PARSEC [16], Phoenix [11], Synchrobench [67], and Huron [10] suites.
- **Run-time environment:** Linux distribution, Docker, and KVM
- **Hardware:** An x86_64 system with minimum 8 cores and 32 GB RAM with support for virtualization.
- **Execution:** The artifact includes scripts to launch experiments and plot graphs.
- **Metrics:** The effectiveness of our approach is measured in terms of the speedup and savings in energy consumption achieved for each application.
- **Output:** After an experiment completes successfully, the gem5 stats file is generated in the directory `/home/prospar/prospar-micro-output` and the corresponding plot is generated in `/home/prospar/false-sharing-scripts-micro24` directory.
- **Experiments:** We provide scripts in the Docker image for reproducing the results discussed in Section VIII.
- **How much disk space required (approximately)?:** Setting up Docker, and downloading and compiling the source code and benchmarks will require around 80 GB. We suggest having at least 100 GB of free space in the root partition of the disk if Docker uses the default installation path for storing and running containers. Refer to Docker storage setting [76] to change the default storage location.
- **How much time is needed to prepare workflow (approximately)?:** Setting up the Docker image will take around 1–2 hours. The initial setup to create directories and build gem5 protocols in the image will take 4–6 hours. Once all the protocols are built, a sanity check can be done by testing the protocols with a microbenchmark. The `test-app-script.sh` executes a microbenchmark and generates a plot, and takes around one hour to complete. The script removes the plot generated for the microbenchmark.
- **How much time is needed to complete experiments (approximately)?:** Reproducing all the results discussed in Section VIII can take up to 3 months if the experiments are run sequentially in a single Docker instance. We suggest running experiments in parallel to speed up the evaluation.
- **Publicly available?:** Yes, the Docker image, the source code of our implementation, the benchmarks, and the resources to build the benchmark images are available on Zenodo [77]. Each directory includes a README file with detailed instructions.
- **Archived (provide DOI):** Leveraging Cache Coherence to Detect and Repair False Sharing On-the-fly [77].

C. Description

1) How to access: The source code and other resources can be accessed via the GitHub repository [72] and Zenodo [77].

2) Hardware dependencies: An x86_64 system with 8 cores, 32 GB RAM, and 200 GB of free space with support for virtualization.

3) Software dependencies: Ubuntu 22.04, Docker, and KVM; user must be added to `kvm` and `docker` user groups.

D. Installation and Execution

Set up the host environment:

(i) Install Docker [78].
(ii) Install KVM [79].
(iii) Post Docker and KVM installation steps:

```
# Docker
sudo groupadd docker
sudo usermod -aG docker $USER # Your user ID
# KVM
sudo adduser $USER libvirt # Your user ID
sudo adduser $USER kvm
```

(iv) Please log in again to the host system to activate permissions. Otherwise, the following steps may require sudo access.
(v) Update the `perf_event_paranoid` variable: `sudo sysctl kernel.perf_event_paranoid=-1`.
(vi) Download the Docker image [73] of size ∼ 13 GB.
(vii) Import the Docker image: `docker import micro-fs-artifact.tar`.
(viii) Get the image ID: `docker image ls`.
(ix) Activate the container: `docker run -it -privileged <image_id> /bin/bash`.
(x) The user should log in to a new Bash shell in the Docker container. Do not close the current terminal (say `terminal-0`).
(xi) Open another terminal say `terminal-1` and run the command : `docker ps -a`.
(xii) Copy the container ID of image <image_id>.
(xiii) Rename the container: `docker container rename <container_id> micro-fs`. You can choose any name of your choice.
(xiv) Close `terminal-1`.
(xv) Refer to the docker container page [80] for more commands.

Setting up the execution environment in the `micro-fs` container:

(i) The following is a list of environment variables defined in `micro-fs`:

```
MICRO_VIR_ENV="/home/prospar/micro-virtualenv"
MICRO_HOME="$MICRO_VIR_ENV/false-sharing-micro24"
MICRO_OUT="/home/prospar/prospar-micro-output"
MICRO_RES="/home/prospar/prospar-micro-result"
MICRO_SCRIPT="/home/prospar/false-sharing-scripts-micro24"
```

Please use the following instructions to validate the artifact and build the gem5 source.

```
cd $MICRO_SCRIPT
# Ensure that all scripts and the framework code are up to date
git pull
# Validate the structural setup one time
bash validation−script.sh
# Extract the tar image, and build the necessary directories
# and protocols in gem5. May take up to 6 hours to complete.
bash setup−script.sh
# Validate the protocols
bash test−app−script.sh fslite fsdetect 1
```

Reproducing the results:

(i) Please use the following instructions to reproduce the primary results discussed in Section VIII.

```
cd $MICRO_SCRIPT

# Figure 2, estimated completion time: 9 days/iteration
bash introduction−result.sh <num_of_iter>
# Figure 14, estimated completion time: 13 days/iteration
bash primary−result.sh <num_of_iter>
# Figure 15, estimated completion time: 10 days/iteration
bash parsec−result.sh <num_of_iter>
```

(ii) In Table IV, we list the Bash scripts to reproduce the other results presented in Section VIII. All the scripts can be run with `bash <script-name.sh> <num_of_iter>`.

TABLE IV: List of Bash scripts

Script	Description
fc-ic-result.sh	Study FC and IC thresholds
granularity-result.sh	Explore sensitivity to tracking width
sam-result.sh	Explore sensitivity of SAM table size
baseline-128KB-result.sh	Evaluate with 128KB L1D cache
huron-result.sh	Compare with Huron
opt-reader-result.sh	Script to verify readers optimization
out-of-order-result.sh	Compare performance with OOO core

The estimated run times of the applications for one iteration with the baseline MESI protocol are shown in Table V.

TABLE V: Run-time estimates

Applications	Run time (in hr)	False Sharing Present
Blackscholes	4	No
Bodytrack	12	No
Canneal	5	No
Facesim	24	No
Fluidanimate	12	No
Swaptions	26	No
Boost-spinlock	12	Yes
ESTM-sftree	18	Yes
Linear-reg	10	Yes
Locked-toy	18	Yes
Lockless-toy	34	Yes
Ref-count	14	Yes
Streamcluster	30	Yes
String-match	6	Yes
Total time	215	

E. Experiment workflow

We provide scripts to reproduce all the results presented in Figure 2 and Section VIII. Figures 2, 14, and 15 present the primary results of the paper. The results from these figures can be reproduced first before the other design exploration and optimization studies reported in Section VIII.

F. Evaluation and expected results

After the successful completion of all applications for a configuration, an output directory named after the configuration will be created and will contain the configuration of the simulated hardware and a stats file. For example, the `introduction-result.sh` will create a directory `micro-manual-fix` in the `$MICRO_OUT` and `$MICRO_RES` directories. The `$MICRO_OUT/micro-manual-fix` directory will contain the gem5 result files with the following directory structure for each application: `<protocol-name>/<input-size>/<iteration-number>/<application-name>`.

For each application, there will be `output.txt` and `error.txt` files in addition to gem5 output files in the output directory. The `error.txt` file captures errors encountered during a simulation, while the `output.txt` file logs the progress of the application during simulation.

The `$MICRO_RES/micro-manual-fix` directory will contain a consolidated CSV file for stats of all applications generated after parsing the `stats.txt` (gem5 output file) file of each application. The directory also contains the bar charts for each stat with absolute values for all protocols included in the experiment.

The graph plotting script reads the CSV file to generate relevant plots. The scripts will generate plots in the `$MICRO_SCRIPT` directory in the Docker container (refer to the README file [71] for the names of different plots). The directory `reference_plots` contains the expected output for each experiment.

Lower run time and energy dissipation are the primary performance metrics that showcase the benefits of our proposed approach. Figures 14 and 15 report both the run time and energy consumption results. The other figures focus on the run time of the applications. False sharing can also stress the network resources by flooding the network with coherence invalidation and intervention messages leading to wastage of bandwidth and energy. The intermediate CSV file generated for graph plotting in the `$MICRO_RES` directory for each set of experiments provides the number of intervention and invalidation messages for each protocol. For applications with significant false sharing, the FSLite protocol should report a significantly smaller number of invalidation and intervention messages compared to the baseline non-blocking MESI protocol (Section VIII-A).

G. Experiment customization

The `$MICRO_SCRIPT` directory contains helper scripts and different sets of configuration files. Each config file defines values for the simulated hardware system. Our artifact allows customizing most of the configuration parameters by editing a `config.ini` file located in the `$MICRO_SCRIPT/config-script` directory.

References

[1] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *ISCA*, 1984, pp. 348–354.

[2] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed. Morgan & Claypool publishers, 2020.

[3] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer, "Detecting Data Races on Weak Memory Systems," in *ISCA*, 1991, pp. 234–243.

[4] T. Liu and E. D. Berger, "SHERIFF: Precise Detection and Automatic Mitigation of False Sharing," in *OOPSLA*, 2011, pp. 3–18.

[5] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "DeFT: Design Space Exploration for On-the-Fly Detection of Coherence Misses," *TACO*, vol. 8, no. 2, Jun. 2011.

[6] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive Granularity Cache Coherence," in *ISCA*, 2013, pp. 547–558.

[7] M. Chabbi, S. Wen, and X. Liu, "Featherlight On-the-Fly False-Sharing Detection," in *PPoPP*, 2018, pp. 152–167.

[8] T. Liu, C. Tian, Z. Hu, and E. D. Berger, "PREDATOR: Predictive False Sharing Detection," in *PPoPP*, 2014, pp. 3–14.

[9] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield, "Whose Cache Line Is It Anyway? Operating System Support for Live Detection and Repair of False Sharing," in *EuroSys*, 2013, pp. 141–154.

[10] T. A. Khan, Y. Zhao, G. Pokam, B. Mozafari, and B. Kasikci, "Huron: Hybrid False Sharing Detection and Repair," in *PLDI*, 2019, pp. 453–468.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-Core and Multiprocessor Systems," in *HPCA*, 2007, pp. 13–24.

[12] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti, "Remix: Online Detection and Repair of Cache Contention for the JVM," in *PLDI*, 2016, pp. 251–265.

[13] W. J. Bolosky and M. L. Scott, "False Sharing and Its Effect on Shared Memory Performance," in *USENIX SEDMS*, 1993, p. 3.

[14] V. Filanovsky and H. Sane, "Seeing through hardware counters: a journey to threefold performance increase," Nov. 2022. [Online]. Available: https://netflixtechblog.com/seeing-through-hardware-counters-a-journey-to-threefold-performance-increase-2721924a2822

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995, pp. 24–36.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008, pp. 72–81.

[17] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *OSDI*, 2010, pp. 1–16.

[18] M. Ronstrom, "MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012," Apr. 2012. [Online]. Available: https://mikaelronstrom.blogspot.com/2012/04/

[19] S. Vojtovich, "JIRA Issue," 2017. [Online]. Available: https://jira.mariadb.org/browse/MDEV-14482

[20] "False Sharing in boost::detail::spinlock_pool," 2012. [Online]. Available: https://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool

[21] A. Rukavytsia, "What false sharing is and how JVM prevents it," Mar. 2017. [Online]. Available: https://medium.com/@rukavitsya/what-is-false-sharing-and-how-jvm-prevents-it-82a4ed27da84

[22] J. Rose, "JDK/JDK-8180450: secondary_super_cache does not scale well," Nov. 2017. [Online]. Available: https://bugs.openjdk.org/browse/JDK-8180450

[23] P. Zhang, "[11u] RFR: 8244214: Add paddings for TaskQueueSuper to reduce false-sharing cache contention," Jun. 2020. [Online]. Available: https://mail.openjdk.org/pipermail/jdk-updates-dev/2020-June/003369.html

[24] N. McDonald, "libdes: A framework for parallel discrete event simulation," 2015. [Online]. Available: https://github.com/nicmcd/libdes

[25] G. Holzmann, "The model checker spin," *IEEE TSE*, vol. 23, no. 5, pp. 279–295, 1997.

[26] G. J. Holzmann and D. Bosnacki, "The Design of a Multicore Extension of the SPIN Model Checker," *IEEE TSE*, vol. 33, no. 10, pp. 659–674, 2007.

[27] A. Ather, "Intel Optimization at Netflix," May 2023. [Online]. Available: https://medium.com/@amerather_9719/intel-optimization-at-netflix-79ef0efb9d2

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH CAN*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[29] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020.

[30] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "LASER: Light, Accurate Sharing dEtection and Repair," in *HPCA*, 2016, pp. 261–273.

[31] C. DeLozier, A. Eizenberg, S. Hu, G. Pokam, and J. Devietti, "TMI: Thread Memory Isolation for False Sharing Repair," in *IEEE Micro*, 2017, pp. 639–650.

[32] T. Liu and X. Liu, "Cheetah: Detecting False Sharing Efficiently and Effectively," in *CGO*, 2016, pp. 1–11.

[33] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic Cache Contention Detection in Multi-Threaded Applications," in *VEE*, 2011, pp. 27–38.

[34] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy," in *MICRO*, 2012, pp. 376–388.

[35] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence Decoupling: Making Use of Incoherence," in *ASPLOS*, 2004, pp. 97–106.

[36] M. Tolubaeva, Y. Yan, and B. Chapman, "Compile-Time Detection of False Sharing via Loop Cost Modeling," in *IEEE IPDPSW*, 2012, pp. 557–566.

[37] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu, "Detection of False Sharing Using Machine Learning," in *SC*, 2013.

[38] T. E. Jeremiassen and S. J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," in *PPoPP*, 1995, pp. 179–188.

[39] S. M. Günther and J. Weidendorfer, "Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation," in *WBIA*, 2009, pp. 26–33.

[40] J.-H. Chow and V. Sarkar, "False Sharing Elimination by Selection of Runtime Scheduling Parameters," in *ICPP*, 1997, pp. 396–403.

[41] M. Kandemir, A. Choudhary, J. Ramaujam, and P. Banerjee, "On reducing false sharing while improving locality on shared memory multiprocessors," in *PACT*, 1999, pp. 203–211.

[42] S. J. Eggers and T. E. Jeremiassen, "Eliminating False Sharing," in *ICPP*, 1991, pp. 377–381.

[43] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE TOC*, vol. 43, no. 6, pp. 651–663, jun 1994.

[44] J. B. Rothman and A. J. Smith, "Minerva: An Adaptive Subblock Coherence Protocol for Improved SMP Performance," in *ISHPC*, 2002, pp. 64–77.

[45] M. Kadiyala and L. N. Bhuyan, "A Dynamic Cache Sub-block Design to Reduce False Sharing," in *ICCD*, 1995, pp. 313–318.

[46] W. Shi, W. Hu, and M. Zhu, "An Innovative Implementation for Directory-Based Cache Coherence in Shared Memory Multiprocessors," *ACM SIGARCH CAN*, vol. 25, no. 5, pp. 2–9, Dec. 1997.

[47] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," in *ASPLOS*, 2000, pp. 117–128.

[48] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *POPL*, 2005, pp. 378–391.

[49] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *PLDI*, 2008, pp. 68–78.

[50] S. Kaxiras and A. Ros, "Efficient, Snoopless, System-on-Chip Coherence," in *SOCC*, 2012, pp. 230–235.

[51] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011, pp. 155–166.

[52] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations," in *ASPLOS*, 2015, pp. 545–559.

[53] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism," in *ASPLOS*, 2013, pp. 13–26.

[54] S. Kaxiras and G. Keramidas, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sep. 2010.

[55] R. Zhang, S. Biswas, V. Balaji, M. D. Bond, and B. Lucia, "Neat: Low-Complexity, Efficient On-Chip Cache Coherence," *CoRR*, vol. abs/2107.05453, 2021.

[56] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia, "Valor: Efficient, Software-Only Region Conflict Exceptions," in *OOPSLA*, 2015, pp. 241–259.

[57] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races," in *ISCA*, 2010, pp. 210–221.

[58] S. Biswas, R. Zhang, M. D. Bond, and B. Lucia, "Rethinking Support for Region Conflict Exceptions," in *IPDPS*, 2019, pp. 1095–1106.

[59] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages," in *PLDI*, 2010, pp. 351–362.

[60] V. Balaji, D. Tirumala, and B. Lucia, "Flexible Support for Fast Parallel Commutative Updates," *CoRR*, vol. abs/1709.09491, 2017.

[61] M. J. Garzaran, M. Prvulovic, Y. Zhang, J. Torrellas, A. Jula, H. Yu, and L. Rauchwerger, "Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors," in *PACT*, 2001, pp. 243–254.

[62] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight, "Architectural Support for Uniprocessor and Multiprocessor Active Memory Systems," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 288–307, Mar. 2004.

[63] G. Zhang, W. Horn, and D. Sanchez, "Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems," in *IEEE Micro*, 2015, pp. 13–25.

[64] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates," in *MICRO*, 2019, pp. 1009–1022.

[65] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *ISCA*, 1997, pp. 241–251.

[66] HP Labs, "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," https://www.cs.utah.edu/~rajeev/cacti7/.

[67] V. Gramoli, "More Than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms," in *PPoPP*, 2015, pp. 1–10.

[68] T. A. Khan, Y. Zhao, and B. Kasikci, "huron," 2018. [Online]. Available: https://github.com/efeslab/huron

[69] M. Chabbi, "feather," 2018. [Online]. Available: https://github.com/WitchTools/Feather

[70] A. Akram, "X86 Linux Boot Status on gem5-19," 2015. [Online]. Available: https://www.gem5.org/project/2020/03/09/boot-tests.html

[71] V. Patel, S. Biswas, and M. Chaudhuri, "README file for MICRO Artifact for Paper 506," 2024. [Online]. Available: https://github.com/prospar/false-sharing-scripts-micro24/blob/main/micro24-artifact-README.md

[72] ——, "false-sharing-micro24," 2024. [Online]. Available: https://github.com/prospar/false-sharing-micro24

[73] ——, "Leveraging Cache Coherence to Detect and Repair False Sharing On-the-fly," 2024. [Online]. Available: https://zenodo.org/records/13293424/files/micro-fs-artifact.tar?download=1

[74] QEMU, "Download QEMU." [Online]. Available: https://www.qemu.org/download/#linux

[75] Gem5, "gem5-resources," 2020. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+log/refs/tags/v20.0.0.3/src

[76] Docker, "Docker Community Forums," 2015. [Online]. Available: https://forums.docker.com/t/how-do-i-change-the-docker-image-installation-directory/1169/1

[77] V. Patel, S. Biswas, and M. Chaudhuri, "Leveraging Cache Coherence to Detect and Repair False Sharing On-the-fly," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13293424

[78] Docker, "Install Docker Engine on Ubuntu." [Online]. Available: https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository

[79] phoenixNAP, "How to install kvm on ubuntu," 2024. [Online]. Available: https://phoenixnap.com/kb/ubuntu-install-kvm

[80] Docker, "docker container." [Online]. Available: https://docs.docker.com/reference/cli/docker/container/