

CachePerf: A Unified Cache Miss Classifier via Hybrid Hardware Sampling

JIN ZHOU, University of Massachusetts Amherst, USA

STEVEN (JIAXUN) TANG, University of Massachusetts Amherst, USA

HANMEI YANG, University of Massachusetts Amherst, USA

TONGPING LIU, University of Massachusetts Amherst, USA

The cache plays a key role in determining the performance of applications, no matter for sequential or concurrent programs on homogeneous and heterogeneous architecture. Fixing cache misses requires to understand the origin and the type of cache misses. However, this remains to be an unresolved issue even after decades of research. This paper proposes a unified profiling tool—CachePerf—that could correctly identify different types of cache misses, differentiate allocator-induced issues from those of applications, and exclude minor issues without much performance impact. The core idea behind CachePerf is a hybrid sampling scheme: it employs the PMU-based coarse-grained sampling to select very few susceptible instructions (with frequent cache misses) and then employs the breakpoint-based fine-grained sampling to collect the memory access pattern of these instructions. Based on our evaluation, CachePerf only imposes 14% performance overhead and 19% memory overhead (for applications with large footprints), while identifying the types of cache misses correctly. CachePerf detected 9 previous-unknown bugs. Fixing the reported bugs achieves from 3% to 3788% performance speedup. CachePerf will be an indispensable complementary to existing profilers due to its effectiveness and low overhead.

CCS Concepts: • **Software and its engineering** → **Multiprocessing / multiprogramming / multitasking**; • **Computer systems organization** → **Real-time operating systems**.

Additional Key Words and Phrases: Cache Performance, Cache Miss, Conflict Miss, Coherency Miss, Capacity Miss

ACM Reference Format:

Jin Zhou, Steven (Jiaxun) Tang, Hanmei Yang, and Tongping Liu. 2022. CachePerf: A Unified Cache Miss Classifier via Hybrid Hardware Sampling. In *SIGMETRICS '22, June 6–10, 2022, Mumbai, India*. ACM, New York, NY, USA, 25 pages. <https://doi.org/10.1145/3489048.3526954>

1 INTRODUCTION

Cache accesses are typically orders of magnitude faster (e.g., $200 \times$ [30]) than memory accesses. Therefore, it is critical to reduce cache misses in order to boost the performance of applications, no matter for single-threaded or multi-threaded applications running on homogeneous or heterogeneous hardware architectures. However, it is challenging to identify cache misses statically, as they are related to access pattern [32], hardware feature (e.g., cache capacity, cache line size), or even starting addresses of objects [34].

Many tools aiming to identify cache misses have been developed in the past. Simulation-based approaches, such as different Pin tools [19, 38], or cachegrind (one tool inside Valgrind [40]) [49], typically impose prohibitive performance overhead (e.g., 100 times) that makes them even unsuitable

We have filed a U.S. patent with the serial number 63/281,942. Please contact with tongping@umass.edu for the licence.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGMETRICS '22, June 6–10, 2022, Mumbai, India

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9141-2/22/06.

<https://doi.org/10.1145/3489048.3526954>

for development phases [49, 52]. To solve such issues, many sampling-based tools, such as perf [10], oprofile [31], are proposed to reduce the profiling overhead. They could attribute the percentage of cache misses to the specific lines of the source code based on the sampling. However, they cannot report both the **type** and **origin** of cache misses, making their reports not sufficient to guide the bug fixes.

Different types of cache misses, including compulsory misses, capacity misses, conflict misses, and coherency misses [18, 21], require different fixing methods, as detailed in Section 2. A compulsory miss occurs when the related cache line is accessed for the first time, which is considered to be mandatory and unavoidable. Instead, a capacity miss may occur if the working set of a program exceeds the capacity of the cache. Capacity misses can be reduced by loop optimizations [55] or array regrouping [35]. In contrast, conflict misses can be introduced when more than N cache lines are mapping to the same set in N -way associative cache, and coherency misses may occur when multiple threads are accessing the same cache line simultaneously. Although some conflict and coherency misses can be reduced with padding, they require different padding strategies: fixing conflict misses should prevent the mapping to the same set, while coherency misses can be reduced by avoiding multiple threads accessing the same cache lines. Reducing cache misses also requires to know the origin: whether a problem is caused by the allocator or the application? For application bugs, which objects or which instructions are involved? Without knowing such information, it is impossible to reduce cache misses effectively.

Some tools aim to identify a specific type of cache misses, such as capacity misses [35], coherence misses [6, 25, 33, 39], and conflict misses [46]. However, it is inconvenient to identify all types of cache misses using these tools, as they are designed as exclusive to each other. Further, none of them could correctly identify cache misses caused by the memory allocator. For instance, `cache-thrash` can be slowed down by $38\times$ when using `TCMalloc` (as shown in Table 2). Without knowing the origin of cache misses, programmers may waste their efforts in improving the application but achieve only minor or no improvement. DProf [43] is the only tool that can identify all types of cache misses for data structures of Linux kernel, which unfortunately requires significant manual effort, as further discussed in Section 6.

This paper proposes a novel tool—CachePerf—that overcomes these shortcomings: (1) CachePerf is a unified profiler that could identify all fixable cache misses (except compulsory misses); (2) CachePerf only reports serious issues, saving manual effort spending on trivial issues; (3) CachePerf reports both type and origin of cache misses, providing useful information for bug fixes; (4) CachePerf only imposes reasonable overhead for its identification. Designing such a tool includes the following challenges.

The first challenge is to choose an appropriate profiling method that can classify all types of cache misses with reasonable overhead. Prior work employs different sampling events, including address sampling for capacity misses [35] and coherency misses [6, 33], HITM events [39] for coherency misses, and L1 cache misses for conflict misses [46]. However, it is infeasible to combine these events together, as that will introduce prohibitive overhead and complexity. Although it is intuitive to employ the hardware-based sampling, the Performance Monitoring Units (PMU) supports up to hundreds of events (e.g., 207 events at Intel Xeon Silver 4114 [15]). CachePerf’s selection is driven by the requirement of differentiating the type, reporting the origin, and measuring the seriousness of cache misses, as discussed later. In summary, such an event should capture the detailed information of memory accesses, such as the memory address, the related instruction, and the hit information (indicating a cache miss or not), which is often omitted by existing work [6, 33]. Therefore, “**the PMU-based precise address sampling**” is chosen as the right event, and we elaborate why and how CachePerf exploits this event as follows.

The second challenge is to differentiate all types of cache misses correctly. The PMU-based sampling helps filter out cache misses, but it is impossible to correctly identify the type of each cache miss under the sampling, due to the lack of the history of cache usage and memory accesses. Instead, CachePerf proposes to identify coherency misses based on **the cumulative behavior of many misses**: only very few cache lines (not mapping to the same set) with extensive misses are most likely caused by coherency misses. Unfortunately, this rule cannot differentiate capacity misses from conflict misses, where the detailed access pattern is required for the differentiation, as further discussed in Section 2. Further, CachePerf classifies other types of cache misses based on a key observation: *serious cache misses are typically caused by very few instructions whose access patterns are not altered during the whole execution.* Based on this key observation, we propose a novel approach—**hybrid hardware sampling**—to classify the type of cache misses: *the PMU-based coarse-grained sampling detects susceptible instructions with frequent cache misses, then the breakpoint-based fine-grained sampling is employed to identify memory access patterns of these selected instructions.* This approach combines the best of both worlds, as the coarse-grained sampling could reduce the profiling overhead, while the fine-grained sampling collects a short history of memory accesses that is necessary to classify the access pattern. For instance, it is easy to determine conflict misses if multiple continuous accesses are accessing the same cache set.

The third challenge is to differentiate cache misses caused by the memory allocator from those ones caused by applications. Although allocator-induced cache misses may have a high impact on the performance, they get less attention than they deserve. This paper makes the following *observations*: (1) the allocator may introduce both conflict and coherence misses (mainly false sharing, a type of coherency misses that multiple threads are accessing different words of the same cache line [32]); (2) Allocator-induced cache misses share the same attribute that multiple heap objects are involved unnecessarily, although this is not the sufficient condition. For instance, allocator-induced false sharing should have more than two objects on the same cache line. Further, these objects, accessed by different threads, must be allocated by different threads. Similarly, an allocator may introduce conflict misses, when multiple objects are mapped to the same set of cache lines. To the best of our knowledge, *CachePerf is the first work that reports allocator-induced cache misses.*

CachePerf further designs practical mechanisms that help reduce the detection overhead and avoid reporting minor issues: (1) CachePerf tracks a specified number of the most recent memory accesses (or a window), and then only checks cache misses inside if the miss ratio (i.e., the number of misses divided by the number of accesses) in the current buffer is larger than a threshold. This windowing mechanism also helps filter out sporadic cache misses, e.g., compulsory misses; (2) CachePerf further proposes a “ratio-based filtering” that only reports an issue if the ratio of memory accesses or cache misses is larger than a threshold;

We evaluated CachePerf on a range of well-studied benchmarks and real applications, where some have known cache misses. Based on our evaluation, CachePerf only introduces 14% performance overhead and 19% memory overhead (for large applications), while detecting all known bugs and night previously-unknown cache misses. Guided by CachePerf’s report, we are able to fix most detected cache misses, achieving the performance speedup up to 38×. Overall, the paper makes the following contributions:

- It proposes a novel hybrid sampling scheme that combines coarse-grained PMU-based sampling and fine-grained breakpoint-based sampling, with a better trade-off between performance and accuracy.
- It is the first tool that can classify different types of cache misses without manual involvement.
- It proposes practical mechanisms to differentiate cache misses caused by the allocator from those from applications, and to prune insignificant issues.

- It provides the detailed implementation of a profiler with low overhead (14% on average) and high effectiveness, confirmed by our extensive evaluation.

2 BACKGROUND AND OVERVIEW

This section first introduces some basic background of cache misses, and then discusses the basic idea of CachePerf.

2.1 Types of Cache Misses

Cache miss can be classified into compulsory miss, capacity miss, conflict miss, and coherence miss [43]. Among them, a compulsory miss occurs when the cache line is accessed for the first time, which is mandatory and unavoidable [21]. In the remainder of this paper, we mainly focus on the other three types of cache misses. In the following, we will discuss their definitions, fix strategies, and possible causes.

2.1.1 Capacity Miss. Capacity misses occur when the accessed data of a program exceeds the capacity of the cache [53]. When the cache cannot hold all the active data, some recently-accessed cache lines are forced to be evicted, which leads to cache misses if they are accessed again. As shown in Fig. 1(a), both `for` loops will suffer from cache capacity misses, as both Alpha and Beta’s size is four times of the cache size (with the “CACHE_SIZE” number of integers).

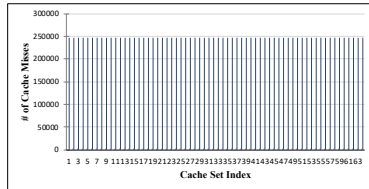
Capacity misses are mainly caused by applications. Not all capacity misses can be completely eliminated. However, some can be significantly reduced via array regrouping [35] or loop optimizations [55] (e.g., loop tiling [2]). For the example shown in Fig. 1(a), we could combine two loops into one loop as Fig. 1(b) to reduce cache misses, also known as loop fusion [11].

<pre>for (int i = 0; i < CACHE_SIZE; ++i) { Alpha[i] = i; } for (int i = 0; i < CACHE_SIZE; ++i) { Beta[i] = Alpha[i]*2; }</pre> <p style="text-align: center;">(a) Original code</p>	<pre>for (int i = 0; i < CACHE_SIZE; ++i) { Alpha[i] = i; Beta[i] = Alpha[i]*2; }</pre> <p style="text-align: center;">(b) Reduced cache misses via loop fusion [13]</p>
--	---

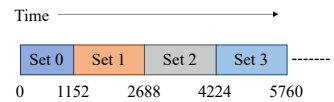
Fig. 1. An example with capacity misses.

```
for (int z = 0; z < num_zones; ++z){
    double vol = sdm.volume[z];
    for(int d = 0; d < num_directions; ++d){
        double w = dirs[d].w;
        for(int g = 0; g < num_groups; ++g){
            part += w * (*sdm.psi)(g,d,z) * vol;
        }
    }
}
```

(a) Loop with conflict misses



(b) Number of cache misses on each set



(c) Access sequence based on cache sets

Fig. 2. A real example of conflict misses from the Kripke application [27, 46].

2.1.2 Conflict Miss. Conflict misses are introduced in direct-mapped or set-associative cache [43, 46]. For an N-way associative cache, conflict misses will occur when more than N cache lines mapping to the same set are accessed recently.

Fig. 2(a) shows a real example of conflict misses: `Kripke` accesses multiple cache lines of Set 0 and then Set 1, as shown in Figure 2(c). For this example, each cache set has exactly the same number of cache misses, as shown in Figure 2(b). This indicates that conflict misses cannot be identified by the portion of misses in cache sets. Instead, the access pattern of the corresponding instruction(s) should be employed to identify such issues.

Conflict misses can not only be caused by applications, but also can be caused by the allocator when multiple allocated objects are mapped to the same cache set. `raytrace`, an application in PARSEC [4], introduces a 27% performance slowdown due to conflict misses of the allocator, as shown in Table 2. Conflict misses can be resolved or reduced by changing the starting addresses of objects, or padding the corresponding structure. Even for allocator-induced applications, we could insert some bogus memory allocations or change the size of the corresponding allocations in order to reduce cache misses.

2.1.3 Cache Coherence Misses. Multithreaded applications are prone to coherence misses when multiple threads are accessing the same cache line. When a thread writes to a cache line, the cache coherence protocol invalidates all existing copies of this cache line, introducing *cache coherency misses*. Coherence misses can be caused by true and false sharing. False sharing occurs when multiple threads are accessing different portions of the same cache line, while threads are accessing the same units in true sharing. When modern architectures are equipped with larger cache lines and more cores, they are more prone to coherence misses with higher performance impacts.

True sharing is typically caused by applications. Although true sharing is considered to be unavoidable [32], programmers could still refactor the code to reduce its seriousness [26, 39]. For example, programmers may reduce the updating of shared variables by using thread-local or local variables. False sharing can be caused by both applications and allocators. For allocator-induced false sharing, multiple threads may access different objects concurrently within the same cache line that are allocated by different threads. False sharing can be reduced by padding the data structure [25], or using per-thread private pages [32]. Therefore, it is important to differentiate between false and true sharing, as they need different fixing strategies.

2.2 Basic Idea of CachePerf

CachePerf aims to identify the type and the origin of cache misses correctly so that programmers can further fix them correspondingly. More specifically, CachePerf not only differentiates capacity misses, conflict misses, and coherency misses, but also differentiates whether some misses are caused by the allocator or the application. If they are caused by the application, CachePerf further reports the lines of code with the issue, e.g., call sites and instructions. For allocator-induced cache misses, CachePerf also reports the sizes of the related objects.

2.2.1 Differentiating Different Types of Misses. As mentioned in Section 1, it is challenging to identify the type of each miss directly. For instance, to identify a capacity miss, it is required to know the working set of the current program [43], which is infeasible under the coarse-grained sampling. CCProf [46] observes that “a relatively larger portion of cache misses in a subgroup of the total cache sets over the others indicates conflicts in those cache sets”. Unfortunately, this method is *neither sufficient nor necessary* condition of conflict misses, although it seems to be valid at the first glance. As shown in Fig. 2(b), all cache sets have exactly the same number of cache misses for the `Kripke` application. However, this issue belongs to “conflict misses” based on the access pattern shown in Fig. 2(c). Further, a `for` loop consecutively accessing an array (e.g., 1.5 times larger than the cache

size) may cause only half of the cache sets to have significantly more cache misses than the other half, but this belongs to capacity misses instead of conflict misses.

In fact, CachePerf’s identification is based on the following observations: (i) Coherence misses typically occur on few cache lines, but not for capacity and conflict misses; (ii) Extensive cache misses are typically caused by few susceptible instructions; (iii) The patterns of memory accesses are necessary to differentiate conflict misses from capacity misses: if multiple memory accesses are accessing the same set of cache lines, then it is an issue of conflict miss; If they are accessing different cache sets, the issue is more likely to be capacity miss.

Observation (i) indicates that coherence misses (e.g., false sharing and true sharing) can be identified by checking the cumulative behavior of cache lines: *if few cache lines (not on the same set) have more cache misses than others, then this issue must be caused by coherence misses*. Like existing work [32], false sharing can be easily differentiated from true sharing using their definitions: if multiple threads are accessing different words of the same cache line, then it is false sharing. Otherwise, it is true sharing. We will use *the Performance Monitoring Unit (PMU)’s address sampling* to collect accesses on a cache line, helping differentiate false sharing from true sharing.

Based on *observation (i) and (ii)*, we propose the **hybrid hardware sampling** to classify cache misses: *the hardware Performance Monitoring Unit (PMU) is employed to collect the coarse-grained samples in order to pinpoint susceptible instructions with extensive cache misses; After that, the breakpoints are further installed on these instructions in order to collect fine-grained memory accesses to understand their memory access patterns*. After collecting memory access patterns, it is possible to differentiate conflict misses from capacity misses using *observation (iii)*.

2.2.2 Differentiating Serious Issues from Minor Ones. Minor issues, although they are not false positives, should be excluded to avoid wasting the time of programmers. Unfortunately, most existing tools [6, 32–35, 39, 46] cannot achieve this goal, as they typically utilize the same absolute metric for different applications, e.g., the number of cache invalidations to evaluate false sharing issues, omitting the temporal effect. However, the same number of cache misses may have different performance impacts for a long-running or short-running program. Further, a program with sparse cache misses and another one with intense misses may benefit differently from the reduction of cache misses, even if they have a similar execution length and cache misses.

CachePerf further proposes two ratio-based mechanisms to exclude minor issues. First, CachePerf proposes a *windowing* mechanism that tracks a specified number of the most recent memory accesses, and then only checks cache misses inside if the miss ratio (i.e., the number of misses divided by the number of accesses) in the past window is larger than a threshold, as discussed in Section 3.2. This windowing mechanism excludes sparse or sporadic cache misses. Second, CachePerf only reports a potential issue if its related memory accesses and cache misses are higher than 0.01% and 1% separately. The access ratio can be utilized to predict the potential performance impact. Assuming that the memory access is $200\times$ slower than the L1 cache access [30], and the access ratio of a potential bug is 0.01% of the total accesses (of the program). We further assume that other accesses of this program can be satisfied at L1 cache, then the total runtime of this program is $0.01\% \times 200X + 99.99\% \times X = 101.9\%X$, if the cycle of L1 cache access is X . Then this bug will introduce at most 2% slowdown, comparing to all accesses are satisfied by the L1 cache ($100\%X$). Similarly, the ratio of cache misses helps prune insignificant instructions.

2.2.3 Differentiating Allocator-Caused Misses from Applications. As discussed in Section 2.1, the allocator may introduce both conflict and coherence misses. When an allocator allocates multiple objects that happen to access few sets of cache lines, it introduces conflict misses. An allocator can introduce false sharing by allocating multiple objects in the same cache line to different threads [3]. CachePerf tracks the allocation information (e.g., the thread, address) that could help differentiate

the bugs caused by applications from those caused by the allocator. To the best of our knowledge, CachePerf is the first work that could report allocator-induced cache misses.

3 DESIGN AND IMPLEMENTATION

This section discusses the detailed design and implementation of CachePerf. CachePerf is designed as a library that can be linked with different applications, without the need of changing and recompiling user programs. In the following, we start with the description of CachePerf’s basic components, and then discuss each component separately.

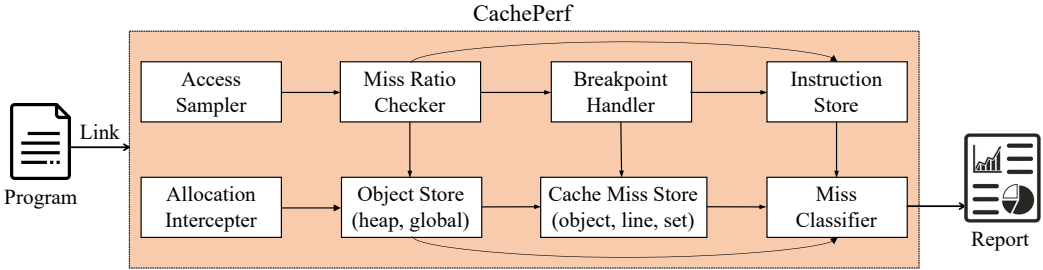


Fig. 3. Basic components of CachePerf

Fig. 3 shows the basic components of CachePerf. As mentioned in Section 1, CachePerf relies on the PMU-based sampling to collect the information of memory accesses and cache misses, which will be handled by its “*Access Sampler*” module. To exclude insignificant cache misses, CachePerf introduces a “*Miss Ratio Checker*” module that computes and checks the cache miss ratio (the percentage of cache misses in all memory accesses). When the cache miss ratio is larger than a predefined threshold (e.g., 0.5%), as further discussed in Section 4.5.2, all recent cache misses will be further updated to “*Miss Store*” and “*Instruction Store*”. Otherwise, all cache misses will be skipped. Due to this filtering mechanism, low-frequency cache misses (such as some compulsory misses) will be excluded automatically. When continuous cache misses from the same instruction are detected or multiple misses are landing on the same cache set, indicating possible capacity or conflict misses, CachePerf further employs breakpoints to collect fine-grained memory accesses information (via “*breakpoint Handler*”), which enables us to differentiate conflict misses from capacity misses.

In order to attribute cache misses to data objects (called “data-centric” analysis [35]), CachePerf further intercepts memory allocations and deallocations, and updates the “*Object Store*” correspondingly. “*Object Store*” tracks address ranges and callsites of heap objects. In the end, CachePerf classifies cache misses by integrating the data in “*Miss Store*” and “*Instruction Store*”, and finally reports helpful information based on “*Object Store*”, including the allocation call sites, object size, and object name (only for global objects). Different from existing tools [6, 46], there is no need for offline analysis, i.e., it has no hiding overhead.

3.1 Access Sampler

For the access sampler, CachePerf employs the Performance Monitoring Units (PMU) to sample memory accesses. The PMU is the ubiquitous hardware in modern architectures (e.g., X86 or ARM) that can provide hundreds of hardware events [16]. There is a trend for profilers to build on top of the PMU [5, 17, 22, 33, 36, 37, 39, 50], due to its low overhead. Currently, Linux also provides a system call `-perf_event_open-` that allows to configure and start the PMU easily.

CachePerf samples two types of events, including memory loads and stores. The configuration for the PMU sampling is shown in Table 1, which is based on Intel’s Xeon machine. To balance the

Configuration	Load Sampling	Store Sampling
type	PERF_TYPE_RAW	
config	0x1cd	0x82d0
sample_period	20000 ($\pm 10\%$)	50000 ($\pm 10\%$)
freq	false	
sample_type	PERF_SAMPLE_IP PERF_SAMPLE_ADDR PERF_SAMPLE_DATA_SRC	
precise_ip	3	1
__reserved_1	3	0
config1	3	0

Table 1. Configuration of the PMU sampling

detection effect on loads and stores, we empirically set the sampling period of loads as 20,000, and the one of stores as 50,000, which has been evaluated in Section 4.5. To avoid different threads from sampling the same instructions, we introduce 10% randomized variance for each thread’s sampling period. Note that it is important to include `PERF_SAMPLE_DATA_SRC` in the sample type so that we can know which level the corresponding instruction is hit, such as L1, L2, LLC, or memory. It is also referred to as “hit information” in the remainder of this paper.

CachePerf employs the following information of the sampling: the type of access (e.g., load or store), hit information, memory address, and instruction pointer (IP). Among them, the hit information helps identify all cache misses from all sampled memory accesses, where all accesses that do not hit on the L1 cache will be treated as cache misses. IP tells the instruction performing the corresponding access, and the memory address helps pinpoint which cache line and cache set have the miss, enabling us to perform the classification.

3.2 Miss Ratio Checker

A miss ratio checker is introduced to filter out sparse cache misses. As mentioned above, since sparse cache misses may not incur significant performance slowdown, they should be excluded in order to avoid wasting the effort of fixing such issues. Further, the filtering reduces the memory overhead of storing such cache misses and the performance overhead of spending in classification.

In the implementation, CachePerf maintains two circular buffers to track the most recently sampled memory accesses for each thread, one buffer for memory loads and the other one for memory stores. These buffers are updated in First-In-First-Out order that the later accesses will overwrite the least-recent memory accesses. CachePerf computes the cache miss ratio upon every access via dividing the number of cache misses by that of accesses. Only when the miss ratio of the buffer is larger than a predefined threshold (e.g., 0.5%), all cache misses in the current buffer will be handled and be updated to “*Instruction Store*” and “*Miss Store*”. Otherwise, they will be skipped. The Instruction Store holds the information related to instructions, such as the number of accesses and cache misses. The Miss Store maintains the detailed information about each cache miss, e.g., object, line, and set.

3.3 Breakpoint Handler

As mentioned in Section 2, CachePerf employs the breakpoints to collect fine-grained memory accesses of the selected instructions, enabling us to differentiate conflict misses from capacity misses. For the susceptible instructions, CachePerf focuses on two types of instructions: (1) instructions introducing multiple continuous cache misses, indicating that they may incur extensive cache misses.

(2) instructions introduce extensive misses on the same set in a time window, which are potential candidates for conflict misses.

After identifying these instructions, CachePerf installs hardware breakpoints via the `perf_event_open` system call by specifying the `type` to be "PERF_TYPE_BREAKPOINT" and the `bp_type` to be "HW_BREAKPOINT_X". After the installation, every time a program executes such an instruction, CachePerf will be interrupted so that it could collect the fine-grained memory accesses of each instruction. However, the interrupt handler provides no information about the memory address, as the breakpoint is typically triggered before the access. CachePerf infers the memory address by analyzing the corresponding instruction. For example, if the instruction is "addl \$0x1, -0x4(%rbp)", then CachePerf could infer the stored memory address via the value of register and `rbp`. CachePerf employs Intel's `xed` library [14] to perform the binary analysis.

To simplify the handling, CachePerf only installs one breakpoint for all threads at a time, collecting all accesses from different threads. To reduce the overhead caused by handling endless interrupts, CachePerf only collects at most 64 accesses from one instruction. If there are 8 accesses landing on the same set, then it is identified as a bug with conflict misses. Otherwise, it is a bug with capacity misses. Based on this, if 8 continuous accesses are landing on the same cache set, which can be clearly identified as "conflict misses", CachePerf will remove the breakpoint so that it could monitor other instructions.

However, it is possible that an instruction has no or few accesses after the installation. When new instructions require to be monitored, CachePerf further introduces an expiration mechanism that a breakpoint will be expired after 100ms. In this way, CachePerf is able to install breakpoints on new instructions. The identification of cache misses is further discussed in Section 3.5.

3.4 Important Data Stores

CachePerf maintains *Object Store*, *Miss Store*, and *Instruction Store*, as further described below.

Object Store. Object Store tracks the information of two types of objects, heap objects and global objects, as most cache misses occur on these objects, which are handled differently.

For heap objects, CachePerf intercepts all memory management functions, such as `malloc()` and `free()`, in order to track their corresponding callsites. For each heap object, CachePerf tracks its size, callsite, and address range. As there are large amounts of heap objects, the Data Store should be carefully designed in order to support the following operations efficiently: adding and updating of an object via the starting address upon memory allocations and deallocations, and searching by an address in the range of a valid object upon each sampled access. Although the hash table can support the adding and updating operations efficiently via the starting address (as the key), it is expensive to search the memory address inside heap objects (different from the key). Instead, an ordered list/array supports the searching better via the binary search. Furthermore, we observe that heap objects are typically classified into small, medium, and large sizes, where the number of small-size objects is much larger than that of medium-size and large-size objects.

Based on these observations, CachePerf designs a three-level data store as shown in Fig. 4 to support efficient adding, updating, and searching. In particular, any object can be stored in one of Page Table, Chunk Table, and Sorted Huge Objects List, which are mutually exclusive. CachePerf updates these tables/lists as follows: (1) if an object exists only in a single page, it is stored in the *Page Table*; (2) If the range of an object crosses two different pages but within the same megabyte, it is stored in the *Chunk Table*; (3) Otherwise, it is inserted into the *Sorted Huge Objects List*. For each address, CachePerf always searches the Page Table at first (with the highest possibility), then the Chunk Table, and finally the Sorted Huge Objects List, and stops if the object is found already. Although three searches are required for some objects, however, we believe that such searches will

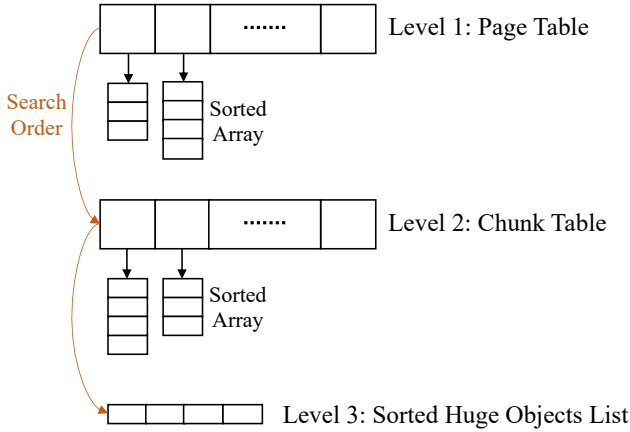


Fig. 4. A three-level object store that combines with the shared memory and the sorted array/list.

be fewer than others. This design is based on the assumption that small objects (less than 1-page size) are typically significantly more than large objects.

For the performance reason, each entry of *Page Table* and *Chunk Table* stores a pointer pointing to a sorted array that stores all allocated objects inside the same page (4KB) and chunk (1MB). If an entry is empty (with the NULL value), then there are no objects in the corresponding page or chunk, indicating the unnecessary of searching for a higher-level store. Both tables are employing the shadow memory [48, 58] to store these pointers, where the index of each entry could be computed simply with a bit-shifting operation. Since the number of huge objects is typically small, a sorted list is used to store huge objects, which is not using the shadow memory. For the sorted arrays and lists, the search can be done efficiently via the binary search.

CachePerf also proposes a callsite-based optimization to reduce the overhead, especially on the updates of memory allocations and sampled accesses: if objects from a particular callsite have a much lower cache miss ratio, compared to the average one, then all allocations and cache misses from this callsite could be safely skipped. Based on our observation, such an optimization reduces the overhead by over 30% for a particular application (Canneal of PARSEC [4]).

CachePerf handles global objects differently, as an application typically has a small number of global objects and they are not increased during the execution. All accesses of global objects (e.g., addresses) will be stored in a hash table. CachePerf obtains the name and address range by analyzing the corresponding ELF header, and then computes the miss ratio of each object, as discussed in Section 3.5.

Miss Store. The Miss Store saves the information of each cache miss, which is not filtered out as described by Miss Ratio Checker (Section 3.2). In particular, cache misses are stored in two separate data structures: an array (with the size of the number of cache sets) stores the information of each cache set, and a hash table stores the information of each cache line (using the starting address as the key). For both data structures, CachePerf stores the number of cache misses (on each cache set and each cache line). For cache lines, CachePerf further stores the thread information for accessing each word, which could help differentiate false sharing from true sharing.

Instruction Store. Instruction Store saves the information of memory accesses (e.g., loads and stores) and cache misses of the selected instructions by the miss ratio checker (as discussed in Section 3.2). The data structure of Instruction Store is a hash table that uses the instruction pointer as

the hash key. For each instruction, CachePerf records the number of cache misses, the related cache set, and the detailed memory access pattern.

Since each line/statement of the code may be related with multiple instructions (at the assembly level), CachePerf further summarizes cache misses of the related statement, and only reports statements with extensive cache misses.

3.5 Miss Classifier

Algorithm 1: The Algorithm of Classifying Cache Misses

```

for cache line c in Miss Store do
  if multiple threads access the same words of c then
    | Report true sharing
  end
  if multiple threads access different words of c then
    | if c has multiple objects allocated by different threads then
      | Report allocator-induced false sharing
    else
      | Report application's false sharing
    end
  end
end
end
for instruction i in Instruction Store do
  if the issue is reported as coherency miss then
    | continue
  end
  if i's misses land on the same cache set then
    | if misses are landing on multiple heap objects then
      | Report allocator-induced conflict miss
    else
      | Report application's conflict miss
    end
  else
    | Report application's capacity miss
  end
end
end

```

CachePerf classifies and reports serious cache misses by combining the information from Cache Miss Store and Instruction Store together. The detailed algorithm is shown as Algorithm 1. CachePerf omits cache misses without significant performance impacts. Instead, it focuses on instructions or cache lines that have passed the “ratio-based filtering”: (1) for an application, if the number of load misses is less than 3% of all load accesses and the number of store misses is less than 1% of all store accesses, then CachePerf will not report any issue; (2) for each instruction, if its memory accesses are less than 0.01% of total accesses, or its cache misses are less than 1% of total misses, this instruction will not be reported; (3) for each cache line and each cache set, it will be reported only if its misses larger than 1% of all misses. These numbers are set based on our experience, which has been evaluated as Section 4.5.

CachePerf reports potential coherence misses by checking all cache lines in Miss Store. As discussed in Section 2.2.1, few cache lines with extensive cache misses but not mapping to the same

cache set can be caused by coherency misses. For each cache line, CachePerf can further determine the type, false sharing and true sharing, via word-level information of the corresponding cache lines. If multiple threads are accessing the same words of the cache line, then it is true sharing of applications. Otherwise, it is a false sharing problem. CachePerf further checks whether multiple objects on the same cache line are allocated by different threads or not. If yes, then it is allocator-induced false sharing. Otherwise, it is the application’s false sharing. If cache lines are identified as coherency misses, the corresponding instructions will be marked as checked, which will be excluded for identifying conflict misses and capacity misses afterward.

CachePerf differentiates capacity misses from conflict misses based on the memory access pattern of each instruction (with extensive cache misses) in the Instruction Store. A simple mechanism is employed to differentiate conflict misses from capacity misses: if the number of accesses mapping to the same cache set is larger than a threshold (e.g., 8), then the corresponding cache misses will be considered as conflict misses. Otherwise, they are capacity misses. For conflict misses, CachePerf further checks whether they are caused by the allocator or not: if they are involved with multiple heap objects, this belongs to allocator-induced conflict miss. Otherwise, it is an application’s conflict miss.

CachePerf could further report the detailed information of cache misses, including the instruction information (from Instruction Store) and object information (from Object Store). The former one tells which instructions introduce cache misses, while the latter one helps locate the heap object with its allocation callsite. This information could guide bug fixes. For instance, if two objects mapping to the same cache set introduce excessive cache misses, such an issue can be significantly reduced by changing the address of objects (by mapping to different sets).

4 EXPERIMENTAL EVALUATION

The experimental evaluation will answer the following research questions:

- How is the effectiveness of CachePerf? (Section 4.2)
- What is the performance overhead of CachePerf? (Section 4.3)
- What is the memory overhead of CachePerf? (Section 4.4)
- What are the impacts of different configurations? (Section 4.5)

4.1 Experimental Setting

Hardware Platform: Experiments are evaluated on a two-processor machine, where both processors are Intel(R) Xeon(R) Gold 6230 with 20 cores. We only enabled 16 hardware cores in one node to exclude the NUMA impact as it is outside the scope of this paper. The machine has 256GB of main memory, 64KB L1 cache, and 1MB of L2 cache.

Software: The OS is Ubuntu 18.04.3 LTS, installed with Linux-5.3.0-40. The compiler is GCC-7.5.0, while we are using `-O2` and `-g` flags for the compilation.

Evaluated Applications: Two types of applications are included in the evaluation, including general applications and applications known to have cache misses. In particular, all 13 applications from the PARSEC-2.0 benchmark are included as general applications [4], but some also have known bugs. Buggy applications with coherence misses (false sharing) include two stress tests `cache-scratch` and `cache-thrash` from Hoard [3], and two Phoenix [45] applications (`histogram` and `linear_regression`). Among them, the first two applications actually have false sharing caused by the allocator. Five applications with conflict misses are collected from CCProf [46]: `ADI` [44], `HimenoBMT` [13], `Kripke` [27], `MKL-FFT` [9], and `NW` [7]. `TinyDNN` [54] is not included, since we did not observe conflict misses and the change (based on CCProf [46]) did not improve the performance. We also include `irs` [28] and `SRAD` [56] applications that were employed by ArrayTool [35] to

evaluate capacity misses. Note: to reproduce false sharing on our machine, `histogram` processes a special BMP file adapted from the original one that all of the red values are set to 0 and the blue values are set to 255. For `linear_regression`, we also use the `volatile` keyword for the `args` variable in order to avoid the optimization of the compiler. For `HimenoBMT`, the grid size is medium and the number of integration is 80. `NW`'s matrix dimension is set to be 16384×16384 , and its penalty is set to be 10.

Evaluated Allocators: To evaluate CachePerf's detection on issues introduced by allocators, we evaluate on two widely-used allocators, `Glibc-2.28` and `TCMalloc-4.5.3`. `Glibc-2.28` includes the default allocator in our machine, and `TCMalloc` is a widely-used allocated designed by Google [12].

Comparison: We compare CachePerf with two state-of-art tools in effectiveness, performance, and memory consumption. One is `CCProf` which detects cache conflict misses [46], and the other one is `Feather` for false sharing detection [6]. We have difficulty running `ArrayTool` [35] successfully, which is the reason why `ArrayTool` is not included for comparison. For these tools, we use their default sampling rates used for their evaluation.

4.2 Effectiveness

We list the effectiveness results of CachePerf's detection in Table 2. Overall, *CachePerf reports all known bugs and detects 9 new bugs, while fixing the reported bugs achieves the performance improvement between 3% and 3788%*. Some applications with capacity misses cannot be easily fixed, marked as “?” in the “Improve” column. This also concurs with our discussion in Section 2.1 that not all capacity misses can be fixed easily. CachePerf correctly identifies all types of bugs, except bug 7 in `ADI`. The type is identified by CachePerf as capacity miss, but it is actually conflict miss. Based on our investigation, the failure of the identification is caused by the skids of the PMU hardware [1]. The PMU hardware fails to pinpoint the exact instruction with the sampled cache miss, with the distance of one instruction. Therefore, CachePerf actually captures the access pattern of an instruction different from the one with cache misses, which does not have the pattern of conflict misses. However, our observation that “*an instruction's access pattern is not changed during the whole execution*” still holds.

Note that although `streamcluster` has been reported by previous tools with a false sharing issue, but achieving no performance improvement after fixing the bug as suggested by previous tools [32]. CachePerf successfully avoids the report of this bug, therefore, preventing programmers to spend the effort on this bug. In contrast, `Feather` still reports this insignificant bug, which is the reason why it is marked as “X”. `Feather` cannot report the origin of false sharing in both `cache-scratch` and `cache-thrash`, which are allocator-induced conflict misses. Similarly, although `CCProf` reports conflict misses of `raytrace`, but it fails to identify as an allocator-induced miss.

4.2.1 Conflict Misses of Applications. CachePerf could correctly report all known conflict misses, including `ADI`, `HimenoBMT`, `Kripke`, `MKL-FFT`, and `NW`. These bugs can be fixed by switching the order of loops (`Kripke`) and using the padding (others).

CachePerf further detects three unknown conflict misses, in `ADI`, `SRAD`, and `swaptions`. The bug of the `SRAD` application is shown in Fig. 5, which can be detected by `CCProf`. CachePerf reports that line 243 of `main.c` introduces around 64% of load misses. As shown in Fig. 5(a), `SRAD` uses two nested for loops to calculate the sum for every pixel in the image ROI. By simply switching these two loops, we improve the performance by 748%.

4.2.2 Allocator-Induced Conflict Misses. CachePerf also detects a serious conflict miss in `raytrace` caused by the default allocator-`glibc-2.28`, as shown in Fig. 6. The report can be seen in Fig. 6(b).

Category	Index	Application	Improve	CCProf	Feather	CachePerf	New
False Sharing	1	cache-scratch*	1007%	✗	✗	✓	✓
	2	cache-thrash*	3788%	✗	✗	✓	✓
	3	histogram	117%	✗	✓	✓	
	4	linear_regression	712%	✗	N/A	✓	
	5	streamcluster	0%	✗	✗	✓	
Conflict Miss	6	ADI	246%	✓	✗	✓	
	7	ADI	18%	✗	✗	✗	✓
	8	HimenoBMT	964%	✓	✗	✓	
	9	Kripke	7%	N/A	N/A	✓	
	10	MKL_FFT	52%	✓	✗	✓	
	11	NW	245%	✓	✗	✓	
	12	raytrace*	27%	✗	✗	✓	✓
	13	SRAD	748%	✓	✗	✓	✓
14	swaptions	3%	✗	✗	✓	✓	
Capacity Miss	15	bodytrack	?	✗	✗	✓	✓
	16	canneal	?	✗	✗	✓	✓
	17	IRS	33%	✗	✗	✓	
	18	SRAD	12%	✗	✗	✓	
	19	streamcluster	?	✗	✗	✓	✓

Table 2. This table lists applications with cache misses. For applications marked with *, `cache-scratch`, `cache-thrash` have allocator-induced false sharing, and `raytrace` has allocator-induced conflict misses. Column “Improve” lists the performance improvement after fixes based on information provided by CachePerf, where column “New” indicates whether it is first discovered by CachePerf. Further, “✓” indicates the tool correctly detects the issue, “✗” indicates an imperfect report, “✗” indicates a failed detection, and “N/A” indicates that the corresponding application crashes or deadlocks when running with the tool. Note that applications marked as “?” in “Improve” cannot be fixed easily, which confirms our discovery in Section 2.1.

<pre> 241: for (I = r1; i <= r2; i++) { 242: for (j = c1; j <= c2; j++) { 243: tmp = image [i + Nr * j]; 246: } 247: }</pre>	<pre> Application's conflict misses, accessed by the instruction at: - main.c: 243 Related to the heap object (419430400 bytes) allocated at: - # 0: main.c: 143</pre>
(a) Source code (main.c)	(b) CachePerf's report

Fig. 5. The conflict miss in SRAD, which can be fixed easily by switching the loops of line 241 and 242.

For this problem, the default `glibc-2.28` happens to allocate many 48-byte objects mapping to the same cache set, causing conflict misses. `TCMalloc` does not have this issue, which runs about 27% faster on this application than `glibc-2.28`. We further confirm whether there exists a systematic method in `TCMalloc` to prevent such an issue. We find that `TCMalloc` always requests two pages at a time, then allocates objects (48 bytes) continuously, and skips non-used bytes in the end. This mechanism luckily avoids conflict misses of `raytrace` application. In summary, allocator-induced conflict misses are not easy to prevent from the design of the allocator. This also shows the importance of CachePerf that could help identify the root cause of performance slowdown. After finding out the issue, programmers may switch to a different allocator, or change the application by introducing unnecessary allocations inside or changing the alignment of the related structure.

4.2.3 Capacity Misses of Applications. We borrowed some buggy applications from ArrayTool [35], including IRS [28] and SRAD [56]. The paper also reports serious issues in a specific version of

<pre> 139: _INLINE int getVertexID (int vtxID,int norID, int txtID) 140: { 145: vertex.push_back(tmpVtx[vtxID]); 151: } </pre>	<pre> Allocator's conflict misses, accessed by: #0 instruction at: - /usr/include/c++/7/bits/stl_tree.h: 1875 #1 instruction at: - # 0: /usr/include/c++/7/bits/stl_pair.h: 456 Related to multiple heap objects (48 bytes) allocated at: - # 0: /usr/include/c++/7/tuple: 1652 - # 9: MiniView/ObjParser.hxx: 145 </pre>
(a) Source code (ObjParser.hxx)	(b) CachePerf's report

Fig. 6. CachePerf reports an allocator-caused conflict miss in raytrace

LULESH [23]. However, we cannot find the exact source code, which is the reason why LULESH is not included. Besides these applications, CachePerf also detects unknown capacity misses in *bodytrack*, *canneal*, and *streamcluster*, which has been confirmed by us manually. However, as mentioned in Section 2.1, not all capacity misses could be fixed easily.

As shown in Table 2, CachePerf successfully reports capacity misses hidden in both IRS and SRAD. As an example, the IRS's source code and report are shown in Fig. 7. IRS's capacity misses occur in line 239 of *aos3.cpp*, although *addr2line* actually reports lines between 239 and 247. This statement accesses many objects of the same size (88824176 bytes), e.g., *dbl*, *xdbl*, *dbc*. Since every object has exactly the same access pattern, these accesses should be grouped together. Using the suggested fix strategy [35], the performance can be improved by 32.7%.

<pre> 235: for (kk = kmin; kk < kmax; kk++) { 236: for (jj = jmin; jj < jmax; jj++) { 237: for (ii = imin; ii < imax; ii++) { 238: i = ii + jj * jp + kk * kp; 239: b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] + 240: dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i] * xdcr[i] + 241: dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i] + 242: cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i] + 243: ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xcrcr[i] + 244: cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i] + 245: ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i] + 246: ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i] + 247: ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i]; 248: } 249: } 250: } </pre>	<pre> Application's capacity misses, accessed by: #0 instruction at: - aos3.cpp: 239 #1 instruction at: - aos3.cpp: 240 Related to multiple heap objects (8824176 bytes) allocated at: # 0 object is allocated at: - # 0: aos3.cpp: 275 # 1 object is allocated at: - # 0: aos3.cpp: 278 </pre>
(a) Source code (aos3.cpp)	(b) CachePerf's report

Fig. 7. Reported capacity miss in IRS

Note that CachePerf cannot report SRAD's capacity miss in the original version when the conflict miss (as shown in Fig. 5) is the dominant performance issue. We also confirmed that applying the suggested fix by ArrayTool [35] achieves almost no performance improvement. In fact, this actually illustrates the effectiveness of CachePerf as its rule-based filtering mechanism avoids reporting minor issues. After fixing the conflicting miss of SRAD, then CachePerf could successfully report the capacity miss. After fixing the report bug, SRAD's performance is improved by 12.4% finally.

4.2.4 Coherency Misses (FS) of Applications. For coherency misses of applications, we utilize three known buggy applications to evaluate CachePerf's effectiveness, including *histogram*, *linear_regression*, and *streamcluster*. CachePerf successfully detects these issues latent in *histogram*

and `linear_regression`, similar to existing work [6, 32]. We show the source code and CachePerf’s report of `linear_regression` in Fig. 8. This is a known bug that the structure of `args` is not aligned to 64 bytes (but only 52 bytes instead). As a result, thread 1 will access the same cache line as thread 2. By simply aligning the related structure, the performance can be improved by 712%. Different from existing tools, CachePerf will not report the issue of `streamcluster`, although it was reported to have false sharing for the `work_mem` object [32]. Based on existing work, we fixed the false sharing by using the padding and observed the reduction of cache misses. However, we do not observable performance impact with this change, less than 1%. That is, CachePerf successfully excludes the insignificant issue, avoiding the waste of manual effort. In contrast, Feather still reports this false sharing of `streamcluster`, although it only imposes little performance impact.

<pre> 91: for (i = 0; i < args->num_elems; i++) 92: { 93: //Compute SX, SY, SYY, SXX, SXY 94: args->SX += args->points[i].x; 95: args->SXX += args->points[i].x*args->points[i].x; 96: args->SY += args->points[i].y; 97: args->SYY += args->points[i].y*args->points[i].y; 98: args->SXY += args->points[i].x*args->points[i].y; 99: } </pre>	<pre> Application’s false sharing, accessed by: #0 instruction at: - linear_regression-pthread.c: 94 #1 instruction at: - linear_regression-pthread.c: 97 Related to the heap object (1056 bytes) allocated at: - # 0: linear_regression-pthread.c: 155 </pre>
(a) Source code (<code>linear_regression-pthread.c</code>)	(b) CachePerf’s report

Fig. 8. CachePerf reports false sharing in `linear_regression`

4.2.5 Allocator-Induced False Sharing. When using the default allocator, CachePerf also reports allocator-induced false sharing as shown in Fig. 9. CachePerf infers allocator-induced false sharing as more than two objects allocated by different threads are located in the same cache line, as shown in Fig. 9 (b). A simple solution is to change the alignment of the structure related to `obj`, which improves the performance by 1007%. CachePerf also reports some serious allocator-caused false sharing issues for both `cache-scratch` and `cache-thrash` with the `TCMalloc` allocator. There will be 3788% performance improvement using the padding. Although allocator-caused false sharing is a bug of the allocator design, it can be prevented by changing the application itself. In addition, users could switch to a new allocator to fix such issues. CachePerf provides helpful information that could help fix such bugs.

<pre> 81: char * obj = new char[w1._objSize]; 82: for (int j = 0; j < w1._repetitions; j++){ 83: for (int k = 0; k < w1._objSize; k++) { 84: obj[k] = (char) k; 85: 87: } 88: } </pre>	<pre> Allocator’s false sharing, accessed by the instruction at: - cache-scratch.cpp: 84 Related to multiple heap objects allocated by different threads: # 0 object (8 bytes) is allocated at: - # 0: cache-scratch.cpp: 81 # 1 object (8 bytes) is allocated at: - # 0: cache-scratch.cpp: 81 </pre>
(a) Source code (<code>cache-scratch.cpp</code>)	(b) CachePerf’s report

Fig. 9. CachePerf reports coherence misses in `cache-scratch`

Comparing with Other Tools: Overall, CachePerf shows **three obvious advantages** when compared with existing tools. First, CachePerf can detect multiple types of cache misses, while others could only report a specific type of cache misses. Note that the other tools are mutually exclusive,

forcing programmers to use them one after the other. Second, CachePerf is the only tool that identifies the performance issues introduced by the memory allocator, preventing programmers from wasting the unnecessary effort of improving applications but achieving no performance improvement. Finally, CachePerf is the only tool excluding minor issues with little performance impact, saving users' time.

4.3 Performance Overhead

We evaluated the performance overhead of CachePerf, CCProf, and Feather. Since CCProf and Feather have online and offline stages, we add their overhead of two stages together. The results (with the AVERAGE and GEOMEAN) are shown in Fig. 10.

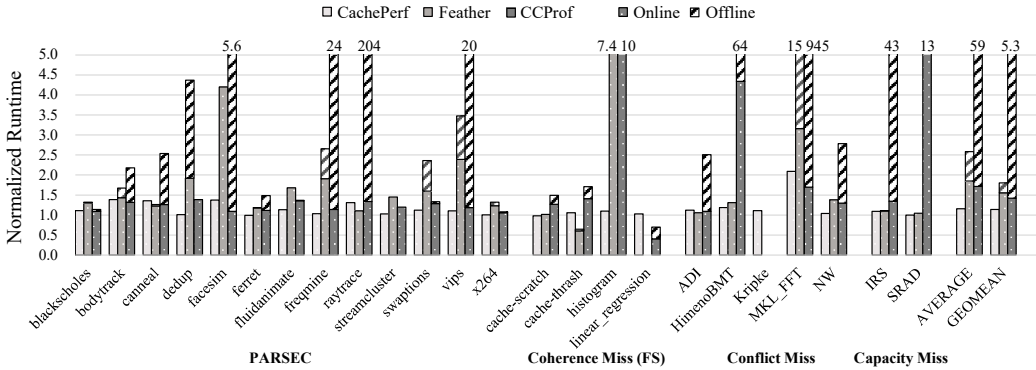


Fig. 10. The performance of CachePerf, CCProf, and Feather, where the results are normalized to the default setting (without running any tool).

On average (GEOMEAN), CachePerf introduces 14% performance overhead, while CCProf's overhead is $5.3\times$ and Feather's overhead is 80% if considering both online profiling and offline analysis. Even only considering their online profiling, CachePerf is still faster than both CCProf and Feather. Based on our understanding, CachePerf's ratio-based mechanism helps reduce much unnecessary overhead by pruning sporadic cache misses, but without compromising its effectiveness. CachePerf's specific data structures also help reduce the overhead. CachePerf's hybrid sampling technique that combines with both coarse-grained and fine-grained sampling also balances the accuracy and the overhead. On one hand, its coarse-grained sampling works as a filter that allows it to focus on only a few susceptible instructions, avoiding installing unnecessary breakpoints. On the other hand, the breakpoints effectively ensure the precision of the tool even with a low PMU-based sampling rate.

MKL_FFT is the only application with an overhead higher than 100%. We confirmed that more than 80% of its overhead is spent in its reporting phase, which could be placed offline if necessary. This application has involved a big amount of cache sets, heap objects, and instructions. For instance, CachePerf requires to invoke the expensive `addr2line` to obtain the line numbers for many lines. We are planning to reduce such overhead with heuristics in the future.

CachePerf introduces 36% performance overhead for `canneal`. The basic reason is that `canneal` has a great number of allocations (about 1.3 million per second), where keeping the information of these objects adds significant overhead (and memory overhead). Similarly, CachePerf introduces high overhead for keeping and updating the information of objects for `raytrace`, as there are around 500 thousand memory allocations each second. CachePerf introduces high overhead for `bodytrack` and `facesim` for a similar reason.

Category	Application	Default	CachePerf	CCProf	Feather
PARSEC	blackscholes	614	632	1850	622
	bodytrack	34	70	1397	44
	canneal	851	1728	2089	860
	dedup	1513	1647	2811	1549
	facesim	311	390	2776	330
	ferret	108	142	1343	116
	fluidanimate	209	234	2675	218
	freqmine	1280	1312	3736	1289
	raytrace	1287	1319	3213	1295
	streamcluster	112	238	2572	122
	swaptions	7	36	1244	16
	vips	55	80	1234	108
	x264	482	514	1711	510
	Coherence Miss (False Sharing)	cache-scratch	3	22	1551
cache-thrash		4	28	1810	11
histogram		1344	1362	2574	1336
linear_regression		1956	1974	3185	N/A
Conflict Miss	ADI	514	528	1743	520
	HimenoBMT	225	302	1455	232
	Kripke	301	360	N/A	N/A
	MKL-FFT	261	283	1490	268
	NW	2050	2068	3279	2058
Capacity Miss	IRS	248	342	3118	256
	SRAD	2404	2420	4869	2412
TOTAL		16174	18032	53726	14182
GEOMEAN			151%	889%	122%

Table 3. The memory consumption (MB) of CachePerf, CCProf, and Feather. Column “Default” lists memory footprints of applications when running alone.

We further checked the reason why Feather runs faster with `cache-thrash` and CCProf runs faster with `linear_regression`. Based on our investigation, Feather allocates some memory from the default memory allocator for its internal usage, which happens to alleviate the false sharing issue introduced by the allocator. Similarly, CCProf’s memory usage also changes the starting address of the false sharing object, reducing the severity of false sharing. That is, they should impose higher overhead if such lucky cases are excluded. We also observe that CCProf’s offline phase is very expensive, e.g., `MKL-FFT`, which could be as much as $945\times$ higher. In contrast, CachePerf does not have the hidden overhead for the offline analysis, which could report cache misses immediately after the execution or when receiving the signal from users (good for long-running applications).

4.4 Memory Overhead

We also evaluated the memory overhead of CachePerf, CCProf, and Feather, as shown in Table 3. Since CCProf crashed for `Kripke`, and Feather encountered the deadlock for `Kripke` and `linear_regression`, these applications are marked “N/A” in the table.

In total, CachePerf adds around 11% memory consumption, although its average overhead is around 51%. When only considering the online stages of CCProf and Feather, CachePerf’s memory overhead is significantly better than CCProf, but slightly worse than Feather. However, if the offline stage is also considered when using the maximum memory consumption of both stages, then CachePerf has the smallest memory consumption.

Table 3 shows that CachePerf introduces high memory overhead for applications with small memory footprints, such as `swaptions`, `cache-scratch`, and `cache-thrash`. Based on our observation,

the overhead is introduced by CachePerf’s initialization overhead for its pre-defined hash tables. However, CachePerf only introduces around 19% memory overhead on average for applications with large footprints (e.g., > 100MB). Considering the functionalities provided by CachePerf, we believe that the memory overhead of CachePerf is reasonable and acceptable.

4.5 Impact of Configurations

In this section, we investigate the performance and effectiveness impacts of different configurations using all PARSEC applications. We investigate the impact of the sampling rate, thresholds of Miss Ratio Checker, breakpoint configurations, and access/miss ratio.

4.5.1 Sampling Rate. We evaluated three sets of sampling periods as shown in Table 4. With its default setting (marked as bold), CachePerf’s GEOMEAN performance and memory overhead are 14% and 48% separately. When the sampling frequencies are 10 times lower than the default setting, the performance overhead is 10% and the memory overhead is 47%. However, as shown in the “CP1” column of Table 5, CachePerf will miss 10 out of 19 issues. When the sampling frequencies are 10 times higher than the default setting, the performance and memory overhead are increased to 18% and 79% correspondingly, but do not report more issues. Overall, CachePerf’s default sampling periods keep a good balance between performance and effectiveness.

	L:200K, S: 500K Miss Ratio: 0.5%	L:20K, S: 50K Miss Ratio: 2.5%	L:20K, S: 50K Miss Ratio: 0.5%	L:2K, S: 5K Miss Ratio: 0.5%	L:20K, S: 50K Miss Ratio: 0%
Performance	10%	12%	14%	18%	18%
Memory	47%	45%	48%	79%	70%

Table 4. This table lists the performance and memory overhead under different sampling periods (“L” is the load sampling period and “S” is the store sampling period) and different thresholds of the Miss Ratio Checker (“Miss Ratio”) using all PARSEC applications. The middle column (in bold) is the default configuration.

4.5.2 Threshold of Miss Ratio Checker. We further investigate the impacts of different thresholds of the Miss Ratio Checker. CachePerf will handle all cache misses inside the buffers, when the cache miss ratio is larger than the pre-defined threshold. As described in Section 3.2, the default threshold is 0.5%. In the default setting, CachePerf’s performance and memory overhead are 14% and 48% separately. When the threshold is increased to 2.5%, indicating CachePerf will only handle all cache misses when there are 25 misses out of 1000 accesses, the performance overhead is 12% and the memory overhead is 45%, as shown in Table 4. However, as shown in “CP2” in Table 5, CachePerf will miss 5 issues under this configuration. Another setting is 0%, indicating CachePerf will handle all cache misses in the buffer, the performance and memory overhead is 18% and 70% correspondingly. However, this setting does not report more issues. Overall, the default threshold of Miss Ratio Checker has a good balance between overhead and effectiveness.

4.5.3 Breakpoint Configuration. We also evaluated the overhead and effectiveness impacts of different breakpoint configurations. As discussed in Section 3.3, CachePerf collects at most 64 accesses from one selected instruction, and identifies the bug as the conflict miss when more than 8 accesses are landing on the same cache set. That is, CachePerf will remove the breakpoint on this instruction if 8 continuous accesses are from the same set. Besides this default setting, we also evaluated using 4 or 16 accesses as the condition for identifying the conflict miss. We also evaluated different expiration time for the breakpoint installing on an instruction, such as 10ms and 1000ms, where the breakpoint will be installed for a new instruction. However, we do not observe a significant difference in overhead or effectiveness for different configurations.

Index	Application	Instructions	Allocation Callsite	CP	CP1	CP2
1	cache-scratch*	cache-scratch.cpp: 84	cache-scratch.cpp: 81	✓	✓	✓
2	cache-thrash*	cache-thrash.cpp: 84	cache-thrash.cpp: 75	✓	✓	✓
3	histogram	histogram-pthread: 126, 132	histogram-pthread: 231	✓	✗	✓
4	linear_regression	linear_regression-pthread.c: 94, 97	linear_regression-pthread.c: 155	✓	✗	✓
5	streamcluster	streamcluster.cpp: 1005, 1015, 1098, 1099	streamcluster.cpp: 985	✓	✓	✓
6	ADI	adi.c: 104	utilities/polybench.c: 524	✓	✓	✓
7	ADI	adi.c: 109	utilities/polybench.c: 524	✗	✗	✗
8	HimenoBMT	himenoBMTxpa.c: 295-316	himenoBMTxpa.c: 231	✓	✗	✓
9	Kripke	Grid.cpp: 262	SubTVec.h: 54	✓	✗	✗
10	MKL_FFT	MKL Library	basic_dp_xx_2d_4096.c: 88, 95	✓	✗	✓
11	NW	needle.cpp: 130, 191, 290	needle.cpp: 262, 263	✓	✗	✗
12	raytrace*	C++ STL Library	MiniView/rtview.cxx: 410	✓	✓	✓
13	SRAD	main.c: 243	main.c: 143	✓	✓	✓
14	swaptions	HJM_SimPath_Forward_Blocking.cpp: 121	nr_routines.cpp: 168	✓	✗	✗
15	bodytrack	ImageMeasurements.cpp: 43	AsyncIO.cpp: 55	✓	✗	✓
16	canneal	C++ STL Library	netlist.cpp: 236	✓	✓	✓
17	IRS	aos3.cpp: 239-247	aos.cpp: 275-304	✓	✓	✓
18	SRAD	main.c: 312	main.c: 188-191	✓	✗	✗
19	streamcluster	streamcluster.cpp: 652	streamcluster.cpp: 1862	✓	✗	✗

Table 5. This table lists the effectiveness of CachePerf under different configurations. “CP” is the default setting, with the load and store sampling periods to be 20K and 50K separately. “CP1” has 10 times lower sampling frequencies (200K and 500K separately), but will miss 10 cases. In “CP2”, its miss ratio checker uses a higher threshold (2.5%), which will miss 5 cases. “✓”, “✗”, and “✗” have the same meaning as Table 2.

4.5.4 Thresholds of Miss Rates. As described in Section 3.5, CachePerf will skip the report if the number of load misses is less than 3% of all load accesses and the number of store misses is less than 1% of all store accesses. The goal is to exclude minor issues. To evaluate the correctness of the two thresholds, we checked the load and store miss rates of all evaluated applications. Overall, for applications with reported issues, as listed in Table 2, their load or store miss rates are higher than the default thresholds. For applications where we do not observe significant issues (not listed in Table 2), the load and store miss rates are both lower than these thresholds. Therefore, the current thresholds of miss rates are helpful to filter out minor issues and highlight significant issues of cache misses.

5 DISCUSSION

This section discusses the compatibility, thresholds, and limitation of CachePerf.

5.1 Compatibility

CachePerf can be easily adapted to different hardware environments, such as cache with different cache line sizes or associativity. Currently, the cache-related parameters (e.g. cache line size, cache associativity) are listed in a configuration file. If users would like to use CachePerf for hardware with a different setting, they only need to change this configuration file.

5.2 Configurable Thresholds

CachePerf introduces some thresholds to control the sampling and the reporting. Such thresholds are confirmed to balance the overhead and effectiveness on the evaluated machine. In a different environment, users may need to change these thresholds. The thresholds used by CachePerf can be easily changed via compilation flags or environmental settings.

5.3 Limitation

CachePerf utilizes the hardware-based sampling techniques to perform the profiling, which has the benefits that do not need to change the programs and imposes little performance overhead. However, the setting of the PMU-based sampling may require some slight changes on different machines with different implementations. Since the PMU-based sampling and the breakpoint-based sampling are generally supported by different hardware architecture, the proposed techniques should be applicable for different hardware.

6 RELATED WORK

We discuss the related work based on the type of cache misses in the following. Although some tools, such as perf [10], oprofile [31], different Pin tools [19, 38], or cachegrind (one tool inside Valgrind [40]), could report the percentage of cache misses in the lines of code, they cannot identify the type of cache misses. Therefore, they are not the focus of this paper. In the following, we only list tools that could identify the type of cache misses.

Detecting Capacity Misses: Tao et al. propose a cache simulator that can identify cache capacity misses using the reuse distance for each memory access [51]. Nikos et al. propose another cache simulation methodology [41]. Both cache simulators could study cache behaviors under various cache configurations, but neither of them can be used as an online profiling tool due to their prohibitive overhead. Delorean [42] improves the simulation efficiency, and identifies cache capacity misses by the number of distinct memory accesses since the last access to the observed cache line. However, it is still a simulation technique that requires the inspection of every memory access, which is slow too. ArrayTool focuses on a special type of capacity misses caused by multiple arrays [35]. It utilizes the PMU-sampling to collect memory samples and determines candidate arrays by the combination of array affinities and array's access patterns.

Detecting Conflict Misses: Cache simulators detect conflict misses by simulating the cache behavior based on the memory trace [29, 57], but they are too slow to be used for online profiling. CCProf proposes to employ Re-Conflict Distance to filter out cache sets with low RCD [46], based on address sampling. However, CCProf may introduce high performance overhead due to the use of a low sampling rate to capture RCD. As shown in Fig. 10, the overhead of CCProf can be as much as 945 times. In contrast, CachePerf imposes significantly less overhead while could identify different types of cache misses.

Detecting Cache Coherency Misses: There exist multiple types of tools that could detect cache coherence issues, mostly focusing on false sharing. Some tools are relying on binary instrumentation [58], compiler-based instrumentation [34], process-based page protection [32], and the PMU-based sampling [6, 8, 20, 25, 33, 39]. Instrumentation-based tools are generally too expensive to be employed in the production environment [34, 58], while Sheriff only supports C/C++ applications using standard synchronizations [32]. In theory, Sheriff cannot be able to support some complicated applications (e.g., MySQL) with ad hoc synchronizations. The approaches with the PMU-based sampling is efficient, but with their own shortcomings: Cheetah utilizes a simplified method to compute the number of cache invalidations [33], instead of relying on the sampled cache misses; Jayasena et al. propose a machine learning approach based on the sampled events [20], Laser utilizes a special type of events (hit-Modified) that may not be available on all hardware [39], while Feather utilizes the combination of the PMU-sampling and watchpoints to identify false sharing [6]; however, all existing tools typically report an absolute number to evaluate the seriousness of false sharing, which may report insignificant issues. They could not detect false sharing caused by external libraries.

Classifying Different Types of Cache misses. Some approaches could classify multiple types of cache misses together. Sanchez et al. propose a data locality analysis tool that can identify compulsory, conflict and capacity misses, but not coherence misses [47]. Its profiling stage incurs reasonable overhead, but it requires a specialized compiler to extract reuse information beforehand and an expensive offline processing stage. These characteristics make this tool inconvenient and inefficient to use. DProf detects datatype-related cache performance issues inside the Linux kernel via the PMU-based sampling and tracing object access histories [43]. DProf employs the definitions of cache misses for its classification, but with the following issues: first, it requires human effort and expertise to summarize data profile, miss classification, working set, and data flow together to identify a particular type of issue, which is not friendly to people without such expertise. Second, it may lose its precision due to its coarse-grained profiling, which is infeasible to find the last write of each miss (and then affect its report). Third, DProf requires the change of the monitored target (e.g., kernel), which may prevent people from using it. Fourth, DProf provides no mechanism of differentiating issues of applications from those of allocators. In contrast, CachePerf overcomes these issues by automatically identifying the type of cache misses, as discussed in Section 3.5. Another difference is that CachePerf requires no change of programs, as it is a library that can be linked to applications. Further, CachePerf could also identify cache misses caused by the allocator that DProf cannot do.

Other Relevant Work: DMon proposes selective profiling that could incrementally increase its monitoring scope (e.g., sampled events) based on the dynamic behavior of execution [24]. In this sense, CachePerf is very similar to DMon. However, CachePerf selectively chooses the instructions to monitor (not hardware events) in order to collect fine-grained information. DMon relies on `perf` to collect sampled events, while CachePerf proposes a new profiler that classifies different types of cache misses based on a set of hardware events and hardware breakpoints. CachePerf could classify the type of cache misses, where DMon only reports the cache miss ratio at different lines of code, inheriting from `perf`, and relies on human expertise to diagnose the issue.

7 CONCLUSION

Cache miss is a well-known performance issue. Although existing tools could report cache miss ratios at different lines of code, significant effort is still mandatory to figure out the type and the origin (e.g., object, allocator) of cache misses in order to reduce cache misses. This paper describes a unified profiling tool—CachePerf—that could correctly identify different types of cache misses while imposing reasonable overhead. This paper further proposes a new method that combines PMU-based coarse-grained sampling and breakpoint-based fine-grained sampling to balance the accuracy and performance. Overall, CachePerf only imposes 14% performance overhead, while identifying multiple known and new cache misses correctly. CachePerf is an indispensable complementary to existing profilers due to its uniqueness.

ACKNOWLEDGMENTS

We thank our Shepherd Sergey Blagodurov and anonymous reviewers for their helpful comments on improving this paper. We also thank Probir Roy, Milind Chabbi for their help on the setup of CCProf and Feather for the comparison, and Xu Liu for the initial discussions on hardware performance counters. This material is based upon work supported by the National Science Foundation under Award CCF-2024253, and the UMass start-up package. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

- [1] Denis Bakhvalov. Advanced profiling topics. pebs and lbr. <https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR>, 2018.
- [2] Bin Bao and Chen Ding. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, page 1–11, USA, 2013. IEEE Computer Society.
- [3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, 2000.
- [4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.
- [5] Bryan R. Buck and Jeffrey K. Hollingsworth. Data centric cache measurement on the Intel Itanium 2 processor. In *SC '04: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, page 58, Washington, DC, USA, 2004. IEEE Computer Society.
- [6] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '18, page 152–167, 2018.
- [7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
- [8] Intel Corporation. Intel vtune profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.ddfte>.
- [9] Intel Corporation. Intel math kernel library. <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html>, 2021.
- [10] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Linux kernel profiling with perf. <https://perf.wiki.kernel.org/index.php/Tutorial>, 2015.
- [11] Guang Gao, Russell Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295. Springer, 1992.
- [12] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2007. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>, 2007.
- [13] Ryutaro Himeno. Himeno benchmark. <https://i.riken.jp/en/supercom/documents/himenobmt/>.
- [14] Intel. Intel xed. <https://intelxed.github.io/>, 2017.
- [15] Intel Corporation. intel@ 64 and IA-32 Architectures Software Developer Manuals. <https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html>.
- [16] Intel 64 and IA-32 Architectures Software Developer's Manual. <https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html>.
- [17] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, and Nicolai Kosche. Memory profiling using hardware counters. In *SC '03: Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.
- [18] R. Iyer. On modeling and analyzing cache hierarchies using casper. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 182–187, 2003.
- [19] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp\$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.
- [20] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. Detection of false sharing using machine learning. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2013.
- [21] Norman P. Jouppi. Reducing compulsory and capacity misses, 1990.
- [22] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 825–836, New York, NY, USA, 2014. ACM.
- [23] Ian Karlin. Lulesh programming model and performance ports overview. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012.
- [24] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 163–181. USENIX Association, July 2021.

- [25] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Huron: Hybrid false sharing detection and repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 453–468, New York, NY, USA, 2019. Association for Computing Machinery.
- [26] Alexey Kopytov and Sunny Bains. Inefficient innodb row stats implementation. <https://bugs.mysql.com/bug.php?id=79454>, 2017.
- [27] Adam J Kunen, Teresa S Bailey, and Peter N Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.
- [28] Lawrence Livermore National Laboratory. Llnl sequoia benchmarks. <https://asc.llnl.gov/sequoia/benchmarks>.
- [29] Alvin R Lebeck and David A Wood. Cache profiling and the spec benchmarks: A case study. *Computer*, 27(10):15–26, 1994.
- [30] David Levinthal. Performance analysis guide for intel core™ i7 processor and intel xeon 5500 processors. <https://software.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf>, 2009.
- [31] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.
- [32] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.
- [33] Tongping Liu and Xu Liu. Cheetah: Detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 1–11, New York, NY, USA, 2016. ACM.
- [34] Tongping Liu, Chen Tian, Hu Ziang, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'14, New York, NY, USA, 2014. ACM.
- [35] X. Liu, K. Sharma, and J. Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 405–415, 2014.
- [36] Xu Liu and John M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2013.
- [37] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 405–416, 2014.
- [38] Brandon Lucia. Multicachesim: A coherent multiprocessor cache simulator.
- [39] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. Laser: Light, accurate sharing detection and repair. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–273, 2016.
- [40] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100. Association for Computing Machinery, 2007.
- [41] Nikos Nikoleris, Erik Hagersten, and Trevor E Carlson. Delorean: Virtualized Directed Profiling for Cache Modeling in Sampled Simulation, 2018.
- [42] Nikos Nikoleris, Erik Hagersten, and Trevor E Carlson. Delorean: Virtualized directed profiling for cache modeling in sampled simulation, 2018.
- [43] Aleksey Pesterev, Nikolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 335–348, 2010.
- [44] L.-N. Pouchet and T. Yuki. Polybench/c 4.1. <https://sourceforge.net/projects/polybench/>, 2016.
- [45] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradschi, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.
- [46] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, and Xu Liu. Lightweight detection of cache conflicts. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 200–213, New York, NY, USA, 2018. Association for Computing Machinery.
- [47] J. Sanchez and A. Gonzalez. Analyzing data locality in numeric applications. *IEEE Micro*, 20(4):58–66, August 2000.
- [48] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC '12, page 28, USA, 2012. USENIX Association.
- [49] J Seward, N Nethercote, and J Fitzhardinge. Cachegrind: a cache-miss profiler, 2004.
- [50] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410, New York, NY, USA, 2011. ACM.

- [51] J. Tao and W. Karl. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 62–es, New York, NY, USA, 2006. Association for Computing Machinery.
- [52] Jie Tao and Wolfgang Karl. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools - valuertools '06*, page 62, Pisa, Italy, 2006. ACM Press.
- [53] Wikipedia Team. Cache performance measurement and metric: Capacity misses. https://en.wikipedia.org/wiki/Cache_performance_measurement_and_metric#Capacity_misses.
- [54] tiny-dnn Team. header only, dependency-free deep learning framework in c++14. <https://github.com/tiny-dnn/tiny-dnn>, 2018.
- [55] Wikipedia. Loop optimization.
- [56] ACTON S T YU YJ. Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, 2002.
- [57] Hongli Zhang and Margaret Martonosi. A mathematical cache miss analysis for pointer data structures. In *SIAM Conference on Parallel Processing for Scientific Computing*. Citeseer, 2001.
- [58] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE' 11*, pages 27–38, New York, NY, USA, 2011. Association for Computing Machinery.